# *An Introduction to IRIS Performer™ 2.1 and InfiniteReality™*

***Brought to you by members of the Performer2.1 Team:***
***Sharon Rose Clay, Tom McReynolds, Jenny Zhao***
***Silicon Graphics Computer Systems***

***Core Rendering Notes,***
***Developer's Forum '96***

## *1 Introduction*

IRIS Performer is a high-performance 3D rendering toolkit for developing visualization applications for the Silicon Graphics product line. Real-time rendering of very large databases is provided via multiprocessing scene and graphics state management techniques. IRIS Performer also provides easy and high-performance access to sophisticated graphics features and the utilities needed for making these features into application solutions.

IRIS Performer 2.1 is designed to bring out the best of the Onyx/InfiniteReality (iR) system by providing direct support for advanced features of the iR. Examples include very large MIPmap textures (up to 32768x32768) for geospecific texture (called clipmapped textures), dynamic video resolution to control pixel rendering load, active surface definitions for object morphing, graphics pipeline instrumentation for tuning and load management, and fast database loading. This introduction will cover the basics for adding these and other new features to your application that will enable you to build applications that handle the infinite, dynamic databases for which InfiniteReality was designed. The following topics are covered:

- Infinite dynamic texture with texture loading and clip-mapping,
- Dynamic terrain with the Active Surface Definition library,
- Dynamic channel resolution with video resizing,
- Fast database loading with database paging and libpfpfb,

- Additional texture affects: image distortion with framebuffer textures, detail textures, 3D textures
- tools for debugging graphics performance

## 2 Infinite Dynamic Texture

Basic texture loading and management of textures in hardware texture memory has been supported in IRIS Performer as of release 2.0. Texture loading operations for base textures and MIPmaps can be configured with pfTexLoad* routines such as **pfTexLoadMode()**, **pfTexLoadOrigin()**, **pfTexLoadsize()**, and **pfTexLoadImage()**. **pfTexList()** can be used to configure sets of mutually-exclusive textures in texture memory. Textures can be specifically removed from texture memory with **pfIdleTex()**. The following code stub demonstrates the use of these routines.

Example 0-1 Texture Loading

```
pfTexture *base, *tex, **texLevels;
int i,j;

base= pfNewTexture();
. . .
/* configure base texture for loading */
pfTexFormat(base, PFTEX_SUBLOAD_FORMAT, 1);
/* turn off MIPmap generation since we will load our own */
pfTexFormat(base, PFTEX_GEN_MIPMAP_FORMAT, 1);
/* set source - could be framebuffer, video, or image */
pfTexLoadMode(base, PFTEX_LOAD_SOURCE, PFTEX_SOURCE_VIDEO);
....
/* set MIPmap levels */
for (i=0; i < numLevels; i++)
{
     tex = texLevels[i];
          pfTexLevel(base, i, tex);
     pfTexLoadSize(tex, 32, 32);
}
. . .
/* do per-frame loading */
if (SwapOutTex)
     pfIdleTex(base);
else
{
     /* update base level image pointer */
     pfTexLoadImage(base, baseImagePtr);
     for (i=0; i < numLevels; i++)
     {
          /* update image pointer for current level */
          pfTexLoadImage(pfGetTexLevel(base,i),
                         levelImagePtrs[i]);
```

```
            }
            /* trigger the updating loads */
            pfLoadTex(base);
        }
```

This of texture management supports the basic loading of new textures for individual objects from a paging database. Typically, these textures are not terribly large, though there may be many of them. This method works well for independent textures, but is not convenient for managing a large set of correlated textures, such as the tiling of a very large terrain.

Previously on SGI graphics platforms, textures too large to fit into texture memory had to be cut up into individual tiles and managed as separate textures with special overlaps and borders to hide filtering artifacts. Polygons in the database had to be cut so that they would always fit in a tile. While good looking results are achievable with this method, it is hard to implement and places constraints on the database.

IRIS Performer 2.1 on InfiniteReality allows you to create textures that are much bigger than will fix in texture memory, and even in system memory, so that you can fit your entire world in a single texture. IRIS Performer will also manage all of the texture loading for you: from disk to system memory and then to texture memory, based on your current position in the database. IRIS Performer 2.1 provides for virtual memory for very large MIPmapped textures by only storing potentially visible texels in system and texture memory. Efficient management of texture memory is made possible by special capabilities of the InfiniteReality texturing hardware. This general technique is called clipmapping and is the subject of the rest of this section.

## *2.1 What is Clipmapping ?*

Clipmapping avoids the size limitations of normal MIPmaps by “clipping” the size of each texture level to a maximum size in the s, t, and r dimensions. The levels that exceed this limit are configured as image caches, which make only the most relevant texel data for that data is loaded into texture memory.

This caching technique assumes that most applications using a large texture need to view only a part of it at any one time. This assumption is even more valid when you consider how texture space varies between MIPmap levels.

### *MIPmapping Texel Requirements*

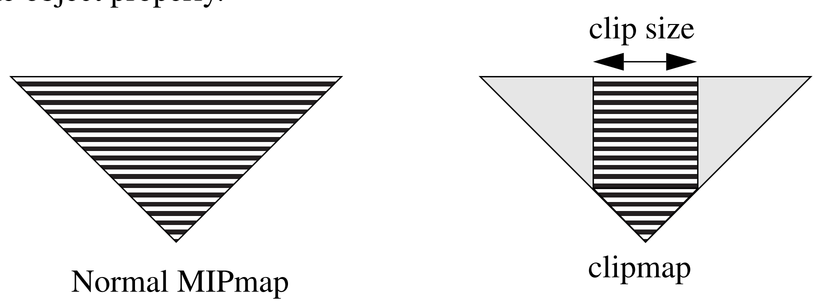
As you proceed down from MIPmap level to MIPmap level, the space requirements for a texture are quartered. Even extremely large textures have very modest space requirements at the lower levels of their mipmaps. A clipmap limits the maximum size of all the levels of MIPmap to a fixed-sized region of texels, called the *clipsize*. Every level of the clipmap is limited to this size in texture memory. If a MIPmap level is smaller than the clipsize, the entire level is stored in texture memory.

Clipping MIPmap levels to a maximum size is a technique that works well with the way large texture maps are actually used. Every texel MIPmap level represents four texels at the level above it. As a texture is viewed from farther and farther away, the number of texels needed to cover a single pixel increases, and the lower the MIPmap level used.

As a textured object gets closer to the viewer, higher and higher MIPmap levels are used, as the texel size increases on the screen. Finally, the top level is used. At this point, texels are the same size as pixels, and the number of level 0 texels visible is limited to the window resolution. If the object gets closer, the texels get bigger than the pixel size, and the amount of texels visible actually decreases.

If texture is viewed from an angle, only a part of the texture is rendered from the level 0 texels; the texels of the more distant parts of the textured object are smaller, and therefor must be rendered from texels at lower MIPmap levels.

Clipmapped textures take advantage of these properties. A clipmap with a well chosen clip size has the same appearance that the unmapped texture would. No matter what orientation and size of the textured object, the MIPmap will never need more than clipmapped texels at each level to render the object properly.




