

Shader manual

Credits

Based on Quake III Arena Shader Manual By Paul Jaquays and Brian Hook

(with additional material by John Carmack, Christian Antkow, Kevin Cloud & Adrian Carmack)

With warsow/quake specific material by Victor Luchitz

(with additional material by Quake 2 Evolved team)

Introduction

What is a shader

Shaders are short text scripts that define the properties of a surface as it appears and functions in a game world (or compatible editing tool). By convention, the documents that contain these scripts usually has the same name as the texture set which contains the textures being modified (e.g; base, hell, castle, etc.). Several specific script documents have also been created to handle special cases, like liquids, sky and special effects.

For warsow, shader scripts are located in /base/sw/scripts.

A shader file consists of a series of surface attribute and rendering instructions formatted within braces ("{" and "}"). Below you can see a simple example of syntax and format for a single process, including the Q3MAP keywords or "Surface Parameters", which follow the first bracket and a single bracketed "stage":

```
textures/liquids/lava
{
    deformVertexes wave sin 0 3 0 0.1
    tessSize 64
    {
        map textures/common/lava.tga
    }
}
```

Shader name & file conventions

The first line is the shader name. Shader names can be up to 63 characters long. The names are often a mirror of a pathname to a .tga file without the extension or basedir (/baseqf in our case), but they do not need to be. Shaders that are only going to be referenced by the game code, not modeling tools, often are just a single world, like "projectionShadow" or "viewBlood". Shaders that are used on characters or other polygon models need to mirror a .tga or a .jpg file, which allows the modelers to build with normal textures, then have the special effects show up when the model is loaded into the game. Shaders that are placed on surfaces in the map editor commonly mirror a .tga file, but the "qer_editorimage" shader parameter can force the editor to use an arbitrary image for display.

Shader pathnames have a case sensitivity issue – on windows, they aren't case sensitive, but on unix they are. Try to always use lowercase for filenames, and always use forward slashes "/" for directory separators.

Shader types

The keywords that affect shaders are divided into two classes. The first class of keywords are global parameters. Some global parameters ("surfaceparms". And all "q3map_" keywords) are processed by Q3MAP, and change physical attributes of the surface that uses the shader. These attributes can affect the player. To see changes in these parameters one must re-bsp the map.

The remaining global keywords, and all Stage Specific Keywords are processed by the renderer. They are appearance changes only and have no effect on game play or game mechanics. Changes to any of these attributes will take effect as soon as the game goes to another level or vid_restarts (type command vid_restart in the game console).

Shader keywords are not case sensitive.

IMPORTANT NOTE: some of the shader commands may be order dependent, so it's good practice to place all global shader commands (keywords defined in this section) at the very beginning of the shader and to place shader stages at the end (see various examples).

Key concepts

Ideally, a designer or artist who is manipulating textures with shader files has a basic understanding of wave forms and knows about mixing colored light (high school physics sort of stuff). If not, there are some concepts you need to have a grasp on to make shaders work for you.

Surface effects vs. content effects vs. deformation effects

Shaders not only modify the visible aspect of textures on a geometry brush, curve, patch or mesh model, but they can also have an effect on both the content, "shape", and apparent movement of those things. A surface effect does nothing to modify the shape or content of the brush. Surface effects include glows, transparencies and rgb (red, green, blue) value changes. Content shaders affect the way the brush operates in the game world. Examples include water, fog, nonsolid, and structural. Deformation effects change the actual shape of the affected brush or curve, and may make it appear to move.

Power has a price

The shader script gives the designer, artist and programmer a great deal of easily accessible power over the appearance of and potential special effects that may be applied to surfaces in the game world. But it is power that comes with a price tag attached, and the cost is measured in performance speed. Each shader phase that affects the appearance of a texture causes the engine to make another processing pass and redraw the world. Think of it as if you were adding all the shader-affected triangles to the total r_speed count for each stage in the shader script. A shader-manipulated texture that is seen through another shader manipulated texture (e.g.; a light in fog) has the effect of adding the total number of passes together for the affected triangles. A light that required two passes seen through a fog that requires one pass will be treated as having to redraw that part of the world three times.

RGB color

RGB means "Red, Green, Blue". Mixing red, green and blue light in differing intensities creates the colors in computers and television monitors. This is called additive color (as opposed to the mixing of pigments in paint or colored ink in the printing process, which is subtractive color). The intensities of the individual Red, Green and Blue components are expressed as number values. When mixed together on a screen, number values of equal intensity in each component color create a completely neutral (gray) color. The lower the number value (towards 0), the darker the shade. The higher the value, the lighter the shade or the more saturated the color until it reaches a maximum value of 255 (in the art programs). All colors possible on the computer can be expressed as a formula of three

numbers. The value for complete black is 0 0 0. The value for complete white is 255 255 255. However, the graphics engine requires that the color range be "normalized" into a range between 0.0 and 1.0.

Normalization: a scale of 0 to 1

The mathematics in the engine use a scale of 0.0 to 1.0 instead of 0 to 255. Most computer art programs that can express RGB values as numbers use the 0 to 255 scale. To convert numbers, divide each of the art program's values for the component colors by 255. The resulting three values are your formula for that color component. The same holds true for texture coordinates.

Texture sizes

Texture files are measured in pixels (picture elements). Textures are measured in powers of 2, with 16 x16 pixels being the smallest (typically) texture in use. Most will be larger. Textures need not be square, so long as both dimensions are powers of 2. Examples include: 32x256, 16x32, 128x16.

Color math

Colors are changed by mathematical equations worked on the textures by way of the scripts or "programlets" in the shader file. An equation that adds to or multiplies the number values in a texture causes it to become darker. Equations that subtract from or modulate number values in a texture cause it to become lighter. Either equation can change the hue and saturation of a color.

Measurements

The measurements used in the shaders are in either game units, color units, or texture units.

- **Game unit:** a game unit is used by deformations to specify sizes relative to the world. 8 game units equals one foot. The default texture scale used by the Q3Radiant map editor results in two texels for each game unit, but that can be freely changed.
- **Color units:** colors scale the values generated by the texture units to produce lighting effects. A value of 0.0 will be completely black, and a value of 1.0 will leave the texture unchanged. Colors are sometimes specified with a single value to be used across all red, green, and blue channels, or sometimes as separate values for each channel.
- **Texture units:** this is the normalized (see above) dimensions of the original texture image (or a previously modified texture at a given stage in the shader pipeline). A full texture, regardless of its original size in texels, has a normalized measurement of 1.0 x 1.0. For normal repeating textures, it is possible to have value greater than 1.0 or less than 0.0, resulting in repeating of the texture. The coordinates are usually assigned by the level editor or modeling tools, but you still need to be aware of this for scrolling or turbulent movement of the texture at runtime.

Waveform functions

Many of the shader functions use waveforms to modulate measurements over time. Where appropriate, additional information is provided with wave modulated keyword functions to describe the effect of a particular waveform on that process. Currently there are six waveforms in use in shaders:

- **sin:** sin stands for sine wave, a regular smoothly flowing wave ranging from -1 to 1.
 - **triangle:** Triangle is a wave with a sharp ascent and a sharp decay, ranging from 0 to 1. It will make a choppy looking wave forms.
 - **square:** a square wave simply switches from -1 to 1 with no in-between.
 - **sawtooth:** in the sawtooth wave, the ascent is like a triangle wave from 0 to 1, but the decay cuts off sharply back to 0.
 - **inversesawtooth:** this is the reverse of the sawtooth ... instant ascent to the peak value (1), then a triangle wave descent to the valley value (0). The phase on this goes from 1.0 to 0.0 instead of 0.0 to 1.0. This wave is
-

particularly useful for additive cross-fades.

