OpenGX

David Guillen Fandos
(david@davidgf.net)

June 18th, 2012

1 Introduction

On summer 2009 I started working on a PSP game project as a result of the console being hacked on the previous months. The idea of Toy Wars was conceived and the working started. My idea was to build a multiplatform game because that gave some advantages over console specific games. I chose OpenGL and SDL as the main libraries because they were available on the platform thanks to the scene contributions. This way I could create a computer game which was compatible with PSP.

After some time it became a decent game (for a homebrew) and I got tired of the PSP scene. I wanted to expand my horizons on console development and searched for a new target to focus on. I had been working on other consoles previously (in 2007-2008), Dreamcast and GB/GBA. Those projects didn't lead to any results (apart from two demos) but gave me some great experiences because their corresponding scenes were long dead, thus requiring extra effort to get into them.

So I found my brother's old Game Cube lying in the wardrobe's bottom. Immediately I had the need to start working on the scene although it was an old console and probably the scene would be dead. But surprisingly for me I found that the scene was quite alive. The secret was (I found out later) that the GC's successor, the Wii, was a cheap copy with some simple additions, so the Wii scene was a natural successor of GC's.

I started the port, which I thought it would be pretty straightforward, but after some time I realized that it would need more time to successfully port

Toy Wars. The main problem was the lack of "standard" libraries such as SDL and OpenGL. The GC and Wii scene was and old one and nobody cared about using multiplatform libraries which I guess it's common on console development. The SDL port for Wii existed and was quite good, but no port existed for the GC. Hopefully I found that there was a guy working on a GC port. I mailed him and he was kindly enough to send me his sources which, I have to say, worked like a charm.

OpenGX was the remaining piece of the puzzle. I needed an OpenGL renderer and the only project which existed by the time was a complete non-working mess. So instead of changing the render code I wrote an OpenGL wrapper so it could be used with other projects. That is the way OpenGX was born.

2 Aim and scope

In this document I want to explain how GX works and how OpenGX uses it to emulate an OpenGL pipeline. GX is messy and a little difficult to understand, so I expect that this contribution helps to lighten some of the darkest areas.

It isn't the aim of the document to be exhaustive about the GX pipeline so, for a fully understanding of it, refer to the official GX documentation [1] and the free libOGC GX implementation documentation [2]

Also the document won't cover all the OpenGX functionality but it's core and most interesting aspects (the ones which I've found more instructive to talk about). For the complete reference it's a good idea to look at the source code.

3 Introduction to GX's pipeline

The GX pipeline can be divided in two big parts: the geometry processing and the pixel processing. The first one takes care of the operations in the "vertex domain", which involves transformation of the vertices, projection and clipping. The pixel processing is responsible for pixel rasterization, which includes color, texture and lighting calculations. To be exact lighting is partly calculated by the geometry unit.

Figure 1 shows a sketch of the pipeline stages.

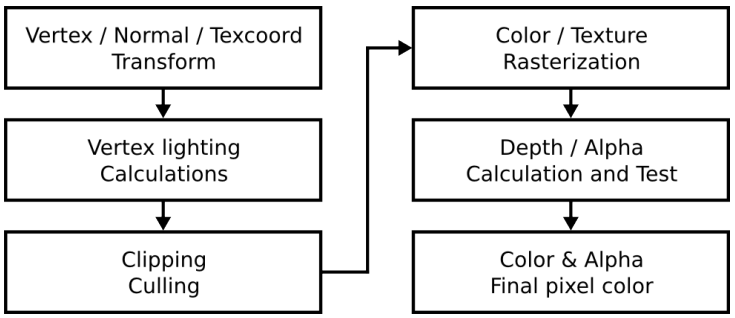


Figure 1: Sketch of the GX pipeline. Left stages care about geometry processing while the right ones carry pixel calculations

3.1 Vertex calculations

The vertex calculation follows a classic scheme. We have vertices, normals and texture coordinates which can be transformed using some matrices.

The model-view matrix transforms the vertices in model-world space to view space. The projection matrix then transforms the vertices in view space to projection space (2D space, the screen). We also have a normal matrix, for transforming the normals and a texture transform matrix for transforming texture coordinates within its space.

