Implicit Fields

Introduction

This application note describes the C++ interface that allows users describe implicit fields ("level set" surfaces) in a plugin. These implicit fields may be used with either RiBlobby to describe an implicit surface or a volume, or with RiVolume to describe a volume.

RiBlobby

This description assumes familiarity with the Blobby Implicit Surfaces section of the documentation.

RiBlobby's code array now understands an additional primitive field opcode with numeric value 1004. The new opcode takes five operands:

1. The index in the strings array of the filename of the plugin.
2. The number of floating point arguments to pass to the plugin.
3. The index in the floats array of the start of the block of floating point arguments.
4. The number of string arguments to pass to the plugin.
5. The index in the strings array of the start of the block of string arguments.

Plugin filenames are looked up using the same path used for RiProcDynamicLoad (i.e. use RiOption("searchpath","procedural", ...); to set the path.)

Because Implicit Field plugins act as primitive fields under RiBlobby they share all of the flexibility of existing primitive field opcodes. They can all blend with each other and have vertex values in exactly the same ways.

RiVolume

Implicit field plugins may be used with RiVolume by using a URI for the "type" parameter, where the URI scheme is the word"blobbydso" and the remainder of the URI specifies the path to the plugin. The plugin filename will be resolved against Option "searchpath" "procedural".

Float parameters to the plugin can be passed by using a primitive variable with the name "blobbydso:floatargs", which must be a"constant float" array; the size of the array in the declaration determines the number of parameters.

Similarly, string parameters to the plugin can be passed by using a primitive variable with the name "blobbydso:stringargs", which must be a "constant string" array; the size of the array implicitly determines the number of parameters.

The threshold used to determine the limit surface of the plugin will be the default threshold used by RiBlobby; this can be overridden by specifying a value for "constant float blobbydso:threshold".

The Plugin Interface

For efficiency, the renderer depends on getting quite a lot of information from the primitives, so this interface is fairly elaborate. It is possible to get away with just writing a constructor and methods that compute the field value and (for the case of surfaces) its gradient; the price you pay is decreased efficiency and accuracy.

Every Implicit Field plugin must have an entry point declared:

```
    extern "C" ImplicitField *ImplicitFieldNew(
        int nfloat, const RtFloat *float0, const float *float1,
        int nstring, const RtString *string);
```

The arguments are just the floating point and string parameters specified when the plugin was mentioned in the code of the RiBlobby call. For motion blur purposes, float0 and float1 give the floating point values at shutter open and shutter close. If RiBlobby orRiVolume was not called in a motion block, they are identical. It is guaranteed that no argument data will be freed during the lifetime of the plugin. In addition, the plugin must define a variable:

```
    extern "C" const int ImplicitFieldVersion=4;
```

specifying that this plugin implements version 4 of the field plugin interface. The value of this variable is used by the renderer to ensure backwards binary compatibility without the need to recompile your plugin. As an aid to the renderer, it is highly encouraged that your plugin use the FIELDCREATE macro from ImplicitField.h rather than defining this variable explicitly. The macro defines theImplicitFieldVersion appropriately and emits the function header for ImplicitFieldNew. (The example below indicates usage.)

