Chapter Outline

Distributed Simulation Architecture and Data Interfaces

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.4.1 Simulation Architecture
Textbook: Secs. 3.2.1.1–3.2.1.2

3.4.2 Spatial Reference Frames
Textbook: Sec. 3.2.2.2; added mounting example

3.4.3 Timing and Data Transfer
Textbook: Secs. 3.2.1.1, 3.2.1.3

3.4.4 Platform Interfaces and Sensors
Textbook: Secs. 3.2.1.3, 3.2.3

3.4.5 Integration and Troubleshooting
Textbook: Secs. 3.2.3, 3.3.1

3.4.1 Simulation Architecture: Overall Framework

Three Paths: Asset Import, Model Loading, and Runtime Data

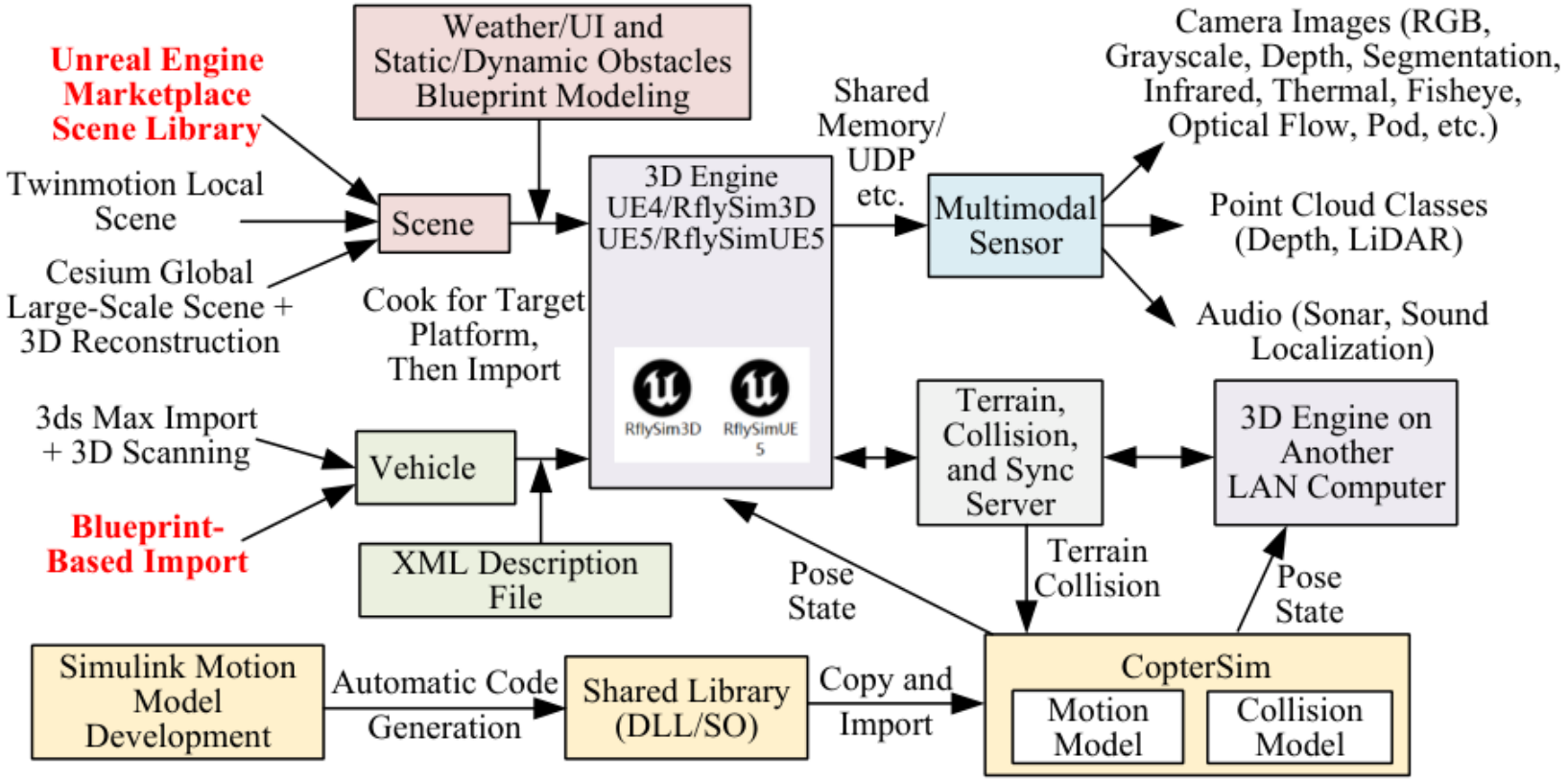


Figure 3.6 Overall Framework of RflySim 3D Simulation

3.4.1 Simulation Architecture: Asset and Model Import

Separate Paths for 3D Appearance and Vehicle Motion Models

- Scene maps: package and load native UE assets; load Cesium geographic data on demand at runtime.
- Vehicle appearance: Import a mesh or Blueprint with XML configuration. Select the registered style using the index formed from ClassID and DisplayOrder.
- Vehicle motion model: generate code in Simulink, compile it into a DLL/SO, and load it in CopterSim at runtime.
- DLL means a Windows dynamic-link library; SO means a Linux shared object. Prepare appearance and motion models separately.

