Pixie Reference Manual

Okan Arikan

February 25, 2003

Contents

Chapter 1

User's Guide

1.1 Running Pixie

1.1.1 From the Binary Distribution

When you open the archive, everything should be under the directory `Pixie-X.Y.Z` where X.Y.Z is the version number. This directory is your `PIXIEHOME`. You may want to set this environment variable to the directory that you extracted so that Pixie can find the default display drivers and shaders. In addition, you may want to add `Pixie-X.Y.Z/bin` into your `PATH` environment variable for easy access. On UNIX platforms, you should also add `Pixie-X.Y.Z/lib` into your `LD_LIBRARY_PATH` variable.

1.1.2 From the Source Distribution

When you extract the Pixie source distribution, everything should be under a single directory `Pixie-X.Y.Z` where X.Y.Z is the version number. You should then compile and create a binary distribution:

Windows

1. Open `Pixie-X.Y.Z/Pixie.dsw`
2. Go to Build->Batch build
3. Select the programs you need to build
4. Hit OK
5. After the compilation is finished, execute `Pixie-X.Y.Z/makeinst.bat`

Unix

1. Under `Pixie-X.Y.Z`
2. Type `./makeunix`
3. Type `./configure --path=Pixie`
4. Type `make`

5. Type `make inst`

You should now have `Pixie-X.Y.Z/Pixie` with the binary distribution in it. You can copy this directory wherever you want. That directory is your `PIXIEHOME` and contains the renderer. You can now do the environment variable changes explained in the previous section to get it ready.

To be able to compile Pixie, you will need `libtiff` which you can download from `http://www.libtiff.org` .

1.2 Hiders

Pixie can generate images using raytracing or scan line rendering. As it is often the case, scan line rendering is much faster than raytracing. However, such methods may not support accurate reflections/shadows. Since Pixie supports both types of algorithms, you can combine the best of the two worlds: speed of scan-line rendering and the accuracy of raytracing.

A hider is the section of the renderer that is responsible for creating the final image. Essentially, every hider implements a different rendering algorithm. You can switch between defined hiders using `Hider` command.

Pixie defines the following hiders:

1. **Raytrace (`Hider "raytrace"`):** As the name implies, this hider creates the final image using raytracing. This involves shooting bunch of rays for every pixel (defined by `PixelSamples`) and then filtering and summing the color of every ray (defined by `PixelFilter`). This hider also supports adaptive supersampling which means the renderer will only shoot rays when something interesting is going on (i.e. there is high frequency detail). The maximum allowable contrast can be specified using `PixelVariance` command.
2. **Stochastic/Hidden (`Hider "stochastic"`):** This hider creates the final image using scan-line techniques very similar to Pixar's Reyes architecture [1]. Every specified primitive is split into smaller primitives and deferred until needed. If the projected size of a subdivided primitive is small enough, a regular grid is sampled on the patch and the polygons in this grid are rendered using scan-line methods. Notice that a raytracer may need to keep the entire scene geometry in the memory in case a future ray can intersect them. On the other hand, the deferred and render-and-forget feature of this rendering algorithm allows it to keep a very small memory footprint.
3. **Zbuffer (`Hider "zbuffer"`):** This hider is a stripped down version of **Stochastic**. It does not support motion blur, depth of field or transparency. If your scene does not involve these effects, this hider can generate an equal quality output with the Stochastic.
4. **Opengl (`Hider "opengl"`):** This is exactly the same with Zbuffer. However, the final stage of the rendering is implemented with OpenGl. The potential hardware acceleration makes this hider much faster than any other hider. If your computer does not have a hardware implemented opengl driver, **Zbuffer** is faster. [2].

[1] The RenderMan (R) Interface Procedures and RIB Protocol are: Copyright 1988, 1989, Pixar. All rights reserved. RenderMan (R) is a registered trademark of Pixar.

[2] OpenGL hider is only available on Windows systems for the time being

	TSM	J	T	OC	AOS	SS	ASS	MB	DOF
stochastic	✓	✓	✓	✓	✓	✓		✓	✓
raytrace		✓	✓	✓	✓	✓	✓	✓	✓
zbuffer				✓		✓			
opengl						✓			

TSM: Transparency shadow maps
J: Jittered sampling
T: Transparency
OC: Occlusion culling
AOS: Arbitrary output samples (output types other than rgbaz)
SS: Super sampling
ASS: Adaptive super sampling
MB: Motion blur
DOF: Depth of field

Table 1.1: Hider capabilities

If you notice, `OpenGL` hider is implemented as a module. That means if you want to use it, `opengl.dll` must be in your procedural path (check `"searchpath" "procedural" option`).