### *Moving Around*

Although there will never be a need for more than clipsize texels at each level of a clip texture, it won't always be the *same* texels. The region of texels needed will change as the view position moves relative to the textured object. clipmapping supports this requirement by allowing each clipped MIPmap level to *roam*. Roaming is the process up reloading a region of texture to support the motion of a viewer moving over a very large texture, only viewing a subset of it.

As the viewer moves, the clip region's contents are updated so that the texel region closest to the viewer in it. As the viewer moves in a given direction, the set of textures in the clip region are constantly updated. New texels appear from edge towards the direction of moment, pass through the clip region, then flow out of the region in the direction opposite the viewers movement.

To support roaming, the clip texture supports the concept of the *clip center*. The clip center specifies the location of the center of the clip region relative to the larger texture. The clipped region of each MIPmap level is updated whenever the clip center moves, to keep it in the middle of the clip region.

As the center moves, the clipped regions are updated by doing *toroidal loads*. A toroidal load assumes that each change in the clip region contents is incremental, dividing the update into the new texels that need to be loaded, the texels that are no longer valid, and the texels that are still in the clip region, but need to shift position. Toroidal loading minimizes the update bandwidth by only updating the new part of the texture region. Shifting the old pixels isn't necessary, since the coordinates of the clip region are defined to wrap around to the opposite side.

Toroidal loading updates the clip region in a regular pattern, which is based on the direction and speed of viewport movement.

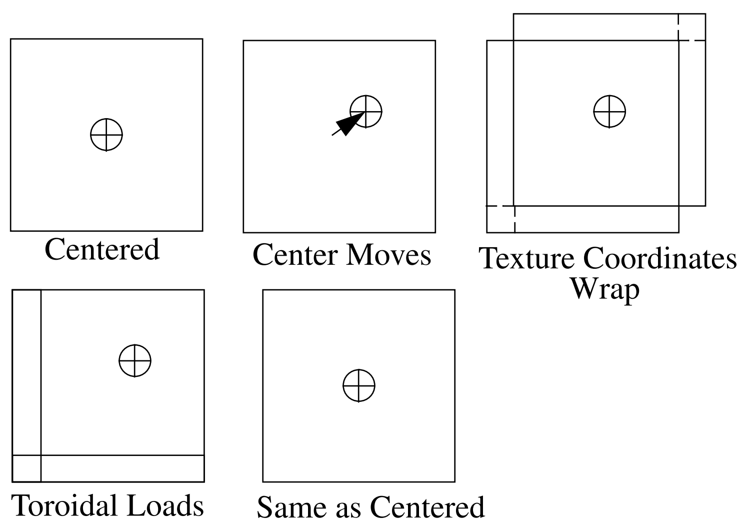



**FIGURE 1. Toroidal Updating of Texture Memory**

Toroidal loads move texels to texture memory from image caches. Image caches are a grid of equal sized tiles containing texel data. They reside in system memory. The image cache grid contains a fixed number of image tiles arranged in a rectangular array. It is a superset of a texture data for a particular MIPmap level. As a MIPmap level is roamed, texture tiles are loaded into the image cache from disk, in anticipation of their use in texture memory.

So clip-textures use a two-level caching scheme: texture data on disk is cached in system memory in image caches, and a clip-region's worth of the image cache data is cached in texture memory.

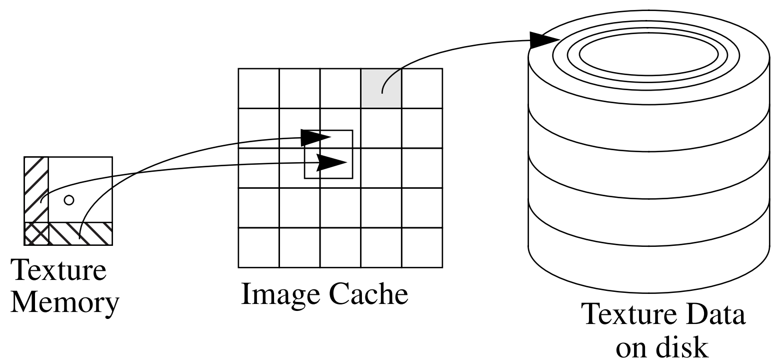



**FIGURE 2. Cliptexture Cache Hierarchy**

As the viewer roams over a clipmap, the centers of each MIPmap level moves at a different rate. This is a normal consequence of MIPmapping, but it means that image caches have to be able to roam independently. The clipmapping system adjusts the center of each level independently, and does toroidal loading to the clipmap levels in texture memory each frame. It also schedules loads of image tiles from disk into the image caches in system memory as they are needed.

### *Invalid Borders*

The clipmap system supports the concept of *invalid borders* They are regions surrounding each clip region in the clipmap. The invalid borders define an area around the perimeter of each clip region that can't be used. It essentially shrinks the usable area of each clip region. When texturing requires texels from a portion of an invalid border at a given MIPmap level, the texturing system moves down a level, and tries again. It keeps going down levels until it finds texels at the proper coordinates that isn't in an invalid region.

This works because each mipmap region contains twice the texel area of the one above, so a texel in an invalid region at one level will be half the distance from the center, and most likely still within the valid region of the clip region.

Invalid borders are used to force the use of lower levels the MIPmap. This effect can be useful for a number of reasons. It can reduce an abrupt discontinuity between MIPmap levels, if the clip region is small, since it starts the blending between levels to happen over a larger textured region. Since the invalid border can be adjusted dynamically, it can be used to improve performance when a texture is being roamed at a very high rate of speed, since it reduces the texture and system memory loading requirements at the expense of a blurrier image.

### *Clip Texture Center/Icache Center*

As the viewer moves around the scene, the clipmap center needs to be updated. This is done by calling **pfClipTextureCenter().** The center should be moved so as to keep the closest part of the texture that is in view at the highest resolution possible.

There are a number of approaches to finding a good center. They trade off quality vs. center computation overhead. A high-quality way to compute the texture center is to intersect of the view volume and the clip-texture's geometry, find edge of the texture closest to the viewer, and put the center there. This can be expensive.

Simpler approaches, such as interesting the direction of view with the texture, or, if the texture is part of the terrain (which is the most likely use of a clip-texture), to make the center the part of the texture directly below the viewer. This last method is very easy to compute, and allows changes in heading with minimum texture loading overhead. However, it doesn't work well for the case of a viewer at high altitude looking out towards the horizon.

## *2.2 Creating and using a pfClipTexture*

A pfClipTexture is created with **pfNewClipTexture()**. A pfClipTexture is a type of pfTexture, so all of the IRIS Performer routines for configuring pfTextures exist for pfClipTextures as well. However, a pfClipTexture has some additional configuration and processing requirements. A pfClipTexture updates its texture image data as needed from disk by means of a pfImageCache. Once configured, pfClipTextures and pfImageCaches will automatically update hardware texture memory as the viewing center is moved.

pfClipTextures are configured with the help of *image cache* and *clip texture configuration files.* These are ascii files containing token and value groups that describe the layout of the image data and options that should be used for the clip texture and its component image caches.

