RenderDotC for Windows '95

Beginners should get started with RenderDotC by going through the functions by category.

Experienced users can get answers quickly from the alphabetical listing of RenderDotC funtions.

RenderDotC functions by category

Blocks
These functions come in pairs: one which starts a new state, and one which ends it. The nesting of these blocks must be strictly maintained. They define the structure of a RenderDotC scene. It is recommended that you indent the lines between each pair so that the structure is clearly visible.

rdcFrameBegin
rdcFrameEnd
rdcSceneBegin
rdcSceneEnd
rdcAttributePop
rdcAttributePush
rdcMatrixPop
rdcMatrixPush
rdcObjectBegin
rdcObjectEnd

Parameters
Parameters apply to everything in the scene. Therefore, they must be set prior to calling rdcSceneBegin. Attempting to change a parameter within a scene block results in an error. Parameters are only saved by rdcFrameBegin and restored by rdcFrameEnd.

rdcRasterViewport
rdcRasterAspect
rdcImageAspect
rdcImageWindow
rdcImageCrop
rdcViewIdentity
rdcViewOrthographic
rdcViewPerspective
rdcClip
rdcSampleUniform
rdcSampleJitter
rdcSampleAdaptive
rdcFilter, rdcCustomFilter
rdcFilterWidth
rdcColorGain
rdcColorGamma
rdcColorQuantize
rdcDepthQuantize
rdcColorJitter
rdcDepthJitter
rdcOutputDisplay
rdcOutputFile
rdcOutputSamples
rdcHider
rdcColorSpace
rdcTune

Attributes
Attributes apply to primitives and may be changed during the scene block. They may also be set prior to the scene block, sort of overriding the default values with the user's defaults. Attributes are saved and restored by frame blocks, scene blocks, object blocks, and of course attribute blocks.

rdcColor

rdcOpacity
rdcLightSource, rdcLightSourceV
rdcLightSwitch
rdcSurface, rdcSurfaceV
rdcAtmosphere, rdcAtmosphereV
rdcShadingRate
rdcShadingModel
rdcMatteObject
rdcBound
rdcFlatness
rdcOrientation
rdcBackface
rdcObjectName
rdcGroupName

Transformations
Technically, the current matrix is an attributes also. It is always saved and restored along with attributes. However, it may be saved and restored independently of attributes with a matrix block. The current matrix defines a coordinate space. At various times when describing a scene, the current matrix is used to define a built-in coordinate space.

The following functions transform the current matrix:

rdcLoadIdentity
rdcLoadMatrix
rdcMultMatrix
rdcPerspective
rdcTranslate
rdcRotate
rdcScale
rdcSkew

These functions allow you to work within custom coordinate spaces:

rdcMarkSpace
rdcProjectPoints

Polygons
There are two types of polygons: convex and general. Both must be planar. General polygons may be concave and/or have holes. Convex polygons are rendered more efficiently. If the current coordinate space is left handed, then the side of the polygon from which the vertices appear in clockwise order will be the front face. Vertices for holes in general polygons must be specified in the opposite order. For the purposes of interpolating shading variables, polygons are broken down into triangles without introducing new vertices. Polygon meshes are a compact way of specifying multiple polygons which share vertices.

rdcPolygon, rdcPolygonV
rdcPolygonMesh, rdcPolygonMeshV
rdcGeneralPolygon, rdcGeneralPolygonV
rdcGeneralPolygonMesh, rdcGeneralPolygonMeshV

Patches and NURBs
There are two types of patches: bilinear and bicubic. Bicubic patches use the basis matrices specified by rdcBasis. NURBs are more flexible than bicubic patches in that the order need not be 3, the knots need not be uniformly distributed in parametric space, and they may also be rational (solve for w as well as x, y, and z).

rdcBasis, rdcCustomBasis
rdcPatch, rdcPatchV
rdcPatchMesh, rdcPatchMeshV
rdcNurb, rdcNurbV
rdcNurbMesh, rdcNurbMeshV

Quadrics

Quadric primitives are created by sweeping a line or curve about the z-axis. If the handedness of the current coordinate space matches the current orientation (set by rdcOrientation) then the surface normals will point away from the z axis. Quadrics may be turned inside out by specifying negative arguments, such as tmax. Quadrics have four corners. You may think of a quadric surface as a square rubber sheet which has been stretched and contorted into position. All angles are in degrees.

rdcSphere, rdcSphereV
rdcCone, rdcConeV
rdcCylinder, rdcCylinderV
rdcDisk, rdcDiskV
rdcTorus, rdcTorusV
rdcParaboloid, rdcParaboloidV
rdcHyperboloid, rdcHyperboloidV

Other primitives

rdcProcedural
rdcObjectCall

Miscellaneous

rdcQuark
rdcErrorHandler, rdcCustomErrorHandler
rdcLastError
rdcComment

Alphabetical listing of all RenderDotC functions

rdcOrientation
rdcOutputDisplay
rdcOutputFile
rdcOutputSamples
rdcParaboloid, rdcParaboloidV
rdcPatch, rdcPatchV
rdcPatchMesh, rdcPatchMeshV
rdcPerspective
rdcPolygon, rdcPolygonV
rdcPolygonMesh, rdcPolygonMeshV
rdcProcedural
rdcProjectPoints
rdcQuark
rdcRasterAspect
rdcRasterViewport
rdcRotate
rdcSampleAdaptive
rdcSampleJitter
rdcSampleUniform
rdcScale
rdcSceneBegin
rdcSceneEnd
rdcShadingModel
rdcShadingRate
rdcSkew
rdcSphere, rdcSphereV
rdcSurface, rdcSurfaceV
rdcTorus, rdcTorusV
rdcTranslate
rdcTune
rdcViewIdentity
rdcViewOrthographic
rdcViewPerspective

Quarks

When using the RenderDotC C language binding, quarks are used to represent strings efficiently as integers. Essentially, the string is hashed and associated with an integer. From then on, the integer, or "quark", is used *in place of* the original string.

Quarks have the following advantages over strings:

- Quarks can be compared quickly (integer compare).
- Integers consume less space when copies are made.
- Typographical errors are caught at compile time rather than run time.

Quarks are more flexible than enumerators in that the user can extend the set of quarks by defining more. While most of the quarks you will need are predefined in the header file rdc.h,
you may define new quarks if necessary by using the rdcQuark function.

When using the RDC binding, use the full string surrounded by quotations marks. Predefined quarks follow an easy naming convention best seen by example: RDC_PLASTIC is "plastic" in RDC bytestream. Even though quarks are represented by the full string in the RDC file, new quarks must be declared before use.

Aside: Why are they called "quarks"?
The name comes from X-Windows. First, there were "atoms", which were integers that represented strings that needed to be communicated between the client and server. Integers are much cheaper than strings when communications are involved. When the concept was carried over to strings which only needed to be represented on the client side, they were called "quarks" because they were smaller than atoms.

Argument lists

Some RDC functions take an optional list of additional arguments after the fixed, required arguments. This is where the user can specify additional information and override default values. The format of this list is a series of quark-value pairs. The quark identifies which internal variable you wish to override, and the value is the new value you wish to assign to that internal variable.

Those RDC functions which take an argument list will have two C bindings. The first one has a signature ending with ellipses (...) and the second one has the same name as the first but with a capital V suffix.

Here is how you would use the first form to add color and opacity which varied across the surface of a sphere:

```
RDCcolor four_colors[4];
RDCcolor four_opacities[4];
rdcSphere(1, -1, 1, 360,
    RDC_COLOR, (RDCvoid *)four_colors,
    RDC_OPACITY, (RDCvoid *)four_opacities,
    );
```

This form takes advantage of the C's variable argument support in <stdarg.h>. First note that the values must all be cast to RDCvoid pointers. This is because the compiler cannot perform type checking/conversion to arguments passed through the stdarg method. Second, note that the list must be terminated with RDC_NULL *even if the list is empty.* This is the way RenderDotC recognizes the end of the list.

Here is the same example only using the second form of rdcSphere:

```
RDCcolor four_colors[4];
RDCcolor four_opacities[4][;
RDCquark quarks[] = {RDC_COLOR, RDC_OPACITY};
RDCvoid *values[] = {four_colors, four_opacities};
rdcSphereV(1, -1, 1, 360, 2, quarks, values);
```

This form takes the count of quark-value pairs, an array of quarks, and an array of values. The two arrays must have at least the number of elements given as the count and the order of the quarks must match that of the values (i.e. quarks[i] goes with values[i]).

There is only one RDC binding for functions with argument lists. Here is the same example in RDC:

```
rdcSphere 1 -1 1 360
    "color" [0 1 0   0 1 0   0 0 1   0 0 1]
    "opacity" [1 1 1   1 1 1   .9 .9 .9   .9 .9 .9]
```

Shading variables

Shading variables are quarks that are associated with values. The quark acts as the name of the variable and the user provides the values. Like variables in a programming language, each shading variable has a class and a type.

The class of a shading variable may be const, vary, or vertex. The class determines how many values are needed to fully specify a variable with respect to a primitive. Shading variables with class const need only one value per primitive, because the same value applies over the entire surface of the primitive. Class vary indicates that the value varies over the surface. Except in the case of polygons, shading variables of class vary will need an array of four values, one for each corner. For polygons, the size of the array of values must equal the number of vertices. Values at the interior of the primitive surface are linearly interpolated Finally, shading variables with class vertex specify vertex positions and naturally must have as many values as there are vertices in a polygon, patch, or NURB (or meshes of the same).

The type of a shading variable may be float, integer, string, pair, point, pointw, or color. Types float, integer and string expect values of types (RDCint *), (RDCfloat *), and (char *), respectively. Types pair, point, and pointw expect arrays of RDCfloat of sizes 2, 3, and 4 respectively. A pointw is a "homogeneous" or "affine" point where the fourth dimension is called w. Type color expects an array of RDCfloat of the same size as that set with rdcColorSpace, 3 by default.

Here are all of the predefined shading variables:

Name	Class	Type	Floats	Quadric	Polygon	Patch	NURB
"point"	vertex	point	3		x	x	x
"pointw"	vertex	pointw	4				x
"height"	vertex	float	1			x	x
"normal"	vary	point	3	x	x	x	x
"plane"	const	point	3	x	x	x	x
"color"	vary	color	3*	x	x	x	x
"opacity"	vary	color	3*	x	x	x	x

*Can be changed with rdcColorSpace.

Polygons, patches, and NURBs (and meshes of the same) require that the position of each vertex be given by a shading variable of class vertex. With polygons, the only choice is "point". With patches, you may also specify a height field using "height". This just gives the z component of each vertex. The x and y components are spaced evenly over the unit square. NURBs may be specified with either of the above variables or with "pointw". The latter is recommended because it's the "w" that makes a NURB rational.

By default, normal vectors are computed automatically at every shading point from the underlying surface geometry. If you wish to override the computed value, you may use "normal" or "plane". The difference between the two is that "normal" is of class vary and "plane" is const. The "plane" normal will be used across the entire surface which most useful for polygons and bilinear patches.

By default, color and opacity are taken from the attributes set by rdcColor and rdcOpacity. These values may be overridden by the shading variables "color" and "opacity". One advantage of overriding is that the color/opacity may vary across the face of the primitive.

rdcAtmosphere

Name

rdcAtmosphere, **rdcAtmosphereV** - set current atmospheric shader

C Binding

RDCvoid rdcAtmosphere(RDCquark *name*,
...)
RDCvoid rdcAtmosphereV(RDCquark *name*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcAtmosphere *name arguments*

Parameters

name

Name of atmospheric shader. Accepted values are RDC_NULL, RDC_DEPTHCUE, and RDC_FOG. RDC_NULL turns off all atmospheric effects which is the default state. The other values are described below.

arguments

The argument list depends upon the name as described below.

Description

rdcAtmosphere applies an atmospheric shader to subsequently defined primitives. Note that the atmospheric shader is an attribute which can vary from primitive to primitive.

Depth-cueing

```
RDCfloat hither = 0;
RDCfloat yon = 1;
RDCcolor background = {0, 0, 0};
rdcAtmosphere(RDC_DEPTHCUE,
    RDC_HITHER, (RDCvoid *)&hither,
    RDC_YON, (RDCvoid *)&yon,
    RDC_BACKGROUND, (RDCvoid *)background,
    RDC_NULL);
```

Surfaces closer than hither are unaffected by this atmospheric shader. Surfaces further than yon fade completely to background. Surfaces at distances in between get their surface color linearly mixed with background. The default values are shown in the example above.

