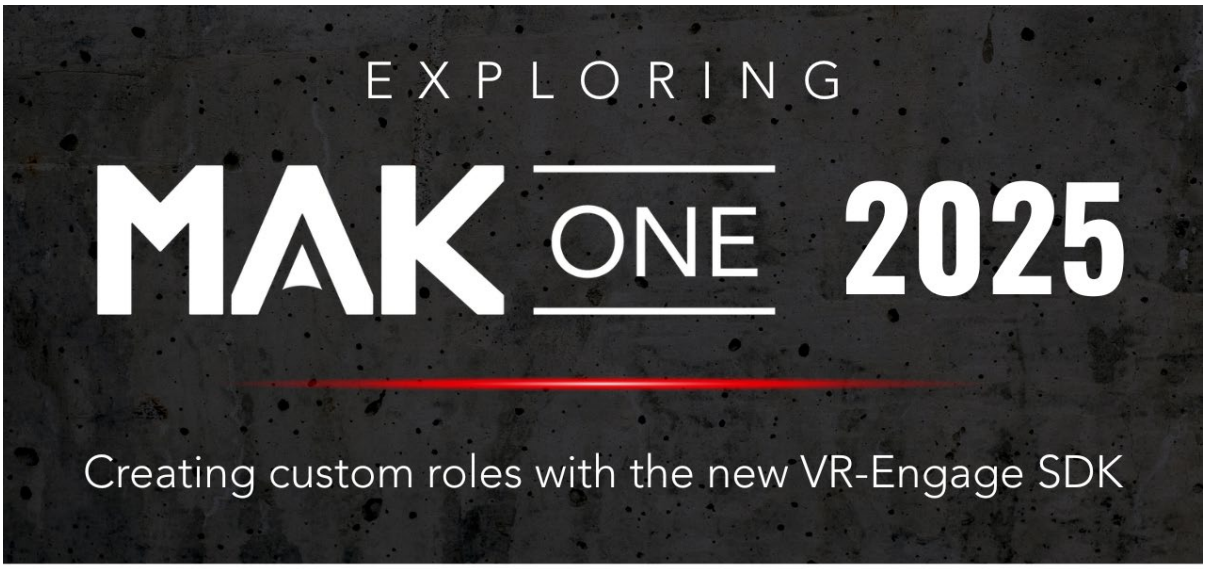

Creating Custom Roles with the new VR-Engage SDK

Let's say you're a solutions engineer and you've been tasked with creating a helicopter door gunner simulator. You've been using VR-Engage in many roles (helicopter pilot, LAV driver, UAV sensor operator), but there is no Door Gunner role for helicopters and you're unsure where to start actually building it. You know how to use the existing roles within VR-Engage and have an idea about how VR-Engage uses a big library of components that make up those roles... but putting it all together to create this custom role is daunting.

Alright pause. Before you freak out about creating this new role for players in your scenario, you should know that **the new VR-Engage SDK in MAK ONE 2025 is about to make your life a lot easier**.

Real quick, let's start with a review of components in VR-Engage. Basically, a role is a configuration of different components - and those components can be anything from vehicles to weapon systems to turn signals. These are the pieces that, when combined, come together to build a role for your first-person simulation.

So back to your task. You need to make a helicopter door gunner role. For that, you'll need a helicopter model (already a component within VR-Engage, thank you RTDynamics) and a gunner system (clue: you can borrow this from an existing ground vehicle role). Here's how to actually make this happen:

Understanding the Component Architecture

These components are defined within Dynamically Linked Libraries (DLLs), which are essentially plug-ins that VR-Engage loads when it runs. When you start VR-Engage, it loads the available components from the plugins folder. Each component has its own set of

parameters that you can adjust to suit your specific scenario and role needs. These parameters are configured in Lua files and loaded at runtime.

Think of it like this: VR-Engage provides a library of LEGO blocks. Your job is to assemble these blocks into a door gunner role.

