AN2505 – CIGI Symbol Drawing

March 2025

Battlespace Simulations

8305 Catamaran Circle

Lakewood Ranch, FL 34202

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

Overview

AN2505 – CIGI Symbol Drawing

This is a guide to creating MACE Plugins that can draw CIGI symbols in ARMOR, using the new Symbols Library available in MACE 2025R1. The main use cases of creating these plugins will be creating new Heads-Up-Displays (HUDs), On-Screen Displays (OSDs), Sensor Overlays, or overlays specific to Emulated Military Equipment (EME).

Contents

Library Basics

Intro

The Symbols library manages all aspects of drawing symbols over CIGI. It uses CIGI 4.0, and any IG should be compatible. However, we have developed this with ARMOR in mind.

Compliant with CIGI, Symbols are drawn on a **Surface**. Surfaces can either be attached to a **CIGI View**, or attached to an **entity**, specified by a View ID or an Entity ID respectively. Surfaces attached to Views are displayed as overlays, and Surfaces attached to Entities are positioned in world-space. Entity attached Surfaces can also optionally be billboarded, or set to appear with a fixed size. View attached Surfaces can be sized with either Fixed, Fit, or Expand. The IntelliSense comments describe Surface properties in detail.

At the loweset level, each symbol created must have a parent. Parents of symbols can be a Surface, an invisible anchor symbol (type `Symbol` [see below]), or another symbol. Top level symbols should be children of the Surface. The parent is passed into the symbol constructor. Calling the constructor of a symbol creates the symbol and adds it to the symbol system, so no code is required to add the new symbol to a collection or manage them directly during the lifetime of the overlay. They will be updated as required by the symbol system when the underlying symbol changes.

A `Symbol` itself is a blank, non-rendered item which can be used to help manage the symbol hierarchy. (i.e. defining a `Symbol` uses an ID and costs a little in network transmission and management, but has no meshing or rendering cost). `Symbols` are commonly used with a

collection of child symbols each with relative positions and/or rotations, and then the whole group can be manipulated by simply adjusting the parent symbol.

Setup Code

> [!WARNING] This example code assumes familiarity with MACE plugins, and also assumes that a MACE plugin has already been created and configured correctly.

First, a `SymbolManager` needs creating at plugin initialization:

```
_manager = SymbolSystem.Instance.CreateManager(1000);
```

Store a reference to this manager as a field, as we will need to access it throughout.

To create symbols, we need to first create the surface, passing the manager in:

```
_surface = new Surface(_manager);
```

Specify the properties of the surface, either using View or Entity attachment:

```
// View attach
_surface.AttachType = SurfaceAttachType.ViewAttached;
_surface.SizingMode = SurfaceSizingMode.Fit;
_surface.ViewID = 1;

// Entity attach
_surface.AttachType = SurfaceAttachType.EntityAttached;
_surface.EntityID = 0; // TODO get Entity ID
// Optional
_surface.Billboard = false;
_surface.PerspectiveGrowth = false;
```

Symbol Creation

Now, symbols can be created.

It is good practice to first define some screen anchors, using homogeneous positions. Here, we define 9 different anchors around the screen-space (as top level symbols, with the surface as the parent), although in practice only the anchor positions you actually need should be created.

```
Symbol botLeft = new Symbol(_surface);
botLeft.UseHomogeneousPosition = true;
botLeft.HomogeneousPosition = new Vector2(0f, 0f);

Symbol centreLeft = new Symbol(_surface);
centreLeft.UseHomogeneousPosition = true;
centreLeft.HomogeneousPosition = new Vector2(0f, 0.5f);

Symbol topLeft = new Symbol(_surface);
topLeft.UseHomogeneousPosition = true;
topLeft.HomogeneousPosition = new Vector2(0f, 1f);
```

```
Symbol topCentre = new Symbol(_surface);
topCentre.UseHomogeneousPosition = true;
topCentre.HomogeneousPosition = new Vector2(0.5f, 1f);

Symbol topRight = new Symbol(_surface);
topRight.UseHomogeneousPosition = true;
topRight.HomogeneousPosition = new Vector2(1f, 1f);

Symbol centreRight = new Symbol(_surface);
centreRight.UseHomogeneousPosition = true;
centreRight.HomogeneousPosition = new Vector2(1f, 0.5f);

Symbol botRight = new Symbol(_surface);
botRight.UseHomogeneousPosition = true;
botRight.HomogeneousPosition = new Vector2(1f, 0f);

Symbol botCentre = new Symbol(_surface);
botCentre.UseHomogeneousPosition = true;
botCentre.HomogeneousPosition = new Vector2(0.5f, 0f);

Symbol centre = new Symbol(_surface);
centre.UseHomogeneousPosition = true;
centre.HomogeneousPosition = new Vector2(0.5f, 0.5f);
```