- **noise**: randomly generated function, ranging from -1 to 1 . It's not likely to be continuous.

Waveforms all have the following properties:

- **<base>**: where the wave form begins. Amplitude is measured from this base value.
- **<amplitude>**: this is the height of the wave created, measured from the base. You will probably need to test and tweak this value to get it correct for each new shader stage. The greater the amplitude, the higher the wave peaks and the deeper the valleys.
- **<phase>**: this is a normalized value between 0.0 and 1.0 . Changing phase to a non-zero value affects the point on the wave at which the wave form initially begins to be plotted. Example: in Sin or Triangle wave, a phase of 0.25 means it begins one fourth (25%) of the way along the curve, or more simply put, it begins at the peak of the wave. A phase of 0.5 would begin at the point the wave re-crosses the base line. A phase of 0.75 would be at the lowest point of the valley. If only one wave form is being used in a shader, a phase shift will probably not be noticed and phase should have a value of zero (0). However, including two or more stages of the same process in a single shader, but with the phases shifted can be used to create interesting visual effects. Example: using rgbGen in two stages with different colors and a 0.5 difference in phase would cause the manipulated texture to modulate between two distinct colors. Phase changes can also be used when you have two uses of the same effect near each other, and you don't want them to be synchronized. You would write a separate shader for each, changing only the phase value.
- **<freq>**: frequency. This value is expressed as repetitions or cycles of the wave per second. A value of 1 would cycle once per second. A value of 10 would cycle 10 times per second. A value of 0.1 would cycle once every 10 seconds.

Distanceramp function

To be added.

General shader keywords

IMPORTANT NOTE: once again, be aware that some of the shader commands may be order dependent, so it's good practice to place all global shader commands (keywords defined in this section) at the very beginning of the shader and to place shader stages at the end (see various examples). These keywords are global to a shader and affect all stages. They are also ignored by Q3MAP.

skyParms <farbox> <cloudheight> <nearbox>

Specifies how to use the surface as a sky, including an optional far box (stars, moon, etc), optional cloud layers with any shader attributes, and an optional near box (mountains in front of the clouds, etc).

- **<farbox>**: specifies a set of files to use as an environment box behind all cloud layers. Specify "-" for no farbox, or a file base name. A base name of "env/test" would first look for files "env/test_px.tga", "env/test_py.tga", "env/test_nx.tga", "env/test_ny.tga", "env/test_nz.tga", "env/test_zy.tga" (the same Renderman naming convention is used for cubemaps), then for "env/test_rt.tga", "env/test_lf.tga", "env/test_ft.tga", "env/test_bk.tga", "env/test_up.tga", "env/test_dn.tga" to use as the right / left / front / back / up / down sides.
- **<cloudheight>**: controls apparent curvature of the cloud layers – lower numbers mean more curvature (and thus more distortion at the horizons). Higher height values create "flatter" skies with less horizon distortion. Think of height as the radius of a sphere on which the clouds are mapped. Good ranges are 64 to 256 . The default value is 128 .
- **<nearbox>**: specified as farbox, to be alpha blended on top of the clouds. This has not been tested in a long time, so it probably doesn't actually work. Set to "-" to ignore.

Design Notes:

- *If you are making a map where the sky is seen by looking up most of the time, use a lower cloudheight value. Under those circumstances the tighter curve looks more dynamic. If you are making a map where the sky is seen by looking out windows most of the time or has a map area that is open to the sky on one or more sides, use a higher height to make the clouds seem more natural. Be aware that the skybox does not wrap around the entire world. The "floor" or bottom face of the skybox is not drawn by the game. If a player in the game can see that face, they will see the "hall of mirrors" effect.*

Example: sky script

```
textures/skies/xtoxicsky_dm9
{
    qer_editorimage textures/skies/toxicsky.tga
    surfaceparm noimpact
    surfaceparm nolightmap
    q3map_globaltexture
    q3map_lightsubdivide 256
    q3map_surfacelight 400
    surfaceparm sky
    q3map_sun      1 1 0.5 150      30 60
    skyparms full 512 -
    {
        map textures/skies/inteldimclouds.tga
        tcMod scroll 0.1 0.1
        tcMod scale 3 2
    }
    {
        map textures/skies/intelredclouds.tga
        blendFunc add
        tcMod scroll 0.05 0.05
        tcMod scale 3 3
    }
}
```

cull <side>

Every surface of a polygon has two sides, a front and a back. Typically, we only see the front or "out" side. For example, a solid block you only show the front side. In many applications we see both. For example, in water, you can see both front and a back. The same is true for things like grates and screens. To "**cull**" means to remove. The value parameter determines the type of face culling to apply. The default value is cull **front** if this keyword is not specified. However for items that should be inverted then the value **back** should be used. To disable culling, the value **disable** or **none** should be used. Only one cull instruction can be set for the shader.

cull front

The front or "outside" of the polygon is not drawn in the world. This is the default value. It is used if the keyword "cull" appears in the content instructions without a <side> value or if the keyword cull does not appear at all in the shader.

cull back

Removes the back or "inside" of a polygon from being drawn in the world.

cull disable, cull none

Neither side of the polygon is removed. Both sides are drawn in the game. Very useful for making panels or barriers that have no depth, such as grates, screens, metal wire fences and so on and for liquid volumes that the player can see from within. Also used for energy fields, sprites, and weapon effects (e.g.; plasma).

Design Note:

- *For things like grates and screens, put the texture with the **cull none** property on one face only. On the other faces, use a non-drawing texture.*

deformVertexes

This function performs a general deformation on the surface's vertexes, changing the actual shape of the surface before drawing the shader passes. You can stack multiple **deformVertexes** commands to modify positions in more complex ways, making an object move in two dimensions, for instance.

Specific parameter definitions for deform keywords:

- <div>: this is roughly defined as the size of the waves that occur. It is measured in game units. Smaller values create a greater density of smaller wave forms occurring in a given area. Larger values create a lesser density of waves, or otherwise put, the appearance of larger waves. To look correct this value should closely correspond to the value (in pixels) set for tessSize (tessellation size) of the texture. A value of 100.0 is a good default value (which means your tessSize should be close to that for things to look "wavelike").
- <func>: this is the type of wave form being created. Sin stands for sine wave, a regular smoothly flowing wave. Triangle is a wave with a sharp ascent and a sharp decay. It will make a choppy looking wave forms. A square wave is simply on or off for the period of the frequency with no in between. The sawtooth wave has the ascent of a triangle wave, but has the decay cut off sharply like a square wave. An inversesawtooth wave reverses this.
- <base>: this is the distance, in game units that the apparent surface of the texture is displaced from the actual surface of the brush as placed in the editor. A positive value appears above the brush surface. A negative value appears below the brush surface. An example of this is the Quad effect, which essentially is a shell with a positive base value to stand it away from the model surface and a 0 (zero) value for amplitude.
- <amplitude>: the distance that the deformation moves away from the base value. See Wave Forms in the introduction for a description of amplitude.
- <phase>: see Waveforms in the introduction for a description of phase.
- <frequency>: see Waveforms in the introduction for a description of frequency.

