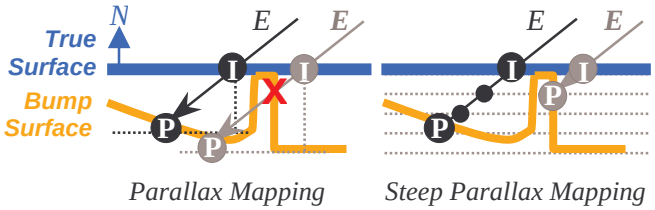
Steep Parallax Mapping

Morgan McGuire*
Brown University

Max McGuire
Iron Lore Entertainment

Bump Mapping *Parallax Mapping* *Steep Parallax Mapping*

Fig. 1 A single polygon with a high-frequency bump map.



Parallax Mapping *Steep Parallax Mapping*

Fig. 2 The viewer perceives an intersection at **(P)** from shading, although the true intersection occurred at **(I).**

Abstract. We propose a new bump mapping scheme that can produce parallax, self-occlusion, and self-shadowing for arbitrary bump maps yet is efficient enough for games when implemented in a pixel shader and uses existing data formats.

Related Work

Let *E* and *L* be unit vectors from the eye and light in a local tangent space. At a pixel, let 2-vector *s* be the texture coordinate (i.e. where the eye ray hits the true, flat surface) and *t* be the coordinate of the texel where the ray *would* have hit the surface if it were actually displaced by the bump map. Shading is computed by Phong illumination, using the normal to the scalar bump map surface **B** at *t* and the color of the texture map at *t*. It is common practice to pack a combined map **NB** with heights in α and normals in *rgb*.

Blinn's original *Bump* (a.k.a. Normal) *Mapping* uses the approximation $t = s$, which produces no parallax. *Parallax Mapping* is the state of the art for **2D bump** maps. It adds self-occlusion through the observation: $t \approx s + \mathbf{B}[s](E_x, E_y)$, for low-frequency bump maps. For high-frequency maps containing *steep* bumps, this breaks down.

There exist several real-time approaches (e.g., [1]) that ray-trace **3D voxel** maps and accurately render steep bumps. The advantage of our technique is that we use only 2D bump maps, which are faster and are already supported by art tools. Because 2D maps require less memory, we can afford high enough resolution to represent fine details like fur. However, we are limited to heightfield surfaces.

The *Shell* method of fur rendering [2] wraps an object in a series of concentric shells textured with 2D slices of a voxel map. When the slices are identical, this also takes advantage of high resolution and fast operations for 2D textures. However, shells can be slow because their high depth complexity consumes fill rate and the alpha blending is limited by frame buffer bandwidth.

Steep Parallax Mapping

Our *Steep Mapping* algorithm combines the strengths of previous methods. We simplify Donnelly's method [1] to use a traditional bump map as input and optimize performance. For developers already using Parallax Mapping, this is a drop-in replacement that changes the offset computation but leaves the art pipeline, existing art assets, and other code unchanged.

In tangent space, the intersections between the eye ray and *n* discrete depth planes of the bump map are easy to compute; they occur at $t_i = s + (E_x, E_y)\ i\ /\ (n\ E_z)$ for integer $0 \le i < n$.

* morgan@cs.brown.edu

We choose *t* to be the first t_i for which $\mathbf{NB}[t_i]_\alpha > 1.0 - i\ /\ n$ and then shade as with other bump mapping methods:

```
float step = 1.0 / n
vec2 dt = E.xy * bumpScale / (n * E.z)

float height = 1.0;
vec2 t = texCoord.xy;
vec4 nb = texture2DLod(NB, t, LOD);

while (nb.a < height) {
    height -= step; t += dt;
    nb = texture2DLod(NB, t, LOD);
}
// ... Shade using N = nb.rgb
```

Here, *LOD* is the MIP-map level; holding it constant during the ray trace reduces the cost of the texture reads.

Self-Shadowing

Because we have a small ray tracer inside the pixel shader, we can just as easily trace shadow rays as eye rays. The shadow loop is the same as the above listing except we increase *height* each iteration and increment *t* along the light vector, by $\Delta t = -L_{xy} * bumpScale\ /\ (n\ L_z)$.

Fur and Grass

Our method can render short, straight fur and grass (Fig. 3), faster but with comparable quality to the Shell method.

Three factors allow high performance. Because the bump map is dense with tall structures, the marching loop is likely to terminate after few iterations, which can be optimized on true hardware with true branches. Unlike Shells, we have zero overdraw and do not use alpha blending. Instead of shadow rays, we use a trick from [2] to approximate self-shadowing as simply increasing with depth.

References

[1] Donnelly, Per-Pixel Displacement Mapping with Distance Functions, to appear in *GPU Gems 2*, 2005

[2] Lengyel, Praun, Finkelstein, and Hoppe. Realtime fur over arbitrary surfaces. in *I3D 2001*.

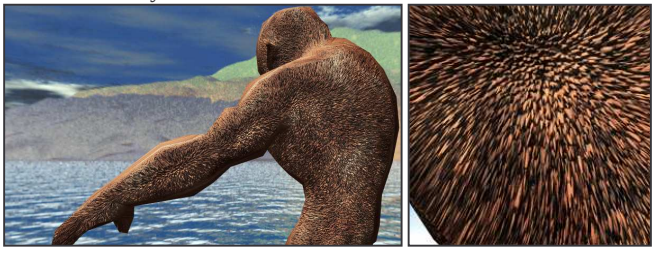


Fig. 3 a) Fur by our method b) Close up of one polygon