

# Omniverse and Unreal Engine Co-Simulation for a Virtual F1 Attendance Prototype

Phillip Stranger

August 9, 2023

## 1 Introduction

The aim of this project was, to build a prototype, showcasing the capabilities a Virtual F1 Attendance application would need and how it could look. The application in mind (Virtual F1 Attendance) would enable a User to watch a virtual F1 race as if he was attending the race at the racetrack, providing the needed immersion into the virtual world for an exhilarating experience.

In order to accomplish this, the choice of a suitable game engine for development needs to be determined. Given the growing popularity of NVIDIA's Omniverse and the increasing adoption of USD (Universal Scene Description) as a standard for 3D development applications, the decision was made to utilize Omniverse in conjunction with Unreal Engine 5. This choice will not only facilitate the creation of the prototype but also allow for testing the collaborative and co-simulation capabilities between Omniverse and Unreal Engine, as Omniverse provides a Connector for Unreal Engine (UE). As a prerequisite we recommend looking at the project "Co-Simulation UE and Omniverse - "SpaceVerse" GTC demo - Return on Experience", as many parts of this project base on it as well as the idea for delving into co-simulation between Omniverse and Unreal Engine was sparked by this project.

### 1.1 USD and Its Advantages

Universal Scene Description (USD) is a versatile and efficient open-source file format developed by Pixar Animation Studios. It serves as a universal standard for storing and exchanging 3D scene data, fostering collaboration and streamlined workflows in the computer graphics industry. The features of USD, such as scene representation, interoperability, scalability and performance, incremental updates, non-destructive editing, and support for procedural and layered workflows, make it a valuable tool for creating and managing complex 3D content across various applications and pipelines.

To further explore the capabilities and potential applications of USD, it is highly recommended to refer to the "Introduction to USD" document available on the official website.

### 1.2 Collaboration and Co-simulation in Omniverse and Unreal Engine

One of the remarkable features enabled by USD is the collaboration and co-simulation capabilities between different applications. In this project, the focus was on testing the effectiveness of collaboration between Omniverse and a major game engine like Unreal Engine. The collaboration between Omniverse and Unreal Engine allows for seamless integration of assets and materials from either engine. As both engines are capable of working with USD as the underlying file format, all assets and materials used in either program are saved within the same USD file in a live collaboration setting. For instance, it becomes possible to build a map in Unreal Engine and utilize the Omniverse Physics Engine to create and record a vehicle driving around the Unreal Engine map. The resulting animation can then be used within Unreal Engine itself. This level of collaboration is achievable without the need for data type conversion, as everything is based on USD.

The collaboration and co-simulation feature can prove highly valuable in various stages of the development process, particularly during the prototyping phase and when rapid iterations are required. Co-simulation facilitates rapid prototyping and iterative development by leveraging different engines for different stages of development. Developers can use a high-level engine for quick prototyping and game-play testing, while employing a specialized engine for optimizing and fine-tuning specific aspects such as physics or AI.

Another significant advantage of Omniverse's collaboration and co-simulation capabilities is evident in cross-platform development. Omniverse provides a connector for both Unreal Engine and Unity, two widely recognized game engines. Co-simulation aids in cross-platform game development by enabling developers to test and optimize their games on multiple platforms simultaneously. Utilizing different engines for different platforms ensures optimal performance and consistent user experiences across various devices.

### 1.3 USD Property Readout

The ability to read out underlying USD properties offers further versatility, allowing the construction of applications on top of applications built using different engines. For example, in a racing game developed in one engine, it becomes possible to build an application that enables users to read out engine values in any program or engine as long as it bases on USD. This flexibility allows for extensive scene editing, including adjustments to models, lighting, physics, and more.

## 2 Prototype: Virtual Attendance of Virtual F1 Race

This section outlines the step-by-step implementation of a prototype showcasing how virtual attendance of virtual sport events, in this case Virtual F1, could be achieved. The objective of such a virtual attendance application is to provide Formula 1 fans with an immersive experience by virtually attending the race as if they were present at the racetrack. The prototype's general idea and technical specifications are as follows:

#### General Idea:

- Simulation of a racetrack with racecars on it.
- Multiple viewpoints available along the track.
- Users can switch between different viewpoints to watch the race from their preferred perspective.

#### Technical Specifications / Implementation:

- Cameras positioned at various locations around the racetrack, allowing users to view in all directions (3DOF).
- Enable users to switch between different camera views.
- Construct a simple 3D model of a racetrack and a racecar.
- Animate the racecar to navigate around the track.

### 2.1 Implementation of the Prototype

#### 2.1.1 Creating a drivable Vehicle

To implement the prototype, the first step is to create the drivable racecar and this is achieved by using the Physics Vehicle Wizard provided by Omniverse. The creation of the drivable vehicle as well as the recording of the animation in a later section bases on this live-stream by *Omniverse3D*, so we strongly recommend watching the live-stream for further insights into the vehicle creation process.

Any 3D car model can be transformed into a drivable vehicle using this tool. In this case, the model chosen is the Red Bull RB6 from the year 2010, which can be downloaded as an FBX file from the provided link. This model was created by *FabianTheFormula1Fan* and is free to download. The FBX format is compatible with Omniverse, eliminating the need for conversion.

To begin, we open the Omniverse USD Composer and navigate to the Content Browser. We search for and open the downloaded 3D model, which should now appear in the center of the scene. However, the model may include unnecessary lighting and camera components, which we can remove. In the Stage, we delete the *LightPreset* and *CameraPreset* Prims included in the RB6 Model. If the scene appears completely black, we select the *Default* Light Rig from the available lighting options. Additionally, the model may require adjustments in terms of its orientation. For instance, in the case of the RB6 Model, the 3D model was vertically oriented, so a rotation of -90 degrees around the X-axis is needed to correct its placement.

Next you may want to delete unnecessary prims from the model. For example, the RB6 model includes a break-light overlay, which could be used to create a functional break-light, but we will not be utilizing it for this prototype. Thus, we delete the Prim (*object\_250*).

We now focus on preparing the model for the Omniverse Physics Vehicle Wizard. The Physics Vehicle created by the Vehicle Wizard consists of five Prims: one for the chassis and one for each wheel. Therefore, we need to separate the individual objects of the 3D model and combine them into their respective Prims. For example, in the case of the Red Bull RB6 Model, prims *object\_20*, *object\_38*, *object\_49*, and *object\_50* (break disc, rim, tire sidewall and tread) are selected and then grouped together by right-clicking and selecting *GroupSelected*. This action creates a new Prim with the four objects as its children. We rename this Prim to *RearRightWheel*. We repeat this process for all four wheels and afterwards combine the remaining objects to form the *Chassis* Prim. Once the objects are separated into the five Prims, we can save the model as a USD file by selecting *File/SaveAs....*

With the 3D model now separated into the chassis and four wheels, we can start with the creation of the Vehicle. However, before we create the vehicle, we first need to create the ground plane on which the vehicle will be able to drive. This can be achieved by selecting *Create/Physics/GroundPlane*.

Next we want to create the Vehicle via *Create/Physics/Vehicle*, which opens the Physics Vehicle Wizard. We begin by focusing on the *ChassisBox* section, where we ensure proper weight distribution for the vehicle. Initially, selecting the *Chassis* Prim of the model results in an incorrect weight distribution, as it would make the vehicle top-heavy. To address this, we go to the Stage, Right Click and Create a Cube Mesh, which we then scale to a box, as wide as the car, but only half its height. An example of how the weight distribution cube can look is presented in Figure 1. This configuration provides a low and stable weight distribution suitable for our F1 Car. We then proceed to scan the newly created cube as our Chassis Box within the Vehicle Wizard. By selecting the cube in the Stage and clicking *Scan*, we obtain the dimensions (length, width, and height) of our weight distribution chassis. As we have a F1 Car we want to adjust the Mass to something lightweight, such as 650 kg. Furthermore, we can customize parameters such as horsepower, RPM, and the number of gears. For the F1 Car, we set 750 horsepower, 15000 RPM, and 8 gears. It is essential to align the longitudinal axis with the axis pointing forward in the model (in this case, axis Z).

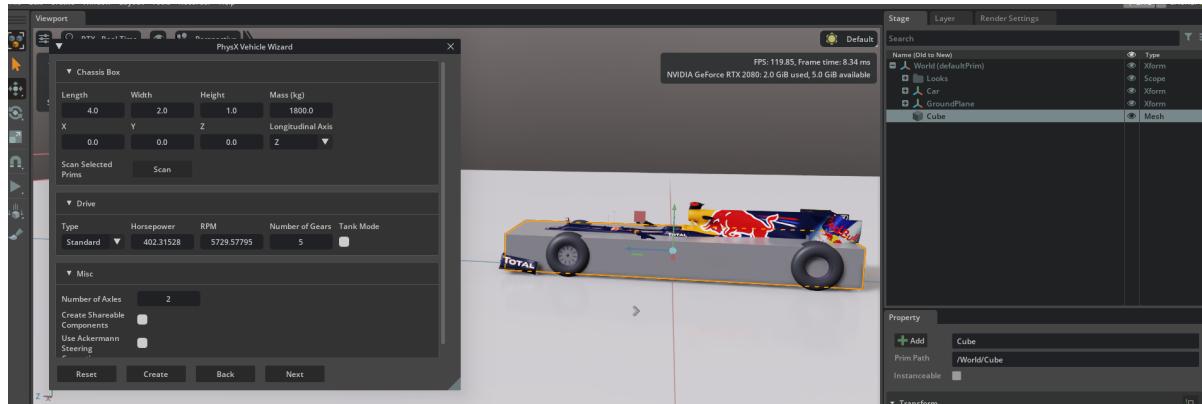


Figure 1: Example of how the weight distribution cube can look.

Moving on to the *Misc* section of the Vehicle Wizard, no additional modifications are required, allowing us to proceed to the next section by clicking *Next*. This section focuses on the axles and wheels of the vehicle. Initially, we set the steering angle to 20 degrees. If desired, four-wheel steering can be enabled by specifying the steering angle for the second axle. In the Vehicle Wizard, *Axle 1* corresponds to the

front axle, while *Axle 2* represents the rear axle of our vehicle. In this case, the rear axle is designated as the drive axle, by only checking the *Drive* checkbox for *Axle 2*. Weight distribution adjustments can be made, although no changes were made in this prototype. *Suspension Damping* is set to 0 for an extremely stiff F1 car without any suspension. However, for more accurate simulations, some level of damping can be introduced.

Now we focus on the Wheels, where each wheel is scanned individually by selecting the corresponding wheel's prim in the Stage and clicking the corresponding button in the Vehicle Wizard. For example, to scan the rear right wheel, we select the prim of the rear right wheel in the Stage and click the *Right* button in the second row of the Vehicle Wizard. The scanned Wheel now appears in the Wheels Section. This workflow is showcased in Figure 2.

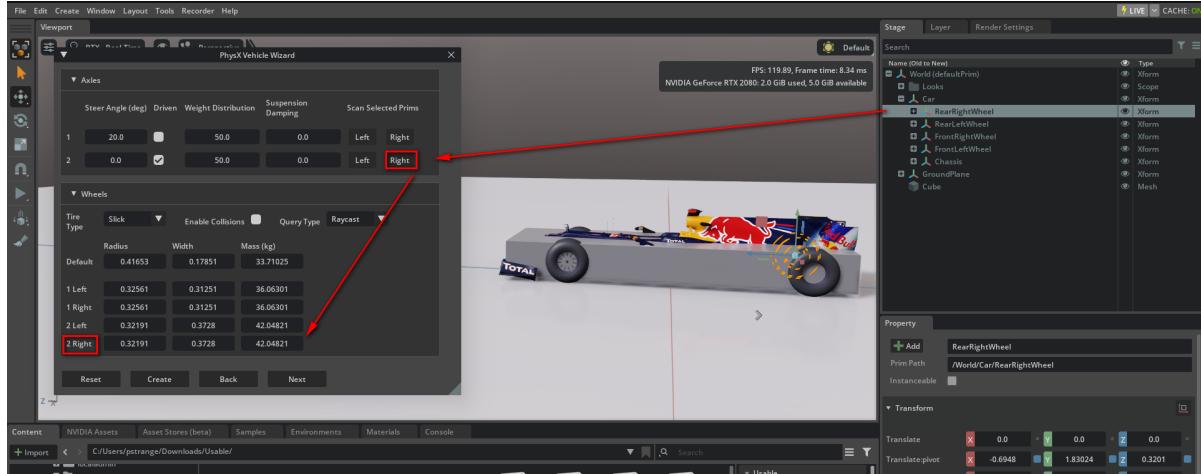


Figure 2: Showcase of the workflow to scan a tire prim in the Vehicle Wizard.

Within the Wheels section, we select the tire type as *Slick* and adjust the mass of each tire to be a bit more lightweight. For a truly accurate simulation these weights should be set more precisely.

With the necessary preparations complete, we can now proceed to create the drivable 3D vehicle model. Upon clicking the *Create* button, we should see the vehicle, which is just a box, equal to our weight distribution cube, with four cylinders attached as tires. If you press the Play button you can now test drive the vehicle for the first time. However, upon turning left and right, you may observe significant wobbling, which requires correction. To address this issue, we adjust the *SpringStrength* and *TravelDistance* of the springs in the Wizard Vehicle.

To begin, we select the *LeftWheel1References* and the *RightWheel1References* under *WizardVehicle1/Vehicle*. With these prims selected we scroll down in the Properties Window until we find the *Vehicle Suspension* section. Here, we set the *Spring Strength* to a high value (e.g., 100000) and set the *Travel Distance* to half of the tire radius. The tire radius can be found two sections above in the *Vehicle Wheel* section. For instance, in the case of the RB6 Model, the tire radius is 0.32561, thus the suspensions *Travel Distance* is set to 0.162805. Similarly, we perform the same adjustments for the rear wheels by selecting the *LeftWheel2References* and *RightWheel2References* under *WizardVehicle1/Vehicle*, setting the *Spring Strength* to 100000, and configuring the *Travel Distance* as half of the rear tire radius. For the used RB6 model, the rear tire radius is 0.32191, resulting in a suspension *Travel Distance* of 0.16095. After making these adjustments, the vehicle should no longer exhibit wobbling during driving and now we can get to the final step of creating our drivable 3D Vehicle Model.

The final Step is to drag the Prims of the 3D model to the corresponding Prims in the created Vehicle Model. We begin by dragging the Front Left Wheel Prim to the *LeftWheel1References* of the Wizard Vehicle. Following this, the Front Left Wheel Prim may become displaced, which can be corrected by adjusting the *Translate : pivot* property, setting all values to 0. This realigns the Left Front Wheel to its correct position. Subsequently, we can either delete or hide the Left Wheel Cylinder (denoted as *Render*) of the Wizard Vehicle. Figure 3 showcases, where the *FrontRightWheel* prim needs to be

dragged to in the Wizard Vehicle.

