# Candera Graphics Programming - Best Practices for Traveo II


- Graphics driver and Hardware related Best-Practices
- CGI Studio related Best-Practices

# Graphics driver and Hardware related Best-Practices

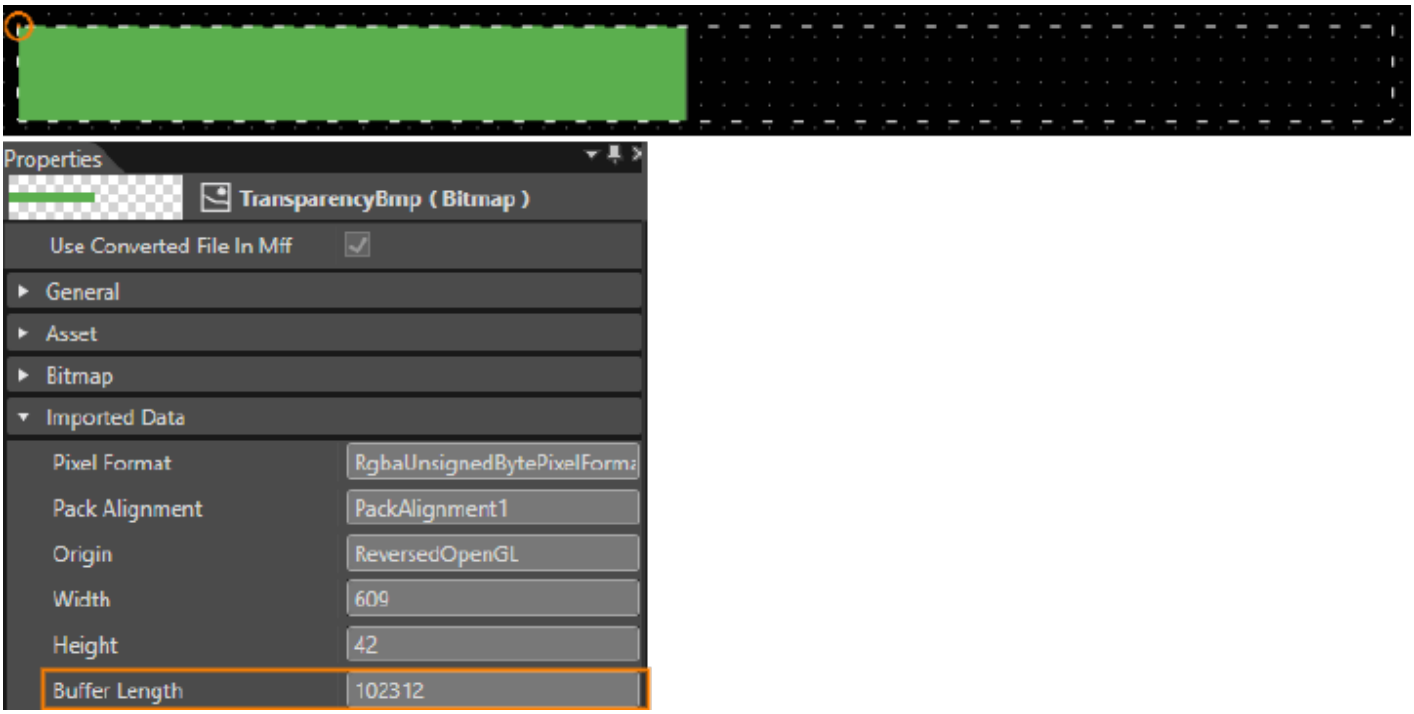
## Images and Color formats

### Best practice for image drawing:

- Always use as small images as possible to save memory consumption and memory bandwidth.
- Cut off transparent area in the bitmap before usage.

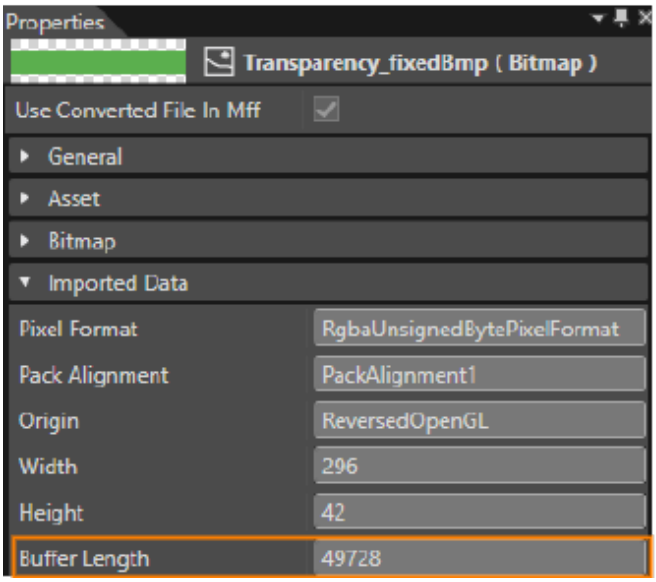
**Bitmaps that contain transparent areas**

**<Origin>**



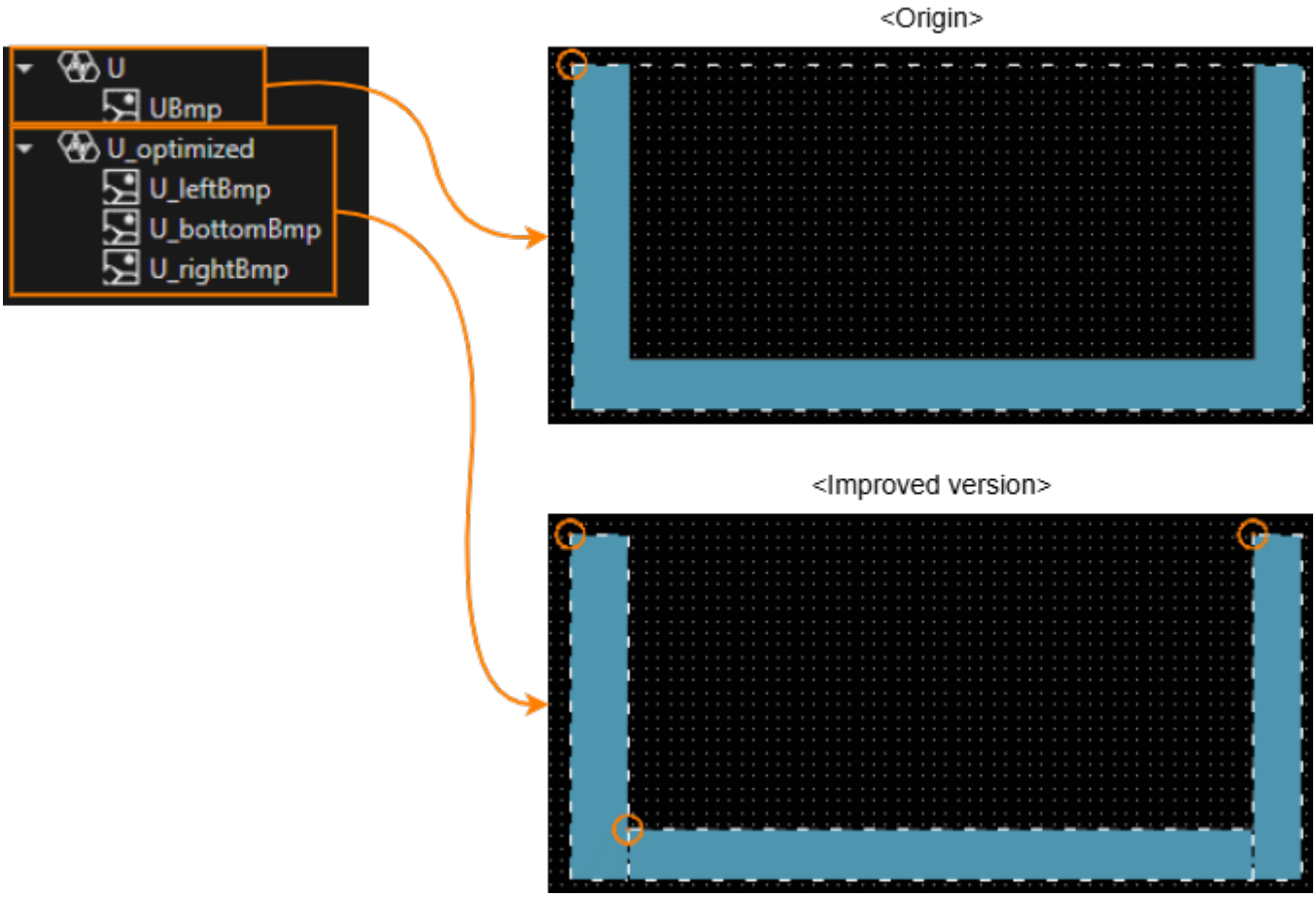

**<Improved version>**



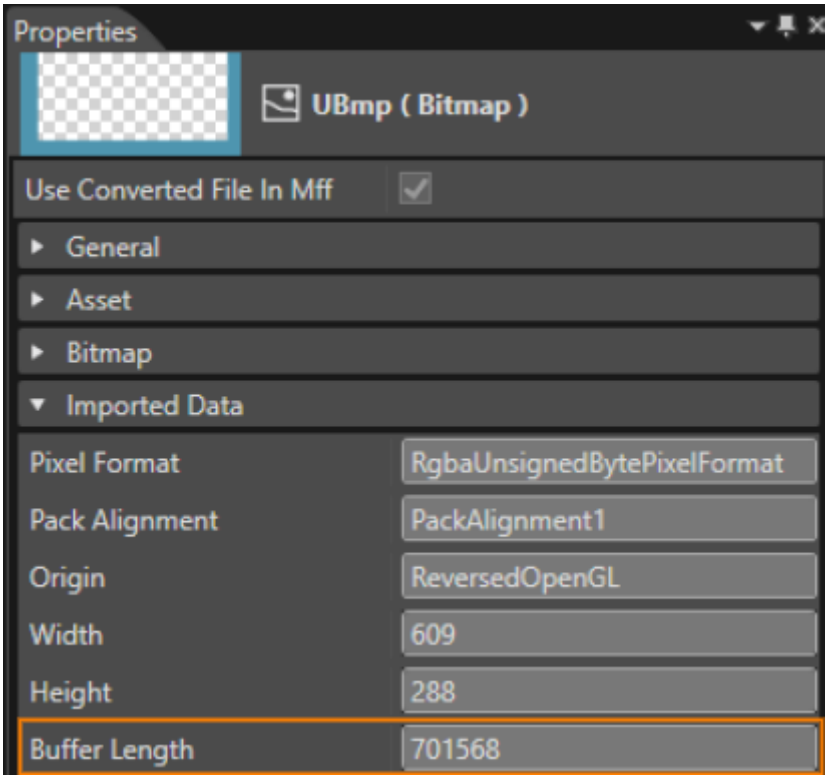



> Bitmap size difference: 102312 ⇔ 49728!