To successfully use clip textures, you must first prepare the texture data, and create the configuration files:

- If you have a single image, Create all the rsets from that image, using whatever filtering is appropriate. Levels that will be used as image caches must be tiled. The tiles can be in separate files, or grouped into multiple tiles per file, in row order.
- Create a image cache configuration file for each image cache in the clip texture. The configuration file should correctly describe the format and tiling of the texture data, the location and names of the files containing the data, and the desired size of the clip region in texture memory, and the size and layout of the cache in system memory.
- When all image cache configuration files are created, a clip texture configuration file should be made. It contains the name and location of each image cache's configuration file, the names and locations of the texture data for each level that isn't configured as an image cache, and general properties of the clip texture

Once all the setup is done, you need to create the clip texture in your performer application. You also need to be able to update the clip texture's center on a per-frame basis:

- Call **`pfdLoadClipTexture(fname)`** to create the clip texture. The fname argument is the name of the clip texture configuration file.
- For each frame, update the clip-texture's center using **`pfClipTextureCenter()`**.
- To update a clip texture with any changes you've made, call **`pfApplyClipTexture()`**

### *Image Cache Configuration File Details*

Image cache configuration files supply the following info to performer:

- Format of the texel data
- Size of the entire texture at this level
- How to find the files containing the texel data for this image cache
- Size and layout of image cache tiles in memory
- Size of the image cache that should be kept in texture memory
- A default image tile to use if one is missing

Configuration files fields are either tokens or parameter values. All fields are character strings, and all must be separated by whitespace. The list of Tokens and arguments are listed in the table below:

TABLE 1.

| Token Name | Parameters | Description |
|---|---|---|
| ic_version1.0 | no data field | Starts file: type and version |
| ext_format | string | External format of stored texels |
| int_format | string | Internal format used by graphics hw |
| img_format | string | Image format of stored texels |
| virt_size | 3 integers | Width, Height, Depth of icache in texels |
| cache_size | 2 integers | Number of rows and columns of tiles in memory |
| valid_region | 3 integers | Area x, y, z in texels in texture h/w |
| *header_offset | 1 integer | Byte offset to skip user's file header |
| *tiles_in_file | 3 integers | Image tile arrangement in each file |
| tile_size | 3 integers | Width, Height, Depth in texels of tile |
| *tile_base | file path string | Root of tile filenames |
| *tile_format | scanf string | Tile filename format |
| *num_tile_params | 1 integer | Number of parameter token arguments |
| *tile_params | string list | Format parameter tokens in order |
| *num_streams | 3 integers | Number of s, t, and r stream servers |
| *s_streams | string list | List of S stream servers |
| *t_streams | string list | List of T stream servers |
| *r_streams | string list | List of R stream servers |
| *default_tile | file path string | Use this one if tile missing |

### *Image Cache Configuration Tokens*

The `ic_version1.0` token must be the first thing in an image cache configuration file. This token identifies the file as an image cache configuration file, and what version the configuration file is formatted in.

After the version token, The parser looks for tokens, and any associated data values. In general, the tokens must be in the file in the order specified in the table above. Some tokens are optional. They are marked with an asterisk. Optional tokens are associated with a default value. If the token and value are omitted, the default value is assumed.

Tokens can have no arguments, a fixed number of arguments, or a variable number of arguments. If a token is specified, whether it is optional or not, it must be followed by a whitespace separated list of the exact number of arguments specified. If the token has a variable number of arguments, there will be a way to specify the number of them. This number must match the number of arguments.

Any time a token is expected by the parser, a comment can be substituted. A comment can't be put anywhere in the file, however. For example, if a token expects arguments, you can't place a comment between any of them; you have to place it after all of the previous tokens arguments. There are a variety of supported comment tokens; they are interchangeable. The comment tokens are **`#`**, **`//`**, **`;`**, **`comment`**, or **`rem`**.

One of the first things that must be specified in an image cache is the format of the texel data. This includes the external format (**`ext_format`**), internal format (**`int_format`**) and image format (**`img_format`**). The arguments expected by these format parameters are the ascii string names of the format's enumerates. For example, a valid external format would be **`ext_format PFTEX_FLOAT`**. Consult the pfTexture man pages for a list of the valid formats of each type.

The next set of parameters that must be specified in the image cache configuration file is its size on disk, in system memory, and in texture memory. The **`virt_size`** token requires the virtual size of the image cache. This means the dimensions, in texels, in the s, t, and r dimensions of the complete texture at this level. Since three dimensional textures aren't currently supported, the r parameter will always be 1.

An image cache's texels are organized into a set of fixed sized pieces, called tiles. Both in system memory and on disk, the texels are broken up this way. At any given time, an array of these texel tiles are cached in system memory. They are arranged as an array in system memory. If the center of the image cache nears the edge of this array, the most distant tiles are dropped out, and new tiles are read in from disk. The larger the array of tiles in system memory, the more of the complete texture is cached there, and the less likely new tiles may need to be swapped in. The benefit is offset by the cost of tying up more system memory to hold the texel tiles.

The arrangement and dimensions of tiles in system is defined for each image cache, and is set with the **`cache_size`** token. This token expects three arguments which determine the number of tiles in the s, t, and r dimensions of the grid. Since three dimensional textures aren't currently supported, the r dimension is always 1.

A subset of the texels in system memory are cached in the texture memory itself. These texels are arranged in a rectangular region. The dimensions of this region are defined by the **`valid_region`** token. It expects three arguments, the number of texels in the s, t, and r dimensions. Again, since three dimensional textures aren't supported, the r value is always 1.

The image cache configuration file allows some leeway in the arrangement of texel tiles on disk. There can be one or more tiles on each disk file, and the file itself could contain non-texel information at the beginning of the file. The tiles themselves can have user-specified dimensions. While there is some flexibility in how tiles are stored in files on disk, there are restrictions also. Any header must be the same size for every file in an image cache. The same is true for the tile size, and the number and layout of tiles in each file. If there is more than one tile in a file, the tiles must be arranged in row-major order. In other words, as you pass from the first tile to the last, the s dimension must be incrementing fastest.

The `header_offset` argument specifies the size of the file's header in bytes. This many bytes will be skipped over as a file is read. The `tiles_in_file` token requires three arguments, specifying the number of tiles in the s, t, and r dimensions. The r dimension must always be 1, since 3d textures aren't supported. The `tile_size` parameter defines the texel dimensions of each tile in s, t, and r. Again, r must be 1. Both the `header_offset` and the `tiles_in_file` tokens are optional. They default to the values 0 and 1 1 1 respectively, specifying no header and a single tile in each file.

The image cache texel data is stored in one or more files. The configuration file provides a way for Performer to find these files. The files usually have similar names, varying in a predictable way, such as by tile position in the image cache array and size of the image cache. The files themselves are grouped in on or more directories. The file name and file path information is divided into a number of groups within the configuration file. There is a scanf-style string specifying the path to find image cache files. There are a number of parameters in the string that vary as a function of the tile required, and characteristics of the image cache.

The `tile_base` token is used to define the constant part of the file path and/or filename to the image cache files. It is often the head of the file path, and can be changed in order to switch between different image databases. If the string starts with the pattern `$ENVNAME`, `${ENVNAME}` or `$(ENVNAME)`, then the value of `ENVNAME` will be assumed to be an environment variable, and expanded into the basename.