Fog

```
RDCcolor background = {0, 0, 0};
RDCfloat distance = 1;
rdcAtmosphere(RDC_FOG,
    RDC_BACKGROUND, (RDCvoid *)background,
    RDC_DISTANCE, (RDCvoid *)&distance,
    RDC_NULL);
```

All surfaces are affected by this atmosphere. The amount of color which is blended with the surface color is given by the exponential decay (1 - e-surfacedepth/distance) where surfacedepth is the actual distance from the eye to the surface. A surface right at the eye point would be unobscured by fog. Even objects at a great distance will retain some of their surface color, assymtotically approaching

background with distance. The default values are shown in the example above.

rdcAttributePop

Name

rdcAttributePop - restore all attributes from stack

C Binding

RDCvoid rdcAttributePop()

RDC Binding

rdcAttributePop

Description

Restores all of the attributes to their state prior to the matching call to rdcAttributePush. Note that this does not affect parameters. It is an error to pop when the stack is empty or when there has been an intervening call to start a new state (e.g. rdcMatrixPush) which has not been properly nested.

Example

```
rdcAttributePop();
```

rdcAttributePush

Name

rdcAttributePush - save all attributes on stack

C Binding

RDCvoid rdcAttributePush()

RDC Binding

rdcAttributePush

Description

Saves all of the attributes on the attribute stack. The attributes may then be changed arbitrarily and later restored with rdcAttributePop. Note that this does not affect parameters. Pushing and popping attributes must be nested properly within other calls which change the graphics state (e.g. rdcMatrixPush and rdcMatrixPop). The stack is initially empty. There is no arbitary limit on maximum stack size.

Example

```
rdcAttributePush();
```

rdcBackface

Name

rdcBackface - turn backface removal on or off

C Binding

RDCvoid rdcBackface(RDCboolean *onoff*)

RDC Binding

rdcBackface *onoff*

Parameters

onoff

RDC_TRUE to turn backface removal on. RDC_FALSE to turn backface removal off.

Description

With backface removal on, only the "outside" facing surfaces are visible. Note that the notion of "outside" can be changed with rdcOrientation. Also, quadric primitives can sometimes be turned inside-out by using a negative sweep angle, etc.

Backface removal is off by default, which results in both the "inside" and "outside" facing surfaces being visible.

Turning backface removal on improves rendering speed only marginally, and must be used care to ensure that the desired face is the one that is visible.

Example

```
rdcBackface(RDC_TRUE);
```

rdcBasis, rdcCustomBasis

Name

rdcBasis, rdcCustomBasis - load built in or custom basis for bicubic patches

C Binding

RDCvoid rdcBasis(RDCenum *which*,
RDCenum *basis*)
RDCvoid rdcCustomBasis(RDCenum *which*,
const RDCfloat *matrix*[4][4],
RDCint *step*)

RDC Binding

rdcBasis *which basis*
rdcCustomBasis *which matrix step*

Parameters

which

specifies which cubic basis matrix you wish to change:
RDC_UBASIS change the basis matrix to be used in the u direction.
RDC_VBASIS change the basis matrix to be used in the v direction..
RDC_UVBASIS use the same basis matrix for both directions.

basis

identifies a built-in basis matrix:
RDC_POWERBASIS use the Power (identity) basis.
RDC_HERMITEBASIS use the Hermite basis.
RDC_CATMULLROMBASIS use the Catmull-Rom basis.
RDC_BSPLINEBASIS use the B-spline basis.
RDC_BEZIERBASIS use the Bezier basis.

matrix

user defined basis matrix. This advanced feature gives the user the power to define new cubic basis matrices and use them for evaluating patches and patch meshes. Controls such as tension and gain can be factored in to the matrix.

step

When evaluating a curve comprised of multiple segments, *step* is the number of control points which affect the first curve segment but not the second. This number must be at least 1 but no more than 4.

For example, it takes 7 control points to define two consecutive Bezier curve segments. Points 1 through 4 define the first segment while points 4 through 7 define the second. Only point 4 is shared by the two segments. The step for a Bezier basis is therefore 3 because we must advance from point 1 to point 4 to evaluate the next segment.

B-splines, on the other hand, only require 5 control points to define two curve segments. Points 2, 3, and 4 are shared by the two segments. The B-spline step is 1.

Description

Use these functions in conjunction with the bicubic variety of rdcPatch and rdcPatchMesh. The default basis matrix is Bezier for both the u and v directions. Note that you may define a bicubic patch which uses a different basis matrix in each direction.

Typically, one of the built-in basis matrices will be sufficient and will be used in both the u and v direction.

Examples

```
rdcBasis(RDC_UVBASIS, RDC_BSPLINE);

RDCfloat basis[4][4] = {
    {-0.01, 0.37, -0.86, 0.51},
    {0.04, -0.41, 0.37, 0},
    {-0.04,-0.42, 0.46, 0},
    {0.01, 0.48, 0.51, 0}
};
rdcCustomBasis(RDC_UVBASIS, basis, 1);
```

rdcBound

Name

rdcBound - tighten bounding box

C Binding

RDCvoid rdcBound(const RDCfloat **bound*)

RDC Binding

rdcBound *bound*

Parameters

bound

six RDCfloats which describe a 3-Dimensional box in object coordinates: {xmin, xmax, ymin, ymax, zmin, zmax}

Description

Promises the renderer that subsequent primitives will lie completely within the box in object coordinates. Any surface not within the box may be rendered partially or not at all.

While the renderer computes bounding boxes for all primitives, sometimes the bounds are not as tight as possible and can be trimmed by the user. The bounding box actually used is the intersection between the bound given and the bound automatically computed.

The default bounding box is infinite in all directions (boundless). Note that you may not make the renderer's bounds any looser, only tighter.

Example

```
RDCbound bound = {0.1, 1, 0.2, 3, 0, 1};
rdcBound(bound);
```

rdcClip

Name

rdcClip - set front and back clipping planes

C Binding

RDCvoid rdcClip(RDCfloat *hither*,
RDCfloat *yon*)

RDC Binding

rdcClip *hither yon*

Parameters

hither
distance from the eye to the near clipping plane

yon
distance from the eye to the far clipping plane

Description

Sets the near and far clipping plane along the direction of view. *hither* must be greater than or equal to RDC_EPSILON and less than *yon*. *Yon* must be greater than *hither* and less than or equal to RDC_INFINITY.

Surfaces between the near and far clipping planes are not rendered. By default, *hither* is RDC_EPSILON and *yon* is RDC_INFINITY. Using clipping planes which fit the scene more tightly is recommended because it can improve performance and image quality.

In image and raster coordinates, points at the near clipping plane are represented by z = 0 and points at the far clipping plane have z = 1, with points in between interpolated linearly.

Example

```
rdcClip(0.3, RDC_INFINITY);
```

rdcColor

Name

rdcColor - set the current color

C Binding

RDCvoid rdcColor(const RDCfloat **color*)

RDC Binding

rdcColor *color*

Parameters

color

an array array of floats (usually 3) which define a color in the current color space

Description

Subsequent primitives will use this color as their surface color unless it is explicitly overriden in the primitive's argument list.

By default, the color space has three components: red, green, and blue. This may be changed by a call to rdcColorSpace. The default color is white: {1, 1, 1}

Example

```
RDCcolor green = {0, 0.7, 0.1};
rdcColor(green);
```

rdcColorGain

Name

rdcColorGain - control gain applied to colors

C Binding

RDCvoid rdcColorGain(RDCfloat *gain*)

RDC Binding

rdcColorGain *gain*

Parameters

gain

scalar which multiplies colors linearly in the exposure process

Description

Colors will be multiplied by gain during the exposure process. This happens after sampling but before quantization to integers. The default is no gain: 1.0

Example

```
rdcColorGain(3.1);
```

rdcColorGamma

Name

rdcColorGamma - control gamma correction applied to colors

C Binding

RDCvoid rdcColorGamma(RDCfloat *gamma*)

RDC Binding

rdcColorGamma *gamma*

Parameters

gamma

inverse of exponent to which colors are raised in the exposure process

Description

Colors will be raised to the power 1/gamma during the exposure process. This happens after sampling but before quantization to integers. The default is no gamma correction: 1.0

Gamma correction is often used to control brightness when displayed on a monitor. While the "correct" value of gamma varies with each monitor, 2.1 is a typical value.

Example

```
rdcColorGamma(2.1);
```

rdcColorJitter

Name

rdcColorJitter - control random jitter applied to colors

C Binding

RDCvoid rdcColorJitter(RDCfloat *ampl*)

RDC Binding

rdcColorJitter *ampl*

Parameters

ampl

absolute value of the maximum random jitter delta that should be added to color samples after quantization

Description

Controls random jitter applied to colors. Jittering colors replaces Mach banding (contours, color aliasing) with noise. Increasing *ampl* decreases the Mach band effect but increases the noise in color spectrum. The default value is 0.5, which can cause the color to bump up or down one integer quantization level.

Example

```
rdcColorJitter(0.9);
```

rdcColorQuantize

Name

rdcColorQuantize - control mapping of colors from real to integer

C Binding

RDCvoid rdcColorQuantize(RDCint *one*,
RDCint *min*,
RDCint *max*)

RDC Binding

rdcColorQuantize *one min max*

Parameters

one

scalar which multiplies real colors (RDCfloat samples) when converting to integer samples. To disable quantization and store floating point values, set *one* to 0.

min

the lowest allowable integer value for a color sample (usually 0). Ignored if *one* is 0.

max

the highest allowable integer value for a color sample (usually the same as *one*). Ignored if *one* is 0.

Description

During rendering, colors are computed with floating point accuracy (32 bits per channel). When it comes time to display or store the image, the colors are usually converted to integers. This function controls the process of converting real color samples to integers, known as color quantization.

The first parameter, *one*, multiplies each color sample, scaling it to fit the desired integer range. After the color sample is multiplied by *one* and converted to an integer, it is clamped between the other two parameters, *min* and *max*. If *one* is 0, color quantization is turned off and floating point samples are output.

The floating point representation of a color channel is usually in the range 0 to 1. By default, *one* is 255, *min* is 0, and *max* is 255, scaling the range 0-1 up to 0-255 which can be represented by 8 bits per channel. Note that you may define real colors in the range 0.0 to 255.0 and set *one* to 1 for the same results.

Example

```
rdcColorQuantize(15, 0, 15);
```

rdcColorSpace

Name

rdcColorSpace - define a custom space in which colors are described

C Binding

RDCvoid rdcColorSpace(RDCint *n*,
const RDCfloat **nRGB*,
const RDCfloat **RGBn*)

RDC Binding

rdcColorSpace *n nRGB RGBn*

Parameters

n

the new number of components per color.

nRGB

an array of n by 3 RDCfloats, which converts colors in the new space to RGB.

RGBn

an array of 3 by n RDCfloats, which convert RGB colors to the new space.

Description

After defining a new color space, all color values passed to the renderer must have *n* components. The renderer will convert the color to RGB for further calculations.

Example

```
RDCfloat nRGB[] = {0.2, 0.3, 0.5};
RDCfloat RGBn[] = {1, 1, 1};
rdcColorSpace(1, nRGB, RGBn);
```

rdcComment

Name

rdcComment - add a comment to the output bytestream

C Binding

RDCvoid rdcComment(const char **fmt*,
...)

RDC Binding

none

Parameters

fmt

a format string in the style of printf. The format string cannot contain a newline (\n) character.

Description

When linking with the RDC bytestream output library, this is a way to add a comment line to the RDC bytestream. The format string will be evaluated just as in printf: the % tokens will be replaced with arguments which follow. The entire line will be prefixed by the RDC comment character and a space: "# ". The line will be automatically terminated with a newline.

