Chapter Outline

Scene and Vehicle Asset Development

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

4 Units in This Section

3.5.1 3D Asset Integration Routes

Textbook: Sec. 3.2.2.1

3.5.2 UE Asset Management

Textbook: Sec. 3.2.2.2

3.5.3 Vehicle XML and Actuators

Textbook: Secs. 3.2.1.2, 3.2.2.3–3.2.2.4

3.5.4 Asset Validation and Reuse

Textbook: Secs. 3.2.1.2, 3.2.3; validation

3.5.1 3D Asset Integration Routes: Three Routes, Two Asset Types

Choose Scene and Vehicle Delivery Routes by Asset Source

Routes: Resolve UE asset dependencies and cook; parse GLB with a runtime plugin; load terrain and imagery through tiles and services.
Selection: Check the input format, then the target application. The following pages develop the UE scene and vehicle branch.

- Scenes: Create maps, materials, and Blueprints in UE, then cook and deploy them.

- **Vehicles:** Pair static meshes or Blueprint models with Extensible Markup Language (XML) configuration to register models and components.
- Runtime: The 3D application loads assets; CopterSim, Python, or Simulink sends IDs, types, and poses.

Deliver scene assets and vehicle models with their dependencies; simulation data drives their displayed states.

3.5.1 3D Asset Integration Routes: Sources and Outputs

How Assets Enter the Simulator

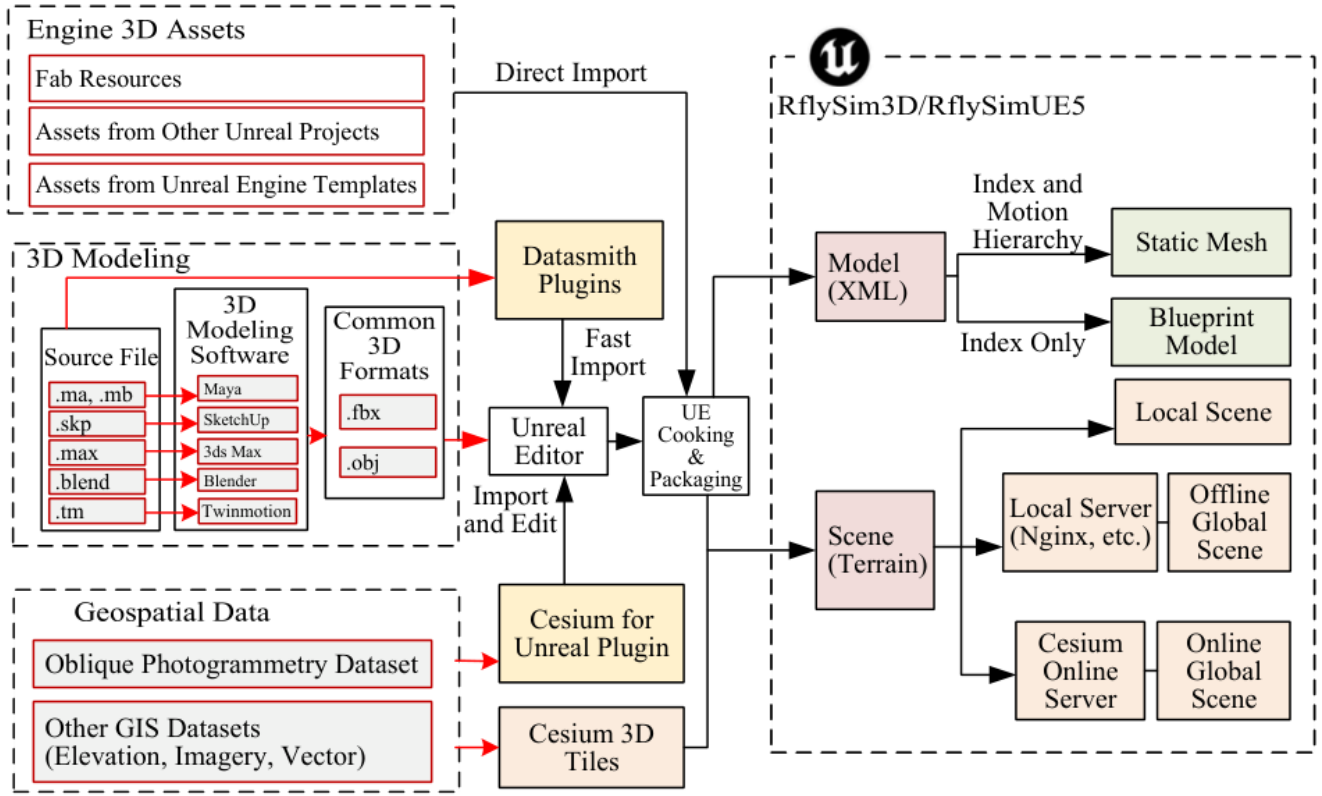


Figure 3.7 3D Asset Import Methods in RflySim3D

Read the diagram: Locate the input, then follow the arrows to check the conversion tool and output.

Examples here: FBX through the editor; GLB parsed at runtime; geographic data converted to tiles.

3.5.1 3D Asset Integration Routes: Engine Assets

Bring Existing UE Assets into the Editor

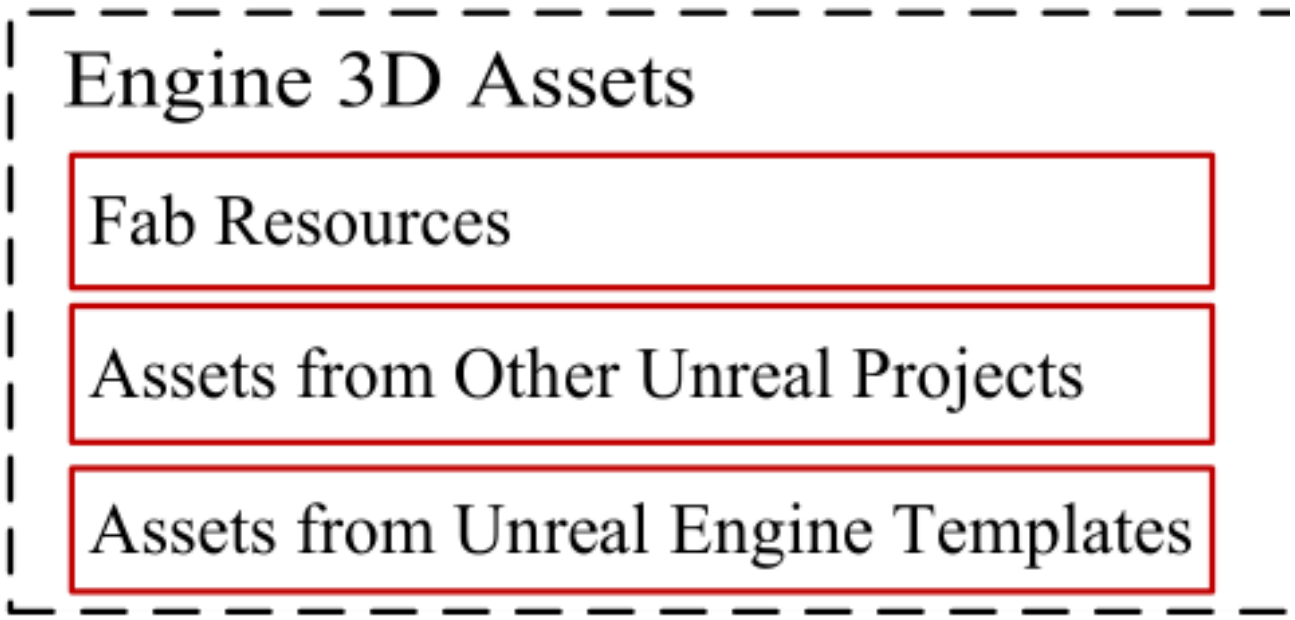


Figure 3.7 (engine assets): Three reusable asset sources

- **Asset library:** Check licenses. Import only authorized assets.
- **Project migration:** Migrate materials, textures, and references together.
- **Development templates:** Start with an existing scene, then modify it.

For UE assets, check licenses and dependencies before adding them to the project and cooking.