Design Notes:

- *The div and amplitude parameters, when used in conjunction with liquid volumes like water should take into consideration how much the water will be moving. A large ocean area would have have massive swells (big div values) that rose and fell dramatically (big amplitude values). While a small, quiet pool may move very little.*

deformVertexes wave <div> <func> <base> <amplitude> <phase> <freq>

Designed for water surfaces, modifying the values differently at each point. It accepts the standard wave functions of the type **sin**, **triangle**, **square**, **sawtooth**, **inversesawtooth** or **noise**. The <div> parameter is used to control the wave "spread" – a value equal to the tessSize of the surface is a good default value (tessSize is subdivision size, in game units, used for the shader when seen in the game world).

deformVertexes normal <div> <func> <base> <amplitude ~0.1--0.5> <frequency ~1.0--4.0>

This deformation affects the normals of a vertex without actually moving it, which will effect later shader options like lighting and especially environment mapping. If the shader stages don't use normals in any of their calculations, there will be no visible effect.

Design Notes:

- *Putting values of 0.1 to 0.5 in Amplitude and 1.0 to 4.0 in the Frequency can produce some satisfying results. Some things that have been done with it: a small fluttering bat, falling leaves, rain, flags.*

deformVertexes bulge <bulgeWidth> <bulgeHeight> <bulgeSpeed> [bulgeBase]

This forces a bulge to move along the given s and t directions. Designed for use on curved pipes.

deformVertexes move <x> <y> <z> <func> <base> <amplitude> <phase> <freq>

This keyword is used to make a brush, curve patch or md3 model appear to move together as a unit. The <x> <y> and <z> values are the distance and direction in game units the object appears to move relative to its point of origin in the map. The <func> <base> <amplitude> <phase> and <freq> values are the same as found in other wave form manipulations. The product of the function modifies the values x, y, and z. Therefore, if you have an amplitude of 5 and an x value of 2, the object will travel 10 units from its point of origin along the x axis. This results in a total of 20 units of motion along the x axis, since the amplitude is the variation both above and below the base. It must be noted that an object made with this shader does not actually change position, it only appears to.

Design Notes:

- *If an object is made up of surfaces with different shaders, all must have matching deformVertexes move values or the object will appear to tear itself apart.*

deformVertexes autosprite

This function can be used to make any given triangle quad (pair of triangles that form a square rectangle) automatically behave like a sprite without having to make it a separate entity. This means that the "sprite" on which the texture is placed will rotate to always appear at right angles to the player's view as a sprite would. Any four-sided brush side, flat patch, or pair of triangles in a model can have the autosprite effect on it. The brush face containing a texture with this shader keyword must be square.

Design Notes:

- *This is best used on objects that would appear the same regardless of viewing angle. An example might be a glowing light flare.*

deformVertexes autosprite2

Is a slightly modified "sprite" that only rotates around the middle of its longest axis. This allows you to make a pillar of fire that you can walk around, or an energy beam stretched across the room.

deformVertexes autoparticle

This creates a "sprite" that is automatically scaled with distance to prevent it from disappearing (usually used for small particles).

fogparms <red value> <green value> <blue value> <distance to opaque> [clear distance]

Note: you must also specify "surfaceparm fog" to cause q3map to identify the surfaces inside the volume. Fogparms only describes how to render the fog on the surfaces.

- **<red value> <green value> <blue value>**: these are normalized values. A good computer art program should give you the RGB values for a color. To obtain the values that define fog color, divide the desired color's Red, Green and Blue values by 255 to obtain three normalized numbers within the 0.0 to 1.0 range.
- **<distance to opaque>**: this is the distance (in game units) until the fog becomes totally opaque, as measured from the point of view of the observer. By making the height of the fog brush shorter than the distance to opaque, the apparent density of the fog can be reduced (because it never reaches the depth at which full opacity occurs).
- **[clear distance]**: this is the distance (in game units) until the fog begins, as measured from the point of view of the observer. If not specified it will be set to 0.

The fog volume can only have one surface visible (from outside the fog). Fog must be made of one brush. It cannot be made of adjacent brushes. Fog brushes must be axial. This means that only square or rectangular brushes may contain fog, and those must have their edges drawn along the axes of the map grid (all 90 degree angles).

Design Notes:

- *If a water texture contains a fog parameter, it must be treated as if it were a fog texture when in use.*
- *If a room is to be filled completely with a fog volume, it can only be entered through one surface (and still have the fog function correctly).*
- *Additional shader passes may be placed on a fog brush, as with other brushes.*

nopicmip

This causes the texture to ignore user-set values for the r_picmip cvar command. The image will always be high resolution. **Example:** used to keep images and text in the heads up display from blurring when user optimizes the game graphics.

nomipmap

This implies nopicmip, but also prevents the generation of any lower resolution mipmaps for use by the 3d card. This will cause the texture to alias when it gets smaller, but there are some cases where you would rather have this than a blurry image. Sometimes thin slivers of triangles force things to very low mipmap levels, which leave a few constant pixels on otherwise scrolling special effects.

nofiltering

Disables bilinear and trilinear filtering modes for the shader.

polygonOffset

Surfaces rendered with the `polygonOffset` keyword are rendered slightly off the polygon's surface. This is typically used for wall markings and "decals". The distance between the offset and the polygon is fixed. It is not a variable.

portal

Specifies that this texture is the surface for a portal or mirror. For portal surfaces (mirrors do not require this setup anymore), in the game map, a portal entity must be placed directly in front of the texture (within 64 game units). All this does is set `sort portal`, so it isn't needed if you specify that explicitly.

sort <value>

Use this keyword to fine-tune the depth sorting of shaders as they are compared against other shaders in the game world. The basic concept is that if there is a question or a problem with shaders drawing in the wrong order against each other, this allows the designer to create a hierarchy of which shader draws in what order. The default behavior is to put all blended shaders in `sort additive` and all other shaders in `sort opaque`, so you only need to specify this when you are trying to work around a sorting problem with multiple transparent surfaces in a scene. The value here can be either a numerical value or one of the keywords in the following list (listed in order of ascending priority):

- **portal (1)**: this surface is a portal, it draws over every other shader seen inside the portal, but before anything in the main view.
- **sky (2)**: typically, the sky is the farthest surface in the game world. Drawing this after other opaque surfaces can be an optimization on some cards. This currently has the wrong value for this purpose, so it doesn't do much of anything.
- **opaque (3)**: this surface is opaque (rarely needed since this is the default with no `blendfunc`).
- **banner (6)**: transparent, but very close to walls.
- **underwater (8)**: draw behind normal transparent surfaces.
- **additive (9)**: normal transparent surface (default for shaders with `blendfuncs`).
- **nearest (16)**: this shader should always sort closest to the viewer, e.g. muzzle flashes and blend blobs.

noModulativeDlights

This keyword is used to tell the engine that it shouldn't apply standard modulative dynamic lights after this shader has been rendered (usually it is used in combination with manually placed additive dynamic lights pass).

if [!] <operand1> [<operator1 value1> <logic op [!] <opd1> [opr2 val2]>...]

