Vircon32

32-BIT VIRTUAL CONSOLE

System specification

Part 4: The graphics chip (GPU)

Document date 2024.01.06 Written by Carra

What is this?

This document is part number 4 of the Vircon32 system specification. This series of documents defines the Vircon32 system, and provides a full specification describing its features and behavior in detail.

The main goal of this specification is to define a standard for what a Vircon32 system is, and how a gaming system needs to be implemented in order to be considered compliant. Also, since Vircon32 is a virtual system, an important second goal of these documents is to provide anyone with the knowledge to create their own Vircon32 implementations.

About Vircon32

The Vircon32 project was created independently by Carra. The Vircon32 system and its associated materials (including documents, software, source code, art and any other related elements) are owned by the original author.

Vircon32 is a free, open source project in an effort to promote that anyone can play the console and create software for it. For more detailed information on this, read the license texts included in each of the available software.

About this document

Summary

Part 4 of the specification defines the console's graphics chip (GPU). This document will describe the behavior of this chip, its control ports, its provided drawing commands and the process it uses to produce video output.

1 Introduction

The GPU is the video chip in the Vircon32 console. It is responsible for drawing everything that can be seen on the screen. To do this it will use the textures contained in both the BIOS and the cartridge and apply basic graphical effects to them.

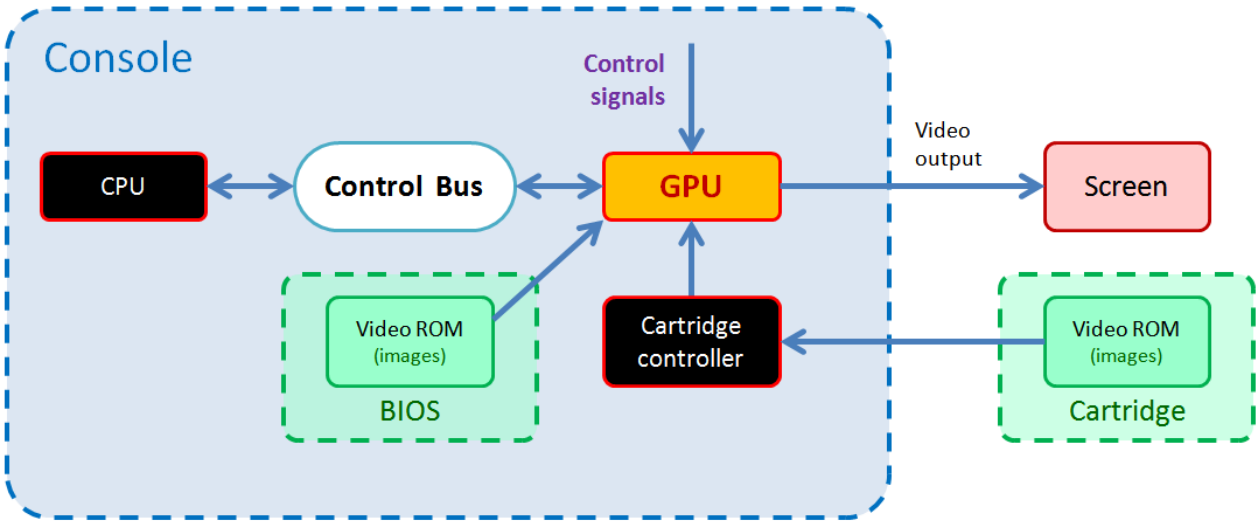
This graphics chip is very simplified: there are only 2 types of actions that the GPU can do to draw on screen:

- It can clear the screen with a constant color.
- It can draw on screen a region from one of the available textures.

Both of them will be executed as drawing commands requested by the processor. Aside from this there are a few graphical effects that can be applied to these actions, that will be covered in later sections.

2 External connections

Since the GPU is just one of the chips forming the console, it cannot operate in isolation. This figure shows all communications of the GPU with other components. As shown, the GPU has connections to all available video ROMs to be able to use their contained images as textures. Note that, for clarity, the console diagrams in part 2 of the specification purposely omitted these connections.



Each of these connections will be explained individually in the sections below.

2.1 Control signals

As all console components, the GPU receives the signals for reset, new frame and new cycle. The responses to those signals are detailed in section 12 of this document.

2.2 Control Bus

The GPU is connected as slave device to the Control Bus, with device ID = 2. This allows the bus master (the CPU) to request read or write operations on the control ports exposed by the GPU. The list of GPU ports as well as their properties will be detailed in later sections.

2.3 BIOS chip

The BIOS, which is always present, contains a video ROM with exactly 1 image. The GPU can access that image through its texture slot with ID = –1.

2.4 Cartridge controller

Same as with the BIOS, the GPU can access any images present in the cartridge through texture slots with IDs 0 to 255. Note however that the connection must use the Cartridge Controller as a proxy, since there may not be a cartridge present. And even when a cartridge is connected, each of them can contain a different number of images (0 to 256).

2.5 Screen

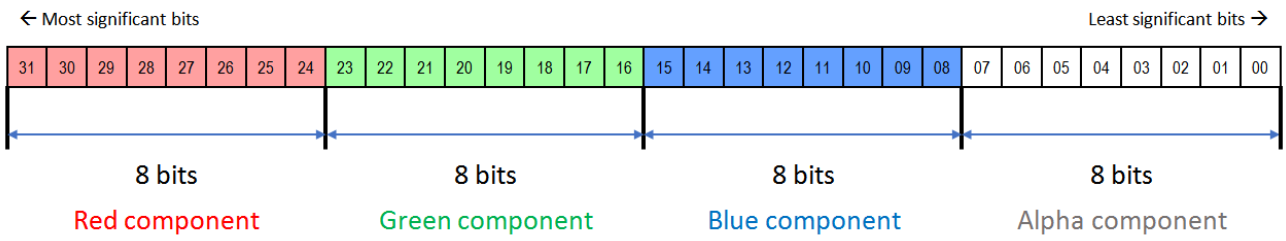
Every frame, once it has finished drawing, the GPU needs to be able to send the result to the screen so that it can be seen. This is done via the video output connection. Section 11 will cover how this output is done.

3 Working concepts

Before explaining the GPU's graphic functions or the internal variables that affect them, we must present a series of basic concepts that the GPU is built around.

3.1 Pixels and colors

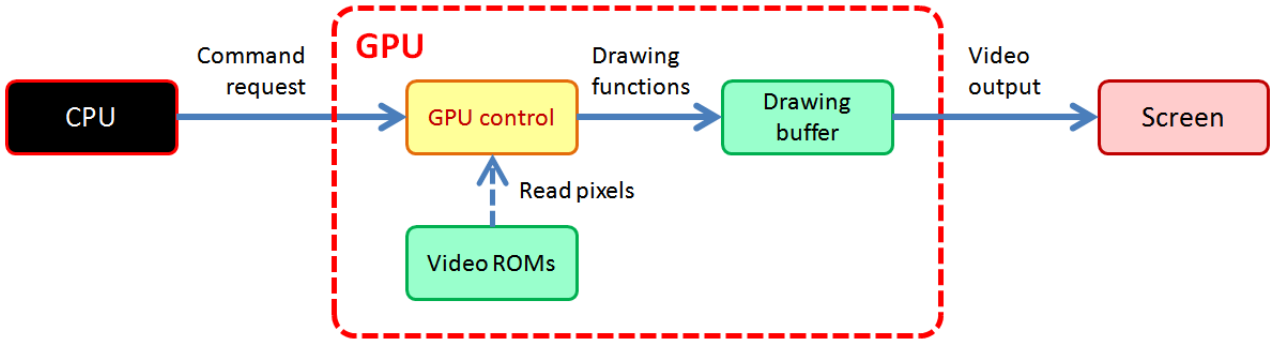
The minimal unit of video information handled by the GPU is the pixel, which represents a single discrete position in either the screen or a texture. Every pixel is represented as a single 32-bit RGBA color, as already documented in part 2 of the specification:



3.2 The drawing buffer

The drawing buffer is a rectangular 2D array of pixels, the same size of the screen (640 x 360 pixels). Each individual pixel is identified by its position from the top-left corner, i.e. from (0, 0) to (639, 359).