The VR-Engage Player interface is organized into several broad categories:

Vehicle Control Components - Observer positioning and camera management - Vehicle movement and navigation controllers - Environmental interaction systems

Weapon & Targeting Systems - Weapon control logic and firing systems - Targeting and sensor interfaces - Ammunition management

Player Interface Components - Input device handlers (joystick, keyboard, VR controllers) - Qt Quick UI overlays and HUDs - Audio systems and voice communications - Menu and dialog systems

Data & Communication Components - Network protocol handlers - Entity state synchronization - Mission data and scenario management

This modular approach means you can mix and match front-end components to create exactly the player experience you need - whether that's a submarine sonar operator, tank commander, or helicopter door gunner.

Step 1: Identify Your Required Components

For a helicopter door gunner role, you'll need these key components:

- **RTDynamics helicopter model** (already available as `DtRTDVehicleComponent`)
- **Weapon system logic** (borrow `DtWeaponResourceLogic` from existing ground vehicle gunner roles)
- **Observer/camera positioning** (use `DtGunnerObserverUpdater` to position the player's viewpoint)
- **Input handling** (configure `DtGunnerControlLogic` with appropriate input mappings)
- **UI overlay** (adapt the existing gunner HUD using `DtQtQuickOverlay`)

Step 2: Create Your Role Configuration Script

Next, write a simple Lua script to assemble these components into a coherent role. The Lua code acts as the glue between all the connections of your new role. It directs which components will be instantiated when you engage into the role at runtime. You'll create a file like `helicopterDoorGunner.lua` in your roles directory of your simulation model set (SMS):

```lua
-- Define the application modes this role supports
-- "Air" indicates this role is for aircraft entities
appModes = {"Air"};
```

```
-- Define the role names that this configuration supports
-- These must match the role names in your entity's vrengage-station-names
roles = {"Door Gunner"};

-- Inherit from base gunner role to get core gunner functionality
-- This gives us weapon controls, targeting systems, and basic UI
inherits = "@(vre-roles-dir)/gunner.lua";

-- Configure the components that make up this role
components = {
   -- Weapon system component handles ammunition, firing, and weapon state
   ["weaponLogic"] = {
      componentType = "DtWeaponResourceLogic";
      priority = 5; -- Lower numbers execute first in the update cycle

      -- Map entity types to weapon systems
      -- "2:8:225:2:5:1:0" is the DIS entity type for 7.62mm machine gun
      resourceMappings = {
         ["2:8:225:2:5:1:0"] = {shortName = "M60"}
      };
   };

   -- Observer component controls camera positioning and viewpoint
   ["observerUpdater"] = {
      componentType = "DtGunnerObserverUpdater";
      priority = 10; -- Execute after weapon logic is established

      -- Attach to articulated part 4416 (primary gun mount on helicopter)
      attachPart = 4416;

      -- Position offset from attach point (forward, right, down in meters)
      -- This places the camera at the door gunner position
      attachOffset = {x = 0.88, y = -1.8, z = 1.26};

      -- Orientation offset as quaternion (x, y, z, w)
      -- Identity quaternion means no rotation from mount orientation
      oriOffset = {x = 0.0, y = 0.0, z = 0.0, w = 1.0};
   };

   -- Control logic handles input device mapping and player controls
   ["controlLogic"] = {
      componentType = "DtGunnerControlLogic";
      priority = 5; -- Same priority as weapon logic for coordinated updates

      -- Reference to input configuration file for this role
      -- Defines joystick/keyboard mappings for helicopter door gunner
      inputConfigFile = "helicopterGunnerInput.lua";
```

```
    };
};
```

This script inherits base gunner functionality and adapts it for helicopter use. The `attachPart` and `attachOffset` parameters position the gunner's viewpoint at the helicopter door, while `resourceMappings` connects to the appropriate weapon system. Component priorities ensure proper initialization order - weapon systems load first, then observer positioning, then control mappings.