1.3 Writing Custom Display Drivers

A display driver is a module [dll/so] that handles the image output. The renderer will essentially send the rendered tiles of the image to a display driver and then the driver can display/save it. To write a display driver, you need to create a [dll/so] that exports the following three functions:

1. ```
   void *displayStart( const char *name,
                       int width,
                       int height,
                       int numSamples,
                       const char *samples,
                       TDisplayParameterFunction parameterFunc)
   ```

   This function is called right after `WorldBegin` and should initialize your display driver. Most of the parameters have obvious meanings: `name` is the display name given by `Display`. `width, height` give the image size given by `Format`. `numSamples, samples` give the number of samples per pixel and the textual definition of the samples given in `Display`. For example `Display "ri" "rgb" "file"` will call the `displayStart` function of the display driver **file** with `numSamples=3, samples="rgb"`. The last parameter `parameterFunc` can be used to fetch individual options or parameters given in the `Display`. The prototype for this function is defined in `dsply.h` in the include directory. The return value from this function is a transparent display handle. This handle will be used to identify the display in the subsequent calls. A return value of `NULL` indicates an error. In this case, no data about this image will be sent to the driver and thus the functions below will not be called.
   ```

Name	Type	Description
quantize	float[4]	The quantization (min,max,zero,one)
dither	float	The dithering amount
near	float	The near clipping plane
far	float	The far clipping plane
Nl	float[16]	Column major world to camera transformation matrix
Np	float[16]	Column major world to NDC transformation matrix
gamma	float	The gamma correction amount
gain	float	The gain amount
Software	char *	Always "Pixie"

Table 1.2: Default display driver parameters

2. `int displayData(void *image,`
   `int x,`
   `int y,`
   `int w,`
   `int h,`
   `float *data)`

   This function is called to deliver data to the display driver. `image` is the transparent display handle returned by `displayStart`. `x,y` give the coordinates of the top left corner of the tile and `w,h` give the width and height of the tile. The renderer will always cover every pixel once and only once. However, the renderer makes no guarantees about the order or the size of the tiles (i.e. the renderer may change the tile size and send tiles out of order). The raw data is given in `data` where every float gives a sample (i.e. first `numSamples` floats give the first pixel). The data is not quantized or clamped to (0,1) before calling this function. It is the receiver's responsibility to do that. This function must return 1 on success and 0 on failure. On failure, the renderer will not send any more data and will not call the function below (So you should clean whatever data you allocated before returning).

3. `void displayFinish( void *image)`

   This function is called after `WorldEnd` to signal the driver that the rendering is complete and all the pixels have been sent. The `image` is the transparent display handle returned by `displayStart`. This function will **not** be called if `displayStart` returns `NULL` or `displayData` returns 0.

Notice that `displayStart` is called in `WorldBegin` and `displayFinish` is called in `WorldEnd`. The implementation of the default display drivers **file** and **framebuffer** are provided in source distribution directory. The parameters in table 1.2 are always defined and the function `parameterFunc` will return them. A display driver must implement all three functions to be accepted by the renderer.

Pixie supports opening multiple displays with any combination of global or user defined variable output types. So you can define a variable before the `"Display"` statement and use it as the output type. Note that if your shaders do not set this global output variable, the result will be garbage.

You can also pass different quantization constants to different displays by specifying `"custom"` as the quantization type in `"Quantize"` call (i.e., `Quantize "custom" 0 0 0 0 0`). This will cause the last display to use this quantization settings.

1.4 Non-Standard Options

This section explains the non-standard Pixie options. The values indicated in the commands are the default values.

1.4.1 Seachpath Options

These options control the directories where the renderer searches for external resources. The different directories can be separated by a `":"` character. The directories are searched in the order they are specified in the value. Environment variables can be inserted by enclosing them between `"$"` characters. The previous value of this item can be inserted by `"@"` character.

```
Option "searchpath" "string archive" ".:$PIXIEHOME$/models:$RIBS$"
```

Set the directories where `.rib` files will be searched.

```
Option "searchpath" "string procedural" ".:$PIXIEHOME$/lib:$PROCEDURALS$"
```

Sets the directories for the modules. Modules include DSO shaders and display drivers. In practice, any dynamically loaded object `[dll/so]` must be located in these directories.

```
Option "searchpath" "string textures" ".:$PIXIEHOME$/textures:$TEXTURES$"
```

Sets the directories where the texture files are located.

```
Option "searchpath" "string shaders" ".:$PIXIEHOME$/shaders:$SHADERS$"
```

Sets the directories where the shader files are located.

