Writing Displacements

Displacement shaders: moving surface points (and normals)

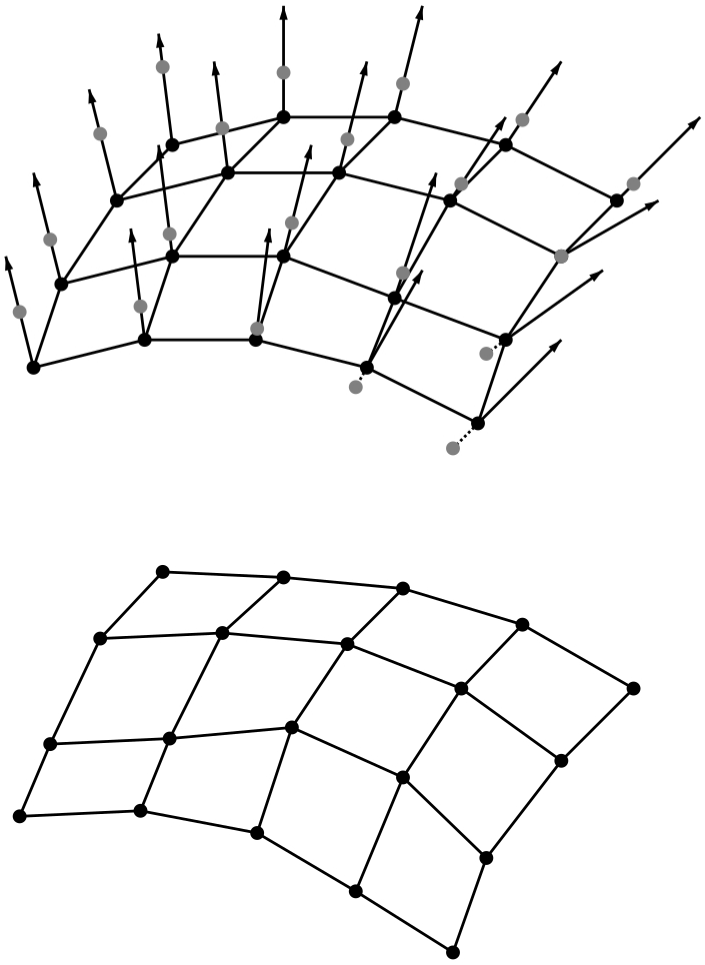
After a surface has been tessellated, its surface points can be moved by a displacement shader. The displacement shader is executed on each vertex of the tessellated surface. In general, each point can be moved by an arbitrary direction and amount:

```
for (int i = 0; i < numPts; i++)
{
  P[i] += <offset vector>;
}
```

Typically the offset vector is along the surface normal as in this example:

```
for (int i = 0; i < numPts; i++)
{
  P[i] += amplitude[i] * Nn[i];
}
```

This figure shows a small grid of surface points before and after displacement:



Left: original surface points and normals, along with displacement amounts. Right: displaced surface points. (Note that some displacement amounts are negative.)

The surface positions and normals are in "current" space. Usually it is convenient to transform them to object space, since it means that the displacements scale and rotate along with the object. (In actuality, "current" space in displacement shaders is object space and the transform turns into an identity, but it is good form to generalize and future-proof any displacement shader you write and do the transformation to object space anyway.) In the rare cases where displacements in world space are preferred, the amplitude simply needs to be scaled by the length of the normal transformed to world space. (If the object is instantiated to more than one instanced object, then the world transform of the first instance is used.)

After the displacement has been done, any smooth analytical normals that the original surface may have had (for example a smooth subdivision surface or NURBS patch) are no longer valid – they do not correspond to the displaced surface. However, each micropolygon has a geometric normal Ngn. In addition, for some surface types, a smooth shading normal will be automatically computed for each ray hit – this is done by considering the orientation of not only the micropolygon that the ray hit, but also adjacent micropolygons. For other surface types – most prominently displaced polygon meshes and displaced pretessellated subdivision surfaces – the shading normal is the same as the geometric normal, i.e. facetted. (We plan to provide smooth shading normals also on these surface types in a future release.)