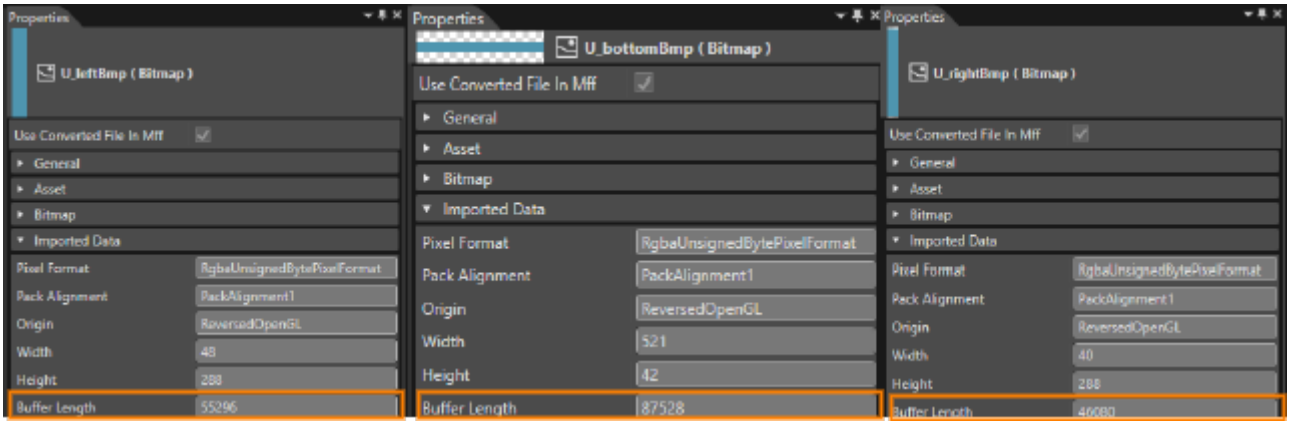
**Modify bitmaps with large transparent areas ("U" shape)**



**<Origin>**



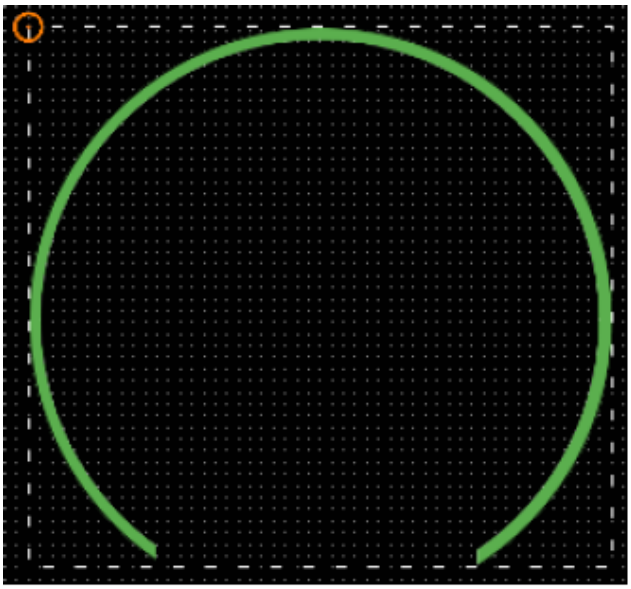

**<Improved version>**

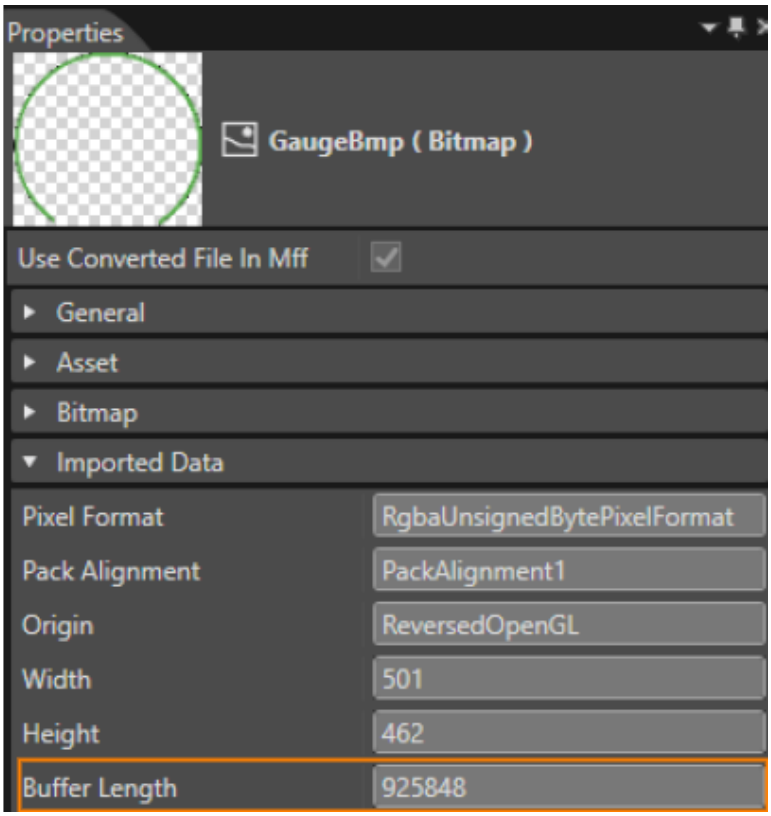



> Bitmap size difference: 701568 ⇔ 188904 (55296 + 87528 + 46080)!

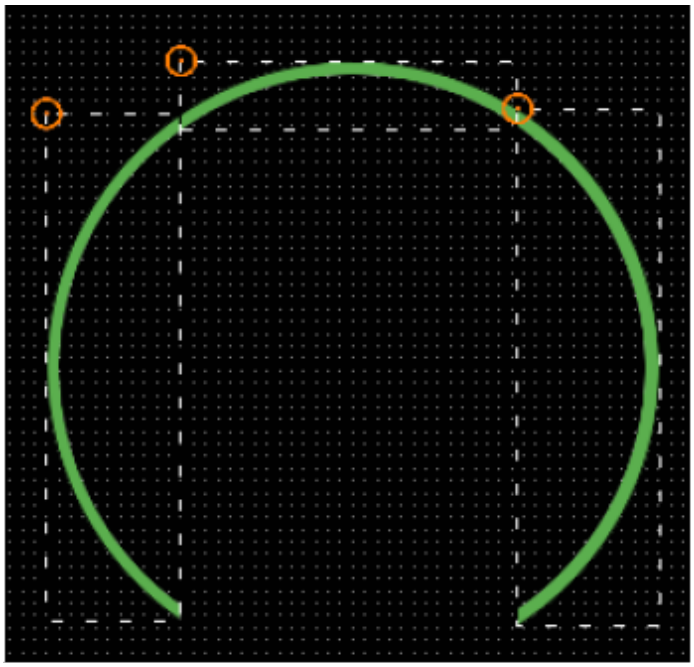
**Modify bitmaps with large transparent areas (Gauge)**

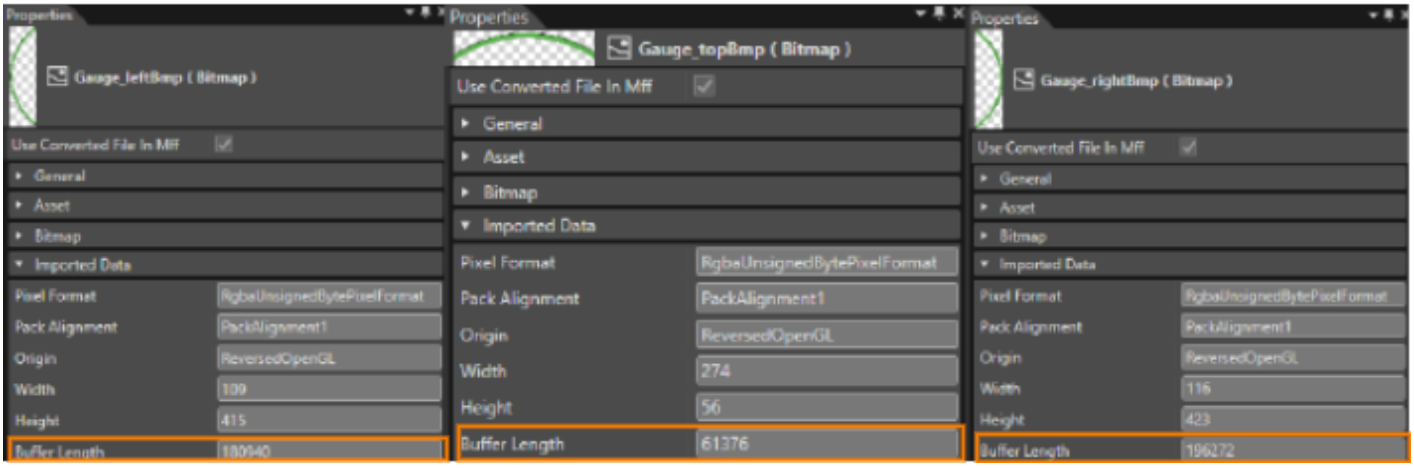
**<Origin>**






**<Improved version>**






> Size difference: 925848 ⇔ 438588 (180940 + 61376 + 196272)!

