ABSTRACT

A Visually Realistic Simulator for Autonomous eVTOL Aircraft

Seth M. Nielsen
Department of Electrical and Computer Engineering, BYU
Master of Science

Electrically powered vertical takeoff and landing (eVTOL) aircraft could provide a new mode of air transportation of people and cargo that is low-cost, on-demand, and able to reach more areas than is possible with current technology. They have the unique ability to takeoff and land in congested spaces yet efficiently travel long distances which makes them a promising technology for applications such as rapid medical assistance, automated package delivery, or human transportation. This type of aircraft has only recently become a possibility, owing to advancements in battery technology, computing power, and sensor technologies, and thus support for eVTOLs is lacking among high-fidelity graphics simulation software. High-fidelity graphics are important for the goal of fully autonomous eVTOL aircraft in order to accurately simulate vision-based navigation using camera sensors.

In this work, we present *VTOL-AirSim*, an extension of Microsoft AirSim with full integration of a tiltrotor eVTOL aircraft. Built on Unreal Engine, a high-end graphics game engine, it includes photorealistic graphics for simulated camera images and for high-quality presentations. We created the visual components of a fully animated tiltrotor vehicle and a detailed city environment for it to fly in. The tiltrotor can be controlled via motor PWM commands, by overriding its state in the world with the use of an external dynamics simulation, or by using the PX4 Autopilot. We give a tutorial on how to use VTOL-AirSim where we provide examples for each method of control, including scripts for controlling the tiltrotor via a geometric controller and control allocation module developed by others in the BYU MAGICC Lab. We also show how to further develop VTOL-AirSim, and how to add custom aircraft or custom environments so that others may use it in their own research.

Keywords: UAV, eVTOL, simulator, game engine

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Electrically powered vertical takeoff and landing (eVTOL) aircraft could provide a new mode of air transportation of people and cargo that is low-cost, on-demand, and able to reach more areas than is possible with current technology. Winged eVTOLs, or simply eVTOLs, are a class of aerial vehicles that can travel using aerodynamic lift yet take off and land vertically. This means eVTOLs don't require runways or other large infrastructure to operate; they can take off and land virtually anywhere, even in compact or congested spaces. They are much more energy-efficient than most vertically flying aircraft, like helicopters and multirotors, because of the aerodynamic lift produced by their wings during horizontal flight. This gives them the ability to travel longer distances and carry heavier payloads. With these advantages, there is great interest in eVTOLs as a cost-effective technology for rapid, point-to-point air transportation of people and cargo.

Figure 1.1: *VTOL-AirSim*: our tool for simulating eVTOL aircraft in realistic environments.

Advancements in computing power, electronics, digital sensors, and software have enabled great progress towards increasingly automated flight of these aircraft, which is further made possible by their fully electric propulsion. With autonomous flight capabilities, eVTOLs will have greater safety, efficiency, and ease of operation. Key to the research and development of autonomous vehicles is the ability to simulate their control and navigation software; however, because they have only recently become a possibility, the field of eVTOL research is less mature, and simulators with support for eVTOLs are scarce.

Many modern unmanned aerial vehicles (UAVs) that can fly autonomously rely on camera sensors. This is because visual data contains a large amount of information about the environment that can be used to detect and avoid obstacles, navigate through cluttered spaces, and identify and track targets. It is therefore important to simulate vision-based navigation for eVTOLs. Simulating visual data from a camera sensor, however, requires high-fidelity graphics to accurately test the performance of the vision-based navigation software. There are few publicly available simulators which utilize high-quality graphics engines, and this further complicates finding a viable simulator for eVTOLs.

This thesis explains our contribution of creating a capable, high-graphics simulation of winged eVTOL aircraft that others can use. In addition, we show a system that we developed for others to easily create their own unique simulations of eVTOLs. The simulation we developed is an extension of Microsoft AirSim, a popular open-source tool used by many researchers for simulating multirotor aircraft.

1.1 Review of Available Simulation Tools

To the best of our knowledge, there are currently no publicly available simulators for research that provide the ability to simulate eVTOL aircraft with high-quality graphics. Therefore, it was necessary for us to choose from the simulation tools that are available and design a way to modify, extend, or otherwise work around these tools to produce something that they currently do not offer.

There are two well-known products in the world of flight simulation that are renowned for their high-fidelity graphics, massive amount of features, and accurate modeling of the physics of flight: *Microsoft Flight Simulator* [1], and *X-Plane 11* from Laminar Research [2].

While different, these two products fit into similar categories as simulation tools. A brief comparison of the two are given in [3]. They offer creating customized aircraft, but with different implementations. The flight model in Flight Simulator uses a set of tables which are computed by supplied parameters. X-Plane 11 computes flight dynamics directly from the visual model, and it allows for the use of one's own flight model, in which case the aircraft becomes just a visual representation of the externally simulated dynamics. The latter is advantageous in the case of simulating an eVTOL aircraft, in which the dynamic model can greatly differ from that of standard fixed-wing, helicopter, or multirotor aircraft. However, neither software allows for winged eVTOL designs, such as animating tiltable rotors. This is undesirable — not only for the purpose of presenting simulations to others, but also for the engineering analysis and insight that is lost when using a visualization that misrepresents core components. In addition, both simulators are closed-source, proprietary software products that are heavily geared toward simulating the piloting of aircraft rather than tools for researching autonomous navigation software for UAVs. They don't provide easy-to-use, flexible programming interfaces to accommodate typical software written by researchers. Despite this, some researchers have found them useful as UAV simulators, due to their high fidelity graphics and flight dynamics, such as the use of Microsoft Flight Simulator in [4], [5] and the use of X-Plane in [6], [7]. Note that these studies only simulated standard fixed-wing or helicopter aircraft.

A simulator that has arguably become the standard simulator for robotics research is the free and open-source Gazebo project [8]. It is widely used in both robotics and UAV research and thus has a large community. It is highly customizable, even to allow for simulating eVTOLs, as in [9]. However, this comes at the cost of requiring a great deal of manual setup on the part of the researcher, as Gazebo is a rather generic robotics simulator and does not offer built-in features such as eVTOL aircraft. In addition, the graphics quality of simulations made in Gazebo are, by today's standards, of moderate or low fidelity. This makes it less useful for accurately simulating vision-based navigation and control.

Another simulator from Microsoft, *AirSim* [10], is becoming increasingly popular among multirotor UAV researchers. It is free, open-source, and is built on *Unreal Engine* [11], a professional game engine from Epic Games that is known for its very high-quality graph-

ics. It has many built-in features, such as support for the PX4 Autopilot [12] and ROS [13], and it provides simple interfaces in Python and C++. It also offers a number of prebuilt environments of high-quality graphics that can be downloaded and ran on anyone's machine in a number of minutes. AirSim has been used for simulating vision-based navigation in [14], [15], [12]. A core problem with AirSim is that it's completely structured around three separate simulation modes: Multirotor, Car, or Computer Vision (i.e., a camera with no dynamics or vehicle). This means that eVTOL aircraft, or even fixed-wing vehicles, can not be simulated in AirSim. Moreover, while AirSim is open-source and does contain documentation on how to customize it, doing so requires installing and using Unreal Engine, which has high computer hardware requirements and a steep learning curve.

And finally, there is the *ROSflight Holodeck* simulation system. This is a tool developed by the MAGICC Lab that combines two other projects, the ROSflight autopilot created by former students of the MAGICC Lab, and the Holodeck simulator, created by the BYU Perception, Control, and Cognition Lab, into a full software-in-the-loop simulation with a high-quality graphics environment. Like AirSim, Holodeck uses Unreal Engine for its graphics, and thus can be a great tool for computer vision applications. In contrast to AirSim, the project's sole developers are a few graduate students, and it currently lacks many important features, such as support for multiple vehicles.

For our solution, we chose to develop an extension to Microsoft AirSim that adds the capability of simulating eVTOL aircraft, which we gave the name of *VTOL-AirSim*.

1.2 Contributions

This thesis makes contributions in the following areas:

- An extension to AirSim for simulating eVTOL aircraft control and dynamics, named VTOL-AirSim. It includes high-quality graphics for realistic simulated camera images. We also created visual components for the simulation: a fully animated tiltrotor aircraft mesh, and a detailed city environment to fly it in. An introduction to AirSim and tutorials for VTOL-AirSim, which includes scripts we created for interfacing VTOL-AirSim with a trajectory generator, a geometric controller and an eVTOL control

allocation module from the MAGICC Lab's VTOLsim project [16], [17], are given in Chapter 2. A tutorial for interfacing the PX4 Autopilot with VTOL-AirSim is also given in that chapter.

- A guide on all the tools that are needed to extend VTOL-AirSim is given in Chapter 3. We show how to build Unreal Engine from source and install it on Linux, after which we provide a short user guide.

- Tutorials for creating custom aircraft and custom environments for use in VTOL-AirSim are given in Chapters 4 and 5, respectively. We provide full examples of how to obtain prebuilt content from online and then integrate the content using the Unreal Editor to create a customized VTOL-AirSim simulation.

CHAPTER 2. VTOL-AIRSIM USER GUIDE

This chapter will be a detailed guide involving the setup, configuration, and basic uses of Microsoft AirSim and our extension to AirSim, VTOL-AirSim. Sections 2.1–2.3 are written for those who have never used AirSim previously, and serves as an introduction to AirSim. Sections 2.4–2.6 are about VTOL-AirSim and examples of how to use it.

All instructions throughout this text are written for Linux users. The instructions have been tested on Arch Linux and Ubuntu 20.04 — the latest Long Term Support (LTS) release as of this writing — and are written with a focus on Ubuntu users.

If this is your first time using AirSim, regardless of your use case, it is recommended that you follow the setup guide described in this chapter before continuing to the advanced chapters. This will help you to be aware of the various components involved in a complete simulation environment, which will be very beneficial knowledge when debugging a problem or attempting to add functionality to the sim later.

2.1 Description of AirSim

It is important to first understand what exactly AirSim is before setting it up. AirSim should be thought of not as a single, self-contained simulator program, but rather as a collection of simulation tools which you can link together in various ways and interface with at various levels. Because of this, there are multiple ways to use AirSim, and individual AirSim-based simulations can appear to be quite different from one another depending on which components are used. Nevertheless, there are two parts to any complete AirSim simulation: an environment (the server) and one or more clients interacting with the environment. While it is technically possible to have zero clients connected to the environment server, the result would be an environment containing one or more vehicles that never perform any actions,

and this is not very useful. Thus, any complete AirSim simulation has at least one connected client. What follows is a brief description of these two parts and their major components.

Note that for simplicity, the following sections are worded in terms of a single vehicle in the simulation; however, AirSim has full multi-vehicle support, so any mention of a single vehicle can be substituted with multiple vehicles. In addition, AirSim also has a car mode, but it is outside the scope of this work.

2.1.1 Definitions

The following terms can have different meanings depending on the context, and so we have decided on some fixed definitions for this text.

- **mesh** — The collection of vertices, edges, and faces that make up the graphical representation of an object.
- **model** — The dynamic model of the vehicle; or, in other words, the equations of motion, control inputs and motor outputs that are particular to a vehicle or vehicle type.
- **viewport** — The area of the screen in which rendered graphics are displayed to the user.

Environment

The environment is a 3D graphics application which has been compiled for use with AirSim. The environment contains a scene made up of various meshes, which may be static or mobile. The vehicle is an animated mesh representing the state of the aircraft which the user controls. The graphics engine rendering the environment and the vehicle is *Unreal Engine*: a game engine used for producing high-end graphics video games, but which can also be a useful tool for engineering simulations. When the environment is running, Unreal Engine is responsible for the following tasks: render images for the simulated camera sensor from the environment; handle vehicle collisions with objects in the scene; animate any moving parts of the vehicle plus the movement of the vehicle itself; and display rendered images to

the user's viewport. Meanwhile, invisible to the user, AirSim performs a number of other tasks, namely: simulate the physics of the vehicle and the non-camera sensors; process control inputs and simulate motor responses — either through its own autopilot stack for multirotors, SimpleFlight, or by interfacing with an external autopilot such as PX4 — communicate with Unreal Engine about the vehicle's state and the physics of any collisions that occur; and run a remote procedural call (RPC) server, which processes commands sent by the client.

Client

The client is a process initiated by the user which sends commands to the server (the RPC server which runs inside the environment). Any client is initiated independently from the environment. There can be multiple clients running and simultaneously passing data to and from the server. Clients can be created, shut down, or restarted any number of times while the server is running. The environment is initiated first, because it is the server, and any clients are initiated after. This is because a client won't perform any actions until a connection with the server is established, and if it tries to create a connection but can't, it will close with an error.

2.1.2 Outline of AirSim Components

We have just explained the two primary parts of an AirSim simulation: the environment and a client. With that knowledge in hand, we will now outline the major components of AirSim.

- AirLib — The core C++ code of AirSim consisting of many subcomponents, including:
  - physics (dynamics, drag model, thrust and torques, collision handling)
  - non-camera sensor models
  - control inputs and motor responses
  - a small, self-contained multirotor autopilot called SimpleFlight
  - interfaces for the autopilots PX4 and Ardupilot

  - RPC server and client

- MavLinkCom — C++ library that uses MavLink to communicate with PX4 or Ardupilot
- Python client — a Python wrapper around the C++ RPC client found in AirLib
- Compiled environments (binaries) — a selection of prebuilt, downloadable environments
- Unreal Engine plugin, plus a simple example project named Blocks
- Unity plugin — a plugin for the Unity game engine, which will not be covered in this text

Each of these components may or may not be used in a given AirSim simulation, but they are all part of the stack of software that AirSim offers. It is up to the user to decide which of these components to use, interface with, or modify based on the needs of the project at hand.

2.2 Basic Setup: Prebuilt Environment

We will now begin with the most basic setup of AirSim: using a prebuilt environment. This is the least configurable way to use AirSim, but it is the fastest and easiest way to get an AirSim simulation running on your machine. Prebuilt environments allow for some configuration by way of a settings file. For some projects, the settings offered for prebuilt environments may be sufficient.

The first step is to download a prebuilt environment from the AirSim GitHub repository. The repository is located online at `https://github.com/microsoft/AirSim`. This is the repository containing all of the source code behind the various components outlined in the previous section. On the right of the page you will see several sections containing information about the repository. The first section is labeled **About**, and underneath it is a section labeled **Releases**. Click on the header **Releases** to go to the AirSim Releases page. Alternatively, you can go directly to the page at `https://github.com/microsoft/AirSim/releases`.

2.2.1 Explanation of GitHub Releases

If you are not familiar with the concept of releases, it is essentially a snapshot of the software contained in the repository at a certain moment (i.e., at a specific commit). Developers create releases at particular commits that they have deemed important enough in some way to warrant packaging the code in a deliverable format, and to give the commit a special label called a *tag*. A commit can be tagged without creating a release (hence the **Tags** subsection on the Releases page), but a release is always associated with a tag. Releases are most often used to mark a version of the software. Every release on GitHub includes links to download a copy of the source code at that particular commit, but they may also include release notes or additional files for others to use. Often, the managers of a repository will include links to binary files which have been compiled from the source code at that release. This is how the AirSim developers structure their releases.

2.2.2 Download an Environment

On the AirSim Releases page, you will find links to download environments built by the AirSim team. AirSim usually creates two releases for every new version: one containing environments compiled for Windows, and another with environments compiled for Linux; the source code is the same in each. Scroll down the page until you find the most recent release for Linux. The release should be titled in the form of `vX.Y.Z -- Linux`, where X, Y, and Z are numbers making up the version of that release. For example, as of this writing, the most recent release is titled `v1.5.0 -- Linux`; it contains environments built for Linux using version 1.5.0 of the AirSim source code. Be sure that you find the release for Linux and not Windows — although the files may have the same names, the environments listed under the Windows release are *not* compatible with Linux.

Once you have found a Linux release, at the bottom of the release notes you will find all the files that are a part of the release under the expandable section labeled **Assets**. Expand this section by clicking on the label, after which you should see a number of links pertaining to each environment that has been built for Linux. (For brief descriptions of the available environments, see Appendix B.1.) A good environment to start with is the AirSimNH

environment (*NH* meaning *Neighborhood*). Download the ZIP file for this environment and unzip it to a directory of your choice.

Finishing this step will give you the first part of AirSim: the environment. You still need to be able to start a client before you can do anything with AirSim. To start a client, we will use AirSim's Python interface with one of the example Python scripts that AirSim provides.

2.2.3 Create a Python Virtual Environment for AirSim

If you haven't used a Python virtual environment before, it is very easy to create and use one. What it does is create an isolated space (a directory) where non-built-in Python packages will be installed to and searched for when importing or executing code from those packages. All you have to do is create the virtual environment and *activate* it each time you want to use it; you then *deactivate* it when you don't.

It is strongly recommended that you create an entirely fresh Python virtual environment specifically for AirSim. As of this writing, the `airsim` Python package has a dependency on a deprecated and very old package named `msgpack-rpc-python`, which itself has other outdated dependencies that can create problems if you integrate it with your other Python virtual environments, or your user-wide or system-wide Python environments.

You can place the virtual environment wherever you like. It is common to create a folder dedicated to storing virtual environments, often in the user's home directory. The following instructions will create a directory named `.virtualenvs` in the home directory; feel free to replace the path with whichever path and directory name you choose.