Use this keyword when you want to skip other globally defined keywords or passes based on the operands' values. Generally it must be used when you have a shader stage that requires a specific hardware extension like `GL_ARB_texture_cube_map`.

The following mathematical operators are possible: `<` (less than), `<=` (less than or equal to), `==` (equal to), `>=` (greater than or equal to), `>` (greater than), `!=` (not equal to). The following logical operators are possible: `&&` (AND), `||` (OR). Use them to separate mathematical expressions. The optional `!` sign preceding a logical expression means NOT. There can be up to 8 mathematical and 7 logical operators in one if block.

The following operands may be used for building expressions:

- **maxTextureSize**: the hardware limit on the maximum texture size (width or height). It's NOT affected by the current `r_picmip` value.
- **maxTextureCubemapSize**: the hardware limit on the maximum cubemap texture size (width or height). It's NOT affected by the current `r_picmip` value.
- **maxTextureUnits**: the maximum number of texture units. It's ALWAYS greater or equal to 1.
- **textureCubeMap**: TRUE (1) if the hardware extension `GL_ARB_texture_cube_map` is present and FALSE (0) otherwise. It's affected by the current value of `gl_ext_texture_cube_map`.
- **textureEnvCombine**: TRUE (1) if the hardware extension `GL_ARB_texture_env_combine` is present and FALSE (0) otherwise. It's affected by the current value of `gl_ext_texture_env_combine`.
- **textureEnvDot3**: TRUE (1) if the hardware extension `GL_ARB_texture_env_dot3` is present and FALSE (0) otherwise. It's affected by the current value of `gl_ext_texture_env_dot3`. (deprecated)
- **GLSL**: TRUE (1) if the hardware extension `GL_ARB_shader_objects` is present and FALSE (0) otherwise. It's affected by the current value of `gl_ext_GLSL`.
- **deluxe** or **deluxeMaps**: TRUE (1) if GLSL is TRUE, deluxemaps are available in the current map and the value of `cvar r_lighting_deluxemapping` is non-zero.
- **portalMaps**: TRUE(1) if the value of `cvar r_portalmaps` is non-zero.

Example: checking whether the `GL_ARB_texture_cube_map` extension is present

```
textures/liquids/calm_poollight_cube
{
    ...

    if textureCubeMap
        // the same as 'if textureCubemap != FALSE'
        // the same as 'if ! textureCubemap == FALSE'
        // the same as 'if textureCubemap != FALSE && maxTextureUnits >= 1'
        {
            cubemap cubemaps/test
            rgbgen identity
        }
    endif

    ...
}
```

endif

This keyword marks the end of the **if** block. This makes nested **if** blocks possible.

template <name> [arg0] [arg1]...

This keyword indicates that the remaining part of the shader text should be reused from a different shader, matching the given name. Arguments passed to the template are then used to replace the placeholders in the template. Arguments containing whitespaces should be double quoted. For example, consider this template:

Example: simple shader template with placeholders

```
weapon_Hit_Template
{
    nopicmip
```

```

    cull none
    {
        map models/weapon_hits/$1/$2.tga
        rgbgen entity
        alphagen entity
        blendfunc GL_SRC_ALPHA GL_ONE
    }
}

models/weapon_hits/bullet/hit_bullet
{
    template weapon_Hit_Template bullet hit_bullet
}

```

The resulting shader is identical to the one below:

Example: processed shader template

```

models/weapon_hits/bullet/hit_bullet
{
    nopicmip
    cull none
    {
        map models/weapon_hits/bullet/hit_bullet.tga
        rgbgen entity
        alphagen entity
        blendfunc GL_SRC_ALPHA GL_ONE
    }
}

```

Q3Map2 doesn't support shader templates and therefore, if you want to use templates for non-game objects or 2D gfx you must duplicate Q3Map2 specific keywords used in the template in the shader as well.

skip [arg0] [arg1]...

This keyword indicates to the renderer that the rest of the line should be ignored (skipped). Can be used both in as a general shader command and stage specific command. Mainly intended to optionally disable features in templates.

Example: simple shader template with placeholders

```

weapon_Hit_Template
{
    $3
    cull none
    {
        map models/weapon_hits/$1/$2.tga
        rgbgen entity
        alphagen entity
        blendfunc GL_SRC_ALPHA GL_ONE
    }
}

```

```
models/weapon_hits/bullet/hit_bullet
{
    template weapon_Hit_Template bullet hit_bullet skip
}

models/weapon_hits/bullet/hit_bullet2
{
    template weapon_Hit_Template bullet hit_bullet nopicmip
}
```

In the example above only the second shader will be nopicmipped.

Q3MAP specific shader keywords

These keywords change the physical nature of the textures and the brushes that are marked with them. Changing any of these values will require the map to be re-compiled. These are global and affect the entire shader. For the latest version of q3map manual please refer to Q3Map2 Shader Manual ^[1].

Stage Specific Keywords

Stage specifications only affect rendering. Changing any keywords or values within a stage will usually take effect as soon as a vid_restart is executed. Q3MAP ignores stage specific keywords entirely.

A stage can specify a texture map, a color function, an alpha function, a texture coordinate function, a blend function, and a few other rasterization options.

Texture map specification

map [**\$rgb** or **\$alpha**] <texturePath/textureName>

Specifies the source texture map (a 24 or 32-bit TGA file) used for this stage. The texture may or may not contain alpha channel information. Keywords **\$rgb** and **\$alpha** are optional and may be used to extract the specified color range from the input texture: the **\$rgb** keyword takes red, green and blue components from the texture and ignores the alpha; **\$alpha** has an opposite meaning. The special keywords **\$lightmap**, **\$whiteimage**, **\$particleimage**, **\$dlight** may be substituted in lieu of an actual texture map name. In those cases, the texture named in the first line of the shader becomes the texture that supplies the light mapping data for the process.

- **\$lightmap** This is the overall lightmap for the game world. It is calculated during the Q3MAP process. It is the initial color data found in the framebuffer. Note: due to the use of overbright bits in light calculation, the keyword **rgbGen identity** must accompany all **\$lightmap** instructions.
- **\$whiteimage** This is used for specular lighting on models. This is a white image generated internally by the game. This image can be used in lieu of **\$lightmap** or an actual texture map if, for example, you wish for the vertex colors to come through unaltered.
- **\$particleimage** This is a name for internally computed image used for particles (always the same).
- **\$dlight** This is an instruction indicating that this pass is reserved for dynamic lights usually used in combination with **blendFunc add** and **depthfunc equal**.
- **\$portalmap/\$mirrormap** An instruction indicating that contents of a portal have to be rendered to a texture. After that, this texture may be used as a regular shader image.

clampMap <texturePath/textureName>

Dictates that this stage should clamp texture coordinates instead of wrapping them. During a stretch function, the area, which the texture must cover during a wave cycle, enlarges and decreases. Instead of repeating a texture multiple times during enlargement (or seeing only a portion of the texture during shrinking) the texture dimensions increase or contract accordingly. This is only relevant when using something like **tcMod stretch** to stretch/compress texture coordinates for a specific special effect.

