Special Effects With Current Graphics Hardware

João L. D. Comba 1

Rui Bastos 2

Abstract: New developments in graphics hardware bring performance improvements and new or more general features over previous generations of hardware. Given the already powerful performance attainable with current hardware, more emphasis has recently been put in introducing new and more general features. The combination of powerful performance with new features allows for advanced special effects, which improve the realism of the generated images in real-time applications. This tutorial will summarize features recently introduced in graphics hardware, more specifically in the GeForce3 board developed by NVIDIA. The presentation will cover three main topics: high-order surfaces, and vertex and pixel programming, presented along the description of special effects that can be obtained with each technology.

Resumo: Novos desenvolvimentos em hardware gráfico trazem melhorias em performance e novas ou mais genéricas características com relação a gerações de hardware anteriores. Dada a performance poderosa do hardware atual, mais ênfase tem sido posta em introduzir novas e mais genéricas características. A combinação da poderosa performance com as novas características permite efeitos especiais avançados, os quais melhoram o realismo das imagens geradas por aplicações de tempo real. Este tutorial resumirá características e aplicações recentemente introduzidas em hardware gráfico, mais especificamente da GeForce3 desenvolvida pela NVIDIA. O texto cobrirá três tópicos principais: superfícies de alta ordem, e programação de vértices e pixels, cada qual acompanhado de descrições de efeitos especiais que podem ser obtidos com cada tecnologia.

1 Instituto de Informática, UFRGS, Caixa Postal 15064, Porto Alegre, RS 91501-970, Brasil
{comba@inf.ufrgs.br}
2 NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, Estados Unidos
{rbastos@nvidia.com}

1 Introduction

Graphics hardware is under an incredible development process, causing great impact in computer graphics and related areas. In a recent past, displaying real-time imagery was only possible with powerful and expensive supercomputers. This scenario started to change with the improvement of graphics boards in cheaper platforms (e.g. PC systems), allowing more people to experience better graphics effects. Over the years, the advances in the semiconductor industry allowed more and more features to be incorporated into graphics boards, turning them into very complex and amazing pieces of hardware. As a result, graphics boards can now render in real-time millions of polygons shaded with very complex effects.

In particular, this development has experienced a great deal of changes in the last couple of years. One important change is the innovative programmable aspect, which marks a change from incrementally adding dedicated hardware to perform every new feature designed, into an orthogonal archictecture that can be programmed to perform a larger subset of the existent funcionality. Because it is programmable, the power of this new functionality is directly related to the expression power of the language used to program the hardware.

Specific parts in the graphics pipeline are more suitable for being exposed as programmable than others. Vertex processing, for instance, is a good candidate. In the beginning of the pipeline, a graphics processing unit (GPU) receives triangle data composed of global information shared by its vertices (e.g. triangle color), and specific information about each vertex (e.g. position, color, normal, etc.). This information is passed one vertex at a time to the GPU, to perform vertex operations such as transformation and lighting (T&L) and texture coordinate computations. Programmable hardware is exposed here by means of a vertex program, that describes the operations to be performed over vertex data, allowing the creation of effects that include vertex skinning, morphing, or even other lighting calculations. These aspects will be described when we discuss vertex programming.

Another part of the graphics pipeline made programmable is the one responsible for generating the final color for a pixel. Here, a pixel program describes how this can be done by sequencing a series of components that can mathematically operate on pixel attributes. This solution enables hardware-accelerated procedural shaders [7], which are frequently used in software [8]. By breaking the complexity of pixels into different components, pixel programs became possible with a special hardware that can perform a sequence of texture lookups where the texture coordinates can derive from a previous texture lookup, and that can combine the final texture results with other pixel attributes produced in the graphics pipeline.

In this paper, we review features and special effects of current graphics hardware, using as test case the GeForce3 board produced by NVIDIA [1]. The presentation follows the order of the graphics pipeline, starting with a discussion of a new set of evaluators to create high-order surfaces. In sequence, we show how vertex programs can be used to change the geometric shape and illumination of vertices. Finally, the ultimate goal of the pipeline is to generate pixel colors, and the discussion on pixel programming describes how pixel data can

be combined using texture shaders and register combiners to produce pixel colors. The special effects that can be created are discussed alongside the presentation of the features.

2 High-Order Surfaces

Points, lines and triangles are the basic primitives used to describe geometric shapes in graphics boards. Because they are representations of linear functions, a mechanism to describe high-order functions became necessary. In OpenGL, for instance, evaluators were proposed as a way to triangulate polynomial or rational curves (or surfaces) described using Bézier basis functions. The polynomials are parametrically evaluated at various values for the *u* parameters of a curve, or at *u* and *v* parameters of a tensor-product surface.

The difficulty in optimizing evaluators as they were proposed is one reason why evaluators were not popular. In turn, the fact that they were not popular, made developers not bother to update them to reflect the changes happening at per-vertex attributes, such as multiple texture coordinates, secondary color, fog coordinates, etc. As a result, evaluators became obsolete and the need for a revised and improved evaluator mechanism arose. The general evaluator was proposed to allow more general high-order surfaces, with the following main differences with the previous mechanism:

- **Triangular patches**: the previous evaluator relied only in rectangular patches. Although a triangular patch can be converted into three rectangle patches, having direct support for a single triangular patch is more efficient and convenient.
- **Generalized number of subdivisions in all sides of a patch**: to prevent cracks, the previous evaluator only allowed the number of rows and columns to be specified, forcing opposite sides of a patch to have the same number of subdivisions. This is relaxed in the new proposal, allowing all sides to specify the number of subdivisions used.
- **Fractional number of subdivisions:** The number of subdivisions can be described in floating point, which allows smooth level of detail transitions for a surface.
- **Consistent evaluation on shared edges**: the previous evaluator presented numerical problems that could create cracks in the surface if shared edges had opposite orientations. The use of rules for the evaluation of the functions in general evaluators guarantee that shared edges between two patches match up. It requires that shared vertices positions be equal, and that normals do not change in a discontinuous fashion.
- **Variable order for the polynomial basis**: allow the specification of the order of the polynomial basis, which reflects the number of linear interpolations used to construct the surface. Currently up to quintic (5^{th} order) is supported.

- **Support for all vertex attributes**: including vertex position, normal, colors, and texture coordinates for all texture units. Fully integrated with vertex programs.