3.5.1 3D Asset Integration Routes: Model Assets

XML Has Different Roles for Two Vehicle Asset Types

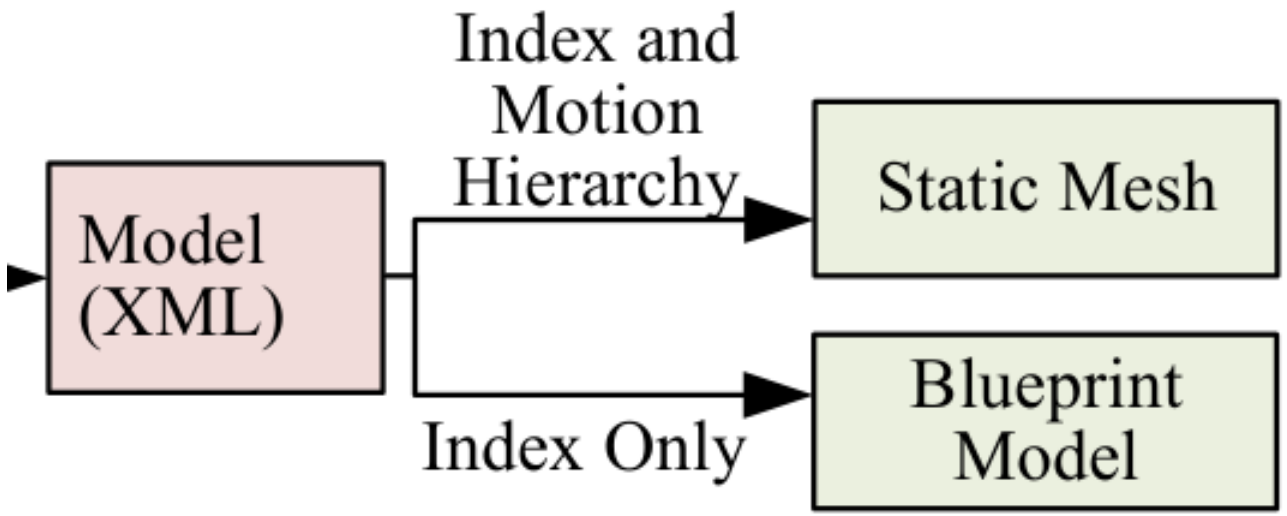


Figure 3.7 (model assets): XML indexing and motion hierarchy

- **Static meshes:** Define component motion hierarchies in XML.
- **Blueprint models:** Encapsulate the hierarchy and motion logic in the asset.
- **Shared entry point:** XML registers the asset the 3D application loads.

Static meshes define motion hierarchies in XML; Blueprints encapsulate motion within the asset.

3.5.1 3D Asset Integration Routes: Modeling Files

Choose FBX, OBJ, or Datasmith by What Must Be Preserved

- FBX: Carries meshes, hierarchies, and some material references; check component pivots after import.
- **OBJ:** Primarily transfers static geometry and UVs; appearance depends on MTL and texture files.
- Datasmith: Transfers scene hierarchies, cameras, and lights; check plugin compatibility by version.
- Example: Choose FBX for a multipart rotor; check pivots and the component hierarchy after import.
- Acceptance: Check mesh count, texture channels, and Blueprint motion against the task.

Check hierarchy, materials, textures, and motion before choosing an import format.

3.5.1 3D Asset Integration Routes: Three Routes

Select an Import Route for the Asset and Target Environment

- **Editor import:** FBX or UE assets → resolve dependencies and cook → Content assets; XML registers models.
- **Runtime parsing:** GLB or other local meshes → UMG configuration and parsing → repository entry → scene instance.
- **Geographic data:** OSGB, elevation, or imagery → corresponding tile products → Cesium components and data services.

Cooking: Convert editor assets and their dependencies into runtime files the target application can load.

Older TM example: e1_TwinmotionContent; TM2022.1.2 → .tm → UE4.27.

TM house example: 5.TwinmotionUsage; TM2023.1.2 → .udatasmith plus assets → matching UE5.

Asset format, target application, and data dependencies jointly determine the route.

3.5.1 3D Asset Integration Routes: Minimal Mesh Workflow

Runtime Import: Configure the Model Repository

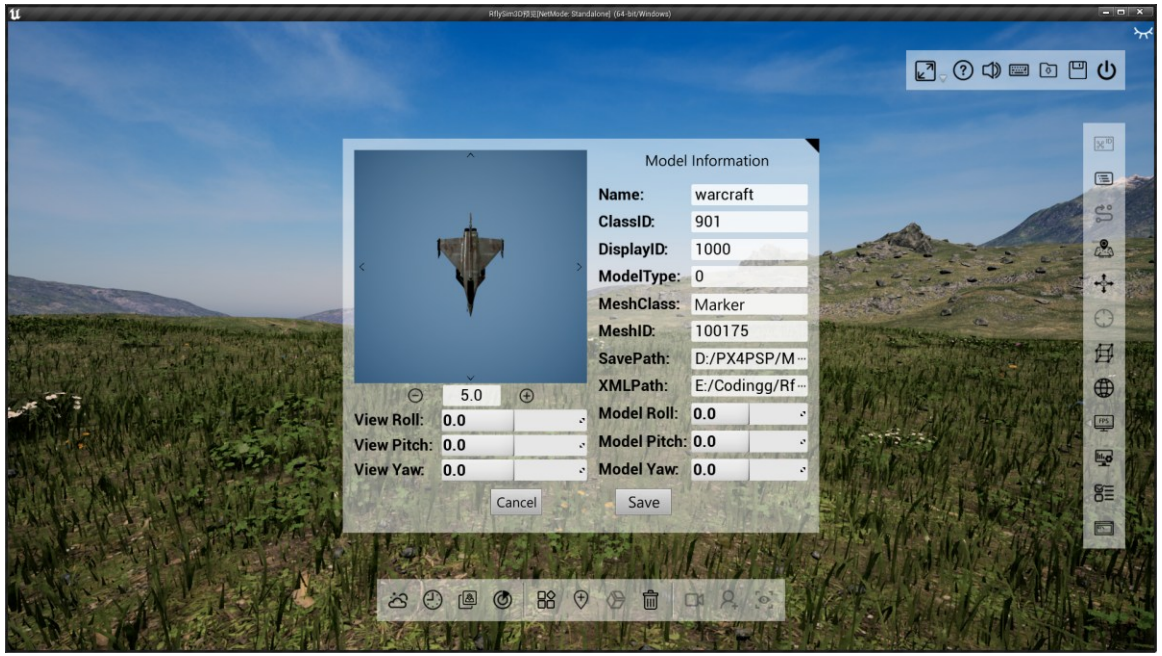


Figure: Repository configuration for the warcraft model

- **Shown:** Configuration for the warcraft aircraft; use test.glb for the exercise.
- **Configure:** Enter a name and an unused asset ID; adjust the pose and save path.
- **Requirement:** This GLB example uses the runtime plugin in the Full edition.

Repository configuration stores an asset definition; scene creation produces an independent instance.

3.5.1 3D Asset Integration Routes: Runtime Steps and Outputs

From a GLB Repository Entry to a Scene Instance

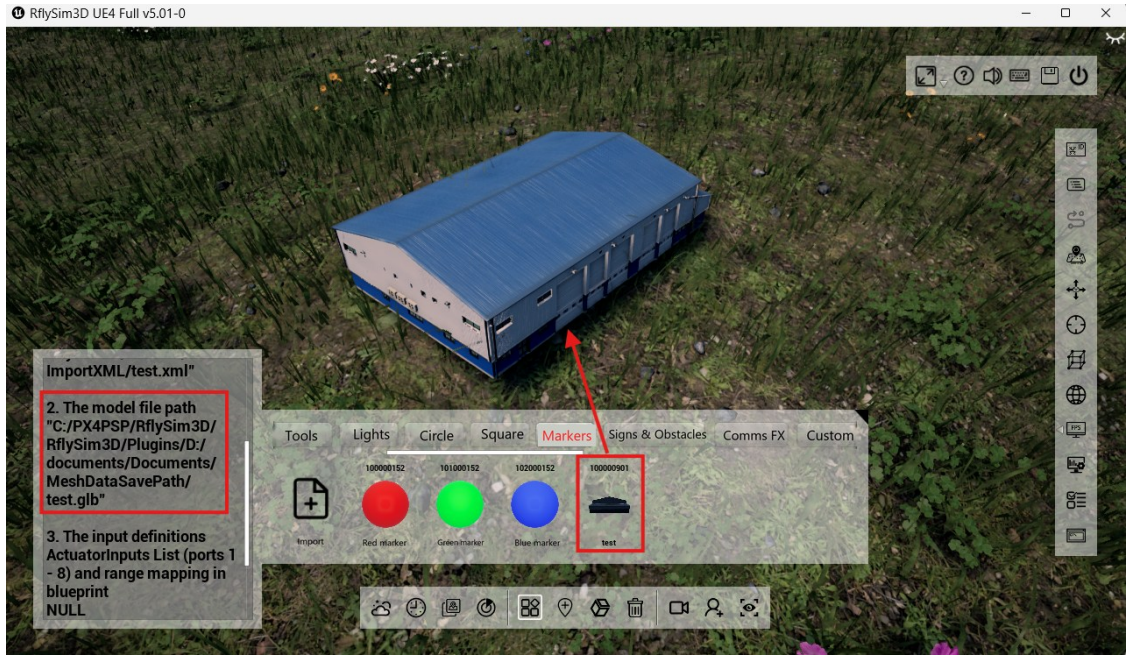


Figure: test.glb repository entry and scene instance (UE4 Full v5.01)

Entry: 1.BasicExps/ e4_CusLoadObj/ 2.loadSimpleMesh.