**Single-colored bitmaps**

For single-colored bitmaps use BitmapBrushColorBlend and use an alpha only color format e.g. A8 or A4. If further size reduction is required there are also compressed formats but those prohibit scaling and rotation on all types of LBO render targets.

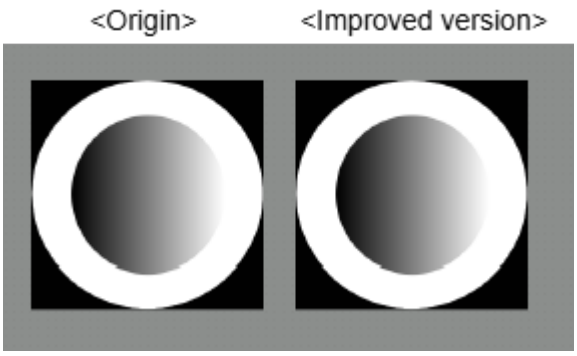
**<Origin>**



**<Improved version>**




**Visual result:**




> size reduction: RGBA: 222028 ⇔ 74092!

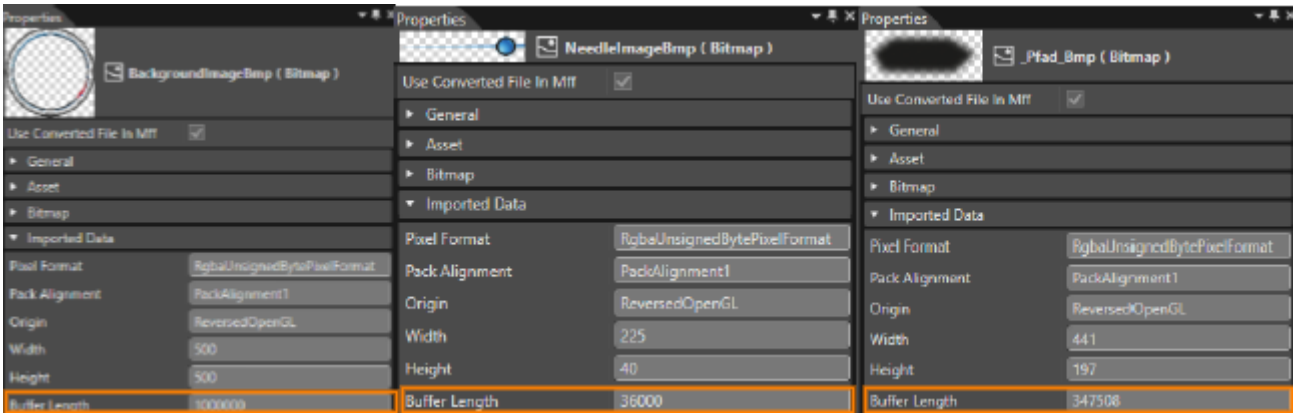
### Combine Bitmaps

Bitmaps can be combined to one bitmap at design time. E.g., tube, scale, and numbers should already be combined if possible.

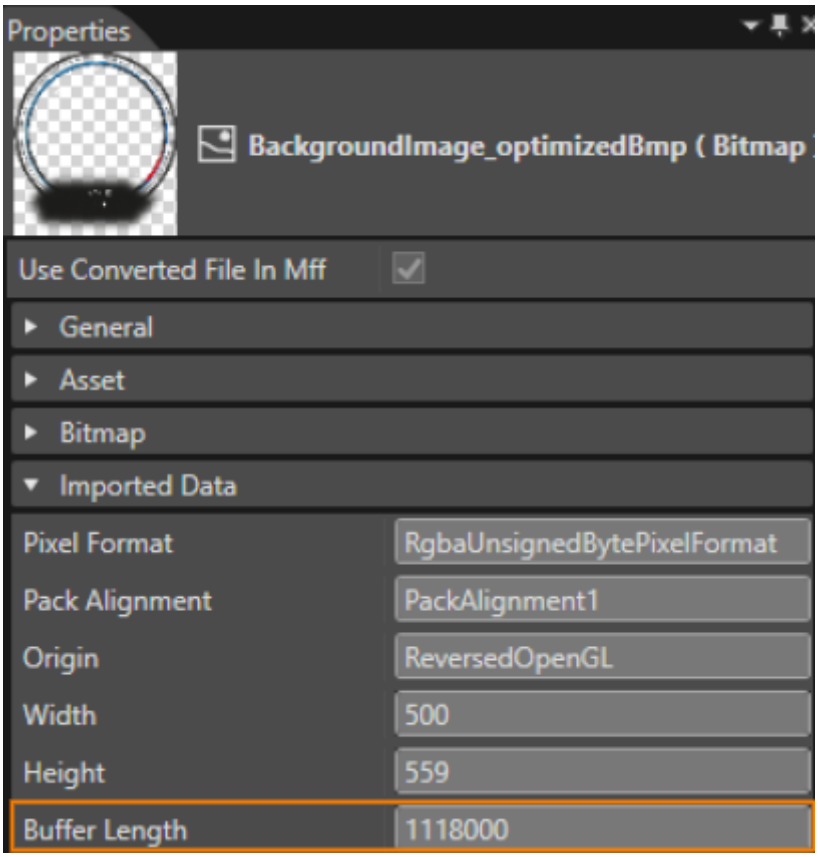



Reduce number of objects from 5 to 2. Avoid rendering of non-translatable static text. Only one moving object left and maybe the gauge itself can be put into a background image.

**<Origin>**




**<Improved version>**




> Bitmap size reduction: 1383508 (1000000 + 347508 + 36000) ⇔ 1154000 (1118000 + 36000)!

---

# Compressed image formats

**Best practice:**

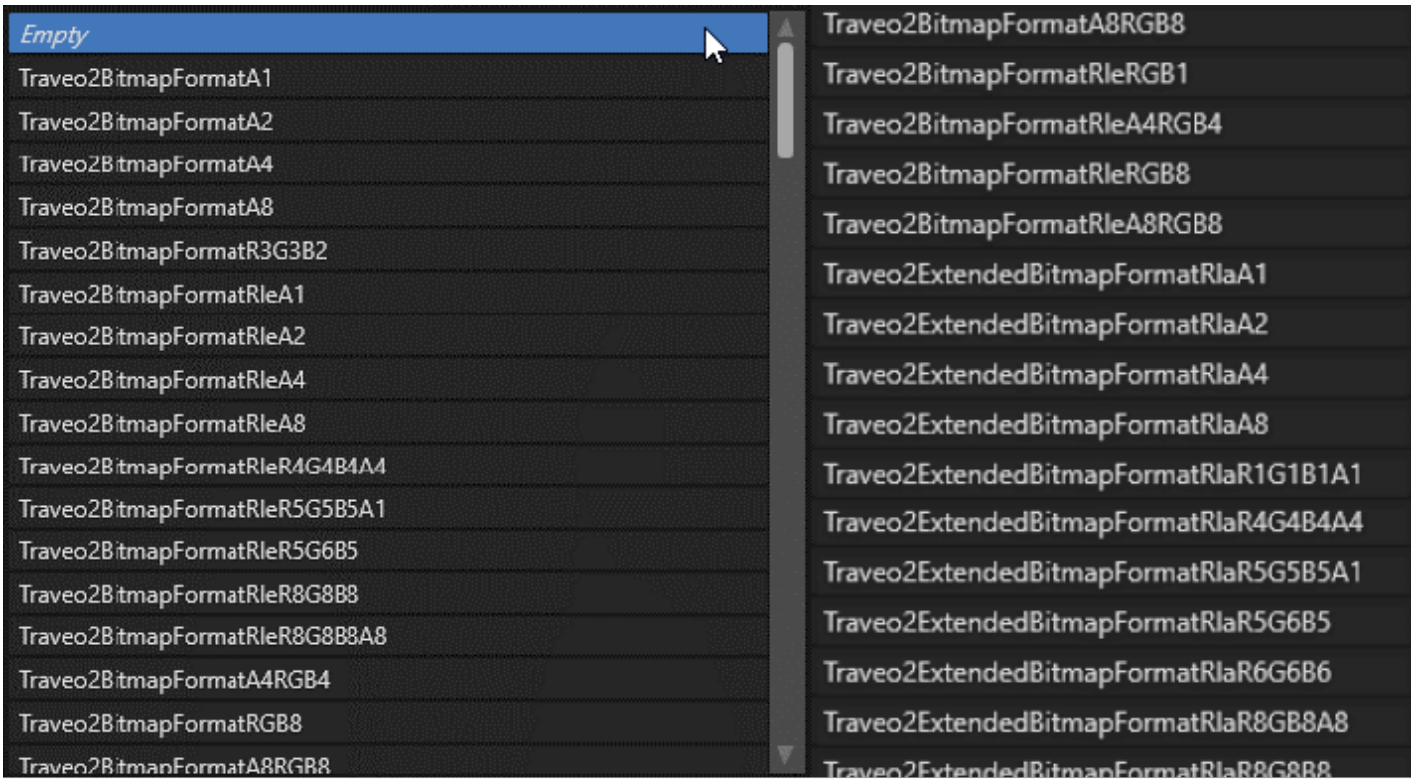
- Use RLE/RLA and CLUT compressions as much as possible wherever possible.
- Prepare the images that they will fit to an image-with alignment multiple of 8.
  If you have a bitmap which has transparencies at the left or right side, you can remove some pixels in x direction to get a multiple of 8. If there are no transparencies, you must add some transparent pixels at the right side of the bitmap to have a multiple of 8.