1.4.2 Limits Options

These options control the renderer specific limits.

```
Option "limit" "int gridsize" "512"
```

This number controls the maximum number of points to be shaded at a time. The bigger numbers decrease the rendering time but consume more memory.

```
Option "limit" "int maxrecursion" "5"
```

This number controls the maximum traytracing recursion depth.

```
Option "limit" "int texturememory" "2000"
```

This controls the maximum texture amount to keep in the memory. This number is specified in kilobytes.

```
Option "limit" "int shadercache" "1000"
```

This controls the size of the memory allocated as shader cache. Bigger numbers decrease the rendering time in scenes with complicated shaders but increase the memory usage. This number is specified in kilobytes.

```
Option "limit" "int hierarchydepth" "40"
```

The maximum depth of the raytracing hierarchy.

```
Option "limit" "int hierarchyobjects" "10"
```

The maximum number of raytracing objects allowable per leaf.

```
Option "limit" "int bucketsize[2]" "[32 32]"
```

The image is rendered in buckets of this size. Bigger numbers decrease the rendering time but increase the memory usage.

```
Option "limit" "int eyesplits" "10"
```
The maximum number of splits before we give up on a primitive that spans the image plane.

1.4.3 Hider Options

Options in this category control the hider behavior. For compatibility purposes, you can pass these options directly to the renderer in the `Hider` command. (i.e., `Hider "hidden" "jitter" [1]`).

```
Option "hider" "float jitter" "0.5"
```
This option controls the stochastic jittering amount for each sample. 0 means no jittering, 1 means jitter the size of a sample.

```
Option "hider" "int falsecolor" "0"
```
For the `raytracer` hider, you can set this option to 1 to generate an image of the computation time used for each pixel. The whiter a pixel is, more samples for that pixel is computed.

1.4.4 IO Options

Options in this category control the renderer input/output to the console and compatibility features. These options can be used for debugging.

```
Option "io" "int maskresolution" "1"
```
If this value is 0, the renderer will print a message saying where each file that it opens are found.

```
Option "io" "int maskprintf" "1"
```
If this value is 0, the renderer will print the `printf` messages in the shaders.

```
Option "io" "int maskstats" "1"
```
If this value is 0, the renderer will print the statistics at the end of each frame.

```
Option "io" "int maskprogress" "1"
```
If this value is 0, the renderer will display a progress bar.

1.5 Non-Standard Attributes

The parameters defined in this section are Pixie specific attributes and must be set by `Attribute` command.

1.5.1 Dice Attributes

The attributes in this category control how a primitive will be tesselated for raytracing or scan-line rendering.

```
Attribute "dice" "int minsubdivision" "2"
```
This is the minimum number of subdivisions that have to be performed to triangulate the surface of a primitive for raytracing. If the primitive has a very large curvature, you may want to increase this number (although default value is largely sufficient).

```
Attribute "dice" "int maxsubdivision" "10"
```
This is the maximum number of subdivisions to make in tesselating a surface for raytracing.

Category	Option	Type	Default
searchpath	archive	string	".:$PIXIEHOME$/models:$RIBS$"
searchpath	procedural	string	".:$PIXIEHOME$/lib:$PROCEDURALS$"
searchpath	textures	string	".:$PIXIEHOME$/textures:$TEXTURES$"
searchpath	shaders	string	".:$PIXIEHOME$/shaders:$SHADERS$"
limit	gridsize	int	512
limit	maxrecursion	int	5
limit	texturememory	int	2000
limit	shadercache	int	1000
limit	hierarchydepth	int	40
limit	hierarchyobjects	int	10
limit	bucketsize	int[2]	[32 32]
limit	eyesplits	int	10
hider	jitter	float	0.5
hider	falsecolor	int	0
io	maskresolution	int	1
io	maskprintf	int	1
io	maskstats	int	1
io	maskprogress	int	1

Table 1.3: Non-standard options summary

```
Attribute "dice" "int numprobes" "3"
```