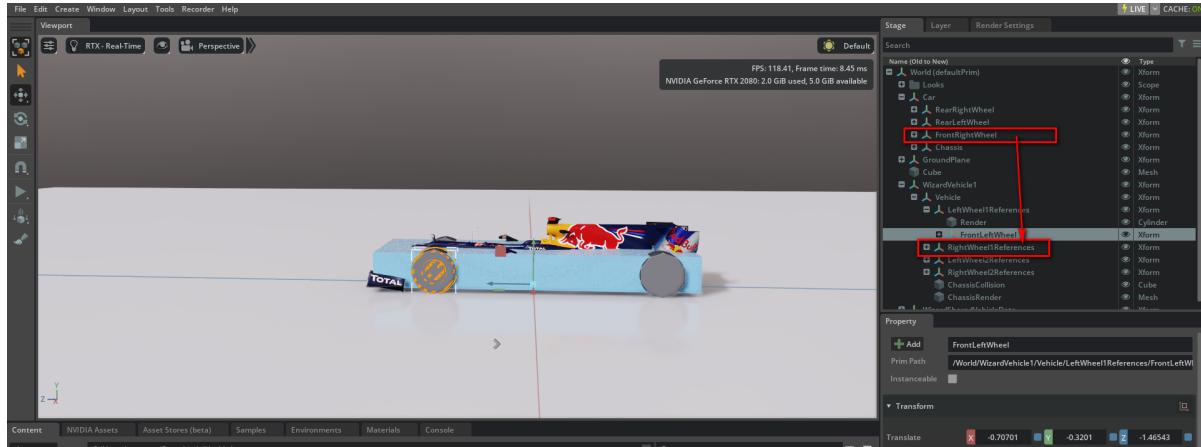


Figure 3: Showcase of where the *FrontRightWheel* prim needs to be dragged to in the Wizard Vehicle.

We repeat this process for all four Wheels and then move the Chassis Prim of our 3D Model into the *Vehicle* Prim of the Wizard Vehicle. Finally, we hide or remove the *ChassisRender* prim. Upon ensuring that all Prims of our model are correctly positioned within the Wizard Vehicle and the renders of the original Wizard Vehicle are hidden or removed, we achieve a drivable model based on the original 3D model.

When considering the performance of the vehicle, there are additional configuration options available. We can adjust the *Mass* of the car or modify the *PeakTorque* of the Vehicle Engine to fine-tune its performance. Additionally, if the wheels exhibit excessive spinning, we can modify the tire friction in the *SlickTireFrictionTable* within the *WizardSharedVehicleData*. There you just have to go down to the Raw USD Properties and increase the *Default Friction Value*.

### 2.1.2 Creating the Map

After creating the vehicle our focus now shifts to the creation of a Map. In this section, we will not provide a detailed, step-by-step guide on map creation. Instead, we give some pointers on how to design a racetrack suitable for use in Omniverse, along with a brief introduction to the collaboration feature of the Omniverse and Unreal Engine Connector.

The first possibility to create a racetrack is to export the Vehicle Test map from Unreal Engine to Omniverse and then use the Spline Meshes from this Test track to build a custom racetrack. However, this is quite tedious and therefore we can use a quicker approach, by creating the Track in Unreal Engine with the help of Splines and then export the racetrack to Omniverse. An excellent tutorial on road creation using splines in Unreal Engine can be found [here](#).

### 2.1.3 Live Collaboration

In this section, we will demonstrate the live collaboration feature by utilizing the Vehicle Test map from Unreal Engine, exporting it to Omniverse, and showcasing the addition of assets while working on the map simultaneously in both Omniverse and Unreal Engine. This project has also a very good introduction into the process of creating a live collaboration between Omniverse and Unreal Engine so we highly recommend checking it out.

First, we want to create a new Unreal project, selecting the Vehicle Project from the Games Section as our starting project. After creating the project, we want to add our Nucleus Server to the Unreal Engine Content Browser. We do this by clicking *Add Server* in the Omniverse Menu in the left upper corner of the Unreal Engine Window and then specifying the name of the Server we want to connect to i.e. the name of your Nucleus Server. To set up a Nucleus Server, the Nucleus Workstation must be installed through the Omniverse Application, followed by the configuration of a local server. Further details regarding the Nucleus Workstation installation and setup can be found in the official Omniverse documentation.

After successfully connecting to the Nucleus Server, an Omniverse Folder will be visible in the Unreal Engine Content Browser. Next, we export the Unreal Engine Map to Omniverse by selecting *Export Level to Omniverse*. Within the Omniverse Exporter, the option *Export as Y – up Axis* must be chosen, as Omniverse employs a Y-Up Axis, while Unreal Engine uses a Z-Up Axis as default. Save the level to the pre-established Nucleus Server, allowing for easy accessibility within Omniverse, without the need to search for the file within local storage.

Once exported we switch over to Omniverse USD Composer and open the exported UE Level. Here we want to create a .live file, by selecting the Root Layer in the Layer Section and then generating a new Sublayer, which we then save as a .live file, as shown in Figure 4. If prompted to transfer the Root Layer contents to the new sublayer, select *Yes*.

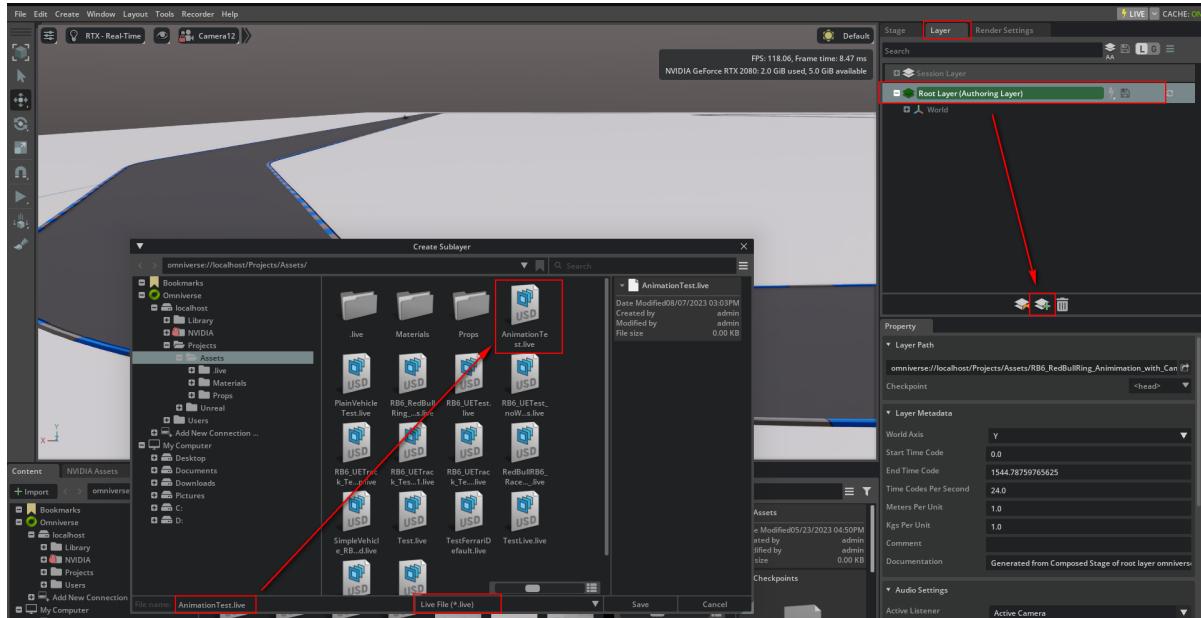


Figure 4: Showcase of how a .live file is created.

After successfully creating the .live file, open it in Omniverse and then switch back to Unreal Engine, where we add an Omniverse Stage Actor to the Scene. In the Details section of the Omniverse Stage Actor, locate the USD Property and drag and drop the previously created .live file onto it. This action loads the .live file, allowing for real-time editing from both applications. It is important to note that when you add materials in Unreal Engine you have to export the Unreal Engine scene to Omniverse after finishing editing the .live file. Similarly, if you add materials and assets in Omniverse you need to save the file in Omniverse.

Once you are finished with editing your map, make sure to save the scene in Omniverse as a .usd file. To add the previously created vehicle, there are two possible approaches. The first approach involves dragging the 3D car model into the scene and utilizing the Vehicle Wizard to create the drivable vehicle. Alternatively, the drivable vehicle created earlier can be dragged into the newly created scene. It is also feasible to reverse the process, where the just-created map is added to the scene containing the vehicle.

#### 2.1.4 Driving and Recording the Car Animation

With a functioning car and racetrack, we can now proceed to drive the car on the track and record it using the *Recorder* to create an animation. The *Recorder* is a native Extension, which can be activated in the Extensions Window (In Omniverse USD Composer you find the Extension Window under *Window/Extensions*). In the Extension Window simply search for *Recorder* and activate it. It should now appear in the Toolbar between *Tools* and *Help*.

Utilizing the *Recorder* is straightforward. In the Stage, simply select the desired Prim(s) to be recorded. For our purposes, we select the *Vehicle* Prim in the Stage and then navigate to the *Recorder*, clicking

on *Record Selection*, as is shown in Figure 5. It is important to note that this action only selects the prims to be recorded and does not initiate the recording process. Next, activate the *Timeline* by going to

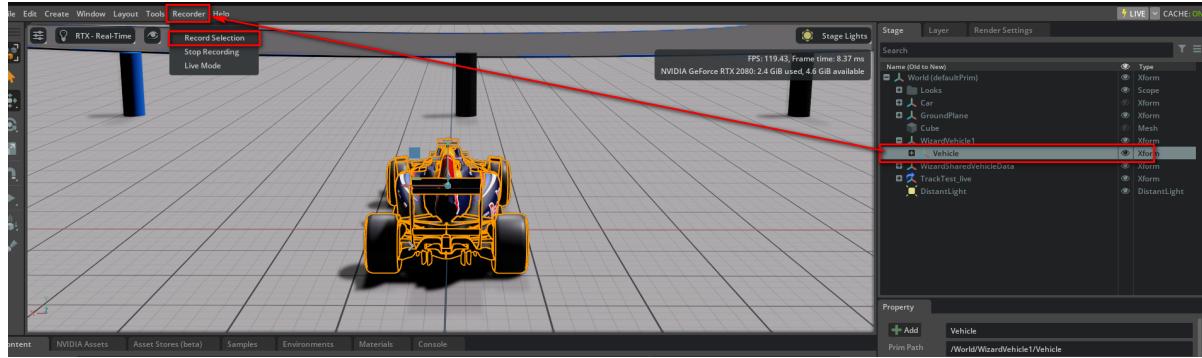


Figure 5: Showcase of how to enable the recording for the *Vehicle* prim.

*Window/Animation/Timeline*, which will appear at the bottom of the window. Within the *Timeline*, specify the desired duration for the animation. In this example, a length of 1000 units is chosen. Upon pressing play, the animation recording starts, and it is crucial to ensure that the recording does not exceed the specified animation length. Overstepping this limit may result in the recorder overwriting and potentially crashing the animation. Therefore, closely monitor the recording duration displayed in the timeline. After driving around the racetrack for the set duration stop the scene, which stops the timeline, and click *Stop Recording* in the *Recorder*. Following the stopping of the recording, save the file separately as a .usda file by navigating to *File/SaveAs....* Here it is important to save the file as a usda (USD ASCII – human readable) file, as we need to edit the file in the next step.

### 2.1.5 Editing the Animation File

Note: This step could change in future versions of Omniverse USD Composer, but at the time of project release, this editing of the animation file is still the required workflow.

After saving the animation, the next step involves editing the .usda file and removing the physics properties from it, as the recorded animation of the car should no longer be drivable. For this process the USDA Editor Extension needs to be enabled. Access the Extensions menu by selecting *Window/Extensions* and ensure that the USDA Editor extension is enabled. To edit the file, navigate to the Content Browser and right-click on the recently created and saved file, then choose *Edit....* The file should open in Visual Studio Code, assuming it is installed. It is recommended to use Visual Studio Code, as Omniverse supports VS Code Link, which enables the connection of an Omniverse app to VS Code's Python debugger. This feature proves beneficial when creating custom extensions, a topic that will be explored later on.

In the editor, the objective is to delete all physics references from the USDA file. Consequently, all physics-related variables must be removed from the vehicle prim (referred to as *WizardVehicle1* in this example) in the .usda file. Each component of the vehicle that has been separated (*Chassis*, *LeftFrontWheel*, ...) will contain certain physics properties that need to be deleted. Meaning all properties containing *physics* or *physx* need to be removed.