- **Proper Alignment** When using **clampMap** be make sure the texture is properly aligned on the brush. The **clampMap** function keeps the image from tiling. However, the editor doesn't represent this properly and shows a tiled image. Therefore, what appears to be the correct position may be offset. This is very apparent on anything with a **tcMod rotate** and **clampMap** function.
- **Avoiding Distortion** When seen at a given distance (which can vary, depending on hardware and the size of the texture), the compression phase of a stretch function will cause a "cross"-like visual artifact to form on the modified texture due to the way that textures are reduced. This occurs because the texture undergoing modification lacks sufficient "empty space" around the displayed (non-black) part of the texture. To compensate for this, make the non-zero portion of the texture substantially smaller (50% of maximum stretched size) than the dimensions of the texture. Then, write a scaling function (**tcMod scale**) into the appropriate shader phase, to enlarge the image to the desired proportion.

animMap <frequency> <texture1> ... <texture16>

The surfaces in the game can be animated by displaying a sequence of 1 to 16 frames (separate texture maps). These animations are affected by other keyword effects in the same and later shader stages.

- **<frequency>** the number of times that the animation cycle will repeat within a one second time period. The larger the value, the more repeats within a second. Animations that should last for more than a second need to be expressed as decimal values.
- **<texture1> ... <texture16>** the texturepath/texture name for each animation frame must be explicitly listed. Up to sixteen frames (sixteen separate .tga files) can be used to make an animated sequence. Each frame is displayed for an equal subdivision of the frequency value.

Example: *AnimMap 0.25 animMap 10 textures/sfx/b_flame1.tga textures/sfx/b_flame2.tga textures/sfx/b_flame3.tga textures/sfx/b_flame4.tga* would be a 4 frame animated sequence, calling each frame in sequence over a cycle length of 4 seconds. Each frame would be displayed for 1 seconds before the next one is displayed. The cycle repeats after the last frame in sequence is shown.

Design Note:

To make a texture image appear for an unequal (longer) amount of time (compared to other frames), repeat that frame more than once in the sequence.

animClampMap <frequency> <texture1> ... <texture16>

This is a combination of both **clampMap** and **animMap** commands.

videoMap <videoPath/videoName>

This command allows level designers to play a cinematic sequence on any kind of surface. Be aware that this causes a big performance hit since the video frames are streamed and decompressed on the fly. Only RoQ video format is supported. It is recommended to create videos at resolutions not higher than 256x256 to save processing time.

cubeMap <basePath/baseName>

Specifies a set of files to use as a cube map texture for environment mapping and reflections. A base name of "env/test" would first look for files "env/test_px.tga", "env/test_nx.tga", "env/test_py.tga", "env/test_ny.tga", "env/test_nx.tga", "env/test_ny.tga" (the Renderman naming convention), then for "env/test_rt.tga", "env/test_lf.tga", "env/test_bk.tga", "env/test_ft.tga", "env/test_up.tga", "env/test_dn.tga" to use as the right / left / front / back / up / down sides.

Design Notes:

- Cube map textures must be square (64x64, 128x128, 256x256, and so on) and all 2D images that form a cube map must have equal dimensions. It is possible to quickly create cube map textures from within the qfusion engine using the console command "envshot".
- You can use as many cube maps as you want in a level, but keep in mind that they take up 6 times the memory resources that a normal texture takes. Not all video cards support cube mapping, so it's a good idea to create a good 2D texture that will be used as a fall back if cube mapping is not available on the user's computer.
- This command requires `GL_ARB_texture_cube_map` extension, so it is recommended to check if it is present via "ifendif" commands.

normalMap [\$heightMap bumpScale**] <normalPath/normalName>
<\$noimage\diffuseName>**

Indicates that this is a normal mapping stage. Normal mapping is used to produce realistic lighting effects on game objects.

- [**\$heightMap bumpScale**] two optional parameters indicating that you provide your normalmap in a form of a heightmap. **bumpScale** tells the engine how "bumpy" the resulting normalmap should look.
- **<\$noimage\diffuseName>** **\$noimage** as the final diffuse map stage it tells the engine to skip it (you'll just see the result of dot3 bumpmapping with no colors and no texture). **diffuseName** parameter points to a diffuse texture.

Design Notes:

- This keyword is deprecated in favor of 'material'. Backwards compatibility via GLSL is kept though. If you specify **\$heightmap** you should also provide the **bumpScale**.
- This command requires `GL_ARB_texture_env_dot3` extension, so it is recommended to check if it is present via "ifendif" commands.
- This keyword would probably not be used by a level designer.

material [\$rgb\$alpha\$left\$right] diffusemap [bumpmap bumpScale/normalmap] [glossmap] [decalsmap]

Materials are used to define a deluxemap/dot3 shader stage. Any numeric value after diffusemap (texture) defines bumpscale - the "bumpyness" of heightmap. If no bumpmap has been specified, the engine will attempt to load the default images (with `_bump` and `_gloss` suffixes). **material textures/walls/stupid_wall.tga** loads textures/walls/stupid_wall_bump as heightmap with default bumpscale (or loads textures/walls/stupid_wall_norm as normalmap), textures/walls/stupid_wall_gloss as glossmap and either textures/walls/stupid_wall_decal or textures/walls_stupid_wall_add as decalsmap.

Materials are effectively shader metapasses: they also include additive dlights pass and optional modulative diffuse pass for world surfaces. A material stage also automatically sets `noModulativeDlights` to true for its parent shader.

Design Notes:

- Glossmaps are not required for materials to work properly, but heightmaps/normalmaps are.
- If one specifies both decalsmap and `rgbgen` or `alphagen` for a material pass, the later will be applied to the decalsmap only.
- Decalsmaps with no alpha channel use additive blending (equivalent of **blendfunc gl_one gl_one**), otherwise alpha blending is used.

distortion dudvmap [normalmap]

Distortion pass is a special rendering stage that's similar to portals and mirrors. It renders the surrounding geometry into two textures (front plane and back plane) and may be possibly used for creating water ripples, refraction and reflection effects. dudvmap is an image describing the post-rendering deformations applied to geometry on the back plane. normalmap, on the other hand, may be optionally used to create Fresnel effect when the angle between the viewer and the surface of the object affects the amount of light that is reflected and refracted. Reflectiveness of the distortion pass is controlled via `alphagen`. I.e. '`alphagen const 1.0`' creates a perfect reflective surface, '`alphagen const 0`' creates a perfect refractive surface, '`alphagen wave`' makes the distortion surface change its reflection/refraction properties dynamically.

Design Notes:

- The stage currently requires 'portal' keyword declaration in the shader or alternatively 'sort portal'.

Blend Functions

Blend functions are the keyword commands that tell the qfusion graphic engine's renderer how graphic layers are to be mixed together.

Simplified blend functions:

The most common blend functions are set up here as simple commands, and should be used unless you really know what you are doing.

blendfunc add

This is a shorthand command for **blendfunc gl_one gl_one**. Effects like fire and energy are additive.

blendfunc filter

This is a shorthand command that can be substituted for either blendfunc **gl_dst_color gl_zero** or **blendfunc gl_zero gl_src_color**. A filter will always result in darker pixels than what is behind it, but it can also remove color selectively. Lightmaps are filters.

blendfunc blend

Shorthand for blendfunc **gl_src_alpha gl_one_minus_src_alpha**. This is conventional transparency, where part of the background is mixed with part of the texture.