Example

rdcComment("Frame Number %d", framenum);

rdcCone, RdcConeV

Name

rdcCone, rdcConeV - draw a cone

C Binding

```
RDCvoid rdcCone(RDCfloat radius,
    RDCfloat height,
    RDCfloat zmax,
    RDCfloat tmax,
    ...)
RDCvoid rdcConeV(RDCfloat radius,
    RDCfloat height,
    RDCfloat zmax,
    RDCfloat tmax,
    RDCint n,
    const RDCquark *quarks,
    RDCvoid *const *parms)
```

RDC Binding

rdcCone *radius height zmax tmax argumentlist*

Parameters

radius

the radius of the cone at the bottom

height

the height of the apex of the cone

zmax

if less than *height*, where to truncate the top of the cone

tmax

sweep angle about Z-axis

argumentlist

see Shading Variables

Description

The base of the cone lies in the X-Y plane and is open. The apex of the cone is on the Z-axis at z = *height*. If *zmax* >= *height*, the apex of the cone is visible. If *zmax* < *height*, the top of the cone is cut off. *zmax* will be clamped between the range [0, *height*]. *tmax* can vary from - 360 to +360 to render sections of the cone.

Variables which vary over the surface (such as RDC_COLOR) must have four components.

Example

```
RDCcolor fourcolors[4] = {
    {0.2, 0.9, 0.1},
    {0, 0.1, 0.5},
    {0.3, 0.4, 0.5},
    {0.3, 0.3, 0.3}
};
rdcCone(0.2, 1, 0.9, 360, RDC_COLOR,   (RDCvoid *)fourcolors, RDC_NULL);
```

rdcCylinder, rdcCylinderV

Name

rdcCylinder, rdcCylinderV - draw a cylinder

C Binding

```
RDCvoid rdcCylinder(RDCfloat radius,
    RDCfloat zmin,
    RDCfloat zmax,
    RDCfloat tmax,
    ...)
RDCvoid rdcCylinderV(RDCfloat radius,
    RDCfloat zmin,
    RDCfloat zmax,
    RDCfloat tmax,
    RDCint n,
    const RDCquark *quarks,
    RDCvoid *const *parms)
```

RDC Binding

rdcCylinder *radius zmin zmax tmax argumentlist*

Parameters

radius
the radius of the cylinder

zmin
the z coordinate of the bottom of the cylinder

zmax
the z coordinate of the top of the cylinder

tmax
sweep angle about Z-axis

argumentlist
see Shading Variables

Description

The top and bottom of the cylinder are parallel with the X-Y plane and are open. The bottom is at z = *zmin* and the top is at z = *zmax*. *tmax* can vary from - 360 to +360 to render sections of the cylinder.

Variables which vary over the surface (such as RDC_COLOR) must have four components.

Example

```
RDCcolor fourcolors[4] = {
    {0.2, 0.9, 0.1},
    {0, 0.1, 0.5},
    {0.3, 0.4, 0.5},
    {0.3, 0.3, 0.3}
};
rdcCylinder(0.5, 0, 5, 180, RDC_COLOR, (RDCvoid *)fourcolors, RDC_NULL);
```

rdcDepthJitter

Name

rdcDepthJitter - control random jitter applied to depth values

C Binding

RDCvoid rdcDepthJitter(RDCfloat *ampl*)

RDC Binding

rdcDepthJitter *ampl*

Parameters

ampl

the amplitude of the random jitter value added to depth values.

Description

By default, the depth jitter is 0 for no jittering of depth values. Jittering depth can affect the hidden surface removal algorithm and generally does not improve the quality of the image. It is included here for completeness and is only recommended if one is producing a greyscale depth map image, in which case, the depth will be represented as a color and is subject to Mach banding.

Example

```
rdcDepthJitter(0.5);
```

rdcDepthQuantize

Name

rdcDepthQuantize - control mapping of depth values from real to integer

C Binding

RDCvoid rdcDepthQuantize(RDCint *one*,
RDCint *min*,
RDCint *max*)

RDC Binding

rdcDepthQuantize *one min max*

Parameters

one

scalar which multiplies real depth values (RDCfloat samples) when converting to integer samples. To disable quantization and store floating point values, set *one* to 0.

min

the lowest allowable integer value for a depth sample (usually 0). Ignored if *one* is 0.

max

the highest allowable integer value for a depth sample (usually the same as *one*). Ignored if *one* is 0.

Description

During rendering, depth values are computed with floating point accuracy (32 bits). If depth values are to be displayed or stored, they may be converted to integers. This function controls the process of converting real depth values to integers, known as depth quantization.

The first parameter, *one*, multiplies each depth sample, scaling it to fit the desired integer range. After the depth sample is multiplied by *one* and converted to an integer, it is clamped between the other two parameters, *min* and *max*. If *one* is 0, depth quantization is turned off and floating point samples are output.

The floating point representation of a depth sample lies in the range 0 to 1. By default, *one* is 0 and depth quantization is turned off.

Example

```
rdcDepthQuantize(65535,  0, 65535);
```

rdcDisk, rdcDiskV

Name

rdcDisk, rdcDiskV - draw a disk

C Binding

```
RDCvoid rdcDisk(RDCfloat innerradius,
    RDCfloat outerradius,
    RDCfloat tmax,
    ...)
RDCvoid rdcDiskV(RDCfloat innerradius,
    RDCfloat outerradius,
    RDCfloat tmax,
    RDCint n,
    const RDCquark *quarks,
    RDCvoid *const *parms)
```

RDC Binding

rdcDisk *innerradius outerradius tmax argumentlist*

Parameters

innerradius

the radius of the hole in the middle of the disk (0 for no hole)

outerradius

the radius of the outer rim of the disk

tmax

sweep angle about Z-axis

argumentlist

see Shading Variables

Description

The disk lies in the X-Y plane. If *innerradius* > 0, there is a hole in the center. *innerradius* must be less than *outerradius*. *tmax* can vary from - 360 to +360 to render sections of the disk.

Variables which vary over the surface (such as RDC_COLOR) must have four components.

Example

```
rdcDisk(3,  6,  360, RDC_NULL);
```

rdcErrorHandler, rdcCustomErrorHandler

Name

rdcErrorHandler, rdcCustomErrorHandler - use built in or custom error handler

C Binding

RDCvoid rdcErrorHandler(RDCenum *severitymask*,
RDCenum *handler*)
RDCvoid rdcCustomErrorHandler(RDCenum *severitymask*,
RDCerrorFunc *function*)

RDC Binding

rdcErrorHandler severitymask handler
(none for rdcCustomErrorHandler)

Parameters

severitymask

Which types of errors to be handled this way. Predefined severity masks include RDC_INFO, RDC_WARNING, RDC_ERROR, RDC_SEVERE and all combinations thereof. RDC_IWES is all of the above. Other possible combinations are RDC_IW, RDC_IE, RDC_IS, RDC_IWE, RDC_IWS, RDC_IES, RDC_WE, RDC_WS, RDC_WES, and RDC_ES.

handler

a built-in error handler:
RDC_ERRORIGNORE - the error is ignored and rendering continues
RDC_ERRORPRINT - an error message is printed and rendering continues
RDC_ERRORABORT - an error message is printed and rendering is aborted

function

pointer to a user defined error handling function. The required signature is:
RDCvoid function(RDCint code, RDCint severity, const char *msg);

Description

There are four classes of errors which might arise at run time:
RDC_INFO - rendering progress, statistics, etc.
RDC_WARNING **-** warnings such as unimplemented feature.
RDC_ERROR **-** user input error such as a parameter out of range.
RDC_SEVERE **-** severe errors such as out of memory, renderer should abort.

You may specify how you want each type of error handled. By default, RDC_INFO errors are ignored, RDC_WARNING and RDC_ERROR are printed, and RDC_SEVERE are aborted. You may choose one of the built-in error handling function or write your own.

Example

rdcErrorHandler(RDC_IWES, RDC_ERRORIGNORE);

rdcFilter, rdcCustomFilter

Name

rdcFilter, rdcCustomFilter - use built in or custom filter function

C Binding

RDCvoid rdcFilter(RDCenum *filter*)
RDCvoid rdcCustomFilter(RDCfilterFunc *filterfunc*)

RDC Binding

rdcFilter *filter*
(none for rdcCustomFilter)

Parameters

filter

a predefined filter for filtering supersamples:
RDC_BOXFILTER use a box filter.
RDC_TRIANGLEFILTER use a triangle filter.
RDC_CATMULLROMFILTER use a Catmull-Rom filter.
RDC_GAUSSIANFILTER use a Gaussian filter.
RDC_SINCFILTER use a sinc filter.

filterfunc

a user defined filter function for filtering supersamples. The required signature is:
RDCfloat filterfunc(RDCfloat x, RDCfloat y, RDCfloat xwidth, RDCfloat ywidth)
where x and y are the signed distances from the pixel center to the sample and xwidth and ywidth give the support width of the filter.

Description

For each pixel, all samples within the support width from the center of the pixel are weighted and averaged. The filter function determines the weighting. The support width is given by rdcFilterWidth. The default filter is Gaussian.

Example

rdcFilter(RDC_TRIANGLEFILTER);

rdcFilterWidth

Name

rdcFilterWidth - Set support width of filter function

C Binding

RDCvoid rdcFilterWidth(RDCfloat *xwidth*,
RDCfloat *ywidth*)

RDC Binding

rdcFilterWidth *xwidth ywidth*

Parameters

xwidth

x component of the width of the filter function

ywidth

y component of the width of the filter function

Description

Describes a rectangle with dimensions *xwidth* by *ywidth* which will be placed symmetrically about the center of each pixel. Any samples which fall within that rectangle will be passed to the filter function specified by rdcFilter or rdcCustomFilter. Depending on which filter is called, the increasing the width may affect the shape of the filter (stretch it) or it may just cause the same function to be evaluated at further points.

Note that if a filter width is greater than 1, then the same sample may affect more than one final pixel color. If a filter width is less than 1, then some samples may be missed entirely.

By default, both widths are 2.

Example

```
rdcFilterWidth(1, 1);
```

rdcFlatness

Name

rdcFlatness - set tesselation threshhold for geometry

C Binding

RDCvoid rdcFlatness(RDCfloat *value*)

RDC Binding

rdcFlatness *value*

Parameters

value

the maximum allowable deviation in pixels between the actual surface and its polygonal approximation.

Description

Polygon tesselation is most noticable around the silhouette edges. Even if you intentionally make the tesselation coarse with rdcShadingRate, you may still want the silhouette to look smooth. The renderer takes both of these factors into consideration when adaptively tesselating a curved surface.

The default flatness is 0.5 which means that the approximated surface will never miss the true surface by more than half a pixel. Decreasing this value increases the number of polygonal facets, especially at the silhouette edges.

Example

```
rdcFlatness(0.1);
```

rdcFrameBegin

Name

rdcFrameBegin - start another frame of animation

C Binding

RDCvoid rdcFrameBegin()

RDC Binding

rdcFrameBegin

Description

If you are rendering an animated sequence, you make break up your scene into a series of blocks bracketed by rdcFrameBegin and rdcFrameEnd. Each block is repsonsible for rendering one frame of animation. The advantage to this approach is that elements common to all frames may be initialized once and for all before the first rdcFrameBegin.

rdcFrameBegin saves the current parameters, attributes, and transformation on their respective stacks. Frame blocks may not be nested.

Example

```
rdcFrameBegin();
```

rdcFrameEnd

Name

rdcFrameEnd - end a frame of animation

C Binding

RDCvoid rdcFrameEnd()

RDC Binding

rdcFrameEnd

Description

Ends a block started by rdcFrameBegin. Parameters, attributes, and transformation are restored to their previous states. It is an error to call rdcFrameEnd without a previously unmatched rdcFrameBegin or if intervening calls to state changing functions (e.g. rdcAttributePush) have not been properly nested.

Example

```
rdcFrameEnd();
```

rdcGeneralPolygon, rdcGeneralPolygonV

Name

rdcGeneralPolygon, rdcGeneralPolygonV - draw a general polygon

C Binding

RDCvoid rdcGeneralPolygon(RDCint *nloops*,
    const RDCint **nverts*,
    ...)
RDCvoid rdcGeneralPolygonV(RDCint *nloops*,
    const RDCint **nverts*,
    RDCint n,
    const RDCquark *quarks,
    RDCvoid *const *parms)

RDC Binding

rdcGeneralPolygon *nloops nverts argumentlist*

Parameters

nloops

the number of loops. The perimeter counts as one loop, so nloops must be at least 1. Each hole counts as an additional loop.

nverts

the number of vertices in each loop. The number of vertices in the perimeter loop is given by *nverts*[0]. Likewise for each hole.

argumentlist

The argument list must contain at least point information, RDC_POINT. There must be as many points as the sum of all the elements in *nverts*. Also see Shading Variables.