During the scan line rendering, the renderer needs to estimate the bounding box of a piece of a primitive. Pixie does this by sampling points on the surface and then extending the bounding of these points. `"numprobes"` controls the number of samples in u and v directions to take in order to estimate the bound. Notice that this is just an estimate and the renderer may underestimate the correct bound. The right way to do this is to actually subdivide the surface. But this consumes lost of computation and memory.

```
Attribute "dice" "int minsplits" "2"
```

This controls the minimum number of times a surface is split before dropping into the reyes pipeline. Since the pixie estimates the surface bounds by sampling, this number must be greater than 0 for some primitives.

```
Attribute "dice" "int boundexpand" "0.5"
```

This is the factor by which Pixie will overestimate the bound of the surface pieces. The bounding box computed by point sampling on the surface will be expanded by this factor.

```
Attribute "dice" "int maxsilhuetteedge" "10"
```

This value gives the maximum projected size (in pixels) of an edge along the silhuette for tesselating for raytracing.

```
Attribute "dice" "int maxborderedge" "10"
```

This controls the maximum projected size (in pixels) of an edge along the border of a surface (u=0,v=0,u=1,v=1).

```
Attribute "dice" "int maxedgelength" "50"
```

This is the maximum projected size (in pixels) of an edge for tesselating for raytracing.

`Attribute "dice" "int maxinvisibleedgelength" "0.1"`

This controls the maximum length of an edge that is out of the view frustum in the camera space for tesselating for raytracing.

`Attribute "dice" "int binary" "0"`

If this value is 1, when sampling a grid on a surface piece, create a power of two edges on the boundaries. This attribute is used to help the patch cracking problem.

1.5.2 Displacementbound Attributes

These attributes are used to tell renderer how much a displacement shader actually displaces the surface. This information is vital for accurately tesselating and rendering surfaces.

`Attribute "displacement" "float sphere" "0"`

This is the amount in the displacement coordinate system by which the displacement shader can move the surface.

`Attribute "displacement" "string coordinatesystem" "current"`

This command sets the coordinate system that the displacement bound is expressed in.

1.5.3 Visibility Attributes

The attributes in this class control the visibility behaviors of objects.

`Attribute "visibility" "int camera" "1"`

If 1, the object is visible to camera rays (and to the scanline renderer).

`Attribute "visibility" "int trace" "0"`

If 1, the object is visible to rays created by the `trace` shading language command.

`Attribute "visibility" "string transmission" "opaque"`

This value controls the shadowing behaivor of the object. If this value is `"transparent"`, the object is not visible to the transmission rays and is considered complately transparent. If it is `"opaque"`, the object is considered opaque. If it is `"shader"`, the surface shader is executed to find out the opacity. If it is `"Os"`, the opacity of the surface is copied from the `Os` attribute.

1.5.4 Trace Attributes

These attributes control the raytracer behavior.

`Attribute "trace" "int displacements" "0"`

If this value is 1, then the surface is actually displaced during the tesselation for raytracing purposes. This can be quite expensive.

`Attribute "trace" "int numarealightsamples" "1"`

This value controls the number of area light source samples to take.

1.5.5 Object Attributes

These attributes control features that can not be classified into any other category above.

`Attribute "object" "string name" ""`

This attribute can be used to attach a name the object. When reporting errors, this name will also be printed.

Category	Attribute	Type	Default
dice	minsubdivision	int	2
dice	maxsubdivision	int	10
dice	numprobes	int	3
dice	minsplits	int	2
dice	boundexpand	float	0.5
dice	maxsilhuetteedge	float	10
dice	maxborderedge	float	10
dice	minsubdivision	float	2
dice	maxedgelength	float	50
dice	binary	int	0
displacement	sphere	float	0
displacement	coordinatesystem	string	"current"
visibility	camera	int	1
visibility	trace	int	0
visibility	transmission	string	"opaque"
trace	displacements	int	0
trace	numarealightsamples	int	1
object	name	string	""

Table 1.4: Non-standard attributes summary

1.6 Area Light Sources

Pixie supports true area light sources. That means, the area light source shader will be executed on the surface of the area light source and thus can use any geometry variable related to the attached surface. The attribute `"numarealightsamples"` control the number of points **for each individual primitive** that Pixie will sample on the light and execute the light shader. Each of these sampling points are treated as individual light sources in the `diffuse,specular` calls and inside the `illumination` statement. Note that each primitive will be sampled **separately**. So if there are 10 primitives inside a light source and the number of area light samples is 5, 50 samples will be collected/shaded. I know this sucks. The theoretically correct way of operation is to sample a fixed number directions uniformly sampled toward the light source from the point being shaded. However, this sampling is quite difficult. I have some ideas about this though, stay tuned.

