AN2503 – Getting started with CIGI

January 2025

Battlespace Simulations, Inc.

8305 Catamaran Circle

Lakewood Ranch, FL 34202

If you have questions or comments, please contact us at support@bssim.com.

Overview

CIGI is a powerful toolset you can use in your MACE plugins, codescripts, or commands to draw custom information in any CIGI compliant IG like ARMOR. This document will give a brief overview of CIGI and some basic guidelines for its use with MACE and ARMOR.

What is CIGI?

The Common Image Generator Interface (CIGI) protocol is an open, standardized communication interface widely used in the simulation and training industries. It enables interoperability between image generators (IGs) and host applications, such as flight simulators, driving simulators, or military training systems.

Key Features of the CIGI Protocol: 1. **Real-Time Communication:**

CIGI facilitates high-performance, real-time communication between the host and the image generator. It is optimized for low-latency operations, ensuring smooth simulation experiences.

2. **Flexible Data Exchange:**

Supports a wide range of data types, including positional data, environmental settings, camera configurations, and entity updates. Data is exchanged using structured packets, allowing precise control over various aspects of the simulation.

3. **Extensibility:**

Designed to accommodate future requirements with backward-compatible updates. Supports the addition of new features and packet types as simulation needs evolve.

4. **Scene Management:**

The host can control the virtual environment by defining the placement and behavior of entities (e.g., vehicles, characters, terrain). Commands can adjust visibility, animations, and environmental conditions like weather and lighting.

5. **Camera and Sensor Control:**

Enables the host to control camera parameters, such as position, orientation, field of view, and clipping planes. Supports advanced sensor modeling for applications requiring simulated camera or sensor outputs (e.g., infrared or night vision).

6. **Event Handling:**

Allows the host to send event triggers or request information from the image generator, such as collision detection or entity status updates.

7. **Standardized Protocol:**

Provides a consistent interface that simplifies integration of image generators from different vendors. Reduces development time and costs by providing a shared standard. Multiplatform Support:

The protocol can be implemented across various platforms and hardware configurations, promoting versatility and scalability.

CIGI grants a great deal of control, and there are far more configurations and options available than can be reasonably conveyed in a quick note. CIGI is a SISO standard; please refer to published standard for a full overview of what CIGI can do. The CIGI standard is published in SISO-STD-013-2014, and you can download from https://www.sisostandards.org/page/StandardsProducts.

Getting Started w/ MACE and ARMOR

The MACE API has support for sending CIGI control packets to a CIGI compliant IG like ARMOR. You will find CIGI capabilities exposed on via the `ICIGINetwork` interface, exposed via `IMission.CIGINetwork`.

The code samples in this app note come from the `DualViewCigiDraw.cs` codescript. It illustrates how to draw text to both the primary (Display) and secondary (Sensor) views in the ARMOR IG.

A basic example

In order to draw things in the IG, you first create a "view", then create a "surface" (or surfaces) associated w/ the view, then define "symbols" (shapes or text) to draw on a particular surface.

In the example codescript, a single function handles drawing user submitted text to a specified view. That function is summarized below.

```
public static bool DrawStringToView(ushort viewID, string displayText)
{
    bool result = false;
    ushort surfaceID = viewID;

    double FOV = 30; // 30 degree field-of-view

    ICIGINetwork.ViewParameters.ViewTypes spectrum =
ICIGINetwork.ViewParameters.ViewTypes.OTW;

    ICIGINetwork.ViewParameters parameters = new
ICIGINetwork.ViewParameters();
```

```
        parameters.type = (int)spectrum;
        parameters.fovLeft_deg = -(FOV / 2);
        parameters.fovRight_deg = (FOV / 2);
        parameters.fovTop_deg = (FOV / 2);
        parameters.fovBottom_deg = -(FOV / 2);

        _mission.CIGINetwork.CreateView(viewID, parameters);

        if (_mission.CIGINetwork.PositionViewRelativeTo(viewID,
    _mission.Map.SelectedEntity, new Vector3D(0.5, 0, -2), new RPY(0, 0, 0)))
        {
            ICIGINetwork.SurfaceParameters surfaceParameter = new
    ICIGINetwork.SurfaceParameters();
            surfaceParameter.windowWidth_pixels = 1280;
            surfaceParameter.windowHeight_pixels = 1024;
            surfaceParameter.surfaceAttachmentType =
    ICIGINetwork.SurfaceParameters.SurfaceAttachmentTypes.View;

            _mission.CIGINetwork.CreateSurface(surfaceID, surfaceParameter);
        }

        ushort symbolID = viewID;
        result |= _mission.CIGINetwork.SendOverlayText(
                        surfaceID: surfaceID,
                        symbolID: symbolID,
                        color: 0xffff0000,
                        textString: displayText,
                        positionX: 200,
                        positionY: 200,
                        fontSize: 30,
                        fontFamily:
    CIGISymbolTextStructure.FontEnum.MonospaceSansSerifBold,
                        alignment:
    CIGISymbolTextStructure.AlignmentEnums.Center
                        );
        return result;
    }
```