Step 3: Configure Entity File Parameters

This is where you'll associate your new role with a particular simulated entity. In your entity file, add the door gunner role:

```
<vrEngageRoles paramName="vr-engage-roles">
   <role paramName="Pilot">
      <string paramName="role-script">helicopterPilot.lua</string>
   </role>
   <role paramName="Door Gunner">
      <string paramName="role-script">helicopterDoorGunner.lua</string>
   </role>
</vrEngageRoles>
```

Weapon System Integration

The door gunner role requires a weapon system component that can handle the M60 machine gun. VR-Engage includes a number of weapon systems out of the box which can adapted for a variety of weapons (rifles, hand guns, cannons, missiles, etc.). Add a weapon system to your entity configuration:

```
      <componentSystem systemName="weapon" platform="@(system-
dir)\weapons\vre-M60-7_62mm-mach-gun.sysdef">
         <int paramName="gun-art-part-type">4416</int>
         <int paramName="num-rounds">100</int>
         <int paramName="rapid-fire-rate">80</int>
         <int paramName="sustained-rate">40</int>
         <int paramName="turret-art-part-type">4096</int>
         <string paramName="ballistic-gun-group">weapon:M60-7-62mm Ballistic
Gun</string>
         <string paramName="display-name">M60 Machine Gun</string>
      </componentSystem>
```

This weapon system connects to the articulated part we defined earlier and references an ammunition table that maps the DIS entity types to specific ammunition loads.

You'll also need to ensure your helicopter entity file includes the proper articulated parts for weapon mounting and role support. Configure the weapon mount points:

```
<DtSimArtPartDescriptor paramName="primary-gun-4416">
   <bodyPosition paramName="attach-point" forward="0.88" right="-1.8"
```

```
down="1.26"/>
   <int paramName="art-part-type">4416</int>
   <!-- Additional weapon system parameters -->
</DtSimArtPartDescriptor>
```

Input Configuration

Create a separate input configuration file (`helicopterGunnerInput.lua`) to define how player controls map to weapon functions.

```
-- Input configuration for helicopter door gunner role
inputDevices = {
    -- Gamepad controls
    {
        DeviceType = "gamepad",
        Mappings = {
            {type="axis"; id=5; action="fire"};
            {type="axis"; id=3; action="turret-elevation";};
            {type="axis"; id=2; action="turret-azimuth";};
        }
    },

    -- Mouse controls
    {
        DeviceType = "mouse",
        Mappings = {
            {type="button"; id=0; action="fire"};
            {type="motion"; axis="x"; action="turret-azimuth";};
            {type="motion"; axis="y"; action="turret-elevation";};
        }
    },

    -- Keyboard controls
    {
        DeviceType = "keyboard",
        Mappings = {
            {type="key"; key="F"; action="fire"};
        }
    }
};
```

This input configuration file is referenced by the `DtGunnerControlLogic` component in your role script (as shown in Step 2 with `inputConfigFile = "helicopterGunnerInput.lua"`). The control logic component reads these mappings and automatically connects them to the weapon system's input ports.

When a player moves the mouse, the mouse motion mappings translate those movements into `turret-azimuth` and `turret-elevation` commands that are sent to the weapon

controller. Similarly, when the gamepad axis is moved or the mouse button is pressed, the `fire` action mapping sends a firing command to the weapon system.

The input system works through a straightforward message flow: your input configuration maps physical device actions (joystick buttons, mouse movement, keyboard presses) to logical commands ("fire weapon", "aim left", "reload"). The `DtGunnerControlLogic` component converts these device inputs into standardized simulation messages that flow through VR-Forces' component communication system. These messages are then received by the appropriate simulation components - weapon controllers receive firing commands, observer components receive aiming commands, and so on.

This approach keeps your role configuration simple while ensuring that complex input handling, device compatibility, and simulation message routing are handled automatically by the underlying VR-Forces infrastructure.