For clarity Figure 6 shows the *Vehicle* prim in the pre-edited state, while Figure 7 shows the *Vehicle* prim with all physics references removed. Figures 8 and 7 show the same thing for the *LeftWheel1Reference*. Once all physics references are deleted, save the file and return to Omniverse USD Composer. Right-click on the file in the Content Browser and select *Finished editing* to finalize the changes.

```

4107
4108     def Xform "Vehicle"
4109     {
4110         preprend apISchemas = ["PhysxVehicleDriveStandardAPI", "PhysxVehicleEngineAPI", "PhysxVehicleGearsAPI", "PhysxVehicleAutoGearBoxAPI", "PhysxVehicleClutchAPI", "PhysxVehicleBrakesAPI", "PhysxVehicleSteeringAPI",
4111         "PhysxVehicleWithWheelDifferentialAPI", "PhysxRigidBodyAPI", "PhysxMassAPI", "PhysxRigidBodyAPI", "PhysxVehicleAPI", "PhysxVehicleControllerAPI"]
4112     }
4113
4114     vector3f physics:angularVelocity = (0, 0, 0)
4115     vector3f physics:angularVelocity:timeSamples = (-
4116
4117         point3f physics:centerOfMass = (-0.16, -0.2361679)
4118         float3 physics:diagonalInertia = (169.28, 1268, 106.84799)
4119         float physics:mass = 487.8802
4120         const point3 physics:centerOfMass = (0, 0, 0)
4121         vector3f physics:velocity = (0, 0, 0)
4122         vector3f physics:velocity:timeSamples = (-
4123
4124             bool physics:rigidBody:isMobileGravity = 1
4125             float physics:rigidBody:sleepThreshold = 0.0046251426
4126             float physics:rigidBody:stabilizationThreshold = 0.0009250285
4127             int physics:wheelIndex = 0
4128             int physics:wheelLowForwardSpeedStepCount = 1
4129             float physics:wheelMinActiveLongitudeSlipDenominator = 0.1
4130             float physics:wheelMinForwardSpeedStepCount = 1
4131             float physics:wheelMinForwardSpeedLongitudeSlipDenominator = 4
4132             float physics:wheelSuspensionStepThresholdLongitudeSlipSpeed = 5
4133             uniform token physics:suspensionLineQueryType = "raycast"
4134         )
4135         const point3 physics:autocenterBoxPositionRelPos = []
4136         float physics:autocenterBoxOrder = 1
4137         float physics:autocenterBoxOrder:latency = 2
4138         float physics:clutchMaxForce = 1
4139         float physics:clutchMaxForceBreakawayForce = 6399.231
4140         float physics:clutchStrength = 16.789599
4141         float physics:controllerAccelerator = 0
4142         float physics:controllerAccelerator:timeSamples = (-
4143
4144             float physics:controllerBraked = 0
4145             float physics:controllerBraked:timeSamples = (-
4146
4147                 int physics:controllerTargetBrake = 255
4148                 float physics:engineDampingRateWithZeroTrottleClutchEngaged = 0.25064263
4149                 float physics:engineDampingRateZeroTrottleClutchEngaged = 0.5848328
4150                 float physics:engineDampingRateZeroTrottleClutchEngaged = 3.3419018
4151                 float physics:engineMaxTorque = 10000.0
4152                 float physics:engineMaxTorque = 3487.1975
4153                 float physics:engineMaxTorque = 1.6795999
4154                 float physics:enginePeakTorque = 712.0756
4155                 float2f physics:engineTorqueCurve = ((0, 0), (0.35, 1), (1, 0.0))
4156                 float2f physics:engineTorqueCurve = ((0, 0), (0.35, 1), (1, 0.0))
4157                 float physics:gearboxRatioScale = 4
4158                 float physics:gears:shiftTime = 0.5
4159                 float physics:wheelDifferentialTorqueRatios = [0.5, 0.5]
4160                 float[] physics:wheelDifferentialWheels = [0, 1]
4161                 float[] physics:wheelSteeringAngleMultipliers = [1, 1]
4162                 float physics:wheelSteeringWheels = [0, 1]
4163                 float physics:wheelSteeringWheels = [0, 1]
4164                 quat xformOp:origin = (0, 0, 0)
4165                 quat xformOp:origin = (1, 0, 0)
4166                 quat xformOp:origin = (3.3842793377277, 1, -0.080802996245, 3.206322e-11, -1.0038564e-9),
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
4399
4400
4401
4402
4403
4404
4405
4406
4407
4408
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
4549
4550
4551
4552
4553
4554
4555
4556
4557
4558
4559
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4579
4580
4581
4582
4583
4584
4585
4586
4587
4587
4588
4589
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
4599
4600
4601
4602
4603
4604
4605
4606
4607
4608
4609
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
4699
4700
4701
4702
4703
4704
4705
4706
4707
4708
4709
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
4749
4750
4751
4752
4753
4754
4755
4756
4757
4758
4759
4759
4760
4761
4762
4763
4764
4765
4766
4767
4768
4769
4769
4770
4771
4772
4773
4774
4775
4776
4777
4778
4778
4779
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
4799
4800
4801
4802
4803
4804
4805
4806
4807
4808
4809
4809
4810
4811
4812
4813
4814
4815
4816
4817
4818
4819
4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4829
4830
4831
4832
4833
4834
4835
4836
4837
4838
4839
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
4849
4850
4851
4852
4853
4854
4855
4856
4857
4858
4859
4859
4860
4861
4862
4863
4864
4865
4866
4867
4868
4869
4869
4870
4871
4872
4873
4874
4875
4876
4877
4878
4878
4879
4879
4880
4881
4882
4883
4884
4885
4886
4887
4888
4889
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
4899
4900
4901
4902
4903
4904
4905
4906
4907
4908
4909
4909
4910
4911
4912
4913
4914
4915
4916
4917
4918
4919
4919
4920
4921
4922
4923
4924
4925
4926
4927
4928
4929
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
4949
4950
4951
4952
4953
4954
4955
4956
4957
4958
4959
4959
4960
4961
4962
4963
4964
4965
4966
4967
4968
4969
4969
4970
4971
4972
4973
4974
4975
4976
4977
4978
4978
4979
4979
4980
4981
4982
4983
4984
4985
4986
4987
4988
4989
4989
4990
4990
4991
4992
4993
4994
4995
4996
4997
4998
4998
4999
4999
5000
5001
5002
5003
5004
5005
5006
5007
5008
5009
5009
5010
5011
5012
5013
5014
5015
5016
5017
5018
5019
5019
5020
5021
5022
5023
5024
5025
5026
5027
5028
5029
5029
5030
5031
5032
5033
5034
5035
5036
5037
5038
5039
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
5049
5050
5051
5052
5053
5054
5055
5056
5057
5058
5059
5059
5060
5061
5062
5063
5064
5065
5066
5067
5068
5069
5069
5070
5071
5072
5073
5074
5075
5076
5077
5078
5078
5079
5079
5080
5081
5082
5083
5084
5085
5086
5087
5088
5089
5089
5090
5091
5092
5093
5094
5095
5096
5097
5097
5098
5099
5099
5100
5101
5102
5103
5104
5105
5106
5107
5108
5109
5109
5110
5111
5112
5113
5114
5115
5116
5117
5118
5119
5119
5120
5121
5122
5123
5124
5125
5126
5127
5128
5129
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
5149
5150
5151
5152
5153
5154
5155
5156
5157
5158
5159
5159
5160
5161
5162
5163
5164
5165
5166
5167
5168
5169
5169
5170
5171
5172
5173
5174
5175
5176
5177
5178
5178
5179
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5189
5190
5191
5192
5193
5194
5195
5195
5196
5197
5198
5199
5199
5200
5201
5202
5203
5204
5205
5206
5207
5208
5209
5209
5210
5211
5212
5213
5214
5215
5216
5217
5218
5219
5219
5220
5221
5222
5223
5224
5225
5226
5227
5228
5229
5229
5230
5231
5232
5233
5234
5235
5236
5237
5238
5239
5239
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249
5249
5250
5251
5252
5253
5254
5255
5256
5257
5258
5259
5259
5260
5261
5262
5263
5264
5265
5266
5267
5268
5269
5269
5270
5271
5272
5273
5274
5275
5276
5277
5278
5278
5279
5279
5280
5281
5282
5283
5284
5285
5286
5287
5288
5289
5289
5290
5290
5291
5292
5293
5294
5295
5296
5297
5297
5298
5299
5299
5300
5301
5302
5303
5304
5305
5306
5307
5308
5309
5309
5310
5311
5312
5313
5314
5315
5316
5317
5318
5319
5319
5320
5321
5322
5323
5324
5325
5326
5327
5328
5329
5329
5330
5331
5332
5333
5334
5335
5336
5337
5338
5339
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
5349
5350
5351
5352
5353
5354
5355
5356
5357
5358
5359
5359
5360
5361
5362
5363
5364
5365
5366
5367
5368
5369
5369
5370
5371
5372
5373
5374
5375
5376
5377
5378
5378
5379
5379
5380
5381
5382
5383
5384
5385
5386
5387
5388
5389
5389
5390
5391
5392
5393
5394
5395
5395
5396
5397
5398
5399
5399
5400
5401
5402
5403
5404
5405
5406
5407
5408
5409
5409
5410
5411
5412
5413
5414
5415
5416
5417
5418
5419
5419
5420
5421
5422
5423
5424
5425
5426
5427
5428
5429
5429
5430
5431
5432
5433
5434
5435
5436
5437
5438
5439
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
5449
5450
5451
5452
5453
5454
5455
5456
5457
5458
5459
5459
5460
5461
5462
5463
5464
5465
5466
5467
5468
5469
5469
5470
5471
5472
5473
5474
5475
5476
5477
5478
5478
5479
5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5489
5490
5490
5491
5492
5493
5494
5495
5495
5496
5497
5498
5499
5499
5500
5501
5502
5503
5504
5505
5506
5507
5508
5509
5509
5510
5511
5512
5513
5514
5515
5516
5517
5518
5519
5519
5520
5521
5522
5523
5524
5525
5526
5527
5528
5529
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
5549
5550
5551
5552
5553
5554
5555
5556
5557
5558
5559
5559
5560
5561
5562
5563
5564
5565
5566
5567
5568
5569
5569
5570
5571
5572
5573
5574
5575
5576
5577
5578
5578
5579
5579
5580
5581
5582
5583
5584
5585
5586
5587
5588
5589
5589
5590
5591
5592
5593
5594
5595
5595
5596
5597
5598
5599
5599
5600
5601
5602
5603
5604
5605
5606
5607
5608
5609
5609
5610
5611
5612
5613
5614
5615
5616
5617
5618
5619
5619
5620
5621
5622
5623
5624
5625
5626
5627
5628
5629
5629
5630
5631
5632
5633
5634
5635
5636
5637
5638
5639
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
5649
5650
5651
5652
5653
5654
5655
5656
5657
5658
5659
5659
5660
5661
5662
5663
5664
5665
5666
5667
5668
5669
5669
5670
5671
5672
5673
5674
5675
5676
5677
5678
5678
5679
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5689
5690
5691
5692
5693
5694
5695
5695
5696
5697
5698
5699
5699
5700
5701
5702
5703
5704
5705
5706
5707
5708
5709
5709
5710
5711
5712
5713
5714
5715
5716
5717
5718
5719
5719
5720
5721
5722
5723
5724
5725
5726
5727
5728
5729
5729
5730
5731
5732
5733
5734
5735
5736
5737
5738
5739
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
5749
5750
5751
5752
5753
5754
5755
5756
5757
5758
5759
5759
5760
5761
5762
5763
5764
5765
5766
5767
5768
5769
5769
5770
5771
5772
5773
5774
5775
5776
5777
5778
5778
5779
5779
5780
5781
5782
5783
5784
5785
5786
5787
5788
5789
5789
5790
5791
5792
5793
5794
5794
5795
5796
5797
5798
5799
5799
5800
5801
5802
5803
5804
5805
5806
5807
5808
5809
5809
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
5849
5850
5851
5852
5853
5854
5855
5856
5857
5858
5859
5859
5860
5861
5862
5863
5864
5865
5866
5867
5868
5869
5869
5870
5871
5872
5873
5874
5875
5876
5877
5878
5878
5879
5879
5880
5881
5882
5883
5884
5885
5886
5887
5888
5889
5889
5890
5891
5892
5893
5894
5895
5895
5896
5897
5898
5899
5899
5900
5901
5902
5903
5904
5905
5906
5907
5908
5909
5909
5910
5911
5912
5913
5914
5915
5916
5917
5918
5919
5919
5920
5921
5922
5923
5924
5925
5926
5927
5928
5929
5929
5930
5931
5932
5933
5934
5935
5936
5937
5938
5939
5939
5940

```

without requiring any modifications. In the Content Browser of Unreal Engine, search for the .usda file containing the animation and open it. In the *Outliner* you should now see a *LevelSequenceActor* holding the Animation. If you now Play the Level you should see your Animation being played. If the animation does not play as expected, access the Level Sequence and adjust the Start and End time in the Playback Options, which should resolve the issue.

### 2.1.7 Creating Cameras

Next, we focus on the last piece of the Virtual F1 Attendance Prototype, which involves creating cameras positioned around the track to provide different viewpoints for viewing the animation. Further, we will develop an extension to facilitate switching between these cameras.

To create a camera, right-click within the stage and select *Create/Camera*. Place the cameras in the desired locations. For easier placement, switch to the viewpoint of each camera and adjust its position accordingly. With the cameras in place, the extension for the switching between viewpoints can be created. To create a new extension, go to *Window/Extensions* and in the Extension Window click the green Plus icon and select *New Extension Template Project*. Choose a folder location to save the extension project. After selecting a folder, provide a name for the extension project, such as *kit-exts-camera-switcher*, followed by assigning an ID for the extension, which will be displayed in the Extension window. For instance, *company.camera.switcher* can be used as the ID.

Once the extension is created, Visual Studio Code should open with the extension's files displayed. In the root of the extension, locate the *exts* folder, which holds the extension. Inside this folder, find the directory named after the specified extension ID. Navigate further into this folder until you locate the *extension.py* file and open it. In this File you will find an *on\_shutdown* and an *on\_startup* function. We can ignore the *on\_shutdown* function and focus on the *on\_startup* function. To begin, retrieve the active viewport window, representing the open scene's viewport, using the *get\_active\_viewport\_window()* function from the viewport utility library, which needs to be imported. Then, obtain the stage of the scene by calling *omni.usd.get\_context().get\_stage()*. This retrieves the currently opened stage.

Once the stage is obtained, traverse the stage to search for all cameras. Utilize the *Traverse* function and examine each Prim to determine if it is a camera. For cameras found, save their names and prim paths. Once all cameras in the scene are gathered, create buttons for each camera and assign a function to execute when a button is clicked. This function will switch the current viewport to the corresponding camera associated with the clicked button.

The complete *on\_startup* function as well as all needed imports for this extension are provided below and it should be possible to simply copy and paste the code into the *extension.py* file without any errors. With the completion of this extension, the animation can be played, and users can easily switch between the different cameras placed within the scene. Note that the traversal of the scene also includes the built-in cameras *Perspective*, *Top*, *Front*, and *Right*. With the extension's functionality implemented, the Virtual F1 Viewer Prototype is now complete and Figure 10 presents how the camera switching extension looks.

```
# ----- Imports -----
import omni.ext
import omni.ui as ui
import omni.kit.viewport.utility as vp_utils
import omni.usd
from pxr import UsdGeom

# ----- on_startup -----
def on_startup(self, ext_id):
    print("[strange.camera.switcher] strange camera switcher startup")

    self._count = 0

    self._window = ui.Window("Camera Switcher", width=300, height=300)
    viewportWindow = vp_utils.get_active_viewport_window()

    with self._window.frame:
        with ui.VStack():


```

```

stage = omni.usd.get_context().get_stage()
if stage is not None:
    # find all cameras in the stage
    cameras = [[x.GetName(), x.GetPrimPath()] for x in stage.Traverse()
                if x.IsA(UsdGeom.Camera)]

if len(cameras): # If cameras are found -> create button per camera
    with ui.VStack():
        def on_click(camera_name, camera_path):
            viewportWindow.viewport_api.camera_path = camera_path
            print('Switched to camera: ' + camera_name)

        for i in range(len(cameras)):
            var = cameras[i]

            #lambda is needed to make a clickable button - otherwise
            #the click event is triggered on startup
            #further we need to specify the default values of the
            #function and pass them as otherwise only name and path
            #of the last camera are saved
            ui.Button(var[0], clicked_fn=lambda name=var[0],
                      path=var[1]: on_click(name, path))

```

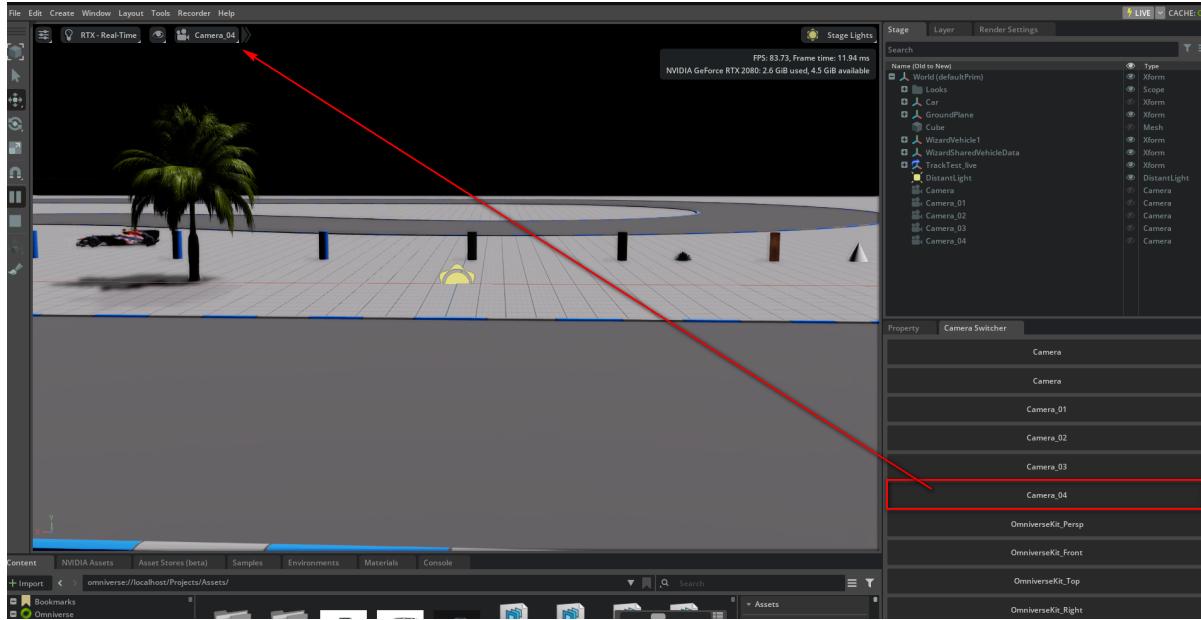


Figure 10: Example of how the camera switching application looks. Clicking its buttons changes the viewport to the clicked camera.