Area light source tip: avoid using complicated area light sources :).

1.7 Occlusion Culling

Pixie supports occlusion culling during the scan-line rendering. This is accomplished by tracking the maximum opaque depth value within the bucket being rendered. If an object's minimum depth is greater that this value, object is deferred to the next bucket or deleted. This potentially doubles the rendering speed although there is a certain overhead involved for sorting the objects in buckets with their depths and maximum depth value tracking. However, for high depth complexity scenes, occlusion culling saves lots of processing.

Since the occlusion culling involves tracking the maximum depth in the framebuffer, patch cracks that usually manifest themselves as missing pixels impair the maximum depth estimate. Although binary dicing can ameliorate some of the problem, Pixie does not have a definitive solution to patch crack problem (any suggestions ?).

An additional consideration is the bucket size. If the size of the buckets are too big, then the depth variation within the bucket may be too large, decreasing the effectiveness of the culling.

Currently only **stochastic** and **zbuffer** hiders support occlusion culling.

1.8 Raytracing in C with Pixie

You can use the Pixie runtime library to raytrace instead of creating an image. There are three functions that can be used **just before** `WorldEnd` to raytrace. Since these functions are "C" specific, they do not have rib bindings. Also these functions are **not** standard RenderMan (they are Pixie specific). If any of these functions are used, `WorldEnd` is skipped and no image will be generated. The auxiliary raytracing commands are:

1. ```
   void RiTrace(RtInt n,
                RtPoint *from,
                RtPoint *to,
                RtPoint *Ci);
   ```
   This function traces `n` rays starting from `"from"` and going towards `"to"` and returns the radiance in `"Ci"`. The parameters are arrays each of which has `"n"` items.

2. ```
   void RiVisibility(RtInt n,
                     RtPoint *from,
                     RtPoint *to,
                     RtPoint *Oi);
   ```
   This function is the same with `RiTrace` except that it returns the opacity between two points.

3. ```
   void RiTraceEx(RtInt n,
                  RtPoint *from,
                  RtPoint *to,
                  RtPoint *Ci,
                  RtPoint *Oi,
                  RtFloat *t,
                  char **names);
   ```
   This function can be used to get both the radiance and the opacity as well as the parametric coordinate of the first hit (in `"t[i]"` such that `P[i] = from[i]*(1-t[i]) + to[i]*t[i]`) and the name of the first object that intersects the ray (in `"names[i]"`).

Note that these functions trace multiple rays. The performance is improved if the number of rays is high and if the rays have strong spatial coherence (i.e., they roughly start from the same position and go towards the same direction). Depending on the demand, I can add more raytracing functions that have more specialized parameters.
   ```

1.9 Network Parallel Rendering

Pixie supports network parallel rendering where bunch of computers work on a single frame. Through this document, I'll use "client" as the computer that's dispatching the rendering job. This is usually the computer that you are working on. I'll use "server" for the computers that actually do the rendering. So for a given "rib" file, multiple servers work for a single client. For a computer to work as a server, a network rendering deamon has to be running on it. You can spawn a network rendering deamon on a computer by `rndr -d <port>`. To render a rib file on the net, you can use: `rndr -s <server1:port1,server2:port2,...> <rib_file>`. Where `<server:port>` is an address descriptor for the server. The address of a server is displayed when you start the deamon. All the external files (rib, sdr, [dll/so], tif etc.) that the server can not find on the local computer will be downloaded from the client computer to a temporary folder (default: `temp`). At the end of rendering, the contents of the default folder is deleted. It is important that the directory that you launch the deamon is writable for this purpose. You also want this directory to be a local one.