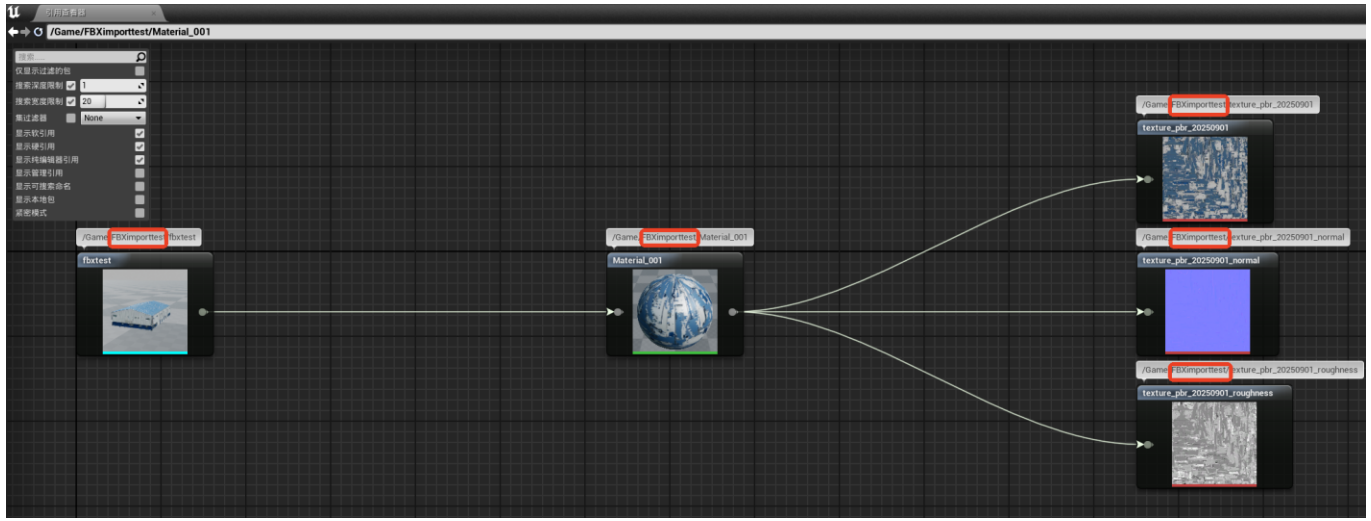
Asset: A test entry appears in the list; use the index in the red box to select the model.

Instance: Drag it into the scene to display the building, including its roof, walls, and materials; assign an object ID separately.

The model file defines appearance, the repository index selects the asset, and the scene instance carries position and attitude.

3.5.1 3D Asset Integration Routes: FBX Dependencies

Follow References to Deliver the Complete Model Appearance



Entry: 1.BasicExps/e4_CusLoadObj/3.loadMeshXML.

Figure: Material and texture references for fbxtest

- **Reference chain:** The mesh links to three textures through a material.
- If dependencies are missing, the shape may appear without colors or lighting detail.
- **Complete deployment:** Cook the model and dependencies; preserve reference paths.

Models, materials, and textures form a reference chain; deploy their corresponding runtime assets together.

3.5.1 3D Asset Integration Routes: Cooking and Paths

Match UE4.27 Cook Output to the XML Path

- **Version branch:** This example uses UE4.27; UE5 requires cooking again in a matching project.
- **Cook output:** Saved/Cooked/WindowsNoEditor/<ProjectName>/Content.

- **Runtime deployment:** RflySim3D/RflySim3D/Content; preserve the model, material, and texture directory hierarchy.
- **Path mapping:** /Game/FBXimporttest/fbxtest locates the asset in that subdirectory; omit the .uasset extension.

Steps: Save all assets, cook content for Windows, and confirm success in the log.

MeshPath uses a UE asset path without the file extension.

3.5.1 3D Asset Integration Routes: Static Model XML

fbxtest.xml Registers a Creatable Model Style

<ClassID>910</ClassID>
<DisplayOrder>100</DisplayOrder>
<ModelType>0</ModelType>
<MeshPath>/Game/FBXimporttest/fbxtest</MeshPath>

Index: 100×100000+910=10000910.

Loading: ModelType=0 selects a static mesh; MeshPath locates the deployed asset.

Registration: Place XML with the assets in Content/FBXimporttest; restart and check the TXT file with the same base name.

Prerequisite: Prepare and cook the FBX first; the example supplies configuration and scripts, but no matching model.

3.5.1 3D Asset Integration Routes: Styles and Instances

Create an Object in Python with a Style Index

```
ue.sendUE4Cmd('RflyChangeMapbyName VisionRing')
ue.waitForMapLoaded('VisionRing')
time.sleep(3)
ue.sendUE4Pos(100,10000910,0,[2.5,0,0],[0,0,0])
```

Parameters: Instance ID 100; style 10000910; position [2.5,0,0] in meters; attitude in radians.
Run: Launch loadmodel.bat from this example directory; create the object after VisionRing loads.
Expected: The model appears if all assets are present; another pose update to ID 100 changes only that instance.

One style can create multiple instances; position updates select an instance by ID.

3.5.1 3D Asset Integration Routes: Scene and Terrain Files

Deliver the 3D Scene and Terrain Heights Separately

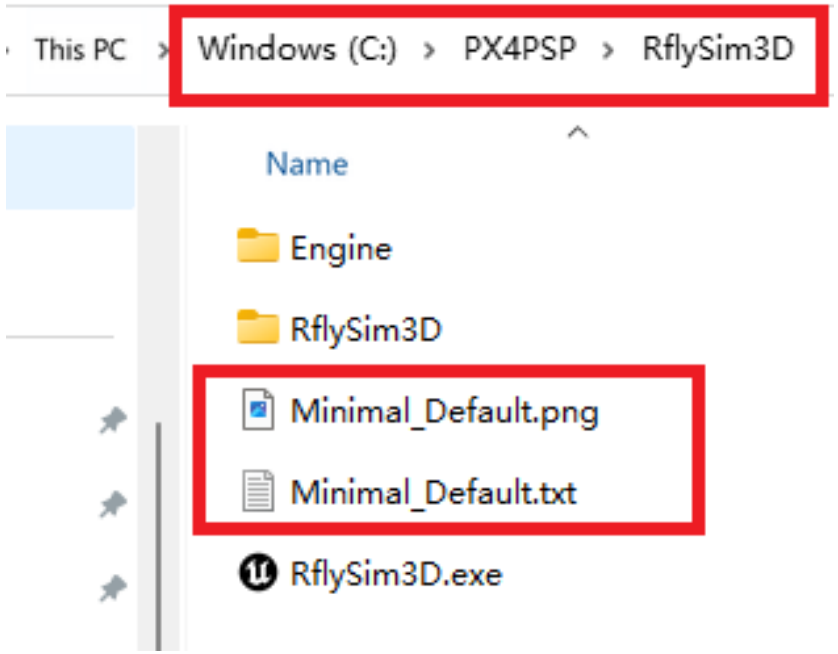


Figure: Terrain files for Minimal_Default

- **Visualization:** RflySim3D loads the .umap level and its dependencies.
- **Scan output:** A PNG height matrix and TXT calibration file with the same base name are generated in the RflySim3D directory.
- **Deployment:** Copy the PNG and TXT together to CopterSim/external/map.

Visual assets provide the display; height files support terrain queries.

3.5.1 3D Asset Integration Routes: Geographic Data Products

Distinguish Models, Elevation, and Imagery

Input	Processing or metadata	Entry point and product
OSGB model	Data/Tile_*, metadata.xml	tileset.json and child tiles
Elevation TIF	Terrain height tiles	layer.json and terrain tile hierarchy
Imagery TIF	Surface imagery tiles	tilemapresource.xml and TMS

SRS: Coordinate reference system; SRSOrigin: Source model origin.
The map origin connects to simulation coordinates; a model height offset adjusts the relative positions of datasets.

The three geographic data types retain distinct products and loading entry points.

3.5.1 3D Asset Integration Routes: Geographic Prerequisites

Choose a GIS Workflow for the Data and Target Version

- **OSGB route:** 2.AdvExps/e3_CusGIS/1.ObliModelMap; first prepare Data/Tile_* and metadata.xml.
- **Offline terrain/imagery:** 3.CustExps/e3_CusGIS/e5_CesiumOffline; requires the Full UE5.8 environment and local data services.
- **Preparation:** Obtain source OSGB and TIF data and service software separately; deliver the tile root directory, service address, and map origin, with the height offset recorded separately.

The OSGB route delivers model tiles; offline terrain also depends on elevation, imagery, and data services.