The return value of ImplicitFieldNew is an instance of a subclass of class ImplicitField, whose definition is inImplicitField.h in the PRMan include directory.

```
class ImplicitVertexValue {
private:
    ImplicitVertexValue(const ImplicitVertexValue &);
    ImplicitVertexValue &operator=(const ImplicitVertexValue &);

public:
    ImplicitVertexValue() {}
```

```
    virtual ~ImplicitVertexValue() {}

    virtual void GetVertexValue(RtFloat *result, const RtPoint p) = 0;

    virtual void GetVertexValueFiltered(RtFloat *result, const RtPoint p,
        const RtPoint dPdu, const RtPoint dPdv, const RtPoint dPdw) {
        GetVertexValue(result, p);
    }

    virtual void GetVertexValueMultiple(int neval, RtFloat *result,
        int resultstride, const RtPoint *p) {
        for (int i = 0; i < neval; ++i) {
            GetVertexValue(result, *p++);
            result += resultstride;
        }
    }

    virtual void GetVertexValueMultipleFiltered(int neval, RtFloat *result,
        int resultstride, const RtPoint *p, const RtPoint *dPdu,
        const RtPoint *dPdv, const RtPoint *dPdw) {
        for (int i = 0; i < neval; ++i) {
            GetVertexValueFiltered(result, *p++, *dPdu++, *dPdv++, *dPdw++);
            result += resultstride;
        }
    }
};

class ImplicitField{
public:

    RtBound bbox;

private:
    ImplicitField(const ImplicitField &);
    ImplicitField &operator=(const ImplicitField &);

public:
    ImplicitField(){}
    virtual ~ImplicitField(){}

    virtual RtFloat Eval(const RtPoint p) = 0;

    virtual RtFloat EvalFiltered(const RtPoint p, const RtPoint dPdu,
        const RtPoint dPdv, const RtPoint dPdw) {
        return Eval(p);
    }

    virtual void EvalMultiple(int neval, float *result, int resultstride,
        const RtPoint *p) {
        for (int i = 0; i < neval; ++i) {
            *result = Eval(*p++);
            result += resultstride;
        }
    }

    virtual void EvalMultipleFiltered(int neval, float *result,
        int resultstride, const RtPoint *p, const RtPoint *dPdu,
        const RtPoint *dPdv, const RtPoint *dPdw) {
        EvalMultiple(neval, result, resultstride, p);
    }

    virtual void GradientEval(RtPoint result, const RtPoint p) = 0;

    virtual void GradientEvalFiltered(RtPoint result, const RtPoint p,
        const RtPoint dPdu, const RtPoint dPdv, const RtPoint dPdw) {
        GradientEval(result, p);
    }

    virtual void GradientEvalMultiple(int neval, RtPoint *result,
        const RtPoint *p) {
        for (int i = 0; i < neval; ++i) {
```

```
            GradientEval(*result++, *p++);
        }
    }

    virtual void GradientEvalMultipleFiltered(int neval, RtPoint *result,
        const RtPoint *p, const RtPoint *dPdu, const RtPoint *dPdv,
        const RtPoint *dPdw) {
        GradientEvalMultiple(neval, result, p);
    }

    virtual void Range(RtInterval result, const RtPoint corners[8],
        const RtVolumeHandle h){
        result[0] = -1e30f;
        result[1] = 1e30f;
    }

    virtual bool ShouldSplit() {
        return false;
    }

    virtual void Split(std::vector<ImplicitField *> &children) {
    }

    virtual void Motion(RtPoint result, const RtPoint p) {
        result[0] = 0.0f;
        result[1] = 0.0f;
        result[2] = 0.0f;
    }

    virtual void MotionFiltered(RtPoint result, const RtPoint p,
        const RtPoint dPdu, const RtPoint dPdv, const RtPoint dPdw) {
        Motion(result, p);
    }

    virtual void MotionMultiple(int neval, RtPoint *result, const RtPoint *p) {
        for (int i = 0; i < neval; ++i) {
            Motion(*result++, *p++);
        }
    }

    virtual void MotionMultipleFiltered(int neval, RtPoint *result,
        const RtPoint *p, const RtPoint *dPdu, const RtPoint *dPdv,
        const RtPoint *dPdw) {
        MotionMultiple(neval, result, p);
    }

    virtual void BoxMotion(RtBound result, const RtBound b){
        for (int i = 0; i < 6; i++) {
            result[i] = b[i];
        }
    }

    virtual void VolumeCompleted(const RtVolumeHandle h) {
    }

    virtual ImplicitVertexValue *CreateVertexValue(const RtToken name,
        int nvalue) {
        return 0;
    }

    virtual float MinimumVoxelSize(const RtPoint corners[8]) {
        return 0.0f;
    }
};
```

The bbox field must be filled in the constructor with a bounding box in the object coordinate system that is active at the call to RiBlobby, at shutter open, outside of which the field value is guaranteed to be identically lower than the field function threshold. Note that typeRtBound is defined in ri.h to be an array of 6 floats. bbox[0], bbox[2], and bbox[4] are the lower bounds on x, y, and z, andbbox[1], bbox[3], and bbox[5] are the upper bounds.