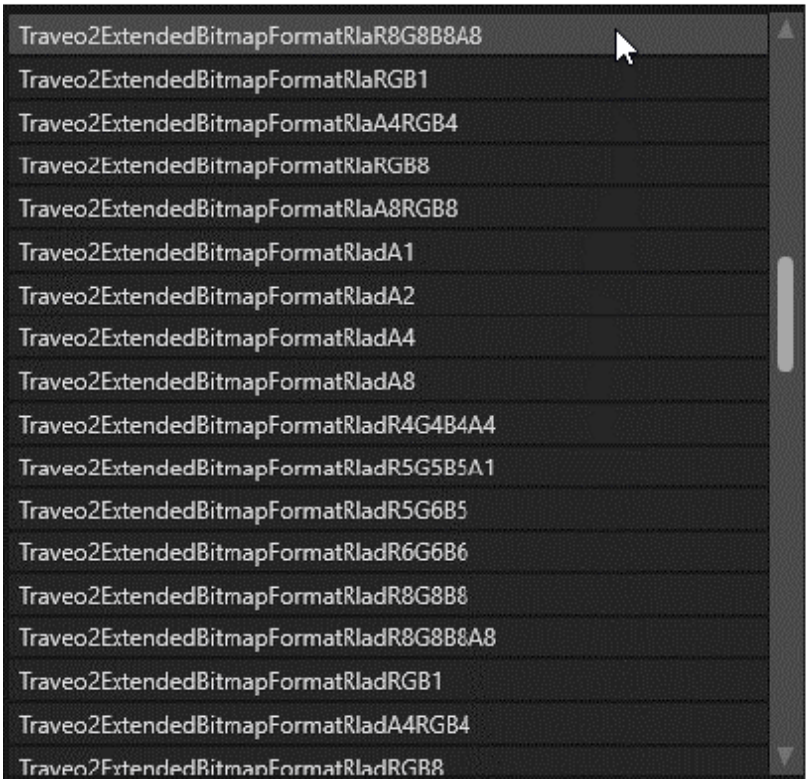
- When using RLE or RLA formats on Traveo2, users must also take into account the hardware-defined 4-byte alignment requirement for compressed image buffers. If this requirement is not met, RLE/RLA compressed images may not be rendered correctly on the target device.

---

## Usage of Bitmap profiles

There are many different bitmap formats available for Traveo2. The goal is always to use a format which uses least memory as possible. The drawback is that some formats have restrictions regarding transformation (rotation, scaling, ...).
Best practice is to use RLE formats to reduce memory size. AX (A1, A2, ...) formats are used to reduce memory further if just recoloring is needed (white or grayish bitmaps can be recolored in CGI Studio).
If a channel (RGBA) has less than 8 bits you will have a lossy compression.
For special use cases there are formats which can dither bitmaps (Nq, Mc, Nq_FsDithered, Mc_FsDithered. Based on the look and feel on the target, usually different formats of such type must be tested to decide what to use.



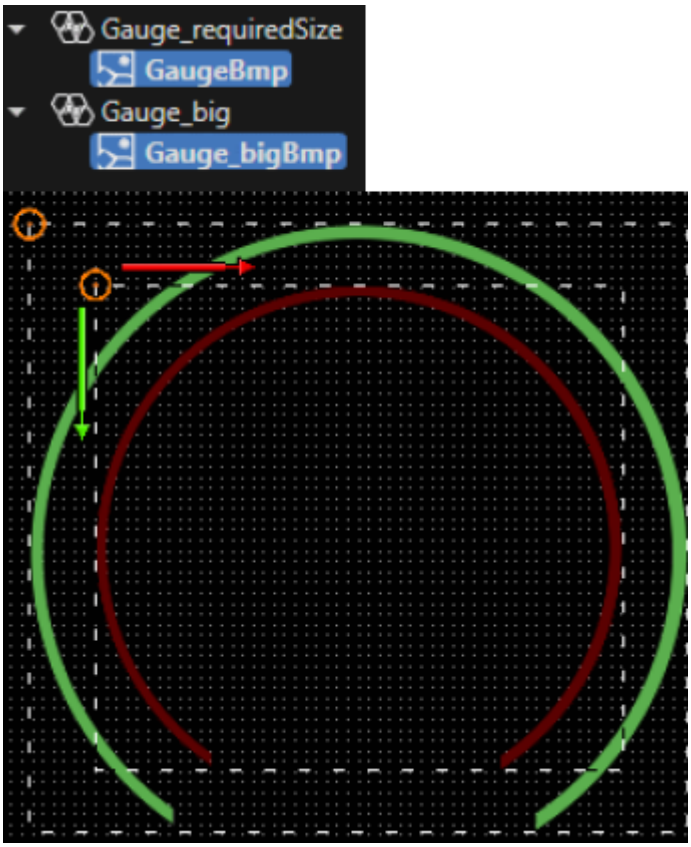




---

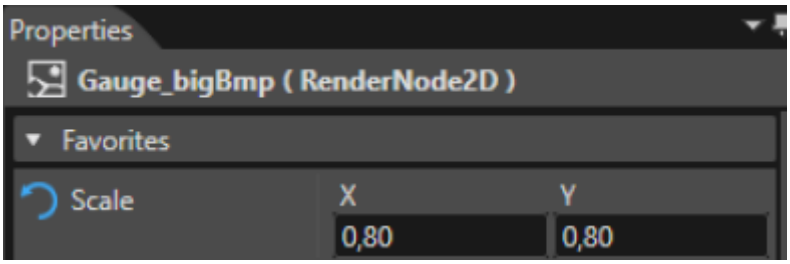
# Transformations

**Best practice Scaling and Rotation:**

- These operations require reverse lookup and therefore need more bandwidth and compute power than a simple BLIT. If not necessary, avoid these.
- It is better to use images with already corrected dimensions instead of scaling down.

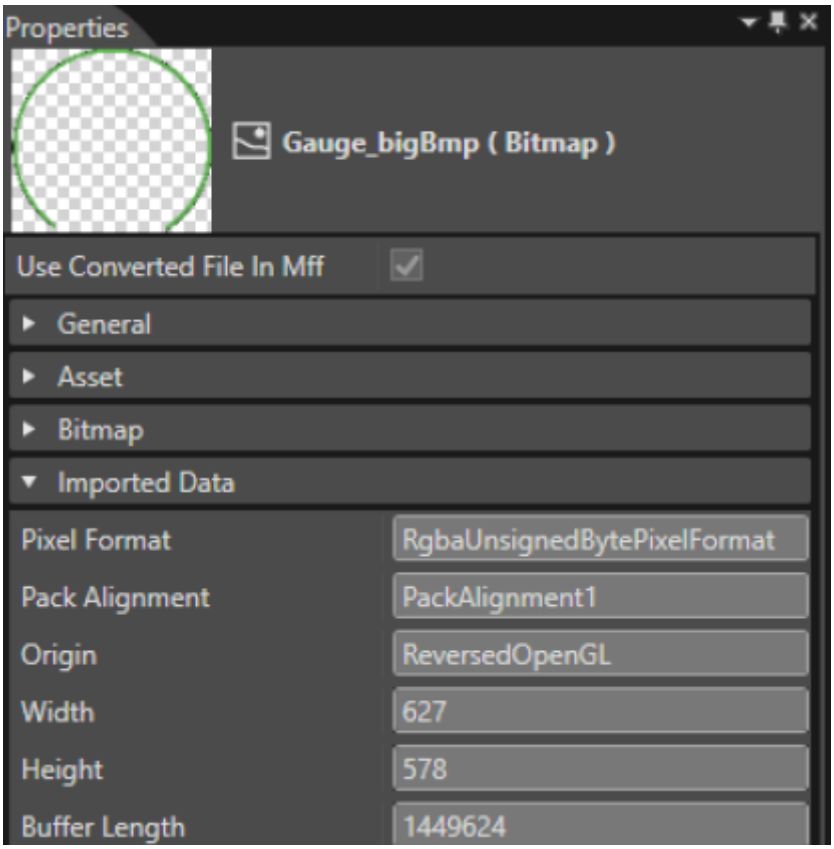



The red gauge should be the required size. The green scale is too big and must be scaled down by to 80%.