Visual assets are deployed in advance; CopterSim loads the vehicle motion model at runtime.

3.4.1 Simulation Architecture: Runtime Responsibilities

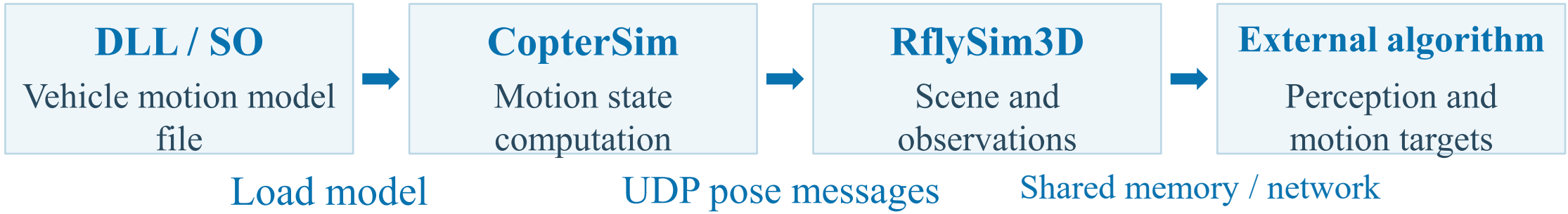
CopterSim Solves Motion; the 3D Application Displays and Senses

- CopterSim: load the vehicle motion model DLL, receive motor or control-surface commands, solve position, attitude, velocity, and angular velocity, and handle collision response.
- 3D engine: RflySim3D (UE4) or RflySimUE5 (UE5) renders scenes, detects geometric collisions, and generates sensor images; UE physics does not solve rigid-body motion.
- Synchronization server: return terrain heights and collision information from the 3D application to CopterSim so the motion solver can account for the scene.
- Local area network (LAN): 3D engines on different computers receive poses and return their own interaction data to the simulator.

CopterSim solves motion; the 3D application renders and detects collisions, with interaction data returned through the synchronization server.

3.4.1 Simulation Architecture: Loading and Data Transfer

DLL Loading and UDP Communication Connect Different Stages



- **Dynamic-link library (DLL):** CopterSim calls the compiled equations of motion within its process; SO is the corresponding shared-library format.
- **Software-in-the-loop (SITL; SIL in the textbook):** the algorithm sends motion targets to the PX4 autopilot;
  autopilot control outputs act on the vehicle motion model to produce the next pose.

The simulator loads the dynamics library; pose messages drive the 3D application.

