



**SIGGRAPH2005**



SIGGRAPH2005

---

# Practical Dynamic Parallax Occlusion Mapping

Natalya Tatarchuk

3D Application Research Group  
ATI Research, Inc.



# Outline

---



- Introduction
- Related work
- Parallax occlusion mapping algorithm
- Results discussion
- Conclusions

# Objective

---



- We want to render very detailed surfaces
- Don't want to pay the price of millions of triangles
  - Vertex transform cost
  - Memory footprint
- Want to render those detailed surfaces accurately
  - Preserve depth at all angles
  - Dynamic lighting
  - Self occlusion resulting in correct shadowing

# Approximating Surface Details



SIGGRAPH2005

- First there was bump mapping...
  - Rendering detailed and uneven surfaces where normals are perturbed in some pre-determined manner
  - Popularized as *normal mapping* – as a *per-pixel* technique

[Blinn78]



# Approximating Surface Details



SIGGRAPH2005

- First there was bump mapping...
- Doesn't take into account geometric surface depth
  - Does not exhibit **parallax**
  - No self-shadowing of the surface
  - Coarse silhouettes expose the actual geometry being drawn

[Blinn78]



*apparent displacement of the object due to viewpoint change*

The surface should appear to move correctly with respect to the viewer

# Parallax Occlusion Mapping



SIGGRAPH2005

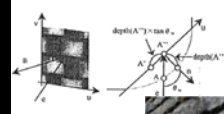
- Per-pixel ray tracing at its core
- Correctly handles complicated viewing phenomena and surface details
  - Displays motion parallax
  - Renders complex geometric surfaces such as displaced text / sharp objects
  - Uses occlusion mapping to determine visibility for surface features (resulting in correct self-shadowing)
  - Uses flexible lighting model

The effect of motion parallax for a surface can be computed by applying a height map and offsetting each pixel in the height map using the geometric normal and the eye vector. As we move the geometry away from its original position using that ray, the parallax is obtained by the fact that the highest points on the height map would move the farthest along that ray and the lower extremes would not appear to be moving at all. To obtain satisfactory results for true perspective simulation, one would need to displace every pixel in the height map using the view ray and the geometric normal.

# Selected Related Work



- Horizon mapping [Max88]
- Interactive horizon mapping [Sloan00]
- Parallax mapping [Kaneko01]
- Parallax mapping with offset limiting [Welsh03]
- [Policarpo05] Real-time relief mapping on arbitrary surfaces



We would like to generate the feeling of motion parallax while rendering detailed surfaces.

Recently many approaches appeared to solve this for real-time rendering.

Parallax Mapping was introduced by Kaneko in 2001

Popularized by Welsh in 2003 with offset limiting technique

## Parallax mapping

- Simple way to approximate motion parallax effects on a given polygon
- Dynamically distorts the texture coordinate to approximate motion parallax effect
- Shifts texture coordinate using the view vector and the current height map value
- Issues:
  - Doesn't accurately represent surface depth
  - Swimming artifacts at grazing angles
  - Flattens geometry at grazing angles
- Pros:
  - No additional texture memory and very quick (~3 extra instructions)

Horizon Mapping:

- Encodes the height of the shadowing horizon at each point on the bump map in a series of textures for 8 directions
- Determines the amount of self-shadowing for a given light position
- At each frame project the light vector onto local tangent plane and compute per-pixel lighting
- Draw backs: additional texture memory

## Offset Limiting

- Same idea as in [Kaneko01]
- Uses height map to determine texture coordinate offset for approximating parallax
- Uses view vector in tangent space to determine how to offset the texels
- Reduces visual artifacts at grazing angles ("swimming texels") by limiting the offset to be at most equal to current height value
  - Flattens geometry significantly at grazing angles
  - Just a heuristic



# Our Contributions

---



- Increased precision of height field – ray intersections
- Dynamic real-time lighting of surfaces with soft shadows due to self-occlusion under varying light conditions
- Directable level-of-detail control system with smooth transitions between levels
- Motion parallax simulation with perspective-correct depth

Our technique can be applied to animated objects and fits well within established art pipelines of games and effects rendering.

The implementation makes effective use of current GPU pixel pipelines and texturing hardware for interactive rendering.

The algorithm allows scalability for a range of existing GPU products.