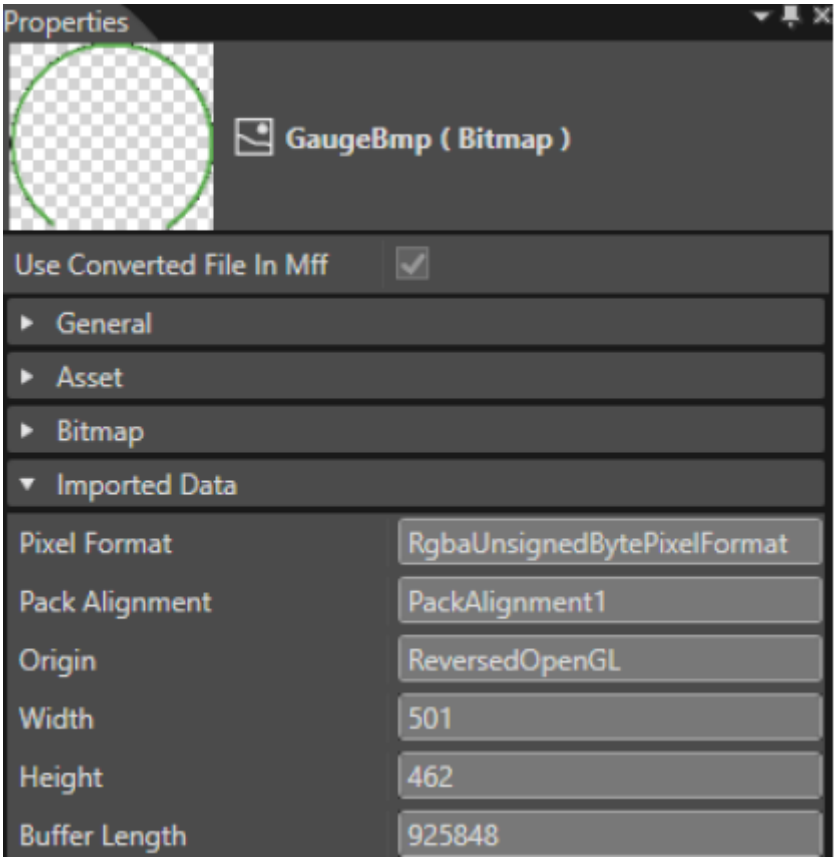



Needed size of gauge will be 925848

**<Origin>**



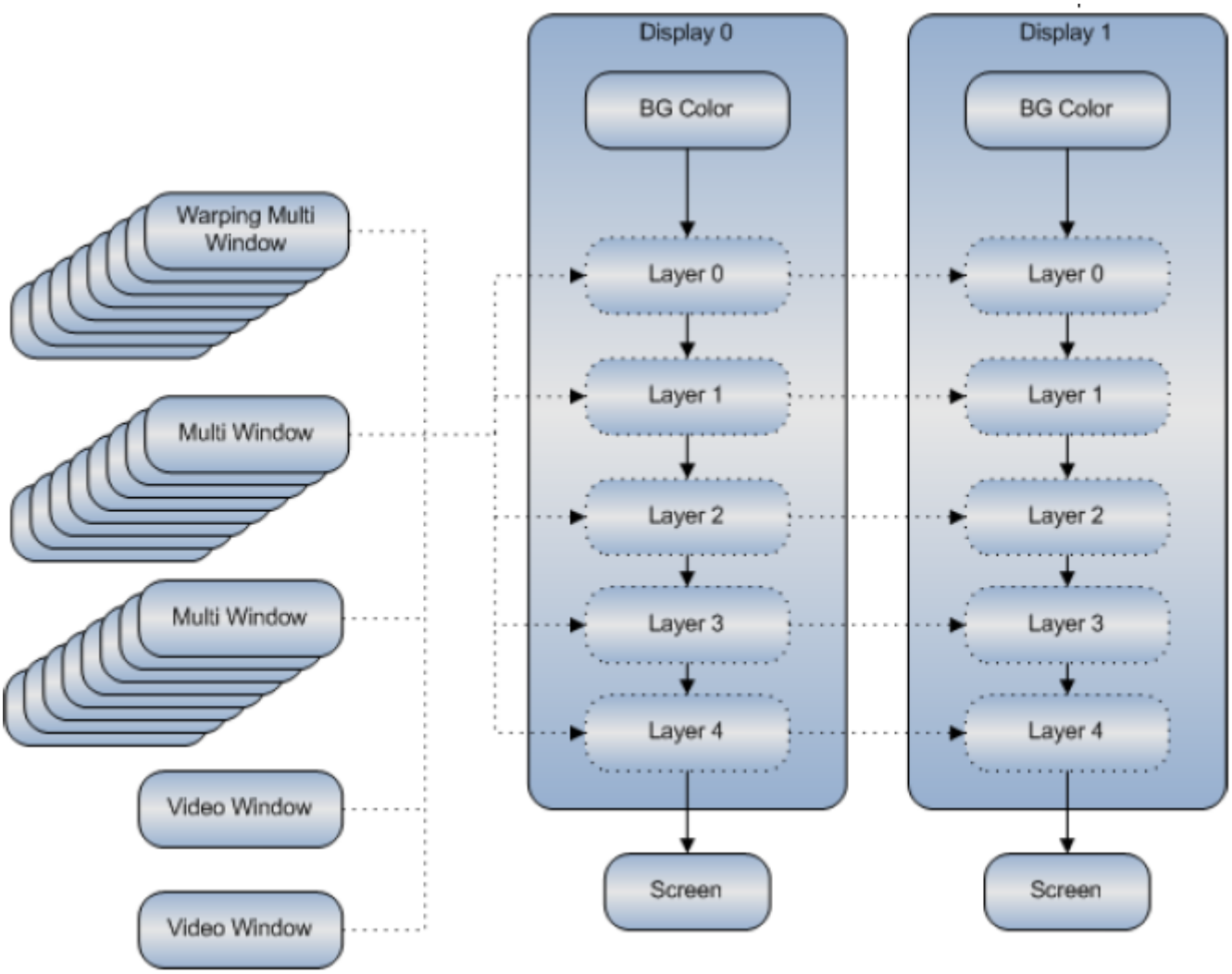

**<Improved version>**




> Bitmap size comparison: 1449642 ⇔ 925848!

## Using Layers

In general, there is always a tradeoff between performance (incl. memory band-with) and memory consumption (e.g., VRAM). Traveo II allows, with the new introduced “Layer-modes” IBO, LBO and OTF, to steer in one of these directions.
Traveo II supports (like Cypress Amber) a huge amount of “separated” Layers, up to 5 “Main-Windows”, 3 of the windows can be split up to 8 sub-windows.
Large area screens shall always be split up to several sub screens to save VRAM consumption.




## Layer types

- **IBO Mode:** regular “dual render target / buffer” approach.
  - **Best practices:**
    - Use IBO layers for dynamic text drawing with Text engine.
    - Use for content which will not change every frame.
    - Use only small areas/content with IBO (For more details, please refer to "Notes for the different modes (IBO/LBO/OTF)" on this page)
- **LBO Mode:** Line based rendering operation
  - Increased BLIT Engine performance
  - **Note the restrictions, following features are NOT supported in LBO Mode:**
    - ROP2 or ROP3 ➔ Mask operations
    - Scaling of compressed images
    - Color Lookup Table (CLUT)

    - **Note the limited functionality:**
        - Rotation: only in steps of 90°
        - Scaling: limited quality for scale factors <0.5
        - Rendering images with compression in LBO mode requires "COPSes" (For more details, please refer to Layer usage)
- **OTF Mode:** On-The-Fly mode
    - Display to a window with a small buffer (a few lines only) in memory
    - All BLIT operations use LBO mode

    - **Note the restrictions/limitation:**
        - All source images shall be copied to VRAM!
        - Maximum OTF width is limited to 1600!

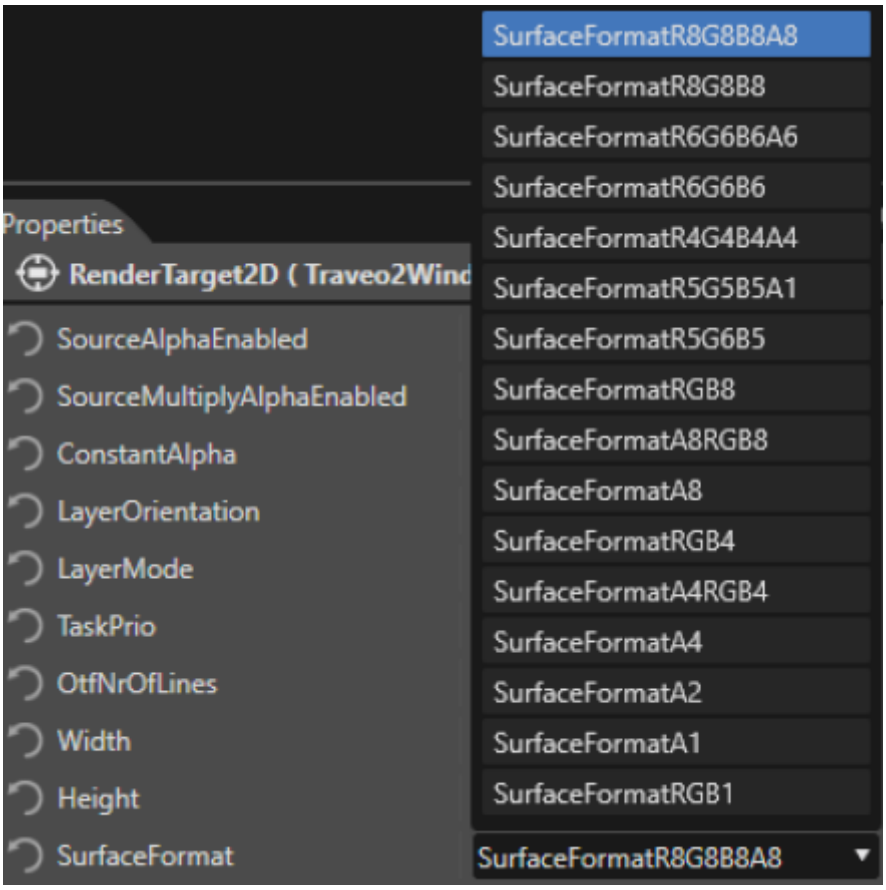
    - **Best practice:**
        - Use as less as possible OTF layers to ensure the targeted performance.
        - Attention to the number of line buffers: There is only a setting of a power of 2 available for the number of line buffers. 16, 32, 64, 128, ... because this will increase the VRAM consumption significantly!
        - Use OTF layers whenever you must draw high frequently changed content (e.g., gauges, pointer, ...)