First, create a directory to store virtual environments. If you already have a directory that fulfills this purpose, skip this step.

```
mkdir ~/.virtualenvs
cd ~/.virtualenvs
```

Next, create a virtual environment. We will name it `airsim`.

```
python3 -m venv airsim
```

This will create a folder at the current working directory named `airsim` containing a brand new virtual environment. Note that a virtual environment can only contain a single version of Python, and the above command will create a virtual environment according to whichever Python version the `python3` command points to on your system. If you need to some other Python version, replace the number 3 with the specific version you need. For example, if you need to use Python 3.9 but `python3` points to Python 3.8 (verify with the command `python3 --version`), replace `python3` in the above command with `python3.9`. *Warning: be sure you are using Python 3. Do not attempt to use Python 2 with AirSim.*

To activate the new virtual environment, you need to source its `activate` script, which can be done with the following command.

```
source ~/.virtualenvs/airsim/bin/activate
```

This will do a number of things to configure your shell's environment to use the `airsim` directory for importing and installing new Python packages. You have now *activated* your virtual environment for AirSim. Whenever you need to use your AirSim virtual environment in the future, you must either run the above command directly, or, to make it easier to remember, you can create an alias for that command and then use the alias. To stop using the virtual environment, run the command `deactivate` in the terminal.

If you would like a setup that is a bit more convenient and robust for activating and deactivating virtual environments, see Appendix A.2.

Install the AirSim Python Package

The AirSim Python package can be installed through `pip`. Activate your virtual environment, ensure that `pip` is up-to-date, then install the `airsim` package.

```
source ~/.virtualenvs/airsim/bin/activate
pip install --upgrade pip
pip install airsim
```

Get the Python Client Example Scripts

The easiest way to get all the example scripts for using the Python client from AirSim is by cloning the AirSim GitHub repository. Assuming you have added an SSH key for your machine to your GitHub account (see Appendix A.1), you can run the following command in a terminal to clone the repository using SSH.

```
git clone git@github.com:microsoft/AirSim
```

2.3 Basic Use

This section explains the most basic method of using AirSim.

2.3.1 Run the Neighborhood Environment

Open a terminal and navigate to where you placed the `AirSimNH` folder that was extracted from the `AirSimNH.zip` archive, then go to `AirSimNH/LinuxNoEditor`. This `LinuxNoEditor` folder is created by Unreal Engine; it follows a naming convention which indicates that it is a packaged, standalone application built for Linux that doesn't require the Unreal Editor to run.

To start the environment, run the script `AirSimNH.sh` found inside this directory. The script simply sets the application (i.e., the compiled binary) file named `AirSimNH` located under `AirSimNH/Binaries/Linux` to be executable, then executes it. This will start the application, and you will see a new blank window open. A dialog box will appear asking if you would like to use the car simulation. Click **No** to proceed. The window will then begin rendering the Neighborhood environment with a lone quadrotor (Fig. 2.1). You are now running the AirSim server. However, because there are no clients running, the quadrotor will remain stationary until a client connects and sends commands to it.

Figure 2.1: AirSim running in the Neighborhood environment.

2.3.2 Start the Client

It's time to start an AirSim client. We will use one of the example scripts for using the Python client contained in the AirSim repository. Create a new terminal session and activate your AirSim virtual environment. Navigate to the directory where you cloned the AirSim repository and find the Python script `AirSim/PythonClient/multirotor/takeoff.py`, then run the script. The commands for these steps are as follows:

```
source ~/.virtualenvs/airsim/bin/activate
cd <path to AirSim repository>/PythonClient/multirotor
python takeoff.py
```

In the AirSim window, you should see the multirotor take off, reach a certain height, then descend and land back on the pavement. Congratulations, you have just executed a complete AirSim simulation. In the next sections, we will shift our focus to VTOL-AirSim.

2.4 VTOL-AirSim

Our additions to AirSim for simulating eVTOL aircraft is called *VTOL-AirSim*. The previous sections in this chapter apply equally well to VTOL-AirSim, as it is simply AirSim with extended capabilities. From a user standpoint, the biggest difference is the addition of a new *simulation mode*. AirSim's three simulation modes are: `Multirotor`, `Car`, and `ComputerVision`. In each mode, a multirotor, or a car, or a controllable camera will spawn, respectively, and each type has its own set of control schemes. The simulation mode is specified in the settings file (Section 2.5); or, if it isn't specified there, AirSim displays the dialog box when you start an environment asking if you would like to use the car simulation, after which it will start the `Car` or the `Multirotor` mode according to the user's selection. In VTOL-AirSim, we added a fourth simulation mode: `Vtol`. In this mode, a tiltrotor vehicle will spawn, and it has its own set of control schemes, which we cover in the examples given in Section 2.6.

2.4.1 The Tiltrotor Model

It is important to know that the dynamic model for the tiltrotor vehicle is the E-flite Convergence aircraft (Fig. 2.2b). This is because another project by the MAGICC Lab, named VTOLsim, had previously created a dynamics simulation utilizing a model of the Convergence aircraft. Given that we already had the various physical and aerodynamic parameters of the model, we had the actual hardware, and the Convergence was one of the airframes supported by the PX4 Autopilot, we chose to use it for the first iteration of an eVTOL vehicle in AirSim.

Note that the Convergence actually has three rotors — it is a *tri-tiltrotor* design — but the tiltrotor in VTOL-AirSim only has two. We were not able to find a suitable mesh for a tri-tiltrotor aircraft when we created the tiltrotor vehicle. In addition, the Convergence is a small UAV that requires a rear rotor for stability, while a passenger-size tiltrotor would typically not have a rear rotor, so we decided that this was acceptable. However, internal to VTOL-AirSim, the tiltrotor does in fact have a rear rotor; it is simply not a part of the visual representation.

(a) The tiltrotor vehicle in VTOL-AirSim.

(b) The E-flite Convergence aircraft that serves as the dynamic model.

Figure 2.2: Comparison of the tiltrotor mesh in VTOL-AirSim and the E-flite Convergence.

2.4.2 Environments Available in VTOL-AirSim

One component of AirSim that is not shared by VTOL-AirSim is the availability of AirSim's prebuilt environments. Because we do not have access to the Unreal Engine assets that the AirSim developers used to create their environments (other than the Blocks environment), VTOL-AirSim can't be compiled into, say, the Neighborhood environment. There are, however, two environments which we have compiled for VTOL-AirSim: Blocks, and CityBlocks. More information on the CityBlocks environment is in Section 2.6.2, and this is the environment we use in the VTOL-AirSim examples.

2.5 The Settings File

There are many settings available for configuring the AirSim environment. All settings for AirSim are placed in a `settings.json` file at the path `~/Documents/settings.json`. For the official documentation on most of the available settings, go to `https://microsoft.github.io/AirSim/settings`. This section will outline the most important configuration options for VTOL-AirSim pertaining to eVTOL aircraft.

2.5.1 Settings Are Read at Runtime

At startup, AirSim checks for a JSON file at `~/Documents/settings.json`, then, if it exists, it reads the file's contents and attempts to apply all the settings that are recognized.

The settings file is read at the time of initialization; in other words, it is performed at runtime. This means that if a value for any setting is invalid, the application will generally crash during startup. In addition, the file must strictly follow official JSON syntax; if there are any syntax errors, the JSON parsing library that AirSim uses will throw an exception of type `std::invalid_argument`, causing the application to crash without printing any helpful information as to why it crashed. Be aware of this when you edit the settings file, and know that this is a common cause of startup crashes.

A key-value entry (i.e., a setting and a chosen value for that setting, which are written in the format `"key": "value"`) is valid if the AirSim environment was compiled with support for it. For example, the entry `"SimMode": "Vtol"` is valid only if using an environment that was compiled with VTOL-AirSim. If you specify that same entry and then run one of AirSim's prebuilt environments, the application will crash at startup because AirSim does not include `"Vtol"` as a possible simulation mode in its source code.

2.5.2 Settings for VTOL-AirSim

With the addition of the new `Vtol` simulation mode, several new configuration options were added. The affected settings are outlined in the following list.

- `SimMode`
  - Added new possible value: `Vtol` (case insensitive)
  - Explanation: Set the value to `Vtol` to fly an eVTOL aircraft (currently tiltrotor only).
- `Vehicles` ... `VehicleType`
  - Added two new possible values: `VtolSimple`, `PX4Vtol` (case insensitive)
  - Explanation: The default is `VtolSimple`, which runs dummy firmware that simply passes through any PWM commands sent to it. Set this to `PX4Vtol` to run VTOL-AirSim with PX4, as in the example in Section 2.6.6.

2.6 How to Fly eVTOL Aircraft — Examples

In this section we walk you through several complete examples of flying eVTOL aircraft in VTOL-AirSim. In the first example (Section 2.6.4), we cover a full demonstration that includes trajectory generation and following by sending PWM commands to AirSim. In the second example (Section 2.6.5), rather than sending control inputs to AirSim, we show how to override the vehicle's state in AirSim — also referred to as *teleporting* the vehicle — for the case of utilizing an external dynamics simulation. In the third example (Section 2.6.6), we show how to integrate the PX4 Autopilot in which we create and send a mission for the PX4 flight controller to execute and fly the aircraft.

As with the other simulation modes in AirSim, VTOL-AirSim comes with one mesh for the `Vtol` vehicle in the form of a simple tiltrotor. Therefore, this is the aircraft that we use in each of the examples. We will also use the CityBlocks environment in these examples, which is one of two environments that we provide (the other being the Blocks world). If you are interested in changing what is available in VTOL-AirSim see Chapter 3 for information on how to get started.

2.6.1 Required Settings

For the first two examples in this section, *Geometric Controller With PWM Commands* (Section 2.6.4) and *Teleporting* (Section 2.6.5), you should have the following settings in your `settings.json` file. As always, it should located at `~/Documents/AirSim`.

Required Settings for eVTOL Examples

```
{
  "SettingsVersion": 1.2,
  "SimMode": "Vtol",
  "ClockSpeed": 1.0,
  "LogMessagesVisible": false,
  "Vehicles": {
    "uav0": {
      "VehicleType": "VtolSimple"
    }
  }
}
```

You can lower the `ClockSpeed` setting if it appears that your machine is struggling to run the simulation. Also, the setting `"LogMessagesVisible": false` is optional; however, when recording video or screenshots for presentations, it's usually best to disable the log messages that are displayed in the window.

2.6.2 Download the CityBlocks Environment

The CityBlocks environment is an environment that we created for use with VTOL-AirSim that is filled with static buildings, skyscrapers, streets, fields, and water. It was purchased and downloaded from the Unreal Engine Marketplace and afterwards modified to mimic the Blocks environment file structure; hence, we gave it the name *CityBlocks*.

The compiled CityBlocks environment is available on the MAGICC Lab's Box storage at `https://byu.box.com/v/magicc-airsim-cityblocks`. Download the ZIP file and extract the archive to somewhere on your machine. You can then run the CityBlocks application by executing the script `LinuxNoEditor/CityBlocks.sh`, just as you would for the official environments provided by AirSim. In the following examples, the instruction "run the CityBlocks environment" means to execute this script.

2.6.3 Setup for the Python Client for VTOL-AirSim

To get the VTOL-AirSim-modified version of the AirSim Python Client, you will also need to have the BYU-MAGICC AirSim fork cloned onto your computer. In Section 2.2.3, you created a Python virtual environment specifically for AirSim. You may use the same virtual environment or you can create a new virtual environment for VTOL-AirSim. The instructions that we provide for these examples will simply use the same virtual environment named `airsim.`

Inside the AirSim repository lies the directory `PythonClient`. The files contained in this repository are actually what make up the `airsim` Python package. We have made some additions to the package in order to command eVTOL aircraft. Run the following commands to install the modified package:

Install the `airsim` Python Package from the BYU-MAGICC Fork

```
git clone git@github.com:byu-magicc/AirSim
cd AirSim/PythonClient
source ~/.virtualenvs/airsim/bin/activate
pip install -e .
```

The syntax to install a local Python package is `pip install -e <path>`. The `-e` flag means to install it in *editable* mode. This creates a file in your virtual environment that points to the `<path>` that you specify, i.e. `<path to AirSim fork>/PythonClient`. Now when importing the `airsim` package in Python, it will actually use the files contained in the `PythonClient/airsim` folder. If you modify these files, those modifications will be read the next time you import the package.

2.6.4 eVTOL Example — Geometric Controller With PWM Commands

In this and the next example, we will use the MAGICC Lab's VTOLsim project. This project is a repository on the MAGICC GitLab server under `urbanmobility/vtolsim/vtolsim`. You need to clone the repository to somewhere on your machine as well as pull down its submodules. In addition, this example was tested at the commit `fc8da33`, which has been given the tag `airsim_gm_ctl_ex`. We can only guarantee that the instructions given in this example work when using that particular commit. Nevertheless, you may use a newer commit if you choose to do so. You can clone, pull down the submodule `trajectorygenerator`, and check out the tag `airsim_gm_ctl_ex` with the following commands:

Commands to Get the VTOLsim Repository

```
git clone git@magiccvs.byu.edu:urbanmobility/vtolsim/vtolsim.git
cd vtolsim
git checkout airsim_gm_ctl_ex
git submodule update --init --recursive
```

We will use the script `geometric_control_airsim_sim.py`. It is a driver script which creates a VTOL-AirSim Python client for the tiltrotor vehicle (`airsim.VtolClient`), generates a static spline trajectory with states parameterized by time, runs a high-level

Figure 2.3: The tiltrotor vehicle flying in VTOL-AirSim with a geometric controller and control allocation module via PWM commands.

geometric controller for following the trajectory, then uses a control allocation module for producing desired rotor tilts and thrust. The tilt and thrust commands are converted to PWM commands for the individual motors which are then sent to the tiltrotor vehicle in VTOL-AirSim through the command `client.moveByMotorPWMsAsync`.

In a terminal, start running the CityBlocks environment. In another terminal session, navigate to the directory `vtolsim/geometric_control`. Activate your virtual environment for VTOL-AirSim, then run:

Python Script for Geometric Controller With PWM Commands

```
python3 geometric_control_airsim_sim.py
```

You should see the tiltrotor take off and follow a trajectory from one rooftop to another, as seen in Fig. 2.3.

2.6.5 eVTOL Example — Teleporting

Inside the same `geometric_control` directory of VTOLsim (see the previous section) is another script which does many of the same things as in the previous example, but instead of sending PWM commands to the motors, it directly sets the state of the aircraft. The script is named `geometric_control_airsim_teleport.py`. It uses the same geometric controller and control allocation module, though it uses VTOLsim's dynamics simulation rather than VTOL-AirSim's. At each iteration of the main loop, VTOLsim computes the new state of the aircraft and sets the state of the aircraft in VTOL-AirSim through VTOL-AirSim's client command `simSetVtolPose`. This method of using a graphics tool like AirSim for visualization of an external dynamics simulation is often referred to as *teleporting*.

At the time of this writing, there are a few limitations to the teleport method. The first is that the tilt of the rotors cannot be animated. Although `simSetVtolPose` has the parameter `tilt_angles`, the implementation of setting the tilt angles of the rotors in this way is broken, and it produces bad results. We were not able to resolve the issue due to time constraints. For now, you should pass in values of `np.nan` for the tilt angles (meaning the values won't be used), as is done in this script. The rotors will remain tilted at their *nominal angle* — set as 32.5° for this aircraft — throughout the flight. There is another parameter to the function, `spin_props`, which if set to `True` (the default value), should animate the spinning of the propellers, though with some minor visual artifacts.

The second limitation is that it requires using an environment compiled with a special zero gravity feature. With gravity, the aircraft can be successfully teleported to a pose in the world, but it begins falling between client calls of `simSetVtolPose`. Without gravity, it stays at the commanded pose indefinitely. A version of the CityBlocks environment has been compiled specifically for this purpose, which we named `CityBlocks_nogravity`. You need to download and use the latest version of the `CityBlocks_nogravity` environment to run this teleport example.

in a terminal, start the `CityBlocks_nogravity` environment. In another terminal session, navigate to the directory `vtolsim/geometric_control`. Activate your virtual environment for VTOL-AirSim, then run:

Python Script for Teleporting

```
python3 geometric_control_airsim_teleport.py
```

You should see the tiltrotor take off and travel from one rooftop to another, though without animation of the rotor tilts.

2.6.6 eVTOL Example — PX4 Integration

In this example, we will use the PX4 Autopilot to fly the tiltrotor. There are three items you need to run the example: the software *QGroundControl*, the BYU-MAGICC fork of the PX4-Autopilot GitHub repository, and the right settings in your `settings.json`.

Install QGroundControl by going to `https://docs.qgroundcontrol.com/master/en/getting_started/download_and_install.html`, then follow the instructions for Linux. Place the App Image file wherever you like; for example, in `~/.local/bin`. Launch QGroundControl, then select your preferred units (metric is recommended).

Next, clone the `byu-magicc/PX4-Autopilot` repository to somewhere on your machine, then switch to the `v1.10.1-convergence` branch. We created this branch by making some additions to PX4 v1.10.1 in which we added the mixer for the E-flite Convergence airframe to the software-in-the-loop (SITL) configurations. The normal command you would run is `make px4_sitl_default none_tiltrotor` to launch the PX4 firmware in SITL mode using the default tiltrotor mixer. We created the configuration `none_convergence`, and this is the one you should use instead of `none_tiltrotor`.

There are a number of extra settings that are needed for VTOL-AirSim to communicate with the PX4. The following settings should be in your `settings.json` file:

Settings for VTOL-AirSim With PX4

```
{
  "SettingsVersion": 1.2,
  "SimMode": "Vtol",
  "ClockSpeed": 1.00,
  "ViewMode": "SpringArmChase",
  "CameraDirector": {
      "FollowDistance": -20.0
  },
  "OriginGeopoint": {
    "Latitude": 40.246255,
    "Longitude": -111.647835,
    "Altitude": 1418
  },
  "Vehicles": {
    "uav0": {
      "VehicleType": "PX4Vtol",
      "Model": "TriTiltrotor",
      "UseSerial": false,
      "UseTcp": true,
      "TcpPort": 4560,
      "ControlPort": 14580,
      "Parameters": {
        "NAV_RCL_ACT": 0,
        "NAV_DLL_ACT": 0,
        "LPE_LAT": 40.246255,
        "LPE_LON": -111.647835,
        "COM_OBL_ACT": 1
      }
    }
  }
}
```

The values for `Latitude`, `Longitude`, `Altitude`, `LPE_LAT`, and `LPE_LON` were set for the Brigham Young University campus in Provo, Utah, USA. You may change these values to whichever location you are interested in simulating.