## 2.2 Co-simulation with Unreal Engine

A goal of this project was to test the co-simulation capabilities between Omniverse and Unreal Engine. Thus, we try to establish a co-simulation that enables concurrent animation playback in both Omniverse and Unreal Engine. To achieve this, we explore the utilization of a Boolean USD Property as a means of controlling the animation start and stop functionalities. However, implementing this approach is not straightforward due to the lack of direct exposure of raw USD Properties within Unreal Engine. Therefore, modifications to the Omniverse Plugin in Unreal Engine are necessary to enable the retrieval and manipulation of USD Properties. Specifically, we focus on expanding the *OmniverseStageActor* Class with additional UFunctions, which can then be utilized in Unreal Engine Blueprints to interact with these properties.

To enable the desired functionalities, it is necessary to have C++ support in our project. If a Blueprint project was created initially, it is necessary to either create a new C++ project or incorporate C++ into the existing project. We opted to create a new *Simulation Blank* C++ project.

Once the new project is created, the first step is to create a Plugins folder in the root directory of the Unreal Engine project. Subsequently, we navigate to the Plugins folder within the Unreal Engine installation directory and copy both the *MDL* and *Omniverse* Plugin folders into our project's local Plugins directory. On the used system, the path to these plugins is *C:\Program Files\Epic Games\UE\_5.1\Engine\Plugins\Marketplace\NVIDIA*. The *MDL* Plugin is included alongside the *Omniverse Plugin*, as the latter is dependent on the former.

With the plugins successfully copied to the local Plugins folder, we proceed to modify the *OmniverseStageActor* Class. To accomplish this, we open the *OmniverseStageActor.h* (located at *Omniverse/Source/OmniverseUSD/Public*) and *OmniverseStageActor.cpp* (located at *Omniverse/Source/OmniverseUSD/Private*) files in a suitable editor. Within the *OmniverseStageActor.h* file, we define several UFunction headers for obtaining and setting Boolean, float, and double values (in the examples shown only the Boolean and float functions are necessary), as shown in Figure 11. In the *OmniverseStageActor.cpp* file, we implement the UFunctions specified in the header file. This is shown in Figures 12 and 13. Furthermore, we need to include the *Blueprintable* keyword in the *UCLASS* definition within the *OmniverseStageActor.h* file. This addition is essential as it enables the creation of a Blueprint Class derived from the *OmniverseStageActor* Class. It is important to note, that the UFunctions provided are modified versions of the functions provided in this tutorial.

```
C:\OmniverseStageActor.h ● C:\OmniverseStageActor.cpp
OmniVerse > Source > OmniVerseUSD > Public > C:\OmniverseStageActor.h
33 }
34 }
35 UCLASS(Blueprintable, ClassGroup = OmniVerse, HideCategories = (Transform, Actor, LOD, Cooking), config=Engine, defaultConfig)
36 class OMNIVERSEUSD_API OMNIVERSEStageActor : public AInfo
37 {
40     GENERATED_BODY()
41
42     protected: // Blueprint-accessible functions & properties
43         //> Events to load ...
44
45     UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "OmniVerse")
46     TObjectPtr<UDOMAINUSD> USD;
47
48     /* Default prim of USD stage. */
49     UPROPERTY(EditAnywhere, BlueprintReadonly, Category = "OmniVerse", meta = (DisplayName = "Default Prim"))
50     TObjectPtr<AActor> USODefaultPrim;
51
52     /* Automatically attach new created prim under default prim. */
53     UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "OmniVerse")
54     bool bAutoAttach = true;
55
56     UFUNCTION(BlueprintCallable, Category = "OmniVerse (E)")
57     void GetPrimBoolAttribute(const FString PrimPath, const FString& AttributeName, bool& OutSuccess, bool& OutValue);
58
59     UFUNCTION(BlueprintCallable, Category = "OmniVerse (E)")
60     void SetPrimBoolAttribute(const FString PrimPath, const FString& AttributeName, bool& OutSuccess, bool InValue);
61
62     UFUNCTION(BlueprintCallable, Category = "OmniVerse (E)")
63     void GetPrimFloatAttribute(const FString PrimPath, const FString& AttributeName, bool& OutSuccess, float& OutValue);
64
65     UFUNCTION(BlueprintCallable, Category = "OmniVerse (E)")
66     void SetPrimFloatAttribute(const FString PrimPath, const FString& AttributeName, bool& OutSuccess, float InValue);
67
68     UFUNCTION(BlueprintCallable, Category = "OmniVerse (E)")
69     void GetPrimDoubleAttribute(const FString PrimPath, const FString& AttributeName, bool& OutSuccess, double& OutValue);
70
71     UFUNCTION(BlueprintCallable, Category = "OmniVerse (E)")
72     void SetPrimDoubleAttribute(const FString PrimPath, const FString& AttributeName, bool& OutSuccess, double InValue);
73
74     UFUNCTION(BlueprintCallable, Category = "OmniVerse")
75     FString ExportUSDToString();
76
77     UFUNCTION(BlueprintCallable, Category = "OmniVerse")
78     void SetUSD_UParam(DisplayName = "USD")class UOmniVerseUSD* InUSD, bool LoadScene = true);
79
80     UFUNCTION(BlueprintCallable, Category = "OmniVerse")
81     FMovingComponent* AddMovingReference(FIDParam<UDOMAINUSD> InUSD, const FTransform TransFrom, FTransform TransTo);
82 }
```

Figure 11: UFunction Headers in *OmniverseStageActor.h*.

```

File Edit Selection View Go Run Terminal Help
OPEN EDITORS ... C: OmniverseStageActor.h C: OmniverseStageActor.cpp
Omniverse > Source > Omniverse > Private > C: OmniverseStageActor.cpp
195     } LoadUSDScene();
196 }
197 }
198 }
199 }
200 void AOMniVerseStageActor::GetPrimBoolAttribute(const FString& PrimPath, const FString& AttributeName, bool& OutSuccess, bool& OutValue)
201 {
202     pxr::UsdAttribute Attr;
203     Attr = GetUSDStage()->GetPrimAtPath(ToUSDPath(*PrimPath)).GetAttribute(pxr::TfToken(std::string(TCHAR_TO_UTF8(*AttributeName))));
204     if (!Attr)
205     {
206         OutSuccess = false;
207         return;
208     }
209     OutValue = GetUSDValue<bool>(Attr);
210     OutSuccess = true;
211     return;
212 }
213
214 void AOMniVerseStageActor::SetPrimBoolAttribute(const FString& PrimPath, const FString& AttributeName, bool& OutSuccess, bool InValue)
215 {
216     pxr::UsdAttribute Attr;
217     Attr = GetUSDStage()->GetPrimAtPath(ToUSDPath(*PrimPath)).GetAttribute(pxr::TfToken(std::string(TCHAR_TO_UTF8(*AttributeName)));
218     if (!Attr)
219     {
220         OutSuccess = false;
221         return;
222     }
223     // Assume that the type is correct
224     Attr.Set(InValue);
225     OutSuccess = true;
226     return;
227 }
228
229 void AOMniVerseStageActor::GetPrimFloatAttribute(const FString& PrimPath, const FString& AttributeName, bool& OutSuccess, float& OutValue)
230 {
231     pxr::UsdAttribute Attr;
232     Attr = GetUSDStage()->GetPrimAtPath(ToUSDPath(*PrimPath)).GetAttribute(pxr::TfToken(std::string(TCHAR_TO_UTF8(*AttributeName)));
233     if (!Attr)
234     {
235         OutSuccess = false;
236         return;
237     }
238     OutValue = GetUSDValue<float>(Attr);
239     OutSuccess = true;
240     return;
241 }
242
243 void AOMniVerseStageActor::SetPrimFloatAttribute(const FString& PrimPath, const FString& AttributeName, bool& OutSuccess, float InValue)

```

 Figure 12: UFunction Definitions in *OmniverseStageActor.h*.

```

File Edit Selection View Go Run Terminal Help
OPEN EDITORS ... C: OmniverseStageActor.h C: OmniverseStageActor.cpp
Omniverse > Source > Omniverse > Private > C: OmniverseStageActor.cpp
243 void AOMniVerseStageActor::SetPrimFloatAttribute(const FString& PrimPath, const FString& AttributeName, bool& OutSuccess, float InValue)
244 {
245     pxr::UsdAttribute Attr;
246     Attr = GetUSDStage()->GetPrimAtPath(ToUSDPath(*PrimPath)).GetAttribute(pxr::TfToken(std::string(TCHAR_TO_UTF8(*AttributeName)));
247     if (!Attr)
248     {
249         OutSuccess = false;
250         return;
251     }
252     // Assume that the type is correct
253     Attr.Set(InValue);
254     OutSuccess = true;
255     return;
256 }
257
258 void AOMniVerseStageActor::GetPrimDoubleAttribute(const FString& PrimPath, const FString& AttributeName, bool& OutSuccess, double& OutValue)
259 {
260     pxr::UsdAttribute Attr;
261     Attr = GetUSDStage()->GetPrimAtPath(ToUSDPath(*PrimPath)).GetAttribute(pxr::TfToken(std::string(TCHAR_TO_UTF8(*AttributeName)));
262     if (!Attr)
263     {
264         OutSuccess = false;
265         return;
266     }
267     OutValue = GetUSDValue<double>(Attr);
268     OutSuccess = true;
269     return;
270 }
271
272 void AOMniVerseStageActor::SetPrimDoubleAttribute(const FString& PrimPath, const FString& AttributeName, bool& OutSuccess, double InValue)
273 {
274     pxr::UsdAttribute Attr;
275     Attr = GetUSDStage()->GetPrimAtPath(ToUSDPath(*PrimPath)).GetAttribute(pxr::TfToken(std::string(TCHAR_TO_UTF8(*AttributeName)));
276     if (!Attr)
277     {
278         OutSuccess = false;
279         return;
280     }
281     // Assume that the type is correct
282     Attr.Set(InValue);
283     OutSuccess = true;
284     return;
285 }
286
287 bool AOMniVerseStageActor::HasValidUSD() const
288 {
289     return USD != nullptr;
290 }

```

 Figure 13: UFunction Definitions in *OmniverseStageActor.h*.

Once the necessary UFunctions for retrieving and setting USD Boolean, float, and double values have been implemented, we proceed to open the Visual Studio Project by double-clicking the .snl file located in the project's root folder. After Visual Studio is opened we simply need to recompile the project. To accomplish this, we select *Development Editor* and *Win64* as the target platform and start the project build (*Build/Build/ProjectName*).

After the project has been rebuilt, we proceed to open it in Unreal Engine and create a Blueprint Class based on the modified *OmniverseStageActor* Class. To do so, make sure the Content Browser displays the Plugins folder since this is where the modified class resides (can be enabled via *Settings* in the Content Browser). Within the Plugins Folder, we find four sub-folders, with two dedicated to the

*MDL* Plugin and two dedicated to the *Omniverse* Plugin.

Navigating to *NVIDIA Omniverse C++ Classes/OmniverseUSD/Public* within the Plugins Folder, we locate the *OmniverseStageActor* class, which is the class we modified.

Right-click on the *OmniverseStageActor* class and create a Blueprint class based on it. Name the Class as you wish (or leave the default) and click *Create Blueprint Class*, which creates the Blueprint Class within the Content folder and opens it. By right-clicking within its *Event Graph*, we can access the Blueprint functions that we added to the *OmniverseStageActor*. These Blueprint functions enable us to read and modify USD Properties of the underlying USD stage. In the context of our specific objective, we aim to add a Boolean Value to the USD Stage. This Boolean Value will serve as a mechanism to control the concurrent playback and pausing of the animation within Omniverse. Before we get to this specific objective we want to showcase shortly how-to read-out USD Properties in Unreal Engine.

For this we first want to head over to Omniverse USD Composer and look for a USD Property we want to read-out in Unreal Engine. Here, we utilize the previously created map containing the drivable vehicle model, allowing us to explore various vehicle-related USD properties. By selecting the vehicle within the stage and navigating to its Raw USD Properties, we observe an abundance of properties that describe different aspects of the vehicle.

For our real-time read-out demonstration, we choose to focus on the *Steer* USD property. This property represents the current amount of steering applied to the car, ranging from values of up to 1 for left steering, as displayed in Figure 14, and down to -1 for right steering. To enable real-time read-out, we must generate a .live file, as previously outlined. Returning to Unreal Engine, we proceed to implement the functionality required to read the *Steer* property.

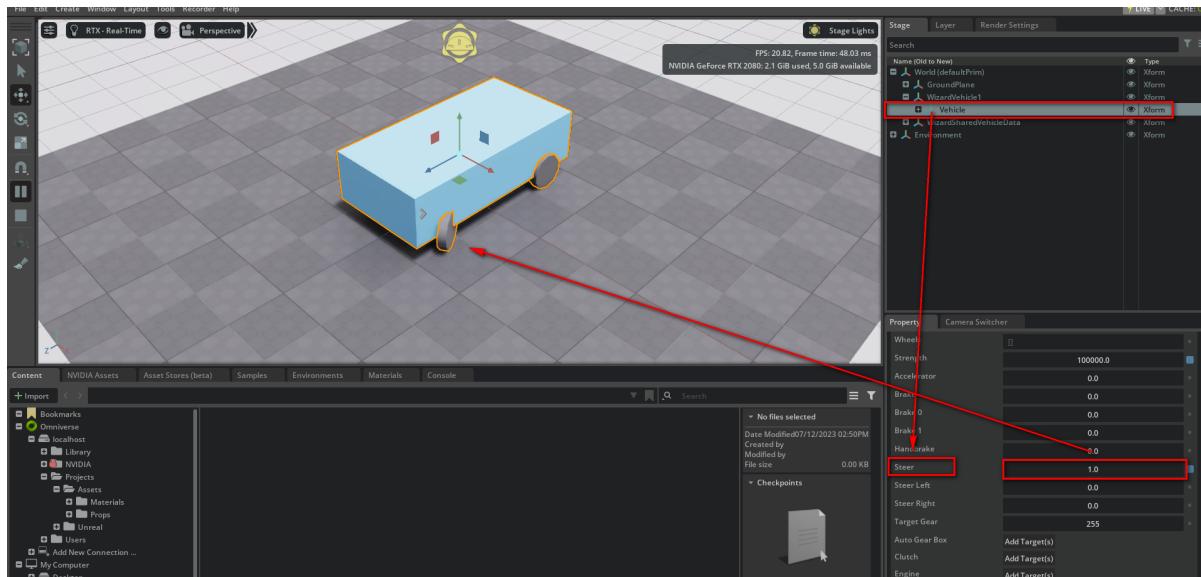


Figure 14: Example showcasing the *Steer* property.