3.4.1 Simulation Architecture: State and Sensor Data Links

Check State, Rendering, Sampling, and Reading Separately

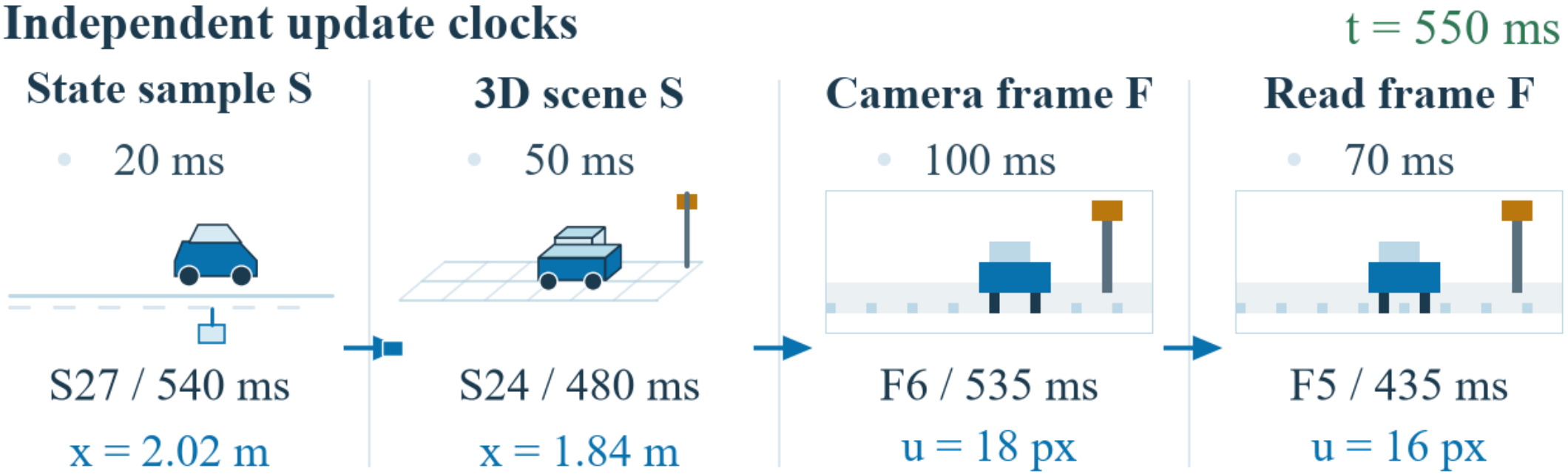


Figure: Four Stages from State to Image (Conceptual Illustration)

During integration, record the configuration, timestamps, and data shape at each stage.

3.4.1 Simulation Architecture: Main Benefits

Four Benefits of Distributed Simulation

- Lower computational demand: UE handles display and sensor rendering; CopterSim computes motion and collision response without using UE physics resources.
- Distributed computation: CopterSim runs independently, with each computer handling some vehicles; swarm scale can grow as computers are added.
- Customizable vehicle motion models: develop models in Simulink, refine parameters through system identification, compile to DLL or SO, and load them in CopterSim.
- Large-scale simulation: multiple computers render collaboratively and share the workload of many camera views (see the rendering workload example).

Separating motion computation from 3D rendering lets multiple computers share the swarm simulation workload.

3.4.1 Simulation Architecture: Rendering Workload Example

40 Streams of 1080P: Why Distribute Rendering?

- Given: 10 vehicles, each with 4 streams at 1920×1080, for 40 streams total; assume 30 Hz per stream.
- Pixels per second per stream: 1920×1080×30 ≈ 62.21 million; 40 streams yield about 2.49 billion pixels/s.
- Raw RGB transfer at 3 B per pixel is about 7.46 GB/s; this is an uncompressed estimate, not measured bandwidth.
- Rendering and transfer can both become bottlenecks; distribute cameras and measure each computer's GPU and network loads.

$$40 \times 1920 \times 1080 \times 30 \times 3 \approx 7.46 \times 10^{9}\ \mathrm{B/s}$$

40 streams of 1080P yield about 2.5 billion pixels/s; use measured GPU and network loads to decide whether to distribute rendering.

3.4.1 Simulation Architecture: Limitations and Remedies

Two Limitations of Distributed Simulation and Their Remedies

- Terrain and collision fidelity: CopterSim solves dynamics outside UE's PhysX/Chaos physics engine, making tire deformation and mud-induced slipping difficult to simulate; assess aerial missions against their ground-contact and latency requirements too.
- Remedy 1: for unmanned ground vehicles (UGVs) needing detailed ground interaction, use RflySim3D's native UE vehicle Blueprints: motion, control, and the 3D model stay inside UE.
- Motion synchronization: the 3D application and CopterSim run in separate processes or computers; states arrive at different times and with delay, so the display is less synchronized than in a single-process design.
- Remedy 2: timestamp data, interpolate by time, and use position, velocity, and acceleration for motion prediction and synchronization filtering (examples in Unit 3.4.3).

Moving physics computation out of UE improves scalability at the cost of ground-interaction fidelity and cross-process synchronization.

Chapter Outline

Distributed Simulation Architecture and Data Interfaces

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.4.1 Simulation Architecture
Textbook: Secs. 3.2.1.1–3.2.1.2

3.4.2 Spatial Reference Frames
Textbook: Sec. 3.2.2.2; added mounting example

3.4.3 Timing and Data Transfer
Textbook: Secs. 3.2.1.1, 3.2.1.3

3.4.4 Platform Interfaces and Sensors
Textbook: Secs. 3.2.1.3, 3.2.3

3.4.5 Integration and Troubleshooting
Textbook: Secs. 3.2.3, 3.3.1

3.4.2 Spatial Reference Frames: Definitions

Define World, Body, and Sensor Mounting Frames First

- **Local world frame:** choose a scene reference point as the origin; north-east-down (NED) has positive north, east, and down axes
  and describes an object's position in its environment.
- **Body frame:** forward-right-down (FRD) axes rotate with the vehicle's attitude.
  Body angular velocity RateB is in rad/s.
- **Mounting transform:** a sensor has a translation and rotation relative to the vehicle; to obtain its world pose, rotate the relative offset first,
  then add the vehicle's world position.