The general evaluators bring back surface representations into graphics special effects. The ability to generate more general surfaces that can be adaptively represented allow better multi-resolution rendering, free of crack artifacts. Effects like cloth simulation using high-order surfaces and vertex programs can be found at NVIDIA developer's site [2].

3 Vertex Programming

Simple primitives as triangles are used to represent shapes, and their information is sent through the graphics pipeline to a graphics-processing unit (GPU). Triangle data is either common to all its vertices, or specific, such as vertex positions (x, y, z, w), normals (x, y, z), colors (red, green, blue, alpha, specular, fog, etc) or texture coordinates (s, q, r, t) for each texture unit. In the GPU, vertex transformations and lighting (T&L) computations are performed, as well as texture coordinate generation. These operations require lower level computations that are expensive to compute, such as matrix multiplications, exponentials and square roots. Since they are typically performed in large quantities, vertex processing can account for a big computational cost in the graphics pipeline.

Using dedicated T&L hardware to perform these tasks was proposed in the first GeForce hardware, allowing the extra CPU cycles to be used elsewhere in complex operations like physics, simulation, and AI. Because many possibilities of vertex operations are possible, adding more dedicated hardware would help even further. However, designing dedicated hardware to each idea that comes to mind is expensive and almost impractical. The idea of **vertex programming** is to solve this problem in a much more elegant way, with the design of an orthogonal architecture that allows the hardware to be programmed to create many high-level vertex operations.

A **vertex program** consists of a program, written in assembly-like language, that describes operations on vertex data, including all performed by the hardware T&L unit and many more. This code is compiled and passed into a special-purpose hardware unit that executes the operations, and writes the results into the next stage in the graphics pipeline. In this section, we describe in detail the vertex programming computational model, consisting of a graphics-oriented assembly language for description of programs, and the different types of memory it has available. We conclude the section with examples of small vertex programs and discuss special effects that can be created with them.

3.1 Vertex Programs

A vertex program is composed of a sequence of instructions that perform operations on a set of parameters composed of input vertex data, temporary data, and parameters data, which generates a set of per-vertex output parameters. A special-purpose GPU receives as

input a vertex program with customized operations over vertex data, and is responsible for executing the instructions and generating the results to the next unit in the graphics pipeline.

An application program can create and maintain in memory a pool of vertex programs. At any specific time, the application can choose to load one vertex program into the unit that executes vertex programs. Because vertex programs work in a per-vertex basis, the only operations possible are the ones that affect individual vertex data, and operations that require the vertices to be assembled as a primitive are performed elsewhere, like clipping and culling. Besides, the information calculated in a vertex program is not maintained for the next execution, neither a new vertex can be created or deleted.

On GeForce3, a dedicated hardware T&L unit is provided in addition to the one that executes vertex programs, mainly to support compatibility with previous hardware. However, only one is active at a single time, and all functionality of the dedicated unit can be performed using vertex programs with similar performance results.

A schematic representation of the vertex program computational model is described in Figure 1. Vertex programs access four types of memory: vertex-attribute registers (read-only), program parameters (read-only), temporary registers (read-write) and per-vertex output registers (write-only), defined in more details in the sections to follow.

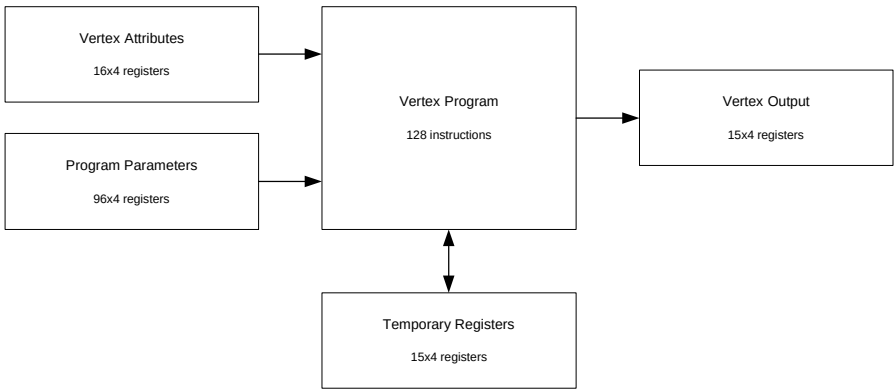

Figure 1 – GeForce3 Vertex Program Computational Model

3.1.1 Vertex Attribute Registers

Vertex input data is stored in vertex attribute registers and can be read by vertex programs. They contain vertex data such as position, color, texture-coordinates, etc. On GeForce3, there are 16 vector-attribute registers, named v[0] through v[15], with the default values listed in table 1.

Table 1. Vertex Attribute Registers

i	*Register*	*Description*	*Format*	*i*	*Register*	*Description*	*Format*
0	v[OPOS]	vertex position	x,y,z,w	6	-		
1	v[WGHT]	vertex weight	w,0,0,1	7	-		
2	v[NRML]	vertex normal	x,y,z,1	8	v[TEX0]	texture coord 0	s,t,r,q
3	v[COL0]	primary color	r,g,b,a	…	…	…	…
4	v[COL1]	secondary color	r,g,b,1	15	v[TEX7]	texture coord 7	s,t,r,q
5	v[FOGC]	fog coordinate	f,0,0,1				

3.1.2 Program Parameters Registers

The application can write to constant memory registers that can be read by vertex programs. They are read-only registers, and provide a great deal of flexibility for the creation of special effects, as the user decides what to store in these registers. Usually, values stored in these registers include matrix transformations, lighting parameters, pre-computed values, frame-dependent values, etc. For convenience, important matrices in OpenGL[4] as the modelview or perspective matrices can be tracked and made automatically available in parameter registers. There are 96 vector parameter registers in GeForce3, named c[0] through c[95].

3.1.3 Temporary Registers

The temporary registers are twelve floating-point vector registers used to hold temporary results during vertex program execution, referenced by R0 through R11. These registers are initialized to (0,0,0,0), and are private to each vertex program invocation. They are read-write registers that can be read and written during vertex program execution. Also, writes to these registers can be masked on a per-component basis. There is also a write-only address register, A0, used to perform indirect addressing in parameter register memory. Only A0.x can be written using the MOV instruction. Indirect addressing is helpful, for instance, to index several matrices during vertex skinning computation.