Customizing the User Interface

Beyond simulated behavior, you might want to customize the user interface for your door gunner role. VR-Engage uses Qt Quick/QML for its UI overlay system, making it straightforward to create custom interfaces. For example, to add a specialized ammunition counter and targeting reticle for your door gunner:

```
// helicopterGunnerHUD.qml
import QtQuick 2.15

Rectangle {
    anchors.fill: parent
    color: "transparent"

    // Ammunition display
    Rectangle {
        anchors.top: parent.top
        anchors.right: parent.right
        anchors.margins: 20
        // ... styling properties ...

        Text {
            text: "AMMO: " + ammoCount
            color: "#00ff00"
        }
    }

    // Custom targeting reticle
    Item {
        anchors.centerIn: parent
        // ... crosshair rectangles ...
    }
```

```
    // Data bindings to component
    property int ammoCount: gunnerComponent.currentAmmo
    property string ammoType: gunnerComponent.weaponType
}
```

Then reference this custom QML file in your Lua role configuration:

```
components = {

displayLayouts ={
   ["1 Screen Horizontal"] = {
      ["customHUD"] = {
         componentType = "DtQtQuickOverlay";
         priority = 15;
         qmlFile = "helicopterGunnerHUD.qml";
         overlayName = "GunnerHUD";
      };
   };
};

   -- ... existing components ...
```

Step 5: Test and Iterate

Once you've written your Lua configuration script, place it in your VR-Engage roles directory and test it in a scenario. The great thing about this approach is that you can rapidly iterate - adjust viewpoint positions, weapon parameters, and input mappings by simply editing the Lua files.

Common adjustments include fine-tuning the observer position for realistic door gunner viewpoints, configuring appropriate weapon elevation and azimuth limits for helicopter door mounts, and ensuring proper ammunition handling.

Going Further with Custom Components

Let's take it a step further now and say that you've been tasked with extending a ground vehicle's functionality – for example, adding a turn signal system. This is functionality that doesn't exist out of the box in VR-Engage. That's where the new VR-Engage C++ API comes in. You can write C++ code to extend our library of input controllers, plug them in, and configure them into your new role.

If the existing components don't meet your needs, you can develop custom ones using the VR-Engage C++ API. For example, to create a custom turn signal system for your ground vehicle, you'd need to work on both sides of the VR-Engage architecture:

Front-end Player Component:

1. **Inherit from `DtPlayerComponent`** to create your custom UI and input handling component

2. **Implement the required interfaces** for initialization, update cycles, and cleanup

3. **Define custom data messages** for communicating turn signal state to the simulation engine

Back-end Simulation Component:

1. **Extend the entity's simulation behavior** by adding turn signal logic to the sim engine

2. **Modify entity state data** to include turn signal status in the data exchanged between front-end and back-end

3. **Compile both components into DLLs** and place them in the plugins directory

4. **Register your components** in your Lua role configuration

This will get your turn signals working correctly in the simulation (affecting AI behavior, network traffic, visual models) while providing intuitive player controls in the VR-Engage interface. The SDK provides the means for message passing between your front-end player controls and back-end simulation logic. And any new state data can be sent from the simulation to the front-end and reflected in a custom UI for the role.

And at a high level, that's it.

You can rest easy because the VR-Engage SDK comes with examples and documentation so you aren't starting from scratch when you're building your new roles. The examples directory contains over a dozen working implementations covering everything from vehicle blinkers to custom projectile systems - perfect references for your door gunner development.

Hope you enjoyed this walkthrough of creating custom roles within VR-Engage using the new SDK coming with the MAK ONE 2025 release!

(Oh and just so you know, VR-Engage comes with ground models [including many 10- and 12-wheeled vehicles - new with this release] for you to experiment with!)

Learn more about this and the rest of what's coming in MAK ONE 2025 at the MAK ONE 2025 Release Webinar on September 16, presented by Jim Kogler, VP of Products. Register for the session that best fits your schedule: 10:30AM or 9:00PM EDT.

All registrants will receive a link to watch the recording after the webinar!