Within the *Event Graph* of the *OmniverseStageActor* Blueprint Class we created, we first drag-out the execution pin from the *Event BeginPlay* node and search for the *Set Timer by Event* node, which allows us to repeatedly call a specific event at a defined interval. Enabling the desired polling behavior, we ensure the *Looping* property is activated. Next, we establish an event that is consecutively triggered within the loop. To achieve this, we drag-away from the *Event* property of the *Set Timer by Event* node and create a *Create Event* node. By selecting the *[Create a matching function]* option within the *Signature : ()* drop-down menu, an auto-generated function is created. It is advisable to rename this function to a name reflecting its purpose, such as *PrintFloatValue* in the case of the *Steer* float property.

Within the *PrintFloatValue* function, we drag-away the execution pin and call the previously specified *Get Prim Float Attribute* function. To correctly specify the *Prim Path* and *Attribute Name* parameters, we return to Omniverse and navigate to the *Steer* property. Right-click on the property and select *Copy Property Path*. In Unreal Engine, we paste the path into the *Prim Path* field and remove

everything following the period, which is then pasted into the *Attribute Name* field. It is important to delete the period from the *Prim Path* field.

To help with debugging we add a *Branch* node that allows for differentiation between successful attribute retrieval and cases where the attribute is not found. When the attribute is successfully retrieved, we utilize a *Print String* node to display the corresponding string value. Additionally, an *Append* node is introduced to concatenate the "Steer: " prefix with the float attribute obtained from the USD file. The resulting value is then provided as input to the *Print String* function. To prevent the display of multiple string values on the screen, the *Duration* parameter of the *Print String* nodes is set to 0.1 seconds, aligning with the interval at which the *PrintFloatValue* function will be called. The complete graph of the *PrintFloatValue* function is presented in Figure 15.

Returning to the Event Graph, we configure the *Time* attribute of the *Set Timer by Event* node to 0.11 seconds, ensuring that the *PrintFloatValue* function is called every 0.11 seconds. The slight difference to the *Print String* duration and the *Time* attribute ensures, that only one string is displayed at a time. The Blueprint Class, displayed in Figure 16, is then compiled, saved, and ready for testing.

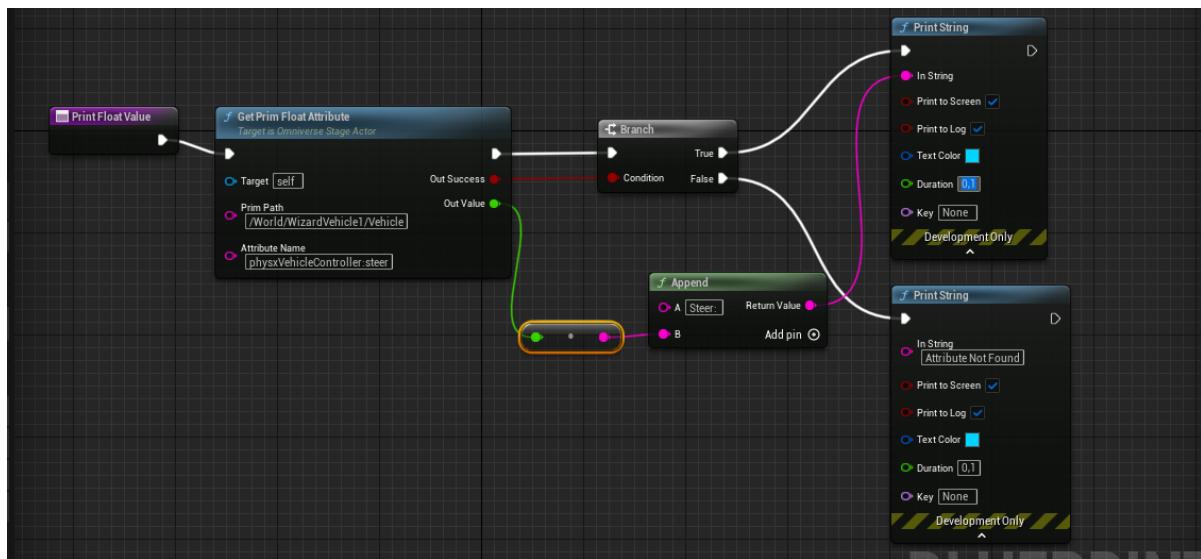


Figure 15: Graph of the *PrintFloatValue* Function.

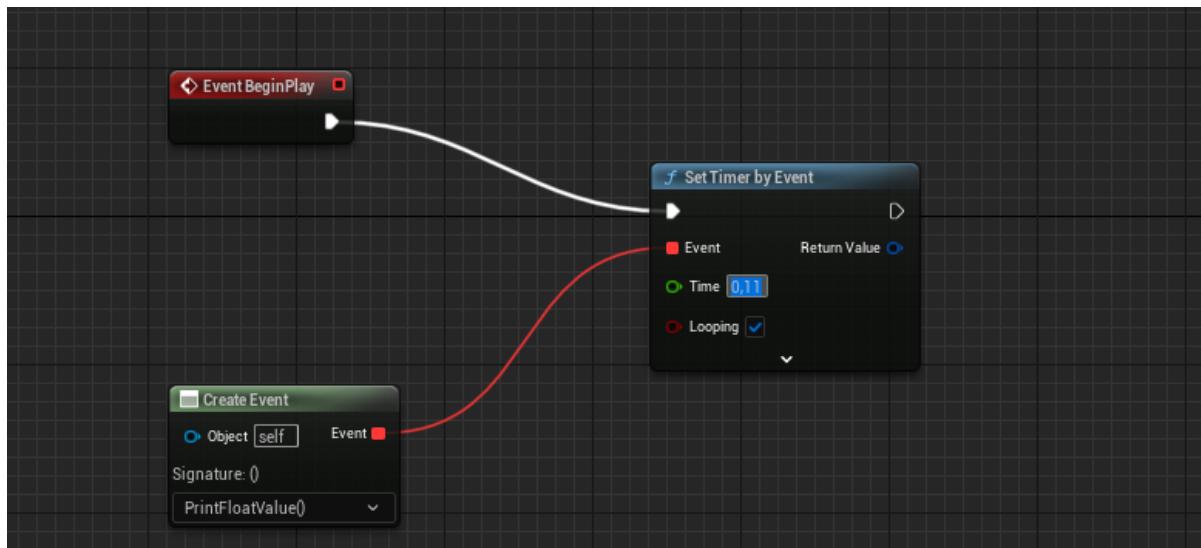


Figure 16: Event Graph of the *OmniverseStageActor* Blueprint Class.

To assess the functionality, we add the *OmniverseStageActor* Blueprint Class to the scene and attach the previously created .live file to the *USD* property. If the scene appears overly bright or white, navigate to the *GlobalPostProcessVolume* and select *Auto Exposure Histogram* within the *Exposure* settings. By playing the Unreal Engine Level, we should observe the output "Steer: 0.0." Playing the Omniverse level and steering the vehicle leads to the steer value being read-out in real-time within Unreal Engine. As can be seen in Figure 17, if we steer left in Omniverse we get an output of "Steer: 1.0" in Unreal Engine, validating that the read-out of float properties works.

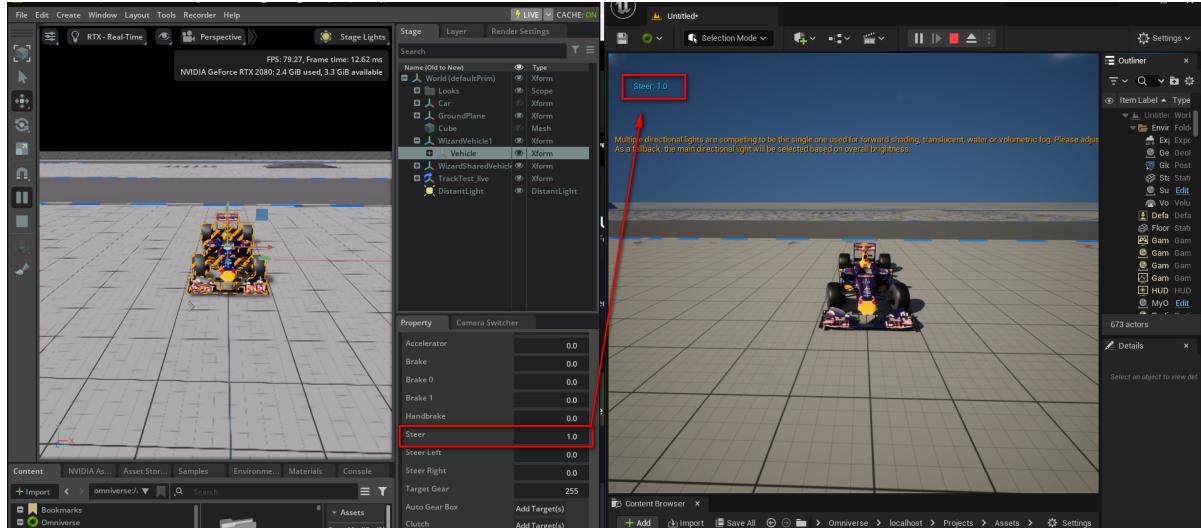


Figure 17: Live read-out of the *Steer* float property in Unreal Engine.

However, it is worth noting that performance issues may arise when the scene contains a large number of vertices, as seemingly Unreal Engine requires updating the entire scene whenever a single value changes. These performance considerations and limitations will be discussed in greater detail in a later section.

### 2.3 Concurrent Animation Playback in Omniverse and Unreal Engine

Having explored the process of reading USD properties in Unreal Engine, the final objective of this project is to establish concurrent animation control in both Omniverse and Unreal Engine. This will be achieved by introducing a Boolean property to the USD file and utilizing its value to initiate or pause the animation. To enable this functionality, we will develop an extension within Omniverse, which can set this Boolean property in the beginning i.e. start the animation. On the Unreal Engine side, we will implement a polling mechanism to play or pause the animation based on the Boolean value.

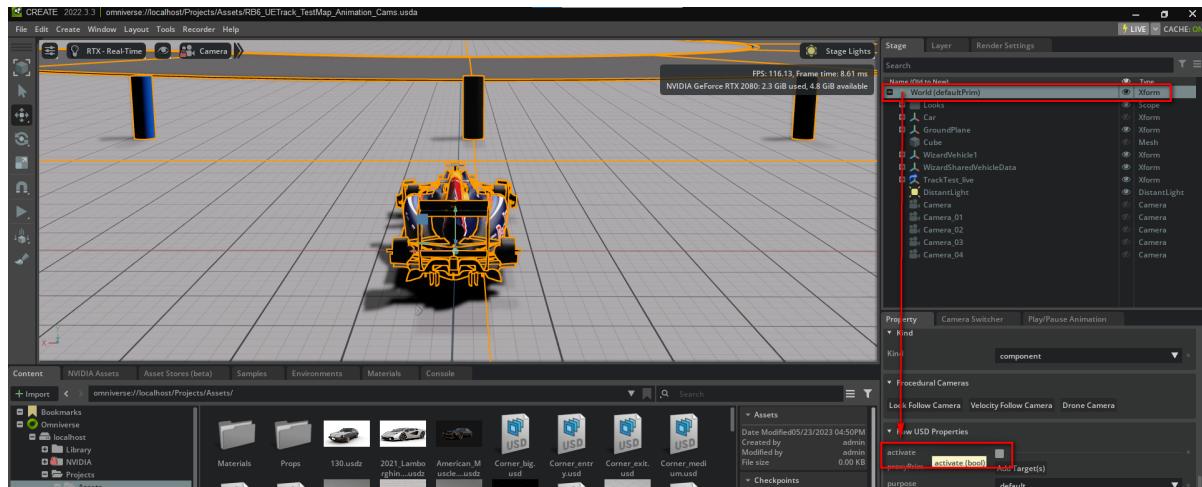
The initial step is to add the Boolean property to the USD file. Within the Omniverse Content Browser, the file containing the previously created animation is selected and right-clicked, followed by selecting *Edit....*. In the USD file, find the *World* primitive and add a Boolean variable named *activate* and initialized to 0 to the top of the property list, like shown in Figure 18. The file is then saved, and the editing process is finished by right-clicking on the file and selecting *Finish editing*.

By accessing the Raw USD Properties of the file, while the *World* Prim is selected in the Stage, the newly added Boolean property should be visible, as presented in Figure 19.

```

64    )
65
66    def Xform "World" (
67        kind = "component"
68    )
69    {
70
71        bool activate = 0
72
73        float3 xformOp:rotateXYZ = (0, 0, 0)
74        float3 xformOp:scale = (1, 1, 1)
75        double3 xformOp:translate = (0, 0, 0)
76        uniform token[] xformOpOrder = ["xformOp:translate", "xformOp:rotateXYZ", "xformOp:scale"]
77
78        def Scope "Looks"
79        {
80            def Material "info"

```

Figure 18: Addition of the *activate* Boolean property to the *World* prim.Figure 19: Showcase of the *activate* Boolean property as part of the *World* prim.

Next, an extension is developed to control the animation based on the Boolean property. Within Omniverse open the Extensions window (*Window/Extensions*) and create a new extension template project by clicking the green plus icon. Similar to the previous extension creation process, a folder location and project name, along with an extension ID, are specified. This generates the extension project and opens Visual Studio Code where we open the *extension.py* file. First, the current stage and its associated timeline are retrieved, as the timeline provides control over starting and stopping the animation. The desired play and pause functionality should be achieved by clicking a button, which internally changes the underlying USD Property, while also playing/pausing the animation.

The stage is retrieved using the *omni.usd.get\_context().get\_stage()* function, and the timeline is accessed through the *get\_timeline\_interface()* function from the imported *omni.timeline* library. The timeline interface provides functions for playing and pausing the animation. To retrieve the Boolean property, the *GetPrimAtPath('/World').GetAttribute('activate')* function is used, while utilizing the *Get()* and *Set()* functions to access and modify its value, respectively. All the needed imports as well as the complete *on\_startup* function is provided below.

After completing the modifications in the *extension.py* file, it is saved, and we go back to Omniverse USD Composer, where the extension with its two buttons should be visible. By clicking the *Play* button, the animation starts, and when inspecting the Raw USD Properties of the World prim, the *activate* Boolean property is now set to true. Pressing the *Pause* button halts the animation, and the *activate* property reverts to false, confirming the successful play and pause functionality of the extension.

```

import omni.ext
import omni.ui as ui
import omni.timeline

```

```

def on_startup(self, ext_id):
    print("[strange.usd.timeline.player] strange usd timeline player startup")

    self._count = 0

    self._window = ui.Window("Play/Pause Animation", width=300, height=300)
    with self._window.frame:
        with ui.VStack():
            # get the stage:
            stage = omni.usd.get_context().get_stage()

            if stage is not None:
                # get a reference to the timeline
                timeline = omni.timeline.get_timeline_interface()

                with ui.VStack():
                    def on_click():
                        # get activate Boolean Attribute reference
                        activate = stage.GetPrimAtPath('/World')
                            .GetAttribute('activate')
                        if activate.Get():
                            print("activate: True")
                        else:
                            print("activate: False")

                        if activate.Get():
                            if timeline.is_playing():
                                timeline.pause()
                                activate.Set(False)
                            else:
                                if not timeline.is_playing():
                                    timeline.play()
                                    activate.Set(True)

                    ui.Button('Play', clicked_fn=on_click)
                    ui.Button('Pause', clicked_fn=on_click)

```

With the Omniverse extension functioning as intended, we focus on implementing the counterpart functionality within Unreal Engine. Open the project that includes the previously created *OmniverseStageActor* Blueprint Class. Here we want to have a Play/Pause Button to play and pause the animation, by utilizing a UI Widget. To generate the UI Widget, go to the Content Browser, right-click and then under *User Interface* create a *Widget Blueprint* and when prompted to pick the root widget select the *User Widget*. Name the UI widget accordingly (e.g. *BtnPlayPause*). Upon opening the widget, the Play/Pause button is constructed by first dragging a *Button* from the *Common* section into the Designer view. Within the Hierarchy, right-click the button, and wrap the button with a *Size Box*. After this we place a *Text Block* onto the button and change its Text to “Play/Pause”. To ensure appropriate button placement and avoid occupying the entire screen, horizontal left alignment and vertical center alignment are selected for the button. Adjustments such as resizing the *Size Box* or applying padding can be made to achieve the desired button position.