Table 2. Vertex Output Registers

Register	***Description***	***Format***
o[HPOS]	transformed vertex's homogeneous clip space position	(x,y,z,w)
o[COL0]	transformed vertex's front-facing primary color	(r,g,b,a)
o[COL1]	transformed vertex's front-facing secondary color	(r,g,b,a)
o[BFC0]	transformed vertex's back-facing primary color	(r,g,b,a)
o[BFC1]	transformed vertex's back-facing secondary color	(r,g,b,a)
o[FOGC]	transformed vertex's fog coordinate	(f,*,*,*)
o[TEX0..7]	transformed vertex's texture coordinates for texture units 0..7	(s,t,r,q)

3.1.4 Vertex Output Registers

The vertex result registers are 15 4-component floating-point vector registers used to write the results of a vertex program. Each register value is initialized to (0,0,0,1) at the

invocation of each vertex program. Similarly to temporary registers, writes to the vertex result registers can also be masked on a per-component basis. Primitive coloring may operate in two-sided color mode, with the selection between back-facing and front-facing colors depending on which primitive the vertex belongs to.

3.1.5 Instruction Set

The instruction set used for vertex programs is composed of 17 instructions that can perform up to four operations simultaneously, over the 4 components of the vector registers. The instructions work with either vector or scalar operands, the latter corresponding to a component of a vector operand. The format of a vertex program instruction is defined in Figure 2.

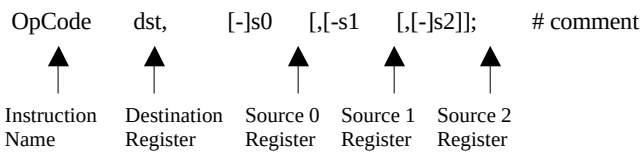

Figure 2 – Vertex Program Instruction Format

The instructions and their respective input and output parameters are summarized in table 3. As a convention, operands that start with 'v' are 4-component float vectors, while operands that start with a 's' are scalar floats.

Table 3. Vertex Program Instruction Set

Instruction	***Description***
ARL s_dest, s_src;	Loads floor of s_src into one coordinate of the address register
MOV v_dest, v_src;	moves source vector into destination vector
MUL v_dest, v_src0, v_src1;	component-wise multiply on two vectors
ADD v_dest, v_src0, v_src1;	component-wise add on two vectors
MAD v_dest, v_src0, v_src1, v_src2	adds third vector to the product of first two
RCP v_dest, s_src	inverts source scalar, replicates result in vector
RSQ v_dest, s_src	computes inverse square root of absolute value of source scalar, replicates result in vector
DP3 v_dest, v_src0, v_src1	three-component dot-product of two vectors
DP4 v_dest, v_src0, v_src1	Four-component dot-product of two vectors
DST v_dest, s_src0, s_src1	Computes a distance attenuation vector (1, d, d^2, 1/d). Assumes s_src0 contains d^2, and s_src1 contains 1/d.
MIN v_dest, v_src0, v_src1	component-wise minimum on two vectors
MAX v_dest, v_src0, v_src1	component-wise maximum on two vectors
SLT v_dest, v_src0, v_src1	Component-wise assignment of 1.0 or 0.0 (1 if value of first source is less than second source, 0 otherwise)
SGE v_dest, v_src0, v_src1	Component-wise assignment of 1.0 or 0.0 (1 if value of first

	source is greater or equal than second source, 0 otherwise)
EXP v_dest, s_src	Exponential base 2, z contains 2^{s_src}, x and y contain intermediate results, w set to 1
LOG v_dest, s_src	Logarithm base 2, z contains approximation of $\log_2\|s\_src\|$, x and y contain intermediate results, w set to 1
LIT v_dest, v_src	Lighting computation. Assumes input x contains (N.L), y contains (N.H), and w contains specular power. Resulting x and w has 1.0, y and z has diffuse and specular coefficients

The source registers can receive a mapping before an operation occurs. Instructions use either scalar source values or swizzled source values. Either type of source value is negated when the optional sign "-" is used. Scalar source register values select one of the source register's four components based on the component used (the characters "x", "y", "z", and "w" match the x, y, z, and w components, respectively). Similarly, destination register can mask which components are written. In table 4 we have examples that help to illustrate the modifications discussed.

Table 4. Examples of mapping over source registers

```
MOV R1, -R2;     # R1.x=-R2.x R1.y=-R2.y R1.z=-R2.z R1.w=-R2.w
MOV R1, R2.yzwx; # R1.x=R2.y R1.y=R2.z R1.z=R2.w R1.w=R2.x
MOV R1, R2.w;    # R1.x=R2.w R1.y=R2.w R1.z=R2.w R1.w=R2.w
MOV R1.xw, -R2;  # R1.x=-R2.x R1.w=-R2.w
```

3.1.6 Examples of Vertex Programs

To illustrate the use of the vertex program instruction set, tables 5, 6, and 7 present simple examples that perform typical graphics operations.

Table 5. Computation of a 3-component cross product

```
MUL R2, R0.zxyw, R1.yzxw;      # R2=(z0*y1, x0*z1, y0*x1, w0*w1)
MAD R2, R0.yzxw, R1.zxyw, -R2; # R2=(y0*z1, z0*x1, x0*y1, w0*w1) – R2
```

Table 6. Transformation of a vertex position by a matrix, with a homogeneous divide

```
# c[20-23]   transformation matrix
DP4 R5.x, v[OPOS], C[20];   # position transformation
DP4 R5.y, v[OPOS], C[21];
DP4 R5.z, v[OPOS], C[22];
DP4 R5.w, v[OPOS], C[23];
RCP R11, R5.w;              # computing 1/w
MUL R5, R5, R11;            # homogeneous divide
```