The methods are:

```
RtFloat Eval(const RtPoint p)

RtFloat EvalFiltered(const RtPoint p, const RtPoint dPdu,
     const RtPoint dPdv, const RtPoint dPdw)

void EvalMultiple(int neval, float *result, int resultstride,
     const RtPoint *p)

void EvalMultipleFiltered(int neval, float *result,
     int resultstride, const RtPoint *p, const RtPoint *dPdu,
     const RtPoint *dPdv, const RtPoint *dPdw)
```

Eval and its variants return the implicit function field value at a point p, in object coordinates, at shutter open time. At minimum, your subclass must provide an implementation of Eval; the base class implementation provides default implementations of the other three functions in terms of Eval. However, your plugin can also override EvalFiltered andEvalMultipleFiltered functions, which provide three derivatives along with the point of evaluation in object coordinates. These derivatives can allow your plugin to perform a field function evaluation filtered over an region, which can greatly aid with anti-aliasing.

The two Multiple variants (EvalMultiple and EvalMultipleFiltered) are preferentially used by the renderer over the single evaluation and may allow your plugin to amortize the cost of some setup over multiple points of evaluation. When using these variants, the DSO is required to skip by resultstride when storing values in result.

```
void GradientEval(RtPoint result, const RtPoint p)

void GradientEvalFiltered(RtPoint result, const RtPoint p,
    const RtPoint dPdu, const RtPoint dPdv, const RtPoint dPdw)

void GradientEvalMultiple(int neval, RtPoint *result,
    const RtPoint *p)

void GradientEvalMultipleFiltered(int neval, RtPoint *result,
    const RtPoint *p, const RtPoint *dPdu, const RtPoint *dPdv,
    const RtPoint *dPdw)
```

Gradient and its variants return the field gradient at a point p, in object coordinates, at shutter open time. This information is primarily used to compute normals for surfaces defined by RiBlobby. If the plugin is used only for volumes, this entry point will not be used by the renderer (because the normals are always zero). Your DSO is still required to provide an implementation of at least GradientEval, and setting the return value to zero is sufficient.

Similar to Eval, your plugin can override the Filtered variants of Gradient to perform gradient evaluation filtered over a region, and can override the Multiple variants for better efficiency.

```
void Range(RtInterval result, const RtPoint corners[8], RtVolumeHandle h)
```

Range returns in result the bounds of the field function values inside the region of space defined by the given corners in object space, at shutter open. The corners may not necessarily define an aligned bounding box.

While implementing this method is optional, it is **highly encouraged** that your plugin implement this function, as it allows the renderer to perform culling operations on large regions of space, which can have a dramatic impact on execution speed. Specifically, the renderer performs at least the following optimizations based on Range:

- For surfaces defined by RiBlobby, if the interval returned by Range does not include the blobby limit threshold, the entire region defined by the corners is considered empty and will be culled.
- For volumes defined by RiBlobby or RiVolume, if the interval returned by Range is less than (or equal to) the threshold (result[1] <= threshold), the entire region defined by the corners is considered to be outside the volume and will be culled.
- If the range interval is identical on the region, the renderer may perform optimizations assuming that the region has a constant field function (such as, but not limited to, skipping calling Eval on the region).

The default base-class implementation stores result[0]=-1e30 and result[1]=1e30, resulting in exhaustive evaluation of the field function in the entire region, as no region will be considered to be trivially culled.

The volume handle h identifies the volume. The same value will later be passed to a call of VolumeCompleted.

```
bool ShouldSplit()

void Split(std::vector<ImplicitField *> &children)
```

ShouldSplit and Split allows an instance of ImplicitField to split into children instances (which are also themselves instances of ImplicitField). This mechanism is very similar to how RiProcedural operates, and has a range of possible uses.

For example, as an alternative to letting the renderer probe for "interesting" regions of space via repeated calls to Range, your plugin may instead (after interrogating some offline storage) already have better knowledge of bounding boxes that contain non-zero field function values, and would like to direct the renderer to focus its attention directly on those bounding boxes. The plugin can accomplish this by implementing ShouldSplit and Split; the latter would return a child or multiple children whose bbox fields are potentially much smaller subregions of the parent. The renderer will treat each child as its own separate ImplicitField instance and proceed to call Range, Eval on each.