The `tile_format` token expects a scanf-style argument. The argument contains constant parts, interpersed with %d or %s parameters. The number of parameters must match the integer given with the `num_tile_params` token. The tile parameters themselves follow the `tile_params` token. The number of parameters must match both the number of parameters in `tile_format`, and the integer given by `num_tile_params`. All of these parameters are optional. The default values are given below:

```
tile_format %s%s.%d.r%03d.c%03d.raw8
num_tile_parameters 5
tile_params
PFIMAGECACHE_TILE_FILENAMEARG_STREAMSERVERNAME
PFIMAGECACHE_TILE_FILENAMEARG_CACHENAME
PFIMAGECACHE_TILE_FILENAMEARG_VSIZE_S
PFIMAGECACHE_TILE_FILENAMEARG_TILENUM_T
PFIMAGECACHE_TILE_FILENAMEARG_TILENUM_S
```

If **tile_format**, **num_tile_parameters** and **tile_params** is omitted, then **tile_base** must be supplied. The **tile_format**, **num_tile_parameters** and **tile_parameters** tokens work as a group. You must either omit them all, or supply them all, and the number of parameters must be the same for all of them.

The possible values of the image tile file name parameters is given in the table below:

TABLE 2.

| Image Tile Filename Tokens | Description |
|---|---|
| PFIMAGECACHE_TILE_FILENAMEARG_VSIZE_S | Virtual size S width |
| PFIMAGECACHE_TILE_FILENAMEARG_VSIZE_T | Virtual size T width |
| PFIMAGECACHE_TILE_FILENAMEARG_VSIZE_R | Virtual size R width |
| PFIMAGECACHE_TILE_FILENAMEARG_TILENUM_S | Tiles from origin in S |
| PFIMAGECACHE_TILE_FILENAMEARG_TILENUM_T | Tiles from origin in T |
| PFIMAGECACHE_TILE_FILENAMEARG_TILENUM_R | Tiles from origin in R |
| PFIMAGECACHE_TILE_FILENAMEARG_TILEORG_S | Texels from origin in S |
| PFIMAGECACHE_TILE_FILENAMEARG_TILEORG_T | Texels from origin in T |
| PFIMAGECACHE_TILE_FILENAMEARG_TILEORG_R | Texels from origin in R |
| PFIMAGECACHE_TILE_FILENAMEARG_STREAMSERVERNAME | From streams |
| PFIMAGECACHE_TILE_FILENAMEARG_CACHENAME | The tile_base value |
| PFIMAGECACHE_TILE_FILENAMEARG_FILENUM_S | Files from origin in S |
| PFIMAGECACHE_TILE_FILENAMEARG_FILENUM_T | Files from origin in T |
| PFIMAGECACHE_TILE_FILENAMEARG_FILENUM_R | Files from origin in R |

One of the major bottlenecks to sustained clip-texture performance is the speed of copying texels from disk to system memory. Clip-textures can be configured to maximize the bandwidth of this transfer, by distributing image tiles over multiple disks, and downloading them in parallel. The streams section of the configuration file is used for this purpose.

A stream, short for stream device, can be thought of as a separate disk that can be accessed in parallel with other disks. Each disk is mounted in a file system, and therefore has a unique filepath segment. The streams tokens allow the user to identify these stream filepath segments, and how the image tiles are distributed among them. The stream devices are arranged in a three dimensional grid, with s, t, and r dimensions, just like the image tiles are in memory. The stream device is accessed by taking the position of the tile, counting tiles from the origin in the s, t, and r directions, and generating a coordinate, modulo the number of stream devices in the corresponding s, t, and r directions. The s, t, and r values generated are used to look up the appropriate stream device. If the stream server name is part of the tile file name format string, it effects which disk is used to find the tile.

Stream servers improve bandwidth at the expense of duplicating image tiles over multiple disks. You must insure that the proper image tiles are available for any disk which is addressed by the tile's s, t, and r coordinates modulo the available number of stream servers for each of those dimensions. The stream server tokens are optional. The default value is 0 0 0 for **`num_streams`**, and no argument for **`s_streams`**, **`t_streams`**, and **`r_streams`**.

The **`num_streams`** token expects three integers; the modulo number for the s, t, and r directions. The number of stream server names for each coordinate must exactly match their corresponding values.

The **`s_streams`** token is followed by a list of filepaths. These are the names that will be indexed from the list by taking the s coordinate of the tile's position in the image cache grid, modulo the number of s stream devices. The names in the **`s_stream`** list do not have to be unique.

The **`t_streams`** and **`r_streams`** tokens work in exactly the same way, in the t and r directions, respectively.

Sometimes only a subregion of the entire clipmap is of interest to the application. This is especially true when you consider that the number of tiles in the s, t, and r directions must all be a power of two. To save space, improve performance, and make creating image caches more convenient, a default tile can be defined, and tiles of no interest can simply be omitted. If a tile can't be found, and a default tile is defined, then the default one is used in place of the missing one.

Unlike normal tiles, which are read from disk as they are needed, the default tile is loaded as part of the configuration process. The tile is named in the configuration file as the argument to the **`default_tile`** token. The argument is a filepath to the default tile. If the **`tile_base`** token has been defined, it is pre-pended to the file path, otherwise it is used as-is.

### *Clip Texture Configuration File Details*

Clip-texture configuration files supply the following info to performer:

- Format of the texel data

- Size of the top level of the texture
- The size each level should be clipped to
- The amount of border that should be invalidated at each level
- The dimensions of the smallest image cache in the texture
- How to find the image cache configuration files
- How to find the tiles consisting the levels that aren't image caches

Configuration files fields are either tokens or parameter values. All fields are character strings, and all must be separated by whitespace. The list of Tokens and arguments are listed in the table below

TABLE 3.

| Token Name | Arguments | Description |
|---|---|---|
| ct_version1.0 | no argument | must be at top of file |
| ext_format | format string | external format of stored texels |
| int_format | format string | internal format used by graphics hw |
| img_format | format string | image format of stored texels |
| virt_size | 3 integers | size of entire texture |
| clip_size | integer | size of square of texture levels in hardware |
| invalid_border | integer | width of perimeter to not use |
| *smallest_icache | 3 integers | smallest icache level dimensions |
| *icache_base | file path string | root of icache config filenames |
| *icache_format | scanf string | icache fnames: no field? list files |
| *num_icache_params | integer | number of format arguments to follow |
| *icache_params | string list | format tokens in order |
| *tile_base | file path string | root of pyramid tile filenames |
| *tile_format | scanf string | tile fnames: no field? list tile files |
| *num_tile_params | integer | number of format arguments to follow |
| *tile_params | string list | format tokens in order |
| *icache_files | list of filenames | only if icache_format is default |
| *tile_files | list of filenames | pyramid; only if tile_format default |

### *Clip Texture Configuration Tokens*

The `ct_version1.0` token must be the first thing in an clip-texture configuration file. This token identifies the file as an image cache configuration file, and what version the configuration file is formatted in.

Any time a token is expected by the parser, a comment can be substituted. A comment can't be put anywhere in the file, however. For example, if a token expects arguments, you can't place a comment between any of them; you have to place it after all of the previous tokens arguments. There are a variety of supported comment tokens; they are interchangeable. The comment tokens are **#**, **//**, **;**, **comment**, or **rem**.