After compiling and saving the button, we switch from the *Designer* to the *Graph* and in the *Event* Section we add the *OnClicked* Event to the button by clicking the plus icon. After this, an *EventDispatcher* is created by clicking the small plus icon in the *EVENT DISPATCHER* section, and name it ”Play-Pause”. After creating the *Event Dispatcher*, drag it into the graph and select *Call*, which should create a *Call Play Pause* Node. Connect this node to the *On Clicked* Event, compile and save the Button Widget and close it. For reference the complete graph of the ”PlayPause” button is presented in Figure 20.

Within the Event Graph of the *OmniverseStageActor* Blueprint Class, first create a *Set Timer by Event* node by dragging-away the execution pin from the *Event BeginPlay* node. Similarly to the previous implementation for printing the *Steer* property, a *Create Event* node is created by dragging away from the *Event* pin. In the *Create Event* node, we select the option to create a matching function from the ”*Select Function...* drop-down. The function is renamed to ”TriggerAnimation” to signify its purpose of playing or pausing the animation based on the Boolean property value. Additionally, this function will also print the current value of the Boolean property for debugging purposes. In the ”TriggerAnimation”

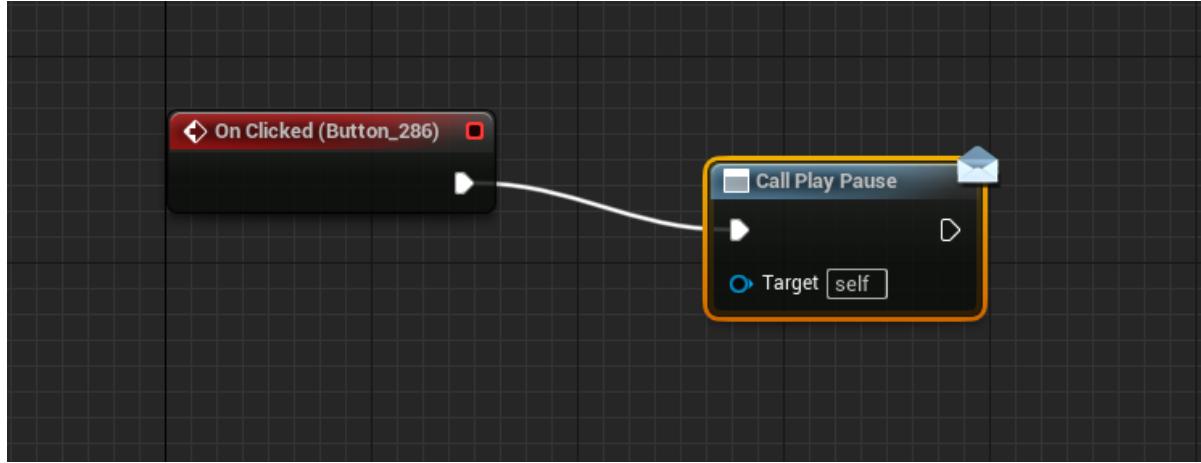


Figure 20: Graph of the "PlayPause" button.

function we first create a *Get Prim Bool Attribute* Node, where we set the *Prim Path* to */World* (the path to our *activate* USD Property) and the *Attribute Name* to *activate*. Then based on the success of the *Get Prim Bool Attribute* Function we either print the value of the Boolean property or we print, that the attribute could not be found.

Next, a *Get Actor of Class* node is created and connected to the *Print String* Node printing the property, where the *Actor Class* is set to *LevelSequenceActor*. The intention is to poll the Boolean property consecutively and either play or pause the animation accordingly. From the *Get Actor Of Class* node, a *Play (Sequence Player)* node is created, which also generates a *Sequence Player* node. Similarly, a *Pause (Sequence Player)* node is created by dragging away a line from the *Sequence Player* node. To determine the animation state (play or pause), a *Branch* node is connected to the *Get Actor Of Class* node. The True Branch is connected to the *Play* node, while the False Branch is connected to the *Pause* node. The output value of the *Get Prim Bool Attribute* node is used as the condition for the *Branch* node, determining the play or pause state. For reference, Figure 21 contains the complete graph for the "TriggerAnimation" function.

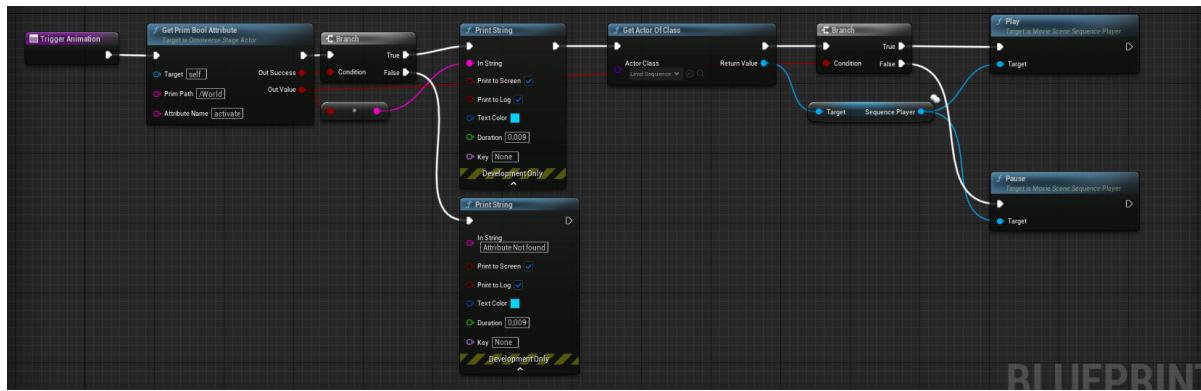


Figure 21: Complete graph for the "TriggerAnimation" function.

Returning to the *Event Graph*, a *Create Widget* node is created and connected to the *Set Timer by Event* node. Within the *Create Widget* node, go to the *Select Class* drop-down and select the previously created Widget Blueprint (*BtnPlayPause*). The Widget is then added to the viewport using an *Add to Viewport* node. The output value of the *Create Widget* node is connected to a *Bind Event to Play Pause* node, which establishes an event for the button click. From the *Event* pin of the *Bind Event to PlayPause* node, a *Create Event* node is created to generate a matching function named "SetBoolProperty". In the "SetBoolProperty" function, first a *Get Prim Bool Attribute* node is initialized with the *Prim Path* set to */World* and the *Attribute Name* set to *activate* to retrieve the *activate* USD property. Then, a *Branch* node is added, using the success output as the condition. The desired logic is to invert the Boolean property value when the button is pressed, toggling between true (playing) and false (paused).

This means we simply have to invert the Boolean properties value whenever the button is pressed. Therefore, a *Set Prim Bool Attribute* node is connected to the True Branch, and the inverted output of the *Get Prim Bool Attribute* node is used as input. The Boolean value inversion is accomplished by using the *NOT Boolean* node. Lastly, the *Prim Path* and *Attribute Name* in the *Set Prim Bool Attribute* node are set to the same values used in the *Get Prim Bool Attribute* node and then the Blueprint Class is compiled and saved. Figures 22 and 23 show the graph for the "SetBoolProperty" function and the Event Graph of the *OmniverseStageActor* Blueprint Class.

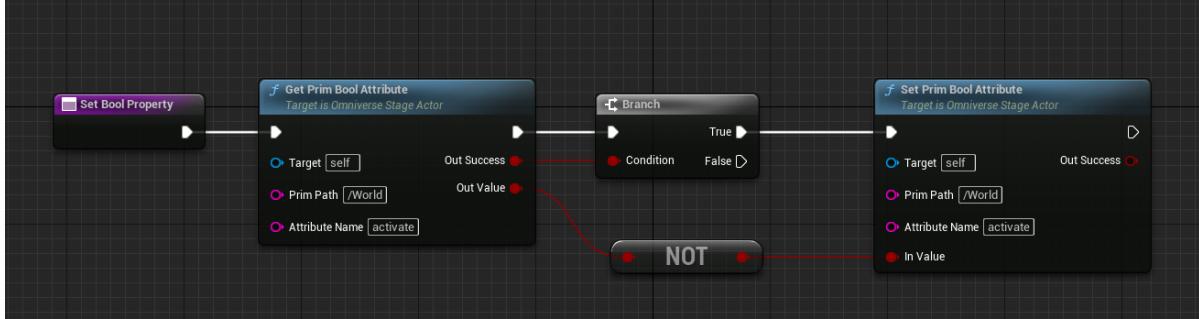


Figure 22: Complete graph for the "SetBoolProperty" function.

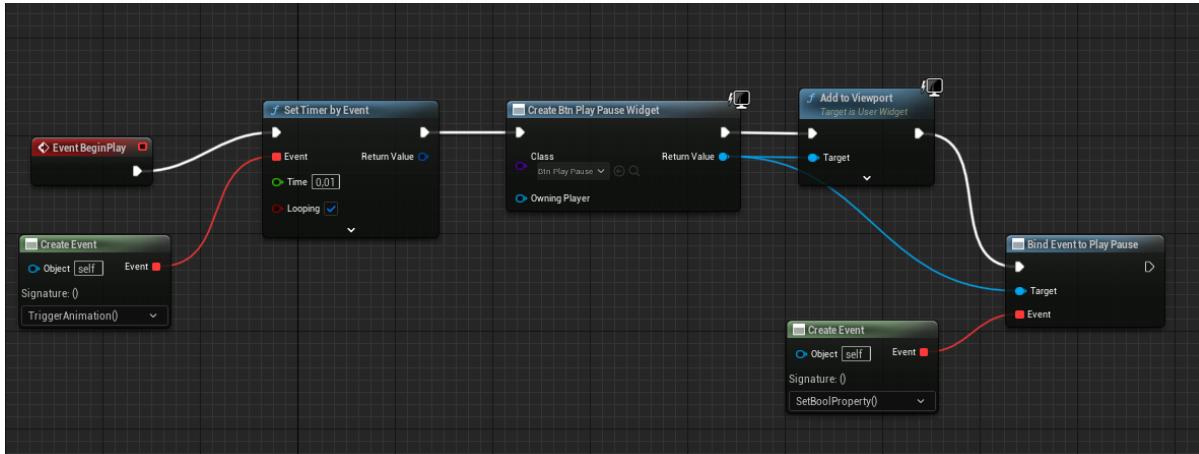


Figure 23: Complete Event Graph of the *OmniverseStageActor* Blueprint Class enabling the playing/-pausing of the animation based on the *activate* Boolean Property.

To test the concurrent animation control, we create a .live file from the USD file containing the animation previously modified to include the *activate* Boolean property in the World Prim. Also make sure your Omniverse Extension for the playing and pausing of the animation is loaded and the timeline is enabled. In case the extension is not loaded or not visible it can be loaded or reloaded from the Extensions menu. The *OmniverseStageActor* Blueprint Class is then added to the Scene in Unreal Engine, and the newly created .live file is dragged onto the USD Property of the *OmniverseStageActor*. If the screen appears overly bright, the *Exposure* in the *GlobalPostProcessVolumeshould* be set to *Auto Exposure Histogram*. Now click the Play Button in Omniverse and then pause the scene, followed by resetting the animation to 0, by dragging the blue pin in the animation timeline to its beginning. Further make sure the *activate* Property is set to false, as this should be the initial value, to make sure the Unreal Engine animation is paused at the beginning. In Unreal Engine, upon playing the Level, the Play/Pause Button should be displayed, and the animation should be paused (with "false" printed as the value of the *activate* property). To start the concurrent simulation, click the Play Button in the previously created Omniverse extension, which sets the *activate* property to true, resulting in the animation playing concurrently in both applications.

## 2.4 Development of Standalone Application

In this section, we address the development of a standalone application for our prototype. The objective is to create an executable that allows users to open and view the prototype independently. The application will load the scene with the animated car and provide functionality to switch between cameras and play/pause the animation, enhancing interactivity. Note that Omniverse installation is still required, as the application relies on Omniverse Kit as the core application runtime.

To accomplish the standalone application, we utilize the Omniverse Kit App Template, which can be cloned or downloaded from GitHub. After cloning the template, open it in Visual Studio Code and in the root folder under *source/extensions* you see *my\_name.my\_app.setup* and *my\_name.my\_app.window*, which are the Setup and Window Extensions. The Setup Extension is where we can set the layout of the window, configure the menu and most importantly, where we load our USD scene. The Window Extension creates a Window equal to the Window created by the Extensions in Omniverse USD Composer. We aim to recreate the Extension that enables camera switching. First, we focus on loading the USD Scene, thus opening *setup.py*.

In *setup.py*, we want to load the USD scene containing our animation. Before loading the scene, we switch to Omniverse USD Composer and open the scene with the animation. To ensure the scene includes all necessary data, we convert Payloads to References by right-clicking each prim with a blue arrow and selecting *Convert Payloads to References*. The arrow color should change to orange, indicating successful conversion. After converting all payloads to references, we save the file on our disk. After saving the USD file we go back to the *setup.py* and we add the call to open the stage at the bottom of the “on\_startup” function. The call to open the stage is as follows:

```
omni.usd.get_context().open_stage('C:\User\user\Documents\USD_file.usd')
```

If you want to package and distribute the application you should move the USD file into the data folder under *source/extensions/my\_name.my\_app.setup/data* and open the stage via a relative path. To test if the stage is successfully opened, we open a new terminal and navigate to the root folder. Calling *\build.bat* compiles the application, which should result in a “Build Success” message. If the build is unsuccessful, rerun the build script or refer to the documentation or Omniverse forum for assistance. Next, we launch the application by executing *\\_build\windows-x86\_64\release\my\_name.my\_app.viewport.bat* in the terminal. It is important to add *viewport* to indicate that we want to create a viewport app. An editor app can also be created, but for our purposes, we restrict the user from modifying the scene, thus utilizing a viewport app. For more information on the difference between the two visit the documentation.

After launching the application, a window should open with the viewport displaying the loaded USD stage. The Window base app should also be visible on the right side of the application. If this is the case, the application can be closed.

We then open *extension.py* and *window.py* in the Window Extension (*my\_name.my\_app.window*). In *window.py*, we create an extension that enables camera switching and play/pause functionality for the animation. The code is a refactoring of the two previously created extensions and is provided below. Initially, we retrieve the current stage, timeline, and all cameras in the scene. Then, buttons are created for each camera, and their click events are handled to switch to the corresponding camera (while ensuring that all cameras are hidden in the scene). For play/pause functionality, a simple function is implemented to toggle the timeline state between paused and playing. The full code of the *MyWindow* application is presented at the end of this section.