3.5.1 3D Asset Integration Routes: Geographic Data

Convert Photogrammetry and GIS Data into Tiled Assets

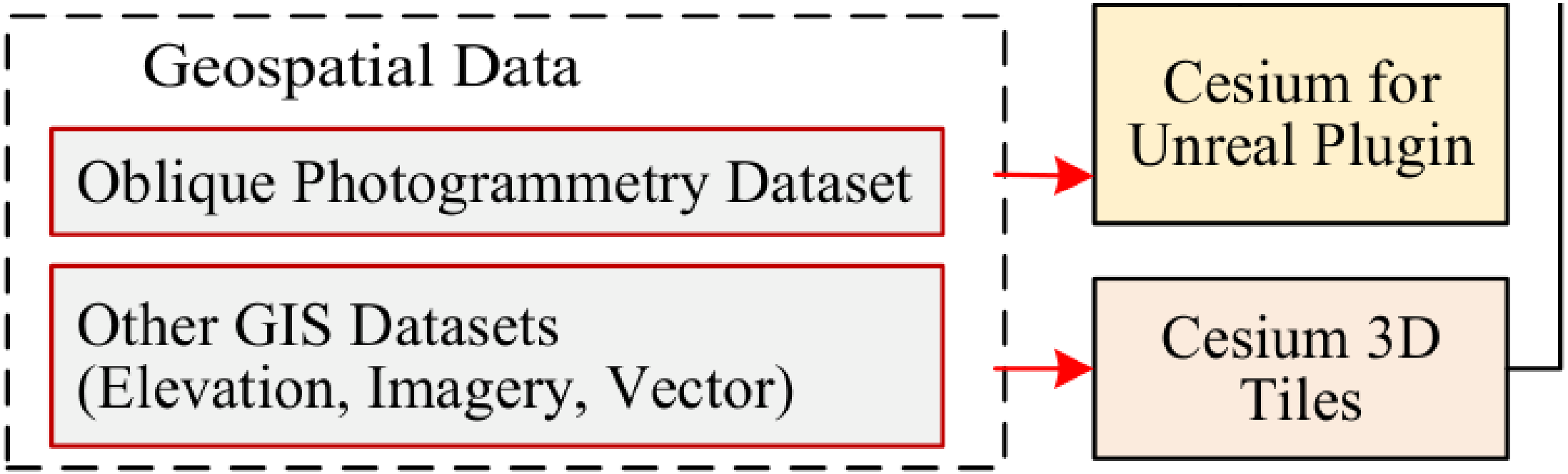


Figure 3.7 (geographic data): Tiling tools convert OSGB, elevation, and imagery into loadable assets.

Chapter Outline

Scene and Vehicle Asset Development

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

4 Units in This Section

3.5.1 3D Asset Integration Routes

Textbook: Sec. 3.2.2.1

3.5.2 UE Asset Management

Textbook: Sec. 3.2.2.2

3.5.3 Vehicle XML and Actuators

Textbook: Secs. 3.2.1.2, 3.2.2.3–3.2.2.4

3.5.4 Asset Validation and Reuse

Textbook: Secs. 3.2.1.2, 3.2.3; validation

3.5.2 UE Asset Management: Scene Loading Routes

One Scene Configuration Can Reference Local or Global Environments

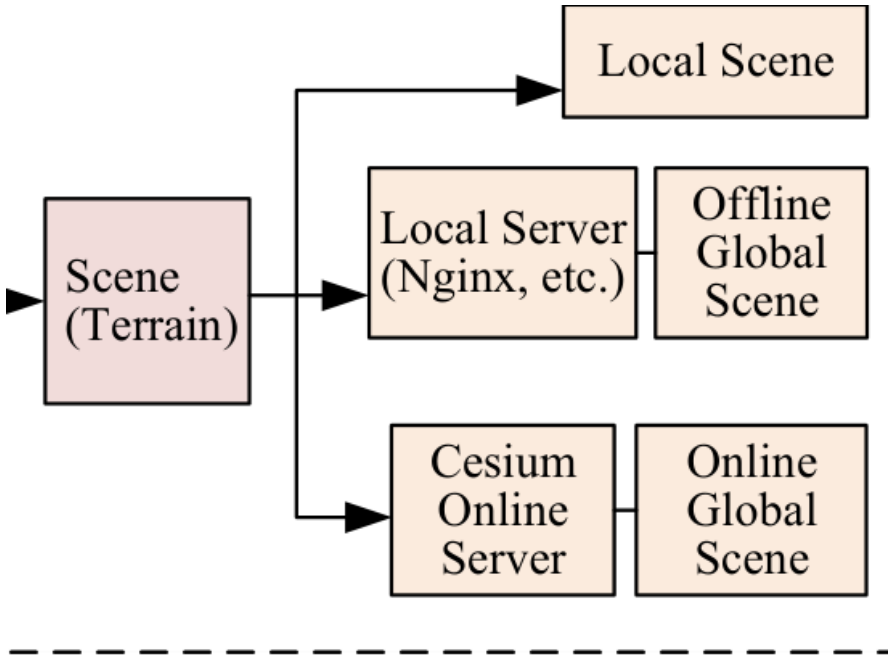


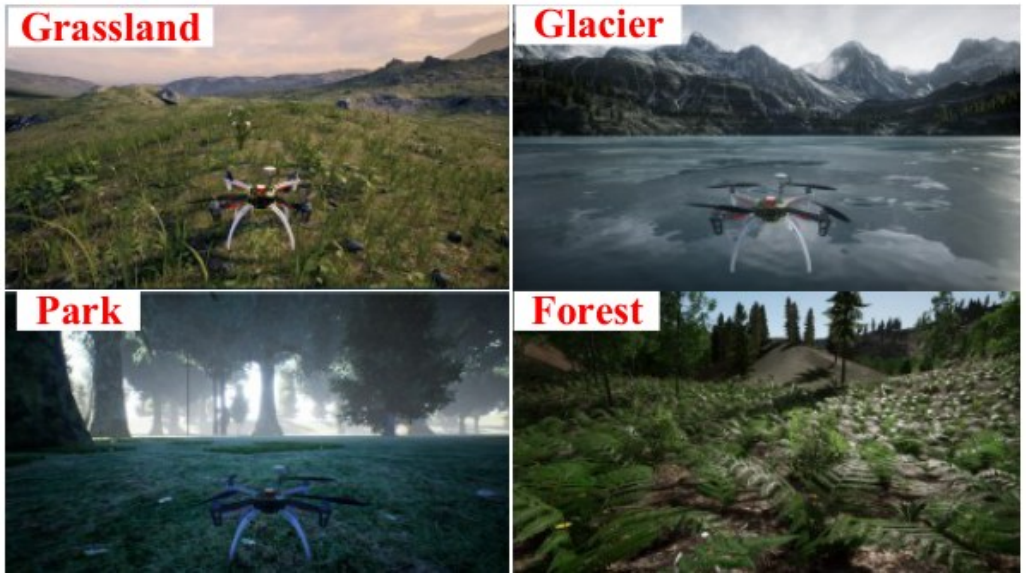
Figure 3.7 (scenes): Three loading routes

- **Local scene:** Cook and deploy the project assets.
- Offline global scene: A local service provides tiles.
- **Online global scene:** Cesium services load data on demand.

Local assets, offline services, and online services have different dependencies.

3.5.2 UE Asset Management: Local Environment Variations

Organize Environment Assets for Repeatable Comparisons



- **Map version:** Save environment assets
and the coordinate origin.
- **Shared task:** Keep camera parameters, target trajectory, and evaluation rules fixed.
- **Record variables:** Change only the surface or occlusion; record the asset version.

Figure 3.8(a) Local Scene Models

Hold task conditions fixed; record changed backgrounds and occlusion through asset versions.

3.5.2 UE Asset Management: Scale, Axes, and Pivots

Configure Component Pivots and Axis Directions

- **Units:** In this example, an XML relative position of 25.426 cm equals 0.25426 m. Use the field
units to avoid a 100× scale error.
- **Pivot position:** Put the rotor component origin on its shaft; XML relative position places that pivot on the body.
- **Axis direction:** The direction vector specifies the rotation axis, not a second position or a displacement.
- **Motion:** The component rotates about its own correct pivot; an offset pivot makes the blades sweep an eccentric path.

The component origin sets the pivot; relative position places it; the direction vector defines the rotation axis.

3.5.2 UE Asset Management: Units and Axes

Set Centimeters and Forward–Right–Up Before Rotor Pivots

- Scale: In this example, 1 Unreal Unit in UE equals 1 cm; standardize model units before import.
- **Coordinates:** UE uses Forward–Right–Up; this 3ds Max model has the opposite lateral axis.
- Check: The front-right rotor belongs ahead of and to the right of the body, with its pivot on the shaft.
- Example: A 100 cm arm should measure 100 units in UE.

$$y_{\mathrm{UE}} = -y_{\mathrm{source}}$$

Skipping unit or Y-axis conversion gives a 100× scale error or a left–right mirror image.

3.5.2 UE Asset Management: Coordinate Conversion Example

Calculate the X680 Front-Right Rotor Position

- Given: Body center (-0.449,-0.363,0) cm.
- Rotor pivot (24.977,-24.439,11.222) cm.
- Result: The source lateral difference is -24.076 cm; reversing its sign gives UE Y=+24.076 cm.

$$\Delta x = 24.977 - (-0.449) = 25.426 \text{ cm}$$

$$\Delta y = -24.439 - (-0.363) = -24.076 \text{ cm}$$

Subtract to obtain local position, then convert the source lateral axis to the UE direction.

3.5.2 UE Asset Management: Visible Meshes and Collision

Materials, Collision, and Level of Detail Affect Different Outputs

- **Materials:** Complete texture references let surface color and detail load with cooked assets.
- **Collision:** Boxes, spheres, capsules, or simple convex hulls support detection; ground placement requires a usable surface.
- **Preprocessing:** DCC decimation reduces polygon count; retopology rearranges face connectivity. Remove unnecessary hidden detail as needed, then import into UE.
- **LOD handoff:** Generate or import levels of detail for static and skeletal meshes in UE. For skeletal meshes, also check the skeleton, skin weights, and motion deformation.
- **Evaluate:** Switch LOD by screen coverage while preserving task-relevant outlines; compare frame time from the same viewpoint (Section 3.2). Check collision separately for ground placement and raycasts.