Displacement bounds

In order to ray trace a scene efficiently, RenderMan needs to know where the objects are. Objects are organized into a ray acceleration data structure: a bounding volume hierarchy (BVH) where each node in the hierarchy is a bounding box for the objects below it in the hierarchy. Computing these bounding boxes is a bit tricky for displaced surfaces because we don't know where the surface points will end up until the displacement shader has run. But we don't want to run the displacement shader on *all* displaced surfaces before tracing the first rays – if we did, the time-to-first-pixel would suffer. What we need is a rough indication of how large the the displacement might be, without the expense of running the displacement shader to determine the exact displacement. Such an indication must be provided with a *displacement bound* for each displaced object; the displacement bound is an upper limit on the displacement on that object. For example, if we know that the maximum magnitude of displacement on a given object is 0.5 units, then we can specify the displacement bound like this:

```
 Attribute "displacementbound" "float sphere" [0.5]
```

This bound means that any ray that is farther than 0.5 units away from the undisplaced surface can ignore that surface. Only when a ray hits this "padded" bounding box (the bounding box of the undisplaced surface points, padded by 0.5 in x, y, and z) do we need to run the displacement shader. Once the displacement shader has run and all the positions are known, their bounding box is computed and the BVH node bounding box is updated (tightened). Selecting an appropriate displacement bound is important: if it is too large, the time to first pixel will be slow; if it is too small, the image will have holes (see the figure below). In most cases, the displacement bound is the same as the *magnitude parameter* of the displacement. To help select an appropriate displacement bound in harder cases, RenderMan will give a warning (after rendering is completed) if the specified displacement bound was too small or more than ten times too large.

Note that an object is only displaced if three conditions are fulfilled: 1) the object has a displacement shader attached to it, 2) the object has an Attribute "displacementbound" greater than 0.0, and 3) the object's Attribute "trace" "displacements" is 1 (which it is by default).

Example

Here is an example of a fragment of a RIB file with a sphere being turned into a five-pointed star:

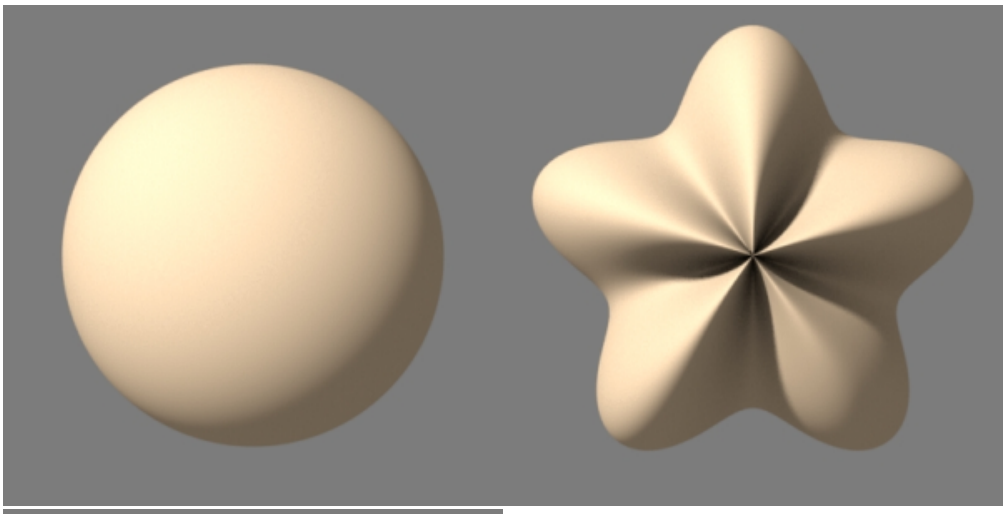
```
  Attribute "displacementbound" "float sphere" [0.21]
  Pattern "dispstar" "sin5theta" "float scale" 0.2
  Displace "PxrDisplace" "pxrdisp" "reference float dispScalar" "sin5theta:resultF"

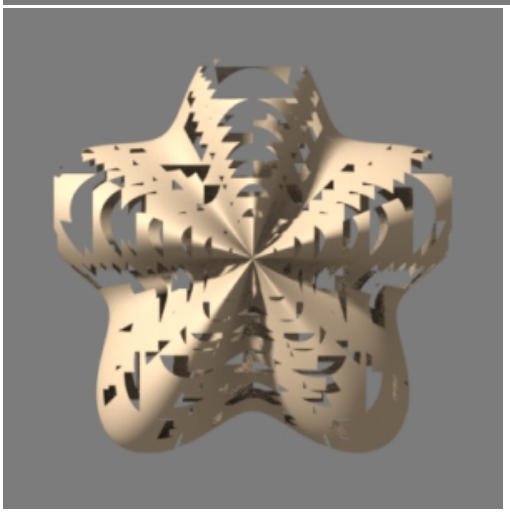
  Sphere 1 -1 1 360
```