In the *extension.py* we only need to make minor changes, as we here only set the initial value of *show\_window* to false (last Boolean in line 17 and the Boolean in line 18), as we don’t want to display the extension immediately. The cameras in the scene can only be found once the stage is fully loaded, so the user should open the window after the stage is fully loaded to ensure full functionality.

When you open the application now, only the stage with the loaded scene should be visible. To open the extension window, navigate to *Window* and select *MyWindow*. This should open the extension window and you can drag the window to the side and interact with the scene via the provided buttons.

As you can see the toolbar offers various options like *File*, which allows users to open a different scene. However, this functionality is not required and thus we disable it. In *setup.py*, the *\_setup\_menu* function can be modified to comment out the lines *self.\_file\_open* and *self.\_help\_menu*. Additional menus can

also be disabled in the `my_name.my_app.viewport.kit` file under `source/apps`. Below we show, which lines can be commented out, as their functionalities are not needed for this application. Important to note is, that we comment-out the viewport bundle, while un-commenting the viewport window. This removes the different viewport options in the left upper corner of the scene and only provides the window, in which the scene is displayed.

```
# File and Edit Menu -> can be commented out
#"omni.kit.menu.file" = {}
#"omni.kit.menu.edit" = {}

# Comment out the viewport bundle
#"omni.kit.viewport.bundle" = {}

# Uncomment the viewport window
"omni.kit.viewport.window" = {}

# Toolbar and Extensions -> can be commented out
#"omni.kit.window.toolbar" = {}
#"omni.kit.window.extensions" = {}
```

At this point, all necessary adjustments have been made, and the application can be opened to evaluate the final result. As a reference point, Figure 24 displays how the completed application looks.

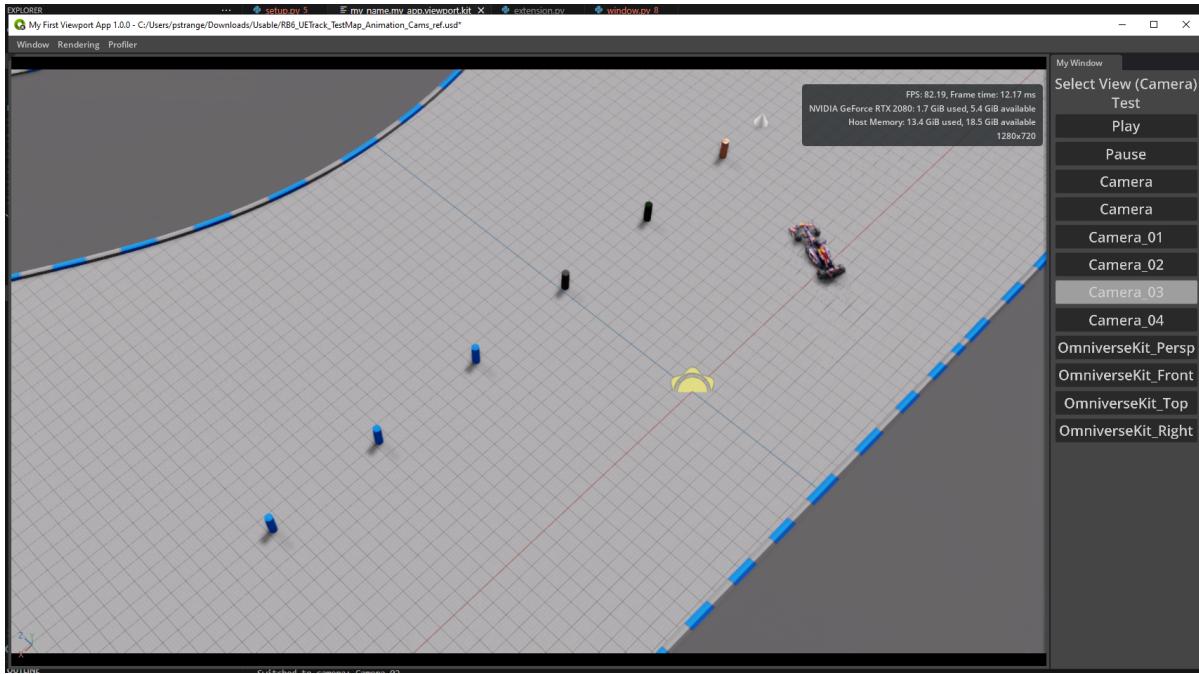


Figure 24: Showcase of the completed application with the `MyWindow` Extension enabled and the animation playing.

To package the application, simply execute the `package.bat` script in the root folder. This creates a zip file in the `_build/packages` directory. The zip file can now be distributed, and users should extract it before running `link_app.bat`. The `link_app.bat` script establishes a link between the application and one of the Omniverse Applications (typically Omniverse USD Composer, formerly Omniverse Create), which is necessary to run the application. After linking, the application can be launched by executing `my_name.my_app.viewport.bat`.

This concludes the development section and we now shift our focus to evaluating the performance of Omniverse and the Connector between Omniverse and Unreal Engine.

```
# Full code for the MyWindow Extension

import omni.usd
from pxr import UsdGeom
import omni.timeline

class MyWindow(ui.Window):
    """My Window"""

    title = "My Window"

    def __init__(self, **kwargs):
        super().__init__(MyWindow.title, **kwargs)

        viewportWindow = vp_utils.get_active_viewport_window()

        with self.frame:
            with ui.VStack():
                ui.Label("Select View (Camera)", height=0,
                        style={"font_size": 24, "alignment": ui.Alignment.CENTER})
                stage = omni.usd.get_context().get_stage()

                if stage is not None:
                    timeline = omni.timeline.get_timeline_interface()
                    # find all cameras in the stage
                    ui.Label("Test", height=0, style={"font_size": 24,
                        "alignment": ui.Alignment.CENTER})
                    cameras = [[x.GetName(), x.GetPrimPath()] for x in stage.Traverse()
                               if x.IsA(UsdGeom.Camera)]
                    if len(cameras): # If cameras are found -> create button per camera
                        with ui.VStack(height=50, style={'font_size':24}):
                            def play_pause():
                                print("Play/Pause pressed")
                                if timeline.is_playing():
                                    print("Pause")
                                    timeline.pause()
                                else:
                                    print("Play")
                                    timeline.play()
                            def on_click(camera_name, camera_path):
                                viewportWindow.viewport_api.camera_path = camera_path
                                print('Switched to camera: ' + camera_name)

                            ui.Button('Play', clicked_fn=play_pause)
                            ui.Button('Pause', clicked_fn=play_pause)

                            for i in range(len(cameras)):
                                var = cameras[i]
                                UsdGeom.Imageable(stage.GetPrimAtPath(var[1]))
                                    .GetVisibilityAttr().Set('invisible')
                                #lambda is needed to make a clickable button -
                                #otherwise the click event is triggered on startup
                                #further we need to specify the default values of the
                                #function and pass them as otherwise only name and path
                                #of the last camera are saved
                                ui.Button(var[0], clicked_fn=lambda name=var[0],
                                          path=var[1]: on_click(name, path))
```

### 3 Performance and Limitations

In this section, we discuss the performance of Omniverse and the limitations encountered during the development of the prototype. We also explore potential strategies to improve performance.

#### 3.1 General Omniverse Performance and Optimization

During the testing of the animated car prototype, it is observable, that the RTX Renderer in Omniverse is GPU-intensive, resulting in lower FPS counts even with a good GPU. To improve performance, certain render settings can be adjusted. The obtained FPS values were measured while driving along the main straight of a test track, continuously steering the car, and capturing the lowest FPS values within a 2 FPS interval. It is important to note that FPS values are influenced by factors such as window size (resolution), scene complexity (vertices, materials), and the hardware used. Hence, it is recommended to test and customize the performance settings based on individual requirements.

##### 3.1.1 Lighting

Lighting significantly impacts performance, and the default Light Rigs perform worse compared to adding custom lights. Default Light Rigs are combinations of different lights, which contribute to their lower performance. Among the Light Rigs, the *Colored Lights* Light Rig (43-45 FPS) performed the best, followed by the *Default* Light Rig (40-42 FPS), *Sunny Sky* Light Rig (40-42 FPS), and *Grey Studio* Light Rig (38-40 FPS). When considering individually created lights, the *Distant Light* (49-51 FPS) showed the best performance, followed by the *Cylinder Light* (48-50 FPS), *Sphere Light* (46-48 FPS), *Rect Light* (45-47 FPS), *Disk Light* (45-47 FPS), and *Dome Light* (43-45 FPS). Switching from the *Default* Light Rig to a *Distant Light* improved FPS counts by approximately 9 FPS (from 40-42 FPS to 49-51 FPS).

##### 3.1.2 NVIDIA DLSS

The next major performance increase can be achieved by setting the Mode for *NVIDIA DLSS* to *Performance* (the default value here is *Quality*). DLSS stands for Deep Learning Super Sampling and with this the image is rendered in a lower resolution and then upscaled via an AI algorithm. Other DLSS settings did not yield significant performance improvements. Enabling Performance Mode for DLSS increased FPS counts by approximately 10 FPS (from 49 – 51 to 61 – 63).

##### 3.1.3 Reflections, Ambient Occlusion, and Shadows

Further FPS improvements can be achieved by disabling certain rendering features. Disabling reflections resulted in a significant performance boost (+22 FPS), taking FPS from 61-63 FPS to 83-85 FPS. Additionally, turning off ambient occlusion in the *Indirect Diffuse Lighting* section provided a minor improvement (+3 FPS) (83-85 FPS to 86-88 FPS). Finally, disabling shadows in the *Direct Lighting* section increased FPS counts by another 5 FPS, increasing FPS from 86-88 FPS to 91-93 FPS.

For more information on all rendering settings for the RTX Renderer visit the documentation.

#### 3.2 Unreal Engine Lag without Lighting

When working with an empty scene consisting only of a ground plane and the RB6 model placed on top of it, Unreal Engine (UE) experiences significant lag if no light is added to the scene. If the *Default* Light Rig is added to the scene Unreal Engine has no problem rendering the scene, but if the *Default* Light Rig (or any Light) is not added to the scene Unreal Engine struggles heavily. This behavior suggests that UE relies on its own shading calculations, when a light is present in the scene. Without a light, UE might need to retrieve lighting information from Omniverse, as in Omniverse selecting the *Default* Light applies the lighting, but does not add the Lights to the Stage.

To verify this behavior, we can establish a live connection between UE and Omniverse, by loading the USD stage (.live file) in UE. With only the *Default* Light Rig selected in Omniverse, severe lag can be observed when viewing the RB6 model in UE. Tracing the application revealed that a single render frame in the *RenderThread* took approximately 530ms, when looking at the F1 Car Model, with the *DeferredShadingSceneRenderer\_Render* function accounting for most of this time (520-530ms). This

is presented in Figure 25, where the period up to 3m 53s correlates with the above described scenario. We can see, that frame times in *RenderThread0* are somewhere between 530ms and 590ms.

If we keep looking at the RB6 Model in the scene and add a *Directional Light* into the UE scene, frame times decrease to around 10ms and the *DeferredShadingSceneRenderer\_Render* takes around 4ms. In Figure 25 the yellow flag at 3m 56s marks the addition of the *Directional Light* into the scene and the difference in frame times is clearly visible. This indicates that when UE can perform lighting calculations based on the *Directional Light* added to the scene, frame times decrease from 530ms to 10ms. To be clear, it does not matter if the Light is added in Omniverse or Unreal Engine, as long as Unreal Engine has a Light, which it can use for lighting calculations.

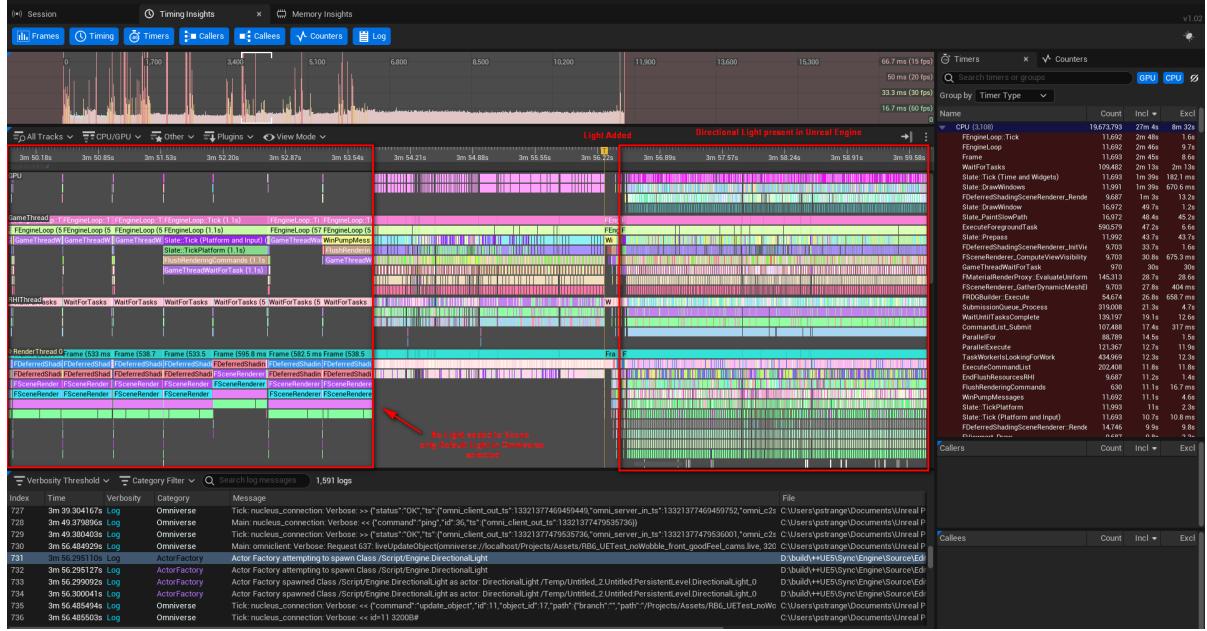


Figure 25: Timing Trace in Unreal Insights, showcasing the impact to the rendering performance, when adding a dedicated light to the scene.

Figure 26 shows such an example, where Figure 26a shows the scene with only the *Default Light Rig* selected in Omniverse, which leads to very laggy performance in Unreal Engine (17 FPS in this example). Figure 26b then shows the scene, after adding a *DirectionalLight* in Unreal Engine and we can see, performance increases substantially.