The GPU uses this buffer as a proxy for the screen itself: the drawing buffer's contents keep being modified by every GPU graphic operation. Then, for every new frame, the video output signal is updated with the current contents of the drawing buffer and that makes any changes from the last frame become visible. As a whole, the operation of the GPU over the screen buffer looks like this:



Note that the alpha component of a pixel only needs to be used during drawing operations, for the process of blending colors onto the drawing buffer. After this, the drawing buffer itself has no need to store the alpha component of its pixels. It is up to the implementation to decide if pixel format for the drawing buffer uses 32 bits or just 24 bits with alpha omitted. In any case, pixels in the drawing buffer can be assumed to always be full opacity (alpha = 255).

3.3 GPU textures

A GPU texture is a square, 2D pixel array with a fixed size of 1024 x 1024 pixels. Coordinates for pixels in a texture begin at the top-left corner, which is pixel (0, 0). Aside from clearing the screen, all GPU drawing operations need to use a texture to draw.

The GPU identifies and accesses its available textures through an array of numbered texture slots. There are 256 slots usable by the cartridge (with texture IDs from 0 to 255), and an additional slot with ID = –1 for the BIOS texture.

3.4 Connection to video ROMs

The GPU does not contain any textures itself, so it needs to read the images stored in the BIOS and cartridges by connecting to their respective video ROMs. A video ROM is a read-only memory region that contains a sequence of images. Each of these images can have any possible width and height, from 1 x 1 pixels up to the GPU texture size.

When a cartridge containing N images is connected, the first N slots from ID = 0 become assigned to each of the images, in order. The rest of slots up to ID = 255 will become unused and not accessible. The slot with ID = -1 gets assigned to the BIOS image, which is guaranteed to exist and be unique.

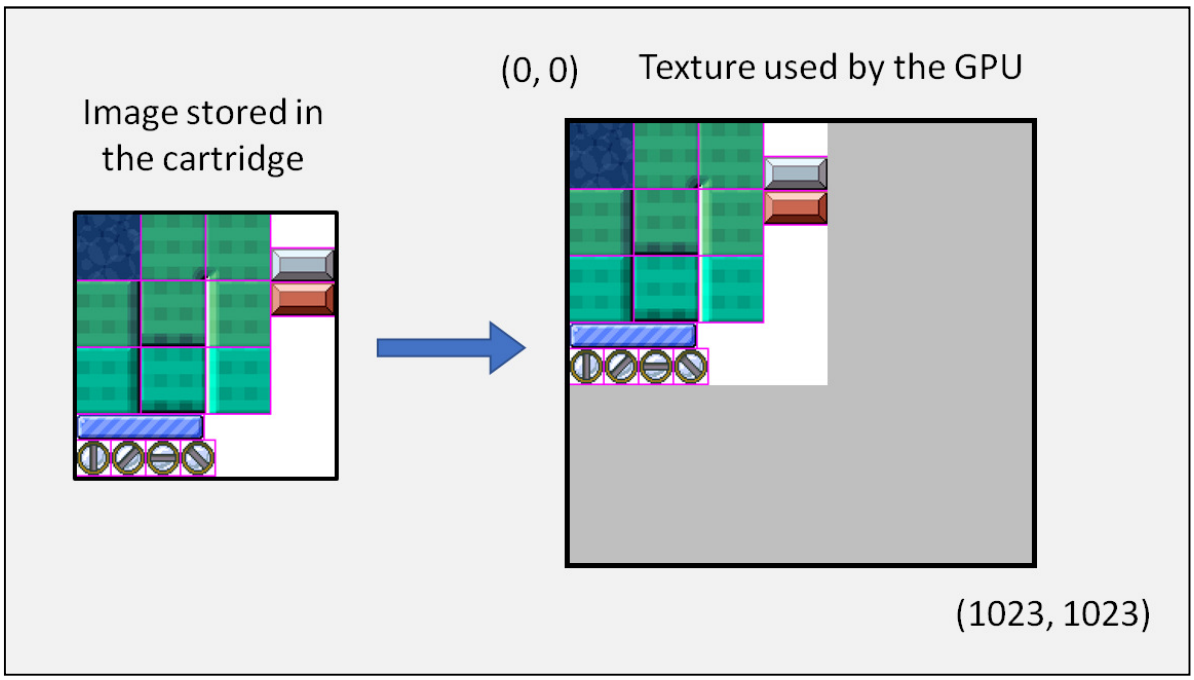
This connection process happens whenever a new cartridge is inserted. Still, in Vircon32 systems, cartridges can only be inserted while the console is off. Therefore, it is safe for implementations to delay any needed procedures for the GPU to establish this connection until the next console power on.

It is up to the implementation to decide how to establish a connection to video ROMs and locate and read their images. For example, it is possible to read all images in video ROMs beforehand upon connection. Another option would be to keep pointers to images and have the GPU will read pixel values on the fly.

Extending images

As stated before, GPU textures are a fixed size. However, slots read their pixels from images that may be of different sizes each. To solve this problem, each texture slot will adapt its assigned image to be usable as a GPU texture by automatically extending their size to 1024 x 1024, and considering the rest of pixels empty (i.e. their 4 components are 0). Extension is done at the bottom right, so the coordinates of all pixels are preserved.

This image shows how an image stored in video ROM would be extended with empty pixels (shown here in gray color for clarity).

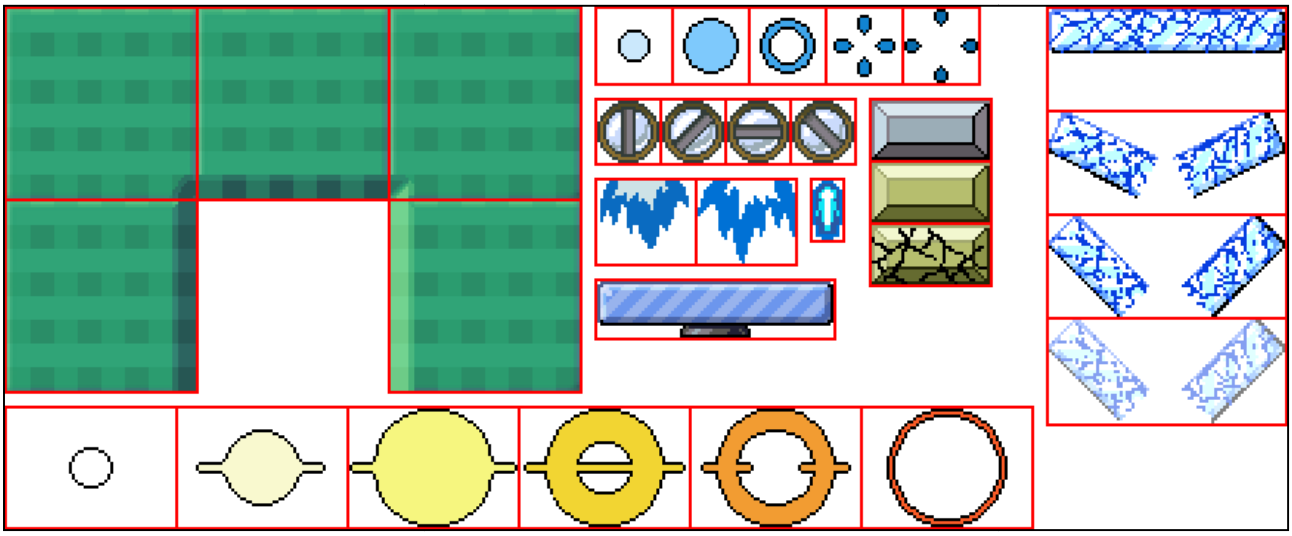


Again, the methods used by texture slots to read and extend images (pixel mapping, internal copies, hardware connections, etc…) are also up to the implementation.

