
NOTE: This article and all content are provided on an "as is" basis without any warranties of any kind, whether express or implied, including, but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. In no event shall Wonderware North be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortuous action, arising out of or in connection with the use or performance of information contained in this article.

Introduction

Wonderware ArcestrA graphics allow developers to create vivid, powerful visualization tools that can be embedded into InTouch applications. However, the new vector based graphic tools can be more taxing on runtime memory than standard InTouch graphics, and in some cases can take longer to bind to the tag/object attributes set to be displayed. The following guidelines, when followed, optimize ArcestrA graphics in runtime, allowing them to render fast, bind quickly, and optimize memory usage.

Gradient Usage

Gradients and transparencies will add a great deal of visual flair to an object, but at a cost. The graphics engine calculates the colors in the gradient at runtime, first when the graphic is initially called, and then anytime an animation changes the graphic. Due to the number of intermediary colors that may be involved in a gradient, this information consumes more memory than a solid fill, and also takes more time to render than a solid fill.

Additionally, taking a graphic with a large amount of gradients and embedding it multiple times in another graphic can amplify the performance hit.



Figure 1: Gradient Usage Example

Example: A developer creates a graphic using a large number of warning symbols. This symbol is developed full size (Figure 1, left), then embedded multiple times in another graphic and scaled to a much smaller size (Figure 1, right). Even though the graphic has been shrunk, the symbol is rendered exactly the same as if it were full

425 Caredean Drive, Horsham, PA 19044

Tel: 877.900.4996

www.wonderwarenorth.com

TechTip: Optimizing ArcestrA Graphics

size; the same detail, calculations, and memory/processor cost are identical. However, the graphic itself has a different visual value; the gradients are more muted, and some of the border gradients cannot even be seen.

Thus, limiting the use of gradients and sizing graphics appropriately is recommended when embedding a graphic multiple times on the same screen; either on an InTouch window or in another embedded symbol.

Bitmaps

ArcestrA graphics support multiple rasterized graphic formats (BMP, GIF, JPG, TIF, PNG, ICO, EMF). Many customers import bitmaps to represent certain, more complex, pieces of equipment, or to present a more photo-realistic representation of the system.

Of these graphic types, the bitmap (BMP) is the most common graphic type. However, a bitmap is also an uncompressed graphic format; therefore it is also the most expensive at runtime from a memory usage and initial retrieval aspect. Thus, using a compressed file format, such as a JPEG, GIF, or PNG, would greatly reduce the memory usage and retrieval time for the ArcestrA graphic where the file is embedded.

Additionally, graphic size also plays a part in the memory usage on a graphic; if a developer imports a graphic that is 1024x768 and shrinks it to 320x240, the memory footprint is still the same as if it were displayed full size. Therefore, using a simple photo editor or drawing program to shrink the graphic to its desired display size, then embedding the small version, will greatly reduce memory usage.



Figure 2: Shrinking bitmaps in InTouch does not decrease the size on disk, and rendering takes the same amount of time as the bitmap at full size.

Custom Property Density

Custom properties allow the developer to create modular graphics that can be used for a variety of different applications. For example, a valve symbol created with custom properties could be used for several different

425 Caredean Drive, Horsham, PA 19044

Tel: 877.900.4996

www.wonderwarenorth.com

Application Server templates as well as traditional InTouch visualization. However, each custom property needs to bind to its reference. Because of this, graphics that are rich in custom properties can take a longer time to call up or close. Oftentimes, graphics dense in custom properties are dense due to several layers of embedded symbols; each time a symbol is embedded, it carries its custom properties with it.

To minimize the amount of call up time, one should consider minimizing the amount of custom properties in a graphic, especially if those custom properties are not going to be used. As an example the PipePipe graphic has custom properties on it in order to change its color on a true value. If the developer is not planning on using the PipePipe in an animated fashion, it would be better to make a duplicate graphic of PipePipe, strip out the animation and custom properties, and use the new graphic to create piping. Another, more efficient option would be to change the embedded PipePipe symbol to a group in the main graphic, then not import the custom property - this will be discussed later in the document.

Multi-Variable Expressions

One of the common pitfalls that traditional InTouch developers fall into when creating ArcestrA graphics for System Platform is to use long, complicated expressions for animation.