Description

Draw a general planar, concave polygon with or without holes. The first loop specified is the outer boundary of the polygon. All other loops are holes. The holes must be specified in the reverse order of the outer boundary (i.e. the holes are counterclockwise if the outer boundary is clockwise.)

Example

```
RDCint nverts = {4, 3};
RDCpoint points = {{0, 0, 0}, {0, 1, 0}, {0, 1, 1}, {0, 0, 1},
    {0, 0.25, 0.5}, {0, 0.75, 0.75}, {0, 0.75, 0.25}};
rdcGeneralPolygon(2, nverts, RDC_POINT, (RDCvoid *)points, RDC_NULL);
```

rdcGeneralPolygonMesh, rdcGeneralPolygonMeshV

Name

rdcGeneralPolygonMesh, rdcGeneralPolygonMeshV - draw a mesh of general polygons

C Binding

RDCvoid rdcGeneralPolygonMesh(RDCint *npolys*,
const RDCint **nloops*,
const RDCint **nverts*,
const RDCint **verts*,
...)
RDCvoid rdcGeneralPolygonMeshV(RDCint *npolys*,
const RDCint **nloops*,
const RDCint **nverts*,
const RDCint **verts*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcGeneralPolygonMesh *npolys nloops nverts verts argumentlist*

Parameters

npolys
number of general polygons in the mesh

nloops
number of loops in each polygon in the mesh. The perimeter of a polygon counts as one loop. Each hole counts as another. The number of loops in polygon i is given by *nloops*[i]. There must be as many entries in this array as there are polygons in the mesh.

nverts
number of vertices in each loop. The order is to specify the number of vertices for each loop in a polygon before proceeding to the next polygon. There must be as many entries in this array as the sum of all the entries in the *nloops* array.

verts
indeces into the vertex array. Since polygons in the mesh may share vertices, a set of actual vertices need only be declared once (in the argument list). The vertices for each loop are then specified as indeces into the actual vertex array.

argumentlist
The argument list must contain at least point information, RDC_POINT. There must be as many points as the largest value in *verts* plus one. Also see Shading Variables.

Description

Draws a mesh of general concave polygons, with or without holes. Holes must be specified in the reverse order of the outer boundary (i.e. the holes are counterclockwise if the outer boundary is clockwise.)

Example

```
RDCint nloops = {2, 2};
RDCint nverts = {4, 3, 4, 3};
RDCint verts = {0, 1, 4, 3, 6, 8, 7, 1, 2, 5, 4, 9, 11, 10};
RDCpoint points = {{0, 0, 1}, {0, 1, 1}, {0, 2, 1}, {0, 0, 0}, {0, 1, 0}, {0, 2, 0},
    {0, 0.25, 0.5}, {0 0.75, 0.75}, {0, 0.75, 0.25}, {0, 1.25, 0.5},
```

```
    {0, 1.75, 0.75}, {0, 1.75, 0.25}};
rdcGeneralPolygonMesh(2, nloops, nverts, verts,
    RDC_POINT, (RDCvoid *)points,
    RDC_NULL);
```

rdcGroupName

Name

rdcGroupName - name a group of objects

C Binding

RDCvoid rdcGroupName(const char *name)

RDC Binding

rdcGroupName *name*

Parameters

name

name to give the group.

Description

For debugging purposes, a name may be associated with a group. This name will be printed out with error messages which occur during rendering. This can help identify which group of primitives is causing the error.

Example

```
rdcGroupName("Bronze Objects");
```

rdcHider

Name

rdcHider - select a hidden surface removal technique

C Binding

RDCvoid rdcHider(RDCenum *type*)

RDC Binding

rdcHider *type*

Parameters

type

one of the predefined hidden surface removal techniques:
RDC_ZBUFFER - use the Z-buffer hidden surface removal technique.
RDC_PAINT - use manual hidden surface removal. The user is responsible for declaring primitives in a back to front order.
RDC_NULL - produce no output. Used to verify scene correctness.

Description

The default hidden surface removal technique is Z-buffer, which will automatically resolve hidden surfaces. Manual hidden surface removal reduces the resources required for rendering and puts the burden of hidden surface removal on the user, who must specify primitives in a back to front order.

Example

```
rdcHider(RDC_PAINT);
```

rdcHyperboloid, rdcHyperboloidV

Name

rdcHyperboloid, rdcHyperboloidV - draw a hyperboloid

C Binding

```
RDCvoid rdcHyperboloid(const RDCfloat *point1,
    const RDCfloat *point2,
    RDCfloat tmax,
    ...)
RDCvoid rdcHyperboloidV(const RDCfloat *point1,
    const RDCfloat *point2,
    RDCfloat tmax,
    RDCint n,
    const RDCquark *quarks,
    RDCvoid *const *parms)
```

RDC Binding

rdcHyperboloid *point1 point2 tmax argumentlist*

Parameters

point1, point2

define a line segment which will be swept about the Z-axis.

tmax

sweep angle about Z-axis

argumentlist

see Shading Variables

Description

creates a hyperboloid by taking a line segment between *point1* and *point2* and sweeping it around the Z-axis by *tmax* degrees.

Example

```
RDCcolor fourcolors[4] = {
    {0.2, 0.9, 0.1},
    {0, 0.1, 0.5},
    {0.3, 0.4, 0.5},
    {0.3, 0.3, 0.3}
};
RDCfloat point1 = {0.3, 1.2, 2};
RDCfloat point2 = {1.3, 3.2, 5};
rdcHyperboloid(point1, point2, 360, RDC_COLOR, (RDCvoid *)fourcolors, RDC_NULL);
```

rdcImageAspect

Name

rdcImageAspect - set the aspect ratio of the image window

C Binding

RDCvoid rdcImageAspect(RDCfloat *aspect*)

RDC Binding

rdcImageAspect *aspect*

Parameters

aspect

the aspect ratio of the image window (width divided height.)

Description

The image window is the rectangle on the image plane which gets mapped to the raster viewport for display or storage. The image plane is always 1 unit away from the eye (focal length is 1). By default, the image window has the same aspect ratio as the raster viewport (after taking into account the raster aspect ratio). You may override that default by specifying the image aspect ratio. In either case, the image window is centered along the line of sight:

If image aspect ratio *aspect* is greater than 1, then the image window is given by {-*aspect*, *aspect*, -1, 1}. If *aspect* is less than 1, the image window given by {-1, 1, -1/*aspect*, 1/*aspect*}

Note that rdcImageWindow will override rdcImageAspect.

Example

```
rdcImageAspect(2.0);
```

rdcImageCrop

Name

rdcImageCrop - crop the image window

C Binding

RDCvoid rdcImageCrop(RDCfloat *xmin*,
RDCfloat *xmax*,
RDCfloat *ymin*,
RDCfloat *ymax*)

RDC Binding

rdcImageCrop *xmin ymax ymin ymax*

Parameters

xmin

the left edge of the crop window

xmax

the right edge of the crop window

ymin

the top edge of the crop window

ymax

the bottom edge of the crop window

Description

Identifies rectangular subregion of the image window to be rendered by specifying the minimum and maximum x and y components. The rest of the image will be left black. These components range from 0 to 1. 0 is the left side of the screen and 1 is the right side of the screen. 0 is the top of the screen and 1 is the bottom of the screen. There may only be one crop window in effect for an entire image.

Example

```
rdcImageCrop(0, 0.1,  0.4, 1);
```

rdcImageWindow

Name

rdcImageWindow - explicitly set the window on the image plane

C Binding

RDCvoid rdcImageWindow(RDCfloat *left*,
RDCfloat *right*,
RDCfloat *bot*,
RDCfloat *top*)

RDC Binding

Parameters

left
the left edge of the window

right
the right edge of the window

bot
the bottom edge of the window

top
the top edge of the window

Description

The image window is the rectangle on the image plane which gets mapped to the raster viewport for display or storage. The image plane is always 1 unit away from the eye (focal length is 1). By default, the image window has the same aspect ratio as the raster viewport (after taking into account the raster aspect ratio) and is centered about the line of sight.

rdcImageWindow explicitly sets the window on the image plane, by bounding it with the left, right, top and bot parameters. If left > right, or bot > top, the image is inverted. For image windows symmetric about the line of sight, use rdcImageAspect.

Example

rdcImageWindow(-1, 1, 0, 1);

rdcLastError

Name

rdcLastError - recall number of most recent error

C Binding

RDCint rdcLastError()

RDC Binding

none

Description

Initially, the last error number is RDC_NOERROR. All of the built-in error handlers save the number of the last error for later use. Custom error handlers have no effect on rdcLastError.

Example

```
RDCint errno = rdcLastError();
```

rdcLightSource, rdcLightSourceV

Name

rdcLightSource, rdcLightSourceV - add a light to the scene

C Binding

RDClight *rdcLightSource(RDCquark *name*,
...)
RDClight *rdcLightSourceV(RDCquark *name*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcLightSource *name number argumentlist*

Parameters

name

Name of light source shader. Accepted values are RDC_AMBIENT, RDC_DISTANT, RDC_OMNI, and RDC_SPOT.

number

Because light handles cannot be passed back when processing RDC bytestream, each light must be given a unique number between 0 and 65535. This number may be refernced later when turning the light on and off. Low, sequential numbers are recommended as any drastic jump in light numbers will produce a warning message.

argmentlist

The argument list depends upon the name as described below.

Description

Adds a light source to the scene for the duration of the frame (permanently if defined outside of any frame.) Returns a handle to the light so that it may be turned on and off with rdcLightSwitch. Lights are on by default. Note that the list of lights which are on is an attribute.

Ambient lights

```
RDCfloat intensity = 1;
RDCcolor lightcolor = {1, 1, 1};
RDClight *handle = rdcLightSource(RDC_AMBIENT,
    RDC_INTENSITY, (RDCvoid *)&intensity,
    RDC_LIGHTCOLOR, (RDCvoid *)lightcolor,
    RDC_NULL);
```

Ambient light models random, atmospheric light which illuminates objects from all directions. It is used to make visible parts of the model which are not directly illuminated by other lights. The default values for intensity and lightcolor are shown in the above example.

Distant lights

```
RDCfloat intensity = 1;
RDCcolor lightcolor = {1, 1, 1};
RDCpoint from = {0, 0, 0};
RDCpoint to = {0, 0, 1};
RDClight *handle = rdcLightSource(RDC_DISTANT,
    RDC_INTENSITY, (RDCvoid *)&intensity,
    RDC_LIGHTCOLOR, (RDCvoid *)lightcolor,
    RDC_FROM, (RDCvoid *)from,
```

```
    RDC_TO, (RDCvoid *)to,
    RDC_NULL);
```

Distant lights are characterized by a shower of parallel light rays. The classic example is light coming to earth from the sun. Such light has only a direction, not a position. The direction is given by the vector from the point RDC_FROM to the point RDC_TO. Note that these points are influenced by the prevailing transformation at the time the light is added to the scene. The default values are shown in the example above.

Omni lights

```
RDCfloat intensity = 1;
RDCcolor lightcolor = {1, 1, 1};
RDCpoint from = {0, 0, 0};
RDClight *handle = rdcLightSource(RDC_OMNI,
    RDC_INTENSITY, (RDCvoid *)&intensity,
    RDC_LIGHTCOLOR, (RDCvoid *)lightcolor,
    RDC_FROM, (RDCvoid *)from,
    RDC_NULL);
```

Omnidirectional lights have a position from which light emanates in all directions (also known as point light sources). The intensity of light from omni light sources follows the inverse square law: the intensity is proportional to the recipricol of the distance squared. Therefore, the intensity of these lights is often greater than 1. Note that RDC_FROM is influenced by the prevailing transformation at the time the light is added to the scene.The default values are shown in the example above.

Spot lights

```
RDCfloat intensity = 1;
RDCcolor lightcolor = {1, 1, 1};
RDCpoint from = {0, 0, 0};
RDCpoint to = {0, 0, 1};
RDCfloat coneangle = 30;
RDCfloat conedeltaangle = 5;
RDCfloat beamdistribution = 2;
rdcLightSource(RDC_SPOT,
    RDC_INTENSITY, (RDCvoid *)&intensity,
    RDC_LIGHTCOLOR, (RDCvoid *)lightcolor,
    RDC_FROM, (RDCvoid *)from,
    RDC_TO, (RDCvoid *)to,
    RDC_CONEANGLE, (RDCvoid *)&coneangle,
    RDC_CONEDELTAANGLE, (RDCvoid *)&conedeltaangle,
    RDC_BEAMDISTRIBUTION, (RDCvoid *)&beamdistribution,
    RDC_NULL);
```

Spot lights emit light in the shape of a cone from a point in a certain direction. The origin is given by RDC_FROM and the cone will be centered about a vector from RDC_FROM to RDC_TO. Note that these points are influenced by the prevailing transformation at the time the light is added to the scene.