The main differences in the process is how the matrices are represented. First of all OpenGL uses column-major matrices while GX uses row-major matrices. This is OpenGL stores columns contiguously in the memory while GX stores rows contiguously in the memory. Also the matrices have different size. The projection matrix is 4x4 but the modelview matrix is a 3x4 matrix (as the last row is never used for three component vector transformations). The normal matrix is a 3x4 matrix which only uses the 3x3 submatrix (ignoring the last column) because normals can only be rotated, not translated. The texture transformation matrix can be either a 2x4 matrix or a 3x4 matrix, depending on the usage.

The hardware is capable of storing up to ten model-view and normal matrices, one projection matrix and ten texture matrices. The matrices must be loaded and flushed to the GPU pipeline before they can be used, so it's a good idea to have some slots to load matrices which never change or are used frequently. The hardware allows to switch between matrix slots very easily (and fast).

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{pmatrix}$$
(a) Projection matrix

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$
(b) Model-view matrix

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$
(c) Normal matrix

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \end{pmatrix} \qquad \begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \end{pmatrix}$$
(d) Texture coordinates matrices

Figure 2: Matrices involved in vertex transformations. Greyed out columns represent elements of the matrix which are not used by the calculations but must be present in memory

3.2 Pixel calculations

After the vertex transform, projection and clipping it all comes to pixel work. The hardware calculates the resulting color of every pixel for each piece of geometry sent to the GPU. The pixel is tested against the depth component to determine its visibility and then may be rendered on the screen.

The process of determining the pixel color is very complex because the GX allows the programmer to control the fixed pipeline in a similar way pixel or fragment shaders do. The final pixel color is calculated using a custom combination of the rasterized color (from vertex and lighting), the texture color and constant colors of our own choose. This combination occurs right before fogging and blending.

The operation is calculated in a *Texture Environment* unit (TEV), which is a combinational circuit which has colors as entries and a color output. The unit calculates the output color using up to four input colors as can be seen in Figure 3. The programmer can choose which operation (OP) wants to perform, which *scale* factor wants to apply to the color and an optional additive *bias*. The functions GX_SetTevColorOp and GX_SetTevAlphaOp are used to setup all the named parameters.

The TEV unit can be reused multiple times for the same screen pixel, so the operation that can be computed can be very complex. To achieve this unit re-usage the output is connected to four registers and the input is connected to those for registers as well as other inputs. This allows the programmer to use the output of previous calculations as the input of another

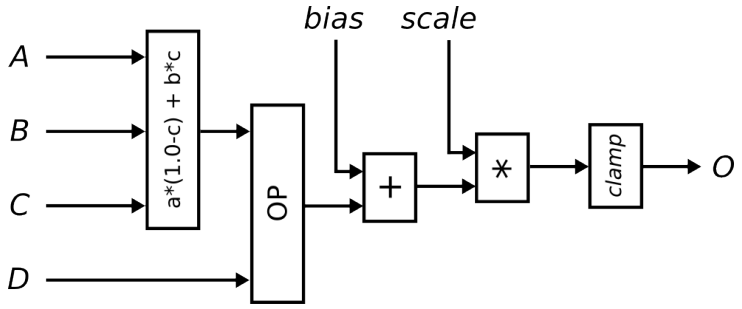


Figure 3: Diagram of the Texture environment unit, which calculates a color using up to four input colors

calculation. The function GX_SetNumTevStages sets the number of stages.

The unit can be used up to sixteen times for the same pixel. The less cycles needed for a pixel calculation the fastest the rendering will be. An example is shown in Figure 4

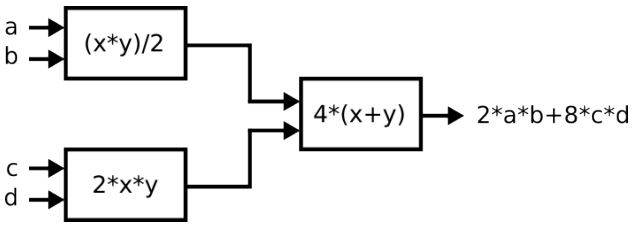
So, what inputs can be used in a TEV stage (not counting the four auxiliary registers)? We can use the color from a texture (in fact multiple textures), the color from the rasterizer (which is calculated using lighting equations) and some constant registers.