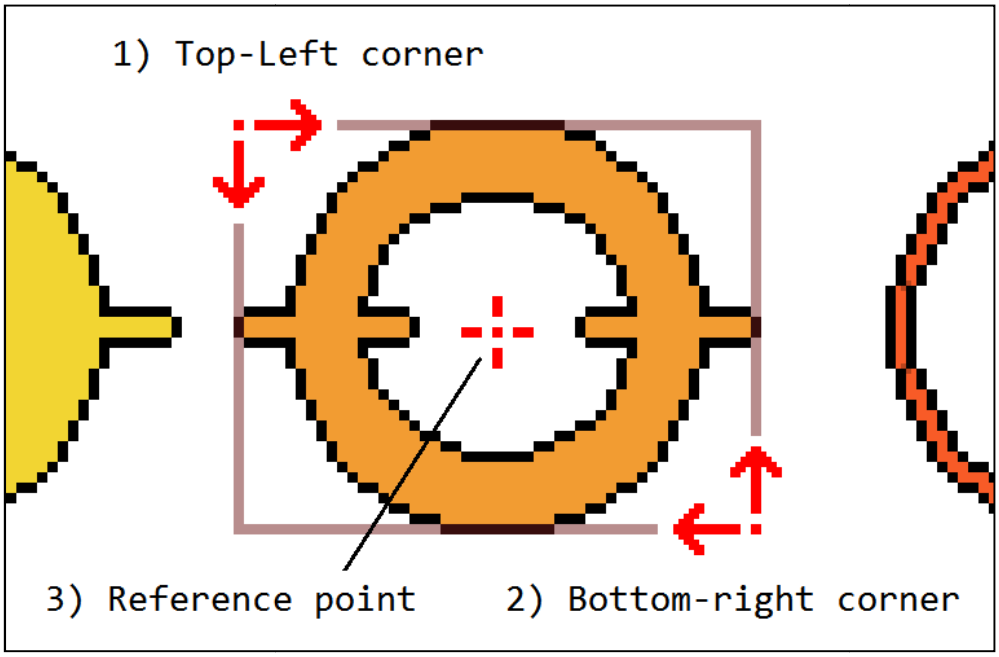
3.5 Texture regions

GPU textures are larger than the screen, so the GPU does not work with textures directly. Instead, a set of delimited regions can be defined within each texture so that GPU drawing functions operate with them.

Defining regions for each texture allows Vircon32 programs to use textures more efficiently by grouping several images as part of a single texture. An example of this strategy can be seen in the sample below:



A texture region is a rectangular part of that texture, defined by the coordinates of 3 pixels as seen in the following image. Pixels 1 and 2 define the region limits (both are included in the region) whereas pixel 3, commonly called the hotspot, is used as a reference to place the region on the screen when drawn. The hotspot can be outside of the region limits.

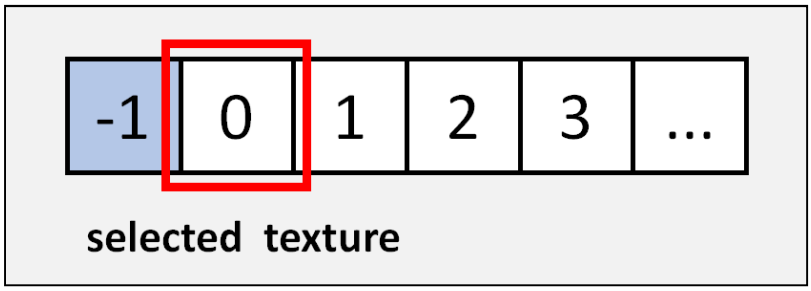


3.6 Selected elements

The GPU organizes its graphic elements (textures and regions) in sets, but typically it will only operate with one element from each set at any given time.

Selected texture

All assigned texture slots form a continuous range of usable texture IDs as shown in this image. Still, only 1 of the available textures will be active at any time. To determine which texture to use when drawing or using any other functions, the GPU always considers one of those IDs as "selected". The texture selected by default is the BIOS texture, since it is the only one guaranteed to always exist.



Selected region

For each assigned texture slot there are 4096 configurable regions, accessible through region IDs 0 to 4095. Note that all of these will always exist and be usable, even if the program has not defined their values. Same as with textures, the GPU will always consider one of the region IDs as "selected", to determine which region is used by the GPU functions. Note however, that this parameter is global: there is not a different selected region for each texture.

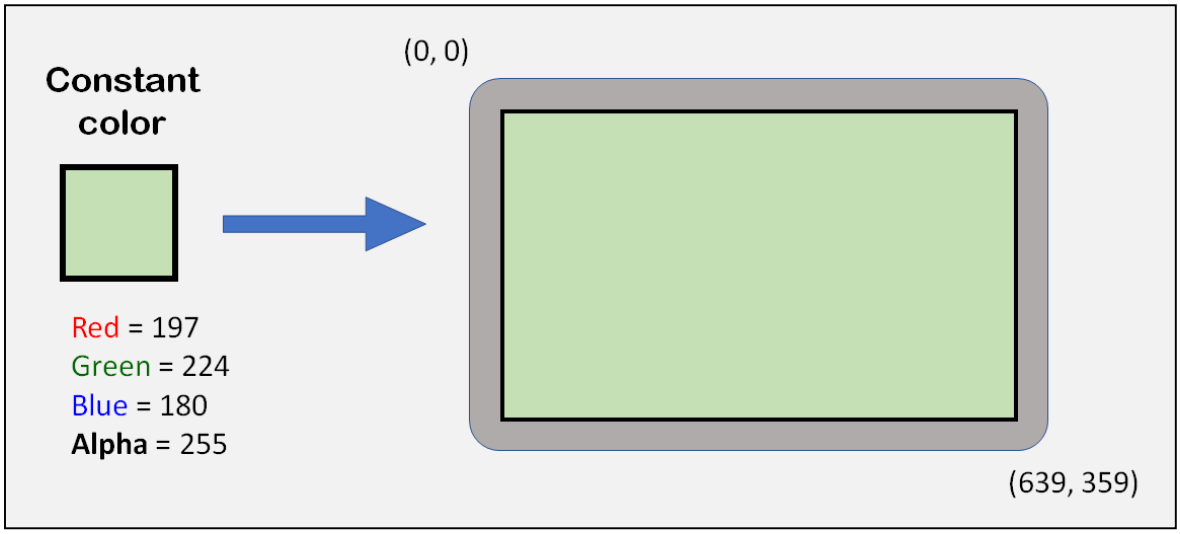
4 Drawing functions

As mentioned in the introduction section, there are only 2 ways the GPU can draw on the screen: either clearing the screen or drawing a texture region. This section will provide a basic description of how these functions are performed.

Note however that the full detail of how graphic functions are performed will be specified later, in the section covering command execution. Also, while it is possible to apply certain modifying effects, this description will only describe the basic unmodified version.