Table 7. Computation of specular and diffuse effects with an eye-space normal

```
# c[0-3]  modelview projection composite matrix
# c[4-7]  modelview inverse transpose
```

```
# c[32]   eye-space light direction (LDIR)
# c[33]   eye-space half-angle vector (H)
# c[34]   pre-multiplied diffuse light and diffuse material
# c[35]   pre-multiplied ambient light and ambient material
# c[36]   specular color
# c[38].x specular power
DP4 o[HPOS].x, c[0], v[OPOS];     # output transformed position
DP4 o[HPOS].y, c[1], v[OPOS];
DP4 o[HPOS].z, c[2], v{OPOS];
DP4 o[HPOS].w,c[3], v[OPOS];
DP3 R0.x, c[4], v[NRML];          # R0 = N'= transformed normal
DP3 RO.y, c[5], v[NRML];
DP3 RO.z, c[6], v[NRML];
DP3 R1.x, c[32], R0;              # R1.x = LDIR * N'
DP3 R1.y, c[33], R0;              # R1.y = H * N'
MOV R1.w, c[38].x;                # R1.w = specular power
LIT R2, R1;                       # R2 = lighting computation
MAD R3, c[35].x, R2.y, c[35].y;   # R3 = diffuse + ambient
MAD o[COL0].xyz, c[36], R2.z, R3; # output color0 = R3 + specular
```

3.1.7 Special effects using Vertex Programs

There are many special effects that can be obtained using vertex programs, which correspond to different ways of creating custom transformation and lighting, as well as generating texture coordinates to index material properties textures (environmental attributes, reflectance, etc). Effects already described in the NVIDIA developers site include:

- **Skinning**: Skeletal animation of characters requires applying weights to vertices of the model, proportional to the impact of each bone. This technique is called skinning, and requires applying several transformation matrices to each vertex. Using vertex programs, up to 32 3x3 matrices can be loaded into the parameter registers.
- **Dynamic Surface Deformations**: The ability to change vertex positions allows vertex programs to apply deformations to the shape of an object. For instance, a ripple effect can be created by moving vertices over time along a defined function.
- **Procedural Modeling**: Instead of deformations, a procedural modeling can be used to describe the shape of an object, and use the vertex program to generate the shape. Special effects such as particle systems can be created this way.
- **Morphing**: Vertex weights can be used in vertex programs to morph a shape into another.
- **Distortion effects:** The creation of custom transformation matrices allows effects that cannot be created with the fixed pipeline**.** One example is fisheye lens effects.
- **Environmental Effects**: Fog effects can be produced by changing the fog coordinates of a vertex depending on its height or elevation.

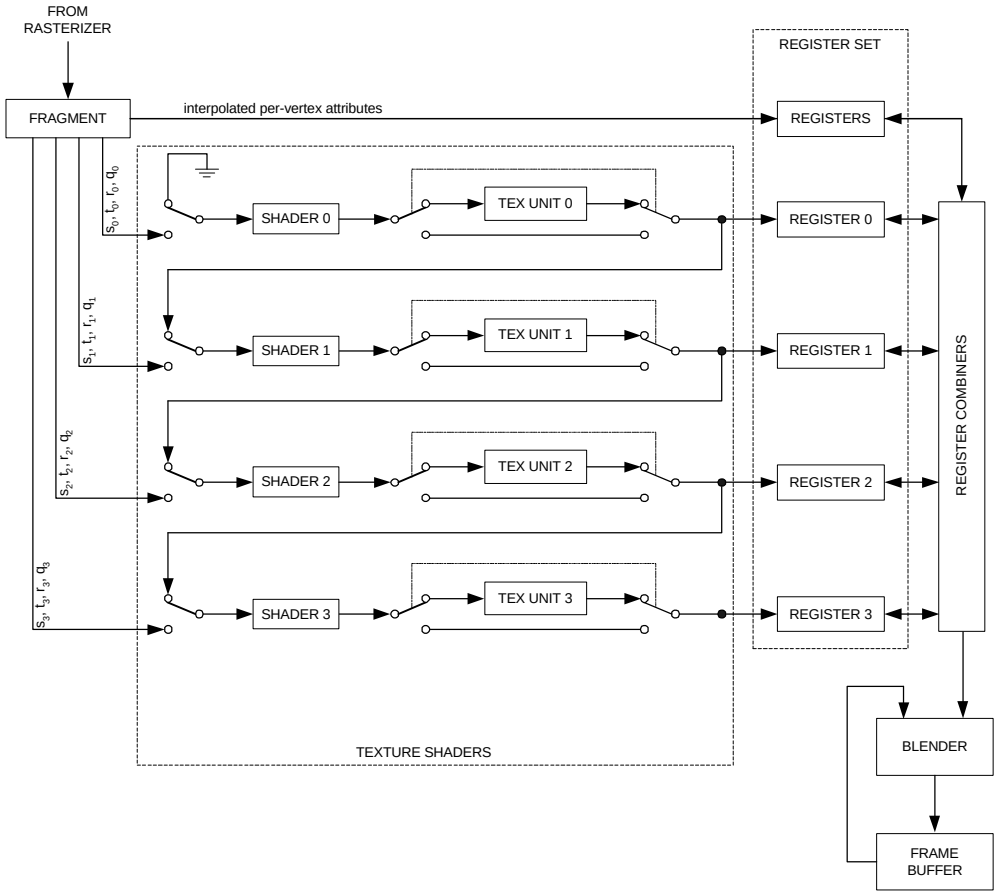

Figure 3 - GeForce3 Pixel Shaders. Fragments coming from the rasterizer go through a sequence of up to four texture shaders and then have the resulting texel colors combined with other interpolated attributes in the register combiners. Additionally, the fragment color can be blended with the corresponding pixel already in the frame buffer.

4 Pixel Programming

Most 3D-graphics applications require the computation of a shading function at each pixel on the screen. Even though vertex programming provides great flexibility in computing and interpolating low-frequency (per-vertex) data, per-pixel programming is necessary for dealing with high-frequency information, i.e. dealing with information that changes much faster than the vertex data can capture. The ultimate goal is to run an

independent program for each pixel on the screen, and several approaches have been taken to allow programmability at pixel level. In this section, we discuss the GeForce3 pixel-shading architecture, which enables real-time per-pixel shading with great flexibility. For convenience, we use OpenGL terminology where necessary [4][5].