The four registers are named *reg0*, *reg1*, *reg2* and *prev*. When concatenating TEV stages it's common to use the *prev* register. Also the *prev* register is used to store the final pixel color and alpha, so the last TEV stage should have *prev* register as its output. The functions GX_SetTevColorIn and GX_SetTevAlphaIn set the inputs for each stage.

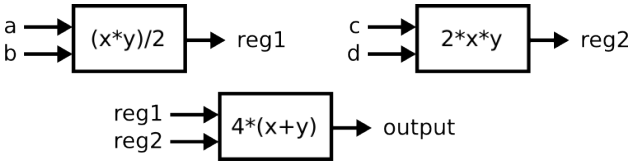
The TEX input represents the rasterized texture color. The rasterization is done using the texture coordinates and the color is calculated in the texture unit (which applies decompression and/or filtering if required) and passed to the stage as an input. There's only one texture input so in theory only one texture can be rasterized. But the hardware allows to rasterize multiple textures at each TEV stage. So if we want to combine two textures in the TEV we *must* use two stages. The first stage would retrieve the first texture color and the second stage would retrieve the other and combine them.

The RAS input is the color outputted from the rasterizer. This color is calculated using lighting equations, so it takes in account the materials, lights and vertex colors. We'll discuss extensively as it's one of the most important inputs.

The KONST input provides a constant color which can be specified by



(a) Example of a color operation using three TEVs



(b) Implementation of the operation using one TEV thrice (using auxiliary registers and thus taking three cycles)

Figure 4: Example of a color operation which is calculated using a TEV unit and using the four auxiliary registers for intermediate data. We are using three *TEV stages*

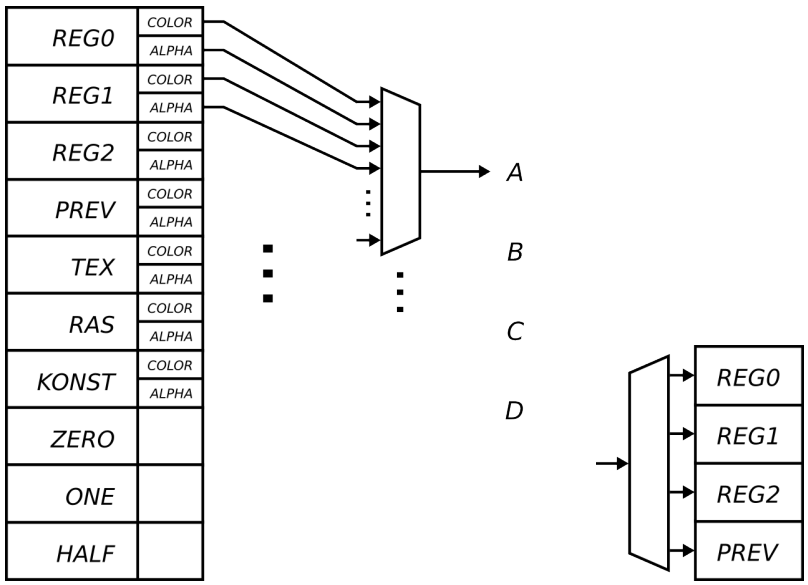
the programmer for each stage. There are hardwired colors but also four registers for the programmer to write. Those registers can be written using GX_SetTevKColor. For each stage the constant color to use can be selected with the function GX_SetTevKColorSel.

In addition to the constant colors provided by the input KONT, the inputs ZERO, ONE and HALF can be selected as inputs for a TEV stage. This inputs, as their names suggest, are hardwired values to zero, one and 0.5f. They're useful (specially zero) for performing simple operations which only require two or three operands.

3.2.1 Texture rasterization

Each TEV stage can only lookup one texture pixel, as can be seen in Figure 5. So we can control which texture is going to be rasterized in each TEV stage. The function GX_SetTevOrder allows to specify which texture is going to be rasterized and presented at the TEX input. Also we must specify the texture coordinates that will be used for the rasterization (as vertices can have up to eight texture coordinates).