Body axes rotate with attitude, changing the world direction of a fixed mounting offset.

3.4.2 Spatial Reference Frames: Length Units

Example: Convert an NED Position from Meters to Centimeters

Given: lengths inside the UE scene are in centimeters. Convert the NED position [2, 3, −1] m to centimeters, keeping the same origin and axes.

North N	East E	Down D
2 m → 200 cm	3 m → 300 cm	−1 m → −100 cm

$$p_{\mathbf{NED,cm}} = 100 \times p_{\mathbf{NED,m}} = [200, 300, -100]\ \mathbf{cm}$$

Negative D is above the origin; an interface expecting meters still takes [2, 3, −1]. Do not multiply by 100 first.

Converting length units changes the scale of the numbers, not the reference frame.

3.4.2 Spatial Reference Frames: Axis Transformation

Example: Specify the Target Axes Before Transforming Position

Given: the two origins coincide; target X points north, Y east, and Z up, with units of cm.

Source NED position
[2, 3, −1] m

→

Target world position
[200, 300, 100] cm

$$[X, Y, Z] = 100 \times [N, E, -D]$$

Z points up and D down, so negate the third component; Euler angles require a separate attitude transformation.

Axis transformation and length conversion together determine the target coordinates.

3.4.2 Spatial Reference Frames: Camera Mounting Transform

Example: Transform a Configured Camera Offset into the World Frame

Given: the vehicle faces east (yaw 90°) at [2, 3, −1] m; camera 0 has SensorPosXYZ = [0.3, −0.15, 0] m, or 0.3 m forward and 0.15 m left.

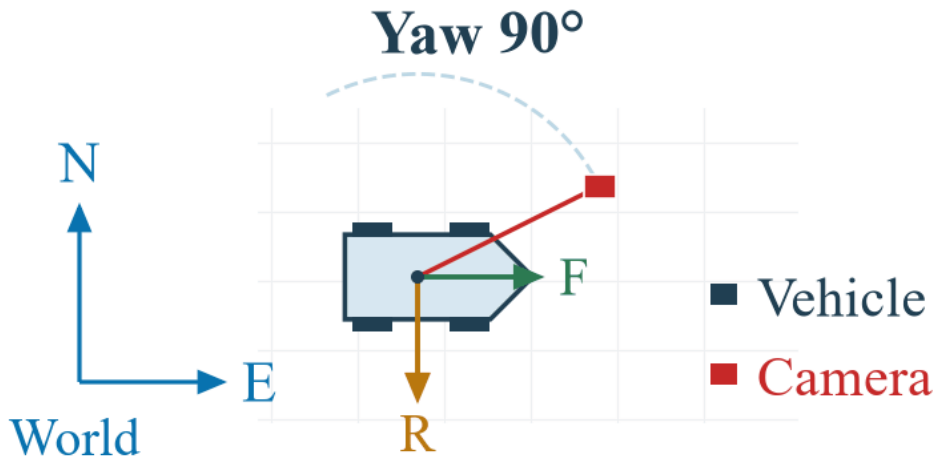


Figure: Camera turns with vehicle; world axes stay fixed

$$p_{\mathbf{c}} = p_{\mathbf{b}} + Rr$$

p_{b} / p_{c}: vehicle / camera world position R: rotation matrix from body to world frame r: camera mounting offset in the body frame

$$Rr = [0.15, 0.3, 0]\ \mathrm{m}$$

$$p_c = [2.15, 3.3, -1]\ \mathrm{m}$$

Rotate the mounting offset with the vehicle, then add it to the world position.

Chapter Outline

Distributed Simulation Architecture and Data Interfaces

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.4.1 Simulation Architecture
Textbook: Secs. 3.2.1.1–3.2.1.2

3.4.2 Spatial Reference Frames
Textbook: Sec. 3.2.2.2; added mounting example

3.4.3 Timing and Data Transfer
Textbook: Secs. 3.2.1.1, 3.2.1.3

3.4.4 Platform Interfaces and Sensors
Textbook: Secs. 3.2.1.3, 3.2.3

3.4.5 Integration and Troubleshooting
Textbook: Secs. 3.2.3, 3.3.1

3.4.3 Timing and Data Transfer: Rates and Time Domains

Rates Define Update Rhythm; Time Domains Define Instants

Stage	Display rendering	Camera sampling	Control computation
Assumed rate	60 Hz	30 Hz	100 Hz
Period	16.7 ms	33.3 ms	10 ms