ShouldSplit is called at least once, usually prior to Range. Currently, these two functions are only used only with volumes defined by RiVolume.

```
void Motion(RtPoint result, const RtPoint p)

void MotionFiltered(RtPoint result, const RtPoint p,
     const RtPoint dPdu, const RtPoint dPdv, const RtPoint dPdw)

void MotionMultiple(int neval, RtPoint *result, const RtPoint *p)

void MotionMultipleFiltered(int neval, RtPoint *result,
     const RtPoint *p, const RtPoint *dPdu, const RtPoint *dPdv,
     const RtPoint *dPdw)
```

Motion and its variants compute the motion blur of a point in space. Given a point p, in object coordinates, at shutter open time, the return value is how much that point should move between shutter open and shutter close. For the purposes of the renderer this is considered deformation motion blur (i.e. your plugin is not responsible for computing any transform blur applied to the RiBlobby or RiVolume primitives).

The default base-class implementation assumes no motion and sets result to (0,0,0). If your plugin implements aMotion that doesn't return a zero result, your plugin should also implement BoxMotion, otherwise bounding box artifacts will occur, due to incorrect bounds in the renderer.

Similar to Eval, your plugin can override the Filtered variants of Motion to perform motion vector evaluation filtered over a region, and can override the Multiple variants for better efficiency.

```
void BoxMotion(RtBound result, const RtBound b)
```

BoxMotion computes the motion blur of an object-aligned bounding box in space. Given as an input the bounding box b, which defines a bounding box at shutter open space, it should store in result the corresponding bounding box at shutter close.

The default base-class implementation just copies b to result, corresponding to the default base-class implementation ofMotion (i.e: no motion blur takes place). If your plugin implements a non-trivial Motion then it is expected that your plugin also implements a BoxMotion to match. That is: for any point p inside the input bounding box b, the motion-blurred result from calling Motion on p should be within the bounding box returned by BoxMotion. If this contract is violated, you may find bounding box artifacts (tearing at bucket boundaries, etc.) in your rendered result.

```
void VolumeCompleted(RtVolumeHandle h)
```

VolumeCompleted is a courtesy callback, hinting that the renderer has finished processing all points inside the volume with the given handle, so that the plugin can discard data that it no longer needs. UsingVolumeCompleted is a little tricky: PRMan calls Range with a particular RtVolumeHandle when it starts to work on a part of the level-set, and calls VolumeCompleted with the same handle when it's done. But it may in the interim have subdivided and called Range on smaller contained volumes in which it may maintain an interest after it has called VolumeCompleted on the parent volume. The handle passed toVolumeCompleted may be reused in a subsequent call to Range, but it will never ambiguously identify two volumes in which PRMan simultaneously maintains an interest.

```
ImplicitVertexValue *CreateVertexValue(const RtToken name, int nvalue)
```

ImplicitVertexValue informs the plugin of a vertex variable declaration, asking that the plugin provide PRMan with an entry point that evaluates the variable. Arguments are the full name of a vertex variable (including inline type declaration, e.g. "color Cs"), and the number of float components it has: 1 for scalars or 3 for point types. If your plugin chooses to evaluate this variable your plugin must allocate (using C++'s new operator) and return an instance of a subclass ofImplicitVertexValue. PRMan will call delete on the result when it is done with it. If *name* is unknown to the plugin the call should return NULL. (The base-class implementation always returns NULL.)

The ImplicitVertexValue class itself has one required virtual method and three variants on that method.GetVertexValue(RtFloat *result, const RtPoint p) is required and should be defined by the plugin to store in result the value of the named vertex variable, evaluated at point p in object space. Note that on entry to the function, the result parameter may already be initialized: it may store the value given to the vertex variable directly in theRiBlobby call or RiVolume call.

The variants to ImplicitVertexValue::GetVertexValue are similar to the variants to ImplicitField::Eval: your plugin can override the Filtered variants to perform vertex value evaluation filtered over a region, and can override the Multiple variants for better efficiency.

```
float MinimumVoxelSize(const RtPoint corners[8])
```

The MinimumVoxelSize callback allows the plugin to hint at the minimum size of a voxel in object space over a region of space delineated by corners. This information can be used by the renderer as a way of avoiding potential overdicing. Specifically, it will be called in PRMan if Attribute "dice" "float minlength" is set to -1. The return value will be interpreted as a length in object space, and the renderer will strive not to deliver any micro-voxels with any dimensions smaller than this length (even if those micro-voxels are very close to the camera).