Configure visible surfaces, collision geometry, and near/far detail separately.

3.5.2 UE Asset Management: Lighting and Materials

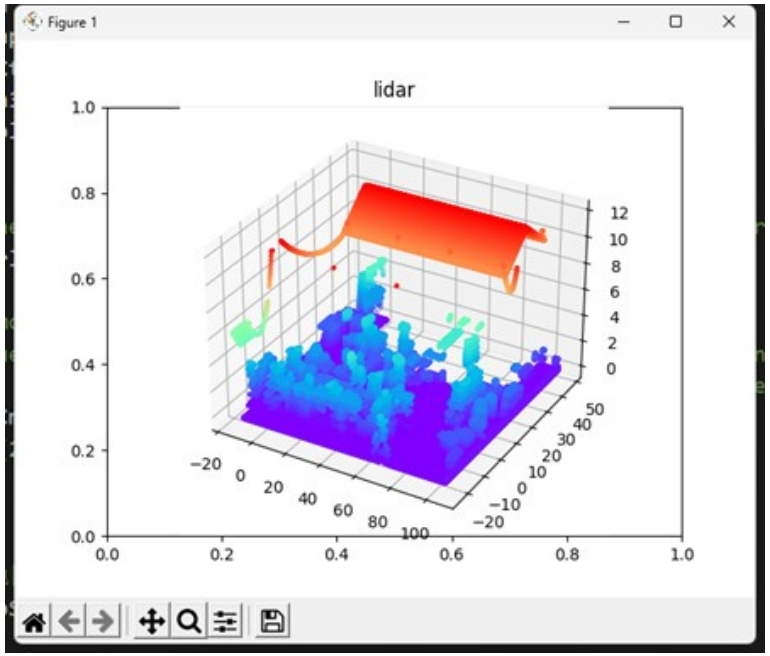
Check Lights, Materials, and Texture References After Import

- Lights: Scene transfer may convert lights; verify their real-time appearance in UE.
- **Materials:** FBX cannot fully preserve the source application's shader graph; check texture channels.
- References: Deliver materials and textures with the project, then check appearance again after cooking.
- Check order: Find missing textures first, then check material slots and light types.
- Acceptance: Compare appearance before and after cooking from the same viewpoint; retain screenshots of differences.

If geometry is correct but the image is all gray, check materials and dependencies first.

3.5.2 UE Asset Management: Terrain Samples

Convert a Terrain Height Grid into Sample Points



- **Input:** Height PNG and TXT calibration.
- **Transform:** Combine horizontal sample positions with elevations to form 3D points.
- **Source:** Terrain grid readings, not recorded LiDAR measurements.

Figure: Display of terrain elevation samples

The height point cloud comes from grid sampling and calibration.

Chapter Outline

Scene and Vehicle Asset Development

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

4 Units in This Section

3.5.1 3D Asset Integration Routes

Textbook: Sec. 3.2.2.1

3.5.2 UE Asset Management

Textbook: Sec. 3.2.2.2

3.5.3 Vehicle XML and Actuators

Textbook: Secs. 3.2.1.2, 3.2.2.3–3.2.2.4

3.5.4 Asset Validation and Reuse

Textbook: Secs. 3.2.1.2, 3.2.3; validation

3.5.3 Vehicle XML and Actuators: Body and Components

X680 Consists of a Body and Four Rotors



Figure: Droneyee X680 model

- **Entry:** 1.BasicExps/ e1_CusLoadDroneyeeX680.
- **Template:** Includes FBX and XML; check runtime assets for the target version.
- **Structure:** body defines the body; ActuatorList registers components.
- **Two routes:** List static components in XML, or index an event-driven Blueprint.

Asset geometry and motion configuration together define vehicle appearance.

3.5.3 Vehicle XML and Actuators: Current Index Formula

Calculate Unique3DClassID from the Current XML Fields

Unique3DClassID = DisplayOrder * 100000 + ClassID

X680: 1015 * 100000 + 3 = 101500003

Check: After loading XML, press F2 and find 101500003 in ModelData.csv.
Call: Put the style index in vehicleType; choose the scene instance ID separately for copterID.
Reuse: Create IDs 100 and 101 from the same style; updating only 100 should leave 101 in place.

The style index selects an asset; the instance ID selects a scene object.

3.5.3 Vehicle XML and Actuators: Configuration Structure

XML Links Model Category, Display Order, and Asset Path

- ClassID and DisplayOrder: Together determine the style index.
- ModelType and MeshPath in body: Select the loading method and asset location.
- Actuator nodes: In the static mesh route, define component positions, axes, and motion modes.
- Example: Check ClassID and DisplayOrder first, then confirm that MeshPath exists.
- Acceptance: After startup scanning, the style index can create the selected model.

Place the configuration in a Content directory the 3D application can scan.

3.5.3 Vehicle XML and Actuators: Styles and Instances

One Style Can Create Multiple Independent Instances

- Register: Scan XML at startup to add the style index to the model library.
- **Create:** Send the style index and initial pose to each new instance ID, 100 and 101.
- Update: Send a new position only to ID 100; ID 101 should remain in place.

- Criterion: After updating only ID 100, the position and attitude of ID 101 remain unchanged.

Instances share an asset, while each ID receives its own position and motion updates.

3.5.3 Vehicle XML and Actuators: X680 Body Configuration

Body XML Locates the Asset and Sets Display References

```
<ClassID>3</ClassID>
<DisplayOrder>1015</DisplayOrder>
<ModelType>0</ModelType>
<MeshPath>/Game/DroneyeeX680/DroneyeeX680Body</MeshPath>
```

Body: MeshPath starts at /Game to locate the static mesh in Content.
Components: ActuatorList lists the rotors, with relative positions and rotation axes.
Scope: Visual scale changes display size; the dynamics model still defines mass, inertia, and thrust.

ModelType selects the loading method; MeshPath locates the asset.

3.5.3 Vehicle XML and Actuators: Actuator Position and Axis

Each Rotor Needs a Position, Axis, and Motion Mode

```
<RelativePosToBodyCm>
  <x>25.4260</x><y>24.0760</y><z>11.2220</z>
</RelativePosToBodyCm>
<RotationModeSpinOrDefect>0</RotationModeSpinOrDefect>
```

Axis: RotationAxisVectorToBody=[0,0,1], about the local Z axis.
Mode 0: Input is in revolutions per minute; position fields are in centimeters.

The three fields answer where it is, which axis it rotates about, and how it moves.

3.5.3 Vehicle XML and Actuators: Motion Modes and Units

The XML Mode Determines What an Input Value Means

XML mode	Input units	Motion
0	Revolutions/min (rpm)	Continuous spin at the specified speed
1	Degrees (deg)	Deflect to the specified angle
2	Stationary mode	Keep the component stationary
3	Centimeters (cm)	Translate by the specified displacement

XML mode 0: 60 rpm = 360 deg/s.
This Blueprint example: 60 deg/s = 10 rpm; the Blueprint event defines the speed.

The same input value represents different physical quantities in different modes.

3.5.3 Vehicle XML and Actuators: Inputs and Visible Motion

Component Configuration Maps Inputs to Rotor Motion



Figure: Inputs drive the displayed rotor motion

- **Channel:** The value reaches the specified actuator.
- **Configuration:** List order, rotation axis, and mode determine the motion.
- **Result:** The selected rotor spins continuously.

Match each input channel to its component, axis, and mode for the intended rotor motion.

3.5.3 Vehicle XML and Actuators: Parent and Child Actuators

Tilting a Motor Changes Its Child Rotor Attitude

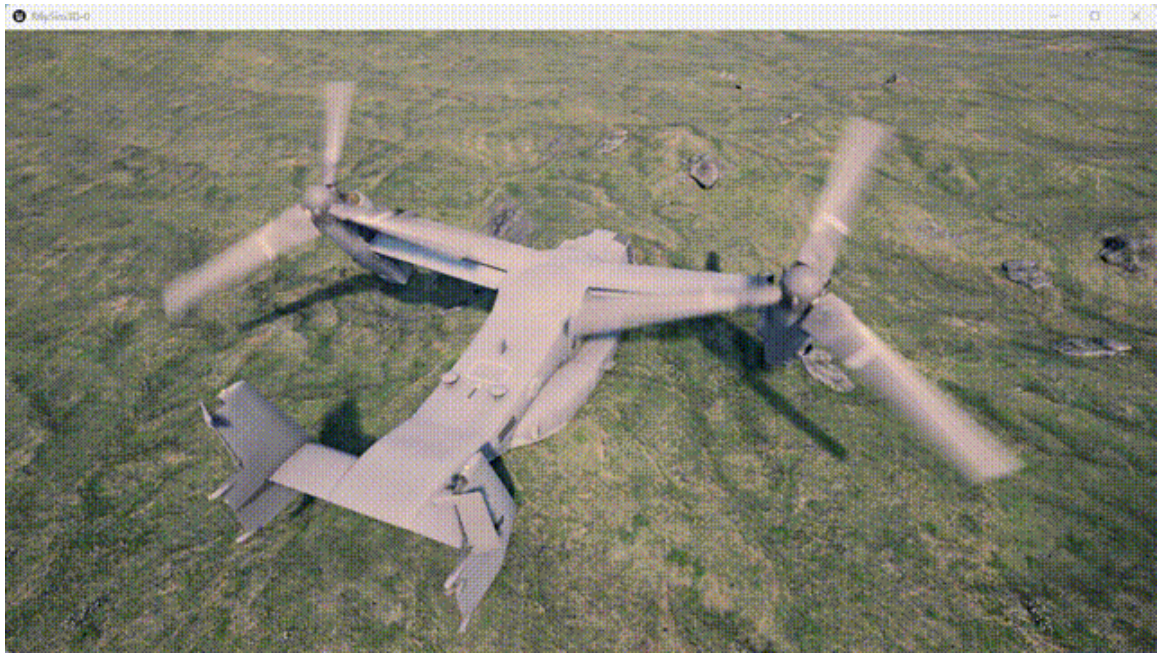


Figure: A tilting motor and child rotor move together

- **Parent entry 2:** Right motor, deflecting about Y.
- **Child entry 4:** Right propeller, attached to parent 2 and spinning about X.
- **Transform:** Parent attitude carries the local coordinate frame of the child.