Explicit blend functions

Getting a handle on this concept is absolutely key to understanding all shader manipulation of graphics. BlendFunc or “Blend Function” is the equation at the core of processing shader graphics. The formula reads as follows:

```
[Source * <srcBlend>] + [Destination * <dstBlend>]
```

- **Source** is usually the RGB color data in a texture TGA file (remember it’s all numbers) modified by any rgbgen and alphagen. In the shader, the source is generally identified by command MAP, followed by the name of the image.
- **Destination** is the color data currently existing in the frame buffer.

Rather than think of the entire texture as a whole, it may be easier to think of the number values that correspond to a single pixel, because that is essentially what the computer is processing: one pixel of the bit map at a time.

The process for calculating the final look of a texture in place in the game world begins with the precalculated lightmap for the area where the texture will be located. This data is in the frame buffer. That is to say, it is the initial data in the **Destination**. In an unmanipulated texture (i.e.; one without a special shader script), color information from the texture is combined with the lightmap. In a shader-modified texture, the **\$lightmap** stage must be present for the lightmap to be included in the calculation of the final texture appearance.

Each pass or “stage” of blending is combined (in a cumulative manner) with the color data passed onto it by the previous stage. How that data combines together depends on the values chosen for the Source Blends and Destination Blends at each stage. Remember it’s numbers that are being mathematically combined together that are ultimately interpreted as colors.

A general rule is that any **Source Blend** other than **GL_ONE** (or **GL_SRC_ALPHA** where the alpha channel is entirely white) will cause the Source to become darker.

Source Blend <srcBlend>

The following values are valid for the Source Blend part of the equation.

- **GL_ONE** This is the value 1. When multiplied by the Source, the value stays the same the value of the color information does not change.
- **GL_ZERO** This is the value 0. When multiplied by the Source, all RGB data in the Source becomes zero (essentially black).
- **GL_DST_COLOR** This is the value of color data currently in the Destination (frame buffer). The value of that information depends on the information supplied by previous stages.
- **GL_ONE_MINUS_DST_COLOR** This is nearly the same as **GL_DST_COLOR** except that the value for each component color is inverted by subtracting it from one (i.e.; $R = 1.0 - \text{DST.R}$,

$G = 1.0 - \text{DST.G}$, $B = 1.0 - \text{DST.B}$, etc.).

- **GL_SRC_ALPHA** The TGA file being used for the Source data must have an alpha channel in addition to its RGB channels (for a total of four channels). The alpha channel is an 8-bit black and white only channel. An

entirely white alpha channel will not darken the Source.

- **GL_ONE_MINUS_SRC_ALPHA** This is the same as **GL_SRC_ALPHA** except that the value in the alpha channel is inverted by subtracting it from one. (i.e.; $A=1.0 - SRC.A$)

Destination Blend <dstBlend>

The following values are valid for the Destination Blend part of the equation.

- **GL_ONE** This is the value 1. When multiplied by the Destination, the value stays the same the value of the color information does not change.
- **GL_ZERO** This is the value 0. When multiplied by the Destination, all RGB data in the Destination becomes Zero (essentially black).
- **GL_SRC_COLOR** This is the value of color data currently in the Source (which is the texture being manipulated here).
- **GL_ONE_MINUS_SRC_COLOR** This is the value of color data currently in Source, but subtracted from one (i.e.: inverted).
- **GL_SRC_ALPHA** The TGA file being used for the Source data must have an alpha channel in addition to its RGB channels (four a total of four channels). The alpha channel is an 8-bit black and white only channel. An entirely white alpha channel will not darken the Source.
- **GL_ONE_MINUS_SRC_ALPHA** This is the same as **GL_SRC_ALPHA** except that the value in the alpha channel is inverted by subtracting it from one. (i.e.; $A=1.0 - SRC.A$).

Doing the Math: The Final Result

The product of the **Source** side of the equation is added to the product of the **Destination** side of the equation. The sum is then placed into the frame buffer to become the **Destination** information for the next stage. Ultimately, the equation creates a modified color value that is used by other functions to define what happens in the texture when it is displayed in the game world.

Default Blend Function

If no **blendFunc** is specified then no blending will take place.

Technical Information/Limitations Regarding Blend Modes:

Cards running in 16 bit color cannot use any **GL_DST_ALPHA** blends.

rgbGen <func>

There are two color sources for any given shader, the texture file and the vertex colors. Output at any given time will be equal to **TEXTURE** multiplied by **VERTEXCOLOR**. Most of the time **VERTEXCOLOR** will default to white (which is a normalized value of 1.0), so output will be **TEXTURE** (this usually lands in the Source side of the shader equation). Sometimes you do the opposite and use **TEXTURE = WHITE**, but this is only commonly used when doing specular lighting on entities (i.e.; shaders that level designers will probably never create)

The most common reason to use **rgbGen** is to pulsate something. This means that the **VERTEXCOLOR** will oscillate between two values, and that value will be multiplied (darkening) the texture.

If no **rgbGen** is specified, either **identityLighting** or **identity** will be selected, depending on which blend modes are used.

Valid <func> parameters are **wave**, **colorwave**, **identity**, **identityLighting**, **entity**, **entityColorWave**, **oneMinusEntity**, **vertex**, **exactVertex**, **oneMinusVertex**, **lightingDiffuse**, **lightingDiffuseOnly**, **lightingAmbientOnly**, **custom**, **customWave**, **customColorWave**, **teamColor**, **teamColorWave** and **const**.

rgbGen identityLighting

Colors will be (1.0,1.0,1.0) if running without overbright bits (NT, linux, windowed modes), or (0.5, 0.5, 0.5) if running with overbright. Overbright allows a greater color range at the expense of a loss of precision. Additive and blended stages will get this by default.

rgbGen identity

Colors are assumed to be all white (1.0,1.0,1.0). All filters stages (lightmaps, etc) will get this by default.

rgbGen wave <func> <base> <amp> <phase> <freq>

Colors are generated using the specified waveform. An affected texture will become darker and lighter, but will not change hue. Hue stays constant. Note that the rgb values for color will not go below 0 (black) or above 1 (white). Valid waveforms are **sin**, **triangle**, **square**, **sawtooth**, **inversesawtooth**, **distanceramp** and **noise**.

- **<wave>** May be any waveform
- **<base>** Baseline value. The initial RGB formula of a color (normalized).
- **<amp>** Amplitude. This is the degree of change from the baseline value. In some cases you will want values outside the 0.0 to 1.0 range, but it will induce clamping (holding at the maximum or minimum value for a time period) instead of continuous change.
- **<phase>** See the explanation for phase under the waveforms heading of Key Concepts.
- **<freq>** Frequency. This is a value (NOT normalized) that indicates peaks per second.

rgbGen colorwave <red> <green> <blue> <func> <base> <amp> <phase> <freq>

Colors are generated multiplying the constant value with the specified waveform. Note that the rgb values for color will not go below 0 (black) or above 1 (white). Valid waveforms are **sin**, **triangle**, **square**, **sawtooth**, **inversesawtooth** and **noise**.

rgbGen entity

Colors are grabbed from the entity's **modulate** field. This is used for things like explosions.

Design Note:

- This keyword would probably not be used by a level designer.

rgbGen oneMinusEntity

Colors are grabbed from 1.0 minus the entity's modulate field.

Design Note:

- This keyword would probably not be used by a level designer.

rgbGen vertex

Colors are filled in directly by the data from the map or model files.