Compiling Your Plugin

Using g++ on Linux, if your plugin is in a file called field.cpp, you can compile it by typing the following (augmented, of course by whatever other flags and filenames your code needs to compile):

```
g++ -I$RMANTREE/include -fPIC -shared -o field.so field.cpp
```

Other compilers and other systems will doubtless require other procedures.

Field Plugins Supplied with PRMan

PRMan ships with several implicit field plugins that offer support for simple primitives as well as several useful file formats.

Example

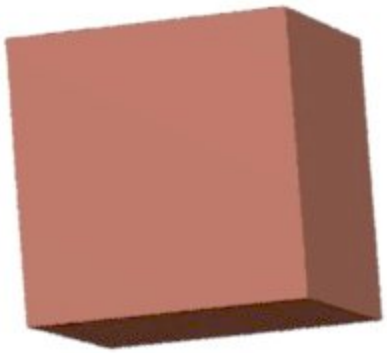
Here is a plugin for a field function whose level sets are cubes centered at the origin. The field cross-section is the same as that ofRiBlobby's sphere primitives, to make it blend nicely in compound objects. Note that this plugin implements the bare minimum required of a plugin: it does not support deformation motion blur or primitive variable evaluations, and it does not perform any filtered evaluations for anti-aliasing.

```
#include <ImplicitField.h>
class Cube: public ImplicitField{
public:
    Cube();
    virtual ~Cube();
    virtual RtFloat Eval(const RtPoint p);
    virtual void GradientEval(RtPoint grad, const RtPoint p);
    virtual void Range(RtInterval r, const RtPoint corners[8],
        RtVolumeHandle h);
};
Cube::Cube(){
    bbox[0]=-1.;
    bbox[1]=1.;
    bbox[2]=-1.;
    bbox[3]=1.;
    bbox[4]=-1.;
    bbox[5]=1.;
}
/*
 * This is the same field falloff (as a function of the
 * square of distance from the center) that RiBlobby uses
 * for its primitive blobs.
 * It has
 *    geoff(-1)=0 geoff'(-1)=0 geoff"(-1)=0
 *      geoff( 0)=1 geoff'(0)=0
 *    geoff( 1)=0 geoff'( 1)=0 geoff"( 1)=0
 */
static float geoff(float r2){
    if(r2>=1.f) return 0.f;
    return ((3.f-r2)*r2-3.f)*r2+1.f;
}
/*
 * d geoff(r2)
 * -----------
 *    d r2
 */
static float dgeoff(float r2){
    if(r2>=1.f) return 0.f;
    return (6.f-3.f*r2)*r2-3.f;
}
/*
 * geoff(max(x^2, y^2, z^2))
 */
float Cube::Eval(const RtPoint p){
    RtPoint sq;
    float r2;

    sq[0]=p[0]*p[0];
    sq[1]=p[1]*p[1];
    sq[2]=p[2]*p[2];
    if(sq[0]>sq[1]) r2=sq[0]>sq[2]?sq[0]:sq[2];
    else r2=sq[1]>sq[2]?sq[1]:sq[2];
    return geoff(r2);
}
void Cube::GradientEval(RtPoint grad, const RtPoint p){
```