The dispstar shader is a very simple shader written in Open Shading Language (OSL). It computes a displacement amount depending on radial angle in the x-y plane. When applied to a spherical shape, it produces a round star with five soft spikes. When applied to a teapot (with higher frequency), it produces a pumpkin-like shape.

```
shader
dispstar(float scale = 1.0, float freq = 5.0,
         output float resultF = 0.0)
{
   // Compute displacement amount
   float angle = atan2(N[1], N[0]);
   float disp = scale * sin(freq*angle);
   resultF = disp;
}
```

In this example, the displacement amount computed by dispstar gets passed on to the PxrDisplace displacement shader, which does the actual displacement. Here is an image of a sphere and the same sphere with displacement (along with an image with holes due to too small displacement bound):





Left: undisplaced sphere. Middle: displaced sphere in the shape of a fat five-pointed star. Right: the displaced sphere with holes due to too small displacement bound.

The seasoned RenderMan user will notice this is all very similar to how displacement was done in the classic RenderMan Shading Language (RSL) – see for example the "Advanced RenderMan" book by Apodaca and Gritz, section 8.2. One notable difference is that the displacement calculations used to be in camera space, but in RenderMan 22 they are in object space.

The `RixDisplacement` class

For most users, it will be sufficient to write OSL patterns to calculate displacement amounts, and simply pass their output to the standard PxrDisplace displacement shader. However, PxrDisplace is just one example of a `RixDisplacement` class displacement shader. The `RixDisplacement` interface characterizes the displacement of points on the surface of an object.

If a developer wishes to write their own displacement plug-in, two classes are of interest: `RixDisplacementFactory` and `RixDisplacement`.

The `RixDisplacementFactory` interface is a subclass of `RixShadingPlugin`, and defines a shading plugin responsible for creating a `RixDisplacement` object from a shading context and the set of connected patterns (`RixPattern`). Since `RixDisplacementFactory` is a subclass of `RixShadingPlugin`, it shares the same initialization, synchronization, and parameter table logic as other shading plugins. Therefore to start developing your own displacement plug-in, you can `#include "RixDisplacement.h"` and make sure your displacement factory class implements the required methods inherited from the `RixShadingPlugin` interface: `Init()`, `Finalize()`, `Synchronize()`, `GetParamTable()`, and `CreateInstanceData()`, as well as the methods `BeginDisplacement()` and `EndDisplacement()`. Generally, there is one shading plugin instance of a `RixDisplacementFactory` per bound `RiDisplacement` (RIB) request. This instance may be active in multiple threads simultaneously.

Integrators (`RixIntegrator`) use `RixDisplacementFactory` objects by invoking `RixBxdfFactory::BeginDisplacement()` to obtain a `RixDisplacement`. Because a `RixDisplacement` is expected to be a lightweight object that may be created many times over the course of the render, `RixDisplacementFactory` is expected to take advantage of the lightweight instancing services provided by `RixShadingPlugin`. In particular, `BeginDisplacement()` is provided a pointer to an instance data that is created by `RixDisplacementFactory::CreateInstanceData`(), which is called once per *shading plugin instance*, as defined by the unique set of parameters supplied to the material description. It is expected that the instance data will point to a private cached representation of any expensive setup which depends on the parameters, and `BeginDisplacement()` will reuse this cached representation many times over the course of the render to create `RixDisplacement` objects.

Once a `RixDisplacement` object is obtained, the renderer may invoke its `GetDisplacement()` method. The `GetDisplacement()` method loops over the surface points in a `RixShadingContext` and moves them according to e.g. surface normal and various input parameters. Operating on a collection of shading points allows maximizing shading coherency and supporting SIMD computation. It would be typical for either `BeginDisplacement()` or the implementation of `GetDisplacement()` to invoke `RixShadingContext::EvalParam()` in order to evaluate the relevant displacement input parameters. Since displacements also generally require geometric data, or built-in variables, such as the shading normal (Nn), `RixShadingContext::GetBuiltinVar()` function should be used for each such built-in variable.

For more details, please see the built-in PxrDisplace displacement shader implementation – it is a good example of how a displacement plug-in is structured.