Example: consider the following Visibility expression:

valve1.A > valve1.B AND Valve1.C > Valve1.D OR Valve1.E

This expression has 5 references in it; all five must be subscribed to, evaluated, and published to the graphic engine individually. Furthermore; an expression acts like an ad-hoc script, it must be evaluated continually while the graphic is up. Thus, the more multi-variable expressions in an object, the more CPU time it takes during execution.

Expressions like the one above are much better suited to execution in an Application Object; the Application Server system is built to execute volumes of scripts; and thus should be used as often as possible to handle scripts. Thus, when considering using multi-variable Expressions in ArcestrA graphics, it is best to ask if this expression would be better suited to run server side in an associated Application Object.

To conclude the example, the expression above can be set in a periodic script on the Valve object, and tied to another UDA that provides the result;

me.Result = me.A > me.B AND me.C > me.D OR me.E

Then the Visibility expression can simply be "Valve1.Result".

Script Utilization

The discussion on multi-variable expressions brings up a broader topic of symbol scripts as a whole; the symbol scripting is very dynamic and flexible and allows the developer to implement a vast amount of complex scripts.

425 Caredean Drive, Horsham, PA 19044

Tel: 877.900.4996

www.wonderwarenorth.com

TechTip: Optimizing ArcestrA Graphics

However, these scripts can also be executed cyclically (i.e. while true ever 50 ms, while showing every 250 ms) and care should be taken into how often a script is executed. The larger the script execution count, the more CPU time it takes during execution, and the more RAM the graphic uses in the WindowViewer process. That being said, there are many things that can best be accomplished using symbol scripting. Thus, when creating symbol scripts, it is best to ask the following questions:

- When setting a script to run at a very high time resolution (i.e. 50 ms)- Why? How many references/how long is this script? Is every execution needed? Is the same result be calculated over and over?
- How many scripts are on the symbol? Are they all running at a high resolution?
- How many instances of this symbol will be used on a single screen?
- If the graphic has embedded graphics within it, how many scripts are running on them?
- Can the script be run in an Application Server object, and then the result provided to the graphic?

Again, when dealing with scripting, use good judgment in determining which scripts need to be at the graphic level; Application objects are **logical** code; their primary purpose is to handle execution of I/O and scripting. Graphical objects, on the other hand, are primarily visualization. Utilize each software's strengths in order to optimize your developed project.

Symbol Embedding

Symbol embedding allows larger, more complex graphics to be created by embedding smaller, more modular graphics. However, there are implications associated with embedding symbols that are often not considered. When embedding several layers of symbols, remember that the final symbol will have every custom property, script, and graphic element from all of the individual symbols. Therefore, the more complex the components are, the more costly the overall graphic is going to be to the symbol.

Furthermore, embedding graphics that already have embedded graphics (stacking/layering), then resizing the graphic onto a screen is an easy way to make overview graphics, but comes at cost. The graphics are still rendered at the original size; thus, shrinking them does not change the CPU/RAM cost of the graphic on the screen. When shrinking a graphic in such a manner, there may be details on the symbols that are not visible at the size it is embedded. These details do not make any visual impact at that size, yet still contribute to the memory usage as if they were original size.

Example: The figure below depicts several graphics that may be used to show statuses. The symbols are made up of several complex gradients and status elements, but will be shrunk down to the size in the center. At this point, the visual effect built in the overall graphic is lost; most of the gradients cannot be seen, the status element is not even visible. Yet, all of these gradients will be rendered as if they were full size. A more efficient

TechTip: Optimizing ArcestrA Graphics

solution would be to make a small version of the graphic, not containing the complex gradients and status element, and use that in the destination symbol.