RDC_CONEANGLE gives the angle in degrees from the center axis of the cone to one edge. No light will be cast outside of this cone. The entire width of the cone (from one edge to the opposite) is 2 * RDC_CONEANGLE degrees.

Light from a spot light is brightest at the axis of the cone. At angles theta off the center axis, the intensity is given by (cos(theta))RDC_BEAMDISTRIBUTION. That is, there is an exponential falloff of intensity towards the edges of the cone.

Finally, the intensity is multiplied by a smooth falloff function whenever the angle is within

RDC_CONEDELTAANGLE degrees from the edge of the cone. This last function is 1 at RDC_CONEANGLE - RDC_CONEDELTAANGLE and drops off via smooth Hermite interpolation to 0 at RDC_CONEANGLE. This produces a penumbra effect at the edges of the spotlight.

Like point light sources, spot lights follow the inverse square law with respect to distance. The default values for all of these variables are given in the example above.

rdcLightSwitch

Name

rdcLightSwitch - turn a light on or off

C Binding

RDCvoid rdcLightSwitch(const RDClight **light*,
RDCboolean *onoff*)

RDC Binding

rdcLightSwitch *lightnumber onoff*

Parameters

light

a handle to a light previously created with rdcLightSource

lightnumber

in RDC bytestream, the sequence number assigned to a light previously created with rdcLightSource

onoff

RDC_TRUE to turn the light on, RDC_FALSE to turn it off

Description

Lights are on by default when created. Lights may be turned off and back on with rdcLightSwitch. In order to do this, you must save the handle (or number, in the case of RDC bytestream) for later reference. This list of lights which are on is an attribute which can be saved and restored with rdcAttributePush and rdcAttributePop. Lights are only destroyed when defined within a frame and rdcFrameEnd is called.

Example

```
rdcLightSwitch(light1, RDC_FALSE);
```

rdcLoadIdentity

Name
rdcLoadIdentity - reset the current matrix to the identity

C Binding
RDCvoid rdcLoadIdentity()

RDC Binding
rdcLoadIdentity

Description
Explicitly sets the current transformation matrix to the identity.

Example
```
rdcLoadIdentity();
```

rdcLoadMatrix

Name

rdcLoadMatrix - explicitly set the current matrix

C Binding

RDCvoid rdcLoadMatrix(const RDCfloat *matrix*[4][4])

RDC Binding

rdcLoadMatrix *matrix*

Parameters

matrix

the matrix to be loaded

Description

Sets the current transformation matrix to the one given.

Example

```
RDCfloat matrix[4][4] = {
    {0, 1, 1, 1},
    {2 , 0, 1, 2},
    {3, 1, 1, 1},
    {5, 3, 2, 2.1}
};
rdcLoadMatrix(matrix);
```

rdcMarkSpace

Name

rdcMarkSpace - name the current coordinate space

C Binding

RDCvoid rdcMarkSpace(RDCquark *name*)

RDC Binding

rdcMarkSpace *name*

Parameters

name

a quark representing the name you wish to give the current coordinate space

Description

You may mark the current coordinate space with a name so that later, when another coordinate space is in effect, you may project points to or from the old coordinate space with rdcProjectPoints.

Example

```
RDCquark myspace = rdcQuark("myspace");
rdcMarkSpace(myspace);
```

rdcMatrixPop

Name

rdcMatrixPop - restore matrix from stack

C Binding

RDCvoid rdcMatrixPop()

RDC Binding

rdcMatrixPop

Description

Restores the current transformation matrix to its state prior to the matching call to rdcMatrixPush. It is an error to pop when the stack is empty or when there has been an intervening call to start a new state (e.g. rdcAttributePop) which has not been properly nested.

Example

```
rdcMatrixPop();
```

rdcMatrixPush

Name

rdcMatrixPush - save the current matrix on stack

C Binding

RDCvoid rdcMatrixPush()

RDC Binding

rdcMatrixPush

Description

Saves the current transformation on the matrix stack. The current transformation may then be changed arbitrarily and later restored with rdcMatrixPop. Pushing and popping matrices must be nested properly within other calls which change the graphics state (e.g. rdcAttributePush and rdcAttributePop). The stack is initially empty. There is no arbitary limit on maximum stack size.

Example

```
rdcMatrixPush();
```

rdcMatteObject

Name

rdcMatteObject - turn the matte object flag on or off

C Binding

RDCvoid rdcMatteObject(RDCboolean *onoff*)

RDC Binding

rdcMatteObject *onoff*

Parameters

onoff

RDC_TRUE to turn on the matte object flag, RDC_FALSE to turn it off

Description

By default, the matte object flag is off. With the matte object flag on, subsequent objects are rendered as black 3D objects which hide objects behind them. In the final image, matte objects are black in color and transparent in alpha. Objects in front of matte objects are rendered normally.

This unusual combination is used in special effects for motion pictures where live action will be matted in with computer generated imagery. The matte objects "stand in" for the live action actors, props, or sets.

Example

```
rdcMatteObject(RDC_TRUE);
```

rdcMultMatrix

Name

rdcMultMatrix - transform by an arbitrary matrix

C Binding

RDCvoid rdcMultMatrix(const RDCfloat *matrix*[4][4])

RDC Binding

rdcMultMatrix *matrix*

Parameters

matrix

a transformation matrix to multiply

Description

Premultiplies the current transformation matrix by the matrix given.

Example

```
RDCfloat matrix[4][4] = {
    {0, 1, 1, 1},
    {2 , 0, 1, 2},
    {3, 1, 1, 1},
    {5, 3, 2, 2.1}
};
rdcMultMatrix(matrix);
```

rdcNurb, rdcNurbV

Name

rdcNurb, rdcNurbV - draw a Non-Uniform Rational B-spline patch

C Binding

RDCvoid rdcNurb(RDCint *uorder*,
const RDCfloat **uknot*,
RDCfloat *umin*,
RDCfloat *umax*,
RDCint *vorder*,
const RDCfloat **vknot*,
RDCfloat *vmin*,
RDCfloat *vmax*,
...)

RDCvoid rdcNurbV(RDCint *uorder*,
const RDCfloat **uknot*,
RDCfloat *umin*,
RDCfloat *umax*,
RDCint *vorder*,
const RDCfloat **vknot*,
RDCfloat *vmin*,
RDCfloat *vmax*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcNurb *uorder uknot umin umax vorder vknot vmin vmax argumentlist*

Parameters

uorder
the degree of the polynomial plus 1 in the u direction

uknot
array of knots in the u direction. There must be 2 * *uorder* knots in non-decreasing order.

umin
the minimum value of u where surface is defined. Must be less than *umax* and greater than or equal to *uknot*[*uorder* - 1].

umax
the maximum value of u where surface is defined. Must be greater than *umin* and less than or equal to *uknot*[*uorder*].

vorder
the degree of the polynomial plus 1 in the v direction

vknot
array of knots in the v direction. There must be 2 * *vorder* knots in non-decreasing order.

vmin
the minimum value of v where surface is defined. Must be less than *vmax* and greater than or equal to *vknot*[*vorder* - 1].

vmax

the maximum value of v where surface is defined. Must be greater than *vmin* and less than or equal to *vknot*[*vorder*].

argumentlist

The argument list must contain at least point information, RDC_POINTW or RDC_POINT. If you use RDC_POINT, then w is assumed to be 1 for all points and the result is a Non-Uniform Non-Rational B-spline (NUNB?). There must be *uorder* times *vorder* points in the array. Also see Shading Variables.

Description

Draws the tensor product of the rational splines defined in the u and v directions. The order need not be the same for u and v. The surface is defined over the range [umin, umax] and [vmin, vmax]. Note that umin and umax must lie between the middle two elements in the uknot array. Likewise for vmin and vmax.

Example

```
RDCfloat uknot[] = {0, 0, 0, 1, 1, 1};
RDCfloat vknot[] = {0, 0, 1, 1};
RDCpointw points[] = {{1, 0, 0, 1}, {1, 1, 0, 1}, {0, 2, 0, 2},
    {-1, 1, 0, 1}, {-1, 0, 0, 1}, {-1, -1, 0, 1}};
rdcNurb(3, uknot, 0, 1, 2, vknot, 0, 1, RDC_POINTW, (RDCvoid *)points, RDC_NULL);
```

rdcNurbMesh, rdcNurbMeshV

Name

rdcNurbMesh, rdcNurbMeshV - draw a mesh of Non-Uniform Rational B-spline patches

C Binding

RDCvoid rdcNurbMesh(RDCint *nu*,
RDCint *uorder*,
const RDCfloat **uknot*,
RDCfloat *umin*,
RDCfloat *umax*,
RDCint *nv*,
RDCint *vorder*,
const RDCfloat **vknot*,
RDCfloat *vmin*,
RDCfloat *vmax*,
...)

RDCvoid rdcNurbMeshV(RDCint *nu*,
RDCint *uorder*,
const RDCfloat **uknot*,
RDCfloat *umin*,
RDCfloat *umax*,
RDCint *nv*,
RDCint *vorder*,
const RDCfloat **vknot*,
RDCfloat *vmin*,
RDCfloat *vmax*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcNurbMesh *nu uorder uknot umin umax nv vorder vknot vmin vmax argumentlist*

Parameters

nu

the number of control points in the u direction. Must be greater than or equal to *uorder*.

uorder

the degree of the polynomial plus 1 in the u direction

uknot

array of knots in the u direction. There must be *nu* * *uorder* knots in non-decreasing order.

umin

the minimum value of u where surface is defined. Must be less than *umax* and greater than or equal to *uknot*[*uorder* - 1].

umax

the maximum value of u where surface is defined. Must be greater than *umin* and less than or equal to *uknot*[*nu*].

nv

the number of control points in the v direction. Must be greater than or equal to *vorder*.

vorder

the degree of the polynomial plus 1 in the v direction

vknot
array of knots in the v direction. There must be *nv* * *vorder* knots in non-decreasing order.

vmin
the minimum value of v where surface is defined. Must be less than *vmax* and greater than or equal to *vknot*[*vorder* - 1].

vmax
the maximum value of v where surface is defined. Must be greater than *vmin* and less than or equal to *vknot*[*nv*].

argumentlist
The argument list must contain at least point information, RDC_POINTW or RDC_POINT. If you use RDC_POINT, then w is assumed to be 1 for all points and the result is a Non-Uniform Non-Rational B-spline (NUNB?). There must be *nu* times *nv* points in the array. Also see Shading Variables.

Description

Draws the tensor product of the extended rational splines defined in the u and v directions. The order need not be the same for u and v. The surface is defined over the range [umin, umax] and [vmin, vmax]. Note the relationship between the min and max values and the values contained in the knot arrays.

Example

```
RDCfloat uknot = {0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4};
RDCfloat vknot = {0, 0, 1, 1};
RDCpoint points = {{1, 0, 0, 1}, {1, 1, 0, 1}, {0, 2, 0, 2},
    {-1, 1, 0, 1}, {-1, 0, 0, 1}, {-1, -1, 0, 1},
    {0, -2, 0, 2}, {1, -1, 0, 1}, {1, 0, 0, 1},
    {1, 0, -3, 1}, {1, 1, -3, 1}, {0, 2, -6, 2},
    {-1, 1, -3, 1}, {-1, 0, -3, 1}, {-1, -1, -3, 1},
    {0, -2, -6, 2}, {1, -1, -3, 1}, {1, 0, -3, 1}};
rdcNurb(9, 3, uknot, 0, 4, 2, 2, vknot, 0, 1, RDC_POINTW, (RDCvoid *)points, RDC_NULL);
```

rdcObjectBegin

Name

rdcObjectBegin - start defining an object

C Binding

RDCobject *rdcObjectBegin()

RDC Binding

rdcObjectBegin *number*

Parameters

number

Because object handles cannot be passed back when processing RDC bytestream, each object must be given a unique number between 0 and 65535. This number may be refernced later when calling the object. Low, sequential numbers are recommended as any drastic jump in object numbers will produce a warning message.