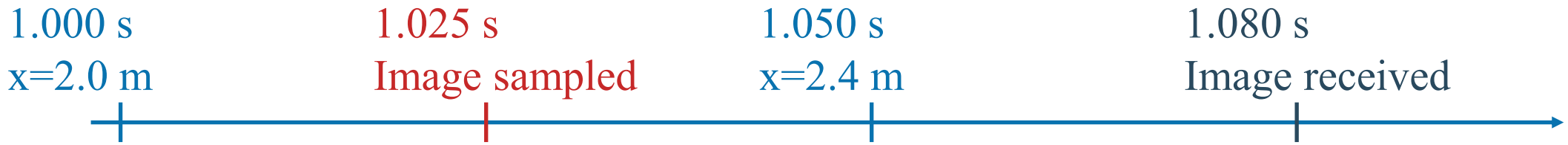
- Simulation time advances with simulation steps and may differ from real time when paused or accelerated; measure local elapsed time with a monotonic clock.
- **Time mapping:** calibrate clock offsets and rate differences; map states and images to one time domain before comparing timestamps.

Timestamps are comparable only after aligning the time origin and clock scale.

3.4.3 Timing and Data Transfer: Interpolation for Pairing

Example: Interpolate the State at the Image Sampling Time

Given: one time domain, both bracketing states, and approximately uniform motion between them.



$$\alpha = \frac{1.025 - 1.000}{1.050 - 1.000} = 0.5$$

$$x(1.025) = (1 - \alpha) \times 2.0 + \alpha \times 2.4 = 2.2\ \mathbf{m}$$

The 55 ms from sampling to receipt includes generation and waiting; pair the image with 2.2 m at its sampling time.

Use receive time to assess delay and sampling time to match states.

3.4.3 Timing and Data Transfer: Motion Prediction

Example: Extrapolate with Velocity Before the Next State Arrives

- Given: x(1.000 s)=2.0 m and x(1.050 s)=2.4 m; the image is sampled at 1.060 s, before the next state arrives.
- Estimate velocity over the preceding interval: (2.4−2.0)/0.050=8 m/s.

$$\hat{x}(1.060) = 2.4 + \frac{2.4 - 2.0}{0.050} \times 0.010 = 2.48\ \text{m}$$

- Extrapolate: $\hat{x}(1.060)=2.4+8\times0.010=2.48$ m; using 2.4 m directly would lag by 0.08 m.
- Extrapolation is an estimate, not a measurement; correct it when subsequent states arrive.

3.4.3 Timing and Data Transfer: State at Sampling Time

A Late Image Still Needs the Position at Its Sampling Time

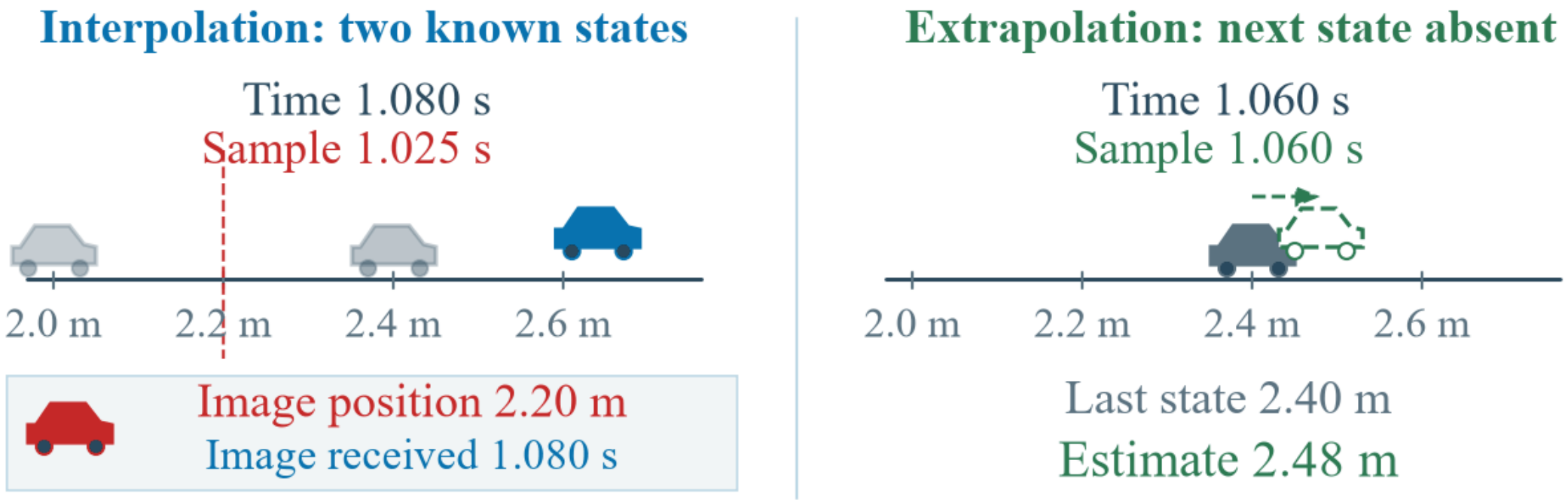


Figure: Solid Vehicles Show the Scene or Known States; Dashed Vehicles Show Extrapolated Estimates

Interpolate when both bracketing states are available; otherwise predict using the motion model.