The `ViewMode` and `CameraDirector` settings are optional; however, we recommend using the `SpringArmChase` mode in this example as it produces smoother visuals. You may find it useful to set this view mode in the other examples as well. Be aware that, in general, if you use `SpringArmChase`, there is a bug in AirSim where the `CameraDirector` is too

close because of the 0.25 scale we set for the tiltrotor mesh to reduce its size. This makes it necessary to have a large value for `FollowDistance`.

With the above settings, start running the CityBlocks environment. In another terminal, run the following command from within the directory containing the PX4-Autopilot repository:

```
make px4_sitl_default none_convergence
```

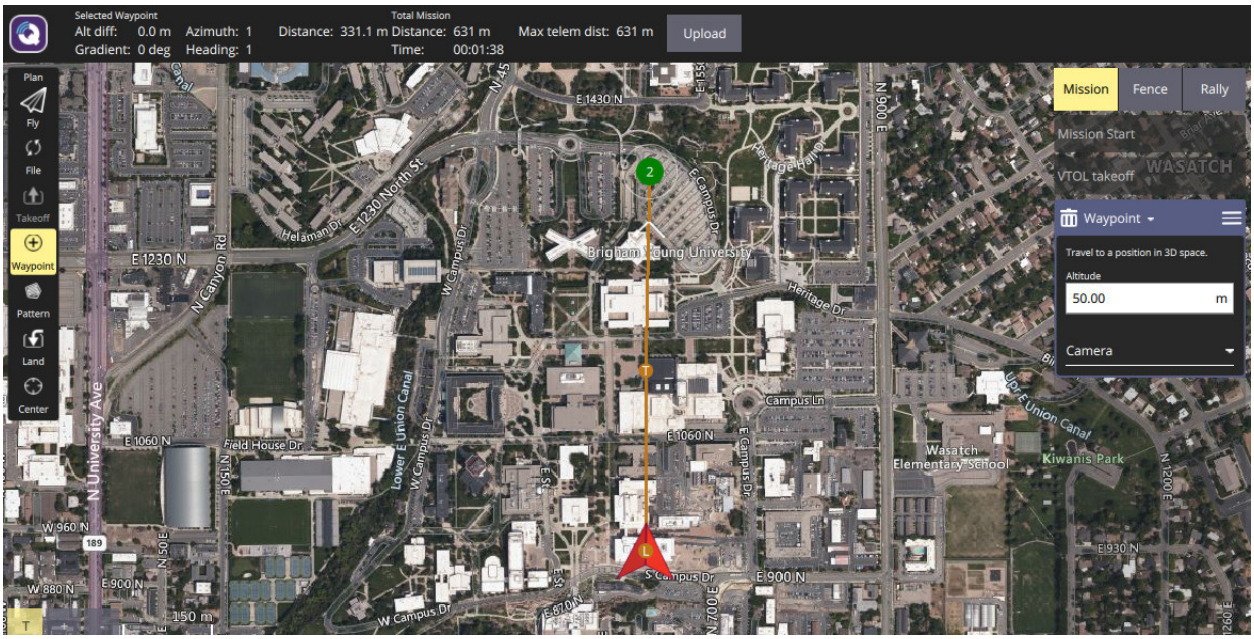

Figure 2.4: QGroundControl screen after planning and uploading mission.

Next, launch QGroundControl. It should show the map centered around the detected GPS coordinate of the VTOL-AirSim vehicle (which are the values set for latitude and longitude in your settings). In the top-left of the map area, click **Plan**. Select **File** (if not already highlighted), then click **Blank**. Next, select **Takeoff**, and a green circle labeled **T** will appear (you may need to zoom out a bit to see it). You can click and drag the **T** to change the transition direction of the vehicle after takeoff, but we'll leave it there for this example. Click **Done** inside the **VTOL takeoff** box. Select **Waypoint**, then click

somewhere to the north of the **T** circle. Finally, click **Upload Required** to upload the mission. Your QGroundControl screen should look like Fig. 2.4.

Now click **Fly**. There should be a slider button on the bottom that says *Slide to confirm*. Slide the button to the right to send the mission to the tiltrotor in VTOL-AirSim. The tiltrotor should begin executing the mission (ignore any warning or error dialogs that pop up). It will first try to reach the target takeoff altitude, 50 m by default, then it will transition into fixed-wing mode in the direction that you set using the **T** circle. After that, it will head to the waypoint that you set, then after arriving, it will loiter about this waypoint — i.e., orbit the waypoint indefinitely. It may actually hit a building while executing the mission; to avoid this, you can raise the altitude of the waypoints when planning the mission, or choose different waypoints.

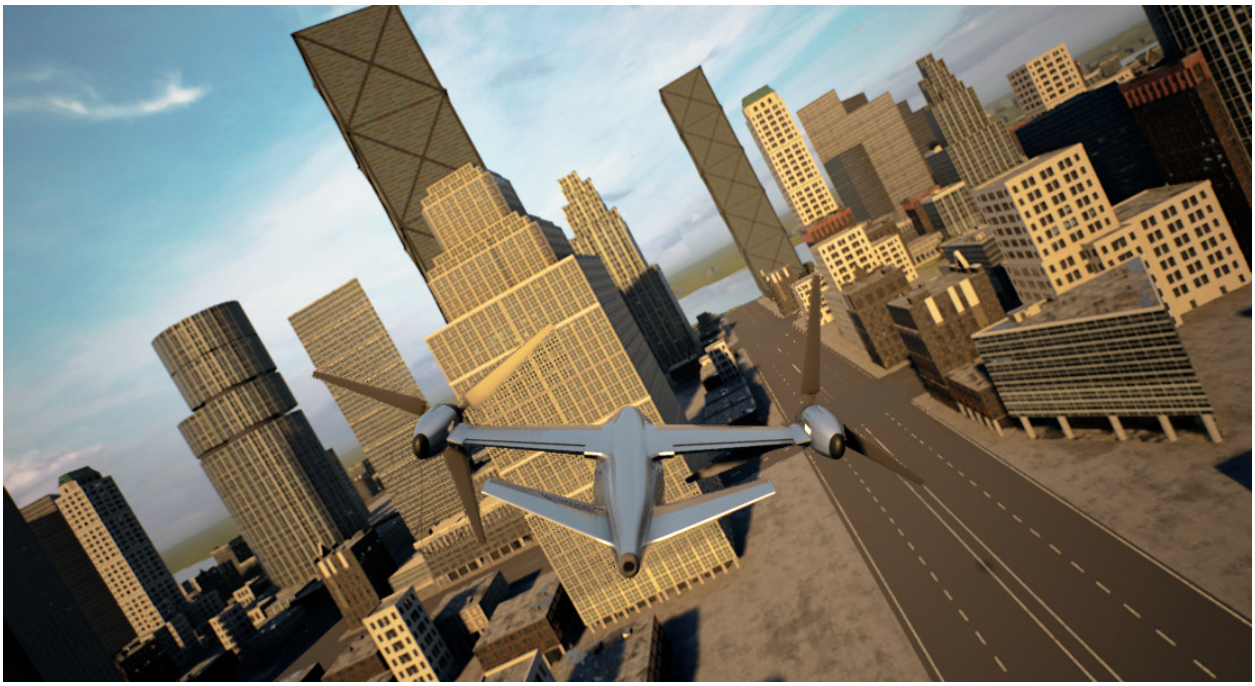

Figure 2.5: The tiltrotor vehicle flying in VTOL-AirSim with the PX4 VTOL controller.

CHAPTER 3. EXTENDING VTOL-AIRSIM: FIRST STEPS

This chapter is structured as a guide to developing eVTOL simulations with VTOL-AirSim and Unreal Engine 4 (UE4). The focus will be on extending the capabilities of VTOL-AirSim and understanding more of the VTOL-AirSim framework.

3.1 Introduction

Extending AirSim — in other words, modifying or adding anything to AirSim — involves conducting an orchestra of many diverse tools and components that all must play together in harmony to be successful. There are a number of basic components which are required in all cases, and a few components that may or may not be required depending on the needs of the project.

At some point, you will need to use *Unreal Engine* — more specifically, the *Unreal Editor* — for testing your work. In addition, you will need to integrate your work with the existing framework of VTOL-AirSim. Your project needs may also require extending the source code. Editing the source code requires a text editor: while you may use any text editor you like, we recommend Microsoft's *VS Code*, for which we provide instructions for setup and use. If your project requires customizing the graphical representation of a vehicle or the environment, then you may also need to use a specialized tool for manipulating 3D computer graphics, for which we recommend the free software *Blender*.

The setup process for this orchestra of tools is significantly more involved than the basic setup explained in Section 2.2. In this chapter, we assume that you have read Chapter 2 (or that you understand the concepts from the chapter), and have completed the basic setup in Section 2.2. We begin with instructions for setting up the two core pieces for development: Unreal Engine (Section 3.2) and VTOL-AirSim (Section 3.3).

3.2 Unreal Engine Setup

Unreal Engine is a game engine developed by Epic Games, Inc. Its source code is written in C++ and is an open-source project on GitHub. It is known for its high-end graphics capabilities, and while it is typically used to produce video games, it has recently found use in other industries such as film, architecture, and engineering.

For our purposes, it has several advantages over a number of other game creation tools, namely that it is free for non-commercial use, has Linux support, is written in C++ and is open-source, and it has an online marketplace with a wide selection of content available to download. It also comes with a few disadvantages, including hefty system requirements, such as a moderately powerful CPU plus dedicated GPU, at least 8 GB of RAM, and at least 100 GB of available disk space on Linux. It can take well over two hours to build Unreal Engine and the Editor from source (it is required to do so on Linux) and a large amount of time to package an Unreal Editor project into a standalone application. Opening the C++ code of an Unreal Editor project with an IDE or a text editor with code analysis can be very demanding on your system due to the immense size of the code base. Furthermore, the learning curve to use the Unreal Editor is steep, and grasping the C++ API is an equally massive undertaking.

That being said, the most important reason we are using Unreal Engine is that it is the central graphics engine used by AirSim, and it was also chosen by the BYU Perception, Control and Cognition Lab for the Holodeck simulator, which we have experience with. AirSim has recently added support for the Unity game engine; however, at the time of this writing, support for it is still experimental.

3.2.1 Requirements

To build Unreal Engine from source, two things are required: at least 100 GB of available disk space to clone and build the engine, and your GitHub account must be a member of the `EpicGames` GitHub organization. We will explain here how to meet the latter requirement.

Unreal Engine is open-source and its source code is contained in the GitHub repository `https://github.com/EpicGames/UnrealEngine`. However, the repository is private, and to gain access to it you must become a member of the `EpicGames` GitHub organization. The instructions for becoming a member are as follows:

1. Create a GitHub account if you don't already have one.
2. Set up SSH access to GitHub (for instructions, see Appendix A.1)
3. Create an Epic Games account at `unrealengine.com`.
4. Follow the instructions listed at `https://unrealengine.com/en-US/ue4-on-github` for linking your GitHub and Epic Games accounts.
    - When asked, accept the End User License Agreement for **Creators**.
5. Join the `EpicGames` GitHub organization via the email invite. You should now have access to the Unreal Engine GitHub repository.
6. Verify by going to `https://github.com/EpicGames/UnrealEngine`, and you should be able to see the repository.

3.2.2 Build the Engine and Editor

Now clone the Unreal Engine repository into a folder of your choice and build it by running the following commands. As of this writing, Unreal Engine 4.25 is the version supported by AirSim, which corresponds to the `4.25` branch of the Unreal Engine repository. **Note**: Unreal Engine is very large; cloning the repository can take between 5–30 minutes, and the build process commonly takes about two hours.

Clone and Build Unreal Engine

```
git clone -b4.25 git@github.com:EpicGames/UnrealEngine.git
cd UnrealEngine
./Setup.sh
./GenerateProjectFiles.sh
make
```

If the build process finishes without any errors, then the engine and the Unreal Editor have successfully been built and are ready to be used. After building, the executable for the editor can be found within the repository at `Engine/Binaries/Linux/UE4Editor`. To start the editor, simply execute that file in a terminal.

3.3 VTOL-AirSim Setup

The VTOL-AirSim project is a composite project that is structured, at the highest level, as an Unreal Engine *Plugin*: a self-contained collection of code, asset files, and data that can be added to (or removed from) any Unreal Engine project. It is accessible as a GitHub repository at `https://github.com/byu-magicc/vtol-AirSim`. The VTOL-AirSim repository began as a copy of the AirSim Unreal Engine Plugin, found inside the official AirSim repository in the folder `Unreal/Plugins/AirSim`, plus the AirLib code, found in the top-level folder `AirLib`. From that starting point, many additions were made, along with a handful of modifications to the original code. Our work eventually resulted in the creation of a fully functional new vehicle type in AirSim for eVTOL aircraft.

Two components of the VTOL-AirSim project exist outside of the VTOL-AirSim repository: a number of large Unreal Engine asset files stored on the MAGICC Lab's Box account, and a fork of the AirSim repository found at `https://github.com/byu-magicc/AirSim`. To learn more about the MAGICC Lab's fork of AirSim, see `MAINTENANCE.md` in the VTOL-AirSim repository.

In this text, we normally use the term *VTOL-AirSim* to refer to the entire VTOL-AirSim project, as opposed to the repository itself. In the latter case, we refer to it as the *VTOL-AirSim repository* if greater clarity is needed.

The full setup instructions can be found in the `README.md` document of the VTOL-AirSim repository, which you can view by going to the repository's main webpage. The instructions stated there are rather detailed, so we will not repeat them in this text. Upon completing this setup, you will have the Unreal Editor running on your machine with the Blocks example project opened. The next section will detail what you need to know for working with VTOL-AirSim in the Unreal Editor.

3.4 VTOL-AirSim in Unreal Editor

This section will serve as a concise guide on the most important concepts for using the Unreal Editor with VTOL-AirSim. The Unreal Editor is a very sophisticated software tool which is much too broad to be covered in any amount of detail in this text. Therefore, we will only cover that which is required for you to be able to do basic work with VTOL-AirSim in Unreal Editor.[1] In this section, we will briefly explain the key parts of the editor interface, how to run simulations in the editor, and how to package projects into standalone applications. More advanced use of the editor for customization of aircraft and environments are covered in Chapters 4 and 5, respectively.

3.4.1 Intro to Unreal Editor

The Unreal Editor is the main graphical user interface (GUI) tool for developing video games (or engineering simulations) that use Unreal Engine. For our goals of simulating eVTOL aircraft in photorealistic environments, there are two main purposes for which the Unreal Editor is required: to *modify* the graphical aspects of the simulation, and to *test* modifications — both modifications to the graphics as well as to the underlying C++ code. This means that if your project requires any customization beyond what is offered by VTOL-AirSim, then you will need to use Unreal Editor to accomplish that.

By now, you should have built and ran the Unreal Editor on your machine. When you first open the Blocks project in the editor, you will need to wait for some shaders of the Blocks environment to be compiled. Once that has finished, you will see a screen like that shown in Fig. 3.1. Note that the interface is divided into six different *panels*. In the top-left corner of each panel is a tab element which contains the name of that panel.

What follows is a quick description of each of these panels. But first, know that in UE4, a *level* is the scene (or the environment, in AirSim terms) which contains every object that you see or interact with. A game is often made up of various levels; however, in our

[1] For more information on using Unreal Editor, see the UE4 documentation at `https://docs.unrealengine.com`. There are also many free resources online such as web articles and YouTube videos that cover these topics.

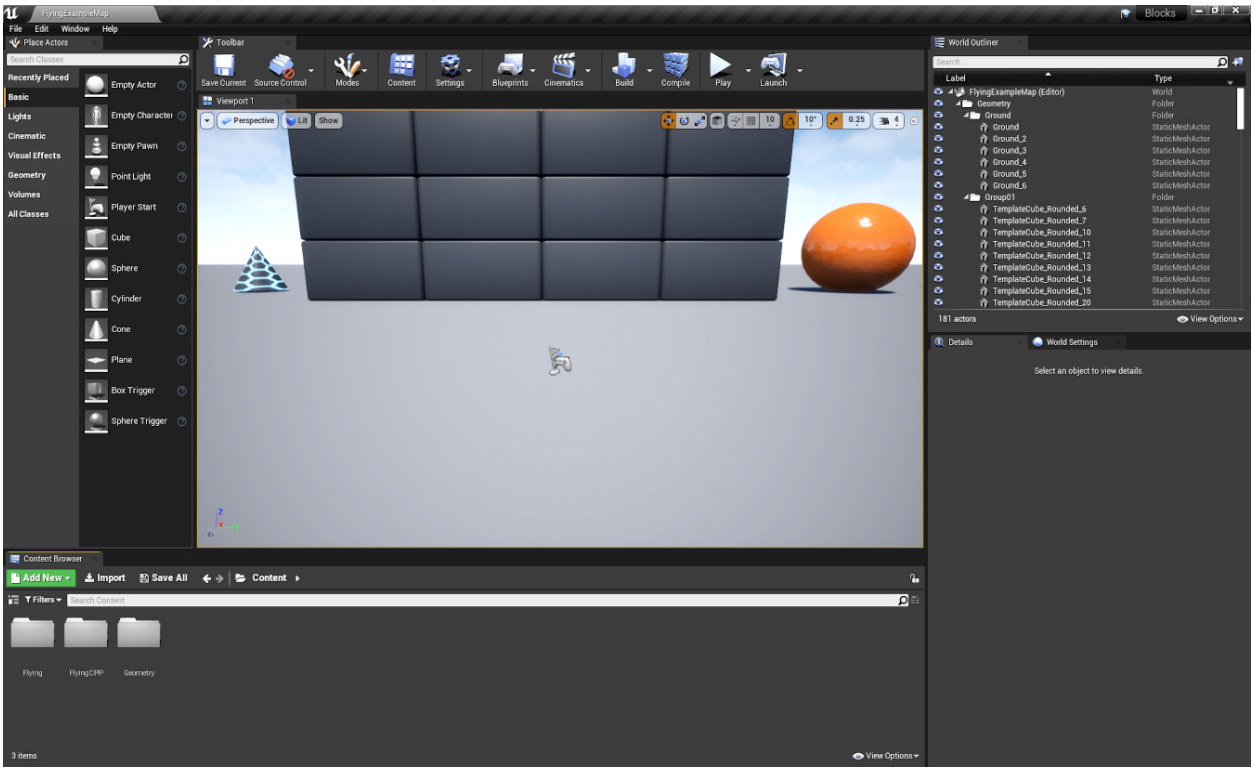

Figure 3.1: The Unreal Editor with the Blocks project opened.