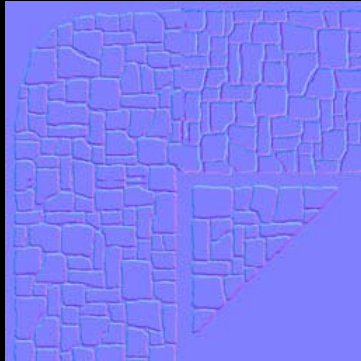
# Parallax Occlusion Mapping



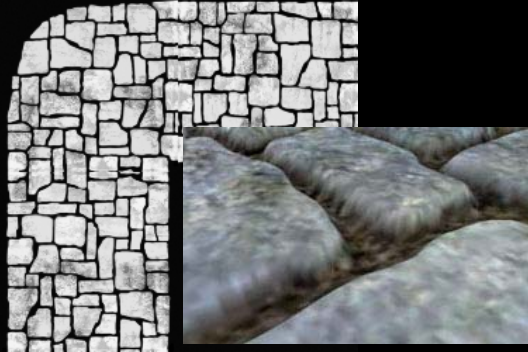
- Introduced in [Browley04]  
“Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing”
- Efficiently utilizes programmable GPU pipeline for interactive rendering rates
- Current algorithm has several significant improvements over the earlier technique



# Encoding Displacement Information



Tangent-space normal map



Displacement values (the height map)

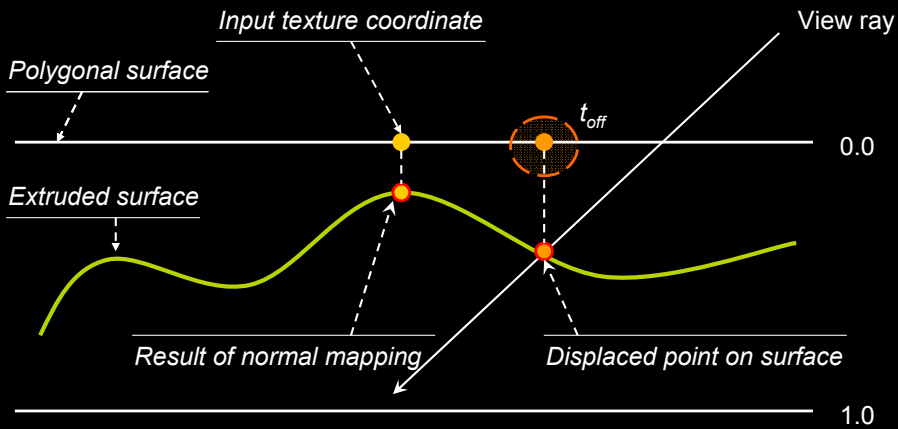
All computations are done in tangent space, and thus can be applied to arbitrary surfaces

We encode surface displacement information in a tangent-space normal map accompanied by a scalar height map.

Since tangent space is inherently locally planar for any point on an arbitrary surface, regardless of its curvature, it provides an intuitive mapping for surface detail information.

We perform all calculations for height field intersection and visibility determination in tangent space, and compute the illumination in the same domain.

# Parallax Displacement



The core idea of the presented algorithm is to trace the pixel being currently rendered in reverse in the height map to determine which texel in the height map would yield the rendered pixel location if in fact we would have been using the actual displaced geometry.

## Implementation: Per-Vertex

---



SIGGRAPH2005

- Compute the viewing direction, the light direction in tangent space
- May compute the parallax offset vector (as an optimization)
  - Interpolated by the rasterizer

Compute the parallax offset vector  $P$  to determine maximum visual offset in texture-space for current pixel being rendered.