3.4.3 Timing and Data Transfer: Raw Image Data Rate

Example: RGB Data Rate from Pixel Count and Frame Rate

Given: 640×480, RGB with 8 bit per channel, 30 Hz; exclude compression and protocol overhead.

Pixels per frame	×	**Per pixel**	×	**Frames per second**
640 × 480		3 × 1 B		30

$$640 \times 480 \times 3 \times 30 = 27{,}648{,}000\ \textbf{B/s}$$

About 26.4 MiB/s, or 221 Mb/s; compression saves bytes but adds encoding and decoding time.

Raw data rate establishes the scale; encoding and transfer add further costs.

3.4.3 Timing and Data Transfer: Image Transport

Shared Memory on One Computer; UDP Across Computers

- Local shared memory: producer and consumer share memory; check the write-complete flag, timestamps, and array shape.
- UDP across computers: images may be encoded or split into packets; use the current SDK example to reassemble and decode them and detect packet loss.
- Uncompressed image byte rate = width × height × channels × bytes per channel × frame rate; network protocols add overhead.
- SendProtocol/sendMode documentation and implementations differ by version; check the current configuration and code before assigning meanings to mode numbers.

Shared memory avoids network transfer; UDP spans computers but requires packetization and handling of packet loss.

Chapter Outline

Distributed Simulation Architecture and Data Interfaces

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.4.1 Simulation Architecture
Textbook: Secs. 3.2.1.1–3.2.1.2

3.4.2 Spatial Reference Frames
Textbook: Sec. 3.2.2.2; added mounting example

3.4.3 Timing and Data Transfer
Textbook: Secs. 3.2.1.1, 3.2.1.3

3.4.4 Platform Interfaces and Sensors
Textbook: Secs. 3.2.1.3, 3.2.3

3.4.5 Integration and Troubleshooting
Textbook: Secs. 3.2.3, 3.3.1

3.4.4 Platform Interfaces and Sensor Configuration: Custom Development

Custom Development through Python, Simulink, and Blueprints

- An application programming interface (API) defines calls; a software development kit (SDK) supplies implementations, examples, and documentation.
- Python: send commands/poses to the 3D application through control interfaces and read responses for automated control.
- Simulink: interact through structures or blocks to connect models with the 3D display.
- Blueprints: extend object behavior inside the 3D application, with external commands triggering custom actions.

Python and Simulink send commands from outside the 3D application; Blueprints implement custom actions inside it.

3.4.4 Platform Interfaces and Sensor Configuration: Sensor Types

RflySim3D Sensor Type IDs: Images, Point Clouds, and Sound

- Images | 1 color, 2 depth, 3 grayscale, 4 segmentation, 8 fisheye | Available in the free edition
- Paid image types | 9 gimbal camera, 40 infrared grayscale, 41 infrared pseudocolor | Paid edition only
- Point clouds | 20/21 LiDAR (sensor-local/world coordinates), 22/23 Livox-style scanning, 7 depth-derived point cloud | 20–23: free edition; 7: paid edition only
- Other | 5 range, 10 optical flow, 30 simple vision, 31 simple radar, 6 sound | 6 sound: paid edition only; others: free edition

Choose the sensor type for the task, then check edition requirements to avoid configuring an unavailable type.

3.4.4 Platform Interfaces and Sensor Configuration: Configuration Example

Sensor IDs, Vehicle IDs, and Array Indices Have Distinct Roles

TypeID: 1 SeqID: 0
TargetCopter: 1 TargetMountType: 0
DataWidth: 640 DataHeight: 480
CameraFOV: 90 DataCheckFreq: 30

- **Configuration units:** CameraFOV=90 is a 90° field of view; DataCheckFreq=30 requests 30 Hz updates. Actual receive rate also depends on rendering and transfer.
- **Mounting:** mode 0 follows the vehicle; SensorPosXYZ sets the offset from its center. TargetCopter=1 binds to vehicle 1; SeqID is the sensor ID, while i is the list index.

TypeID selects the data type, SeqID identifies the sensor, and TargetCopter specifies its mounting target.