AirSim simulations, we will only have one level. When dealing with Unreal Editor, we will use the proper UE4 term of *level* rather than *environment*.

- **Level Viewport** — The center and largest panel. Here you can see and navigate around the level that the vehicle will interact with during the simulation. Because the simulation is not running, there are no vehicles visible yet. A vehicle will only spawn in the world once a simulation has started. When you press the **Play** button to begin a simulation, the **Level Viewport** will display the simulation just as it would appear in the window of a compiled environment.

- **Toolbar** — Directly above the **Level Viewport**, this is where you will find the **Play** button that allows you to run a full simulation for testing (also referred to as *Play In Editor* or *PIE*). The other tool buttons may be ignored.

- **World Outliner** — located in the upper-right of the window. This panel lists all of the *Actors* that are in the scene. Actors in UE4 are any object that can be placed into

a level, such as cameras, light sources, or static meshes. Currently, it only contains the Actors that are always present, but once you enter Play mode, it will additionally show dynamically spawned Actors such as one or more vehicles and the Camera Actors associated with them. Outside of Play mode, the most important Actor to be aware of is the **PlayerStart** Actor: its *Location* and *Rotation* (the terms in UE4 used for position and orientation, respectively) in the level are used by AirSim as the position and orientation for its internally-computed global coordinate frame. This means you can move **PlayerStart** to change where vehicles spawn in AirSim as well as what is set as the global (0, 0, 0) position and orientation.

- **Details** — located in the lower-right of the window. When you click on an Actor in either the **Level Viewport** or the **World Outliner**, its configurable attributes will be found here. The key item to note in this panel is that here you can modify the *Transform* of an Actor, which includes its Location, Rotation, and Scale attributes.

- **Place Actors** — located at the left of the window. This panel contains a number of Actors provided by UE4 which you may place in the level.

- **Content Browser** — located at the bottom of the window. Here you will find all the *Assets* (or *content*) that are part of the currently opened Unreal Project. The terms *Assets* and *content* as well as the Content Browser are covered in the next section, Section 3.4.2.

3.4.2 Content of VTOL-AirSim

This section will cover the UE4 content that is part of VTOL-AirSim. In UE4, the term *content* refers to the objects or functionalities that make up a game (i.e., simulation) that is not C++ code. Examples of content types include Textures, Materials, Static Meshes, or Blueprints. An *Asset* is a binary file that stores content. The **Content Browser** is like a special file manager that shows you all the assets that are part of the project.

By default, the editor will have open the `Content` folder of the project. In our case, this is the folder `Blocks/Content`. To more easily view the directory structure, click on

the small icon to the left of **Filters** to reveal the **Sources panel**. You will find that the files and folders located in `Blocks/Content` and in the Content Browser mostly match, with some files and folders hidden from view. The assets that we are actually interested in are not in `Blocks/Content`, however, but rather in `Blocks/Plugins/vtol-AirSim/Content`. To instead view these files, click on **View Options** at the bottom-right of the **Content Browser**, and check the box for **Show Plugin Content**. The **Sources panel** should now show a new folder named `AirSim Content`. This folder is so named because the VTOL-AirSim Plugin actually retains its original title of `AirSim` within Unreal Engine; but it is in fact the folder `Blocks/Plugins/vtol-AirSim/Content`. See Appendix B.3 for more information on path names in UE4.

There are 3 directories inside `AirSim Content` that we wish to highlight:

1. `Blueprints` — Here you will find the *Blueprints* of several very important Actors in AirSim; in particular, the Actor for the multirotor vehicle, `BP_FlyingPawn`.

2. `Models/QuadRotor1` — This folder contains the Static Meshes, Materials, and Textures that compose the multirotor vehicle, which are specified in `BP_FlyingPawn`.

3. `VTOL` — All assets that are unique to VTOL-AirSim are contained here. Currently there is only one subdirectory, `Tiltrotor`, however any custom eVTOL aircraft that may be added will go here. Inside the `Tiltrotor` folder is all of the assets that make up the tiltrotor vehicle, including the equivalent `BP_TiltrotorPawn` Blueprint and the Static Meshes and Materials that it is composed of, similar to the multirotor vehicle.

Another asset, the `AirSimAssets` Level, is important specifically for packaging a project, as is explained in Section 3.4.4.

3.4.3 Play In Editor

The most useful feature of the editor for our purposes is **Play In Editor** (**PIE**), accessed via the **Play** button. This is how you will test changes you've made to VTOL-AirSim, whether they are graphical changes made in the editor, or changes made to the

source code (though changes to the code will require closing the editor and rebuilding the project before they will be active). Play In Editor allows you to preview a VTOL-AirSim simulation just as you would see in a standalone application, but displayed within the **Level Viewport**. It also has a few extra features, such as **Pause** to pause the simulation, and **Eject** to become a third-person observer where you can click on objects in the scene, such as the aircraft in the simulation, and see real-time attributes about it in the **Details** panel. For example, you could select the tiltrotor vehicle's engines to see their exact angle values while the simulation is running, and see the values change in real time. Keep in mind that you won't have the same performance using Play In Editor that you will when running a standalone application, but you should still have decent performance.

3.4.4 Package a Project

When you are satisfied with the changes you've made to VTOL-AirSim or your Unreal Project, you can package your project into a standalone application. This is how the compiled environments from AirSim and VTOL-AirSim are made. This allows you to get more performance out of your simulation as well as share your simulation environment with others.

There is an interesting quirk about the AirSim Plugin in Unreal Engine that is very important to know if you are going to package your project. Notice how in the **World Outliner** panel there are no tiltrotor, multirotor, or car vehicles in the scene. This is because in AirSim the vehicles that are part of the simulation are spawned at runtime; they are not permanent objects in a level. However, in Unreal Engine, only the Actors which appear in at least one level are packaged into a game. Therefore, the tiltrotor, multirotor, and car vehicles of AirSim must be present in at least one level at the time of packaging. If you specify a simulation mode in your settings file but that specific vehicle was not a part of any level, then the application will crash with an error that says `Couldn't find file for package` proceeded by the path to the vehicle's Blueprint, as specified in the source code.

AirSim's solution to this is the `AirSimAssets` Level, found under `AirSim Content`. Double-click this level to open it, and you will see in the **World Outliner** these three vehicles, plus the Actor for `ComputerVision` mode. The **Level Viewport**, however, will

be completely empty and black; this is because the level contains no lighting, as it doesn't need any. It is simply a dummy level to work with Unreal Engine's packaging system.

In order to package the tiltrotor vehicle into compiled environments, we had to add the tiltrotor's Blueprint, `BP_TiltrotorPawn`, to this level. If you create a custom aircraft, you will also need to add it to this level. You can click and drag the Blueprint of your custom vehicle anywhere onto the **Level Viewport** when this level is loaded (the location does not matter), and that is enough to comply with this UE4 requirement. Go back to the default Blocks level by navigating to `Content/FlyingCPP/Maps` and double-clicking on `FlyingExampleMap`.

Packaging a project is a simple, yet lengthy process. All you need to do is go to **File** > **Package Project** > **Linux** > **Linux**, choose a suitable directory to store your packaged environment, and then click **Open** and it will begin the process of packaging the project. The process can take over 30 minutes, even for the simple Blocks project on moderate computer hardware, so be prepared for that. Packaging is quicker if you choose the same directory the next time you package the project as it will only do an iterative build rather than build the complete project.

When the packaging process has completed, navigate inside the `LinuxNoEditor` directory that was created, and run the `.sh` file for your project just as you would with any compiled AirSim environment.

3.5 Development Using VS Code

VS Code is the text editor that we recommend for editing the VTOL-AirSim source code. Due to the massive size of the UE4 C++ libraries, it can be very demanding on your machine to edit using the code analysis features of VS Code; however, the cost is usually outweighed by the many benefits it brings for navigating and analyzing the C++ code.

The first thing you should do is install the C/C++ extension for VS Code. Next, rather than open specific *files* to edit, you should open *directories* for editing. The easiest way to do this is to run in a terminal `code <path>`, where `<path>` is the path to the directory you want to open for editing files in VS Code. For working with the VTOL-AirSim code, you can open your Unreal Engine Project that contains the VTOL-AirSim Plugin and

that you will use for testing, or you can open the `Plugins/vtol-AirSim` directory, or lastly you can open just the `Source/AirLib` directory of VTOL-AirSim if you won't need to see any of the Unreal Engine code.

You need a special `c_cpp_properties.json` file present that tells VS Code what directories to include when searching for C++ header or source files. Without this file, the C/C++ extension won't be able to do code analysis for any files that are outside the directory of the file currently being edited. There is a vast number of directories that make up the UE4 libraries, so you shouldn't create this file yourself. There is a script in the Unreal Engine repository that will do this for you named `GenerateProjectFiles.sh`. The script is invoked with the syntax:

```
GenerateProjectFiles.sh <path to Blocks>/Blocks.uproject -game -engine
```

If you are using a project other than Blocks, replace `Blocks` with the name of your project. This will create a number of files into your project's top-level directory for several different IDEs and text editors, but the files for VS Code are placed in a hidden directory named `.vscode`. One of those files is `c_cpp_properties.json`. It contains a great number of include directories to allow the C/C++ extension to do its job correctly. If you open your Unreal Project directory in VS Code, then you should be ready to start editing the code. However, since you won't need to edit any of the files outside of the VTOL-AirSim plugin, you can also copy the `.vscode` directory to the `Plugins/vtol-AirSim` directory, and then open just the `vtol-AirSim` directory to reduce the number of files for VS Code to search through.

3.6 Build

You need to build all the VTOL-AirSim code in order to see your changes reflected in Unreal Editor. There is a guide document with build instructions located in the VTOL-AirSim GitHub repo at `https://github.com/byu-magicc/vtol-AirSim/blob/main/DEVELOPMENT.md`. The guide will teach you all you need to know about building the code.

Once you have completed all these setup steps, you are ready to begin developing and customizing VTOL-AirSim. In the next chapters, we show how you can create custom aircraft and environments for your own simulations.

CHAPTER 4. CUSTOM AIRCRAFT

It is quite possible that the vehicles, vehicle functionalities, or the environments that are provided by VTOL-AirSim are not sufficient for the needs of your project. In this case, there exists the option of customizing VTOL-AirSim beyond what it currently offers. This chapter and the next will teach you how to customize VTOL-AirSim. This chapter demonstrates how to create custom eVTOL aircraft, and the next chapter, Chapter 5, is dedicated to creating custom environments.

Because this topic can be quite broad, we have chosen to structure these two chapters on customization around guiding you through specific examples. Following our examples will allow you to see the entire process from start to finish. We begin with a discussion on a few general principles about meshes, after which we focus for the rest of the chapter on a specific example of taking a new eVTOL aircraft mesh all the way to flying it in VTOL-AirSim.

For dealing with meshes on Linux, we recommend the free 3D modeling software tool *Blender*. See Appendix B.2 for more information about Blender and how to get it. You will need to install Blender in order to make some small adjustments to the example mesh before it can be properly used in Unreal Engine.

4.1 Meshes

As explained in Section 2.1.1, a *mesh* is the collection of vertices, edges, and faces that make up the graphical representation of an object. Only one mesh is provided for each type of vehicle in VTOL-AirSim. For the `Vtol` vehicle, we added a simple mesh of a tiltrotor aircraft (Fig. 4.1). The UE4 assets for the aircraft are contained in the VTOL-AirSim repository.

Before we teach how to create a custom mesh, we wish to reiterate that the mesh of a vehicle is separate from the aircraft's actual dynamic model. It is simply a visual representation of the aircraft, which may or may not be an accurate representation of the

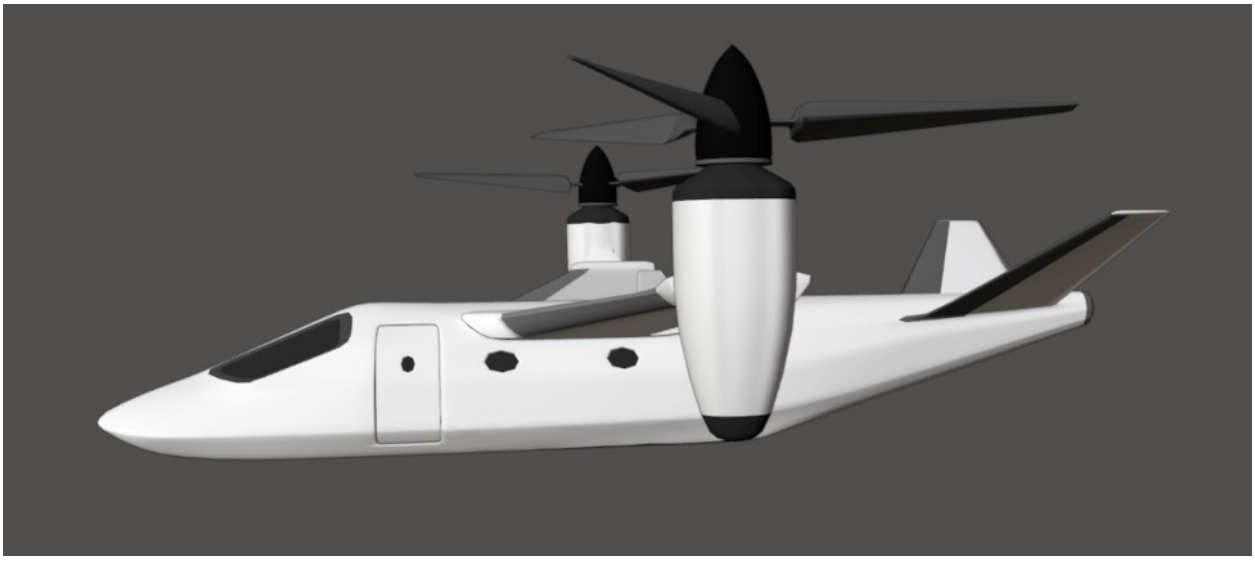

Figure 4.1: The tiltrotor mesh that is provided for the `Vtol` simulation mode in VTOL-AirSim.

vehicle's dynamics, current kinematic state, control outputs, etc. It is up to the developer to decide how the underlying model should affect the mesh, and to do the necessary work for accomplishing that. Customizing the mesh of a vehicle is a rather complicated and laborious task, even for simple cases. Therefore, you should first consider whether the needs of your project do indeed require a custom aircraft mesh or whether changing the model of the aircraft — i.e., parameters such as mass, aerodynamic coefficients, the placement of motors, etc. — would be sufficient. If you only need to change the model, see the VTOL-AirSim GitHub repo for instructions on how to do that.

4.1.1 Obtain a Mesh from the Internet

Modifying the geometry of a mesh is not a straightforward process. We therefore recommend seeking online for a mesh that fits your project well and requires the least amount of modifications as possible. There is a large and ever-growing number of 3D meshes available on the internet which have been created and shared by other people. In the vast majority of cases, obtaining an off-the-shelf mesh that works for your use case will save you a considerable amount of time and effort.

One place to look for downloadable meshes is the official Unreal Engine Marketplace. With this option, you will receive Unreal asset files can be placed directly into an Unreal Project. However, because downloading a mesh from the Unreal Marketplace into an Unreal Project requires using the Epic Games Launcher — an app only available for Windows or MacOS — we recommend looking at third-party 3D model sharing websites first (see Chapter 5 for instructions on obtaining assets from the UE Marketplace). A few examples of websites for sharing 3D meshes include `turbosquid.com`, `cgtrader.com`, and `sketchfab.com`. These and other similar websites contain large databases of meshes made by other people that you can download. You can explore their selections by entering search terms on their website or by viewing categories such as *Aircraft* or *Vehicles*.

Many of the meshes hosted on these websites are not free; nevertheless, the cost is modest compared to the time required to create one from scratch, so do not let the price steer you away from a mesh if it fits well with the requirements of your project.

As you search for an appropriate mesh, there are two things you will want to look out for: the number of polygons (also called *triangles*) and the file types offered. A very high number of polygons in the mesh could cause performance issues, which might manifest itself as slowing down interactions with the Unreal Editor, or possibly slowing performance in the simulation itself. What constitutes a "very high number" is entirely dependent on the computer hardware in question, whether it be your machine, or the machines of others; for instance, consider those who will be working with the mesh, or those who might run simulations containing the mesh. A good rule of thumb could be to look for meshes that have less than 200,000 polygons; though that is by no means a strict limit. For a point of reference, we have tested a mesh that consisted of 440,000 polygons in a VTOL-AirSim simulation that was handled well by some machines, but on other machines it resulted in performance lags and long loading times.