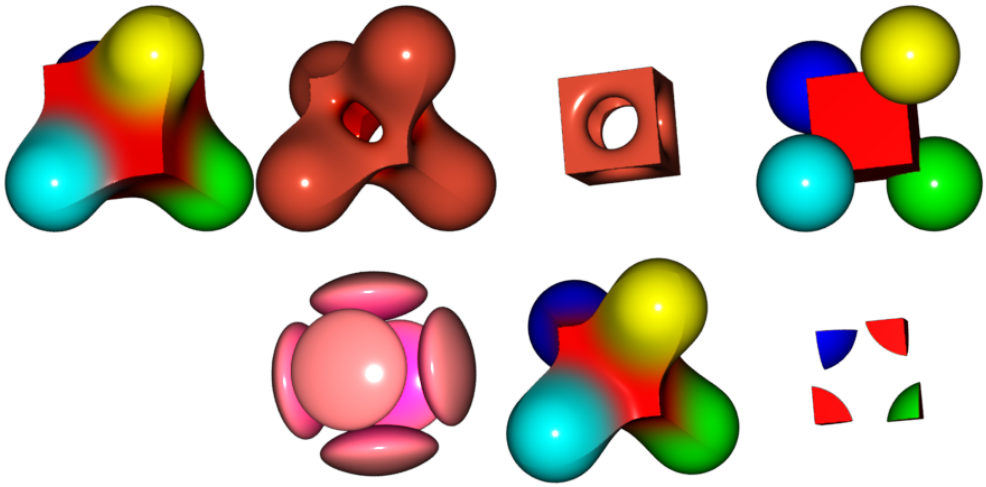
```
    RtPoint sq;

    grad[0]=0.;
    grad[1]=0.;
    grad[2]=0.;
    sq[0]=p[0]*p[0];
    sq[1]=p[1]*p[1];
    sq[2]=p[2]*p[2];
    if(sq[0]>sq[1]){
        if(sq[0]>sq[2]) grad[0]=2.*p[0]*dgeoff(sq[0]);
        else grad[2]=2.*p[2]*dgeoff(sq[2]);
    }
    else if(sq[1]>sq[2])
        grad[1]=2.*p[1]*dgeoff(sq[1]);
    else
        grad[2]=2.*p[2]*dgeoff(sq[2]);
}
void isq(RtInterval sq, RtInterval x){
    if(x[0]>=0){
        sq[0]=x[0]*x[0];
        sq[1]=x[1]*x[1];
    }
    else if(x[1]<=0){
        sq[0]=x[1]*x[1];
        sq[1]=x[0]*x[0];
    }
    else{
        sq[0]=0;
        sq[1]=-x[0]>x[1]?x[0]*x[0]:x[1]*x[1];
    }
}
void imax(RtInterval max, RtInterval a, RtInterval b){
    max[0]=b[0]>a[0]?b[0]:a[0];
    max[1]=b[1]>a[1]?b[1]:a[1];
}
void igeoff(RtInterval g, RtInterval r2){
    g[0]=geoff(r2[1]);
    g[1]=geoff(r2[0]);
}
void Cube::Range(RtInterval val, const RtPoint corners[8],
        RtVolumeHandle h){
    RtInterval r, x, y, z, xsq, ysq, zsq, maxxy, maxxyz;
    int i;

    x[0]=x[1]=corners[0][0];
    y[0]=y[1]=corners[0][0];
    z[0]=z[1]=corners[0][0];
    for(i=0;i!=8;i++){
        if(corners[i][0]<x[0]) x[0]=corners[i][0];
        if(corners[i][0]>x[1]) x[1]=corners[i][0];
        if(corners[i][1]<y[0]) y[0]=corners[i][1];
        if(corners[i][1]>y[1]) y[1]=corners[i][1];
        if(corners[i][2]<z[0]) z[0]=corners[i][2];
        if(corners[i][2]>z[1]) z[1]=corners[i][2];
    }
    isq(xsq, x);
    isq(ysq, y);
    isq(zsq, z);
    imax(maxxy, xsq, ysq);
    imax(maxxyz, maxxy, zsq);
    igeoff(val, maxxyz);
}
Cube::~Cube(){}
FIELDCREATE{
    return new Cube();
}
```

Here is a RIB file that uses the plugin, and the resulting image:

```
FrameBegin 0
    Display "cube.tif" "tiff" "rgba"
    Format 200 200 1
    Clipping 1e-3 1e5
    Projection "perspective" "fov" 3
    Hider "raytrace" "int incremental" [0] "int minsamples" [4] "int maxsamples" [16]
    Integrator "PxrDefault" "default"
    Translate 0 0 27
    DisplayFilter "PxrBackgroundDisplayFilter" "background" "color backgroundColor" [1 1 1]
    WorldBegin
        Bxdf "PxrDisney" "PxrDisney1" "color baseColor" [.9 .3 .2]
        Sides 2
        Rotate 10 0 1 0
        Rotate 10 1 1 1
        Blobby 1 [
            1004  0 0 0 0 0
                ]
        [0]
        ["cube"]
WorldEnd
FrameEnd
```



Just for amusement, here is another picture made using the same cube in various more complicated objects:



And the RIB file that made them:

```
##RenderMan RIB
version 3.04
FrameBegin 0
Option "ribparse" "string varsubst" ["$"]
 Display "${RMANREGRESS_TESTOBJDIR}/cube2.tif" "tiff" "rgba"
 Format 1200 600 1
 Projection "perspective" "uniform float fov" [15]
 Hider "raytrace"
 "int incremental" [0] "int minsamples" [4] "int maxsamples" [16]
 Integrator "PxrDefault" "default"
 Translate 0 0 27
 DisplayFilter "PxrBackgroundDisplayFilter" "background" "color backgroundColor" [.5 .5 .5]
 WorldBegin

 Pattern "PxrPrimvar" "PxrPrimvar1" "string varname" "Cs"
 "string type" "color"
 Bxdf "PxrDisney" "PxrDisney1"
 "reference color baseColor" ["PxrPrimvar1:resultRGB"]
 Sides 2

# (Sphere(2.4, (0, 0, 0)) - Cube()) - 0.1
 TransformBegin
 Translate -1.3 -1.3 0
 Rotate 10 0 1 0
 Rotate 10 1 1 1
 Blobby 3 [1001 0 1004 0 0 16 0 0 1000 16 4 0 1 4 3 2] [2.4 0 0 0 0 2.4 0 0 0 0 2.4 0 0 0 0 1 0.1] ["cube"] "Cs"
[1 0.5 0.5 0 1 0 0 0 1] "uniform color Os" [0.25 0.25 0.25]
 TransformEnd
 TransformBegin
 Translate 1.3 -1.3 0
 Rotate 10 0 1 0
 Rotate 10 1 1 1

# Cube() + (Sphere(1, (.6, -.6, .6)) +
# Sphere(1, (-.6, .6, .6)) +
# Sphere(1, (.6, .6, -.6)) +
# Sphere(1, (-.6, -.6, -.6)))
 Blobby 5 [1004 0 0 0 0 0 1001 0 1001 16 1001 32 1001 48 0 4 1 2 3 4 0 2 0 5] [1 0 0 0 0 1 0 0 0 0 1 0 0.6 -0.6
0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6 0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 0.6 0.6 -0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6
-0.6 -0.6 1] ["cube"] "Cs" [1 0 0 0 1 0 0 0 1 1 1 0 0 1 1] "Os" [1 1 1 0.5 0.5 0.5 0.75 0.75 0.75 0.5 0.5 0.5
0.75 0.75 0.75]
 TransformEnd

TransformBegin
 Translate -1.3 1.3 0
 Rotate 10 0 1 0
 Rotate 10 1 1 1
# (Cube() + (Sphere(1, (.6, -.6, .6)) +
# Sphere(1, (-.6, .6, .6)) +
# Sphere(1, (.6, .6, -.6)) +
# Sphere(1, (-.6, -.6, -.6)))) - Sphere(.75, (0, 0, 0))

Blobby 6 [1004 0 0 0 0 0 1001 0 1001 16 1001 32 1001 48 1001 64 0 2 1 2 0 2 6 3 0 2 7 4 0 2 0 8 4 9 5] [1 0 0 0 0
1 0 0 0 0 1 0 0.6 -0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6 0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 0.6 0.6 -0.6 1 1 0 0
0 0 1 0 0 0 0 1 0 -0.6 -0.6 -0.6 1 0.75 0 0 0 0 0.75 0 0 0 0 0.75 0 0 0 0 1] ["cube"] "Cs" [0.9 0.3 0.2 0.9 0.3
0.2 0.9 0.3 0.2 0.9 0.3 0.2 0.9 0.3 0.2 1.0 1.0 1.0]
 TransformEnd
 TransformBegin
 Translate 1.3 1.3 0
 Rotate 10 0 1 0
 Rotate 10 1 1 1
# Cube() - Sphere(.75, (0,0,0))
 Blobby 2 [1004 0 0 0 0 0 1001 0 4 0 1] [0.75 0 0 0 0 0.75 0 0 0 0 0.75 0 0 0 0 1] ["cube"]
 "uniform color Cs" [.9 .3 .2]
 TransformEnd


 TransformBegin
 Translate 3.9 -1.3 0
 Rotate 10 0 1 0