Parent tilt and child spin can occur simultaneously.

3.5.3 Vehicle XML and Actuators: Parent and Child Actuators

The Child Rotor Still Spins About Its Own Axis After Parent Tilt

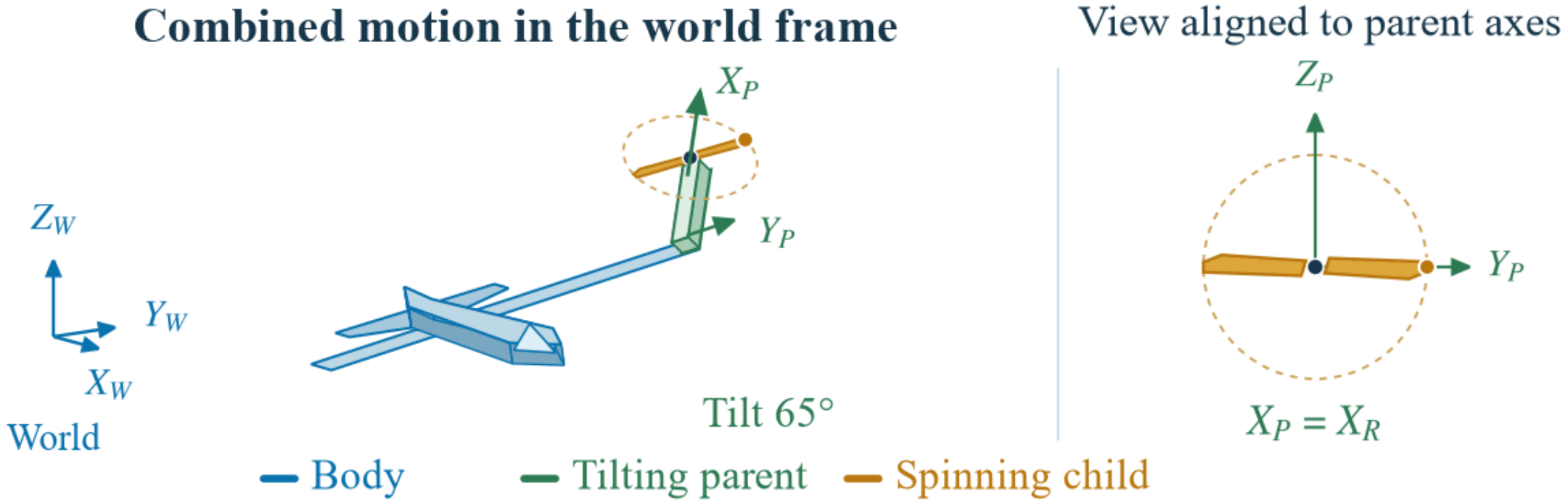


Figure: The motor tilts about the parent axis; the rotor spins about its local axis, which follows the parent.

3.5.3 Vehicle XML and Actuators: Parent Configuration and Channels

Match XML Parent Entry Numbers to Python Array Indices

<RotationModeSpinOrDefect>0</RotationModeSpinOrDefect>
<AttatchToOtherActuator>2</AttatchToOtherActuator>

lis = [0,0,0,100,0,0,0,100]
lis[1] = i

Numbering: XML entries start at 1; Python indices start at 0. Entry 2 → lis[1]; entry 4 → lis[3].

Parent: The right motor is entry 2; lis[1]=i supplies its angle. The child propeller attaches to parent 2.

Child: The right propeller is entry 4; lis[3]=100 holds 100 rpm. It keeps spinning while the parent tilts.

3.5.3 Vehicle XML and Actuators: Binding Experiment

Compare Parent Tilt with Child Spin

- **Entry:** 0.ApiExps/e4_UAVCtrl/1.ActuatorBinding; template QZXY.zip; script ActuatorBindingDemo.py.

- **Prepare:** Create instance 1000 first; in a script copy, fix the other angle entries and retain only lis[1]=i.
- **Compare:** Hold lis[3]=100 while changing the right motor angle; then fix lis[1] and vary only the right propeller speed in lis[3].

Create: Press F2 to find the QZXY style index; create ID 1000 with sendUE4Pos. The script sends only component commands.

Change the parent angle and child speed separately to observe their combined motion.

3.5.3 Vehicle XML and Actuators: Cube Example

Cube Example: From Creating a Blueprint to Checking Speed

Installation root: [InstallationDirectory]\RflySimAPIs\3.RflySim3DUE
Relative path: 0.ApiExps\e0_DevToolsUsage\1.UEBlueprintUsage

Create: Follow Readme.pdf here to make a blank UE4.27.2 project. Create an Actor Blueprint, add StaticMesh, select Cube, compile, and drag it into the level.

Nodes: Connect Event Tick to AddActorLocalRotation with Delta Rotation=(0,0,1); click Play.

Criterion: The cube keeps rotating about its local Z axis. Degrees per second = frame rate × 1°; at 120 FPS, approximately 120°/s.

Troubleshooting: If Cube is missing from Static Mesh, enable Show Engine Content in Content Browser settings.

The cube rotates a fixed 1° per frame, so its angle over a given time depends on frame rate.

3.5.3 Vehicle XML and Actuators: Blueprint Asset Index

The Blueprint Route Encapsulates Motion in the Asset

```
<ClassID>3</ClassID>
<DisplayOrder>1100</DisplayOrder>
<ModelType>2</ModelType>
<MeshPath>
  /Game/Droneyee_WithBP/DroneyeeX680_BP
</MeshPath>
```

Style index: 1100×100000+3=110000003 (original template); ModelType=2 loads a Blueprint. Check for duplicates before deployment. If occupied, change DisplayOrder in an asset copy and recalculate.

XML locates the Blueprint; Blueprint events map inputs to components.

3.5.3 Vehicle XML and Actuators: Blueprint Loops and Debugging

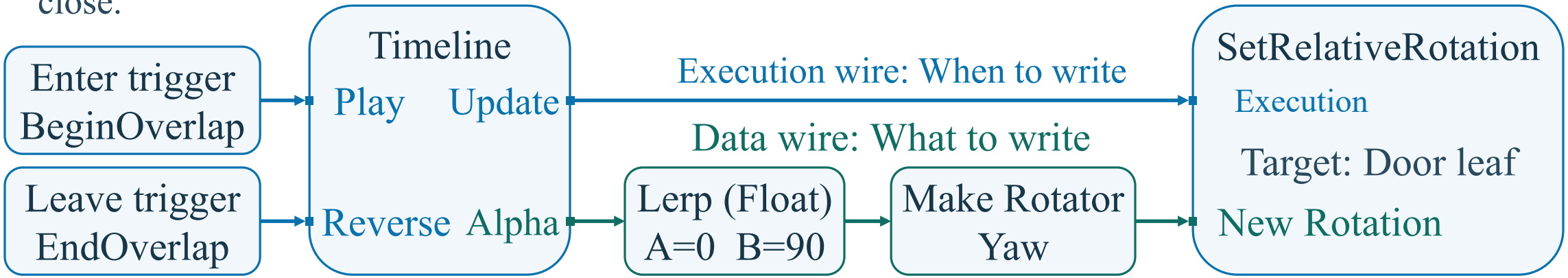
Use Loops for Objects and Breakpoints for Execution Paths

- **Scene:** DoorLeaves contains 3 valid door-leaf components; close them all at BeginPlay.
- **Loop:** Connect BeginPlay to ForEachLoop and DoorLeaves to Array.
  Connect Loop Body to SetRelativeRotation and Array Element to Target; set all three angles to 0.
- **Completion:** Connect Completed to Print String; print "Initialization complete" only once.
  The loop processes a finite set of objects; it does not supply animation progress across frames.
- **Breakpoint:** In Play In Editor (PIE), select the current Blueprint instance and break at SetRelativeRotation.
  Step along the highlighted execution path to check whether this node is reached.
- **Watch:** Use Watch this value for Array Index and Array Element. Indices should be 0, 1, 2, with a different reference each time. No hit: check events and execution wires. Hit but no motion: check Target and angles.