Design Note:

- **rgbGen vertex** should be used when you want the RGB values to be computed for a static model (i.e. mapobject) in the world using precomputed static lighting from Q3BSP. This would be used on things like the gargoyles, the portal frame, skulls, and other decorative models put into the world.

rgbGen exactVertex

Colors are filled in directly by the data from the map or model files. No overbrights bitshifting is performed.

rgbGen oneMinusVertex

Same as **rgbGen vertex**, but inverted.

Design Note:

- This keyword would probably not be used by a level designer.

rgbGen lightingDiffuse

Colors are computed using a standard diffuse lighting equation. It uses the vertex normals to illuminate the object correctly.

Design Note:

- **rgbGen lightingDiffuse** is used when you want the RGB values to be computed for a dynamic model (i.e. non-map object) in the world using regular in-game lighting. For example, you would specify on shaders for items, characters, weapons, etc.

rgbGen lightingDiffuseOnly

Colors are computed using a standard diffuse lighting equation. It uses the vertex normals to illuminate the object correctly. Not affected by ambient lighting.

rgbGen lightingAmbientOnly

Only ambient lighting is applied. Not affected by diffuse lighting.

rgbGen const\constant <red> <green> <blue>

Generates a constant RGB value.

rgbGen custom\teamColor <num>

Generates a constant RGB value set by the clientside game module.

alphaGen <func>

The alpha channel can be specified like the RGB channels. If not specified, either “identity” or “vertex” will be selected.

=== alphaGen identity Alpha is assumed to be 1.0 (no translucency).

alphaGen wave <func> <base> <amplitude> <phase> <frequency>

Alpha is generated using the specified waveform. Note that the alpha value will not go below 0 (transparent) or above 1 (opaque). See the explanation of wave functions.

alphaGen vertex\exactVertex

Alpha is filled in directly by the data from the map or model files.

alphaGen oneMinusVertex

Same as alphaGen vertex, but inverted.

alphaGen entity

Alpha is grabbed from the entity's modulate field. This is used for things like explosions.

alphaGen oneMinusEntity

Same as alphaGen entity, but inverted.

alphaGen dot [min] [max]

Alpha is generated from the dot product of the surface normal and the view angle. Ranges from 0 (transparent) for parallel view to 1 (opaque) for perpendicular view. You can optionally specify **[min]** and **[max]** to clamp the range. Great for simulating the varying translucency of water or reflective surfaces based on view angle.

alphaGen oneMinusDot [min] [max]

Same as alphaGen dot, but inverted.

alphaGen portal [range]

This function accomplishes the "fade" that causes the scene in the portal to fade from view. Specifically, it means "generate alpha values based on the distance from the viewer to the portal". If **[range]** is not specified, a default value of 256 will be used. Use alphaGen portal on the last rendering pass.

alphaGen lightingSpecular [invpow]

Creates specular highlights in the alpha channel, typically used in conjunction with **\$whiteimage**.

alphaGen const\constant <alpha>

Used to specify a constant alpha value.

tcGen <coordinate source>

Specifies how texture coordinates are generated and where they come from. Valid functions are base, lightmap, environment, cellshade, reflection and vector.

- **<base>** = base texture coordinates from the original art.
 - **<lightmap>** = lightmap texture coordinates
 - **<environment>** = Make this object environment mapped.
 - **<cellshade">** = Used for game objects to emulate the cellshading effect.
 - **<reflection">** = Make this object environment mapped using cube mapping.
 - **<vector">** = New texcoord generation by world projection.
-

tcGen vector (<sx> <sy> <sz>) (<tx> <ty> <tz>)

Allows you to project a texture onto a surface in a fixed way, regardless of its orientation.

- S coordinates correspond to the “x” coordinates on the texture itself.
- T coordinates correspond to the “y” coordinates on the texture itself.

The measurements are in game units.

Example:

- tcGen vector (0.01 0 0) (0 0.01 0) would project a texture with a repeat every 100 units across the X/Y plane.

tcMod <func> <...>

Specifies how texture coordinates are modified after they are generated. The valid functions for tcMod are **rotate**, **scale**, **scroll**, **stretch** and **transform**. **Transform** is a function generally reserved for use by programmers who suggest that designers leave it alone. When using multiple **tcMod** functions during a stage, place the **scroll** command last in order, because it performs a mod operation to save precision, and that can disturb other operations. Texture coordinates are modified in the order in which tcMods are specified. In other words, if you see:

- tcMod scale 0.5 0.5
- tcMod scroll 1 1

Then the texture coordinates will be **scaled** then **scrolled**.

tcMod rotate <degrees per per second>

This keyword causes the texture coordinates to rotate. The value is expressed in degrees rotated each second. A positive value means clockwise rotation. A negative value means counterclockwise rotation. For example “tcMod rotate 5” would rotate texture coordinates 5 degrees each second in a clockwise direction. The texture rotates around the center point of the texture map, so you are rotating a texture with a single repetition, be careful to center it on the brush (unless off-center rotation is desired).

tcMod scale <sScale> <tScale>

Resizes (enlarges or shrinks) the texture coordinates by multiplying them against the given factors of **<sScale>** and **<tScale>**. The values “s” and “t” conform to the “x” and “y” values (respectively) as they are found in the original texture TGA. The values for sScale and tScale are NOT normalized. This means that a value greater than 1.0 will increase the size of the texture. A positive value less than one will reduce the texture to a fraction of its size and cause it to repeat within the same area as the original texture (Note: see **clampMap** for ways to control this).

Example: * *tcMod scale 0.5 2* would cause the texture to repeat twice along its width, but expand to twice its height (in which case half of the texture would be seen in the same area as the original).

tcMod scroll <sSpeed> <tSpeed>

Scrolls the texture coordinates with the given speeds. The values “s” and “t” conform to the “x” and “y” values (respectively) as they are found in the original texture TGA. The scroll speed is measured in “textures” per second. A “texture” is the dimension of the texture being modified and includes any previous shader modifications to the original TGA). A negative s value would scroll the texture to the left. A negative t value would scroll the texture down.

Example: * tcMod scroll 0.5 -0.5 moves the texture down and right (relative to the TGA files original coordinates) at the rate of a half texture each second of travel.

This should be the **LAST** tcMod in a stage. Otherwise there may be popping or snapping visual effects in some shaders.

tcMod stretch <func> <base> <amplitude> <phase> <frequency>

Stretches the texture coordinates with the given function. Stretching is defined as stretching the texture coordinate away from the center of the polygon and then compressing it towards the center of the polygon.