3.4.4 Platform Interfaces and Sensor Configuration: Display State Message

Fields in a Display State Message

ue.sendUE4PosFull(1001, 3, [0]*8, [1,0,0], [2,3,-1], [0,0,0], [0,0,1.5708], 0)

copterID=1001 identifies a separate example instance; vehicleType=3 selects a quadcopter; windowID=0 selects the window.

Field	Example value	Meaning and units
PosE	[2, 3, −1]	North-east-down (NED) position, m
VelE	[1, 0, 0]	NED velocity, m/s
AngEuler	[0, 0, 1.5708]	Roll, pitch, yaw, rad

Agree on object identity, reference frame, and units together so that poses have a definite meaning.

3.4.4 Platform Interfaces and Sensor Configuration: Instance Lifecycle

Instance IDs Select Objects for Updates and Deletion



Figure: Deleting Vehicles 1000 and 1001 with RflyDelVehicles (Screen Recording)

A new pose with the same ID updates that object.

Stop sending for these IDs first, then issue the delete command.

RflyDelVehicles "1000,1001"

Continuing to send poses with the same ID recreates a deleted object.

3.4.4 Platform Interfaces and Sensor Configuration: Camera State and Pixels

Camera State and Pixel Arrays Follow Separate Paths

State query

ue.reqCamCoptObj(0, 0)
ue.initUE4MsgRec()
ue.getCamCoptObj(0, 0)

CameraData: identifiers, pose, width, and height;
no pixels.
Target: query the camera with SeqID=0 in this example.
Reception: returns 0 before a state is received; check the request and reception first.

Image capture

vis.jsonLoad()
vis.sendReqToUE4()
vis.startImgCap()

vis.Img[i]: read pixels by configuration-list entry.
SeqID: identifies the camera; verify the list index separately.
New frame: hasData[i] indicates a readable buffer;
then check the sampling timestamp.

3.4.4 Platform Interfaces and Sensor Configuration: Received Arrays

An Image Frame Has Both Pixels and Sampling Information

```
img = vis.Img[i]
shape, dtype = img.shape, img.dtype
sensor_id = vis.VisSensor[i].SeqID
sample_time = vis.timeStmp[i]
```

- **Generation chain:** Scene Capture 2D → Render Target (image buffer) → encoding and reception → array.
- **Fields:** vis is the capture object, i the list index, and SeqID the sensor ID. During continuous capture, save each image with its matching timestamp.

Array interpretation determines displayed colors; sampling information identifies the sensor and when the frame was generated.

3.4.4 Platform Interfaces and Sensor Configuration: Array Formats

Image Array Shapes, Data Types, and Channel Order

Observation	Array format	Interpretation
Color/segmentation	H×W×3; uint8; BGR	Channels: blue, green, red
16-bit PNG depth	H×W; uint16	Raw depth integer codes
Raw depth transfer	H×W×1; uint16	Last axis is 1; can reduce to 2D

Raw depth: uint16 stores integer codes, not meters.
Conversion: mask invalid codes, then convert to distance using the protocol's quantization step.
Parameters: the protocol defines the step and invalid code; see the example in Unit 3.3.2.

BGR specifies color channel order; uint16 depth needs distance decoding rather than color interpretation.

3.4.4 Platform Interfaces and Sensor Configuration: Directions and Ports

Separate Ports for Commands, States, and Images

Data direction	Example address and port	Purpose
Python → 3D window 0	127.0.0.1 : 20010	Poses, console commands
3D application → state receiver	224.0.0.10 : 20006	Camera/object states
3D application → image receiver	192.168.1.20 : 10001	**First entry 1:** encoded images over UDP

Cross-computer UDP example: SendProtocol = [1,192,168,1,20,10001,0,0]

- **Address direction:** send commands to the 3D application and return images to the algorithm; 127.0.0.1 refers only to the local computer.

3.4.4 Platform Interfaces and Sensor Configuration: Interpreting Results

Interpret Results through Array Shape, Log Fields, and Faults

Sensor types, configuration, and output encoding
https://rflysim.com/doc/zh/4.DevRef/vis-cfg.html

Image acquisition interface and call examples
https://rflysim.com/doc/zh/4.DevRef/pysdk/vision/viscap.html

Image acquisition: run GetCamObjDemo.bat first, then use Python38Run.bat to launch GetCamObjDemoWithCam.py.

GetCamObjDemo.py queries states only; it does not acquire pixels.
Expected: two views, Img0 and Img1, with color array shape (480,640,3).
Diagnosis: if two 640 values are printed, check for repeated DataWidth; if no image appears, check the acquisition request and reception.

Explain how observations are generated, then use arrays, configuration, and timing to organize the experiment.