 Rotate 10 1 1 1

# min(Cube(), max(Sphere(1, (.6, -.6, .6)),
# Sphere(1, (-.6, .6, .6)),
# Sphere(1, (.6, .6, -.6)),
# Sphere(1, (-.6, -.6, -.6)))
 Blobby 5 [1004 0 0 0 0 0 1001 0 1001 16 1001 32 1001 48 2 4 1 2 3 4 3 2 0 5] [1 0 0 0 0 1 0 0 0 0 1 0 0.6 -0.6
0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6 0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 0.6 0.6 -0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6
-0.6 -0.6 1] ["cube"] "Cs" [1 0 0 0 1 0 0 0 1 1 1 0 0 1 1] "Os" [1 1 1 0.5 0.5 0.5 0.75 0.75 0.75 0.5 0.5 0.5
0.75 0.75 0.75]
 TransformEnd
```

```
TransformBegin
 Translate 3.9 1.3 0
 Rotate 10 0 1 0
 Rotate 10 1 1 1

# max(Cube(), max(Sphere(1, (.6, -.6, .6)),
# Sphere(1, (-.6, .6, .6)),
# Sphere(1, (.6, .6, -.6)),
# Sphere(1, (-.6, -.6, -.6)))
 Blobby 5 [1004 0 0 0 0 0 1001 0 1001 16 1001 32 1001 48 2 4 1 2 3 4 2 2 0 5] [1 0 0 0 0 1 0 0 0 0 1 0 0.6 -0.6
0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6 0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 0.6 0.6 -0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6
-0.6 -0.6 1] ["cube"] "Cs" [1 0 0 0 1 0 0 0 1 1 1 0 0 1 1] "Os" [1 1 1 0.5 0.5 0.5 0.75 0.75 0.75 0.5 0.5 0.5
0.75 0.75 0.75]
 TransformEnd

TransformBegin
 Translate -3.9 1.3 0
 Rotate 10 0 1 0
 Rotate 10 1 1 1

# (2 * Cube() + (Sphere(1, (.6, -.6, .6)) +
# Sphere(1, (-.6, .6, .6)) +
# Sphere(1, (.6, .6, -.6)) +
# Sphere(1, (-.6, -.6, -.6)))) - Sphere(.75, (0, 0, 0))

Blobby 6 [1004 0 0 0 0 0 1001 0 1001 16 1001 32 1001 48 1000 64 0 4 1 2 3 4 1 2 0 5 0 2 6 7] [1 0 0 0 0 1 0 0 0 0
1 0 0.6 -0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6 0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 0.6 0.6 -0.6 1 1 0 0 0 0 1 0 0
0 0 1 0 -0.6 -0.6 -0.6 1 2.0] ["cube"] "Cs" [1 0 0 0 1 0 0 0 1 1 1 0 0 1 1 1 1 1] "Os" [1 1 1 0.5 0.5 0.5 0.75
0.75 0.75 0.5 0.5 0.5 0.75 0.75 0.75 0.75 0.75 0.75]

TransformEnd

TransformBegin
 Translate -3.9 -1.3 0
 Rotate 10 0 1 0
 Rotate 10 1 1 1

# (Cube() / 2 + (Sphere(1, (.6, -.6, .6)) +
# Sphere(1, (-.6, .6, .6)) +
# Sphere(1, (.6, .6, -.6)) +
# Sphere(1, (-.6, -.6, -.6)))) - Sphere(.75, (0, 0, 0))

Blobby 6 [1004 0 0 0 0 0 1001 0 1001 16 1001 32 1001 48 1000 64 0 4 1 2 3 4 5 2 0 5 0 2 6 7] [1 0 0 0 0 1 0 0 0 0
1 0 0.6 -0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 -0.6 0.6 0.6 1 1 0 0 0 0 1 0 0 0 0 1 0 0.6 0.6 -0.6 1 1 0 0 0 0 1 0 0
0 0 1 0 -0.6 -0.6 -0.6 1 2] ["cube"] "Cs" [1 0 0 0 1 0 0 0 1 1 1 0 0 1 1 1 1 1] "Os" [1 1 1 0.5 0.5 0.5 0.75 0.75
0.75 0.5 0.5 0.5 0.75 0.75 0.75 0.75 0.75 0.75]

TransformEnd


 WorldEnd
FrameEnd
```