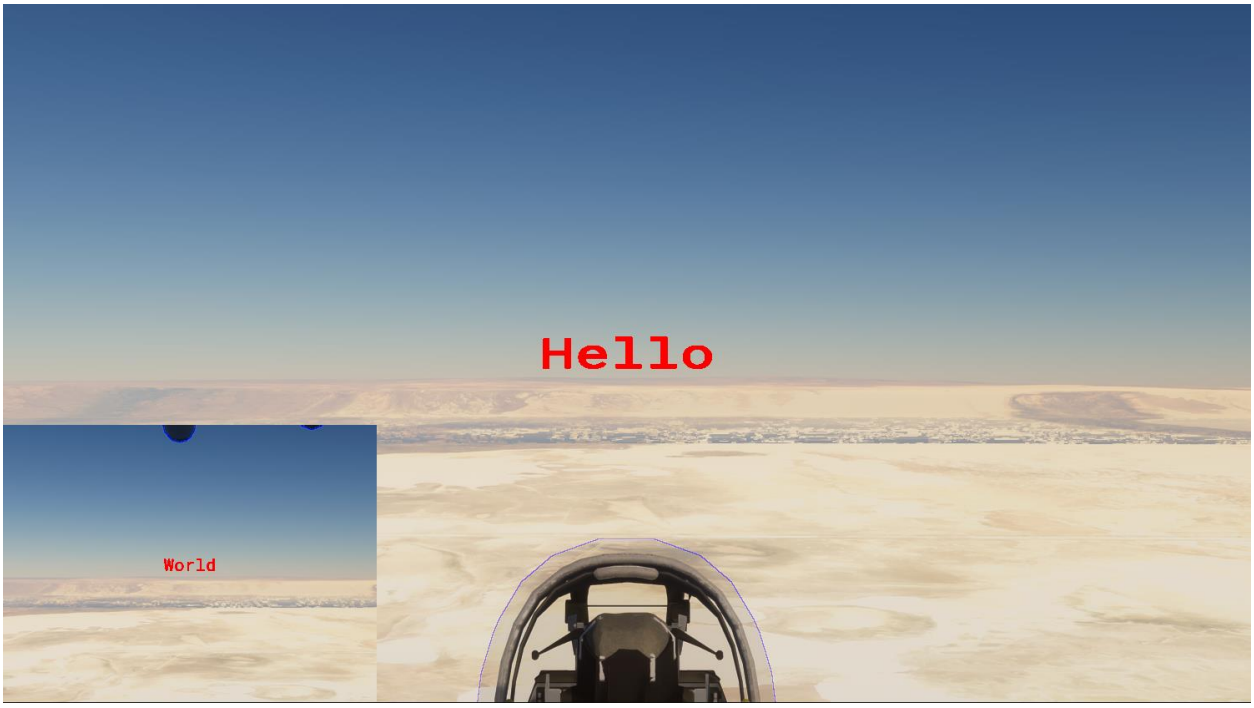
In the sample code, a `ViewParameters` object is defined that outlines how the view should appear, specifically its field of view (and therefore its "zoom" level) and its spectrum. Those view parameters are used to create the view, and the view is then positioned relative to the selected entity in MACE. Notice that the view is offset from the default from eyepoint and given an orientation (pitch, roll, and yaw angles for the view at the offset eyepoint). We then define a new surface or a particular resolution and give it a "View" surface type. The view surface type means the surface exists in the coordinate space of the view, meaning

objects attached to it are placed relative to the camera. The requested text is then written to the surface within the view, using some user defined parameters to set up its appearance and position.

The `DrawStringToView()` method can be called for two separate views. Using ARMOR defaul view IDs, we can send the text "Hello" to the main display (the "out the window" view, default view ID 1) and "World" to the sensor display (secondary camera view, default view ID 2) by calling it sequentially for each view:

```
DrawStringToView(1, "Hello");
DrawStringToView(2, "World");
```

Doing so will draw the following – in this case for an A-10C as the selected entity.



NOTE: There is presently a limitation in the MACE API requiring the view ID and surface ID to match. That limitation will be removed in the MACE2025R1 release cycle.

What next?

If you need to work with CIGI, we highly encourage you to familiarize yourself w/ the CIGI specification. In example to the example codescript discussed in this App Note, you can also find a more full featured CIGI example plugin on the BSI downloads site, https://downloads.bssim.com/file/d/MACE_Development/examples/plugins/CigiDeviceDemo_6Aug2020.zip.