3.5.3 Vehicle XML and Actuators: Blueprint Timelines and Interpolation

A Timeline Gives Progress; Interpolation Gives the Door Angle

- **Assumptions:** A movable door leaf pivots on a vertical hinge. Local Yaw ranges from 0° to 90°; the other axes remain at 0. The trigger responds only to the specified object.
- **Timeline:** A 2 s linear track takes Alpha from 0 to 1. Disable Loop; play forward to open and reverse to close.



- **Example:** After 1 s of playback, Alpha=0.5; Lerp gives 45°, and Update writes it to the door leaf.

$$0° + (90° - 0°) \times 0.5 = 45°$$

3.5.3 Vehicle XML and Actuators: Blueprint Event Chain

Input Events Store Variables; Tick Drives Components Each Frame

- Receive: ActuatorInputs writes each channel input to its corresponding variable.
- **Calculate:** Tick reads the frame interval and calculates an angle increment from each variable.
- Execute: Update the selected component's local rotation; set one channel to zero to check its binding.
- Example: Bind four speeds, 60, 80, 100, and 120 deg/s, to the four rotors.
- Criterion: Setting one input to zero makes only that channel's angle increment zero.

Blueprints encapsulate complex component hierarchies and motion logic within a model asset.

3.5.3 Vehicle XML and Actuators: Input Event

Match Each Input Channel to Its Rotor Component

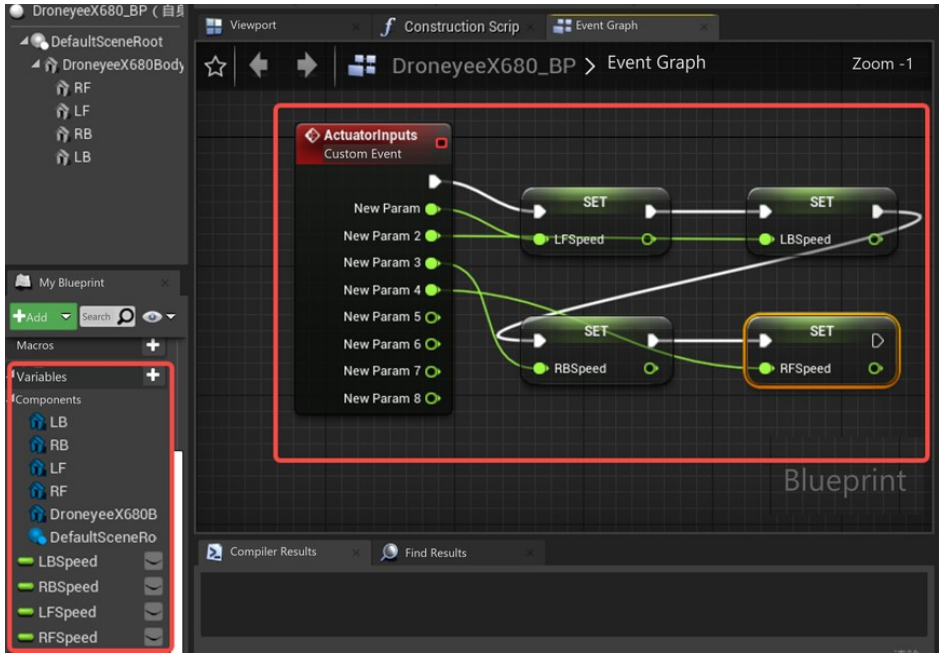


Figure: ActuatorInputs event and variable storage

- **Mapping:** The first four inputs enter the four rotor-speed variables.
- **Single-channel check:** Keep the other entries at zero; change one and observe which component rotates.
- **Model changes:** Check variables, components, and channel order together.

Compare one input at a time with component motion to locate channel-order or binding errors.

3.5.3 Vehicle XML and Actuators: Per-Frame Rotation

Check Blueprint Target Components, Axes, and Units

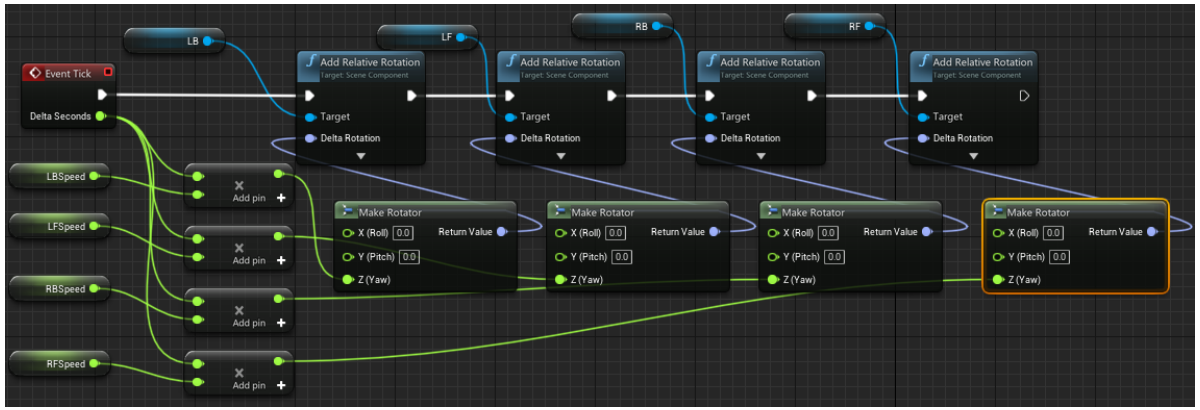


Figure: Complete Tick motion chains for four rotors

- **Target:** Each speed variable feeds its matching rotor branch; the LB branch ultimately updates LB.
- **Local axis:** The angle increment enters Yaw, rotating about that component's local Z axis.
- **Units:** Here, 60 deg/s × 0.02 s = 1.2 deg; XML mode 0 instead interprets 60 rpm as 360 deg/s.

An input number defines motion only together with the asset's channel, axis, and units.

3.5.3 Vehicle XML and Actuators: Blueprint Command

One Command Sets Four Rotor Speeds Separately

RflySetActuatorPWMs 1000 60 80 100 120 0 0 0 0

60 * 0.02 = 1.2

Prerequisite: Press F2 to verify the actual X680 Blueprint style index, then create instance 1000.

Input: The first four values are 60/80/100/120 deg/s; the last four do not set rotor speeds in this example.

Assumption: At a frame interval of 0.02 s, the four increments are 1.2/1.6/2.0/2.4 deg.

Stop: Set the first four values to 0; subsequent Tick angle increments are 0.

3.5.3 Vehicle XML and Actuators: X680 Example

X680 Example: Set Four Rotor Speeds with a Console Command

Installation root: [InstallationDirectory]\RflySimAPIs\3.RflySim3DUE
Relative path: 2.AdvExps\e2_BlueprintModel\1.BlueprintBuild

Deploy: Unpack 1.BlueprintBuild.zip and its Droneyee_WithBP_cooked_UE4.zip; check for duplicate indices as explained in this unit, then run Copy.bat.

Create: Open RflySim3D, double-click the ground, then press O and 3 to create a quadcopter. Press C to select DroneyeeX680_WithBP; S shows IDs starting at 1000.

Command: Press ~ to open the console, then enter RflySetActuatorPWMs 1000 60 80 100 120 0 0 0 0

Criterion: LF, LB, RB, RF rotate at about 60, 80, 100, 120°/s, independent of frame rate. For LB at 60 FPS, the per-frame angle is about $80/60 \approx 1.33°$.

The external command supplies angular speeds; the Blueprint calculates per-frame rotor angles from the frame interval.

3.5.3 Vehicle XML and Actuators: Blueprint Frame Updates

Tick Multiplies Speed by Frame Interval and Accumulates Angle

- Input: The four speeds here are 60, 80, 100, and 120 deg/s in order.
- **One frame:** Over 0.02 s, the angle increments are 1.2, 1.6, 2.0, and 2.4 deg.
- Runtime: Accumulate each frame; setting one input to zero stops that rotor at its current angle.

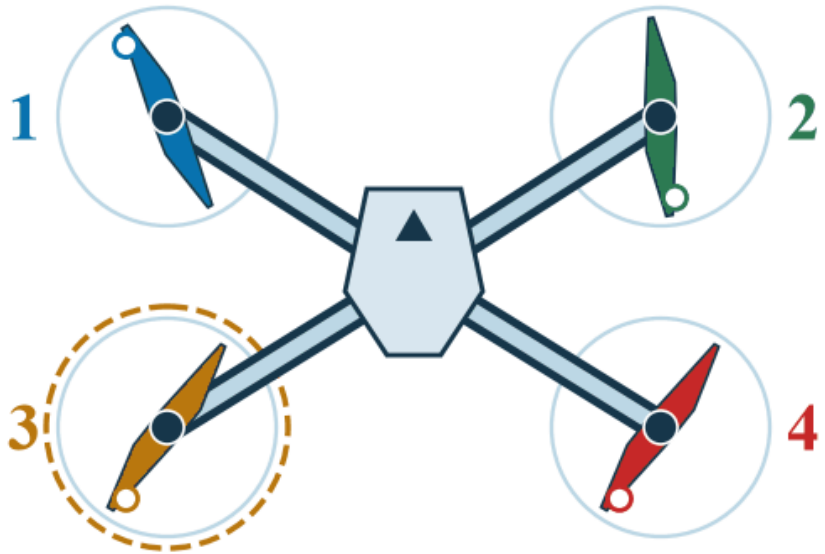
$$\Delta\theta = \omega\Delta t$$

The Blueprint defines input units; do not assume the speed units of XML mode 0.