One of the first things that must be specified in a clip texture is the format of the texel data. This includes the external format (**ext_format**), internal format (**int_format**) and image format (**img_format**). The arguments expected by these format parameters are the ascii string names of the format's enumerates. For example, a valid external format would be **ext_format PFTEX_INT**. Consult the pfTexture man pages for a list of the valid formats of each type.

The next group of tokens characterizes the clip texture itself. The **virt_size** token expects three integer arguments. They define the s, t, and r dimensions of the level 0 layer of the clip texture in texels. The **clip_size** token describes the size of each layer that exists in texture memory. It also takes three integers, describing the s, t, and r dimensions of the clipped region. This value is the same for all levels of a clip texture. If the image cache configuration files' **clip_size** differs from this value, the clip texture overrides it.

The **invalid_border** defines the region of each clipped level that shouldn't be used. If a texel is needed in that region, the next level down is used instead. If the invalid border is large, the system may have to go down multiple levels, or even down to the pyramidal, unclipped part of the MIPmap. The invalid border argument is a single integer, describing the width of the border in texels.

The **smallest_icache** token describes the s, t, and r dimensions of the lowest level that is described as an image cache. This parameter is needed because the unclipped, pyramidal part of the MIPmap can also be configured as image caches. This is an optional token. If it isn't included in the file, the last clipped level is considered the smallest image cache in the clip texture.

The next group of tokens describes the location of the configuration files defining the image cache levels of the clip texture. There are two methods of describing where the image cache configuration files. You can explicitly list the filenames in order with **icache_files**. These names are pre-pended to the value of **icache_base**, and matched with the icache levels in the clip texture.

The other method is to define the image cache configuration filenames with a scanf-style string containing parameter values, as is done with image caches. This is usually the preferred method. To create parameterized image cache names, you must define the **icache_format**, **num_icache_params**, and **icache_params** tokens.

The **icache_base** token is followed by the constant part of the filepath. It is often used to allow a clip texture configuration file to point to different image database containing different image caches. If the string starts with the pattern **$ENVNAME**, **${ENVNAME}** or **$(ENVNAME)**, then the value of **ENVNAME** will be assumed to be an environment variable, and expanded into the basename.

The **icache_format** token is followed by a scanf-style string describing the filepath and filename of the image cache configuration files. The argument contains constant parts, interspersed with %d or %s parameters. The number of parameters must match the integer given with the **num_icache_params** token. The tile parameters themselves follow the **icache_params** token.

The number of parameters must match both the number of parameters in **icache_format**, and the integer given by **num_icache_params**. All of these parameters are optional. If the defaults are omitted, the default configuration string is **%s%s**. The first argument is the value of **icache_base**; the second argument is the appropriate element in the **icache_files** list of filenames.

The list of available parameter tokens is given in the table below:

TABLE 4.

| Parameter Token Name | Description |
|---|---|
| PFCLIPTEX_FNAMEARG_LEVEL | Clip Texture level (top is 0) |
| PFCLIPTEX_FNAMEARG_LEVEL_SIZE | Largest value of level's virtual size |
| PFCLIPTEX_FNAMEARG_IMAGE_CACHE_BASE | Value of icache_base |
| PFCLIPTEX_FNAMEARG_TILE_BASE | Value of tile_base |

The parameter values are used to construct the name of the image cache configuration file; uniquely naming that file for each level of the clip texture.

Near the bottom of the clip texture, the size of lower levels are too small to warrant image caches. These levels are specified directly, referring to a single filename containing a single image tile for each level. The filenames for these tile files are specified in exactly the same way as the image cache configuration files are. Instead of **icache_base**, **icache_format**, **num_icache_parameters**, and **icache_parameters**, **tile_base**, **tile_format**, **num_tile_parameters**, and **tile_parameters** are used. The parameters available for use in the **tile_format** string are identical to the ones used for **icache_format**.

If image cache configuration files and/or image tiles are to be explicitly named, they are listed in order, from the top (largest) level to the bottom, using the `icache_files` and `tile_files` tokens. These tokens can only be used if the corresponding format, num_parameters, and parameter tokens aren't. The number of filenames listed after `icache_files` and `tile_files` must exactly match the number of cached and uncached levels, respectively, in the clip texture.

## *2.3 Creating Clipmap Data Files*

The process of preparing image data for use in clip textures consists of five steps:

- Converting the image data into a format that can be used by texturing hardware
- Creating all the rset levels of the MIPmap
- Breaking up texels in each level into equal size tiles
- Creating and naming the files that contain the texels in a consistent way, so that they can be referenced by the image cache and clip texture configuration files
- Creating the image cache and clip texture configuration files themselves.

Not all these steps are always necessary. It depends on the format and layout of the image data. The number of ways the image data could be stored are too varied to provide a step-by-step process to conditiong data for use in clip-textures. Instead we'll provide some information and advice to help make the process easier.

### *Formatting Image Data*

The data must be in a format that can be used in Performer textures. This means the texels must have all their color components together, which the size and type of the color components matching a format supported by Performer. Keep in mind that these texels will be loaded dyamically, on an as-needed basis, so the smaller the size of each texel, the better the performance of the clip texture. You should choose the smallest texel format that provides acceptable color quality. A good choice might be RGBA 5551, which takes up 16 bits per texel. Performer provides some tools for converting from rgb format to 5551 or 888 RGBA. They are named, *to5551* and *to888*

### *Creating the MIPmap*

The rset levels for the clip-texture can be done in the usual way, by averaging four adjacent texels to form a new texel one level down. If a higher quality filtering is desired, that can be used instead, as long as each rset is half the dimensions of the one above it. One problem in averaging down each level is the large size of the top ones. This can be handled by tiling the image first, then creating the levels. Performer provides a tool for doing this called *rsets,* which will create all the rset levels from a level 0 texture, whether it's tiled or not. The rsets program expects the texture to be in the rgb format. After the tiles are created, they can be converted to a format appropriate for texturing.

### *Tiling Data*

The data used for the image cache layers must be broken into equal-sized tiles. There is a space/performance trade-offs that affects the optimal tile size. The larger the tile, the less toroidal loads are needed to download data into texture memory. Large tiles also have a higher bandwidth when being copied from disk to system memory.

The small tiles, while less efficient, are better at load leveling, since the time it takes to load a new tile into system memory is smaller. It also means that the total size of an image cache in system memory can be less. Although more experiments need to be run, we've found that tile sizes of 512 x 512 and 1024 x 1024 give good results.

If you want to break up a rgb image into tiles, you can use the *subimg* program to do it.

### *Creating and Naming Tile and Configuration Files*

We've found it convenient to group the files containing the texel data in the same directories with the corresponding image cache and clip texture configuration files. This way the entire clip texture can be referred to by setting an environment variable named in the **`icache_base`** or **`tile_base`** arguments. We also found that it is more intuitive to use the size of a level, rather than the level number. This is especially true if your are experimenting with using different sized clip textures.

Setting up stream servers can be an important way to improve clipmap performance. We often give the disks a set of paths from a single directory, then add that section to the filename format string. That way, we can still point to that directory with an environment variable in the base name.

If opening the file has a high overhead, or the data already came that way, you can have multiple tiles in a single file. The header offset can be used to skip over data contained in IL files.