1.10 DSO Shaders

Pixie supports the use of DSO functions. That means, you can call regular C/C++ functions implemented in a dll/so from the shading language as if they were built in functions. The mechanism is quite simple:

1.10.1 In DLL/SO

You need to include `shadeop.h` which contains the necessary macros. For each function you implemented as a DSO, you need to create a `SHADEOP` table:

```
#include <shadeop.h>

//This is the shadeop dispatch table
//The renderer and the shading language compiler will
//  check this table to figure out what function in the
//  dll/so implements what.
//
//Each line in the table designates a function
SHADEOP_TABLE(myfun) = {
{"float myfun_f(vector)","myinit_f","mycleanup_f"},
{"vector myfun_v(float,float,float)","",""},
{""} // End the table with an empty entry
};

// This is the function that implements
//  float   myfun(vector)
SHADEOP(myfun_f) {
    float   *result =   (float *) argv[0];
    float   *input  =   (float *) argv[1];
```

```
    result[0]       =   (input[0] + input[1] + input[2]) / (float)
    3;
}

// This is the init function for the previous function
SHADEOP_INIT(myiniy_f) {
    return NULL;
}

// This is the cleanup function for the previous function
SHADEUP_CLEANUP(mycleanup_f) { }

// This is the function that implements
//  vector myfun(float,float,float)
SHADEOP(myfun_v) {
    float   *result =   (float *) argv[0];
    float   *input1 =   (float *) argv[1];
    float   *input2 =   (float *) argv[2];
    float   *input3 =   (float *) argv[3];

    result[0]   =   input1[0];
    result[1]   =   input2[0];
    result[2]   =   input3[0];
}
```

The macro `SHADEOP_TABLE(name)` defines a table that contains the prototype for the DSO function and the names of the init and cleanup functions for that function for each different polymorphic occurrence of the function `name` in the DSO. The init function is called when the dll/so is first loaded into the memory. It must accept two arguments: a thread ID and a texture context pointer both of which are 0 with PIXIE. These parameters are provided for compatibility reasons. The init function must return a `void *` pointer to a transparent handle for the function. This handle will be passed to the DSO function and the cleanup functions. The cleanup function must do any cleanup required by the DSO (for example, de-allocating the memory that the init function allocates). The only parameter that this cleanup function accepts is the transparent handle that the init function returns. Finally, the actual DSO function executes the function. This function receives 3 arguments:

1. `void * initdata`: This is the handle that the init function returns.
2. `int argc`: This is the number of arguments passed to the DSO. Note that this is a redundant argument as you can clearly see the number of arguments at the prototype of the function. For example, for `float myfun(vector)` it will always be 2 (includes the return value).
3. `void *argv[]`: This is the array of pointers that contains the arguments. The `argv[0]` always points to the return value. If there is no return value (void function), this argument is not used.

For example, `"float myfun_f(vector)","myinit_f","mycleanup_f"` means that the function `myfun_f` implements the function `float myfun(vector)`. So whenever, the renderer needs to execute this particular function, `myfun_f` will be called. The init

and cleanup functions for this function are `myinit_f` and `mycleanup`. These functions are called **only once** before the first usage of the `myfun_f` and after the last usage of `myfun_f`. The code provided above also contains another form of `"myfun"` which is polymorphic to the first one: { `"vector myfun_v(float,float,float)","",""`}. This means, that `myfun_v` implements `vector myfun(float,float,float)`. This version of the function `myfun` does not have any init or cleanup functions (as indicated by ""), so the `initdata` parameter with be `NULL` all the time.

The DSO shaders prepared for the PrMan, Entrophy and RenderDotC should be compatible with Pixie although I did not test it yet.

1.10.2 In SL

In your shader code, you can now use `vector myfun(float,float,float)` and `float myfun(vector)` without any trouble. The dll/so that contains the implementation must be in the include directory (indicated by `-I` parameter). Similarly, the dll/so that has the implementation must be in the procedural search path. One important thing that you need to be careful about is that Pixie assumes all the parameters passed to a DSO function are defined as `output`. That means if DSO changes an argument, the change will stick !!!. One last note: a single dll/so may have multiple DSO's. The Pixie will automatically search all the dll/so's in the input directory for the correct DSO.

1.11 Transparency Shadow Maps

Pixie can generate Transparency Shadow Maps (very similar to deep shadow maps of PrMan). To generate a TSM, you should specify `tsm` as the display driver. A TSM will be generated only if the hider is `stochastic`.

1.12 Version Management

Since Pixie needs external resources, they have to be compatible with the renderer that's being used. As a general rule. Any resource (shader,texture,display driver, etc ...) generated by Pixie V X.Y.Z will be compatible with all Pixie releases V X.Y.*.

1.13 Pixie Liminations

1. For performance considerations, derivatives of the result of `shadow` or `environment` shading language calls can not be evaluated in the raytracing mode. Derivatives of `texture` can be used freely.