Description

Objects are not added to the scene when they are defined. They are merely recorded for later use. You may even define them before rdcSceneBegin or before a series of frame blocks. Save the object handle (or number in the case of RDC bytestream) to recall the object later and add it to the scene. When called, objects will inherit the prevailing matrix and attributes at the time.

Only primitives may be added to an object and they must all be of the same type. Objects may not be nested.

Example

```
RDCobject *object1 = rdcObjectBegin();
```

rdcObjectCall

Name

rdcObjectCall - draw a previously defined object

C Binding

RDCvoid rdcObjectCall(const RDCobject **handle*)

RDC Binding

rdcObjectCall *number*

Parameters

handle

a handle to a previously defined object

number

number of a previously defined object

Description

Objects are not added to the scene when they are defined. They are merely recorded for later use. You may even define them before rdcSceneBegin or before a series of frame blocks. When called, objects will inherit the prevailing matrix and attributes at the time.

Example

```
rdcObjectCall(object1);
```

rdcObjectEnd

Name
rdcObjectEnd - finish defining an object

C Binding
RDCvoid rdcObjectEnd()

RDC Binding
rdcObjectEnd

Description
Turns off the recording of an object started with rdcObjectBegin. It is an error to end an object without a previous, unmatched call to rdcObjectBegin.

Example
rdcObjectEnd();

rdcObjectName

Name

rdcObjectName - name an object

C Binding

RDCvoid rdcObjectName(const char **name*)

RDC Binding

rdcObjectName *name*

Parameters

name

name to give the object

Description

For debugging purposes, a name may be associated with an object. This name will be printed out with error messages which occur during rendering. This can help identify which object is causing the error.

Example

```
rdcObjectName("teapot");
```

rdcOpacity

Name

rdcOpacity - Set the current opacity

C Binding

RDCvoid rdcOpacity(const RDCfloat **color*)

RDC Binding

rdcOpacity *color*

Parameters

color

an array array of floats (usually 3) which define an opacity in the current color space

Description

Subsequent primitives will use this opacity as their surface opacity unless it is explicitly overriden in the primitive's argument list with RDC_OPACITY.

By default, the color space has three components: red, green, and blue. This may be changed by a call to rdcColorSpace. The default opacity is completely opaque: {1, 1, 1}. Note that opacity may vary in each color channel.

Example

```
RDCcolor half = {0.5, 0.5, 0.5};
rdcOpacity(half);
```

rdcOrientation

Name

rdcOrientation - set the rule for which side is outside

C Binding

RDCvoid rdcOrientation(RDCenum *orientation*)

RDC Binding

rdcOrientation *orientation*

Parameters

orientation

desired orientation of surfaces. Accepted values are RDC_LH, RDC_RH, RDC_INSIDE, RDC_OUTSIDE, and RDC_REVERSE.

Description

The outside face of a primitive is the side where the normal vector points outward. It is especially important to know which side is outside when using rdcBackface. It could also affect shading, but all of the built-in surface shaders are careful to shade both the inside and outside surfaces identically.

A reflection transformation, such as rdcScale(-1, 1, 1), reverses the handedness of the current coordinate space. The renderer automatically compensates for all such transformations and preserves the meaning of outside.

The orientation is RDC_LH by default to match the renderer's default coordinate system. You may explicitly set the orientation to RDC_LH or RDC_RH, but remember that the important thing is the relationship to the current coordinate system. Setting the orientation to RDC_OUTSIDE actually sets the orientation to RDC_LH or RDC_RH, whichever is the handedness of the current coordinate space. RDC_INSIDE does the opposite. RDC_REVERSE toggles the current orientation between RDC_LH and RDC_RH.

Example

```
rdcOrientation(RDC_OUTSIDE);
```

rdcOutputDisplay

Name

rdcOutputDisplay - elect to draw directly to the display

C Binding

RDCvoid rdcOutputDisplay(const char **windowname*)

RDC Binding

rdcOutputDisplay *windowname*

Parameters

windowname

name which will be displayed in the caption bar of the image window

Description

By default, images are rendered to the display in a window named "RenderDotC". Use this function to set the name in the caption bar of the image window.

Example

```
rdcOutputDisplay("Utah Teapot");
```

rdcOutputFile

Name

rdcOutputFile - direct file output

C Binding

RDCvoid rdcOutputFile(RDCenum *format*,
const char **filename*)

RDC Binding

rdcOutputFile *format filename*

Parameters

format

the file format of the output file. May be RDC_TIFF for image files, or RDC_BYTESTREAM for converting an RDC program to RDC bytestream.

filename

the name of the file to be saved

Description

There are two distinct purposes of this function.

First, is directing the renderer to save the image as a file in a particular graphics file format. Currently, the only supported file format is TIFF. Note that the free demo version is not capable of saving graphics files and must use rdcOutputDisplay.

Second, when linking with the RDC bytestream output library, this is the way to specify the file name for the RDC bytestream file. The default is to send it to stdout, which isn't very useful under Windows. Make sure this function is called before any other RDC functions to ensure the entire bytestream is saved in the same file.

Examples

```
rdcOutputFile(RDC_TIFF, "teapot.tif");
rdcOutputFile(RDC_BYTESTREAM, "teapot.rdc");
```

rdcOutputSamples

Name

rdcOutputSamples - select which samples should be drawn

C Binding

RDCvoid rdcOutputSamples(RDCenum *samples*)

RDC Binding

rdcOutputSamples *samples*

Parameters

samples

which samples should be output to the display or file. Accepted values are RDC_RGB, RDC_RGBA, RDC_RGBAZ, RDC_A, RDC_AZ, and RDC_Z.

Description

Any combination of color, alpha, and depth information may be selected for output. However, device limitations may result in something less. For instance, the display device (monitor) is capable of showing color or depth (as a greyscale), but not both at the same time. Whenever a device is incapable of granting your request, RGB will be given preference to A and Z, and A will be given preference to Z.

A TIFF file is capable of handling all of the combinations. However, each sample needs to have the same format. You may set the format of RGB and A with rdcColorQuantize. Likewise, the format of Z is set with rdcDepthQuantize. If samples of different formats are requested to be output, the lesser format is promoted to the greater format. Among integer formats, the one with the greater number of bits is the greater format. Floating point is considered greater than any integer format.

Example

```
rdcOutputSamples(RDC_RGBA);
```

rdcParaboloid, rdcParaboloidV

Name

rdcParaboloid, rdcParaboloidV - Draw a paraboloid

C Binding

```
RDCvoid rdcParaboloid(RDCfloat rmax,
    RDCfloat zmin,
    RDCfloat zmax,
    RDCfloat tmax,
    ...)
RDCvoid rdcParaboloidV(RDCfloat rmax,
    RDCfloat zmin,
    RDCfloat zmax,
    RDCfloat tmax,
    RDCint n,
    const RDCquark *quarks,
    RDCvoid *const *parms)
```

RDC Binding

rdcParaboloid *rmax zmin zmax tmax argumentlist*

Parameters

rmax

the radius at the top of the paraboloid (z = *zmax*)

zmin

where the bottom of the paraboloid will be truncated

zmax

where the top of the paraboloid will be truncated

tmax

sweep angle about Z-axis

argumentlist

see Shading Variables

Description

The apex of the paraboloid lies at the origin, opening upward around the Z-axis. If *zmin* is between 0 and *zmax*, then the apex of the paraboloid is cut off. *tmax* can vary from - 360 to +360 to render sections of the paraboloid.

Variables which vary over the surface (such as RDC_COLOR) must have four components.

Example

```
RDCcolor fourcolors[4] = {
    {0, 0.9, 0.1},
    {0, 0, 0.5},
    {0.3, 0, 0},
    {0.3, 0, 0.3}
};
rdcParaboloid(5.1, 0, 4, -180, RDC_COLOR, (RDCvoid *)fourcolors, RDC_NULL);
```

rdcPatch, rdcPatchV

Name

rdcPatch, rdcPatchV - draw a bilinear or bicubic patch

C Binding

RDCvoid rdcPatch(RDCenum *type*,
...)
RDCvoid rdcPatchV(RDCenum *type*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcPatch *type argumentlist*

Parameters

type

the type of patch. Accepted values are RDC_BILINEAR and RDC_BICUBIC.

argumentlist

The argument list must contain at least point information, RDC_POINT, RDC_POINTW or RDC_HEIGHT. Also see Shading Variables.

Description

It takes 4 points to specify a bilinear patch and 16 to specify a bicubic patch. The points are given in u major order (u varies faster than v). Note that the order for a bilinear patch is different than that of a polygon. Bicubic patches use the basis matrices specified by rdcBasis or rdcCustomBasis.

Example

```
RDCfloat corners[] = {1, 0, 0, 1};
rdcPatch(RDC_BILINEAR, RDC_HEIGHT, (RDCvoid *)corners, RDC_NULL);
```

rdcPatchMesh, rdcPatchMeshV

Name

rdcPatchMesh, rdcPatchMeshV - draw a mesh of bilinear or bicubic patches

C Binding

RDCvoid rdcPatchMesh(RDCenum *type*,
RDCint *nu*,
RDCenum *uwrap*,
RDCint *nv*,
RDCenum *vwrap*,
...)
RDCvoid rdcPatchMeshV(RDCenum *type*,
RDCint *nu*,
RDCenum *uwrap*,
RDCint *nv*,
RDCenum *vwrap*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcPatchMesh *type nu uwrap nv vwrap argumentlist*

Parameters

type
the type of patches in the mesh. Accepted values are RDC_BILINEAR and RDC_BICUBIC.

nu
the number of control points in the u direction

uwrap
whether the mesh wraps around in the u direction. Acceptable values are RDC_WRAP and RDC_NOWRAP.

nv
the number of control points in the v direction

vwrap
whether the mesh wraps around in the v direction. Acceptable values are RDC_WRAP and RDC_NOWRAP.

argumentlist
The argument list must contain at least point information, RDC_POINT, RDC_POINTW or RDC_HEIGHT. Also see Shading Variables.

Description

Each patch in the mesh will be rendered just as if it had been specified with rdcPatch. The number of control points is *nu* * *nv*. The points are given in u major order (u varies faster than v). Bicubic patches use the basis matrices specified by rdcBasis or rdcCustomBasis.

Patches may wrap around in the u direction, v direction, or both. When wrapping around, 1 to 3 rows (or columns) of control points from the beginning will be used again at the end. The actual number repeated depends on the step of the basis matrix.

The total number of patches depends on *type*, *uwrap*, *vwrap*, and the steps of the basis matrices.

Example

```
RDCpoint points[28];
RDCpoint two_pts[2];
RDCcolor six_colors[6];
rdcPatchMesh(RDC_BICUBIC, 7, RDC_NOWRAP, 4, RDC_NOWRAP,
    RDC_POINT, (RDCvoid *)points,
    RDC_PLANE, (RDCvoid *)two_pts,
    RDC_COLOR, (RDCvoid *)six_colors,
    RDC_NULL);
```

rdcPerspective

Name

rdcPerspective - transform by a perspective matrix

C Binding

RDCvoid rdcPerspective(RDCfloat *fov*)

RDC Binding

rdcPerspective *fov*

Parameters

fov

the field of view in degrees

Description

The current transformation matrix is premultiplied by a perspective matrix. The perspective matrix is formed from the field of view *fov*, near clipping plane at distance 1, and far clipping plane at RDC_INFINITY. Depth values between the clipping planes are projected to the range 0 to 1 in a nonlinear fashion.

You may achieve your own perspective view by using rdcViewIdentity and rdcPerspective. However, this is not recommended because of the poor, nonlinear mapping of depth values.

Example

```
rdcPerspective(15);
```

rdcPolygon, rdcPolygonV

Name

rdcPolygon, rdcPolygonV - draw a convex polygon

C Binding

RDCvoid rdcPolygon(RDCint *nverts*,
...)
RDCvoid rdcPolygonV(RDCint *nverts*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcPolygon *nverts argumentlist*

Parameters

nverts

the number of vertices in the polygon

argumentlist

The argument list must contain at least point information, RDC_POINT. There must be *nverts* points. Also see Shading Variables.

Description

The polygon must be planar and convex. For concave polygons or polygons with holes, use rdcGeneralPolygon. For non-planar quadrilaterals, see the bilinear variety of rdcPatch.