4.1 Clearing the screen

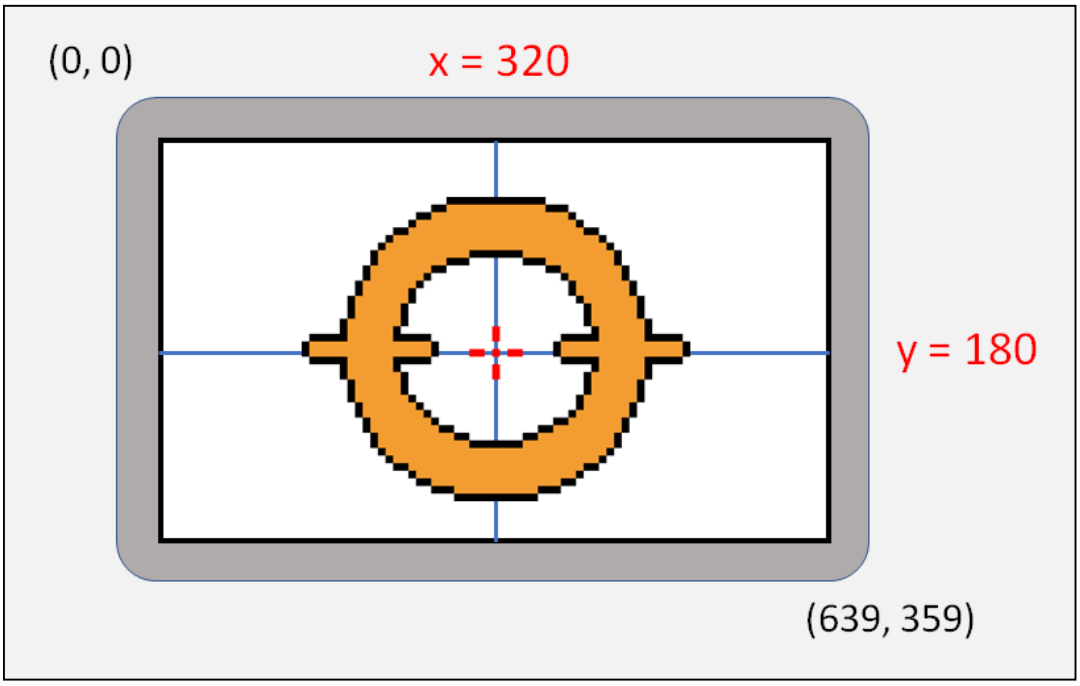
A screen clear will apply a single color (the clear color) over all pixels of the drawing buffer. Clearing the screen is the simplest function: it operates without accessing any textures, and it can't be modified using graphic effects.



4.2 Drawing texture regions

The GPU is able to draw any of the currently available texture regions on the drawing buffer. To determine where to place that region it uses a drawing position, given by its pixel coordinates. The region is drawn so that its hotspot or reference point is placed at said drawing position.

For example, if the drawing position is the center of the screen (pixel 320,180), the reference point of the sample region we defined earlier will be made to match these coordinates as in the example below.

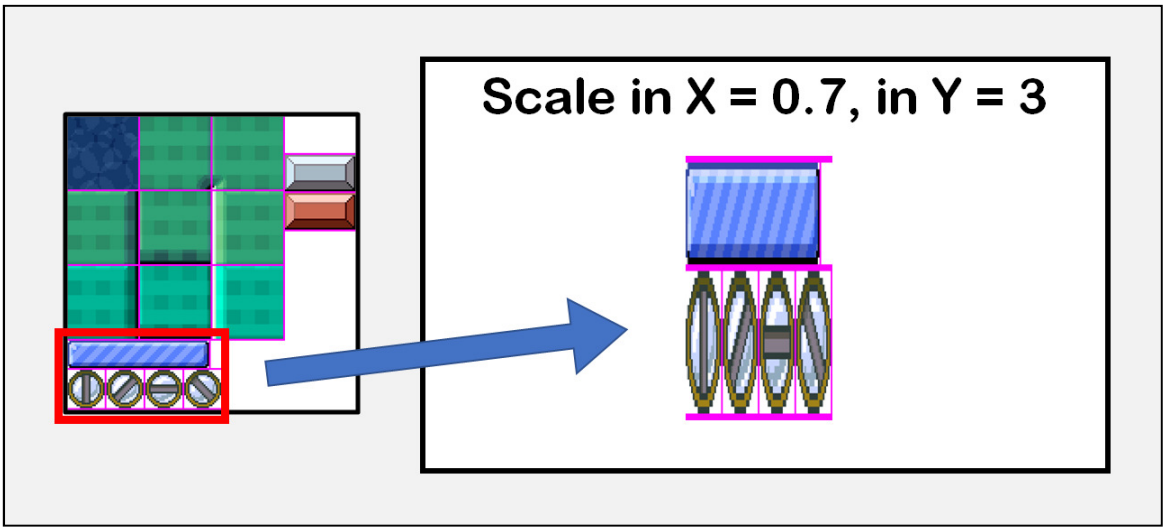


5 Graphic effects

To increase flexibility and achieve more advanced graphic features, different types of effects can be applied to the GPU drawing functions. These effects, when enabled, will modify the way drawing functions work and therefore change the result seen on screen.

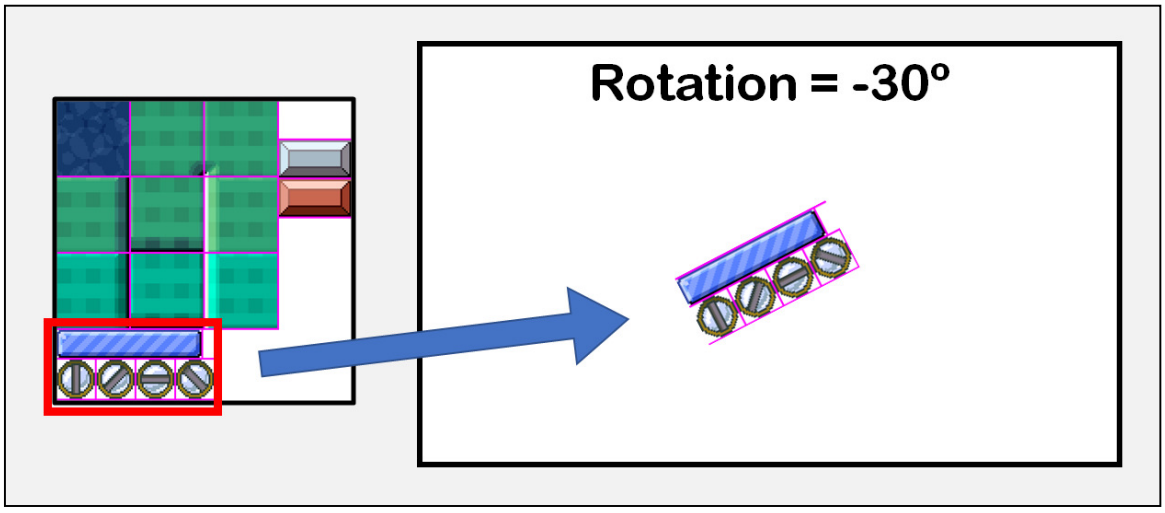
5.1 Scaling effect

When drawing a texture region the GPU can optionally be instructed to apply scaling factors along the region's X and Y axes. As a result the output rectangle gets scaled accordingly, preserving the hotspot position. An example of this effect could look like this:



5.2 Rotation effect

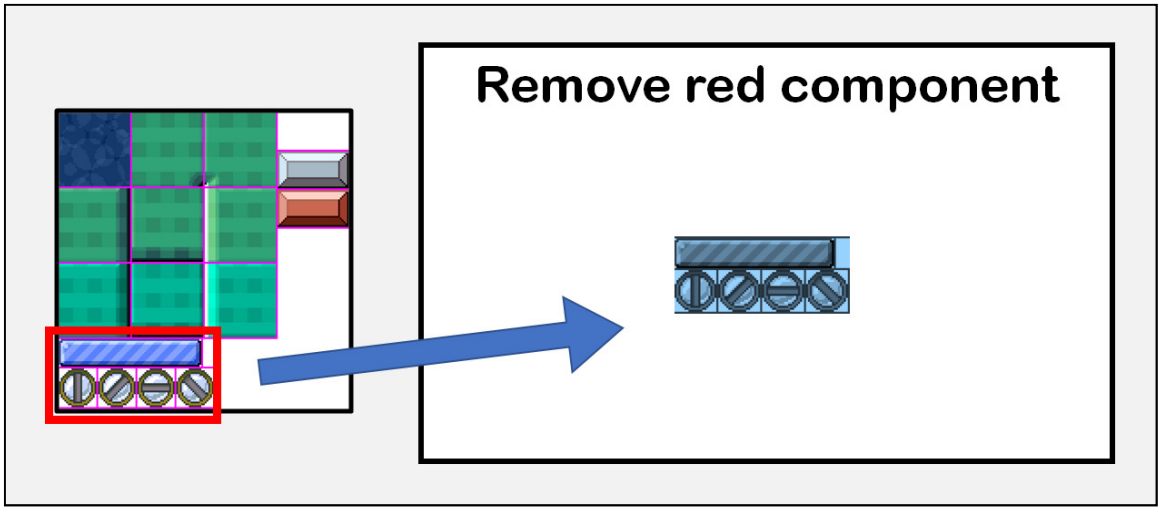
Another optional effect available when drawing a texture region is to apply a rotation angle. The drawn region will be rotated with respect to its hotspot, which again conserves its position on screen. An example of a rotated region could be the following:



5.3 Color multiplication effect

This effect is always applied whenever regions are drawn. The color components of every pixel to be drawn are first multiplied by those of a single given color (the multiply color). The effect of the multiply color is that of a filter: it will modulate the 4 color components of every region pixel before they are drawn to the screen. This effect is the same as when using glColor function in OpenGL.