### *Creating Configuration Files*

Unfortunately, this process has not been automated. Creating working configuration files requires a two prong approach:

- Keep them simple
- Work bottom-up

The best approach is to create each image cache configuration file first, and test it with a demo program, such as *pguide/libpr/C/icache.* When you've got all the image caches working, combine them by creating a clip texture configuration file. You can test this file by running *pguide/libpr/C/cliptex*.

We've found that parameterized naming of the image caches and tile files works the best. If you've named your files consistently, this can be easy. If things don't work, you can fall back and name your file explicitly as a sanity check. Read the error messages carefully; they try to point out where in the configuration file the parser found problems.

A number of example configuration files and clip textures are available on the performer release. Working from one of them can save a lot of time. Some places to look are:

- pguide/libpr/C/config_files
- data/clipdata/hunter
- data/clipdata/moffett
- data/asddata

Finally, take a look at the .im loader. It has support for loading a clip texture, and updating its center as a function of viewposition.

### *Order of Operations*

You don't have to follow the order of conversions laid out above. Many tools are available for rgb files, and it might make sense to keep things in rgb format while creating rsets and creating image tiles.

For examples of performing all of these operations together, look at the Makefiles in **`data/clipdata/hunter`** and **`data/clipdata/moffett`**. These files have targets that will cause the rsets to be made and tiles, and the data converted into the proper format.

## *3 Dynamic Terrain with the Active Surface Definition Library*

Infinite Reality graphics was designed based on the notion that there is an entire class of customers that want to visualize data that is not limited in extent. These customers have enormous volumes of geometry and textures which they would like to manipulate in real time. Many of these types of applications exist in large scale terrain visualization. In this type of visualization, the topology of the earth is provided at very high resolution for areas possibly covering our entire planet. It would be impossible using current technology to render scenes of this complexity and scale in real time without making some key assumptions. One such assumption is that only a finite amount of data can be visualized in any one frame. Further, because of the use of perspective in the visualization, certain triangles can influence the final visual output to a much greater extent than others. Based on this, one can envision adaptively determining what geometry is necessary to render a scene from a given perspective view and that this geometry can be modified continuously on a frame by frame basis. In this manner a finite amount of data can be used to create an image that is indistinguishable from an image created using all of the available data

The Active Surface Definition library provides a flexible framework to support real-time adaptive morphing and meshing for large geometric databases. pfTerrain is the first part of ASD. The terrain database is organized as a tree structure with each node represents a triangular face in the terrain. All the nodes on one level of the tree makes up a complete meshing of the terrain. The faces are on the same LOD. There are fixed rules on how parent and child nodes can morph into each other. pfTerrain selects the appropriate triangles from the tree at run-time using *morphing evaluation functions* provided either as Performer default functions or user callback functions. Every triangle face in the tree has up to 4 child faces. These triangles forms an *active mesh* for the provided *viewing frustum*. A morphing weight is computed for every edge in the active mesh; so morphing is done on a per-edge, per-triangle basis. As a result, the terrain is modeled as combinations of triangles from different LODs and there are no gaps, holes or edge boundaries between those LODs. For example, if the evaluation function is a purely range based function, i.e. an edge should morph if it is within certain distance to the view point, then pfTerrain adaptively meshes the terrain as view point moves, with areas closer to the view point being more finely tessellated than areas further away.

### *3.1 pfTerrain Data Structure*

The data structure consists of vertex input and face input. The vertex input is a list of arrays describing the geometric attributes of all the vertices in the terrain. The face input is an array that represents a recursive subdivision of triangular faces.

Every triangle face has up to 4 child faces. The 4 child faces are formed by connecting the 3 vertices of the triangle and 3 new vertices each of which is associated with an edge of the triangle. The edge morphs into 2 edges and the point on the edge that breaks the edge into 2 edges is called the *reference point*. The reference point can be any where on the edge. This flexible requirements makes cultural feature fitting more efficient. There is a dual relationship between edge and vertex, i.e. each edge has an unique vertex act as its final morphing destination. Nodes in the tree are stored in an array of pfsFace structure and each face is referred to by the index which is called the face id.

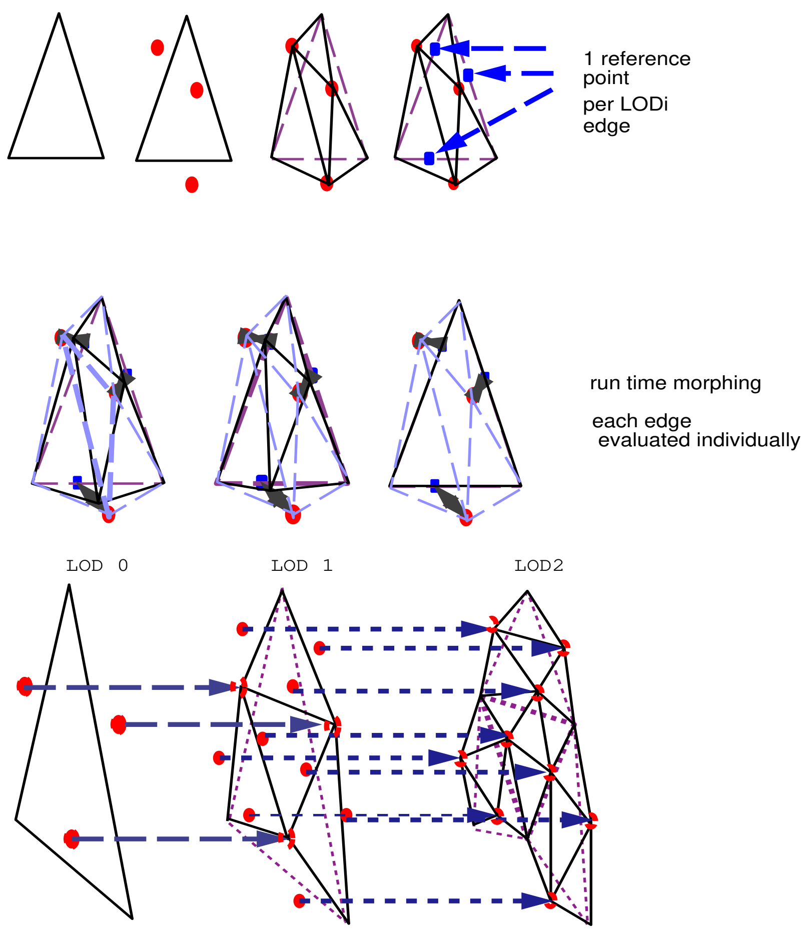



**FIGURE 3. Tessellation of Triangles between Levels of Detail in ASD**

Vertices are referenced by index. The entries in pfsFace are idicies into vertex arrays. Each vertex *i* in the database has a list of 2 values attributes associated with it. For example, coordinate is described this way: *v0[i]* is the final position of the vertex, and *vd[i]* is the vector which is the difference between the start and final position. The start position is the reference point on an edge. The id for a reference point is the same as the id for its final destination point. The index *i* is the vertex Id. It may appear several times in the pfsFace array, sometimes as vertid and sometimes as splitid. Normal, color, and texture coordinates are input the same way. It is very important that each vertex and face must have an unique vertex id and face id

The faces on the first level of LOD must be listed at the beginning of face array. Other faces can be listed in any order. We recommend listing faces on the same level of LOD together.