Chapter Outline

Distributed Simulation Architecture and Data Interfaces

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.4.1 Simulation Architecture
Textbook: Secs. 3.2.1.1–3.2.1.2

3.4.2 Spatial Reference Frames
Textbook: Sec. 3.2.2.2; added mounting example

3.4.3 Timing and Data Transfer
Textbook: Secs. 3.2.1.1, 3.2.1.3

3.4.4 Platform Interfaces and Sensors
Textbook: Secs. 3.2.1.3, 3.2.3

3.4.5 Integration and Troubleshooting
Textbook: Secs. 3.2.3, 3.3.1

3.4.5 Integration and Troubleshooting: Field Diagnosis

Diagnosis: Why Do Logged Dimensions Differ from the Configuration?

Print statement at the end of GetCamObjDemoWithCam.py

```
print('Camera: ', Cam.SeqID, Cam.PosUE,
      Cam.DataWidth, Cam.DataWidth)
```

- **Field evidence:** configuration sets DataWidth=640 and DataHeight=480, but the log reads DataWidth twice, producing 640, 640.
- **Interpretation:** CameraData is state metadata. Read pixel dimensions from vis.Img[i].shape; the expected BGR array is (480,640,3).
- **Diagnostic branches:** if state exists but images are missing, check acquisition requests and hasData. If complete arrays lag in processing, compare image timestamps with processing times.

Identify the fields being printed before deciding whether the data link is faulty.

3.4.5 Integration and Troubleshooting: Isolating Stages

Trace Each Stage from Request to Reading to Find the Break

- Request: confirm that the image request was sent; record its return value and target window/sensor ID.
- Generation: check that the 3D application created the sensor from the current configuration and that sampling timestamps advance.
- Return transfer: verify shared memory or the UDP receive address, port, packetization status, and firewall.
- Reading: inspect hasData, array shape, and old/new timestamps; change only one condition at a time.

Use each stage's own output as evidence and change one condition at a time to locate the break.

3.4.5 Integration and Troubleshooting: Interface Resources

Understand Message Fields and Trace Data Transfer

Display state fields: check PosE units and the structures returned by object queries

https://rflysim.com/doc/zh/4.DevRef/pysdk/ue/ue4ctrl.html

Image transport: trace shared memory, UDP packetization, and return addresses

https://rflysim.com/doc/zh/4.DevRef/vis-comm.html

Object lifecycle: compare stopping updates, deletion by ID, and recreation

https://rflysim.com/doc/zh/4.DevRef/3d-keys.html

Message fields define objects, units, and time; transport configuration determines which receiver gets the data.

In-Class Exercises

Three Exercises: Mounting, Bandwidth, and Troubleshooting

- Mounting: assume an FRD body frame and an NED world frame. The vehicle faces east at [2,3,−1] m, and the camera's body-relative offset is [0.3,0.15,0] m. Find its world position.
- Bandwidth: for two 640×480, 30 Hz, 8-bit RGB cameras, how many uncompressed bytes are produced per second? With a 60,000 B payload limit, how many packets does each frame need?
- Troubleshooting: state data are normal but image hasData[0] is false. List evidence for request, generation, return transfer, and reading.

Receiving data is only the start; also verify spatial meaning, transfer scale, and data-link completeness.

Answer Guide

Answers and Self-Checks for the Three Exercises

- Mounting: the right axis points south, so the rotated offset is $[-0.15, 0.3, 0]$ m and the camera is at $[1.85, 3.3, -1]$ m. These assumed frames do not replace version-specific SensorPosXYZ verification.
- Bandwidth: one camera produces 27,648,000 B/s; two produce 55,296,000 B/s. A frame is 921,600 B and needs 16 packets with a 60,000 B payload limit.
- Troubleshooting: check requests and configuration first, then receive mode, address, port, and packetization; finally inspect buffer flags and timestamps.

For numerical problems, check rotation direction and units first; for troubleshooting, check requests before return transfer.

Section Summary

Match Data through Object, Spatial, and Time Conventions

- Architecture: the vehicle motion model DLL runs in CopterSim; the 3D application handles rendering, collision detection, and sensor data; external programs send commands and receive data through UDP.
- Space: confirm reference frames and units first. Rotate mounting offsets with attitude before adding vehicle position; convert meters/centimeters and down/up directions in separate steps.
- Time: pair by sampling time and assess delay by arrival time; extrapolation before the next state arrives is only an estimate and must be corrected with new states.
- Pitfalls: an ID still receiving pose updates is recreated after deletion; true hasData does not mean a new frame; inspect the print statement before diagnosing the link from log fields.

Cross-program data correspond only when object IDs, coordinate units, and sampling times agree.