# Implementation: Per-Pixel



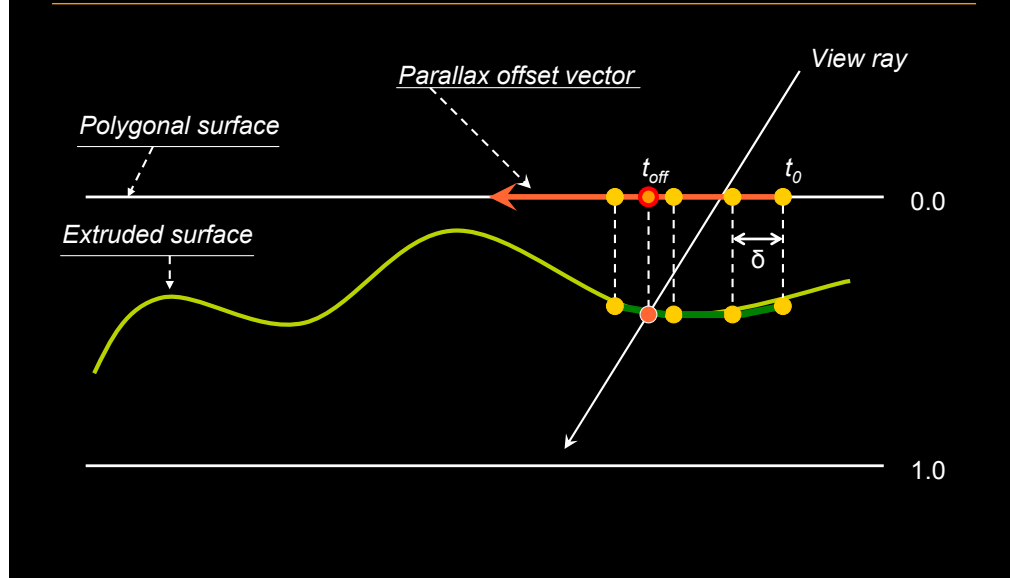
- Ray-cast the view ray along the parallax offset vector
- Ray – height field profile intersection as a texture offset
  - Yields the correct displaced point visible from the given view angle
- Light ray – height profile intersection for occlusion computation to determine the visibility coefficient
- Shading
  - Using any attributes
  - Any lighting model

Ray cast the view ray along the parallax offset vector to compute the height **profile – ray intersection point**. We sample the height field profile along the parallax offset vector to determine the correct displaced point on the extruded surface. Approximating the height field profile as a piecewise linear curve allows us to have increased precision for the desired intersection (versus simply taking the nearest sample). This yields the texture coordinate shift offset (parallax offset) necessary to arrive at the desired point on the extruded surface. We add this parallax offset amount to the original sample coordinates to yield texture offset coordinates.

If computing shadowing and self-occlusion effects, we can use the texture offset coordinates to perform **visibility computation** for light direction. In order to do that, we ray cast the light direction ray sampling the height profile along the way for occlusions. This results in a visibility coefficient for the given sample position.

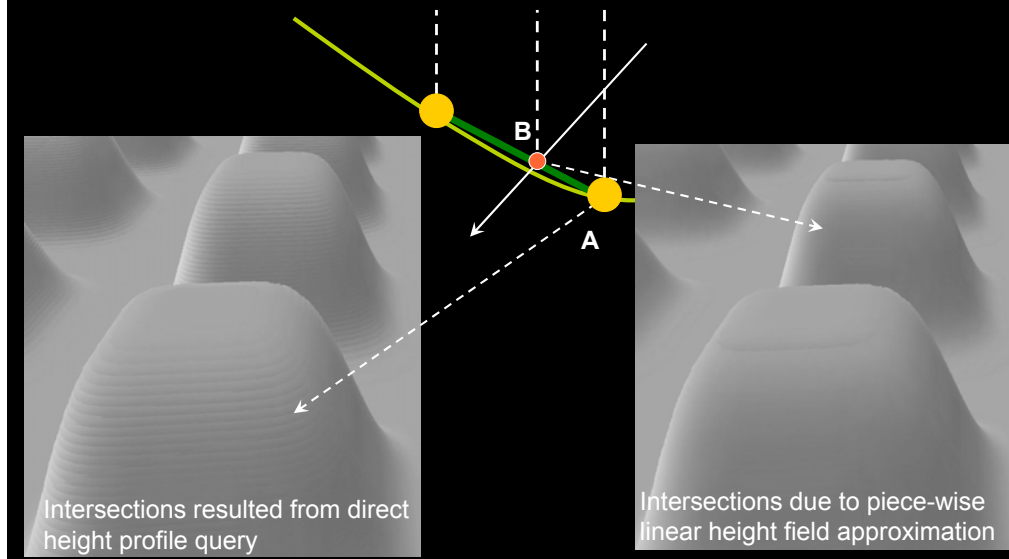
Using the texture offset coordinates and the visibility coefficient, we can **shade the given** pixel using its attributes, such as applied textures (albedo, gloss, etc), the normal from the normal map and the light vector.