Symbols can now be created as children of these anchors, and positioned relatively to the anchors. Here is an example crosshair:

```
Circle crosshairCentre = new Circle(centre);
crosshairCentre.Position = new Vector2(0, 0);
crosshairCentre.Radius = 4;
crosshairCentre.Filled = false;
crosshairCentre.Color = _defaultColor;
crosshairCentre.LineWidth = 2;

Polygon crosshair = new Polygon(centre);
crosshair.PrimitiveType = PolygonPrimitiveType.Line;
crosshair.Position = new Vector2(0, 0);
crosshair.Points.Add(new Vector2(0, 50));
crosshair.Points.Add(new Vector2(0, 150));
crosshair.Points.Add(new Vector2(0, -50));
crosshair.Points.Add(new Vector2(0, -150));
crosshair.Points.Add(new Vector2(50, 0));
crosshair.Points.Add(new Vector2(150, 0));
crosshair.Points.Add(new Vector2(-50, 0));
crosshair.Points.Add(new Vector2(-150, 0));
crosshair.Color = _defaultColor;
crosshair.LineWidth = 2;
```

It can sometimes be helpful to make utility methods when lots of symbols have similar properties that need to be set. Here is an example of a method to create a text label symbol, using a default color field and default parameters for font size and font style so that we don't have to specify these every time we want to display some text:

```
private readonly Color _defaultColor = new Color(0, 255, 0, 255);

private Label CreateLabelSymbol(
    HierarchyItem parent,
    string text,
    Vector2 relativePosition,
    TextAlignment textAlignment,
    float fontSize = 28,
    FontStyle fontStyle = FontStyle.MonospacedSansSerifBold)
{
    Label label = new Label(parent);
    label.Text = text;
    label.Position = relativePosition;
    label.Color = _defaultColor;
    label.FontSize = fontSize;
    label.FontStyle = fontStyle;
    label.Alignment = textAlignment;
    return label;
}
```

A mission time label can be created using this utility method. Note that we are storing this symbol as a field. This is because we will need to update the text of this field later (as the mission time changes), whereas a crosshair does not require any runtime updates.

```
_missionTimeLabel = CreateLabelSymbol(topRight, "00:00:00",
    new Vector2(-100, -50), TextAlignment.MiddleCenter, 24);
```

Update Method

We need an update method to be called regularly. This is recommended to be called from a handler for the MACE API `AllMotionComplete` event.

This method should first update any symbols that need updating (mission time label, in our example), and then call Update on the Symbols Manager. The Update method on the Symbols Manager should always be called regularly.

```
public void Update()
{
    UpdateData();
    _manager.Update();
}

private void UpdateData()
{
    if (_missionTimeLabel != null)
```

```
        {
            _missionTimeLabel.Text =
    _mission.MissionTime.ToString("HH:mm:ss");
        }
        // TODO update other symbols here...
    }
```

When updating `Symbols` that draw on screen, such a `Rectangle`, `Circle`, `Triangle`, and `Polygon`, it is preferable to favor changing `Position` and `Rotation` over modifying the points in the shape directly; changing points requires the entire symbol to be retransmitted via CIGI, and the IG-side representation to re-mesh the symbol. This can be very expensive if done every frame and is frequently not necessary.

Additionally, it's efficient to draw multiple lines as a single symbol, but only if the points do not change. If splitting a symbol would help allow the above rule about managing the shape with position and rotation vice recalculated points to be maintained, this will be the most performant solution.