As an example of this effect, if we use a multiply color of (0, 255, 255, 255), this would remove the red component of the drawn region while preserving the other components. The visual effect shown on screen would look like this:



For each region pixel, the color to be drawn is modified according to these formulas:

```
Drawn R = (Pixel R * Multiply R) / 255
Drawn G = (Pixel G * Multiply G) / 255
Drawn B = (Pixel B * Multiply B) / 255
Drawn A = (Pixel A * Multiply A) / 255
```

The modulation effect for each component could be visualized better if these formulas were written as a proportion: Pixel X * (Multiply X / 255). However note that, for some implementations, this order of operations may cause the result to be incorrect.

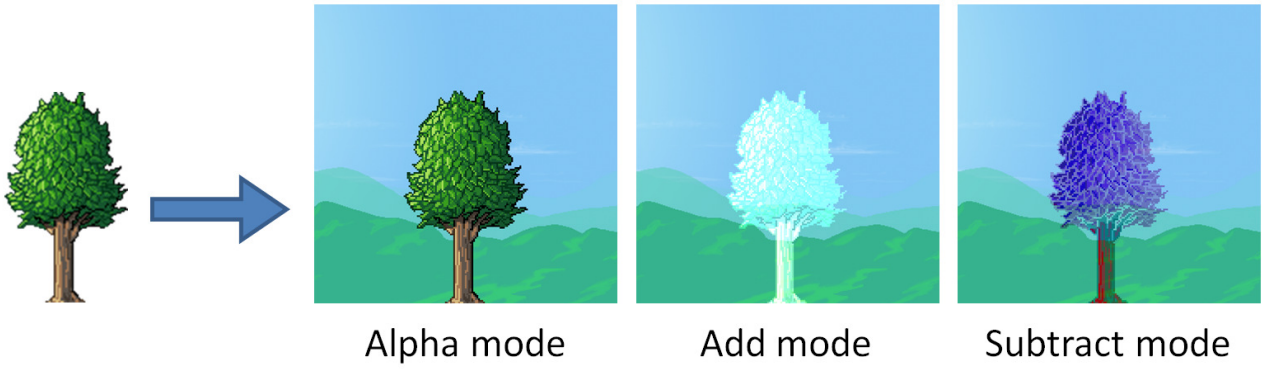
When the multiply color is full-opacity white, it has no effect on drawn colors. This is the default so, in practice, when the console starts up this effect can be considered “disabled”.

6 Color blending

Color blending is the process used to draw a new color on some pixel from the drawing buffer. It is necessary to define this process because Vircon32 graphics support basic transparencies and lighting effects. In such a graphic system, drawing a new color does

not always mean that the drawn color will simply replace the previous pixel color. Instead it may also depend on the previous pixel color, in different ways.

The GPU supports 3 different color blending modes. At any given time one of the 3 modes will be active, and any performed graphic functions will use that mode to draw to the drawing buffer. The following image can give a quick idea of what the 3 modes look like:



6.1 Alpha blending

This is the default blending mode. It just draws colors as they are, while taking into account the alpha channel to determine the level of transparency over the background. In its most basic form (full opacity, no transparencies), its effect would just be: Buffer Color = Drawn Color. When we take alpha into account the effect is modulated in this way:

```
Buffer R = (Drawn R * Drawn A + Buffer R * (255 - Drawn A)) / 255
Buffer G = (Drawn G * Drawn A + Buffer G * (255 - Drawn A)) / 255
Buffer B = (Drawn B * Drawn A + Buffer B * (255 - Drawn A)) / 255
```

The result will already be within range [0-255], so no clipping is required. Any drawn color with full transparency produces no changes.

6.2 Addition blending

This mode is used to brighten colors and produce light effects. In its most basic form (full opacity), its effect would just be: Buffer Color += Drawn Color, then clip to [0-255]. When we add an alpha channel, the effect is modulated in this way:

```
Buffer R =  Buffer R + (Drawn R * Drawn A) / 255
Buffer G =  Buffer G + (Drawn G * Drawn A) / 255
Buffer B =  Buffer B + (Drawn B * Drawn A) / 255
```

The resulting 3 RGB components are then capped to a maximum of 255. When drawn color is either black or fully transparent, it produces no changes. This blending mode

should be analogous to the layer modes called “add” or “linear dodge” in many drawing programs.

6.3 Subtraction blending

This blending is used to darken colors and produce shadow effects. In its most basic form (full opacity), its effect would just be: Buffer Color –= Drawn Color, then clip to [0-255]. When we add an alpha channel, the effect is modulated in this way:

```
Buffer R =  Buffer R - (Drawn R * Drawn A) / 255
Buffer G =  Buffer G - (Drawn G * Drawn A) / 255
Buffer B =  Buffer B - (Drawn B * Drawn A) / 255
```

The resulting 3 RGB components are then capped to a minimum of 0. When drawn color is either black or fully transparent, it produces no changes. This blending mode should be analogous to the layer modes called “subtract” or “difference” in many drawing programs.

7 GPU performance

The GPU would not be completely defined without establishing its performance limits. For this GPU, performance is based on a rough estimation of the amount of pixels drawn. Each frame the GPU is allowed a drawing capacity of 9 times the full screen. This is the same amount of pixels as 1 full screen of 1920 x 1080 pixels.

7.1 Calculation of drawn pixels

The first thing the GPU needs to do after receiving a valid drawing operation request is to perform a simplified estimation of the number of pixels that will be drawn.

For a clear screen command the amount of drawn pixels is always 1 full screen = 640 x 360 pixels. When drawing regions the calculation process follows these steps:

1) Calculate the "effective width” as follows: First take the region width in texture pixels. If scaling is applied, multiply that by the scaling factor in X. Then take the absolute value and, if the result is greater than screen width, limit it to 640 pixels.

2) Calculate the "effective height": as follows: First take the region height in texture pixels. If scaling is applied, multiply that by the scaling factor in Y. Then take the absolute value and, if the result is greater than screen height, limit it to 360 pixels.

3) Finally calculate “drawn pixels” as "effective width" x "effective height".

Note that this calculation ignores the drawing position and rotation angle of regions. This means that any parts of a region that fall outside the screen will also be counted as drawn pixels. However this inaccuracy greatly simplifies the calculation process in some cases.

7.2 Performance factors for different operations

Not all operations are equally expensive for the GPU. To model this, depending on the operation performed, the number of drawn pixels will modified by these factors:

- When clearing the screen: -50%
- When drawing regions with scaling applied: +15%
- When drawing regions with rotation applied: +25%

(if both scaling and rotation are applied, the combined factor is +40%)

Note that color processing operations, such as applying the multiply color and color blending, do not affect performance and have no associated factors.

7.3 Checking consumption for a command

After determining the number of pixels to draw for the received command request, the GPU will compare that amount with its current remaining pixels.

- If there are enough remaining pixels for the drawing operation in this frame then it will perform the command and reduce the remaining pixels as calculated.

- If, on the contrary, the current remaining pixels are not enough for the operation, the GPU will instead set its remaining pixels to -1 and not perform the command. Any further requests received within the current frame will also be ignored.