# Height Field Profile Tracing



In order to compute the height field-ray intersection we approximate the height field (seen as the light green curve in this figure) as a piecewise linear curve (seen here as dark green segments), intersecting it with the given ray (in this case, the view direction) for each linear section. We start by tracing from the input sample coordinates  $t_0$  along the computed **parallax offset vector**  $P$ . We perform a linear search for the intersection along the parallax offset vector. We sample a linear segment from the height field profile by **fetching two samples** step size  $\delta$  apart. We successively **test each segments endpoints** to see if it would possibly intersect with the view ray. For that, we simply use the height displacement value from each end point to see if they are above current horizon level. Once such pair of end points is found, we compute an intersection between this linear segment and the view ray. The intersection of the height field profile yields the point on the extruded surface that would be visible to the viewer.

# Height Field Profile – Ray Intersection



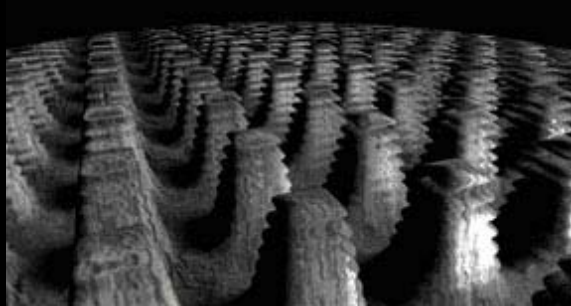
Other ray tracing-based mapping techniques query the height profile for the closest location to the viewer along the view direction. In the case presented here, these techniques would report point A as the displacement point. This results in the stair stepping artifacts visible in the picture on the left. The artifacts are particularly strong at oblique viewing angles, where the apparent parallax is larger. We perform actual line intersection computation for the ray and the linear section of the approximated height field. This yields the intersection point B. In the figure on the right, you see the smoother surface rendered using higher precision height field intersection technique. In both figures the identical number of samples was used during tracing view direction rays.



# Dynamic Sampling Rate



- Sampling-based algorithms are prone to aliasing



One of the biggest problems with the aliasing algorithms exists due to aliasing artifacts. Here you see **the result** of our 2004 technique intersecting the height field with a fixed sampling rate. Note the aliasing artifacts visible with this technique at the grazing angle. In the [Brawley04] approach we applied perspective bias to fix this artifact. Unfortunately, that results in strong flattening of the surface details along the horizon, which is undesirable.

# Dynamic Sampling Rate



- Dynamically adjust the sampling rate for ray tracing as a linear function of angle between the geometric normal and the view direction ray



Perspective-correct depth with  
*dynamic* sampling rate

Dynamically scaling the sampling rate ensures that the resulting extruded surface is far less likely to display aliasing artifacts and certainly does not display any flattening as in **this** figure. Therefore the surfaces rendered with our approach display perspective-correct depth at all angles.

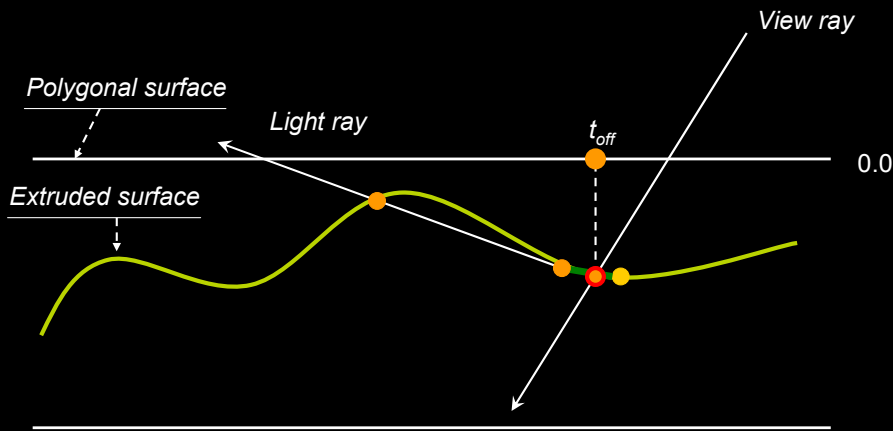
On the latest GPUs we can utilize dynamic flow control instructions to dynamically scale the sampling rate during ray tracing. We express the sampling rate as a linear function of the angle between the geometric normal and the view direction ray. This ensures that we take more samples when the surface is viewed at steep grazing angles, where more samples are desired.

Other techniques encode the surface information in a distance map, which allows sampling the height field as a function of the distance from the height field.