Cleaning Up

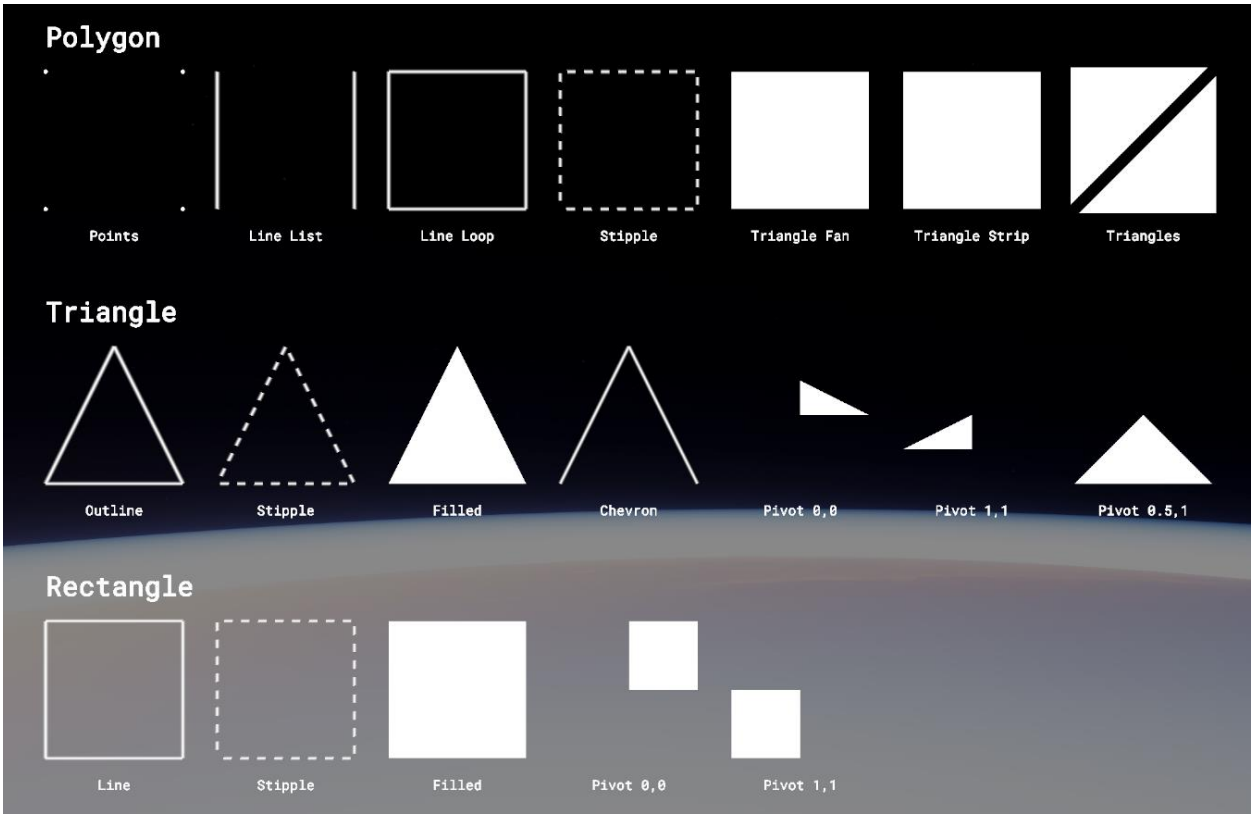
On shut down, or when we want to deactivate our symbols, we can simply destroy our surface. However, we need to make sure to call Update on the manager afterwards.

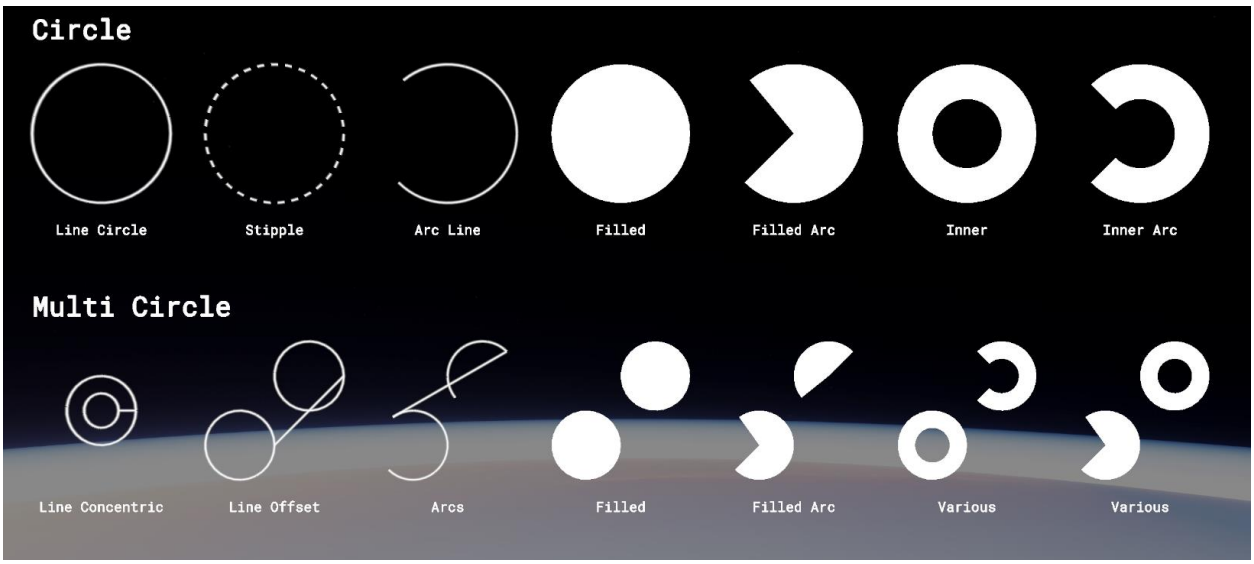
```
public void CleanupSymbols()
{
    _surface?.Destroy();
    _manager?.Update();
}
```