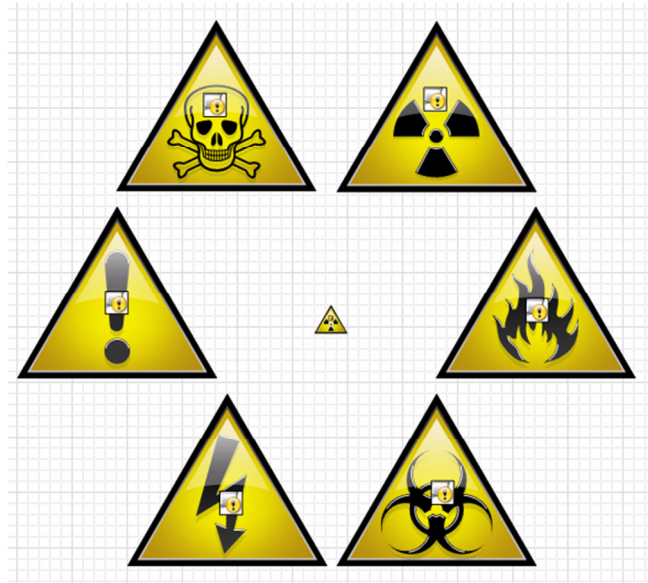


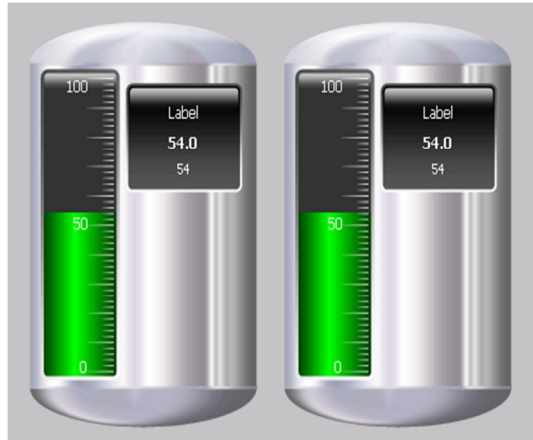
Figure 3: Prolific embedded symbol usage comes at a cost; minimize this cost by making smaller, simpler versions of the symbol.

Element Grouping

One of the lesser known, but key, rendering features of ArcestrA graphics is how the rendering system deals with groups of elements. If an element group does not have any animation on the elements contained within the group, the group is actually rendered and maintained as a single vector graphic. Therefore, grouping elements that are not animated can greatly reduce rendering time and static CPU load.

Example: Consider the two tanks below (Figure 4). The element browser for these "identical" graphics shows that the Tank on the left has a group containing the majority of the non-animated elements, while the Tank on the right does not have such a group. Even though the two graphics have the same visual effect, the tank on the left opens in half the time, with half the static CPU load as the tank on the right. This was achieved by simply grouping the non-animated elements.

TechTip: Optimizing Arcestra Graphics



Elements	Elements
<ul style="list-style-type: none"> [-] [R] OverlayElements [+] [R] SSTank [-] [R] Path8 [+] [R] Path13 [+] [R] Path6 [+] [R] Path4 [-] [R] Line5 [+] [R] Path12 [-] [R] Line7 [-] [R] Line1 [+] [R] Path14 [-] [R] Line4 [-] [R] Line3 [-] [R] Line2 [-] [R] LineBottom [+] [R] Path10 [+] [R] Path11 [+] [R] Path2 [-] [R] Rectangle19 [-] [R] Rectangle18 [-] [R] Rectangle17 [-] [R] Rectangle16 [-] [R] Rectangle15 [-] [R] Rectangle14 [-] [R] Rectangle13 [-] [R] Rectangle12 [-] [R] Ellipse1 [-] [R] Ellipse2 [-] [R] Bottom [-] [R] Top 	<ul style="list-style-type: none"> [-] [R] OverlayElements [+] [R] Path8 [+] [R] Path13 [+] [R] Path6 [+] [R] Path4 [-] [R] Line5 [+] [R] Path12 [-] [R] Line7 [-] [R] Line1 [+] [R] Path14 [-] [R] Line4 [-] [R] Line3 [-] [R] Line2 [-] [R] LineBottom [+] [R] Path10 [+] [R] Path11 [+] [R] Path2 [-] [R] Rectangle19 [-] [R] Rectangle18 [-] [R] Rectangle17 [-] [R] Rectangle16 [-] [R] Rectangle15 [-] [R] Rectangle14 [-] [R] Rectangle13 [-] [R] Rectangle12 [-] [R] Ellipse1 [-] [R] Ellipse2 [-] [R] Bottom [-] [R] Top

Figure 4: "Identical" graphics have different load times due to grouping.

Conclusion

One of the main misconceptions about Arcestra graphics stems from the belief that they must be plain and simple in order to execute fast. However, by following the tactics in this document, stunning graphics can be created that are significantly less taxing on the running system. Our final example is Figure 5. This "complex" graphic utilizes all of the tactics above. Its call-up time is less than a second, and it barely registers 1% for its static CPU Load.



Figure 5: Call up time: under 1 second; Static CPU load: ~1%