However, this utilizes dependent texture fetches which exhibits a higher latency penalty during rendering. Additional texture memory cost can also be prohibitive in real-time production environments, which steered us away from similar approaches. Additionally, we optimize our ray tracing techniques by accurately computing the length of the parallax vector and only sampling along this vector.

Note that the sampling interval  $\delta$  is a function of the sampling rate ( for  $n$  number of samples,  $\delta = 1 / n$  ). We provide control over the sampling interval to the artists by exposing the range for dynamic sampling rate. The accuracy of this technique corresponds to the sampling interval  $\delta$  and, therefore, on the number of samples during ray tracing.

# Self-Occlusion Shadows

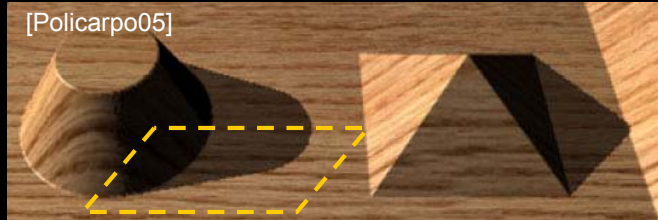


The features of the height map can in fact cast shadows onto the surface. Once we arrive at the point on the displaced surface (**highlighted here**) we can compute its visibility from the any light source. For that, we **cast a ray toward the light source** in question and perform horizon visibility queries of the height field profile along the light direction ray. If there are intersections of the height field profile with the light vector, then there are occluding features and the point in question will **be in shadow**. This process allows us to generate shadows due to the object features' self-occlusions and object interpenetration.

# Soft Shadows Computation



- Simply determining whether the current feature is occluded yields hard shadows



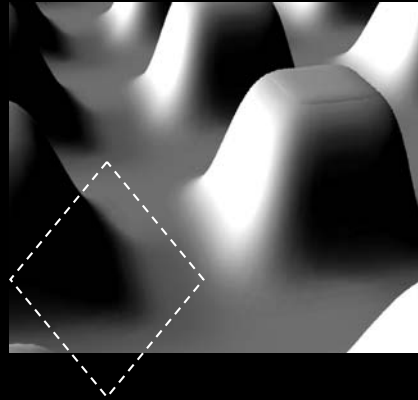
While computing the visibility information, we could simply stop at the first **intersection blocking** the horizon from the current view point. This yields the horizon shadowing value specifying whether the displaced pixel is in shadow. Other techniques, as seen in this picture, use this approach. This generates **hard shadows** which may have strong aliasing artifacts as you can see in the high-lighted portion

# Soft Shadows Computation



- Simply determining whether the current feature is occluded yields hard shadows
- We can compute soft shadows by filtering the visibility samples during the occlusion computation
- Don't compute shadows for objects not facing the light

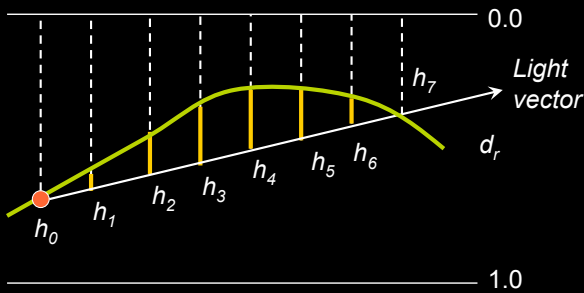
$$N \bullet L > 0$$



In our algorithm, we continue sampling the height field along the light ray past the first shadowing horizon until we reach the next fully visible point on the surface.

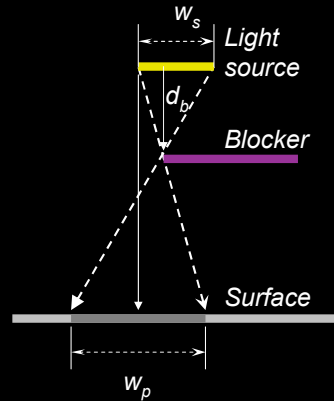
Then we filter the resulting visibility samples to compute soft shadows with smooth edges. We optimize the algorithm by only performing visibility query for areas which are lit by the given light source with a simple test.