As an example we are going to consider the rasterization of a lightmapped



(a) Inputs available for the four TEV inputs

(b) Possible output registers for a TEV

Figure 5: Diagram showing the possible inputs and outputs for a TEV stage. It's possible to choose between color or alpha channel in some inputs

scene. In a lightmapped scene we have two textures (the colored one and the lightmap) and two pairs of texture coordinates. The operation we are going to perform is to modulate the colors, that is, multiplying the colors components so that the original color gets darkened or lightened depending on the gray amount. A DirectX 6 example can be found at [3] with the explanation.

First of all we have to setup two TEV stages and indicate that two sets of texture coordinates are being used.

```
GX_SetNumTevStages(2);
GX_SetNumTexGens(2);
```

Now we're going to setup the first TEV stage, which will select the TEX as input D and zero for the other inputs. The operation doesn't matter at all.

```
GX_SetTevColorIn(GX_TEVSTAGE0, GX_CC_ZERO, GX_CC_ZERO, GX_CC_ZERO,
                 GX_CC_TEXC);
```

```
GX_SetTevAlphaIn(GX_TEVSTAGE0,GX_CA_ZERO,GX_CA_ZERO,GX_CA_ZERO,
                    GX_CA_TEXA);

GX_SetTevColorOp (GX_TEVSTAGE0, GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1,
                    GX_TRUE,GX_TEVPREV);
GX_SetTevAlphaOp (GX_TEVSTAGE0, GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1,
                    GX_TRUE,GX_TEVPREV);
```

We are storing the result in the *prev* register. For the next stage we will use inputs B and C and ignore the other (zeroed). B will be the previous value (texture 1 color) and C will be texture 2 color.

```
GX_SetTevColorIn(GX_TEVSTAGE1, GX_CC_ZERO, GX_CC_CPREV, GX_CC_TEXC,
                    GX_CC_ZERO);
GX_SetTevAlphaIn(GX_TEVSTAGE0, GX_CA_ZERO, GX_CA_APREV, GX_CA_TEXA,
                    GX_CA_ZERO);

GX_SetTevColorOp (GX_TEVSTAGE1, GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1,
                    GX_TRUE, GX_TEVPREV);
GX_SetTevAlphaOp (GX_TEVSTAGE1, GX_TEV_ADD, GX_TB_ZERO, GX_CS_SCALE_1,
                    GX_TRUE, GX_TEVPREV);
```

Now we have to select the texture coordinates used in each stage. We have to select one texture coordinate slot (of the eight available) and the transform matrix. It's possible to use a hardwired identity matrix, so there's no need to upload one.

```
GX_SetTexCoordGen(GX_TEXCOORD0, GX_TG_MTX2x4, GX_TG_TEX0, GX_IDENTITY);
GX_SetTexCoordGen(GX_TEXCOORD1, GX_TG_MTX2x4, GX_TG_TEX1, GX_IDENTITY);
```

Now we have to setup the TEX input so the rasterized colors from the previous slots are passed to the TEVs. The operation selects the texture coordinates slots and the texture itself (the texture load code is not shown as it's very large and doesn't affect this explanation).

```
GX_SetTevOrder(GX_TEVSTAGE0, GX_TEXCOORD0, GX_TEXMAP0, GX_COLORNULL);
GX_SetTevOrder(GX_TEVSTAGE1, GX_TEXCOORD1, GX_TEXMAP1, GX_COLORNULL);
```