Note that, because of this last point, implementations can simplify this process and automatically reject operations without any calculation when remaining pixels are less or equal than 0.

The GPU drawing capacity cannot be transferred between frames. A new frame will restore the same pixel capacity regardless of the pixels remaining in the previous frame.

8 Internal variables

The GPU features a set of variables that store different aspects of its internal state. These variables are each stored as a 32-bit value, and they are all interpreted using the same data formats (integer, float, etc) described in part 2 of the specification. Here we will list and detail all of them, organized into sections.

8.1 Variables for GPU control

Remaining pixels	**Initial value:** 2073600 (= 9 * 640 * 360)
Format: Integer	**Valid range:** From -1 to 2073600

Stores the remaining drawing capacity of the GPU for this frame. When a drawing request cannot be performed (i.e. remaining pixels are insufficient to perform the command), the GPU becomes blocked until the next frame begins and will no longer perform any commands in the current frame. A value of -1 is used to indicate that drawing capacity for this frame has been exhausted.

8.2 Spatial drawing parameters

Drawing point X	**Initial value:** 0
Format: Integer	**Valid range:** From -1000 to 1639

It marks the X coordinate, in pixels, on which the next drawn region's hotspot will be placed on screen. Note that it may fall out of the screen width, and this may result in partial or total visibility loss of the drawn region.

Drawing point Y	**Initial value:** 0
Format: Integer	**Valid range:** From -1000 to 1359

It marks the Y coordinate, in pixels, on which the next drawn region's hotspot will be placed on screen. Note that it may fall out of the screen height, and this may result in partial or total visibility loss of the drawn region.

Drawing scale X	**Initial value:** 1.0
Format: Float	**Valid range:** From -1024.0 to 1024.0

This drawing parameter will only be applied with commands that enable scaling. When used, drawn regions will be scaled along the texture's X dimension by this factor. It also can be negative, for a mirror effect along X.

When drawing with a reduction scale that would produce an output width of less that 1 pixel, the result is considered implementation dependent: it could either compress the region into a 1-pixel vertical line, or draw nothing at all.

Drawing scale Y	**Initial value:** 1.0
Format: Float	**Valid range:** From -1024.0 to 1024.0

This drawing parameter will only be applied with commands that enable scaling. When used, drawn regions will be scaled along the texture's Y dimension by this factor. It also can be negative, for a mirror effect along Y.

When drawing with a reduction scale that would produce an output height of less that 1 pixel, the result is considered implementation dependent: it could either compress the region into a 1-pixel horizontal line, or draw nothing at all.

Drawing angle	**Initial value:** 0.0
Format: Float	**Valid range:** From -1024.0 to 1024.0

This drawing parameter will only be applied with commands that enable rotation. This is the rotation angle that will be applied to regions drawn on the screen. The value is interpreted in radians (1 full circle = 2*pi radians = 360 degrees). An angle of 0 means no rotation, and for positive values regions are rotated clockwise.

8.3 Color processing variables

Clear color	**Initial value:** R = 0, G = 0, B = 0, A = 255
Format: GPU color	**Valid range:** The full RGBA range

The current clear color is the one that will be applied whenever a clear screen command is performed. It supports transparency by modulating the alpha component.

Multiply color	**Initial value:** R = 255, G = 255, B = 255, A = 255
Format: GPU color	**Valid range:** The full RGBA range

This color controls the color multiply effect that is always applied when performing any region drawing commands. The initial value is the neutral multiply color (no effect).

Active blending mode	**Initial value:** 20h (Alpha blending)
Format: Integer	**Valid range:** Only the values listed

This value is interpreted as the currently active color blending mode, which controls how colors are drawn in all drawing functions. Refer to the color blending section for the details of how each mode works. The possible values are the following:

20h: Alpha blending
21h: Addition blending
22h: Subtraction blending

8.4 Selected elements

Selected texture	**Initial value:** -1
Format: Integer	**Valid range:** From -1 to 255 (*)

This value is the numerical ID of the currently selected GPU texture. The selected texture is the one that will be used in all region drawing commands. It is also the texture which regions are affected by any region configuration changes.

(*) The upper limit is given by the currently connected cartridge. If no cartridge is present, then the BIOS texture (ID = -1) is the only selectable texture.

Selected region	**Initial value:** 0
Format: Integer	**Valid range:** From 0 to 4095

This value is the numerical ID of the currently selected GPU region, within the selected texture. Note that changing the selected texture does not change the selected region ID, since it is a global parameter (there is not a selected region for each texture). The selected region (from the currently selected texture) is the one that will be used in all region drawing commands. It is also the region affected by any region configuration changes.

8.5 Configuration of each texture region

The variables listed here are special. The GPU stores a copy of each of these variables for each possible texture region. This means there can be as many as (256+1) textures * 4096 regions/texture. The implementation may decide if all these are always stored, keeping the ones for unused texture slots inaccessible, or if only the needed ones are created each time a new cartridge is inserted.

Together, each set of these variables describes the current configuration for a single GPU region. See the section for GPU texture regions in chapter 3 for an explanation of how regions are defined. Note that the GPU will accept that the minimums and maximums that define region extensions are reversed, and that will produce an image flip effect.

Only one set of these variables is accessible at any time, so each variable in this section is accessed by control ports via a “pointer” proxy. When the selected texture or region change, those ports are all redirected to the copy of the variables for the correct region.

Region minimum X	**Initial value:** 0
Format: Integer	**Valid range:** From 0 to 1023

It represents the leftmost X coordinate from the texture that will be considered part of the region. Given in pixels, in texture coordinates.

Region minimum Y	**Initial value:** 0
Format: Integer	**Valid range:** From 0 to 1023

It represents the topmost Y coordinate from the texture that will be considered part of the region. Given in pixels, in texture coordinates.

Region maximum X	**Initial value:** 0
Format: Integer	**Valid range:** From 0 to 1023

It represents the rightmost X coordinate from the texture that will be considered part of the region. Given in pixels, in texture coordinates.

Region maximum Y	**Initial value:** 0
Format: Integer	**Valid range:** From 0 to 1023

It represents the bottommost Y coordinate from the texture that will be considered part of the region. Given in pixels, in texture coordinates.

Region hotspot X	**Initial value:** 0
Format: Integer	**Valid range:** From -1024 to 2047

It is the X coordinate, in texture coordinates, that will be taken as a reference when drawing the region. The GPU will calculate region placement so that the hotspot will be drawn at the drawing point. It is allowed for hotspots to be out of texture bounds, within the given range limits.

Region hotspot Y	**Initial value:** 0
Format: Integer	**Valid range:** From -1024 to 2047

It is the Y coordinate, in texture coordinates, that will be taken as a reference when drawing the region. The GPU will calculate region placement so that the hotspot will be drawn at the drawing point. It is allowed for hotspots to be out of texture bounds, within the given range limits.

9 Control ports

This section details the set of control ports exposed by the GPU via its connection to the CPU control bus as a slave device. All exposed ports, along with their basic properties are listed in the following table:

List of exposed control ports			
External address	Internal address	Port name	R/W access
200h	00h	Command	Write Only
201h	01h	Remaining Pixels	Read Only
202h	02h	Clear Color	Read & Write
203h	03h	Multiply Color	Read & Write
204h	04h	Active Blending	Read & Write
205h	05h	Selected Texture	Read & Write
206h	06h	Selected Region	Read & Write
207h	07h	Drawing Point X	Read & Write
208h	08h	Drawing Point Y	Read & Write
209h	09h	Drawing Scale X	Read & Write
20Ah	0Ah	Drawing Scale Y	Read & Write
20Bh	0Bh	Drawing Angle	Read & Write
20Ch	0Ch	Region Min X	Read & Write
20Dh	0Dh	Region Min Y	Read & Write
20Eh	0Eh	Region Max X	Read & Write
20Fh	0Fh	Region Max Y	Read & Write
210h	10h	Region Hotspot X	Read & Write
211h	11h	Region Hotspot Y	Read & Write

9.1 Behavior on port read/write requests

The GPU control ports are not simply hardware registers. The effects triggered by a read/write request to a specific port will often be different than reading or writing values. This section details how each of the GPU ports will behave.