The ability to control the shading at each pixel has evolved through generations of graphics hardware, together with the evolution of the graphics pipeline:

- **Single-texture**: The most standard graphics pipeline allows a single texture lookup for each pixel as the only controllable parameter. The texture can encode a 1D, 2D, or 3D function, depending on the dimensionality of the data. For this type of pipeline, pixel shading is the combination of a single texture lookup with other interpolated attributes, such as diffuse and specular colors, and fog factors.
- **Multi-texture**: As a natural evolution over a single texture per fragment, additional texture coordinates can be added to the rendered primitives, and a same number of color blenders added to the pipeline for computing the final pixel color as a blend of the texture results. For such a pipeline, pixel shading is the color blending of the available textures/functions with other interpolated attributes.
- **Multiple textures and register combiners**: As an alternative to the linear color blending of texture results, register combiners allow more flexible and more powerful operations in signed math at pixel level. The register combiners operate on 4-element vectors and allow the computation of component-wise products, sums of products, 3-element dot products, mux of products, and the manipulation of intermediate values, when evaluating an equation at pixel level. For this type of pipeline, pixel shading goes beyond the trivial color blending of the available textures/functions with other interpolated attributes.
- **Multiple texture stages, dependent textures, dot-product dependent textures, special-mode textures, and register combiners**: In addition to allowing more powerful combination of texture results, the graphics pipeline can also manipulate the inputs to the texture units. By connecting texture lookups sequentially, the texture coordinates for a given texture lookup can depend (with a few operations applied upon) on the result of a previous texture result. Special-mode textures can also use texture coordinates as fragment colors, replace the depth (z) value of a fragment with a texture-dependent value, and cull away fragments that do not satisfy given conditions. These additional pieces of functionality allow for extremely flexible and powerful computation at pixel level, as the available textures can encode different parts of the pixel-shading equation to be evaluated. Figure 3 has a high-level view of this pipeline, which allows the implementation of advanced algorithms at fragment/pixel level.

The GeForce3 pixel shaders follow the last model described above. The next sections discuss texture shaders and register combiners, which are the pieces of hardware functionality that enable per-pixel shading on GeForce3.

4.2 Texture Shaders

On GeForce3, a texture shader is a sequence of up to four texture stages (Figure 3). A texture stage performs operations on the incoming texture coordinates to produce either a new set of texture coordinates for the next stage or a final color for the texture stage. A shader program determines the operation mode of a texture stage and the implicit dependency on previous stages. Texture shaders can use 1D, 2D, 3D, cube-map, or shadow-map texture targets. The texture lookups performed in a texture shader use the state specified by the API for the corresponding texture unit.

There are 23 shader programs available on GeForce3 (Table 8), which can perform special-mode operations (where no texture lookup is done), perform a conventional texture lookup, or perform a dot-product texture lookup.

Table 8. Texture Shader Programs

1	**NOP**: Always produces a black (0,0,0,0) fragment
2	**PASS_THROUGH**: Converts texture coordinates (s,t,r,q) into clamped color components (r,g,b,a) in the [0,1] range
3	**CULL_FRAGMENT**: Culls the fragment, based on the incoming texture coordinates. Each texture coordinate (s,t,r,q) is tested with less-than-zero or greater-than-or-equal-to-zero. The fragment is culled, if any of the four comparisons fails
4	**DOT_PRODUCT_DEPTH_REPLACE**: Must be preceded by a DOT_PRODUCT program in the previous texture shader stage. Computes a dot product on the incoming texture coordinates and replaces the fragment's window-space depth (z) value with the first dot product divided by the second. The resulting fragment color is black (0,0,0,0).
5	**TEXTURE_1D** : 1D texture lookup using (s/q)
6	**TEXTURE_2D**: 2D texture lookup using (s/q,t/q)
7	**TEXTURE_RECTANGLE**: 2D texture lookup on a rectangular texture (non-power-of-two) using (s/q,t/q)
8	**TEXTURE_3D**: 3D texture lookup using (s/q,t/q,r/q)
9	**TEXTURE_CUBE_MAP:** Cube-map texture lookup using (s,t,r)
10	**OFFSET_TEXTURE_2D**: 2D texture lookup using a perturbed set of texture coordinates that depends on the incoming texture coordinates and the result of a previous texture shader stage. The result of a previous stage is multiplied by a 2x2 matrix and added to the incoming texture coordinates. The resulting texture coordinates are used to perform the 2D texture lookup
11	**OFFSET_TEXTURE_2D_SCALE:** Similar to the OFFSET_TEXTURE_2D shader program, but the resulting red, green, and blue color components are scaled by a configurable scale value and then clamped to the [0,1] range
12	**OFFSET_TEXTURE_RECTANGLE:** Similar to the OFFSET_TEXTURE_2D shader program, but the lookup is into a rectangular texture (non-power-of-two dimensions) with non-projective texture coordinates (s,t)
13	**OFFSET_TEXTURE_RECTANGLE_SCALE:** Similar to the

	OFFSET_TEXTURE_RECTANGLE shader program, but the resulting red, green, and blue color components are scaled by a configurable scale value and then clamped to the [0,1] range
14	**DEPENDENT_AR_TEXTURE_2D:** Converts the alpha and red color components of a previous stage result into texture coordinates (s,t) for a 2D non-projective texture lookup
15	**DEPENDENT_GB_TEXTURE_2D:** Converts the alpha and red color components of a previous stage result into texture coordinates (s,t) for a 2D non-projective texture lookup
16	**DOT_PRODUCT:** No texture lookup is performed with this shader program and the color result is undefined. It computes the dot product of the incoming texture coordinates (s,t,r) with some mapping of the components of a previous texture shader result. The component mapping depends on the type and signedness of the previous stage texture input
17	**DOT_PRODUCT_TEXTURE_2D:** When preceded by a DOT_PRODUCT in the previous texture shader stage, computes a second similar dot product and composes the two dot product results as texture coordinates (s,t) for looking up into a 2D non-projective texture.
18	**DOT_PRODUCT_TEXTURE_RECTANGLE:** Similar to the previous shader program, but the lookup is performed on a rectangular non-projective texture
19	**DOT_PRODUCT_TEXTURE_3D:** When preceded by two DOT_PRODUCT programs in the previous two texture stages, computes a third similar dot product and composes the three dot products into (s,t,r) texture coordinates to access a 3D non-projective texture
20	**DOT_PRODUCT_TEXTURE_CUBE_MAP:** When preceded by two DOT_PRODUCT programs in the previous two texture shader stages, computes a third similar dot product and composes the three dot product as texture coordinates (s,t,r) for looking up into a cube-map texture
21	**DOT_PRODUCT_REFLECT_CUBE_MAP:** When preceded by two DOT_PRODUCT programs in the previous two texture shader stages, computes a third similar dot product and composes the three dot product results into a normal vector (Nx,Ny,Nz). An eye vector (Ex,Ey,Ez) is composed from the q texture coordinate of the three stages.
22	**DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP:** Similar to DOT_PRODUCT_REFLECT_CUBE_MAP program, but the eye vector is a user-defined constant, instead of being composed by the q texture coordinate of the three stages
23	**DOT_PRODUCT_DIFFUSE_CUBE_MAP**: When used in place of the second dot product preceding DOT_PRODUCT_REFLECT_CUBE_MAP or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP stage, the normal vector forms the texture coordinates (s,t,r) to access a cube-map texture

4.3 Register Combiners

Similar to texture shaders, the register combiners are organized in a sequence of stages. Whereas the texture shaders operate on the texture coordinate inputs to the texture units, the register combiners operate on the output colors from the texture units and on other user-provided constants and interpolated attributes.