The diagram representing the previous steps is shown in Figure 6

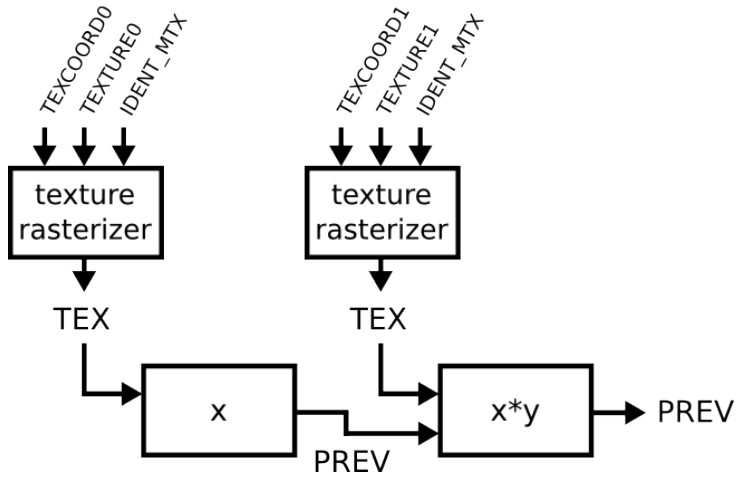


Figure 6: Example showing a simple lightmap renderer using two textures and no lighting. The first stage just passes the TEX input to the output while the texture colors are multiplied in the second stage

3.2.2 Color rasterization

The rasterized color is presented to a TEV stage through the RAS input (Figure 5). It's possible two choose between to color channels named COLOR0A0 and COLOR1A1. To specify which channel to present to the TEV stage we use GX_SetTevOrder.

When lighting is disabled the rasterized color can be either the content of a register or the rasterized vertex color. The register is called Material Color and can be specified with the function GX_SetChanMatColor. In order to choose between them we use GX_SetChanCtrl passing GX_SRC_REG or GX_SRC_VTX. Figure 7 shows a circuit describing the color calculation process.

In order to be able to use two color channels it's mandatory to supply two vertex colors. If only one color is supplied the color is passed to channel number zero, even if the color is supplied as channel 1 color.

Lighting adds complexity to the calculations. In fact the unlit model is a particular case of the lit model with brightness at its maximum value. The final color its calculated as follows (everything are colors and the dot product is a component product):

$$Color = Material \cdot LightFunc$$

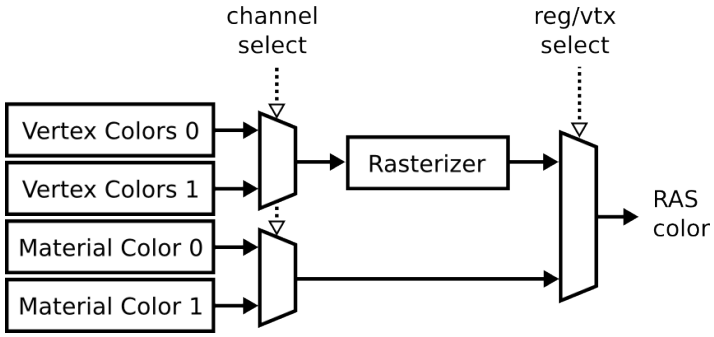


Figure 7: Calculation of the RAS color present at the input of a TEV stage when no lighting is used. It's necessary to choose the color channel from the two available and the color source, which can be register or vertex rasterized

Where $Material$ is the unlit color (the one which comes from Material register or rasterized vertex color) and LightFunc the amount of lighting. With lighting disabled $LightFunc$ is white color, so that $Color$ equals $Material$.

The $LightFunc$ is calculated as follows:

$$LightFunc = Ambient +$$

$$\sum_{i=0}^{7} LightEnabled_i \cdot Attenuation_i \cdot DiffuseAttenuation_i \cdot Color_i$$

The $Ambient$ term is a color which can come from the vertex rasterized color or a register (like $Material$). The register value can be specified using GX_SetChanAmbColor. Each light can be enabled or disabled for a given channel, so if the light is disabled it doesn't contribute to the lighting term.

The attenuation term is evaluated from zero to one indication how the light contributes to the vertex illumination. It depends with the vertex-light distance and can be controlled with the function family GX_InitLightAttn*, which allow to specify the attenuation function in a very flexible way (directional, positional, spotlight, etc.). It's important to specify the light position and direction using GX_InitLightDir and GX_InitLightPos.

The diffuse attenuation is a term calculated taking into account the angle between the light and the surface normal, so it lights the polygon depending on its angle. It's controlled with GX_SetChanCtrl, so it's channel specific. This means that it's not possible to mix spot lights and positional or directional lights in the same channel.

The *Color* term is the color of the light and can be different for every light. It can be specified with GX_InitLightColor.

Figure 8 represents the lighting calculations.