- **Notes for the different modes (IBO/LBO/OTF)**
    - If necessary, it is possible to mix the usage of different modes (IBO and OTF) by using several tasks in the command sequencer.
    - Keep in mind that an IBO mode task will not yield the command sequencer until it has completed.
    - A long running IBO task can therefore prevent an LBO or OTF task from completing in time which can lead to visible artifacts.
    - The same can happen if the complexity in one slice is too high to be completed in time.

---

# SurfaceFormat optimization

Before starting the project, it must be decided which format could match the expectations of a customer. If less bits per pixel are sufficient for the look and feel it should be chosen to save VRAM. Keep in mind that for overlapping Layers/RenderTargets an Alpha-value is necessary, otherwise no blending of layers is possible.



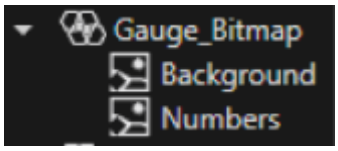

# Text

**Best practice for drawing text related content:**

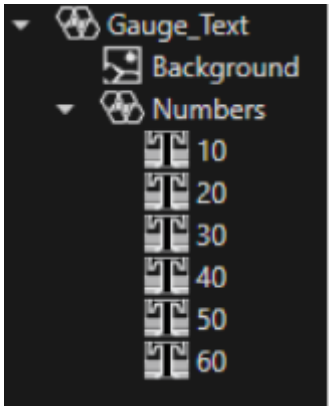
- Always pre-render / rasterize non-translatable text.
- Use static pre-rendered numbers instead of rendering numbers as text (e.g., for Speedometer, RPM, Odometer, ...) Especially for speed use a behavior to switch prerendered digits instead of rendering a text.




*Switch prerendered images instead of text*

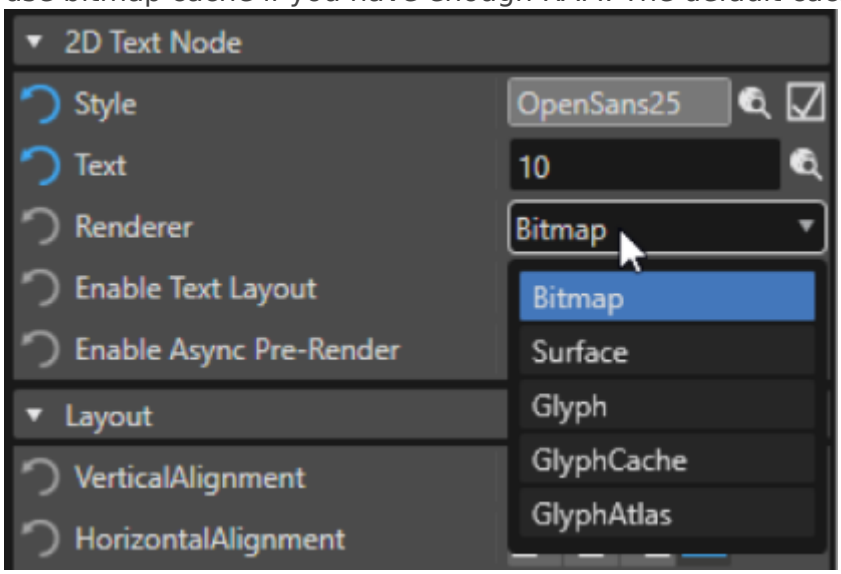



*Numbers are combined in one image*




*Each number as TextNode*

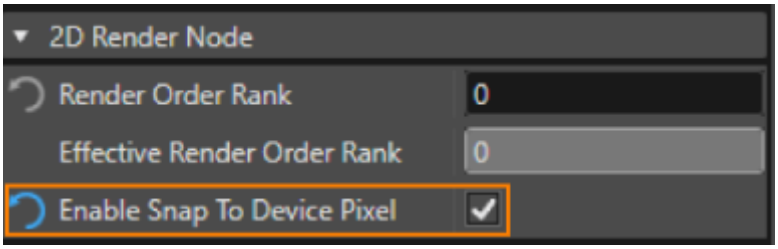
- Text cache type can balance RAM and CPU use. Use no cache if less RAM is available or use bitmap cache if you have enough RAM. The default cache type is set to Bitmap:



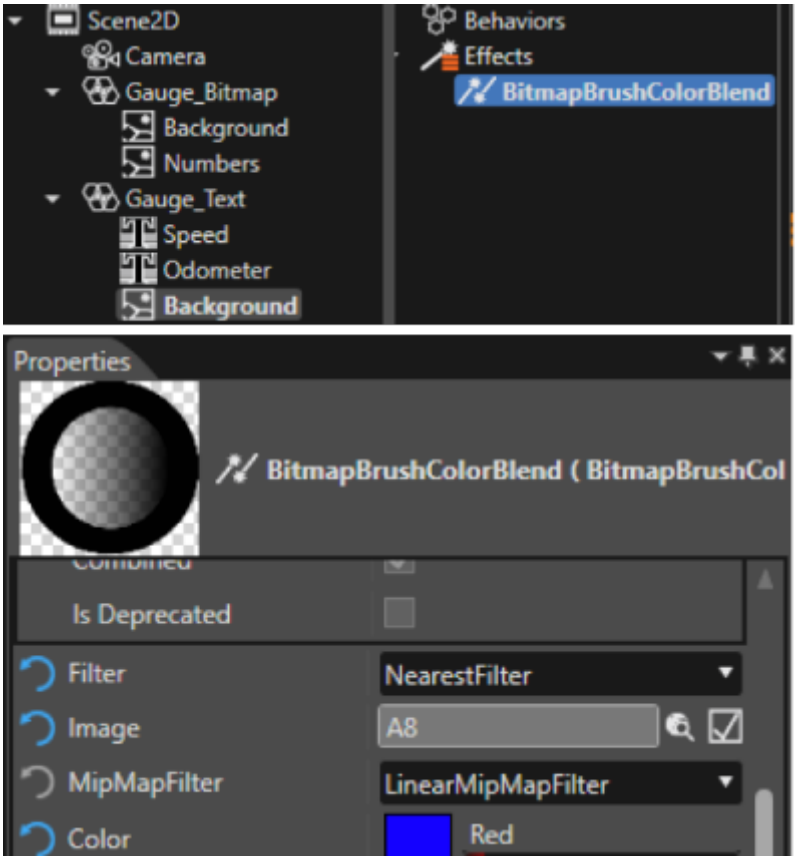
# CGI Studio related Best-Practices

## General recommendation

Enable snap to device pixel. This makes sure no filters must be applied that slow down the BLIT operation.




Only use bilinear filter if an image is rotated, scaled, or positioned between two pixels.
If none of the above happens then stick with the “nearest” filter. Be aware that bilinear is the default filter.