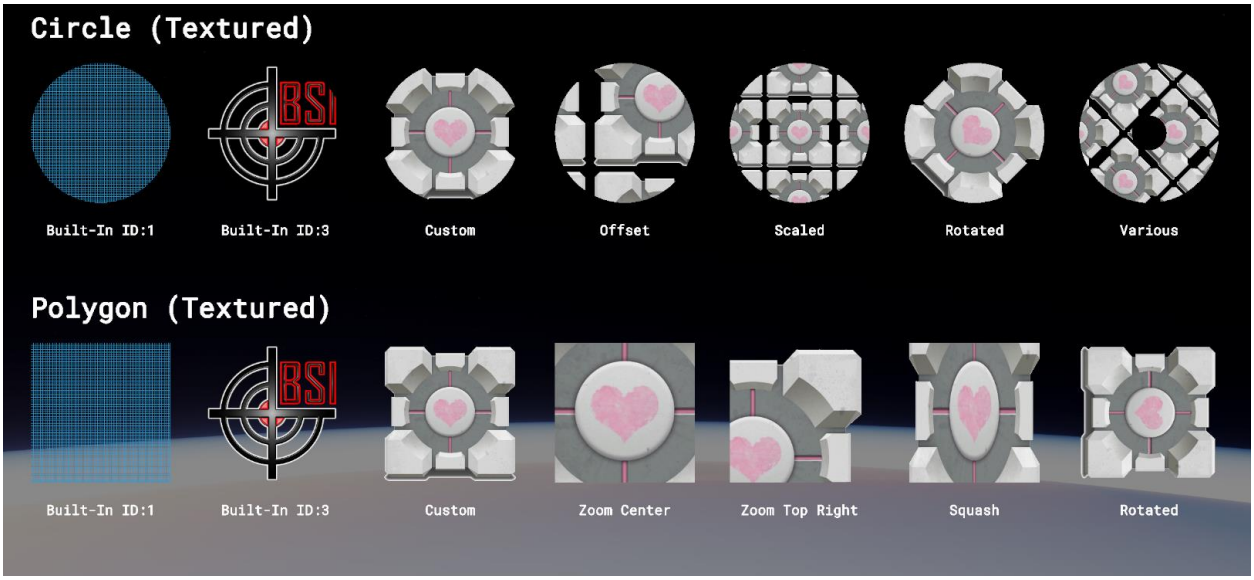
Symbols

Symbol Name	Description	Some Useful Properties
Symbol	A base class for all symbols. If used directly, will behave as an anchor that is never rendered but obeys all hierarchy and control behaviors.Useful to use as a parent to other symbols	PositionRotationScaleUseHomogeneousPositionHomogeneousPosition
Circle	A circle.Can be filled or unfilled, and can also contain an inner circle to make a "donut" shape.Can also be an arc by specifying a start and end angle (counter-clockwise)	RadiusInnerRadiusStartAngleEndAngleFilledLineWidthStipplePatternStipplePatternLength
MultiCircle	A symbol that contains multiple circles.	

Symbol Name	Description	Some Useful Properties
Polygon	A general polygon symbol, defined by points.This symbol is best used for drawing points, lines or unusual shapes.PrimitiveType of Line List or Line Loop can be used to draw multiple lines using one symbol, which is more efficient than using multiple symbols.	PrimitiveTypePointsLineWidthStipplePatternStipplePatternLength
Triangle	A variant of Polygon.Can be filled or unfilled.TriangleStyle can be used to specify fill type, and Chevron can also be selected.PrimitiveType can be set to TriangleStrip or TriangleFan to draw multiple connected triangles. TriangleStrip forms triangles by reusing the last two vertices, and TriangleFan forms triangles by reusing the first vertex.	PrimitiveTypeTriangleStylePivotSize
Rectangle	A variant of Polygon.Can be filled or unfilled.	PivotSizeUseHomogeneousSizeHomogeneousSize
Label	A symbol to draw text.	TextTextAlignmentTextOrientationFontIDFontStyleFontSize


Polygon
Points
Line List
Line Loop
Stipple
Triangle Fan
Triangle Strip
Triangles
Triangle
Outline
Stipple
Filled
Chevron
Pivot 0,0
Pivot 1,1
Pivot 0.5,1
Rectangle
Line
Stipple
Filled
Pivot 0,0
Pivot 1,1


Circle
Line Circle
Stipple
Arc Line
Filled
Filled Arc
Inner
Inner Arc
Multi Circle
Line Concentric
Line Offset
Arcs
Filled
Filled Arc
Various
Various



Plugin Template

A MACE Plugin template utilizing the symbol library is available for download from downloads.bssim.com.

Helpful Tips

- It is more efficient to draw multiple lines as a single symbol, rather than a symbol for each line. Each line is defined by a pair of points. For an example of this, see the crosshair example above.
- For multi-line text, it is advised to have each line as a separate Label symbol. However, the multiple Label symbols can be parented to a common anchor.