Lastly, try to find a mesh that is offered in `.blend` format for use with the 3D modeling software Blender. This is because nearly all meshes will require some adjustments before they can be properly used in Unreal Engine. If the mesh you find is not offered as Blender project files, you can also download the mesh in a file format that can be imported into Blender, but the import process is not guaranteed to produce a clean result.

For a mesh to be imported into UE4, it must be in FBX format. Thus, there is a possibility that you could be successful in importing a mesh directly into UE4 as an FBX file. However, in our experience, this has resulted in problems such as meshes oriented in the wrong direction, or that the axis of rotation is oriented or positioned incorrectly when animating parts of the mesh — for example, when animating the tilt of the rotors. We therefore recommend that you open the mesh in Blender first, then export the mesh to an FBX file.

4.1.2 Creating a Mesh With CAD Software

It is also possible to create a mesh from the ground up using CAD software. If you do choose to go this route, you will still need to import the mesh into Blender in order to export it in FBX format. One way to accomplish this is to export the mesh from the CAD program as a `.wrl` file (also known as VRML, or Virtual Reality Modeling Language file format), then import the resulting `.wrl` file into Blender, then export from Blender to FBX format. For an example of a mesh that was created in SolidWorks and was successfully brought into UE4 using this process, see the Maker aircraft files on the MAGICC Lab's Box storage at `https://byu.box.com/v/magicc-maker`.

4.2 Edit a Mesh in Blender

We will now walk through an example of how to take a mesh obtained from the internet and get it to fly in VTOL-AirSim. For this example, we downloaded a free mesh of a tiltrotor aircraft from `turbosquid.com` and placed it in the MAGICC Lab Box storage. It is available for download at `https://byu.box.com/v/magicc-airsim-example-mesh`. Download this ZIP file and extract it to somewhere on your machine. Note that beyond this point you will need to have Blender installed.

After extracting the ZIP file, navigate to inside of the `convertiplane` folder. The contents that you see are exactly as they appeared when the mesh was first downloaded, and the files are unmodified from the originals. The actual mesh is contained in the `convertiplane.blend` file. Create a copy of this file and name it `aircraft.blend`.

Figure 4.2: The example VTOL mesh open in Blender after removing unneeded objects.

Assuming that you have already installed Blender on your machine, start Blender and open `aircraft.blend`.

At the top of the window, you will notice a number of tabs with names such as **3D View Full**, **Animation**, **Compositing**, **Default**, and others. These are called *workspaces*. Let's create a new workspace of type *Layout*. Click the plus icon to create a new workspace and select **General** > **Layout**.

4.2.1 Delete Unnecessary Objects

Let's delete objects from this mesh that are not needed. In the top-right is the **Outliner** panel that lists all the objects in the scene and their hierarchies. Find the `convertoplan` (*sic*) object and click the arrow to the left of its name to expand it. Next, expand the `b3` object that is nested inside of `convertoplan`. Hold the Shift key and select `b3` and everything inside `b3`, then press Delete. Repeat this for all of the objects in `Collection 5`, `Collection 14`, and `Collection 15`. After removing these objects, you should have a screen similar to Fig. 4.2.

4.2.2 Set the Zero Configuration

We want forward for the aircraft to point in the +X (positive X) direction for importing into UE4. Click on the `convertoplan` object to select it. Find the bottom-right **Properties** panel located directly underneath the **Outliner** and click the orange square icon to open **Object Properties**. Change each of the X, Y, and Z Location values to 0, and change the Rotation Z value to 90. Next, move your mouse over to the **Viewport** area, which is the panel currently displaying the mesh. Press the `A` key to select all objects. Press the key combination `Ctrl+A`, then click **Rotation** from the pop-up menu to apply the new rotation.

Now click on the `Gondolas` object. We will refer to these cylindrical meshes as the *engines*. In UE4, we will animate tilting the propellers by tilting the engines, which the propellers will follow. We want the zero rotation of the engines to be vertical, as this is what VTOL-AirSim expects. With `Gondolas` selected, change the Rotation Y value to -90 and press `Enter`. Again, apply the rotation: mouse over to the **Viewport** area, press `Ctrl+A`, then click **Rotation** from the pop-up menu.

4.2.3 Edit the Engines

Select the `Gondolas` object again. Notice that the two engines are a single object in Blender; we will need independent control of the tilt of each engine, so let's split the engines into two objects. In the upper-left of the **Viewport**, click **Object Mode** and select **Edit Mode**. You should now be able to see all the vertices, represented as black points, that make up the mesh of the engines. For the next part, let's move to a back view of the aircraft: in the upper-right of the **Viewport**, look for a graphic showing the global axes in red, green, and blue, and click on the hollow red circle (labeled **-X** when the holding the pointer over it). Also in the upper-right of the **Viewport** is an icon with two squares named **Toggle X-Ray**; click this icon to allow seeing all the vertices simultaneously. We can choose to separate either of the engines, so let's arbitrarily choose to separate the right engine. Draw a box around all the vertices of the right engine. You should now have a screen similar to Fig. 4.3 where all the vertices of the right engine are highlighted orange. Make sure your

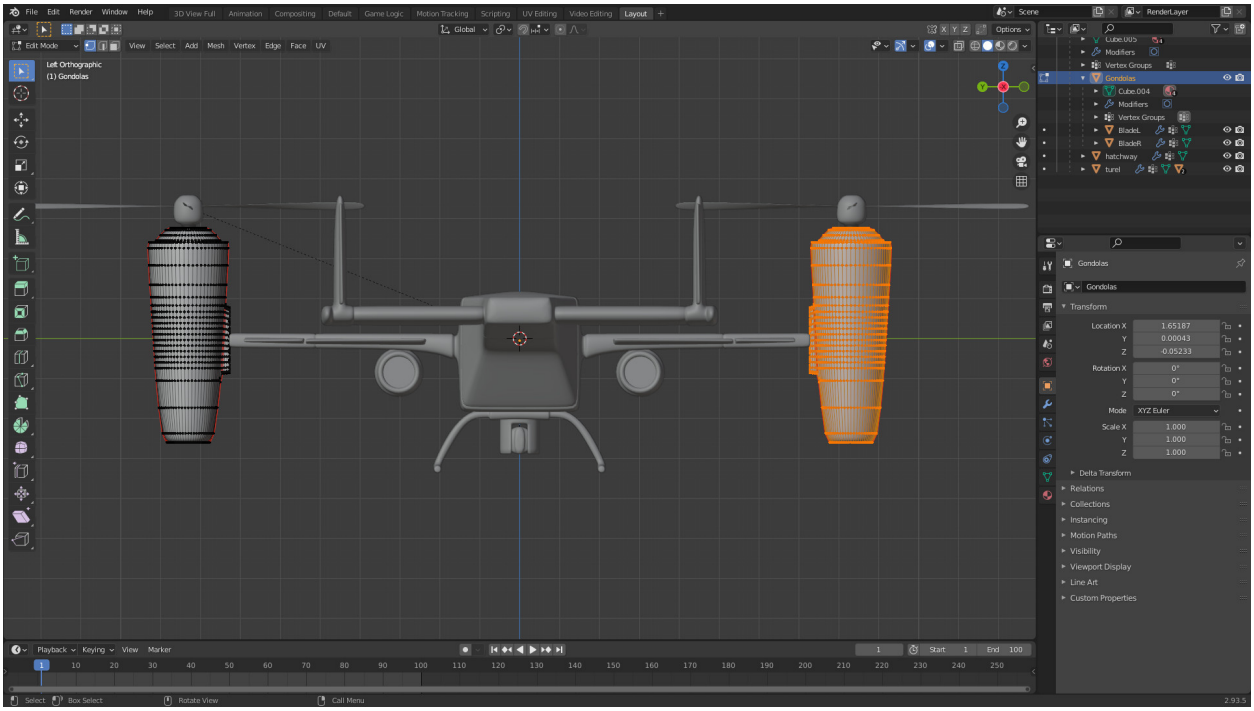

Figure 4.3: This is how the screen should look in Blender after selecting the right engine's vertices.

mouse is over the **Viewport** area, press `P`, then click **Selection** from the pop-up to create a new object. Change from **Edit Mode** back to **Object Mode** and click **Toggle X-Ray** to turn it off.

In the **Outliner**, you should see an object named `Gondolas` and a new object named `Gondolas.001`. Rename `Gondolas` to `GondolaL` and `Gondolas.001` to `GondolaR`. Expand both `GondolaL/R` objects, and find the `BladeL/R` objects under `GondolaL`. Click and drag `BladeR`, then, while dragging, begin pressing `Shift+Alt`, and finally drop it onto `GondolaR`. Doing this *reparents* `BladeR` to `GondolaR`. Next, click and drag `GondolaL`, then again press `Shift+Alt` while dragging, but this time drop it onto `Collection 1`. Repeat this process for `GondolaR`. This will clear their parent, which is needed to avoid a problem that occurs with nested hierarchies when exporting to FBX format.

4.2.4 Move Origins of the Propellers

Next, we will fix the origins of the propellers. These origins will become the pivot points in UE4. Hold the `Ctrl` key and, in the **Outliner**, select `BladeL` and `BladeR`. In the top-left of the **Viewport**, click **Object** to reveal the **Object Menu**. Select **Set Origin** > **Origin to Geometry**. Let's also reset the zero rotations for all objects as their current rotations: hold your pointer over the **Viewport**, press `A` to select all objects, press `Ctrl+A` then click **Rotation**.

4.2.5 Fix Object Names

Finally, let's fix some object names. Rename `convertoplan` to `body`. With `body` expanded, you will see an *Object Data* item with a green triangle mesh icon named `Cube.005`. Give it the same name of `body`. Now do the same for the Object Data items under `GondolaL`, `GondolaR`, `BladeL`, and `BladeR` by renaming them to `GondolaL`, `GondolaR`, etc., respectively. Make sure each object is expanded in order to see its Object Data item. After finishing, press `Ctrl+S` to save the file.

4.2.6 Export to FBX

The mesh is ready for export. Go to **File** > **Export** > **FBX (.fbx)** to open the exporting utility window. On the right column of the window is a list of options. Under **Include** and **Object Types**, hold `Shift` and click **Mesh** and **Other**. Under **Transform** and **Up**, change the value to **Z Up**. Uncheck the option **Use Space Transform**, and check the box for **Apply Transform**. Next, expand the **Geometry** section and set **Smoothing** to **Face**. After that, you should have the settings shown in Fig. 4.4. Finally, keep the file name as `aircraft.fbx` and click **Export FBX**.

4.3 Import Mesh Into UE4

Once you have an FBX file of the mesh, it is time to import it into a project using Unreal Editor. In this section, we will import the mesh that we exported from Blender in the

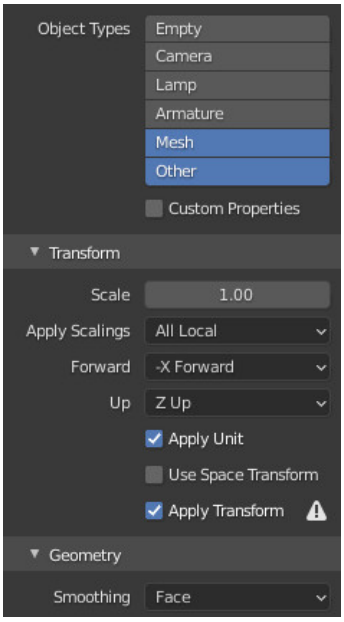

Figure 4.4: These are the required settings when exporting to FBX from Blender.

previous section. To make the instructions easier to follow, let's give a simple, memorable name to the aircraft: Homer, after the ancient Greek poet.

We will begin by creating a new Unreal Project that we can modify for this extended example by making a copy of the Blocks project. Navigate to where you have the Blocks project on your file system and create a simple copy named `Blocks_test`. We want UE4 to rebuild the project, so we need to delete the `Binaries`, `Intermediate`, and `Saved` folders in both the `Blocks_test` and the `Blocks_test/Plugins/vtol-AirSim` folders. The `vtol-AirSim` folder also has a script named `clean.sh` that you can use to automate the removal.

Start Unreal Editor and open the newly created project. To open it, do the following: if Unreal Editor is showing you the **Unreal Project Browser** window, click **More** and then **Browse**; else, if you already have a project opened, go to **File** > **Open Project** and then click **Browse**. Navigate to the `Blocks_test` project and double-click on the file `Blocks.uproject`. In the dialog that appears asking if you would like to rebuild the missing modules of Blocks and AirSim, click **Yes**.

Once the Blocks_test project is opened, look at the **Content Browser** and ensure that **View Options** > **Show Plugin Content** is checked. In addition, show the **Sources**

panel by clicking the icon to the left of the **Filters** icon. Now look at the **Sources panel** and navigate to `AirSim Content/VTOL`. Create a folder here for the new aircraft by pressing `Ctrl+Shift+N`. Name the folder `Homer`. Move inside the `Homer` folder and create two new folders named `Meshes` and `Blueprints`.

Next, let's import the mesh. Go to **File** > **Import Into Level**. In the file browser dialog, find the `aircraft.fbx` file that you exported from Blender earlier. In the next dialog, choose the import location as `AirSim Content/VTOL/Homer/Meshes`. The next dialog will show a number of options. Under **Import Options**, check the boxes for **Import as Dynamic** and **Force Front XAxis**, then click **Import**.

A new editor window will open titled **FbxScene_aircraft**. We will discuss this window in the the next section, so for now, return to the main Unreal Editor window. You should see the `Meshes` folder populated with a number of assets. Note that these new assets are of four types: *Material*, *Static Mesh*, *Blueprint Class*, and *Fbx Scene Import Data*. The type of each asset is indicated by a colored accent along the bottom of the asset's thumbnail: green for Material, cyan for Static Mesh, blue for Blueprint Class, and purple for Fbx Scene Import Data. You can also hover your mouse pointer over any asset and the pop-up that appears will display the asset's name followed by its asset type in parentheses; for example, if you hover over the `body_001` asset, it will display the title *body_001 (Static Mesh)*.

The various materials and the component meshes for the tiltrotor aircraft were originally defined in Blender, and Blender exported data about the parts and materials of the mesh into FBX format. When Unreal Editor imported the FBX file, it tried to interpret the data and create assets that represent what is contained in the data. This process isn't perfect, and you may find that there are discrepancies between what is you can see in the scene in Blender compared with what is generated in UE4 by importing the FBX file. In this example, we won't import any textures; however, in general, you can also import the texture files supplied for a mesh (in the form of image files) separately and apply them in UE4. An easy way to change the look of the aircraft is by making simple modifications to the materials, such as color, roughness, or how metallic they look. UE4 comes with its own Material Editor, so this method allows for changing the look of the aircraft mesh inside Unreal Editor rather than requiring external tools for editing.

4.4 Make a Custom Blueprint

In Unreal Engine, *Blueprints* is a visual scripting system that is used to accomplish a broad range of tasks. It is a way to do visual programming in that virtually anything that the UE4 C++ libraries can do can be done using Blueprints, and vice versa. Nevertheless, some things are more easily done in Blueprints, while others are more easily done in C++ code. If you are familiar with the *LabView* visual programming system from National Instruments, they share some similarities. For our purposes of creating a controllable aircraft, you can also think of it as having similarities with what are called *assemblies* in some CAD software. In addition, a Blueprint is interchangeably called a *Blueprint Class* because it is a class in the same sense of the word's meaning in C++. You create an *instance* of a Blueprint Class by spawning or placing an Actor in the level of that Blueprint Class. As an example, there is one `TiltrotorPawn` C++ class and one `BP_TiltrotorPawn` Blueprint for the tiltrotor vehicle, but we can spawn one or more tiltrotor vehicles in a simulation where each one has a different name, a different state, etc.

While almost everything in VTOL-AirSim is contained in the C++ code, a few aspects concerning the mesh animation is contained in the form of Blueprints. In particular, this includes:

- The Static Meshes that are used to visually represent the aircraft
- The relationships between the meshes (i.e., the hierarchies and relative transforms)
- The mechanism which sets the rotation speed of the propellers
- The Materials assigned to each Material Slot

4.4.1 Notes on Naming Conventions

First, we wish to cover a few definitions and naming conventions. A *Pawn* in Unreal Engine is defined as "the base class of all Actors than can be controlled by players or AI". All of the aircraft in VTOL-AirSim are derived classes of the Pawn C++ class, and thus have the naming convention of appending *Pawn* to the class name. Each vehicle mesh consists of a Blueprint that inherits from one of the vehicle Pawn C++ classes. A Blueprint for a vehicle

mesh has the prefix *BP_*, and this is how AirSim's multirotor vehicle and VTOL-AirSim's tiltrotor got the names `BP_FlyingPawn` and `BP_TiltrotorPawn`, respectively.

4.4.2 Relative Transforms in Blueprints

Section 4.3 had you use the **Import Into Level** feature, and this tries to automatically create a Blueprint based on the data contained in the FBX file. This is the **FbxScene_aircraft** Blueprint. Go back to the editor window for the Blueprint, or double-click on the asset in the **Content Browser** if you no longer have the window. In the Blueprint editor, switch to the **Viewport** tab. On the left of the window, you will see the **Components** panel which contains a hierarchy of *Components* which make up the Blueprint that reflects the same hierarchy of objects that we saw in Blender. Click on the `GondolaR` Static Mesh Component, then look at the **Details** panel on the right side of the window. Notice that its Location X, Y, and Z values reflect their values set in Blender, except that the Y-axis is negated and they are all multiplied by 100 — this is because UE4 uses a left-handed North-East-Up coordinate system, and its base unit of length is the centimeter (Blender is unitless by default, but UE4's process of interpreting the FBX specification causes the values to be multiplied by 100, which is desirable in this case). However, you will see that the same is not true for `BladeR`; the values are a bit different than in Blender. This is because in Blender the Location values are relative to the global coordinate frame (assuming that the Location transform has not been applied), while in UE4, the Location values are relative to the coordinate frame of its *parent*, which in this case is `GondolaR`.