# Penumbra Size Approximation SIGGRAPH2005



- The blocker heights  $h_i$  allow us to compute the *blocker-to-receiver* ratio

$$w_p = w_s (d_r - d_b) / d_b$$



We sample the height value  $h_0$  at the shifted texture coordinate  $t_{off}$ . The sample  $h_0$  is our reference (“surface”) height. We then sample  $n$  other samples along the light ray, subtracting  $h_0$  from each of the successive samples  $h_i$ . This allows us to compute the blocker-to-receiver ratio as in figure.

We note that the closer the blocker is to the surface, the smaller the resulting penumbra. We compute the visibility coefficient by scaling the contribution of each sample by the distance from the reference sample. We apply this visibility coefficient during the lighting computation for generation of smoothly soft shadows. In combination with bi- or trilinear texture filtering in hardware, we are able to obtain well-behaved soft shadows without any edge aliasing or filtering artifacts present in many shadow mapping techniques.

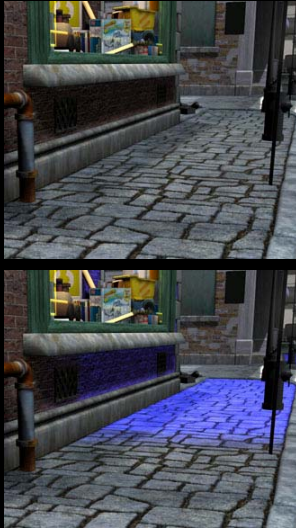
# Illuminating the Surface

- Apply material attributes sampled at the offset corresponding to the displacement point
- Any lighting model is suitable



Since we have already computed the parallax offset, the shifted texture coordinates  $t_{off}$  and the visibility coefficient, in order to shade the current pixel, we can now perform any lighting computation. For example, to compute Phong shading, we can sample the normal map for the surface normal matching the extruded surface at that location, sample the other associated maps (albedo, specular, etc) and compute the Phong illumination result using standard formulas. Other illumination techniques are equally applicable.

# Adaptive Level-of-Detail System



- Compute the current mip map level
- For furthest LOD levels, render using normal mapping (threshold level)
- As the surface approaches the viewer, increase the sampling rate as a function of the current mip map level
- In transition region between the threshold LOD level, blend between the normal mapping and the full parallax occlusion mapping

We designed an explicit level-of-detail control system for automatically controlling shader complexity. We **determine the current mip map level** directly in the pixel shader and use this information to transition between different levels of detail from the full effect to simple normal mapping. We render the **lowest level of details** using regular normal mapping shading. As the surface **approaches the viewer**, we increase the sampling rate for the full parallax occlusion mapping computation as a function of the current mip level. We specify an artist-directable **threshold level** where the transition between the parallax occlusion mapping and the normal mapping computations will occur. When the currently rendered surface portion is in the transition region, we interpolate the result of parallax occlusion mapping computation with the normal mapping result. We use the fractional part of the current mip level computed in the pixel shader.

As you can compare between these two figures, there is no associated visual quality degradation as we move into a lower level of detail and the transition appears quite smooth.

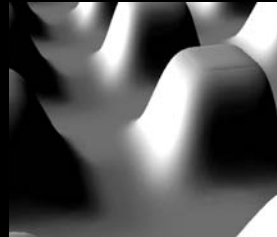


# Results

- Implemented using DirectX 9.0c shaders (separate implementations in SM 2.0, 2.b and 3.0)



RGBa texture: 1024 x 1024,  
non-contiguous *uvs*



RGBa texture: tiled 128 x 128

We use different texture sizes selected based on the desired feature resolution. Notice perspective-correct depth rendering and lack of aliasing artifacts.

We found that depending on selected number of samples range and the size of features in the normal / height map texture aliasing artifacts are possible, as our linear search may miss features. However, in practice with a large selection of texture maps we found this to be not the case.

# Parallax Occlusion Mapping vs. Actual Geometry



An 1,100 polygon object rendered with  
parallax occlusion mapping



A 1.5 million polygon object  
rendered with diffuse lighting

We applied parallax occlusion mapping to an 1,100 polygon soldier character displayed on the left.

We compared this result to a 1.5 million polygon soldier displayed on the right used to generate normal maps for the low resolution model.

We use the same lighting model on both objects.

# Parallax Occlusion Mapping vs. Actual Geometry



An 1,100 polygon object rendered with  
parallax occlusion mapping (wireframe)



A 1.5 million polygon object  
rendered with diffuse lighting  
(wireframe)

We applied parallax occlusion mapping to an 1,100 polygon soldier character displayed on the left.