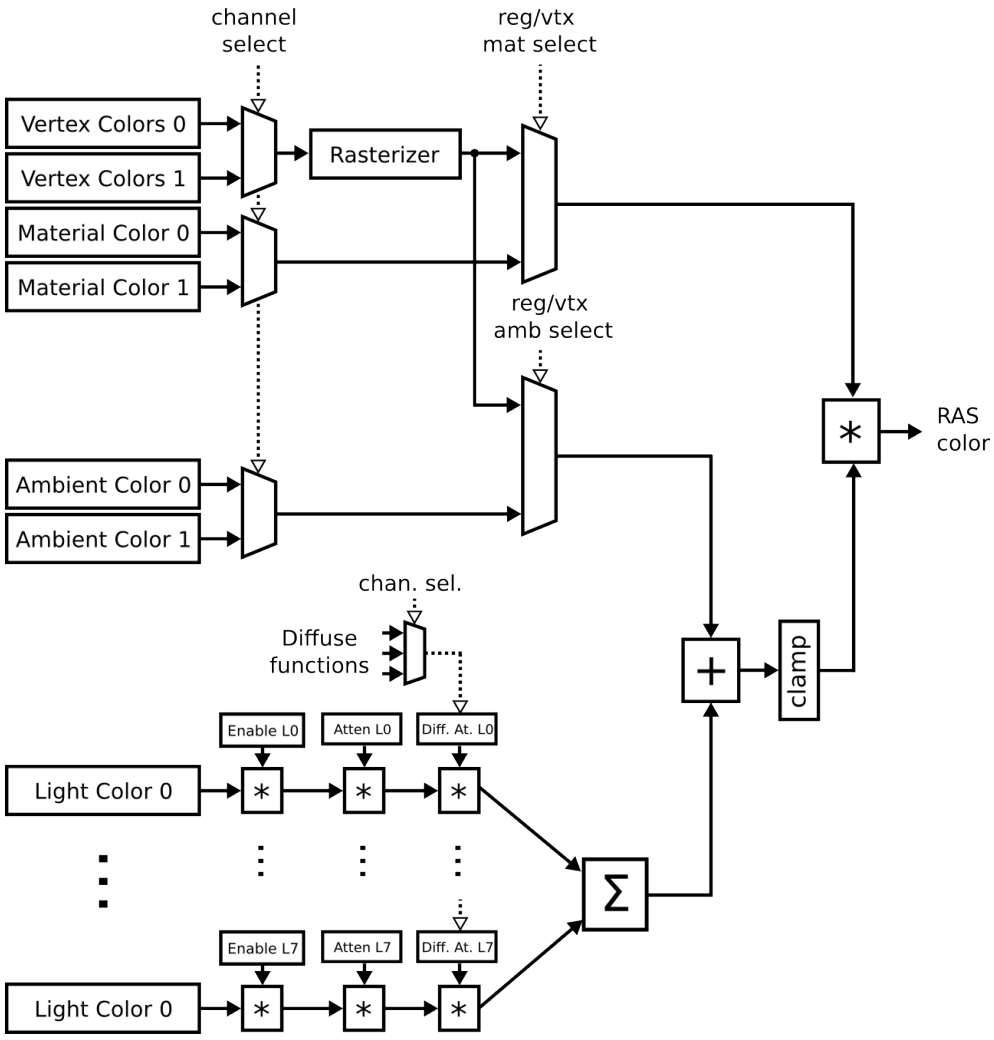


Figure 8: Schematic representing lighting calculation process.

When lighting is disabled the lower subsystem is disabled so that the output color is calculated from the material color. Note also that the diffuse term depends on the channel number instead of the lighting object.

4 OpenGX implementation

GX and OpenGL are both finite state machines. But, since there are many diferences between them, I haven't established a relationship between their states. Instead I've decided to keep the GL state in memory and update GX state accordingly when needed.

The OpenGX state is stored in memory and includes: transform matrices, state bits, textures, geometry data (pointers or data itself), lighting data and any other status data (current color, texture, etc.). This state is used to update the GPU state when needed. In order to avoid unnecessary transfers the state also includes some dirty bits, which indicate the need to update certain information. When GL state changes some bits may become dirty and the rendering process cleans those bits by transferring and updating the state.