4.4.3 Origins (Pivot Points) of Meshes

Understanding where the origin of a mesh (or *pivot point*, as it is called in Unreal Engine) comes from is very important, because *there is no way to change the origin (pivot point) of a mesh in Unreal Engine.* The pivot point of each Static Mesh is read by UE4 at the time of import and then becomes fixed, both its position and orientation, after the asset is generated. This is why we set the origins of the propeller objects to be at their geometric centers in Section 4.2.4, as it is only in Blender (or some other 3D modeling software) that

the position of the point can be changed, as well as the orientation of the coordinate axes with respect to the mesh. If you are having problems getting the right pivot points for meshes, see Appendix A.3 for an alternate method of correcting origins in Blender.

4.4.4 Create a New Blueprint

Each unique vehicle must have its own Blueprint to fly in VTOL-AirSim. We could modify the `FbxScene_aircraft` Blueprint that UE4 created for us, but this would require more work than modifying a copy of the `BP_TiltrotorPawn` Blueprint, so we will do the latter.

Copying a Blueprint must be done in Unreal Editor. In the **Content Browser**, navigate to the folder `AirSim Content/VTOL/Tiltrotor/Blueprints`. With the **Sources panel** open, click and drag the `BP_TiltrotorPawn` asset to the **Sources panel** and drop it onto the `VTOL/Homer/Blueprints` folder, then select **Copy here** from the pop-up. Navigate to that same folder and rename the copied Blueprint to `BP_HomerPawn`, then double-click it to open the Blueprint Editor. Click on the **Viewport** tab in the Blueprint Editor so you can see the changes we will make in the next steps.

4.4.5 Replace the Static Mesh Components

At this point, you should see the default tiltrotor aircraft in VTOL-AirSim. Let's replace the Static Mesh Components in the Blueprint with the new aircraft mesh. Select all the Static Mesh Components under the `Body` component by holding `Shift` while clicking on them, and press `Delete` to remove them. Next, click on the `Body` component and find the **Static Mesh** property in the **Details** panel. To change it, click on the mesh name to reveal a drop-down list, begin typing `body`, and then select `body_001` (or similarly named) from the filtered list. Now you should see the body of the new aircraft in the **Viewport**; the Static Mesh of the `Body` Static Mesh Component (which is the Blueprint's Root Component) has now been set to the body of the new aircraft mesh.

The next task is to copy the configuration of the Static Mesh Components contained in the `FbxScene_aircraft` Blueprint to the `BP_HomerPawn` Blueprint. Select all the Static

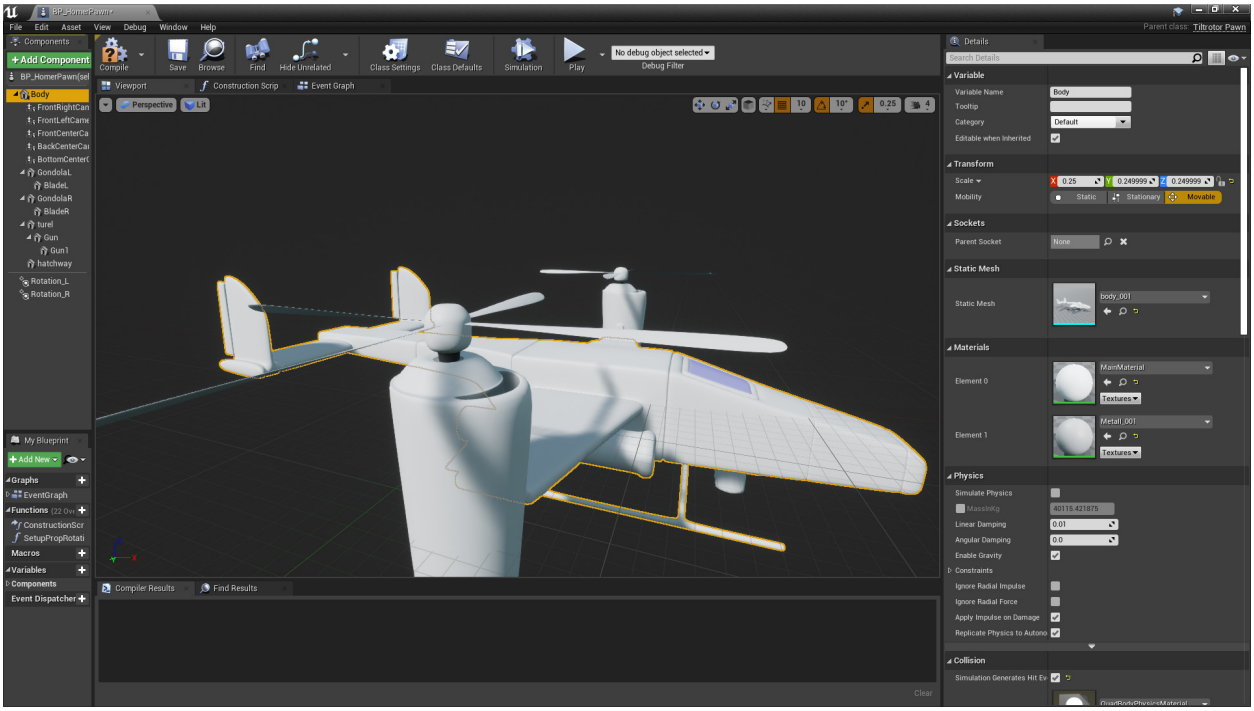

Figure 4.5: The Viewport of `BP_HomerPawn` after adding all of the new aircraft's Static Mesh Components to the Blueprint.

Mesh Components in `FbxScene_aircraft` using `Shift` and click, then right-click on any of the selected components and select **Copy**. In the editor for `BP_HomerPawn`, right-click on `Body` and select **Paste**. Finally, delete the `body1` component from the components that you just pasted. You should now see the full aircraft mesh in the `BP_HomerPawn` Viewport, as shown in Fig. 4.5.

4.4.6 Edit `SetupPropRotationMovement`

There is one visual programming piece of the Blueprint that needs to be edited. In the **My Blueprint** panel located in the lower-left of the window, under the **Functions** section, find the function named `SetupPropRotationMovement` and double-click it to open it in a new tab. Notice the two boxes titled **Set Updated Component** where the left and right boxes have connections to nodes labeled **Rotation L** and **Rotation R**, respectively. Hold your mouse pointer over the **Rotation L** node — but not its label, rather over the empty space to the left of the label text — and the tooltip tells you *Read the value of variable*

Rotation_L. Indeed, this and the **Rotation R** node read the values from the `Rotation_L` and `Rotation_R` Rotating Movement Components of this Blueprint (see the **Components** panel) and feed those values as inputs to the **Set Updated Component** functions.

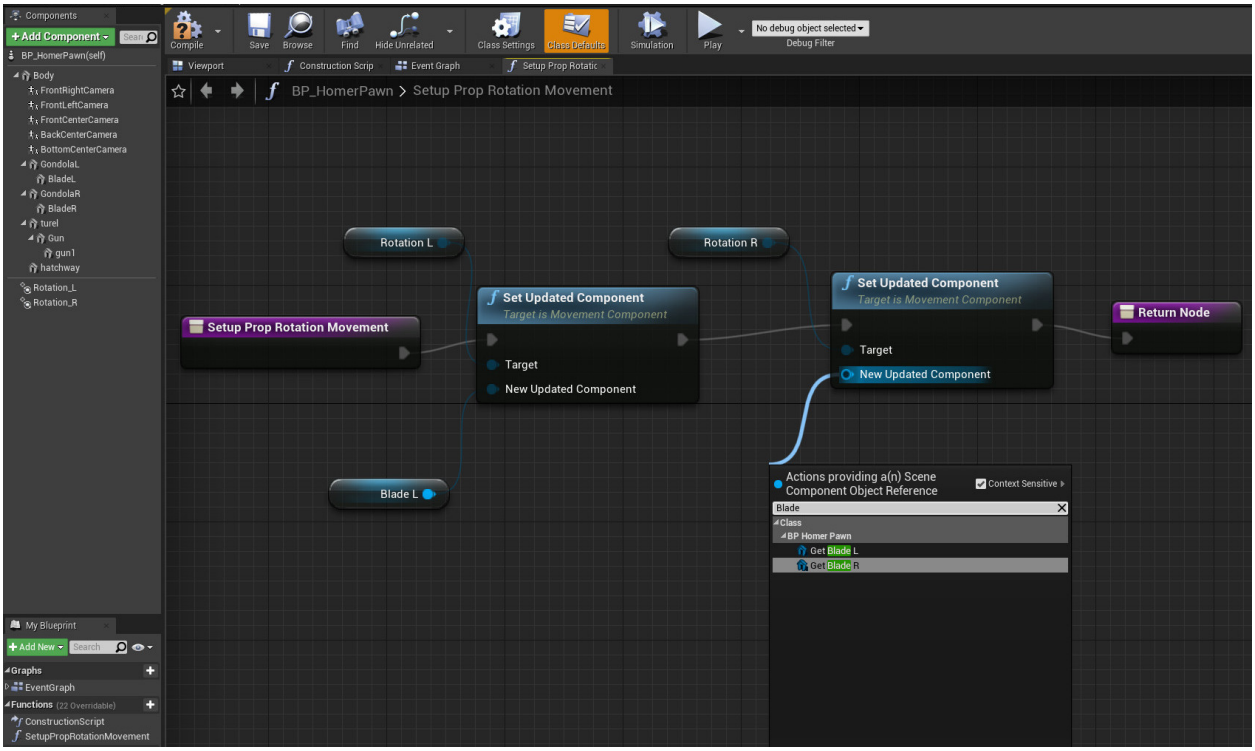

Figure 4.6: In the `SetupPropRotationMovement` function, what appears after clicking on the **New Updated Component** terminal, dragging outward, releasing, and then typing the search term `Blade`.

If you look at the same function in `BP_TiltrotorPawn`, you will see that the each box also has output connections going to nodes labeled **Prop L** and **Prop R**, which correspond to the `Prop_L` and `Prop_R` components of the Blueprint; these nodes and connections were present when we first copied the `BP_TiltrotorPawn` Blueprint, but the nodes were deleted when we deleted the `Prop_L/R` components.

What this function does is it links together the *Rotating Movement Components* in the Blueprint to the Static Mesh Components that we want to animate. In UE4, a Rotating Movement Component is a non-scene component (i.e., it is not rendered and has no position) which simply "performs continuous rotation of a component at a specific rotation

rate". AirSim uses Rotating Movement Components in order to set the speed that the propellers rotate. In the C++ code, it is the `RotationRate.Yaw` of the Rotating Movement Components that get set by their calculated values based on control inputs in AirLib, and it is in this Blueprint function where each of the `Rotation_L/R` components are linked to affect each of the propellers.

In `BP_HomerPawn`, we simply need to have `BladeL/R` be the outputs of the left and right boxes. To create output connections to new nodes for these boxes, click on the terminal (the hollow circle) of **New Updated Component** and drag downward toward any empty area and release; this will reveal a pop-up which allows you to search for the node you want to create (see Fig. 4.6). Type `Blade`, and click on **Get Blade L** for the left box, or **Get Blade R** for the right box. Follow this process for both boxes.

Once you have a connection from the **New Updated Component** terminal of the left box to **Blade L** and a connection from the right box to **Blade R**, you are finished with the Blueprint. Save your changes.

4.5 Perform a Test Flight

We are ready to perform the first test flight of the Homer aircraft in the Unreal Editor. First, you will need to place the following settings in your `settings.json` file:

AirSim Settings For Custom HomerPawn Aircraft

```
{
  "SettingsVersion": 1.2,
  "SimMode": "Vtol",
  "PawnPaths": {
    "DefaultVtol": {
      "PawnBP":
        "Class'/AirSim/VTOL/Homer/Blueprints/BP_HomerPawn.BP_HomerPawn_C'"
    }
  },
  "Vehicles": {
    "uav0": {
      "VehicleType": "VtolSimple"
    }
  }
}
```

In the `PawnPaths` setting, we specify what we want to use as the default Blueprint for all vehicles of type `Vtol` (this can also be configured for each vehicle, if needed) so that AirSim will choose `BP_HomerPawn` rather than the default `BP_TiltrotorPawn`. The somewhat confusing string that is required to specify `BP_HomerPawn` is due to how UE4 deals with paths within projects as well as how UE4 generates object names. In short, the class object for `BP_HomerPawn` becomes `BP_HomerPawn.BP_HomerPawn_C`, contained within the single quotes is the path string to the serialized object, and the text `Class` means the object is of type `Class` — other types include `Material`, `StaticMesh`, etc. See Appendix B.3 for an explanation of path references in UE4.

After you have saved the `settings.json` file, go back to Unreal Editor. In the **World Outliner**, find the **FbxScene_aircraft** Actor that UE4 automatically placed in your level. Select it and press `Delete`. Next, click **Play** in Unreal Editor. You should see a `BP_HomerPawn` Actor spawn in the Blocks world. In a terminal, run the geometric controller example from Section 2.6.4. After running the `geometric_control_airsim_sim.py` script, you should see the aircraft take off and fly until it hits a wall. Notice that the propellers are spinning as they should, but the engines never tilt. Click **Stop** to end the simulation.

There is a very simple reason as to why the engines were not tilting: the names of the engines are hard-coded as `Engine_L` and `Engine_R` in the C++ code, yet HomerPawn's engines are named `GondolaL/R`. There are two ways in which you can solve this problem: you can fix the names of the engine components in the Blueprint, or you can create a new C++ class for the Homer aircraft which uses appropriate names. In order to demonstrate how to create a new vehicle C++ class for those who may need more advanced customizations, we are going to do the latter option. If you do not think you will need to modify any of the VTOL-AirSim code for your project, then you can do the former option and stop there. Otherwise, close out of Unreal Editor and continue to the next section.

4.6 Create a C++ Class

In this section, we create a replica of the `TiltrotorPawn` source code and rename it to `HomerPawn`. Then, we modify the code to correctly animate the tilting of the engines on the Homer aircraft mesh.

4.6.1 Copy Files from TiltrotorPawn

First, go to `Blocks_test/Plugins/vtol-AirSim/Source/Vehicles`. Create a new folder and name it `Homer`. From the `Tiltrotor` folder, copy the files `TiltrotorPawn.h` and `TiltrotorPawn.cpp` into the `Homer` folder and rename them to `HomerPawn.h` and `HomerPawn.cpp`.

Next, open VS Code in the folder `Homer`. Press the keys `Ctrl+Shift+F` to search across all files in the opened directory. Type `Tiltrotor` into the **Search** box. Click the small arrow to the left of the search box to reveal the **Replace** box. Enter `Homer` as the replacement term. Click the **Replace All** icon to the right of the **Replace** box, or press the keys `Ctrl+Alt+Enter`. Click **Replace** in the dialog that appears.

There is one class that should keep its original name — `TiltrotorPawnEvents` — as we don't need to change anything in it to be compatible with `HomerPawn`. Repeat the same process to replace occurrences of `HomerPawnEvents` with `TiltrotorPawnEvents`. Lastly, in `HomerPawn.h`, change the include statement from `"TiltrotorPawnEvents.h"` to use the correct file path of `"Vehicles/Tiltrotor/TiltrotorPawnEvents.h"`. With those changes, you now have a C++ class for `HomerPawn` that can compile without errors.

4.6.2 Correct Names of Engine Components

We still need to fix the names of the engine components before the class will be fully functional. In VS Code, open `HomerPawn.cpp` and find the two lines containing the strings `"Engine_L"` and `"Engine_R"`. Change each string to be `"GondolaL"` and `"GondolaR"`. Notice, in addition, the lines containing the strings `"Rotation_L"` and `"Rotation_R"`; this is where the Rotating Movement Components that we discussed in Section 4.4.6 are specified. We don't need to change those strings because the component names remain the same in `BP_HomerPawn`. Note also that these two arrays of components are looped over in the function `SetRotorRenderedStates` later on in the file. This is where their values are set, and they are set according to the data contained in the vector `rotor_infos`, which ultimately comes from AirLib.

This completes our changes to the code. Run the script `clean.sh` in `vtol-AirSim` and launch Unreal Editor. Open the `Blocks_test2` project and click **Yes** on the dialog to rebuild the modules.

4.6.3 Reparent the Blueprint

Once Unreal Editor has rebuilt and opened the project, open `BP_HomerPawn` again to make a necessary change. In the toolbar near the top of the Blueprint Editor, click on **Class Settings**. In the **Details** panel, find the **Parent Class** field and click on the field's current value of **Tiltrotor Pawn**. Begin typing `Homer` and you should see **Homer Pawn** appear from the list of filtered items. Select **Homer Pawn**, then click **Continue** on the dialog to reparent the Blueprint. The **Homer Pawn** class is the `AHomerPawn` C++ class that you created; if it shows up in the list of classes, that means UE4 successfully built your new class. Lastly, save the changes to the Blueprint.

This step of reparenting creates the actual link between the `BP_HomerPawn` Blueprint and the `AHomerPawn` C++ class. Now, when a `BP_HomerPawn` Actor is spawned, it will use your custom code in `HomerPawn.cpp`.

4.7 Improve the Look With Materials

You are now ready for a fully functional test flight of the Homer aircraft. But before we do the final flight, let's improve the look of the aircraft by changing the Materials assigned to the Material Slots of the various components. We will do that by reusing Materials from the VTOL-AirSim tiltrotor aircraft.