For example, this is the code to set up some terrain without normals:

```
terrain = pfNewTerrain(pfGetSharedArena());
pfTerrainAttr(terrain, PFTERRAIN_LODS, mesh->numlods,
        (float *)mesh->lods, NULL);
    pfTerrainAttr(terrain, PFTERRAIN_COORDS, mesh->numverts,
        (float *)mesh->v0, (float *)mesh->vd);
    pfTerrainAttr(terrain, PFTERRAIN_FACES, mesh->numfaces,
        (float *)mesh->faces, NULL);
    pfTerrainFaceLevel0(terrain, mesh->numfaces0);
```

where mesh is an input data storage. If normal attribute is defined, then normals will also be linearly interpolated when the vertices are morphed. We do not normalize the morphing normals.

## *3.2 pfTerrain Evaluation*

pfTerrain is an libpr object. The libpf part of pfTerrain will be developed in the next release of Performer. **pfTerrainEvaluateMesh()** generates pfGeoSets that stores the active mesh as tri-strips (in Performer 2.1, we only generate one geoset). **pfTerrainPtope()** and **pfTerrainEyePoint()** defines the culling frustum and view point for the current evaluation. Since the pfTerrain LODs are very fine grained, we can do very tight culling as we evaluate the database.

The morphing function is called *morphing evaluation function* and is executed on the edges. Performer provides a default evaluation function to calculate the morph weight of edges in the database. The function is purely distance based. When the user chooses to use this function, a set of LOD switch ranges needs to be entered at set-up time and can be modified at any time before **pfTerrainEvaluateMesh**(). If the user decides to use a different evaluation function, it can be defined with **pfTerrainEvalFunc()**.

The **pfTerrainEvaluateMesh()** can be processed in a pipelined fashion. Each stage processes $x$ percent of the evaluation. The percentage can be adjusted with **pfTerrainLoadRatio()**. The file pfuTerrain.c in sample code asdfly provides a couple of ways to hook the output of **pfTerrainEvaluateMesh()** into the regular Performer scene graph: **terrainNode()** and **terrainMultNode()**.

## *3.3 pfTerrain Data Structure Generation*

The data structures needs to be generated before any pfTerrain visualization. Look at the terrain.sho file in asdfly that is shipped with Performer 2.1

for some suggestions how to select vertices for each level of LODs and their associated referent points. Look at pfdBuildASD.c in libpfdu for a very straightforward example on how to construct the data structures.

### *3.4 pfTerrain features and future directions*

pfTerrain in Performer2.1 is a draft for the basic frame work of ASD. We have defined the data structures. Triangles at different LOD levels are matched together and the details are changed adaptively and continuously. There won't be any gaps or missing holes in the terrain. Flexible evaluation functions are supported in pfTerrain. User can also mark any particular triangles to be rendered as holes at any time. Cultural features can be switched in at the time when the terrain underneath can be rendered as cutouts. Vertices can also be assigned a different morph weight as run time or to be morphed to its final destination at any time.

In the future, we will define more elaborate data structures to better support the integration of cultural features and terrain. Paging and multiprocessing will also be added.

## *4 Load Management with Dynamic Channel Resolution and Pipeline Statistics*

IRIS Performer offers sophisticated techniques for load management so that a constant frame rate may be maintained, even with complex databases. Objects may have levels of detail (managed by the pfLOD node) that are chosen based on distance from the eye point. There are stress management functions defined on pfChannels to then apply scaling factors to the current eyepoint-distance when evaluating LODs so that lower levels of detail will be used when the desired frame rate is not being achieved. As of the IRIS Performer 2.0 release, the use of pfLODStates allow for more per-object control over the stress management system. Objects may have priority classes so that some objects will respond to overload faster than others. Additionally, individual objects may have specific settings for control of their LOD transition behavior, including the size of the transition range and whether or not they should be using fading (where both levels of detail are rendered and blended together as the object fades from one level of detail to another). Finally, Performer provides an extensive set of timing statistics for all significant operations that are accessible by user programs for the additional tracking and controlling of system load.

In IRIS Performer 21., with special hardware support from InfiniteReality, there are two very important improvements to the load management system: direct graphics pipeline timing statistics and dynamic viewport resolution through video resizing.

### *4.1 Graphics Pipeline Statistics*

Traditionally, the timing of rendering on a machine with a graphics pipeline is a tricky business. If time is measured on the host when drawing is started and stopped, it may be very optimistic because the graphics pipeline is busy long after the host has sent its last command. To reduce this problem, the graphics pipeline can be flushed before the final host time is taken. This is reasonable to do once per frame on a graphics pipeline at the end of drawing for the current frame and is the mechanism that IRIS Performer has used before InfiniteReality. However, this machanism does not allow for the accurate timing of rendering to multiple channels on a single graphics pipeline; a graphics pipeline flush for each channel would be far too expensive. Furthermore, the final measurement with a flush ends up being fairly pessimistic since the time for the detection of the flush completion is included in the rendering measurement. InfiniteReality offers a solution to this problem with graphics pipeline timing statistics. The graphics pipeline can now measure the time between specified time stamps, allowing IRIS Performer to accurately measure the rendering time of each pfChannel of a pfPipe. This allows for much more effective load management for both fill-limited and mutli-channel applications.

Graphics pipeline statistics are not enabled by default in IRIS Performer 2.1, but can be explicitly enabled with

```
pfFStatsClass(PFSTATSHW_ENGFXPIPE_TIMES,PFSTATS_ON);
```

The perfly demonstration program has a “Pipe” selection in the statistics GUI button. pfChannels will automatically receive the improved timing information when graphics pipeline statistics are enabled.

### *4.2 Dynamic Resolution through Video Resizing*

Management of load through object level of detail is important for tuning a database to achieve a desired framerate. However, if an overload condition occurs, object LOD is often not sufficient. This is partly because overload is often caused by the number of pixels rendered in a given frame and objects of different levels of detail may have different numbers of polygons but will likely still touch the same number of pixels.

InfiniteReality provides a powerful solution to this problem through dynamically resizable video-output with bilinear interpolation. On a per-frame bases, an application can specify the size of the area to which it will render and InfiniteReality will rescale the render area to the full normal output size, applying bilinear interpolation. This scaling is done as part of the normal video scan-out operation and has no performance penalty!