Example

```
RDCpoint points[] = {{0, 0, 0}, {0, 1, 0}, {1, 1, 0}, {1, 0, 0}};
rdcPolygon(4, RDC_POINT, (RDCvoid *)points, RDC_NULL);
```

rdcPolygonMesh, rdcPolygonMeshV

Name

rdcPolygonMesh, rdcPolygonMeshV - draw a mesh of convex polygons

C Binding

```
RDCvoid rdcPolygonMesh(RDCint npolys,
    const RDCint *nverts,
    const RDCint *verts,
    ...)
RDCvoid rdcPolygonMeshV(RDCint npolys,
    const RDCint *nverts,
    const RDCint *verts,
    RDCint n,
    const RDCquark *quarks,
    RDCvoid *const *parms)
```

RDC Binding

rdcPolygonMesh *npolys nverts verts argumentlist*

Parameters

npolys

number of polygons in the mesh

nverts

number of vertices in each polygon. There must be *npolys* entries in this array.

verts

indeces into the vertex array. There must be as many entries in this array as the sum of all the entries in *nverts*. Since polygons in the mesh may share vertices, a set of actual vertices need only be declared once (in the argument list). The vertices for each polygon are then specified as indeces into the actual vertex array.

argumentlist

The argument list must contain at least point information, RDC_POINT. There must be as many points as the largest value in *verts* plus one. Also see Shading Variables.

Description

All polygons in the mesh are drawn as if they had been specified individually with rdcPolygon. The polygons must all be planar and convex. For concave polygons or polygons with holes, use rdcGeneralPolygonMesh. For non-planar quadrilaterals, see the bilinear variety of rdcPatchMesh.

Example

```
RDCint nverts[] = {3, 3, 3};
RDCint verts[] = {0, 3, 2, 0, 1, 3, 1, 4, 3};
RDCpoint points[] = {{0, 1, 1}, {0, 3, 1}, {0, 0, 0}, {0, 2, 0}, {0, 4, 0}};
RDCcolor five_colors[] = {{0, 0.6, 0.1}, {0, 0.3, 0.1}, {0, 0, 0}, {0, 0.2, 0}, {0, 0.4, 0}};
rdcPolygon(3, nverts, verts,
    RDC_POINT, (RDCvoid *)points,
    RDC_COLOR, (RDCvoid *)five_colors,
    RDC_NULL);
```

rdcProcedural

Name

rdcProcedural - draw a procedural primitive

C Binding

RDCvoid rdcProcedural(RDCvoid **data*,
const RDCfloat **bound*,
RDCsubdivideFunc *subdivfunc*,
RDCfreeFunc *freefunc*)

RDC Binding

none

Parameters

data

a pointer to a block of user data

bound

the bounding box of the primitive in object space. All derived primitives must lie within this box.

subdivfunc

the procedure to subdivide the primitive into smaller procedural primitives or into any various non-procedural primitives. The required signature is:

RDCvoid subdividefunc(RDCvoid *data, RDCfloat area)

where data is the pointer to the user data and area is the on-screen area of the primitive in pixels.

freefunc

a procedure to free the data when no longer needed. The required signature is:

RDCvoid freefunc(RDCvoid *data)

where data is the pointer to the user data.

Description

Allows user to define a primitive by a procedure which is called during rendering.

The user specified bounding box is used to determine when the portion of the image containing the procedural primitive is being rendered. Then the user's subdivide function is called with the data pointer and on-screen area, in pixels, as arguments.

The subdivide function may call rdcProcedural again with smaller models, define other primitives, or both. The recursive nature of calling rdcProcedural from the subdivide function must terminate eventually, *even if the area is always RDC_INFINITY.* This is important for the case when you are generating an RDC bytestream and the actual on-screen area is unknown.

The free function is usually called immediately after the subdivide function. Therefore, the same data pointer should not be passed back to rdcProcedural from within the subdivide function. Otherwise, the free function will be called more than once on the same data pointer. This is fine if your free function does nothing, of course.

rdcProjectPoints

Name

rdcProjectPoints - project points from one coordinate space to another

C Binding

RDCpoint *rdcProjectPoints(RDCquark *fromspace*,
RDCquark *tospace*,
RDCint *n*,
RDCpoint **points*)

RDC Binding

none

Parameters

fromspace

quark representing the coordinate space to project from

tospace

quark representing the coordinate space to project to

n

the number of points being projected

points

an array of points to be projected

Description

Projects points from one coordinate space to another. If the projection is successful, *points* is returned, now containing the projected points. If the transformation is unsuccessful, for instance, because of a non-invertible matrix, NULL is returned.

The quarks *fromspace* and *tospace* may either be built-in coordinate spaces or user defined spaces or one of each. The built-in coordinate spaces are RDC_RASTER, RDC_IMAGE, RDC_EYE, RDC_WORLD, and RDC_MODEL. User defined coordinate spaces are created with rdcMarkSpace.

Example

```
RDCpoint five_pts[5];
rdcProjectPoints(RDC_EYE, myspace, 5, five_pts);
```

rdcQuark

Name
rdcQuark - define a new quark

C Binding
RDCquark rdcQuark(const char **string*)

RDC Binding
rdcQuark *string*

Parameters
string
the string which the new quark will represent

Description
This is how the user may extend the set of quarks by defining new ones. See the discussion on quarks.

Example
rdcQuark("GlowLight");

rdcRasterAspect

Name

rdcRasterAspect - specify the aspect ratio of the physical pixels

C Binding

RDCvoid rdcRasterApsect(RDCfloat *aspect*)

RDC Binding

rdcRasterAspect *aspect*

Parameters

aspect

the physical pixel aspect ratio (width divided by height)

Description

The default raster aspect ratio is 1, which represents square pixels. If you are rendering an image for a specific piece of hardware which does not have square pixels, set the raster aspect ratio to the actual pixel width divided by the actual pixel height.

Example

```
rdcRasterAspect(.667);
```

rdcRasterViewport

Name

rdcRasterViewport - set the viewport

C Binding

RDCvoid rdcRasterViewport(RDCint *left*,
RDCint *right*,
RDCint *top*,
RDCint *bot*)

RDC Binding

rdcRasterViewport *left right bottom top*

Parameters

left

the left edge of the viewport in raster coordinates

right

the right edge of the viewport in raster coordinates

top

the top edge of the viewport in raster coordinates

bot

the bottom edge of the viewport in raster coordinates

Description

The viewport is the rectangle on the display where the rendered image is shown (often called a "window", which used to mean something else). The total width of the image in pixels will be right - left + 1. For an image 640 pixels wide, use a value of *right* which is 639 greater than *left*.

left and *top* are usually both 0. When rendering to the display under Windows, they are both ignored and the viewport is opened in the default position with the given width and height. When saving to a TIFF, however, the *left* and *top* coordinates are correctly stored. It is then up to the tool which reads the TIFF file to correctly intrepret the data.

The default values are 0, 639, 0, and 479.

Example

rdcRasterViewport(0, 199, 0, 124);

rdcRotate

Name

rdcRotate - transform by a rotate matrix

C Binding

RDCvoid rdcRotate(RDCfloat *angle*,
RDCfloat *dx*,
RDCfloat *dy*,
RDCfloat *dz*)

RDC Binding

rdcRotate *angle dx dy dz*

Parameters

angle
the angle of rotation in degrees

dx
x component of the axis of rotation

dy
y component of the axis of rotation

dz
z component of the axis of rotation

Description

Premultiplies the current transformation matrix by a rotation matrix. The axis of rotation goes from the origin through the point (*dx*, *dy*, *dz*). The *angle* is in degrees. If the current coordinate space is left handed (the default), then the rotation follows the left-hand rule.

Example

```
rdcRotate(30, 1, 0, 0);
```

rdcSampleAdaptive

Name

rdcSampleAdaptive - select adaptive sampling

C Binding

RDCvoid rdcSampleAdaptive(RDCfloat *variation*)

RDC Binding

rdcSampleAdaptive *variation*

Parameters

variation

the maximum allowable estimated variance between the true color a pixel should be and the approximated color

Description

By default, adaptive sampling is off and four supersamples are taken at each pixel, as in rdcSampleUniform(2, 2). With adaptive sampling on, a different number of supersamples may be taken for each pixel, depending on the complexity of the scene at that pixel. Areas of constant color, for example, will need fewer samples than at an edge where color changes dramatically.

You specify how accurate the result must be with the argument *variation*. Assuming that colors range from 0 to 1 before quantization and 0 to 255 after, a variation of 1.0/255.0 means that the estimated color should be within one quantization level of the true color. Of course, if the renderer knew for sure what the true color was, it would use it. Instead, it uses statistical methods to achieve a 99% probability that the threshhold is met.

The good news is that adaptive sampling puts the effort where it is most needed. The bad news is that it takes some effort to determine that a pixel is uninteresting and needn't be sampled further. A minimum of 8 samples are taken per pixel. That's twice more than the default 2 by 2 uniform sampling. You will usually get better performance even with 3 by 3 uniform sampling. Consider adaptive sampling only if the quality is not sufficient.

Example

```
rdcSampleAdaptive(1.0/255.0);
```

rdcSampleJitter

Name

rdcSampleJitter - control random jitter applied to sample locations

C Binding

RDCvoid rdcSampleJitter(RDCfloat *ampl*)

RDC Binding

rdcSampleJitter *ampl*

Parameters

ampl

amplitude of the jitter sampling

Description

Used with uniform sampling to add some randomness to the location of the supersamples. Evenly spaced samples are subject to aliasing, even if applied at the subpixel level. Jitter sampling approximates a Poisson distribution, replacing aliasing with noise.

A supersample is initially placed at the center of a cell. The x and y positions are then jittered by a random number between -0.5 and +0.5 times the width of the cell times the jitter amplitude. The default amplitude is 0.5, which places the sample somewhere between 25% and 75% across the cell.

Example

```
rdcSampleJitter(1);
```

rdcSampleUniform

Name

rdcSampleUniform - Select uniform sampling

C Binding

RDCvoid rdcSampleUniform(RDCfloat *xsamples*,
RDCfloat *ysamples*)

RDC Binding

rdcSampleUniform *xsamples ysamples*

Parameters

xsamples

specifies the sampling rate in the x direction

ysamples

specifies the sampling rate in the y direction

Description

Uniform sampling causes supersamples to be taken the same way for each pixel. The number of samples per pixel is *xsamples* * *ysamples*. Each pixel is subdivided into an *xsamples* by *ysamples* grid of cells. Then, a sample is placed at the center of each cell and jittered by the amount specified in rdcSampleJitter.

The default for xsamples and ysamples is 2. Sampling rates less than 1 are clamped to 1. The alternative to uniform sampling is adaptive sampling.

Example

```
rdcSampleUniform(3, 3);
```

rdcScale

Name

rdcScale - Apply a scale transformation

C Binding

RDCvoid rdcScale(RDCfloat *sx*,
RDCfloat *sy*,
RDCfloat *sz*)

RDC Binding

rdcScale *sx sy sz*

Parameters

sx
the x component of scaling

sy
the y component of scaling

sz
the z component of scaling

Description

Premultiplies the current transformation matrix by a scale matrix. The x component of all points is multiplied by *sx*. Likewise for *sy* and *sz*.

A reflection scale such as rdcScale(-1, 1, 1) reverses the handedness of the current coordinate system. The handedness of the current coordinate system is tracked by the renderer so that the sense of "outside" is preserved.

Example

```
rdcScale(0.1, 2, 3.1);
```

rdcSceneBegin

Name

rdcSceneBegin - start describing scene

C Binding

RDCvoid rdcSceneBegin()

RDC Binding

rdcSceneBegin

Description

This is the dividing line between setting parameters and defining primitives. Parameters may only be set before rdcSceneBegin and primitives may only be defined between rdcSceneBegin and rdcSceneEnd. There should be exactly one scene block per frame. Scene blocks may not be nested.

Example

```
rdcSceneBegin();
```

rdcSceneEnd

Name

rdcSceneEnd - finish describing scene

C Binding

RDCvoid rdcSceneEnd()

RDC Binding

rdcSceneEnd

Description

When all of the primitives have been added to the scene, rdcSceneEnd performs the actual rendering. Rendering starts when you call rdcSceneEnd and finishes before it returns. It is an error to call rdcSceneEnd without a previous unmatched call to rdcSceneBegin or if there is an intervening call to a state changing function (i.e. rdcAttributePush) which has not been properly nested.