In the Blueprint Editor for `BP_HomerPawn`, make sure the `Viewport` tab is active. In the **Components** panel, select the `Body` component. Find the Material Slot named **Element 0** and click on the currently selected material named **MainMaterial**. Type `body`, and select the first item with the name **body** (from the VTOL-AirSim tiltrotor assets). Repeat this for the engines `GondolaL/R`, and for the `hatchway` component. Do the same for each of the propellers, `BladeL/R`, but instead search for and select the **blades** Material.

In the **Viewport** of the Blueprint Editor, you should see the Homer aircraft with a greatly improved look.

4.8 Final Test Flight

Now let's perform the test flight. Go to the main Unreal Editor window and click **Play**. In a terminal, run the geometric controller script as before. If everything has gone well up to this point, then you should see the engines properly tilting on the Homer mesh as it takes off and flies through the Blocks world (and into a wall), as seen in Fig. 4.7.

This completes our full custom aircraft example. In the next chapter, we show how you can create a custom environment for use with VTOL-AirSim.

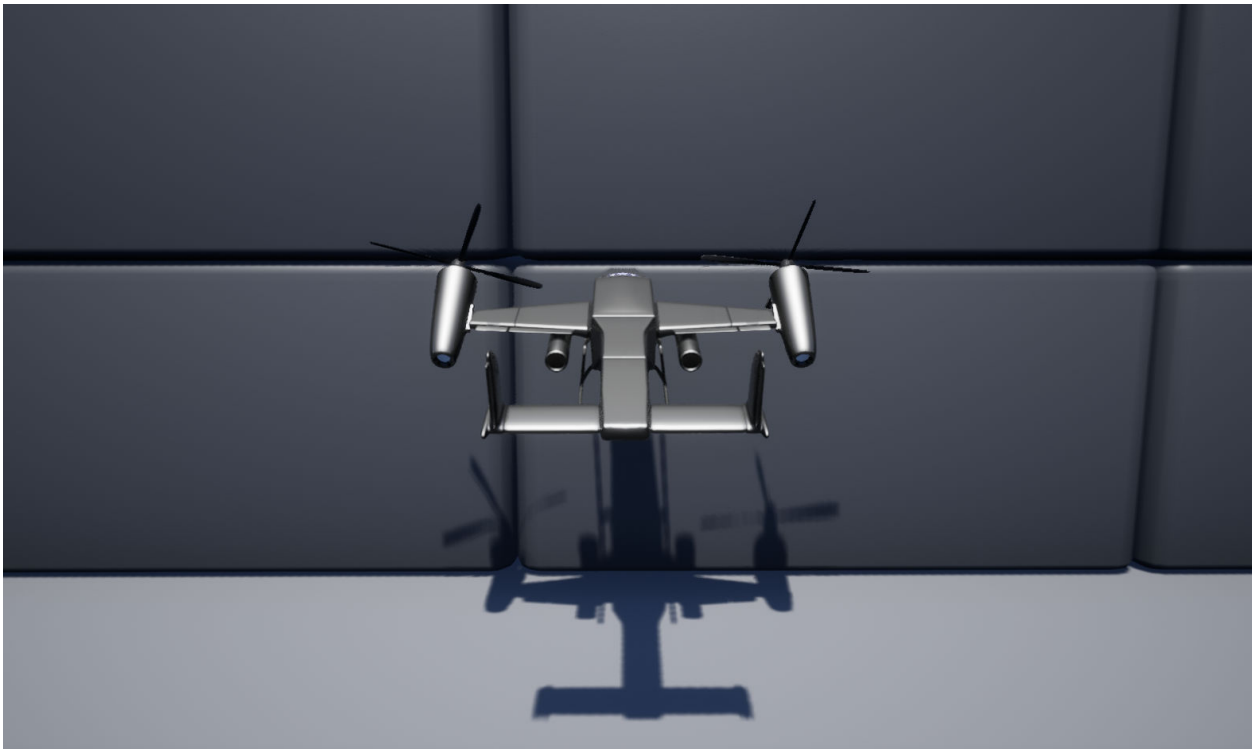

Figure 4.7: The custom Homer aircraft flying in the Blocks world as it hits a wall.

CHAPTER 5. CUSTOM ENVIRONMENTS

This chapter explains how to create a custom environment for use with VTOL-AirSim. Like the previous chapter on custom aircraft, this chapter is structured as a guide through a specific example that demonstrates the complete process starting with obtaining assets from the Unreal Engine Marketplace to a full VTOL-AirSim simulation in the new environment.

We begin with a discussion on obtaining assets from the Unreal Engine Marketplace. The proceeding sections are a guide through a specific example in which we first create a new Unreal Project, then we add to the project Unreal Engine assets from the marketplace, and finally we add the VTOL-AirSim Plugin.

5.1 Unreal Engine Marketplace

The easiest way to create a custom environment is by downloading a prebuilt environment from the Unreal Engine Marketplace, found at `https://unrealengine.com/marketplace`. In contrast to an aircraft in VTOL-AirSim, which is a small mesh made up of a few moving parts, an environment is much larger, and is typically composed of a great number of meshes and other details such as lighting, sky objects, bodies of water, and dynamic actors arranged in a complex layout. Creating an environment — or a *level*, as it is called in UE4 — on one's own requires a deep understanding of Unreal Engine to do well. However, if you are able to obtain a level that is already made, the process is very simple.

The Unreal Engine Marketplace has a large selection of assets available to download. Many of them are paid, though there is a significant amount of free assets to choose from. The assets on the marketplace are of all types of UE4 assets, including Materials, Textures, Static Meshes, Blueprints, etc., so you must be careful in that you choose a product that contains one or more levels among the provided assets. Study the product's description to know whether it comes with a prebuilt level. As an example, the images for a product could

be pictures of different types of buildings with scenic backgrounds, yet the product simply contains an assortment of buildings, without any levels containing those buildings. These are assets that are meant to be used by creators who are designing their own levels. Note that you may find levels referred to as *maps* in product descriptions, due to level assets having the file extension `.umap`. Also, some products do come with prebuilt levels but simply fail to mention this in their descriptions, so you can also check the **Reviews** and **Questions** tabs when viewing a product for more clues.

When searching for assets, you also need to choose assets that are compatible with the version of Unreal Engine which you are using. You can filter results while searching for assets on the UE Marketplace by clicking the label of your version in the **Filter Results** column under **Supported Engine Versions**.

5.2 Obtain Assets from Unreal Engine Marketplace

As of this writing, the only way to download assets obtained on the UE Marketplace is through the *Epic Games Launcher*, and it is only available for Windows or MacOS. While it is theoretically possible to use the Epic Games Launcher on Linux using the compatibility layer *Wine*, that is outside the scope of this text. The example in this chapter has only been tested using Windows, so we will provide instructions for Windows only.

The Epic Games Launcher and Unreal Engine on Windows are free software, so all you need is access to a machine, hard drive, partition, or virtual machine that has an instance of Windows for which you have internet access and admin rights to download and install software. You also need at least 40 GB of free space available in Windows.

5.2.1 Install The Epic Games Launcher and Unreal Engine (Windows)

On Windows, go to `https://epicgames.com/store/en-US/download` to download the Epic Games Launcher installer. Install it and open it. When it opens, it will ask you to sign in to your Epic account. Sign in using your Epic Games account (you should already have one if you've done the steps in Section 3.2.1). Next, click on the **Library** tab. Click the plus icon next to **Engine Versions** to install a version of Unreal Engine (Fig. 5.1). Click

on the version number then select whichever version is compatible with AirSim; at the time of this writing, it is Unreal Engine version 4.25. Note that the last number in the version label, called the patch version, doesn't determine compatibility, so just choose the highest patch version, which is 4.25.4 at the time of this writing. Once you have chosen a version, click **Install**. The download and installation will take a while complete.

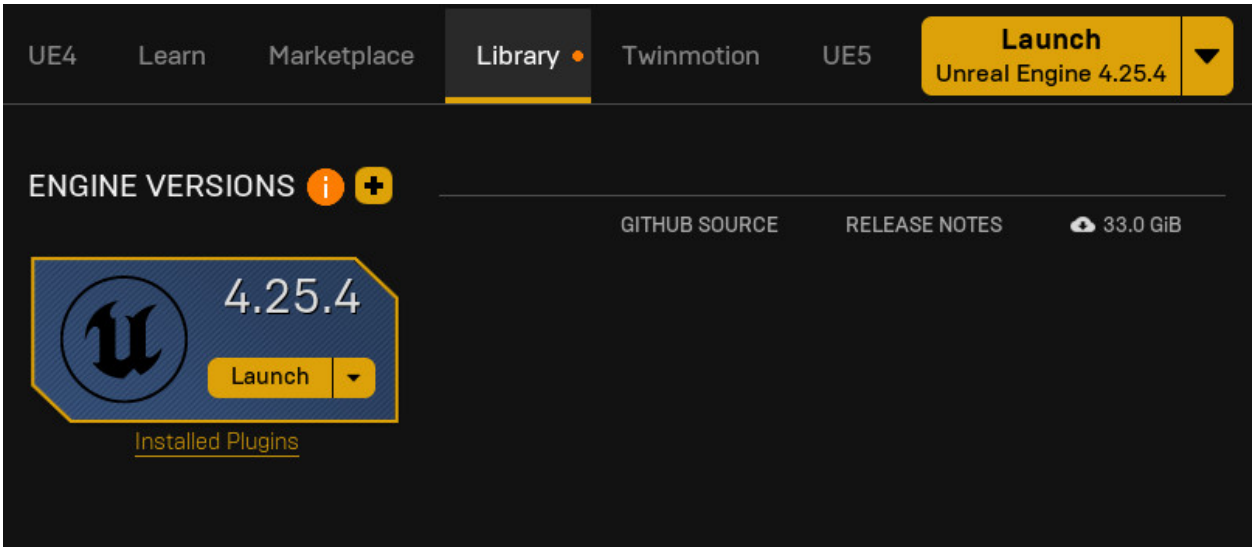

Figure 5.1: The Epic Games Launcher with Unreal Engine 4.25.4 installed.

5.2.2 Create a New Unreal Project (Windows)

When the installation of Unreal Engine has finished, click **Launch** to start Unreal Editor. You will be presented with a window after it initializes. Create a new project by doing the following: select **Games** and click **Next**, then select **Blank** and click **Next**, then click **With Starter Content** and change it to **No Starter Content**, and finally give it the name `CoolBridge` and click **Create Project**.

5.2.3 Download Assets from Marketplace (Windows)

You can explore the UE Marketplace from either a web browser or within the Epic Games Launcher. Let's do so with the Epic Games Launcher, as it allows for direct down-

loading of assets into an Unreal Project. In the Launcher, click the **Marketplace** tab. We are going to download a free asset bundle named *Automotive Bridge Scene* from Epic Games. Search for it using the search bar, and it should be the top or the only result. Click on the product image, then click the button that says **Free** to access it. Once it is yours, click **Add to Project**, then in the pop-up select the `CoolBridge` project and click **Add to Project**. Wait for it to install to your project, then close out of the Epic Games Launcher and Unreal Engine once it finishes.

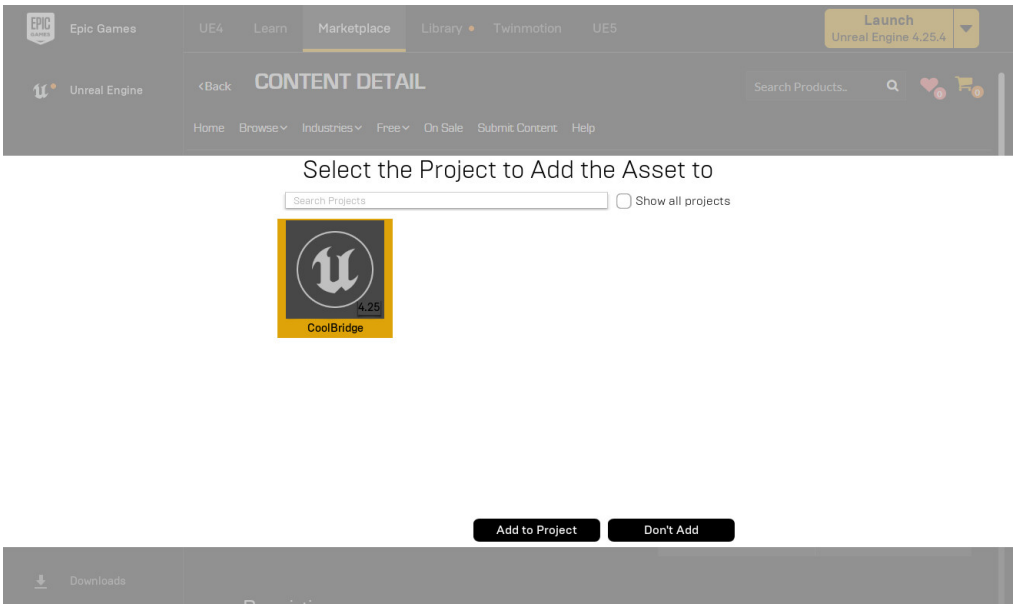

Figure 5.2: In the Epic Games Launcher, this is the screen that allows you to download assets to a project.

As a final step, navigate to the Windows directory where the `CoolBridge` project is located. By default, this should be in your `Documents` folder under `Unreal Projects`. If you look inside the `Content` folder of the project, you'll notice that there is a folder named `AutomotiveBridgeScene`. Note that this folder contains all of the assets that were just downloaded from the marketplace. Now go back to the `CoolBridge` folder and delete the folders `Intermediate`, `Binaries`, `Saved`, `.vs`, and the file `CoolBridge.sln`, if they are present.

That is all we needed to do using Windows. The remaining steps will all be done in Linux.

5.2.4 Set the Project Up on Linux

Copy the `CoolBridge` folder to your Linux file system and place it in a directory of your choice. Create a folder in `CoolBridge` named `Plugins`, and place inside it a working copy of VTOL-AirSim (i.e., not broken) from one of your other Unreal Projects.

5.3 Edit Project in Unreal Editor

Launch Unreal Editor and open the `CoolBridge.uproject` file. In the dialog, select **More Options** then `Convert in-place`, and it will convert your project to work with your version of Unreal Engine on Linux. On the next dialog, click **Yes** to rebuild the modules.

When the project has been built and is opened in the editor, you should get a notification in the bottom-right that says *Project file is out of date. Would you like to update it?* Select **Update**, and this will add an entry to the `CoolBridge.uproject` file that contains the AirSim Plugin as part of the project. Also, before proceeding, go to **Edit** > **Editor Preferences**, and in the search box, type `CPU` and ensure that the setting **Use Less CPU when in Background** is unchecked.

In the **Content Browser**, navigate to the folder `AutomotiveBridgeScene/Maps` and double-click on the level `Bridge_P`. This will load the level in the editor. It may take a while to load the first time. When the Editor has loaded the level and finished compiling the necessary shaders, you should see a picturesque scene of a bridge in the **Level Viewport**.

5.3.1 Configure the Project for VTOL-AirSim

Let's make this level the default one in case we want to package it into a standalone application later. Go to **Edit** > **Project Settings**, then click **Maps & Modes** in the left column. Change both the **Editor Startup Map** and the **Game Default Map** to be the current level, `Bridge_P`. Close out of the project settings.

Next, in order to perform a VTOL-AirSim simulation when we use Play In Editor or when we launch the standalone application, we have to set the *Game Mode* to be `AirSimGameMode`. Do that by going to **Window** > **World Settings**, then in the **World**

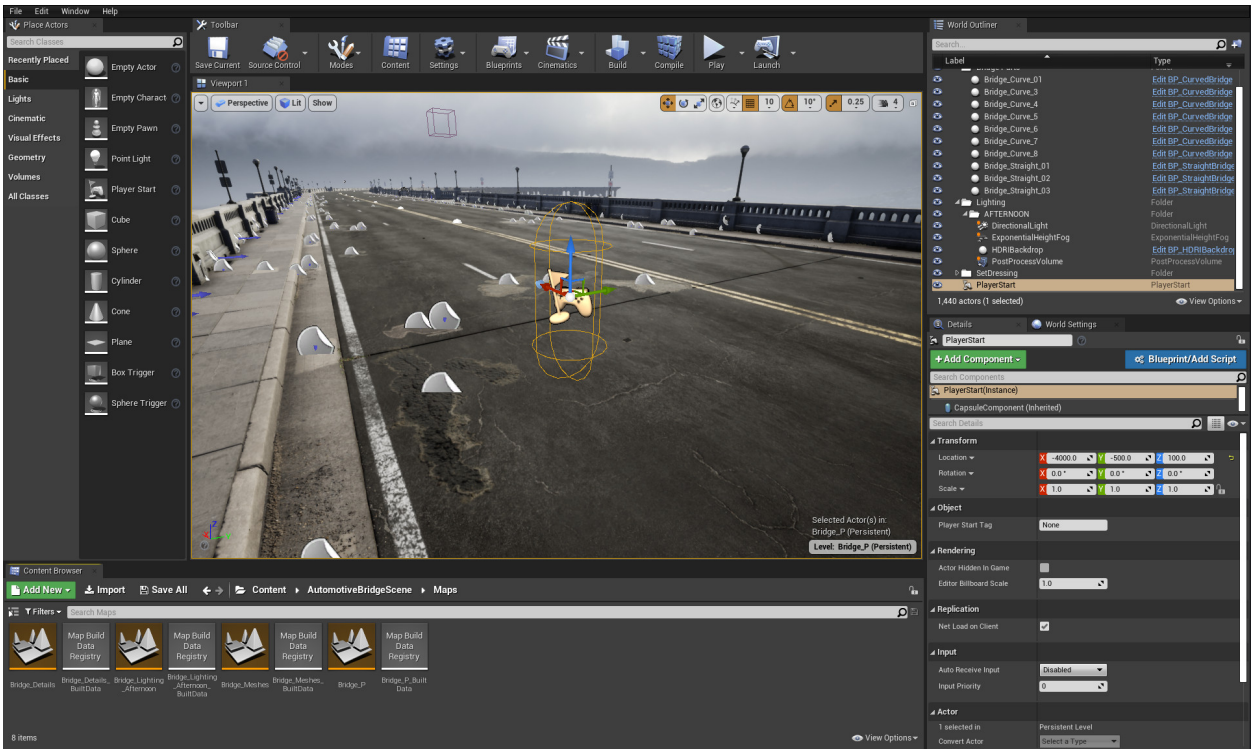

Figure 5.3: The custom environment with a **Player Start** actor placed in the level.