On GeForce3, the register combiners are organized in a sequence of up to 8 general combiners followed by a single final combiner. Figure 4 is a diagram of the GeForce3 register combiners. The following sections describe the input/output register set, the general combiners, and the final combiner.

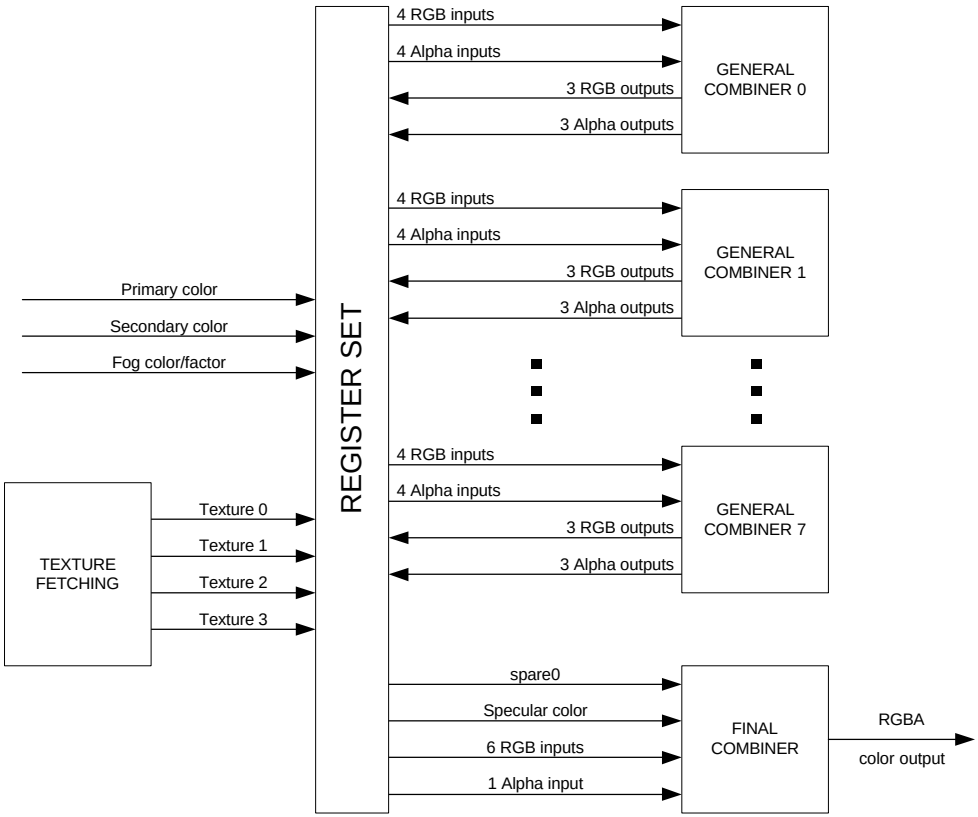

Figure 4 GeForce3 Register Combiners: Available registers from the register set are selected to go through a sequence of up to eight general combiners and then into the final combiner.

4.3.1 Input/Output Register Set

The register combiners operate upon a set of 4-element vector registers available for all the combiner stages. With respect to the combiners, some of the registers are read-only, whereas others are read-and-write access registers. Some of the read-and-write access registers are scratch registers, even though any of the write-access registers can be used, destructively, for temporary storage.

The read-only registers are for values that do not change for an entire primitive or for a set of primitives. Except for the constant zero, the other three read-only registers are configurable from the API level (written by the application):

- Fog color,
- Two RGBA color constants, and

- The value zero.

The read-and-write access registers are primitive or per-vertex attributes interpolated by the rasterizer:

- Primary (or diffuse) color,
- Secondary (or specular) color,
- Filtered texels from enabled texture units/texture stages (up to 4, on GeForce3), and
- Fog factor---Available only for the final combiner.

The scratch registers are read-and-write access registers intended for storage of general combiner results, as temporaries for subsequent combiner stages. Scratch registers are not initialized, except for the alpha component of spare0, which is equal to the alpha component of texture0, when that unit is enabled:

- Spare0 and
- Spare1.

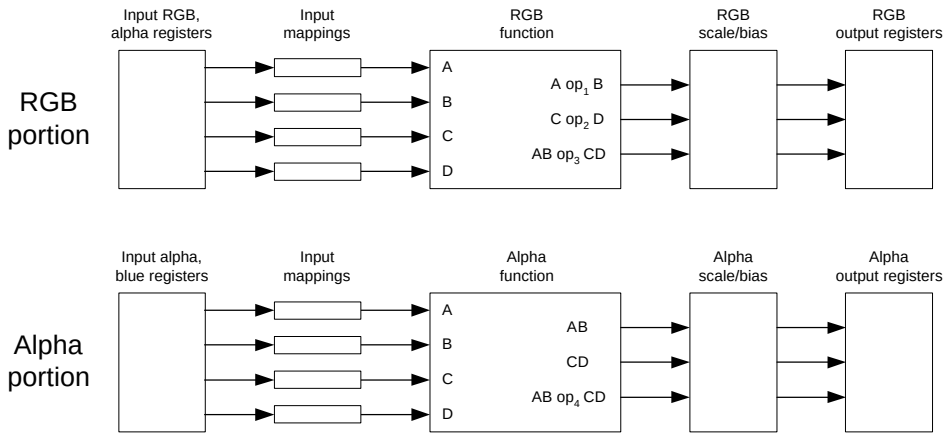

Figure 5 General Combiner: Split into RGB and alpha portions, the general combiners take four inputs (from the register set) for each portion, perform input mappings to the desired range, apply the selected function, scale and bias the results, and output the final result back into the register set.