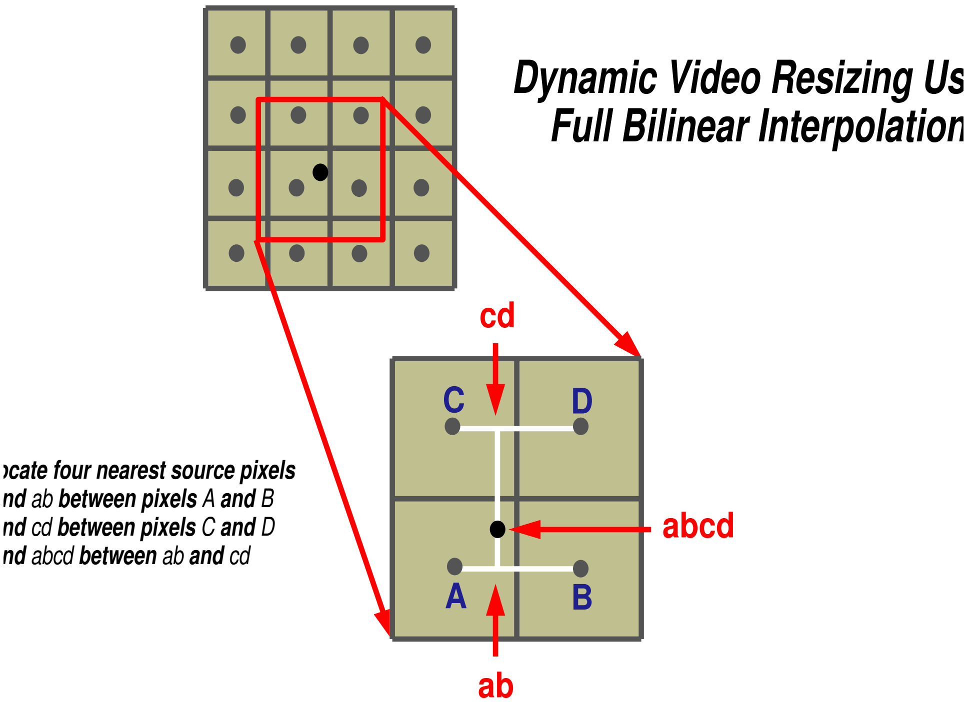



**FIGURE 4. DVR Uses Full Bilinear Interpolation**

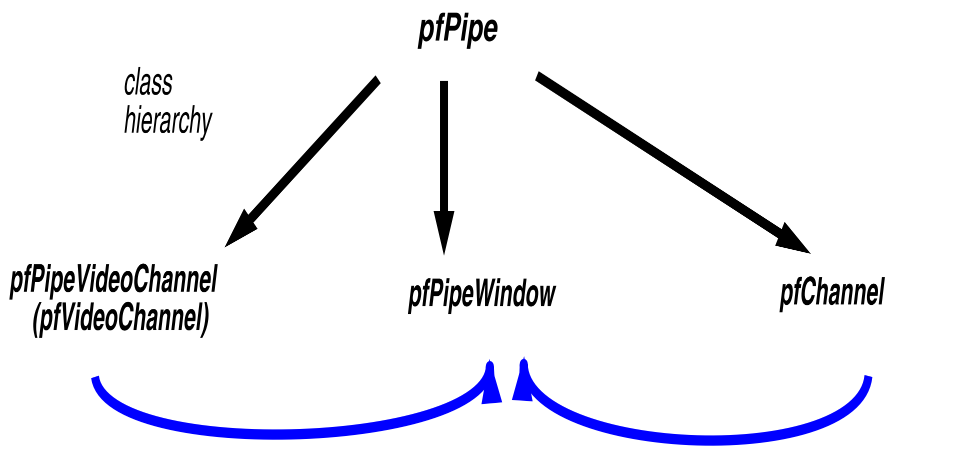
IRIS Performer2.1 provides direct user control over InfiniteReality video channels, as well as the option for automatic resizing based on load. This functionality is provided through two new classes: the libpr pfVideoChannel and the libpf pfPipeVideoChannel.

### *pfVideoChannel*

The pfVideoChannel uses the XSGIvc extensions to X (as of IRIX 6.2) to manage a system video channel. Basic information about the channel can be queried directly through the pfVideoChannel and the actual Xsgivc video channel handle can be gotten directly with **pfGetVChanInfo()** . pfVideoChannels support the resizing of video channels based on channel area in number of pixels, and allow direct specification of render area sizes. The */usr/share/Performer/src/pguide/libpr/C/{queryvchan.c, vchan.c}* sample programs demonstrate this functionality.

### *pfPipeVideoChannel*

pfPipeVideoChannels are built on top of pfVideoChannels and provide load management functionality, multiprocessed control of pfVideoChannels, and the glue between pfPipes, pfPipeWindows, pfChannels, and a system video channel. A pfPipeVideoChannel is created with **pfNewPVChan()** and as with the pfPipeWindow, it is created for a specified pfPipe that cannot be changed. A pfPipeVideoChannel is then assigned to a pfPipeWindow. pfPipeWindows are assumed to cover the full surface of their pfPipeVideoChannels. pfChannels are mapped to pfPipeVideoChannels through their pfPipeWindow with **pfChanPWinPVChanIndex()**. All pfPipeWindows have a default pfPipeVideoChannel. Upon startup, if no system video channel is selected, the pfPipeVideoChannel will connect to the first active video channel for the X display of the pfPipe. By default, all pfChannels are assigned to the first pfPipeVideoChannel of their pfPipeWindow. Figure 5 shows where pfPipeVideoChannels fit into the basic hierarchy under a pfPipe.




```
pfPipe *pipe = pfGetPipe(0);
pfPipeVideoChannel *pvchan = pfNewPVChan(pip
pfPipeWindow *pwin = pfNewPWin(pipe);
pfChannel *chan = pfNewChan(pipe);

pfChanPWin(chan, pwin);
pfPWinPVChan(pwin, 0, pvchan);
pfChanPWinPVChanIndex(chan, 0);
```

**FIGURE 5. pfPipeVideoChannels in the Performer Class Hierarchy**

The pfPipeVideoChannel load management is done employing dynamic video resizing (DVR). By default, no load management or video channel rescaling is done by the pfPipeVideoChannel. Automatic load management is enabled with

```
pfPVChanDVRMode(pvchan, PFPVC_DVR_AUTO);
```

in the application process. When using automatic DVR, the specification of a stress filter is recommended. The stress filter allows the description of a quiet zone in which no rescaling will be done and to which rescaling aims to hit when stress is encountered. A video channel is considered to be stressed when the current load is outside (above or below) this quite zone. Additionally, the response to stress may include a scaling factor to control the growth of stress and a scaling factor to apply to overall pfPipe stress. The following is an example specification of pfPipeVideoChannel stress from the perfly demonstration program:

```
pfPVChanStressFilter(pvchan, frac, 0.8f, 0.95f, 1.5f, 2.0f,
                             10.0f);
```

However, the specification of the stress filter really needs to be tuned to the current application and database.

The control of pfPipeVideoChannel resizing may also be done manually. The pfChannel DVR mode needs to be set to DVR_MANUAL:

```
pfPVChanDVRMode(pvchan, PFPVC_DVR_MANUAL);
```

The requested size for the pfPipeVideoChannel should be specified in the application process after **pfSync()** and before **pfF rame()** so that the size can effect the immediate next draw. Due to hardware limitations, the specification of the video channel size cannot be arbitrary. On InfiniteReality, the origin and size of a video channel must both be an integral multiple of four. IRIS Performer will automatically set the nearest legal size; if the exact size is needed by the application, it can be queried. The following is an excerpt from the perfly demonstration program for controlling manual resizing (selected with the ‘k’ key).

```
pfSync();
pfPVChanOutputSize(pvchan, &ViewState->dvrXSize, &ViewState-
                             >dvrYSize);
pfFrame();
pfGetPVChanOutputSize(pvchan, &ViewState->dvrXSize,
                             &ViewState->dvrYSize);
```

Note that in perfly, the ‘k’ key will cycle through the DVR modes: first manual DVR, then automatic, and then a return to no resizing.