The immediate mode is implemented using data pointers. There are pointers for each data set: vertices, indices, normals, colors and texture coordinates. The glVertex* calls are implemented by copying the data to a temporary buffer and using glDrawArrays function to draw the data (at glEnd). Each pointer has an integer value indicating the stride for the data in the array. The glInterleavedArrays function is emulated using gl*Pointer and glDrawArrays functions with the adequate pointers and strides.

The rendering process consists in a setup stage and a loop for the data transfer. This data transfer consists in writing vertex data into GX's FIFO. The data is written interleaving its components (when no indices are used), so using interleaved buffers should be faster than separated buffers rendering (because of the cache). The data present in the arrays is used selectively depending on the context: if texturing is disabled the texture coordinates are ignored and not sent to the GPU (which is faster). Also if lighting is disabled normals are not sent too. A little optimization is performed for vertex colors: if the vertex color remains constant no color data is sent through the FIFO and the GPU is instructed to use the constant color register.

4.1 Rendering cases

The implemented render setup depends in two main factors: lighting and texturing.

4.1.1 Unlit and untextured render

When the lights and textures are disabled the color of the pixels is provided by the vertex colors. In this case the setup will use one TEV stage which takes the RAS input to the output (PREV register).

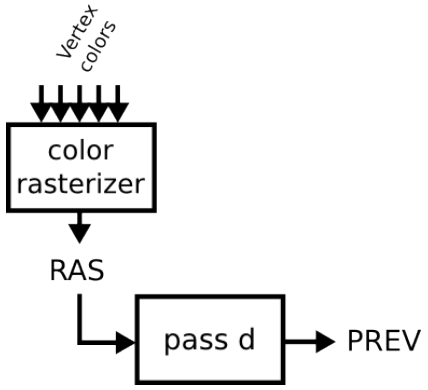


Figure 9: Unlit and untextured case.

4.1.2 Unlit and textured render

The final pixel color in this case is the product between the rasterized color and the texture color. So only one TEV stage is used to multiply the RAS input with the TEX input.

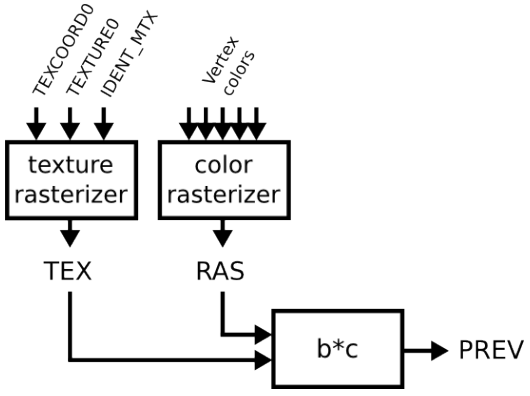


Figure 10: Unlit and textured case.

4.1.3 Lit and untextured render

The light color is calculated using two channel colors. We're going to use channel 0 to emulate ambient lights and channel 1 to emulate diffuse lights. This decision is explained later. So we'll use two TEV stages, the first for rasterizing the channel zero and the second for rasterizing the channel one and combine them.

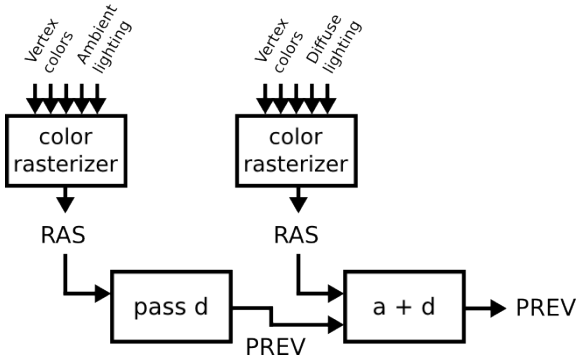


Figure 11: Lit and untextured case.

4.1.4 Lit and textured render

Adding texturing to the lit case is just like adding texturing to the unlit case: just multiply the result color by the texture color. This is done by adding an extra stage at the end of the pipeline to multiply the color.