4.3.2 General Combiners

The eight general combiners on GeForce3 operate identically. Each general combiner takes 4 vector (RGBA) inputs (A, B, C, and D) from the register set and computes, independently, 3 equations for the RGB portion and 3 equations for the alpha portion (see Figure 5). The 3 equations for the RGB and alpha portions are selected from the following set (6 equations selectable for RGB and 4 equations selectable for alpha):

1. Either (A*B) or (A dot B), // dot-product is not available for alpha
2. Either (C*D) or (C dot D), // dot-product is not available for alpha
3. Either (A*B+C*D) or (A*B mux C*D),

where '*' is component-wise multiplication, 'dot' is dot product, and 'mux' is a selection between the left or right terms. Dot-product operations are not available for alpha combiners, because 1-element dot products make no sense.

Although inputs to and outputs from a general combiner are 4-element vectors/registers, the actual computation is split into two distinct operations: A 3-element operation on the RGB components and a 1-element operation on the alpha component. The contents for RGB combiner inputs can come from the RGB portion of any readable register or from the alpha component of a register replicated into R, G, and B. The contents for alpha-combiner inputs can come from the alpha portion of any readable register or from the blue component of the RGB portion.

Computations in the registers combiners are performed in signed [-1,1] range. Each input to the register combiners is mapped into the [-1,1] range using one of the 8 component-wise input mappings described in table 9.

Table 9 – Register Combiner Computation

	Mapping	*Result*
1	UNSIGNED_IDENTITY	new_input = max(0,input)
2	UNSIGNED_INVERT	new_input = 1–min(max(0,input),1)
3	EXPAND_NORMAL	new_input = 2*max(0,input)–1
4	EXPAND_NEGATE	new_input = –2*max(0,input)+1
5	HALF_BIAS_NORMAL	new_input = max(0, input)–0.5
6	HALF_BIAS_NEGATE	new_input = –max(0,input)+0.5
7	SIGNED_IDENTITY	new_input = input
8	SIGNED_NEGATE	new_input = –input

Before the result of a general combiner is stored in its destination register, the resulting value is scaled, biased, and clamped. The scaling can be any of the following: 0.5, 1.0, 2.0, or 4.0. The bias can be either –0.5 or 0.0. The clamping is to the [-1,1] range. The scale and bias are shared by the three components of the RGB portion, but need not be the same for the alpha component. Note that the result of a dot product operation is a scalar, which is replicated on the three (RGB) components of the output register.

Each general combiner can output 6 different values (3 equations for each portion---RGB and A) to be used by the next general combiner or the final combiner. Registers written by a general combiner are available to the next combiner in the sequence, and registers not written preserve their values.

4.3.3 Final Combiner

The final combiner takes 6 inputs (A, B, C, D, E, and F) from the register set to compute the final fragment color (Figure 6). Both the input to and the output from the final combiner are unsigned values (negative input values are clamped to zero).

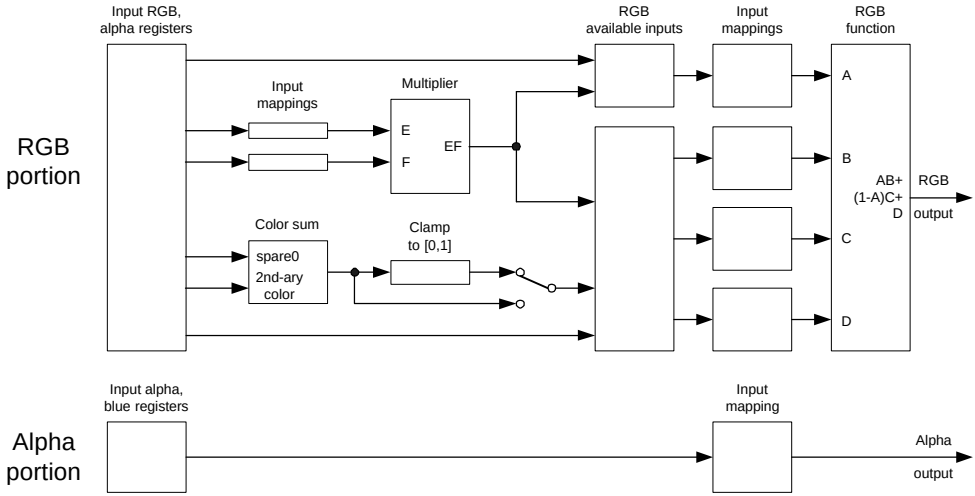

Figure 6 Final Combiner: Split into RGB and alpha portions, the final combiner takes inputs from the register set and performs operations on the data to produce the final fragment color.

Each input to the final combiner can have its value inverted and can come from either the RGB portion or the alpha component (replicated into R, G, and B) of the corresponding input register. There are 2 additional pseudo-registers available for the final combiner: The E*F product is available as a pseudo-register for the A, B, C, and D inputs; and the RGB addition of the spare0 and the secondary-color registers is made available as a pseudo-register for the B, C, and D inputs.

The RGB portion of the final fragment color is computed by evaluating A*B+(1-A)*C+D. The alpha portion is just a copy of the alpha channel (possibly inverted) from any of the registers in the register set.

4.4 Frame-Buffer Blending

All the functionality discussed up to this point can produce shaded pixels with a single pass over the geometry. When the complexity of the pixel equation does not fit on a single pass over the 4-stage texture shaders and the 9-stage register combiners, the user can apply multi-pass rendering.

Multi-pass rendering allows the blending of an incoming pixel color with the color of the corresponding pixel already stored in the frame buffer. This approach allows for the linear combination of multiple independent pixel shaders as described in the previous sections, at the expense of multiple passes over the corresponding geometry and the required memory bandwidth to read from and write to frame buffer memory. The greatest advantage

of multi-pass rendering is the unlimited number of linear operations that can be performed at each frame buffer pixel. The disadvantages are the limited numerical precision in the frame buffer and the required multiple passes over geometry to produce the final result. The limited precision can cause artifacts on the final image, due to error generation and propagation, whereas the multiple passes over geometry impact performance, which may imply non-real-time functioning on multi-pass rendering applications.