3.5.3 Vehicle XML and Actuators: Tick Process Illustration

Four Inputs Accumulate Different Angles over the Same Frame Interval

$\Delta\theta = \omega \times 0.02\ \text{s}$ Step 400 Model time 8.00 s



Component local angles

Input	ω (°/s)	Δθ (°)	Total (°)
1	60	1.2	480.0
2	80	1.6	640.0
3	**0**	**0.0**	**600.0**
4	120	2.4	960.0

Input 3 zero: angle held

Figure: Blueprint Tick accumulates angles frame by frame (original teaching schematic)

3.5.3 Vehicle XML and Actuators: Extended Inputs and Versions

An Extended Event Maps Inputs to Rotor Offsets

```
cmd = 'RflySetActuatorPWMsExt 1000 0 0 20 ' + '0 '*13
ue.sendUE4Cmd(cmd)
```

Event: ActuatorInputsExt receives 16 floating-point values in this example.
Mapping: The first 3 are XYZ offsets; [0,0,20] adds 20 cm to local Z for all four rotors.
Version: Full edition; use the UE4.27.2 build or assets cooked again for UE5.8.

ActuatorInputsExt defines XYZ offsets in this example's Blueprint.

3.5.3 Vehicle XML and Actuators: Blueprint Applications

Model Complex Actuators, Hit Events, and Effects Separately

- Complex components: Organize landing gear, control surfaces, and tilt mechanisms through hierarchies and events.
- **Interaction:** Hit detection changes model state; record its trigger conditions.
- Visual effects: Particles or materials show explosions and smoke; they do not replace a damage model.
- Trigger: Identify the object by instance ID and hit component; record the event time.
- Acceptance: Visual effects must be reproducible; damage quantities require separate model outputs.

The asset Blueprint defines motion entry points; visual effects are not physical results.

3.5.3 Vehicle XML and Actuators: Choosing XML or Blueprints

One Vehicle Can Use Two Ways to Organize Motion

- **XML static meshes:** ActuatorList organizes components; positions, axes, parents, and modes can be read and edited directly.
- **Blueprint asset:** ActuatorInputs receives inputs; variables and Tick organize motion; XML indexes the Blueprint.
- **Choose:** Start with existing XML for regular components; use the matching Blueprint for custom events and its defined input units.

Entry: 2.AdvExps/e2_BlueprintModel/1.BlueprintBuild.

UE4 deployment: Unpack 1.BlueprintBuild.zip, then Droneyee_WithBP_cooked_UE4.zip. Check for duplicate indices, then copy Droneyee_WithBP to RflySim3D/RflySim3D/Content.

Use XML for regular components; encapsulate custom motion in Blueprint events.

Chapter Outline

Scene and Vehicle Asset Development

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

4 Units in This Section

3.5.1 3D Asset Integration Routes

Textbook: Sec. 3.2.2.1

3.5.2 UE Asset Management

Textbook: Sec. 3.2.2.2

3.5.3 Vehicle XML and Actuators

Textbook: Secs. 3.2.1.2, 3.2.2.3–3.2.2.4

3.5.4 Asset Validation and Reuse

Textbook: Secs. 3.2.1.2, 3.2.3; validation

3.5.4 Asset Validation and Reuse: Vehicle Types

Different Vehicles Reuse the Same Asset Interface



- **Shared:** Style index, instance ID, position, and attitude.
- **Different:** Rotor spin, control-surface deflection, and tilt hierarchies.
- **Dynamics:** The corresponding model supplies vehicle dynamics.

Figure 3.8(b) Vehicle Models

Visual assets can be reused; component mechanisms remain specific to each vehicle.

3.5.4 Asset Validation and Reuse: Geographic Scene Integration

Large Scenes Combine Geographic Assets with Simulation State



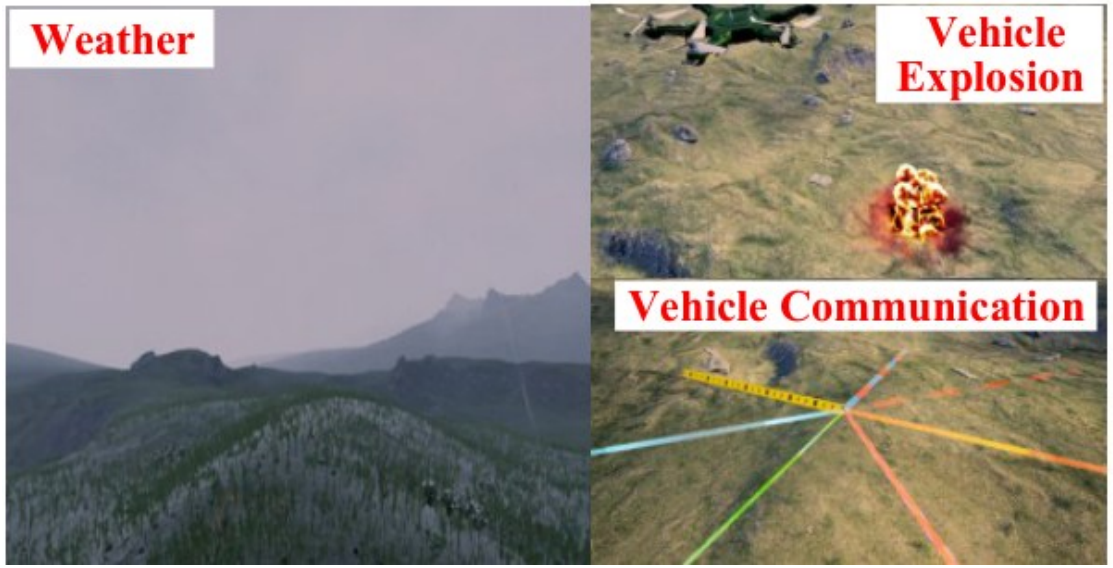
Figure 3.8(c) Large-Scale Scene Models

- **Environment:** A 3D reconstruction of Hong Kong provides geographic geometry.
- **Coordinates:** The map origin connects global positions to local simulation state.
- **Integration:** Send the same vehicle state to both the ground station and visualization.

The map supplies the environment; state links drive the vehicle display and ground station.

3.5.4 Asset Validation and Reuse: Effects and Data Links

Weather, Explosions, and Communication Lines Represent Different Events



- **Weather:** Parameters change environmental appearance.
- **Explosion:** An event triggers particles or animation.
- **Communication:** Lines show relationships; a communication model supplies statistics.

Figure 3.8(d) Common Visual Effect Models

Parameters or events drive visual effects; data models supply statistics.

3.5.4 Asset Validation and Reuse: Troubleshooting

Trace a Missing Object to Its Dependencies

- **Unregistered XML asset:** XML location, ClassID, and DisplayOrder determine the style entry that can be found.
- **Registered but missing geometry/textures:** MeshPath, target version, and cooked dependencies determine the loading result.
- **Geometry visible but components stationary:** Instance ID, input channels, modes, or Blueprint events determine motion.

Reuse check: Create an instance and change its pose, exit, reload the same asset, and create again. Geometry, materials, and motion should match the saved configuration.

Locate loading problems through registration, paths, and state.

3.5.4 Asset Validation and Reuse: Delivery and Practice

Deliver an Asset Package You Can Explain

- **Basic task:** Choose one import route; deliver asset files, configuration, the style index, and an instance-creation call.
- **Mechanism question:** Why do input 60 in XML mode 0 and input 60 in this Blueprint produce different speeds?
- **Comparison:** Hold child speed fixed and vary the parent angle; record both motions, then create and independently update a second instance.

Submit: State the units and converted angular speeds for both inputs of 60. Use instance IDs to explain whether updating the first object changes the second.

Align assets, configuration, and calls so motion can be reused reliably.

Practice Reference Points

Explain Three Common Faults with Indices, Units, and Instance IDs

Missing model (XML route):
First check startup XML scanning, then ClassID, DisplayOrder, MeshPath, and cooked dependencies.

Wrong speed:
XML mode 0: 60 rpm = 360 deg/s.
This Blueprint: Input 60 = 60 deg/s.

Wrong object updated:
A pose message to instance 100 should update only 100; instance 101 keeps its position and attitude.

3.5.4 Asset Validation and Reuse: Learning Resources

Look Up XML Fields, Style Indices, and Asset Categories

XML fields, motion modes, and style indices

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

Available scene and model asset categories

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

Consult documentation for the specific learning task, then return to the selected model.

Section Summary

Choose an Asset Route, Then Load and Validate the Assets

- **UE editor:** Check assets and dependencies, then cook and deploy; register vehicle XML as needed.
- **Runtime GLB:** Check the parsing plugin, model, and textures; create instances from repository entries.
- **GIS data:** Check tile format, coordinate reference, coverage, and service availability.
- **Shared settings:** Check units, axes, origins, and appearance; record application and asset versions.
- **Runtime validation:** Check reuse after restarting; for vehicles, check pose, components, and collision by instance ID.

Successful loading is a starting point; assets must also be reusable and reliable in the target task.