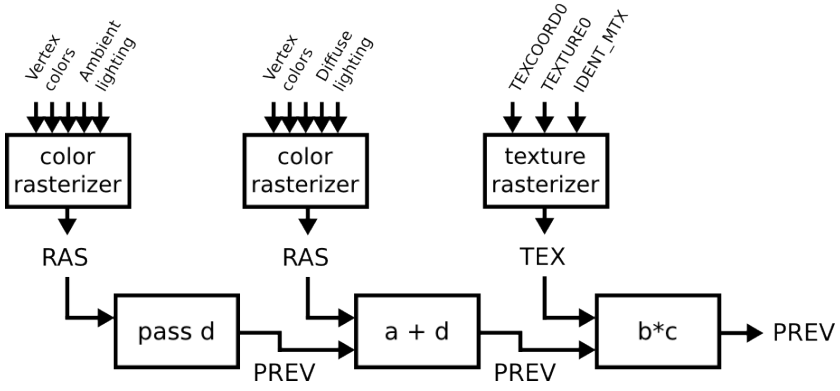


Figure 12: Unlit and untextured case.

As the time of writting I've just realized that it's possible to use only two stages by multiplying the texture color to each color channel before adding them. Anyway this would be an optimization.

4.2 Lighting management

OpenGL has support for up to eight lights. Each light has a direction, position, cutoff, etc. depending how the light behaves (it's a point, a light cone, a directional light...). But all the lights have three color components: ambient, diffuse and specular. The ambient component only depends on the distance of the light and the vertex. The diffuse component depends on the distance (just like the ambient) but also depends on the light angle formed by the light and the surface normal. Finally specular lights depend on the distance as well and the angle formed by the light, the vertex and the viewer.

It's important to remember that the diffuse attenuation factor of a light is on a per channel basis instead by light. So we have to use one channel for ambient lights and another one for diffuse lights. We exclude specular lighting because there aren't enough channels to compute the specular component. For those reasons we are limiting the number of active lights to four. We'll use the four first GX lights to emulate ambient lighting and the other four lights to emulate diffuse lighting.

The material stuff is done in CPU. We multiply the light color by the material color and use the resulting color as light color. The global ambient light, which is an additive light color without attenuation, is implemented using the ambient color present in the lighting ecuation, although it could be just added after as a constant color in a TEV stage (but this way we save a TEV stage).

4.3 Texture management

Only 2D textures are implemented. The formats accepted by the implementation are mainly RGB and RGBA but also other special formats like Luminance and compressed textures. The internal formats accepted by the GPU are shown in Figure 13.

All formats without transparency are converted to RGB565, which saves memory while having a good color quality. Using RGBA8 would be a waste of memory without any quality gain. For transparent textures we use RGBA8 format, except for luminance-alpha textures, which are converted to IA8

Format	Color depth	Alpha support	Compressed
GX_TF_RGB565	5/6 bits	No	No
GX_TF_RGBA8	8 bits	8 bits	No
GX_TF_CMPR	N/A	1 bit	Yes
GX_TF_IA8	8 bits	8 bits	No

Figure 13: Texture formats used by OpenGX

format. Luminance-alpha is a format with one transparency channel and one color channel. All the color components (RGB) have the same value, so it's used in black and white textures with transparency such as text, shadows, etc.

Compressed textures are only used for non-alpha textures because compressed textures only have one bit for transparency (masking), which is not enough. Therefore if the user requests compression on transparent textures the request is ignored. The compression is performed using a DXT compressor by Jonathan Dummer [4] with some specific modifications and big-endian fixing.

Texture levels (used for mipmapping) are implemented in an efficient way. If the user loads one texture level the implementation reserves memory for that level only. If the user loads more levels the memory is resized to accommodate all the levels. GX allows the programmer to specify a level range for a specific texture, so we can have one level textures and multiple level textures.

References

[1] Nintendo *Revolution Graphics Library (GX) Version 1.00* 2006.

[2] LibOGC Team? *GX Subsystem Reference*

[3] Jason Mitchell, Ian Bullard, Michael Tatro *Multitexturing in DirectX 6*

[4] Jonathan Dummer *DXT compressor* (http://nothings.org/)