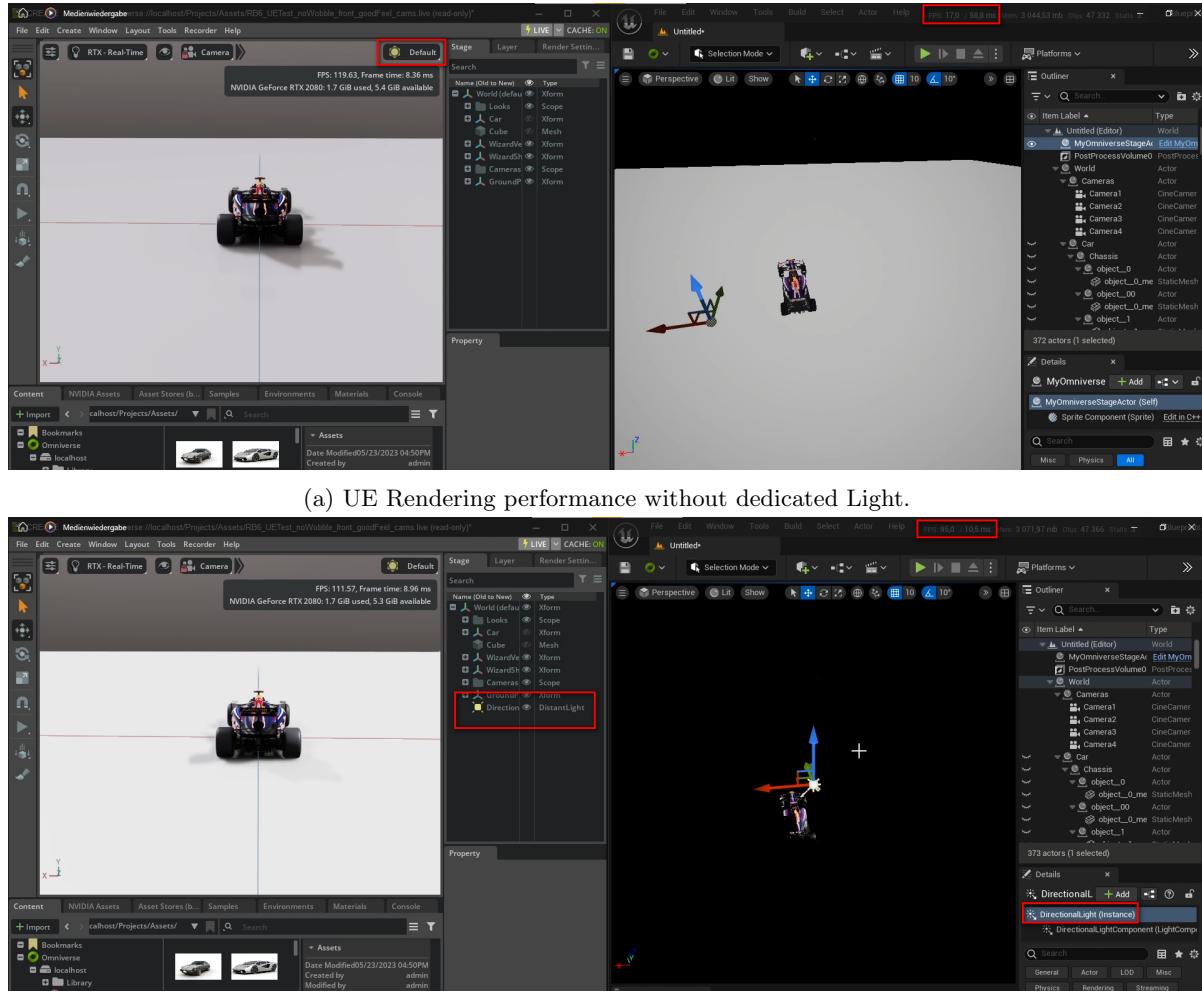


Figure 26: Showcase of the performance impact of adding a Light to the Unreal Engine Scene.

### 3.3 Limitations of Omniverse and Unreal Engine Connector

As touched on in one of the previous sections, the performance of Unreal Engine in the live co-simulation mode depends heavily on the number of vertices in the scene. If we have a scene with a high vertex count and we have a moving prim in this scene (driving a vehicle) Unreal Engine seemingly has to load the whole scene whenever a prim in it changes. As the moving primitive constantly updates (e.g., wheel rotation and position), Unreal Engine must continually load the updated USD file. At a certain vertex count threshold, Unreal Engine struggles to keep up with the USD file updates, resulting in lag. This is our assumption, as it makes no difference if the moving prim has a high or low vertex count i.e. it only depends on the total number of vertices in the scene. This also implies, that UE has to load the whole USD scene again and cannot only update the changed values in the USD file.

This assumption is supported, when we look at a trace in the Unreal Engine Insights, where at first only a simple vehicle (standard Vehicle Wizard vehicle with Wheels from the RB6 Model) was present in the scene. Driving the vehicle leads to no significant lag and allows for a smooth simulation in both Omniverse and Unreal Engine. Then, the RB6 car model (23.4K vertices) was added to the scene, introducing a high vertex count. Now, driving the simple vehicle model in Omniverse leads to significant lag in Unreal Engine. In Figure 27, the point where the RB6 model was added into the scene is marked with the yellow Flag and the time from 26m 45s to 27m 14s is where the simple vehicle was driven in Omniverse, without the RB6 Model being present. As can be seen in Figure 27 neither the Render nor the Game Thread have any performance issues. This changes at around 28m 10s, where again the simple vehicle was driven around in Omniverse, but with also the RB6 Model being present in the scene. It is visible, that the *GameThread* as well as the *RenderThread* are lacking work, while the GPU shows little

to no activity. This indicates that the CPU and GPU are not limiting factors, as they have little to do and are primarily waiting for tasks.

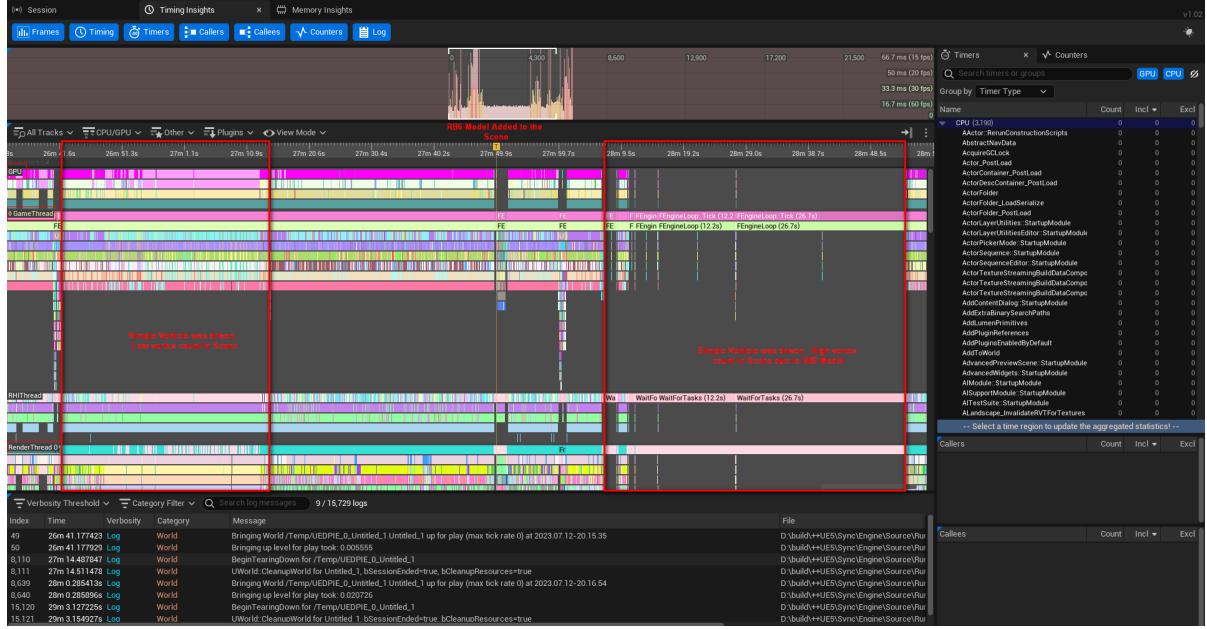


Figure 27: Timing Trace in Unreal Insights, showcasing the performance impact of a high vertex count in the scene.

Looking at the Memory Insights in Figure 28, we see, that at around 28m 0s, when the Level was started (driving of the vehicle started at around 28m 10s) Memory engagement looks similar to when we drove around with the simple vehicle without a high vertex count in the scene (26m 42s – 27m 14s). Thus, Memory engagement is at an equal or higher value (LLM Physics is higher), while GPU and CPU both are waiting for tasks.

Therefore, it is reasonable to assume that the laggy performance in Unreal Engine during live Physics Vehicle co-simulation with a high vertex count in the scene is caused by the way USD scene updates are handled by the connector. As the RB6 model in this example is a stationary non-moving model it is reasonable to assume that the connector updates the whole scene, not just the changed prims or their attributes. Thus, when the vertex count increases, the time required for these updates also increases, resulting in lag.

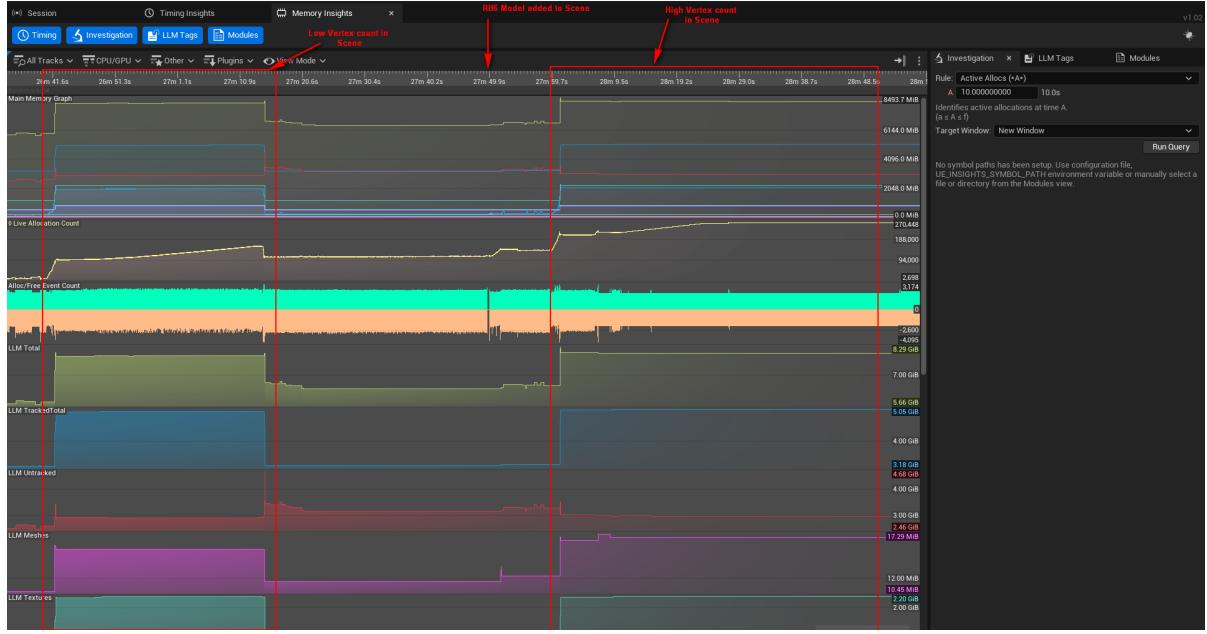


Figure 28: Memory Trace in Unreal Insights, showcasing the impact of a high vertex count to the performance.

There is however a way to circumvent this lag if one locks the stationary prims. This can be achieved by right-clicking the static prim in the Omniverse Stage and selecting *Locks* → *LockSelectedHierarchy*, as presented in Figure 29. In this example, as presented in Figure 29, the RB6 Model Prim is locked, which locks the parent prim, as well as all child primitives with all its properties. This ensures, that the stationary RB6 model does not have to be updated by the Connector, leading to no real lag in Unreal Engine, when reading out the *Steer* property. Thus we recommend locking stationary prims, when doing any live readouts of USD Properties in Unreal Engine.

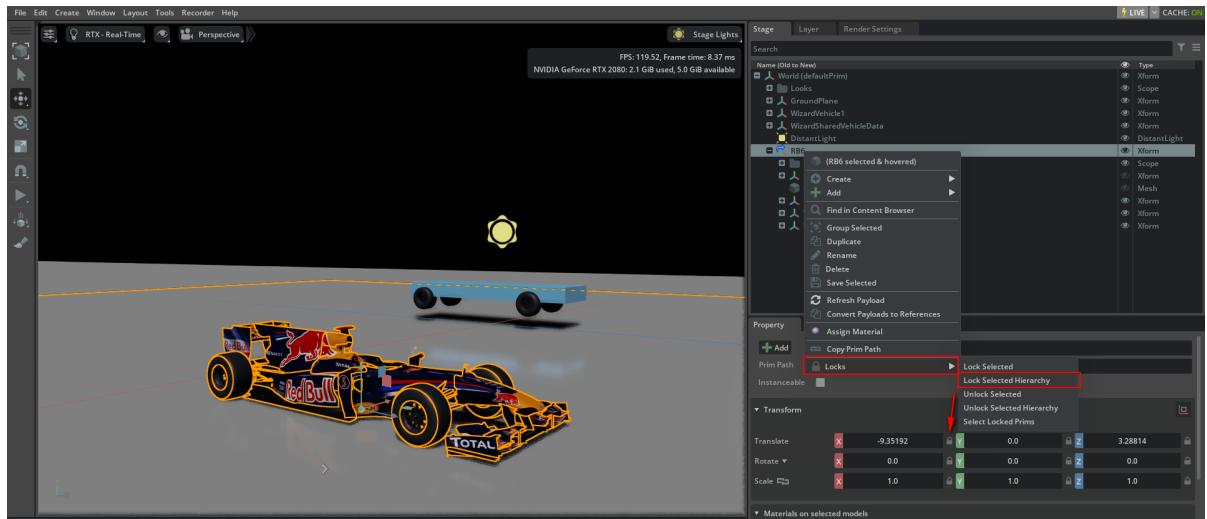


Figure 29: Selecting *Locks* → *LockSelectedHierarchy* locks all properties of a primitive, as well as all properties of its child primitives.

## 4 Technical Requirements

This section discusses the technical requirements for the project, including the software and hardware specifications necessary to run the implemented prototype. The project was developed using Omniverse USD Composer version 2022.3.3 (formerly known as Omniverse Create) and Unreal Engine 5.1.1. The used version of the Omniverse UE Connector was 201.1.244 and the used Local Nucleus Service version was 2022.4.2.

Omniverse Applications are based on the RTX Renderer, which is optimized for NVIDIA RTX GPUs. Therefore, an RTX GPU is required to run Omniverse Applications with the RTX Renderer. While it is possible to run Omniverse with a non-RTX GPU, it limits the available rendering options to the Pixar Storm Renderer, which seemingly lacks provided shaders. Omniverse provides different modes for the RTX Renderer, namely the RTX - Real-Time mode and the RTX - Interactive (Path Tracing) mode. We used the RTX - Real-Time mode, as the Interactive mode is even more performance hungry.

For the implementation of the prototype using Omniverse USD Composer, it is recommended to have at least an RTX 2080 GPU. The minimum GPU requirement is any RTX GPU with 6GB of VRAM.

In terms of CPU requirements, both the minimum and recommended specifications call for an Intel i7 or AMD Ryzen CPU. The minimum specifications require a minimum of 4 cores, while it is recommended to have 8 cores. Regarding RAM, Omniverse USD Composer requires a minimum of 16GB, with 32GB being recommended for optimal performance.

For our project, the system used was in line with the recommended system requirements. It consisted of an RTX 2080 GPU, an Intel i7 8700 CPU, and 32GB of RAM.

## 5 Future Work

In future projects, one potential area of exploration is extending the prototype to receive vehicle data through a web service. This enhancement would enable multiple racecars to be present in the same scene, with updates received via a web service. Such functionality could be achieved by developing a custom Python Omniverse Extension.