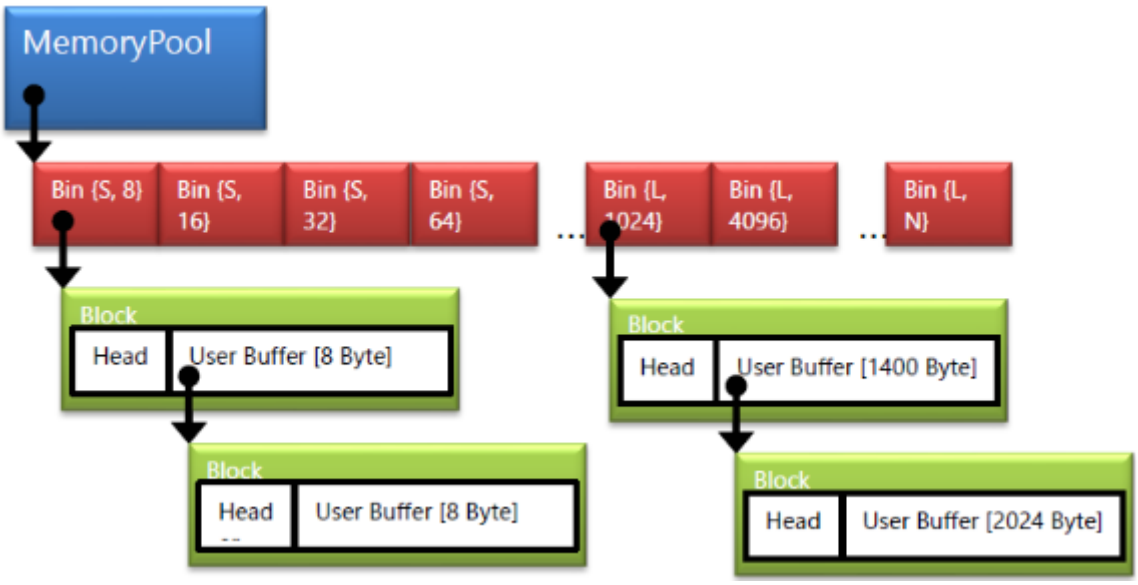



## Memory pools

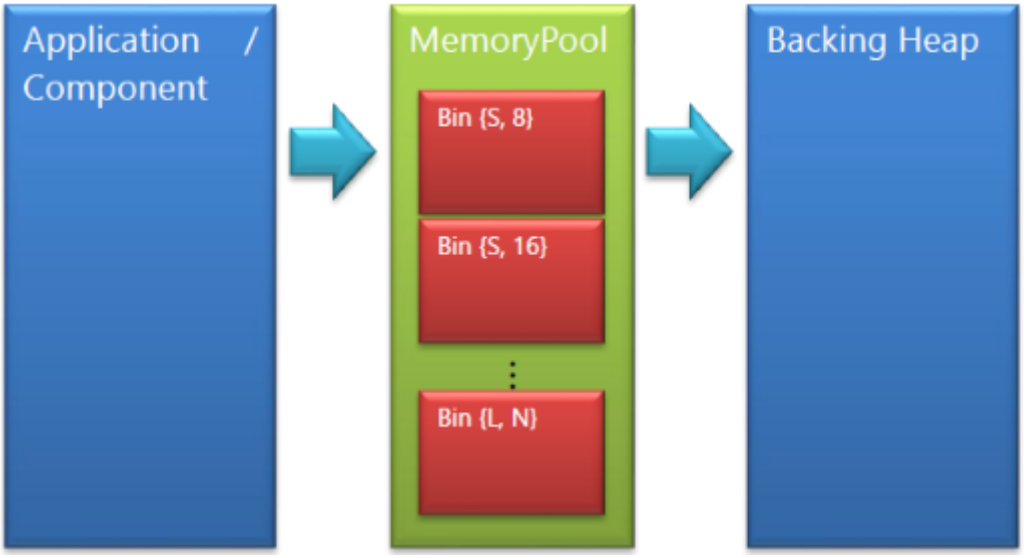
Always use CGI Studio MemoryPool feature to be able to get some freedom to shift the Text-memory area (used for Text rendering with e.g., Freetype) and default memory area (for CGI Studio) to the right memory location (internal RAM, VRAM, HyperRAM)

## Memory pool mechanism:

**Memory Pools, Bins, and Backing Heap:**
A memory pool represented by class MemoryPool consists of a configurable number of Bins (MemoryPoolBin). Each Bin has a defined buffer size, maintains its own free list (in debug mode also a list of allocated blocks), and thread synchronization. A Bin manages so called blocks. Each block consists of a header (BlockHeader) and the user buffer (the actual buffer returned to the application).
There are two different types of Bins – small and large block Bins. A small block Bin maintains blocks which user buffer matches exactly the size of the defined Bin buffer size. A large buffer Bin maintains blocks that have at least a user buffer size of the Bin buffer size. Small block Bins are used – as the name already indicates – for small memory allocations. A small buffer Bin has less runtime and memory overhead per allocation than a large block Bin.
Whenever the application requests an allocation of a certain number of bytes, MemoryPool will look up the Bin which buffer size matches best the requested number of bytes. If the selected Bin has entries in its free list, the Bin will best fit select one of the free list nodes.




If a Bin free list is empty, the Bin will try to allocate a new block from the backing heap.
Thus, Bins maintain lists of blocks allocated from the backing heap. If an allocation request cannot be fulfilled by the already allocated blocks, the Bin tries to allocate the block in the backing heap.



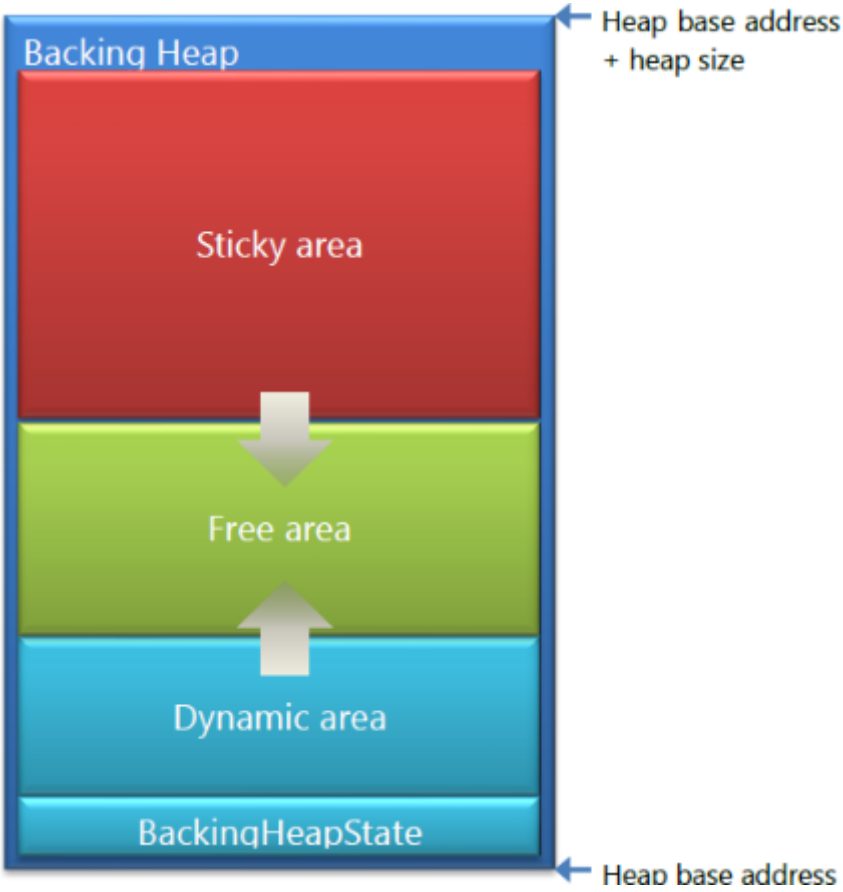

The above figure shows the layered architecture of FeatStd MemoryPool.

- One or more applications may access in parallel a memory pool.
- Multiple memory pools may exist in the system, each configured with its own MemoryPoolConfiguration object.
- Each MemoryPool object is assigned to a backing heap it will allocate memory from.
- Multiple MemoryPool objects may be assigned to a single backing heap.

The whole memory pool configuration is done at compile time. Thus, no runtime overhead is generated.

**Sticky vs Dynamic allocation:**
A sticky allocation is an allocation that will never return the allocated block to the backing heap. The block will be managed by the associated Bin (Bin free list).
A block allocated dynamically (not sticky) will be immediately returned to the backing heap when the application frees the block.
With the knowledge whether an allocation is sticky or not, the backing heap can execute the allocation from two distinct memory areas – the sticky area which will never fragment and the dynamic area which can fragment.

**Backing Heap:**
FeatStd MemoryPool implementation does not depend on a specific backing heap. E.g., standard library malloc and free can be used to implement the backing heap. Such a backing heap is not intended for production environment but might be more convenient than a fixed size heap for development.
FeatStd MemoryPool provides an implementation of a backing heap that can be used out of the box.



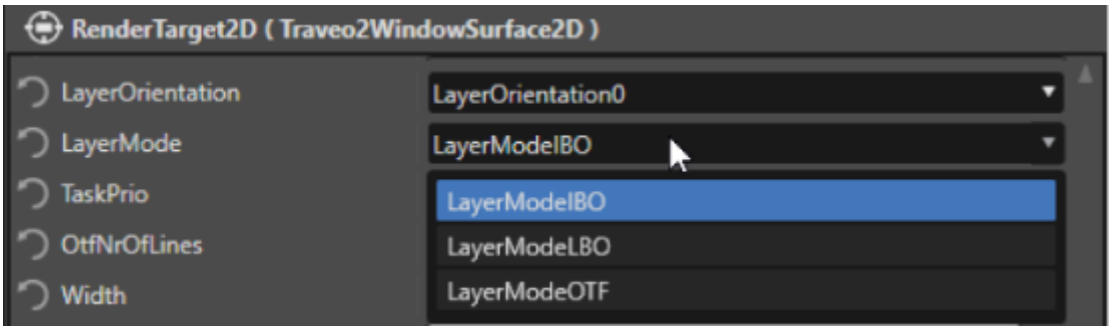

The above figure shows the memory layout of the backing heap. The sticky heap area grows top down, while the dynamic area grows bottom up.
A memory allocation in the sticky area bears no additional overhead. An allocation in the dynamic area has an overhead of 4 bytes required to coalesce consecutive free areas efficiently. Free blocks in the dynamic area are maintained in a free list. On allocation a free list entry will be selected with best fit strategy.