Note that, in addition to the actions performed, a success/failure response will need to be provided to the request as part of the control bus communication. This response will always be assumed to be success, unless otherwise specified. When the provided response is failure, the GPU will perform no further actions and the CPU will trigger a HW error.

Command port

On read requests:

This port is write-only, so a failure response will be provided to the control bus.

On write requests:

The GPU will perform the command execution process detailed in section 10. In all cases the request will be responded with success, even if the command is not executed.

Remaining Pixels port

On read requests:

The GPU will provide the current value of the internal variable “Remaining pixels”.

On write requests:

This port is read-only, so a failure response will be provided to the control bus.

Clear Color port

On read requests:

The GPU will provide the current value of the internal variable “Clear color”.

On write requests:

The GPU will overwrite the internal variable “Clear color” with the received value. This will immediately cause next screen clear operations to apply the new clear color. Note that setting this variable will not cause a screen clear.

Multiply Color port

On read requests:

The GPU will provide the current value of the internal variable “Multiply color”.

On write requests:

The GPU will overwrite the internal variable “Multiply color” with the received value. This will immediately cause next region drawing operations to apply the new multiply color.

Active Blending port

On read requests:

The GPU will provide the current value of the internal variable “Active blending mode”.

On write requests:

The GPU will check if the received value corresponds to a valid blending mode. In case it is not, the request will be ignored. For valid values, the GPU will overwrite the internal variable “Active blending mode” with the received value. This will immediately cause next drawing operations to apply the new blending mode.

Selected Texture port

On read requests:

The GPU will provide the current value of the internal variable “Selected texture”.

On write requests:

The GPU will check if the received value corresponds to a currently valid texture ID. In case it is not, the request will be ignored. For valid values, the GPU will overwrite the internal variable “Selected texture” with the received value. After that, it will redirect all region configuration ports to point to the set of variables for the new selected region (i.e. the same region ID, but taken from the new selected texture).

Selected Region port

On read requests:

The GPU will provide the current value of the internal variable “Selected region”.

On write requests:

The GPU will check if the received value corresponds to a valid region ID. In case it is not, the request will be ignored. For valid values, the GPU will overwrite the internal variable “Selected region” with the received value. After that, it will redirect all region configuration ports to point to the set of variables for the new selected region (i.e. the new region ID, but taken from the same selected texture).

Drawing Point X port

On read requests:

The GPU will provide the current value of the internal variable “Drawing point X”.

On write requests:

The GPU will check if the received value is out of range for the internal variable “Drawing point X” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Drawing point X”. This will immediately cause next drawing operations to apply the new drawing position.

Drawing Point Y port

On read requests:
The GPU will provide the current value of the internal variable “Drawing point Y”.

On write requests:
The GPU will check if the received value is out of range for the internal variable “Drawing point Y” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Drawing point Y”. This will immediately cause next drawing operations to apply the new drawing position.

Drawing Scale X port

On read requests:
The GPU will provide the current value of the internal variable “Drawing scale X”.

On write requests:
The GPU will check if the received value is out of range for the internal variable “Drawing scale X” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Drawing scale X”. This will immediately cause next drawing operations with scaling enabled to apply the new drawing scale in X.

Drawing Scale Y port

On read requests:
The GPU will provide the current value of the internal variable “Drawing scale Y”.

On write requests:
The GPU will check if the received value is out of range for the internal variable “Drawing scale Y” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Drawing scale Y”. This will immediately cause next drawing operations with scaling enabled to apply the new drawing scale in Y.

Drawing Angle port

On read requests:
The GPU will provide the current value of the internal variable “Drawing angle”.

On write requests:

The GPU will check if the received value is out of range for the internal variable “Drawing angle” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Drawing angle”. This will immediately cause next drawing operations with rotation enabled to apply the new drawing angle.

Region Min X port

On read requests:

The GPU will provide the current value of the internal variable “Region minimum X” associated to the currently selected region ID, for the currently selected texture ID.

On write requests:

The GPU will check if the received value is out of range for the internal variable “Region minimum X” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Region minimum X” associated to the currently selected region ID, for the currently selected texture ID. This will immediately cause next region drawing operations to apply the new left limit for the selected region.

Region Min Y port

On read requests:

The GPU will provide the current value of the internal variable “Region minimum Y” associated to the currently selected region ID, for the currently selected texture ID.

On write requests:

The GPU will check if the received value is out of range for the internal variable “Region minimum Y” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Region minimum Y” associated to the currently selected region ID, for the currently selected texture ID. This will immediately cause next region drawing operations to apply the new top limit for the selected region.

Region Max X port

On read requests:

The GPU will provide the current value of the internal variable “Region maximum X” associated to the currently selected region ID, for the currently selected texture ID.

On write requests:

The GPU will check if the received value is out of range for the internal variable “Region maximum X” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Region maximum X” associated to the currently selected region ID, for the currently selected texture ID. This will immediately cause next region drawing operations to apply the new right limit for the selected region.

Region Max Y port

On read requests:

The GPU will provide the current value of the internal variable “Region maximum Y” associated to the currently selected region ID, for the currently selected texture ID.

On write requests:

The GPU will check if the received value is out of range for the internal variable “Region maximum Y” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Region maximum Y” associated to the currently selected region ID, for the currently selected texture ID. This will immediately cause next region drawing operations to apply the new bottom limit for the selected region.

Region Hotspot X port

On read requests:

The GPU will provide the current value of the internal variable “Region hotspot X” associated to the currently selected region ID, for the currently selected texture ID.

On write requests:

The GPU will check if the received value is out of range for the internal variable “Region hotspot X” and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable “Region hotspot X” associated to the currently selected region ID, for the currently selected texture ID. This will immediately cause next region drawing operations to apply the new hotspot position for the selected region.

Region Hotspot Y port

On read requests:

The GPU will provide the current value of the internal variable “Region hotspot Y” associated to the currently selected region ID, for the currently selected texture ID.

On write requests:

The GPU will check if the received value is out of range for the internal variable "Region hotspot Y" and, if it is, it will be clamped to range limits. The resulting value will then overwrite the internal variable "Region hotspot Y" associated to the currently selected region ID, for the currently selected texture ID. This will immediately cause next region drawing operations to apply the new hotspot position for the selected region.

10 Execution of commands

GPU commands are drawing operations that can be requested by the CPU, by sending a value to the GPU's "Command" port. This section describes the GPU behavior when receiving such a request.

There are 5 different commands the GPU can perform. This table shows their numerical values, as well as the graphic effects that are used for each command.

Command name	Numeric value	Scaling	Rotation	Multiply color
Clear Screen	10h	no	no	no
Draw Region	11h	no	no	yes
Draw Region Zoomed	12h	yes	no	yes
Draw Region Rotated	13h	no	yes	yes
Draw Region Rotozoomed	14h	yes	yes	yes

10.1 Common processing

The general operation described here is common for the execution of all commands.

As a first step, the GPU will check if the requested write value corresponds to one of the valid commands listed above. If it does not, the request will be ignored and its processing will be stopped.

Following this, the GPU will check whether the current remaining pixels are enough to perform the requested command, as described in section 7 (GPU performance).

Then, if the command was not aborted due to insufficient remaining pixels, the GPU will draw on the drawing buffer. This part is command-specific and covered in the next subsections. However, for all commands, the color blending process will be done as described in section 6 for the current value of the "Active blending mode" variable.

10.2 Clear Screen command

The GPU will draw over every pixel in the drawing buffer draw using the color currently stored in the "Clear color" internal variable. Aside from blending, no effects are applied.

10.3 Draw Region command

The GPU will draw on the drawing buffer a rectangular part of the currently selected texture. That rectangle is delimited by the region configuration variables corresponding to the currently selected region ID, for that texture.