Settings panel in the lower-right corner, find the **GameMode Override** setting and change it to `AirSimGameMode`.

5.3.2 Place a Player Start Actor

Now we need to set where we want the vehicle to spawn in the world — or, more precisely, where we want the global origin to be for VTOL-AirSim. `AirSimGameMode` is configured to look for the **Player Start** Actor to set the global origin and spawn the vehicle; however, there is no **Player Start** Actor in this level, so we need to add one. In the **Place Actor** panel, find **Player Start** and drag it onto the **Level Viewport**. It should now appear in the **World Outliner** panel, and it should also have its attributes visible in the **Details** panel. While you may set the **Player Start** Location to wherever you like, we found a good starting position for this example to be the Location (X, Y, Z) values of (-4000.0, -500.0, 100.0). Set the **Player Start**'s Location attribute to be these values. You can double-click on the `PlayerStart` Actor in the **World Outliner** to center the **Level**

Viewport over its new location. After centering the view, the **Level Viewport** should look similar to Fig. 5.3.

5.4 Flight in New Environment

We are ready to fly the tiltrotor aircraft in the level. First, make sure your settings file at the path `~/Documents/settings.json` is set with `"SimMode": "Vtol"`. Next, press **Play** in the editor, and in the **Level Viewport** you should see the tiltrotor aircraft spawn in the level. In a terminal, run the geometric controller example from Section 2.6.4. You should now see the tiltrotor flying through the custom environment, as shown in Fig. 5.4.

Figure 5.4: The tiltrotor aircraft flying in the *Automotive Bridge Scene* custom environment.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

Autonomous eVTOL aircraft have the potential to provide a new mode of air transportation for humans and cargo that is cheaper, more rapid and more efficient than has previously been possible. Simulation is key to the development of these aircraft, but there is currently a scarcity of simulation tools that have high-quality graphics, are publicly accessible, are easy to use, and which can simulate eVTOLs.

In this work, we contribute *VTOL-AirSim*, a full integration of eVTOL aircraft into Microsoft AirSim. Our work includes new visual components that are packaged with VTOL-AirSim: a fully animated tiltrotor mesh, and a realistic city environment. We also contribute a tutorial on how to use VTOL-AirSim, and a guide on how to extend the capabilities of VTOL-AirSim. Finally, we demonstrate complete examples of creating custom aircraft and environments for VTOL-AirSim, so that others may customize it according to their needs.

This work aims to make VTOL-AirSim as accessible as possible to those seeking to use the simulator and to those who wish to increase its capabilities. It is our hope that others will find VTOL-AirSim to be a useful tool for simulating eVTOL aircraft in visually realistic environments. VTOL-AirSim has already been used in several research projects at the BYU MAGICC Lab. We hope that this trend will continue, and that future work will increase its reach even further.

6.1 Review of Contributions

A user guide for VTOL-AirSim can be found in Chapter 2. We give an overview of the additions made by VTOL-AirSim in Section 2.4 and its unique settings in Section 2.5. In Section 2.6, we give three complete examples of how to fly eVTOL aircraft: how to interface with a trajectory generator, geometric controller, and control allocation module in VTOLsim to send motor PWM commands (Section 2.6.4); how to set the vehicle's state via teleporting

with an external dynamics simulation (Section 2.6.5); and how to interface with the PX4 Autopilot's VTOL controller to fly a mission (Section 2.6.6).

In Chapter 3, we show how to set up and use all the tools required for developing and extending VTOL-AirSim. Section 3.2 explains how to set up Unreal Engine on Linux, and Section 3.3 explains how to set up VTOL-AirSim for development work. In Section 3.4, we give a basic guide on working with VTOL-AirSim in the Unreal Editor.

We give complete examples of how to customize VTOL-AirSim with custom aircraft and custom environments in Chapters 4 and 5, respectively. In Chapter 4, we show how to: obtain an aircraft mesh (Section 4.1), edit a mesh in Blender (Section 4.2), import the mesh into Unreal Engine (Section 4.3), make a new Blueprint (Section 4.4), and finally create a new C++ class for the custom aircraft (Section 4.6). Chapter 5 details how to use the Epic Games Launcher on Windows to obtain assets from the Unreal Engine Marketplace (Section 5.2) and how to edit an Unreal Project with a custom environment in Linux using the Unreal Editor (Section 5.3).

6.2 Future Work

Our work of creating VTOL-AirSim is only the beginning of what can be done for the simulation of eVTOL aircraft in AirSim. There are a few ways in which VTOL-AirSim can be improved, and there are many things that can be added that would greatly increase its utility for several types of research projects. We present here a few examples of improvements that can be made.

First, the teleport functionality in VTOL-AirSim only works for setting the pose of the vehicle. Setting the tilt of the rotors with the teleport command, `simSetVtolPose`, is broken. We attempted to implement this in code via a new function, named `setPoseCustom`, to the `TiltrotorPawnSimApi` C++ class. This function does successfully set the tiltrotor's pose in the data structure holding its physics state, but it appears that the current implementation for setting the rotor tilts is only successful for a brief moment until the motor outputs are overwritten in `vtol_simple::Firmware::update()`, specifically by the call to `board_->writeOutput()`. The firmware is updated at every iteration of AirSim's main loop, and since the firmware hasn't received any commands, it repeatedly sets the rotor tilts

to zero, i.e., their nominal angle. Commands made by the AirSim client are executed on a separate thread from the main loop, and thus the two threads compete with one another to produce instantaneous, rapid shifting of the rotor tilts between their set values and their nominal angle. A solution could be to bypass the `VtolSimple` firmware entirely, or to find a way to work with the firmware to set new motor outputs.

Second, while VTOL-AirSim includes support for tiltrotor aircraft, this is just one of several types of winged eVTOL. It also contains only one specific eVTOL model, the E-flite Convergence. Adding a new aircraft model is fairly straightforward, though it requires modifying the VTOL-AirSim source code and performing a source build. This is also how AirSim implements different multirotor models, so this is not unique to VTOL-AirSim; but it is more cumbersome than it needs to be. Ideally, the Convergence model would be the default, and a user could specify a parameter file that would be read at runtime to change which internal aircraft model is used. Adding support for other types of eVTOL would be a bit more difficult, but we designed VTOL-AirSim with adding other eVTOL types in mind, and there are empty functions left in the code that simply need to be implemented for another eVTOL type and it should work. The more time-intensive component would be to add a new aircraft mesh to represent the new eVTOL type, and if it is to be built in to VTOL-AirSim, how to specify which eVTOL type is desired in the settings file and then dynamically change the mesh of the vehicle.

Finally, it would be ideal for VTOL-AirSim to use a tiltrotor mesh which represents the actual dynamic model. This could be a different mesh which is a true tri-tiltrotor like the E-flite Convergence, or this could be a new dynamic model for a dual-tiltrotor, or a combination of both solutions.

REFERENCES

[1] Microsoft, "Microsoft Flight Simulator," 2020. [Online]. Available: https://www.flightsimulator.com 2

[2] Laminar Research, "X-Plane 11," 2020. [Online]. Available: https://www.x-plane.com 2

[3] R. Gimenes, D. Silva, L. Reis, and E. Oliveira, "Using Flight Simulation Environments with Agent-Controlled UAVs," in *Autonomous Robot Systems and Competitions*, 2008, pp. 21–26. [Online]. Available: http://www.researchgate.net/publication/228447878_Using_flight_simulation_environments_with_agent-controlled_UAVs/file/3deec51a45c7f14b4f.pdf 3

[4] E. Marcu, "Fuzzy Logic Approach in Real-time UAV Control," *Control Engineering and Applied Informatics*, vol. 13, no. 1, pp. 12–17, 2011. 3

[5] R. Louali, A. Belloula, M. S. Djouadi, and S. Bouaziz, "Real-time characterization of Microsoft Flight Simulator 2004 for integration into Hardware in the Loop architecture," in *2011 19th Mediterranean Conference on Control and Automation, MED 2011*, 2011. 3

[6] R. Garcia and L. Barnes, "Multi-UAV Simulator Utilizing X-Plane," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 57, no. 1-4, 2010. 3

[7] S. Cho, "Development of ROS-based Flight and Mission State Communication Node for X-Plane 11-based Flight Simulation Environment," *Journal of Aerospace System Engineering*, vol. 15, no. 4, pp. 75–84, 2021. [Online]. Available: https://www.koreascience.or.kr/article/JAKO202125761289606.pdf 3

[8] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 2004. 3

[9] S. J. Carlson and C. Papachristos, "The MiniHawk-VTOL: Design, Modeling, and Experiments of a Rapidly-prototyped Tiltrotor UAV," in *2021 International Conference on Unmanned Aircraft Systems, ICUAS 2021*, 2021. 3

[10] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," in *Springer Proceedings in Advanced Robotics*, 2018, vol. 5. 3

[11] Epic Games, "Unreal Engine," 2020. [Online]. Available: https://www.unrealengine.com 3

[12] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2015-June, no. June, 2015. 4

[13] Open Robotics, "ROS (Robot Operating System)," 2013. 4

[14] B. Ruf, S. Monka, M. Kollmann, and M. Grinberg, "Real-time on-board obstacle avoidance for UAVs based on embedded stereo vision," in *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, vol. 42, no. 1, 2018. 4

[15] E. Bondi, D. Dey, A. Kapoor, J. Piavis, S. Shah, F. Fang, B. Dilkina, R. Hannaford, A. Iyer, L. Joppa, and M. Tambe, "AirSim-W: A simulation environment for wildlife conservation with UAVs," in *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies, COMPASS 2018*, 2018. 4

[16] J. B. Willis, "Trajectory Generation and Tracking Control for Winged Electric Vertical Takeoff and Landing Aircraft," Ph.D. dissertation, Brigham Young University, 2021. [Online]. Available: https://scholarsarchive.byu.edu/etd 5

[17] J. B. Willis and R. W. Beard, "Pitch and Thrust Allocation for Full-Flight-Regime Control of Winged eVTOL UAVs," *IEEE Control Systems Letters*, vol. 6, 2022. 5

[18] Blender Foundation, "Blender," 2021. [Online]. Available: https://www.blender.org 74

APPENDIX A. MISCELLANEOUS INSTRUCTIONS

A.1 Authentication for GitHub Using `gh` Client

An easy way to configure your Linux environment to clone, push, pull, etc. from GitHub repositories is to use the official GitHub CLI tool called `gh`. First, go to `https://github.com/cli/cli/blob/trunk/docs/install_linux.md` and follow the instructions to install `gh`. Then, in a terminal, run the command `gh auth login`, which will guide you through a number of steps, for which you should do the following:

- Choose `GitHub.com`, then choose `SSH`
  - If you haven't generated a public SSH key previously, then choose "yes" at the prompt by pressing Enter, then press Enter again to leave passphrase blank
  - If you already have a public SSH key, then choose it when prompted
- Choose `Login with a web browser`
- Copy the given code, then press Enter; this will open your web browser
- Authorize in web browser using the copied code
- Go back to the terminal, then press Enter to finish

You can now use either `git` commands or `gh` commands to interact with GitHub repositories. For example, to clone the AirSim repository, you can run either of the following:

Clone a GitHub Repository Option 1: `git`

```
git clone git@github.com:microsoft/AirSim
```

Clone a GitHub Repository Option 2: `gh`

```
gh repo clone microsoft/AirSim
```

For more information about connecting to GitHub with SSH, see the GitHub docs at `https://docs.github.com/en/authentication/connecting-to-github-with-ssh`.

A.2 Advanced Setup for Python Virtual Environments

You can place these bash functions in either your `.bashrc` file or `.zshrc` file, or in another file which is sourced by either of those files. Then, you can run, for example, the command `workon airsim` to activate your Python virtual environment named `airsim`. Run the command `workoff` to deactivate your virtual environment.

```
workon() {
    if [[ -n "$VIRTUAL_ENV" ]]; then
        deactivate
    fi
    source $HOME/.virtualenvs/$1/bin/activate
}

workoff() {
    if [[ -n "$VIRTUAL_ENV" ]]; then
        deactivate
    fi
}
```

A.3 Alternate Method for Moving Mesh Origins in Blender

In the past, we have accomplished getting the correct pivot points in Unreal Engine using a different method than the procedure outlined in Section 4.2. The method is to:

- create separate `.blend` Blender files for each object mesh that you want to animate on the aircraft (e.g., engines, propellers, ailerons)

- Do the following for each Blender file:
  - If there are multiple meshes, choose one as the root object and make the other objects the children of it using the drag, then `Shift+Alt`, then drop technique
  - Hold pointer over the Viewport and press `A` to select all objects
  - Select **Object** > **Set Origin** > **Origin to Geometry** to set origins of the meshes to their geometric centers
  - Select **Object** > **Snap** > **Cursor to World Origin**
  - Select the object, or, if there are multiple objects, select the root object
  - Select **Object** > **Snap** > **Selection to Cursor**
  - If there are any nonzero values for the Location of the object (or root object), apply the Location by holding the pointer over the Viewport and pressing `Ctrl+A` then clicking **Location**
  - Repeat the process to apply all Rotations
  - Export to FBX file with default settings under the **Transform** section:
    - **Up** set to **Y Up**
    - **Use Space Transform** checked
    - **Apply Transform** unchecked
- After finishing the exports, import each FBX file into UE4
- Create a Blueprint with the aircraft's body as the Root Component
- Add the other meshes to the Blueprint
- Manually set each component's Location to where they should be relative to the aircraft body

APPENDIX B. FURTHER INFORMATION

B.1 List of Available Prebuilt Environments for Linux

This list is based on the available environments as of AirSim v1.5.0.

- AbandonedPark — contains a few scattered objects such as a ferris wheel and some swings
- Africa — open ground covered in patches of water, bordered by trees
- AirSimNH — a suburb with diverse features such as houses, cars, trees, and mobile animals
- Blocks — very simple example world with large geometric shapes scattered around
- Building99 — inside of a mall with a number of people walking about
- LandscapeMountains — large, open, and detailed mountain environment
- MSBuild2018 — tennis courts, trees, and buildings making up the Microsoft campus
- TrapCamera — an open and configurable world with some animated, stationary animals
- ZhangJiajie — an area above the clouds with large, stone pillars of varying heights scattered about

B.2 Blender for Work Involving Meshes

Blender is an extremely powerful 3D modeling application that is open-source, completely free, and has full Linux support [18]. For these reasons, we highly recommend using Blender for any work that involves editing, producing, or preparing meshes for Unreal Engine.

The examples in this text using Blender were done with Blender v2.93.5. Our experience has been good with regards to modifying a mesh and exporting it from Blender in FBX format for import into UE4. While not perfect, it is the only solution we know of that works reliably. This is the pipeline we used for the tiltrotor mesh that comes with VTOL-AirSim. We also used Blender to make a number of modifications to the mesh.

On Ubuntu 20.04, there are three ways to install Blender:

1. via their website at `https://blender.org/download` (Recommended)
2. via `apt` with the command `sudo apt install blender`
3. as a Snap application with the command `sudo snap install --classic blender`

We recommend the first option because it will get you the latest version of Blender (as will the third option, if you use Snap applications). The examples in this text were tested using the latest version of Blender, v2.93.5 — however, at the time of this writing, the latest version of Blender available via `apt` is v2.82. If you do use Blender v2.82, we cannot guarantee that the examples will work according to the instructions we provided.

If you install via the official `blender.org` website, extract the archive file to somewhere on your machine and launch blender by running the `blender` executable found inside the extracted directory.

B.3 Unreal Engine Paths And References

The way that Unreal Engine references paths in a project is rather unconventional. It is important to understand how it works if you are going to do work in the Unreal Editor. The following list explains the different path names that UE4 uses and what you can substitute them with to get the corresponding path on your file system. In the examples given, we will assume a project named `Blocks`; the absolute path to the project is omitted for brevity.

- Prefix `/Game/`: substitute with `<project>/Content`
  - Example path: `/Game/Flying/Meshes`
  - refers to: `Blocks/Content/Flying/Meshes`

- Prefix `/AirSim/` : substitute with `<project>/Plugins/<plugin_dir>/Content`
  - Note: inside UE4, the name of the VTOL-AirSim Plugin is **`AirSim`**. UE4 does not read the directory name; it reads what is inside the `AirSim.uplugin` file, which we have kept as `AirSim` for compatibility with the base AirSim repository. Thus, `<plugin_dir>` could be `Plugins/AirSim` or `Plugins/vtol-AirSim` .
  - Example path: `/AirSim/VTOL/Tiltrotor/Meshes`
  - refers to: `Blocks/Plugins/vtol-AirSim/Content/VTOL/Tiltrotor/Meshes`
  - Example path: `/AirSim/Blueprints`
  - refers to: `Blocks/Plugins/AirSim/Content/Blueprints` (if using the default AirSim plugin rather than VTOL-AirSim)

In the **Content Browser**:

- `Content` : substitute with `<project>/Content`
  - Same as `/Game/` above
- `AirSim Content` : substitute with `<project>/Plugins/<plugin_dir>/Content`
  - Same as `/AirSim/` above