- **<func>** See the explanation of wave functions.
- **<base>** A base value of one is the original dimension of the texture when it reaches the stretch stage. Inserting other values positive or negative in this variable will produce unknown effects.
- **<amplitude>** This is the measurement of distance the texture will stretch from the base size. It is measured, like scroll, in textures. A value of 1 here will double the size of the texture at its peak.
- **<phase>** See the explanation for phase under the deform vertexes keyword.
- **<frequency>** This is wave peaks per second.

tcMod <transform> <m00> <m01> <m10> <m11> <t0> <t1>

Transforms each texture coordinate as follows:

$$S' = s * m00 + t * m10 + t0$$

$$T' = t * m01 + s * m11 + t1$$

This is for use by programmers.

tcMod turb <base> <amplitude> <phase> <freq>

Applies turbulence to the texture coordinate. Turbulence is a back and forth churning and swirling effect on the texture. The parameters for this shader are defined as follows:

- **<base>** Currently undefined.
- **<amplitude>** This is essentially the intensity of the disturbance or twisting and squiggling of the texture.
- **<phase>** See the explanation for phase under the deformVertexes keyword.
- **<freq>** Frequency. This value is expressed as repetitions or cycles of the wave per second. A value of one would cycle once per second. A value of 10 would cycle 10 times per second. A value of 0.1 would cycle once every 10 seconds.

depthFunc <func>

This controls the depth comparison function used while rendering. The default is “lequal” (less than or equal to) where any surface that is at the same depth or closer of an existing surface is drawn. This is used for textures with transparency or translucency. Under some circumstances you may wish to use “equal”, e.g. for lightmapped grates that are alpha tested (it is also used for mirrors) or “gequal” (greater than or equal to).

depthWrite

By default, writes to the depth buffer when depthFunc passes will happen for opaque surfaces and not for translucent surfaces. Blended surfaces can have the depth writes forced with this function.

Detail

Designates this stage as a detail texture stage, which means that if `r_detailtextures` is set to 0 then this stage will be ignored (detail will not be displayed). This keyword, by itself, does not affect rendering at all.

Grayscale

When used in conjunction with a GLSL program, the final pixel color is converted to grayscale.

alphaFunc <func>

Determines the alpha test function used when rendering this map. Valid values are **GT0**, **LT128**, and **GE128**. These correspond to “**GREATER THAN 0**”, “**LESS THAN 128**”, and “**GREATER THAN OR EQUAL TO 128**”. This function is used when determining if a pixel should be written to the framebuffer. For example, if **GT0** is specified, the only the portions of the texture map with corresponding alpha values greater than zero will be written to the framebuffer. By default alpha testing is disabled.

Both alpha testing and normal alpha blending can be used to get textures that have see-through parts. The difference is that `alphaFunc` is an all-or-nothing test, while blending smoothly blends between opaque and translucent at pixel edges. Alpha test can also be used with `depthwrite`, allowing other effects to be conditionally layered on top of just the opaque pixels by setting `depthFunc` to “**equal**”.

skip [arg0] [arg1]...

This command acts identically to the eponymous general shader command.

Notes on alpha channels

To use some blend modes of `alphaFunc`, you must add an alpha channel to your texture files. Photoshop can do this. Paintshop Pro has the ability to make an alpha channel cannot work directly in to it. In Photoshop you want to set the type to Mask. Black has a value of 255. White has a value of 0. The darkness of a pixel’s alpha value determines the transparency of the corresponding RGB value in the game world. Darker means more transparent.

Care must be taken when reworking textures with alpha channels. Textures without alpha channels are saved as 24 bit images while textures with alpha channels are saved as 32 bit. If you save them out as 24 bit, the alpha channel is erased. Note: Adobe Photoshop will prompt you to save as 32, 24 or 16 bit. Choose wisely. If you save a texture as 32 bit and you don’t actually have anything in the alpha channel, the engine may still be forced to use a lower quality texture format (when in 16 bit rendering) than if you had saved it as 24 bit.

To create a texture that has “open” areas, make those areas black in the alpha channel and make white the areas that are to be opaque. Using gray shades can create varying degrees of opacity/transparency.

The alpha channel can also be used to merge a texture (including one that contains black) into another image so that the merged art appears to be an opaque decal on a solid surface (unaffected by the surface it appears to sit on), without actually using an alpha function.

In a like manner, the alpha channel can be used to blend the textures more evenly. A simple experiment involves using a linear gradient in the alpha channel (white to black) and merging two textures so they appear to cross fade into each.

More examples

- **Example: world shader with additive dlights**

```
textures/bump/metal1
{
    qer_editorimage textures/bump/metal1.jpg
    q3map_normalimage textures/bump/metal1b.jpg

    noModulativeDlights

    {
        map $lightmap
    }
    {
        map $dlight
        blendfunc add
    }
    {
        map textures/bump/metal1.jpg
        blendFunc filter
    }
}
```

- **Example: world shader with additive dlights and alpha testing**

```
textures/effects/herbe_1
{
    qer_editorimage textures/effects/herbe.tga
    surfaceparm nonsolid
    surfaceparm nomarks
    surfaceparm alphashadow
    cull none

    {
        map $alpha textures/effects/herbe.tga
        alphaFunc GE128
        rgbGen identity
    }
    {
        map $lightmap
        blendFunc filter
    }
}
```

```
        depthFunc equal
    }
    {
        map $dlight
        blendFunc add
        depthFunc equal
    }
    {
        map $rgb textures/effects/herbe.tga
        blendFunc filter
        depthFunc equal
    }
}
```

- **Example: material (world surface)**

```
textures/bump/metall
{
    qer_editorimage textures/bump/metall.jpg

    if deluxe
    {
        material textures/bump/metall.jpg
    }
    endif

    if ! deluxe

    noModulativeDlights

    {
        map $lightmap
    }
    {
        map $dlight
        blendfunc add
    }
    {
        map textures/bump/metall.jpg
        blendFunc filter
    }

    endif
}
```

- **Example: material (game object)**

```
players/knight/knight
{
```

```
if deluxe
{
    material players/knight/pknight players/knight/pknight_norm players/knight/pknight_gloss
}
endif

// fallback path
if ! deluxe

{
    map players/knight/pknight_prelit
    rgbgen lightingDiffuse
}

endif
}
```

- **Example: material with additive dlights and alpha testing (world surface)**

```
textures/effects/herbe_1
{
    qer_editorimage textures/effects/herbe.tga
    surfaceparm nonsolid
    surfaceparm nomarks
    surfaceparm alphashadow
    cull none

    if deluxe

    {
        map $alpha textures/effects/herbe.tga
        alphaFunc GE128
        rgbGen identity
    }
    {
        material $whiteimage textures/effects/herbe_norm.tga
        blendFunc add
        depthFunc equal
    }
    {
        map $rgb textures/effects/herbe.tga
        blendFunc filter
        depthFunc equal
    }

    endif

    if ! deluxe
```

```
noModulativeDlights
{
    map $alpha textures/effects/herbe.tga
    alphaFunc GE128
    rgbGen identity
}
{
    map $lightmap
    blendFunc filter
    depthFunc equal
}
{
    map $dlight
    blendFunc add
    depthFunc equal
}
{
    map $rgb textures/effects/herbe.tga
    blendFunc filter
    depthFunc equal
}

endif
}
```

- **Example: Distorted mirror or portal**

```
textures/sfx/portal_sfx
{
    portal
    surfaceparm nolightmap
    surfaceparm nomarks
    cull none

    if portalMaps
        if GLSL
        {
            distortion textures/blx_wtest3/dudv_map.jpg
            alphaGen const 0.5
            tcmmod scroll 0.04 0
            depthWrite
        }
    endif

    if ! GLSL
    {
        map $portalmap
    }
}
```

```
        blendfunc blend
        alphagen const 0.5
        tcmmod scroll 0.04 0
        depthWrite
    }
    endif
endif

if ! PortalMaps
    deformVertexes wave 100 sin 0 2 0 .5

    {
        ...
    }
endif
}
```

.

References

[1] http://q3map2.everyonelookbusy.net/shader_manual/