Example

```
rdcSceneEnd();
```

rdcShadingModel

Name

rdcShadingModel - select how polygons will be colored

C Binding

RDCvoid rdcShadingModel(RDCenum *type*)

RDC Binding

rdcShadingModel *type*

Parameters

type

method of coloring polygons. Accepted values are RDC_FLAT and RDC_GOURAUD.

Description

The renderer adaptively tesselates surfaces to polygonal facets which meet the constraints set by rdcShadingRate and rdcFlatness. Typically these facets are very small, on the order of 1/4 pixel. At such a small size, it is usually acceptable to compute a single color for the facet, since it will just be sampled and filtered with the neighboring polygons anyway.

However, if you loosen the constraints in rdcShadingRate and/or rdcFlatness, you may see the polygonal facets as large polygons in the image. A relatively inexpensive way to improve the quality of such an image is to use Gouraud shading.

By default, the shading model is RDC_FLAT, with a single color and opacity being used for a facet. With RDC_GOURAUD, four colors/opacities are computed for each facet and are then linearly interpolated across the face.

Example

```
rdcShadingModel(RDC_GOURAUD);
```

rdcShadingRate

Name

rdcShadingRate - set minimum tesselation threshhold for shading

C Binding

RDCvoid rdcShadingRate(RDCfloat *size*)

RDC Binding

rdcShadingRate *size*

Parameters

size

the area, in pixels, of the largest acceptable polygonal facet

Description

The renderer adaptively subdivides surfaces into polygonal facets, each of which is shaded once. It tries to make the facets equal in area in raster space, so that surfaces perpendicular to the viewing direction are subdivided more than those which are nearly parallel to the viewing direction. This puts the shading effort to best use.

rdcShadingRate sets the maximum allowable area in pixels for a polygonal facet. Surfaces will be shaded at least as often as the given size. The default value is 0.25, which means surfaces are shaded four times per pixel (the Nyquist limit). Note that this is *not* the same as the uniform sampling rate.

If your model consists entirely of constant or diffuse shaded polygons and ambient and/or distant lights, then it is perfectly logical to set the shading rate to RDC_INFINITY. This way, your polygon primitives will be shaded the fewest times without sacrificing quality (this sort of scene is rather crude to begin with).

The other constraint which affects facet size is rdcFlatness.

Example

```
rdcShadingRate(1);
```

rdcSkew

Name

rdcSkew - apply a skew transformation

C Binding

RDCvoid rdcSkew(RDCfloat *angle*,
RDCfloat *dx1*,
RDCfloat *dy1*,
RDCfloat *dz1*,
RDCfloat *dx2*,
RDCfloat *dy2*,
RDCfloat *dz2*)

RDC Binding

rdcSkew *angle dx1 dy1 dz1 dx2 dy2 dz2*

Parameters

angle
specifies the angle of skew in degrees

dx1
the x component of the source vector

dy1
the y component of the source vector

dz1
the z component of the source vector

dx2
the x component of the destination vector

dy2
the y component of the destination vector

dz2
the z component of the destination vector

Description

rdcSkew applies a skew transformation by shifting all points along lines parallel to (dx2, dy2, dz2). Points along (dx1, dy1, dz1) are mapped onto the vector (x, y, z), where angle specifies the angle between (dx1, dy1, dz1) and (x, y, z). It is an error to specify an angle that is greater than or equal to the angle between them. An error also occurs if a negative angle is less than 180° minus the angle between the two axes.

Example

```
rdcSkew(10, 1, 1, 1, 2, 1, 2);
```

rdcSphere, rdcSphereV

Name

rdcSphere, rdcSphereV - draw a sphere

C Binding

RDCvoid rdcSphere(RDCfloat *radius*,
RDCfloat *zmin*,
RDCfloat *zmax*,
RDCfloat *tmax*,
...)
RDCvoid rdcSphereV(RDCfloat *radius*,
RDCfloat *zmin*,
RDCfloat *zmax*,
RDCfloat *tmax*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcSphere *radius zmin zmax tmax argumentlist*

Parameters

radius
the radius of the sphere

zmin
if greater than -*radius*, where to truncate the bottom of the sphere

zmax
if less than *radius*, where to truncate the top of the sphere

tmax
sweep angle about Z-axis

argumentlist
see Shading Variables

Description

The sphere is center at the origin with the given *radius*. *tmax* ranges from -360 to 360. If *tmax* is anywhere in between, the sides are open. If *zmin* > -*radius* or *zmax* < *radius*, the bottom or top of the sphere is also open.

Example

rdcSphere(3, -2, 2.5, 360, RDC_NULL);

rdcSurface, rdcSurfaceV

Name

rdcSurface, rdcSurfaceV - specify surface type

C Binding

```
RDCvoid rdcSurface(RDCquark name,
    ...)
RDCvoid rdcSurfaceV(RDCquark name,
    RDCint n,
    const RDCquark *quarks,
    RDCvoid *const *parms)
```

RDC Binding

rdcSurface *name argumentlist*

Parameters

name

Name of the surface shader. Accepted values are RDC_CONSTANT, RDC_DIFFUSE, RDC_METAL and RDC_PLASTIC.

argumentlist

The argument list depends upon *name* as described below.

Description

rdcSurface applies a surface shader to subsequently defined primitives. The shader is evaluated at least once for each polygonal facet whose size is determined from rdcShadingRate and rdcFlatness.

Constant

```
rdcSurface(RDC_CONSTANT,
    RDC_NULL);
```

This shader takes no argument list (but it still must be terminated by RDC_NULL). The current color and opacity are used as the shaded output color without regard to any light sources.

Diffuse

```
RDCfloat ka = 1;
RDCfloat kd = 1;
rdcSurface(RDC_DIFFUSE,
    RDC_KA, (RDCvoid *)&ka,
    RDC_KD, (RDCvoid *)&kd,
    RDC_NULL);
```

This shader takes two arguments: the coefficients of ambient and diffuse lighting. Incident ambient light is scaled by RDC_KA, diffuse by RDC_KD. The default values are shown in the example above. Diffuse surfaces (also called Lambertian and matte) are not shiny at all. This shader does not take into account the position of the viewer, only the position of the surface with respect to the lighting conditions.

Metal

```
RDCfloat ka = 1;
RDCfloat ks = 1;
RDCfoat roughness = 0.1;
rdcSurface(RDC_METAL,
    RDC_KA, (RDCvoid *)&ka,
    RDC_KS, (RDCvoid *)&ks,
```

```
    RDC_ROUGHNESS, (RDCvoid *)&roughness,
    RDC_NULL);
```

This surface has only ambient and specular components. The coefficients of which are given by RDC_KA and RDC_KS, respectively. RDC_ROUGHNESS controls the sharpness of the specular highlight. Low values give a sharp highlight point. The above examples shows all the default values.

Plastic

```
RDCfloat ka = 1;
RDCfloat kd = 0.5;
RDCfloat ks = 0.5;
RDCfloat roughness = 0.1;
RDCcolor specularcolor = {1, 1, 1};
rdcSurface(RDC_PLASTIC,
    RDC_KA, (RDCvoid *)&ka,
    RDC_KD, (RDCvoid *)&kd,
    RDC_KS, (RDCvoid *)&ks,
    RDC_ROUGHNESS, (RDCvoid *)&roughness,
    RDC_SPECULARCOLOR, (RDCvoid *)specularcolor,
    RDC_NULL);
```

Plastic is the combination of all of the above shaders. It has ambient, diffuse, and specular components scaled by RDC_KA, RDC_KD, and RDC_KS, respectively. RDC_ROUGHNESS controls the sharpness of the specular highlight. Low values give a sharp highlight point. The color of the specular highlight on plastic is not affected by the surface color at all. Instead, it is the product of the light color and RDC_SPECULARCOLOR.

rdcTorus, rdcTorusV

Name

rdcTorus, rdcTorusV - draw a torus

C Binding

RDCvoid rdcTorus(RDCfloat *majrad*,
RDCfloat *minrad*,
RDCfloat *phimin*,
RDCfloat *phimax*,
RDCfloat *tmax*,
...)
RDCvoid rdcTorusV(RDCfloat *majrad*,
RDCfloat *minrad*,
RDCfloat *phimin*,
RDCfloat *phimax*,
RDCfloat *tmax*,
RDCint n,
const RDCquark *quarks,
RDCvoid *const *parms)

RDC Binding

rdcTorus *majrad minrad phimin phimax tmax argumentlist*

Parameters

majrad
the distance from the origin to the center of the tube

minrad
the distance from the center of the tube to the surface

phimin
when orbiting the minor radius, where to begin defining surface

phimax
when orbiting the minor radius, where to end defining surface

tmax
sweep angle about Z-axis

argumentlist
see Shading Variables

Description

The torus is defined in the X-Y plane around the origin. If the difference of *phimax* and *phimin* is less than 360, there will be an open groove running around the "length" of the torus. If the difference of *phimax* and *phimin* is greater than 360, *phimax* will be adjusted so that the difference is exactly 360. *tmax* ranges from -360 to 360. If *tmax* is anywhere in between, the length of the torus will not completely wrap around.

Example

rdcTorus(0.5, 1, 0, 360, 360, RDC_NULL);

rdcTranslate

Name

rdcTranslate - apply a translate transformation

C Binding

RDCvoid rdcTranslate(RDCfloat *dx*,
RDCfloat *dy*,
RDCfloat *dz*)

RDC Binding

rdcTranslate *dx dy dz*

Parameters

dx

x component of translation vector

dy

y component of translation vector

dz

z component of translation vector

Description

Premultiplies the current transformation matrix by a translation matrix. *dx* is added to the x component of all points. Likewise for *dy* and *dz*.

Example

```
rdcTranslate(1.2, 2, 3.1);
```

rdcTune

Name

rdcTune - tune rendering parameters for better performance

C Binding

RDCvoid rdcTune(RDCint *maxtess*,
RDCint *xbucket*,
RDCint *ybucket*)

RDC Binding

rdcTune *maxtess xbucket ybucket*

Parameters

maxtess
maximum allowable tesselation

xbucket
x dimension of buckets in pixels

ybucket
y dimension of buckets in pixels

Description

Tunes the rendering parameters to utilize your memory configuration for better rendering performance. *maxtess* has to do with the way the renderer adaptively tesselates primitives into polygonal facets. *xbucket* and *ybucket* are the sizes of the rectangular regions on the screen which the renderer processes one at a time.

If you notice thrashing on your hard disk, you should set these values down to minimize your rendering time. If you rarely need to swap memory pages to the hard disk, then setting theses values up may improve performance. This function has no effect on quality. The default values for *maxtess*, *xbucket*, and *ybucket* are 32, 12, and 12, respectively.

Example

```
rdcTune(16, 10, 10);
```

rdcViewIdentity

Name

rdcViewIdentity - use the identity viewing projection

C Binding

RDCvoid rdcViewIdentity()

RDC Binding

rdcViewIdentity

Description

The identity matrix may be used as the viewing projection matrix. If so, the user is responsible for mapping depth values to the range 0 to 1 and any projections such as perspective. This function saves the current coordinate space as the RDC_EYE space and resets the current transformation matrix to the identity.

The default view is orthographic.

Example

```
rdcViewIdentity();
```

rdcViewOrthographic

Name

rdcViewOrthographic - use an orthographic viewing projection

C Binding

RDCvoid rdcViewOrthographic()

RDC Binding

rdcViewOrthographic

Description

The orthographic projection simply maps depth values between the near and far clipping plane linearly to the range 0-1 and leaves x and y coordinates unchanged. It saves the current coordinate space as the RDC_EYE space and resets the current transformation matrix to the identity.

This is the default view. However, it is still a good idea to use this function to separate eye space from world space.

Example

```
rdcViewOrthographic();
```

rdcViewPerspective

Name

rdcViewPerspective - use a perspective viewing projection

C Binding

RDCvoid rdcViewPerspective(RDCfloat fov)

RDC Binding

rdcViewPerspective *fov*

Parameters

fov

the field of view in degrees

Description

The perspective projection maps the space filling the field of view to the square between -1 and 1 on the image plane. It also maps depth values between the near and far clipping plane linearly to the range 0-1. *fov* must be less than 180.

The default view is orthographic.

Example

```
rdcViewPerspective(40);
```