## Memory pool Bin-size optimization

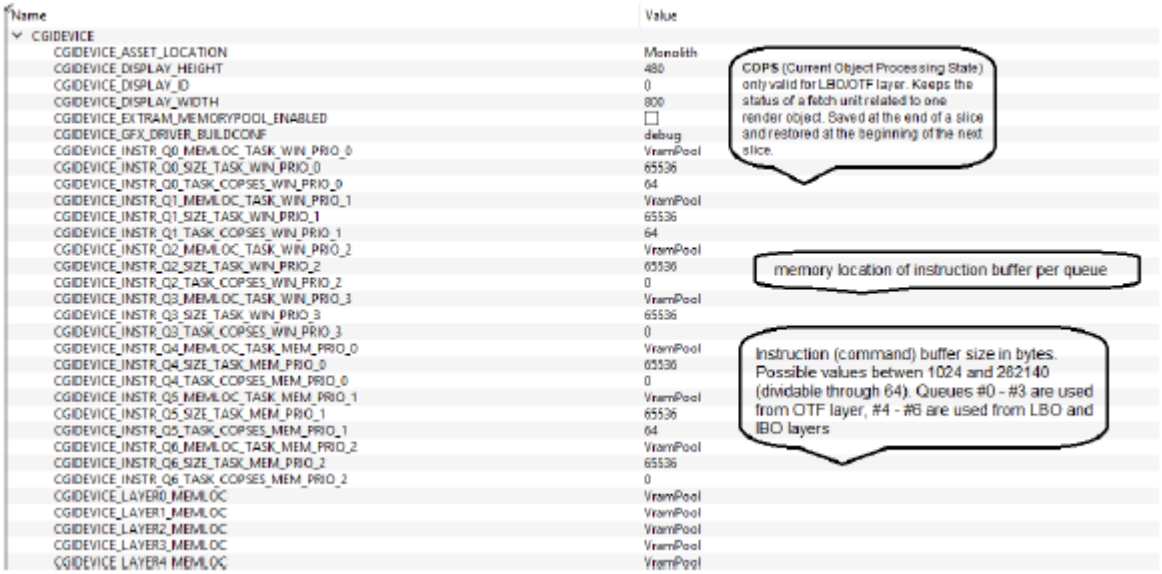
**Best practice:** Do the MemoryPoolBin-optimization to optimize the consumed memory and to achieve a better memory fragmentation while runtime.
To defragment the memory as much as possible and to avoid upcoming “out-of-memory” scenarios you should always do the memory pool optimization on the final application/solution.
Therefor you should perform following steps:

1. Create your default memory pool configuration
2. Let the application run for a while with the config, preferably run through all use cases.
3. Shut down the application and display the statistics on the memory blocks used (e.g., display to console or create a specific text file).
4. Create an optimized memory config based on the output statistics and start from the beginning (-> 2).

# Layer usage

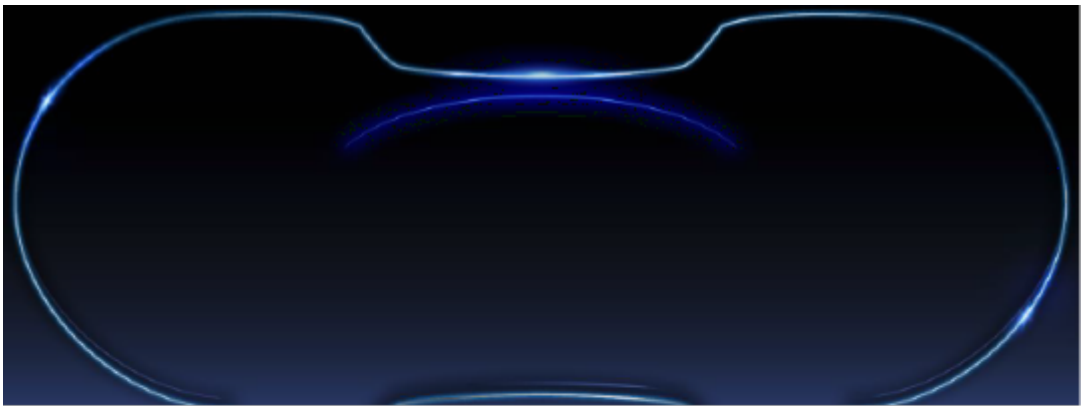


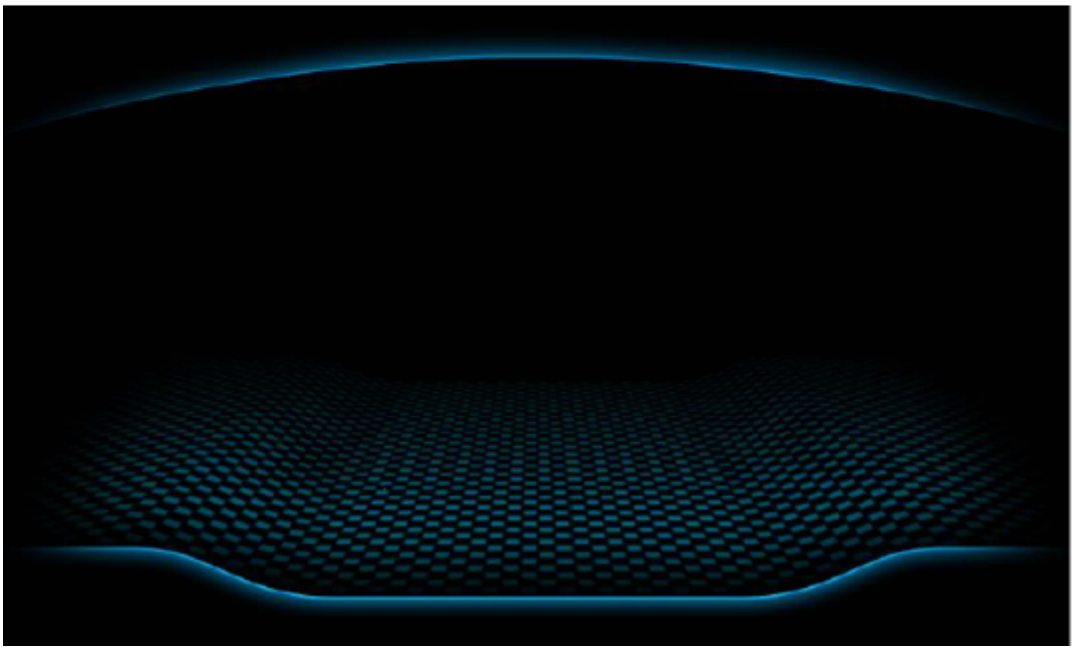

- With CGI Studio the Screen can be partitioned according to the TV-II window/sub-window structure
- Several Render Target Types
  - (Sub-) Window Surface → IBO mode
    - Double Buffered (Surface Size * Byte/px * 2 buffer count)
    - Use images from Hyperflash (Client memory) to save VRAM
  - Image (Sub-) Window Surface
    - Surface from (compressed) Image in Hyperflash (Client memory) to save VRAM
  - LBO (Sub-) Window Surface
    - Double Buffered (Surface Size * Byte/px * 2 + 16 lines * Byte/px)
    - Bandwidth usage improvement versus higher Memory usage
    - Images shall be in VRAM.
  - OTF Window
    - No Frame Buffer (at least 16 lines (or the next power of 2!) * Byte/px for Scanline)
    - Images shall be located VRAM
- **Best practice:**
  - Set the task priority and task "COPSes" accordingly in the related CMake-settings for the CGI studio build:




  - Set the Line-buffers for OTF layers accordingly to the required solution setup. In general: use as less as number of line buffers (line buffer number needs to be a power of 2) because this will increase the VRAM consumption accordingly. Increase only if you face performance issues (e.g., red flicker on screen)

- Use as less as possible OTF layers. If possible, just use 1 (but keep the limitation with maximum width of 1600 in mind)
- Try to use an image Layer whenever possible for static background (see following examples)

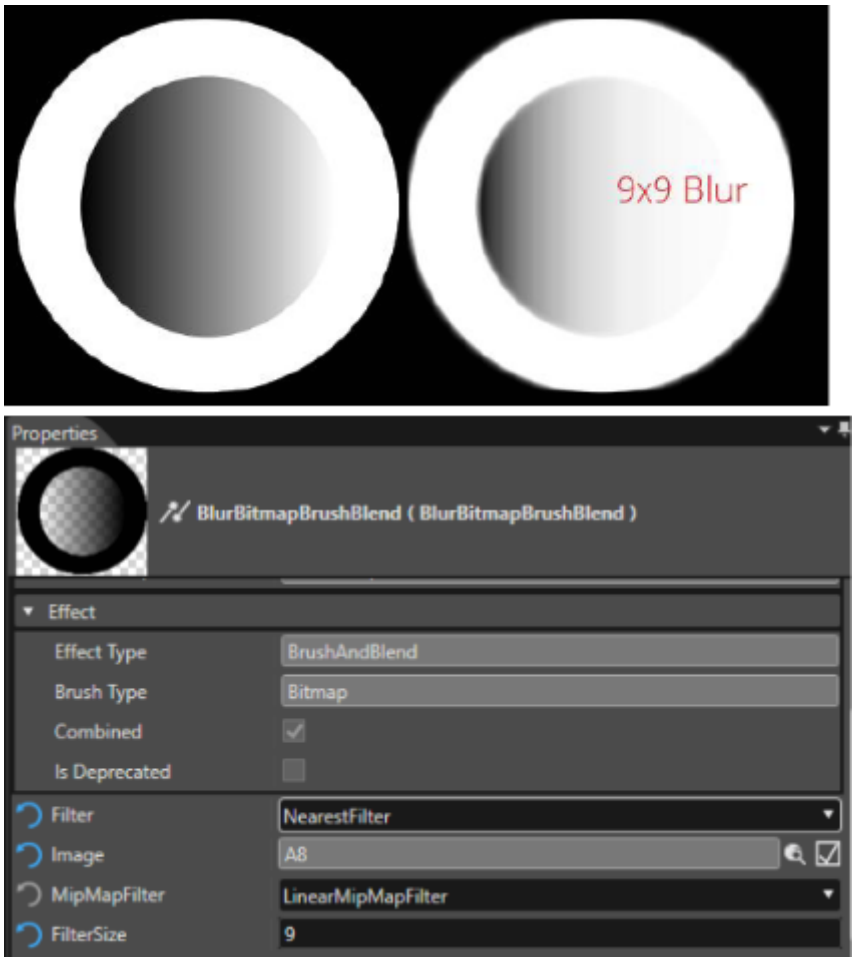




# Effects

**Best practice:**

- Use Blur-effect only if not avoidable, Multi pass BLITS like the Blur effect have a huge performance impact because the image must be blitted multiple times for a 9x9 filter!




- All images for bitmap brush perspective warp blend must be in VRAM.