The region is placed so that the region's pixel marked as hotspot is drawn at the coordinates marked by the current GPU drawing position within the drawing buffer.

When drawing the region, the GPU will apply color multiplication using the current GPU color stored in the "Multiply color" variable. The order of color operations is this:

1) Perform color multiply between the region's pixels and the multiply color.
2) Perform blending of the color resulting from (1) over the drawing buffer contents.

This command will ignore the current values of drawing scale and drawing angle variables. The region is always drawn with no spatial transforms.

10.4 Draw Region Zoomed command

Processing will be the same as in command "Draw Region", but the drawn region will also be scaled. The scaling factors along X and Y will be the current values of "Drawing scale X" and "Drawing scale Y" variables, respectively. Negative scale factors along X or Y result in a mirror effect in that dimension.

Scaling will be done such that, for any scale factors, the region's hotspot keeps being drawn at GPU drawing position within the drawing buffer.

10.5 Draw Region Rotated command

Processing will be the same as in command "Draw Region", but the drawn region will also be rotated. The rotation angle will be the current value of "Drawing angle" variable. It is interpreted in radians, and positive angles will rotate the region clockwise.

Rotation will be done using the drawn region's hotspot as the center. That way the region's hotspot keeps being drawn at GPU drawing position within the drawing buffer.

10.6 Draw Region Rotozoomed command

Processing will be the same as in command "Draw Region", but the drawn region will also be scaled and rotated. These effects are applied in the same way described for the commands "Draw Region Zoomed" and "Draw Region Rotated".

Scaling and rotation effects will be combined ensuring that the region's hotspot keeps being drawn at GPU drawing position within the drawing buffer. Also, the combination of both effects must always perform scaling relative to the texture's X and Y axes and not

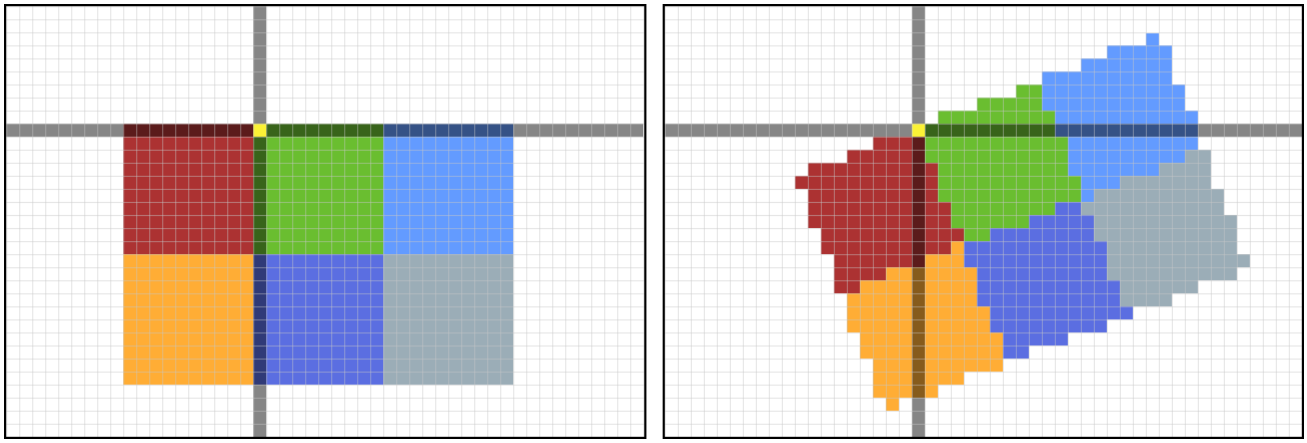
relative to screen coordinates. In other words: the order of transformations must be scaling first and then rotating.

10.7 Notes on 2D transforms

Region placement for large scale factors

When drawing with large scales, a single texture pixel may cover several screen pixels. In these cases we must take into account that the reference for positioning a texture pixel is not the pixel's center, but its top-left corner instead. This means that if we scale a single pixel with factors X = 640.0 and Y = 360.0, and we draw it at position (0, 0) it will exactly cover the whole screen.

As an example, we'll draw the region seen in the image (3 x 2 pixels) with a large scale. This region's hotspot is placed on its green pixel. If we use the drawing position given by the darkened crossing lines, the results on screen will be the following:



When the region is drawn without rotation (left image), the green pixel's top-left overlaps with the drawing position. If we also enable rotation (right image) that same screen pixel will be used as the rotation center, so the green pixel's original top-left corner will keep being drawn at the drawing position.

Interpolation methods

The reference interpolation method in Vircon32 GPU is nearest neighbour. However, implementations are free to choose other interpolation methods for image scaling and rotation. These algorithms, however, are required to preserve the original colors.

This means that color-smoothing methods such as linear or cubic interpolations would not be compliant with Vircon32. But pixel art algorithms like Scale2X or RotSprite would be acceptable, since they are designed to preserve pixel colors.

10.8 Command execution time

This specification makes no mention of the time taken to actually execute any of the drawing commands. Instead, the Vircon32 system abstracts this away and treats all drawing operations as if they were always instantly finished. The consequence of this is that the CPU could request more commands before the previous ones are finished.

The reason for this abstraction is that the Vircon32 system is designed to avoid having to synchronize any operation between components. In many cases this can be a valid approximation: modern GPUs are much faster than Vircon32. However, some real-world drawing operations might still take several CPU cycles to finish. This makes it necessary for implementations to have some kind of strategy to manage drawing operations.

There can be several types of strategies. On some graphic systems such as OpenGL, tasks may be performed in the background and not be a problem for the main program. For hardware implementations, it might be enough to simply implement a command queue that is managed in parallel. On the other hand, a software implementation may just plan the whole console operation around this, and postpone the next cycles until the last requested operation finishes.

Still, for graphical correctness, any implementation must ensure that:

1) No drawing operations accepted by the GPU are left unperformed.
2) Drawing operations are performed preserving their request order.
3) Every drawing operation is completed before the current frame ends.

11 Generation of video output

Video output in this GPU is governed by console's global timing, and is therefore based on frames. At a rate of 60Hz, every time the new frame signal is received, the GPU will trigger the generation and transmission of a new video frame.

The GPU will generate a video frame by collecting all needed color information for every pixel in the drawing buffer at that moment. The GPU will then transfer that information to the screen via the video output signal. If the signal format/protocol needs it, timing or sequencing information will be added to form a valid video frame transmission.

Implementations are free to choose the communication format and physical connectors for audio and video signals. They may be sent separately or, as is common in most displays nowadays, use a joint connector that sends both of them to the same display.

Note that the contents of the drawing buffer are not cleared or modified when a new frame begins or as part of the video output process. The drawing buffer's content is persistent and will only be automatically cleared on a reset or power on event.

12 Responses to control signals

As with all components in the console, whenever a control signal is triggered the GPU will receive it and produce a response to process that event. For each of the control signals, the GPU will respond by performing the following actions:

Reset signal:

- If any drawing operation was still in progress, they are aborted.
- All GPU internal variables are set to their initial values. This includes configuration variables for all regions within all textures.
- Any additional effects associated to those changes in internal variables are immediately applied, as described in the control port write behaviors.
- All pixels in the drawing buffer are set to black (R = 0, G = 0, B = 0).

Frame signal:

- If any drawing operation was still in progress, they are aborted.
- The GPU generates output for a new video frame, as described in section 11.
- The "Remaining pixels" internal variable is reset to its initial value.

Cycle signal:

- The GPU does not need to react to this signal, unless specific implementation details require it.

In addition to reacting to control signals, the GPU will also need to perform the following processing in response to console-level events:

When a new cartridge is connected:

- The GPU will locate the images contained in the BIOS and, if present, the cartridge.
- The GPU will perform the connection process to described in section 3 to assign a texture slot to each image, and gain access to their pixel information.
- The upper limit for variable "Selected texture" will be adjusted to the last used texture slot ID.

(End of part 4)