4.5 Examples of Pixel Shaders

To illustrate the use of texture shaders and register combiners, tables 10 to 13 present examples that perform typical graphics operations. Even though the `nvparse` [5] syntax is not described in this document, we use it in the examples below. It should be trivial to map `nvparse` operations into their respective architectural descriptions in the previous sections. The `nvparse` utility takes a string containing the shader program and converts it into the corresponding OpenGL calls to setup the required state.

Table 10. Texture shader for cube mapping

```
texture_cube_map();
```

Table 11. Texture shader for per-pixel diffuse and specular lighting

```
// texture unit 0:  RGB8 normal map; (s0,t0)=triangle surface coords
// texture unit 1:  no texture map;  (s1,t1,r1) is L vector
// texture unit 2:  no texture map;  (s2,t2,r2) is H vector
// texture unit 3:  RGB8 texture w/ (NdotL,NdotH) for (s,t) in [0,1]
texture_2d();                              // tex0 = Normal map
dot_product_2d_1of2( expanded( tex0 ) ); // (s1,t1,r1) is L vector
dot_product_2d_2of2( expanded( tex0 ) ); // (s2,t2,r2) is H vector
// result: (s,t) is (N dot L, N dot H) to lookup into 2D tex3
```

Table 12. Texture shader for per-pixel reflective bump mapping

```
// texture unit 0:  unsigned RGB normal map; (s0,t0)=surface coords
// texture unit 1:  no texture map;   (s1,t1,r1) is L vector
// texture unit 2:  no texture map;   (s2,t2,r2) is H vector
// texture unit 3:  RGB environment cubemap (reflection map)
// texture coordinates (q0,q1,q2) compose the per-pixel eye vector
texture_2d();
dot_product_reflect_cube_map_eye_from_qs_1of3( expanded(tex0) );
dot_product_reflect_cube_map_eye_from_qs_2of3( expanded(tex0) );
dot_product_reflect_cube_map_eye_from_qs_3of3( expanded(tex0) );
// result: (s,t,r) = reflected eye vector to lookup into cubemap tex3
```

Table 13. Final Combiner for (primary_color*detail_tex + secondary_color)

```
  // col0 is the primary color
  // col1 is the secondary color
  // tex0 is the detail texture
```

```
{   // General combiner definition (combiner 0)
    rgb {   // RGB portion (optional)
          spare0 = col0 * tex0;
    }
}

// Final combiner
out.rgb = spare0 + col1;
out.a = unsigned_invert(zero);  // alpha is 1
```

4.6 Special Effects using Pixel Shaders

There are many special effects that can be obtained using pixel shaders. The following are some examples described in the NVIDIA developer's website [2]:

- **Depth Sprites**: Allow rendering 3D objects as a texture mapping operation on a flat surface. The per-pixel depth of the supporting surface (triangle/quadrilateral) is replaced by the filtered depth coming from a texture lookup on a height map (displacement from the flat surface). Textures shaders are used for replacing per-pixel depth values, and the register combiners are used for computing per-pixel lighting on the resulting geometry.
- **Per-Pixel Specular and Diffuse Bump Mapping with Phong Lighting**: Combines the use of a vertex program for per-pixel lighting setup, texture shaders for specular and diffuse terms computation, and register combiners to perform the final blending of components.
- **Per-Pixel Lighting with Shadow Mapping**: Combining the use of a vertex program, texture shaders, and register combiners to compute per-pixel specular-and-diffuse lighting with true shadows computed with hardware shadow mapping.
- **Bump Reflection and Refraction**: Using texture shaders and register combiners to render cube-map reflections and refractions off a bump-mapped surface.
- **Order-Independent Transparency**: The rendering of overlapping transparent surfaces requires following sorted order (back-to-front). The use of hardware shadow mapping together with texture shaders and alpha kill allows the correct rendering of transparent surfaces with sorting done at the pixel level (no burden on the CPU).

5 Conclusions

The rapid change in graphics hardware is posing a great challenge to users that want to benefit from state-of-the-art graphics. In this text, we summarized a partial list of what we consider to be the most interesting and promising features of current graphics hardware. By no means it represents a complete coverage of the subject, and we encourage the reader to

access the literature on the topic, mainly comprised of documentation only available at the NVIDIA developers site [2]. We hope the material presented here will motivate people to start using the new features and developing their own special effects.

Among all topics discussed, the programmability is the key aspect of the GeForce3. Vertex programs and pixel shaders (texture shaders and register combiners) provide ways to take control of the graphics pipeline at various stages. Vertex shaders allow modifications in the shape and the lighting aspects of vertices. In pixel shaders, the per-pixel computational power is split on texture coordinates manipulation, regular texture mapping, and powerful register combiners. All these pieces together enable real-time per-pixel shading on affordable graphics hardware. Impressive results of the GPU programmability are illustrated by the Stanford Programmable Shading research project [7], which makes available a high-level language for programming low-level GPUs. In the foreseeable future, we can expect to get even closer to a fully hardware-accelerated RenderMan-like programmable shading [8].

As with any other programming language, the better way to learn how to program this new hardware is by doing it yourself. A good starting point is the NVIDIA developer's site [2], which has many presentations, white papers, and code examples for several effects. For API information, we recommend the DirectX 8 (DX8) [6] and OpenGL NVIDIA extensions [4] documents. For application development and implementation, we recommend the nvparse utility [5], which provides a library to enable vertex and pixel shading with a simple and easy-to-use language for vertex programs, texture shaders, and register combiners.

References

[1] NVIDIA Corporation. Web: http://www.nvidia.com,

[2] Developer Section, NVIDIA Corporation. Web: http://www.nvidia.com/developer.

[3] GeForce3 hardware description, NVIDIA Corporation. Web: Search for '*GeForce3*' in http://www.nvidia.com/products.

[4] OpenGL NVIDIA extensions: Search for '*NVIDIA OpenGL specs*' in http://www.nvidia.com/developer.

[5] NVparse utility: Search for '*parse*' in http://www.nvidia.com/developer.

[6] DirectX 8 specification. Web: http://www.microsoft.com/directx

[7] Kekoa Proudfoot, William R. Mark, Pat Hanrahan, Svetoslav Tzvetkov. A Real-Time Procedural Shading System for Programmable Graphics Hardware. Proceedings ACM SIGGRAPH 2001.

[8] Anthony A. Apodaca and Larry Gritz. Advanced RenderMan. Morgan Kaufmann, 2000.