We compared this result to a 1.5 million polygon soldier displayed on the right used to generate normal maps for the low resolution model.

We use the same lighting model on both objects.

# Parallax Occlusion Mapping vs. Actual Geometry



- 1100 polygons with parallax occlusion mapping (8 to 50 samples used)
- **Memory:** 79K vertex buffer  
6K index buffer  
13Mb texture (3Dc)  
(2048 x 2048 maps)

---

Total: < 14 Mb

## **Frame Rate:**

- **255 fps** on ATI Radeon hardware
- **235 fps** with skinning



- 1,500,000 polygons with diffuse lighting
- **Memory:** 31Mb vertex buffer  
14Mb index buffer

---

Total: 45 Mb

## **Frame Rate:**

- **32 fps** on ATI Radeon hardware

We apply a 2048x2048 RGBa texture map to the low resolution object. We render the low resolution soldier using DirectX on ATI Radeon X850 at 255 fps.

From 8 to 50 samples were used during ray tracing as necessary. The memory requirement for this model was 79K for the vertex buffer and 6K for the index buffer, and 13Mb of texture memory (we use 3DC texture compression).

The high resolution soldier model rendered on the same hardware at a rate of 32 fps. The memory requirement for this model was 31Mb for the vertex buffer and 14Mb for the index buffer.

However, using our technique on an extremely low resolution model provided significant frame rate increase with 32Mb memory saving at comparable quality of rendering. Notice the details on the bullet belts and the gas mask for the low polygon soldier. We also animated the low resolution model with a run cycle using skinning in vertex shader rendering at 235 fps on the same hardware.

Due to memory considerations, vertex transform cost for rendering, animation, and authoring issues, characters matching the high resolution soldier are impractical in current game scenarios.



SIGGRAPH2005

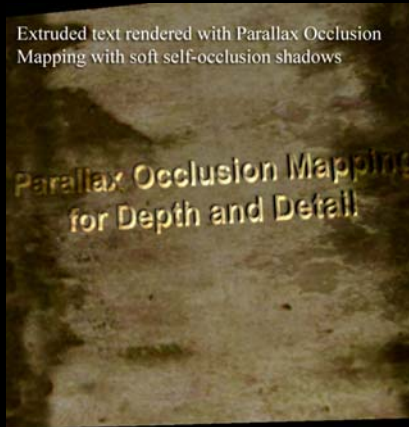
---

**Demo**

*Parallax Occlusion Mapping*

Implemented using DirectX 9.0c shaders (separate implementations in SM 2.0, 2.b and 3.0)

# Difficult Cases



The improvements to the earlier parallax occlusion mapping technique provide the ability to render such traditionally difficult displacement mapping cases such as raised text or objects with very fine features. In order to render the same objects interactively with equal level of detail, the meshes would need an extremely detailed triangle subdivision (with triangles being nearly pixel-small), which is impractical even with the currently available GPUs.

This demonstrates the usefulness of the presented technique for texture-space displacement mapping via parallax occlusion mapping.

# Conclusions



- Powerful technique for rendering complex surface details in real time
- Produces excellent lighting results
  - Higher precision height field – ray intersection computation
  - Self-shadowing for self-occlusion in real-time
  - LOD rendering technique for textured scenes

We have presented a novel **technique** for rendering highly detailed surfaces under varying light conditions.

We have described an efficient algorithm for **computing intersections** of the height field profile with rays with high precision.

We presented a algorithm for generating **soft shadows** during occlusion computation.

An **automatic level-of-detail control system** is used by our approach to control shader complexity efficiently.

A benefit of our approach lies in a **modest texture memory footprint**, comparable to normal mapping. It requires only an grayscale texture in addition to the normal map.

Our technique is designed to take advantage of the **GPU programmable pipeline** resulting in highly interactive frame rates.

It efficiently uses the **dynamic flow control feature** to improve resulting visual quality and optimize rendering speed.

Additionally, this algorithm is designed to easily support **dynamic rendering to height fields** for a variety of interesting effects.

Algorithms based on precomputed quantities are not as flexible and thus are limited to the static height fields

# Conclusions



- Powerful technique for rendering complex surface details in real time
- Produces excellent lighting results
- Has modest texture memory footprint
  - Comparable to normal mapping
- Efficiently uses existing pixel pipelines for highly interactive rendering
- Supports dynamic rendering of height fields and animated objects

We have presented a novel **technique** for rendering highly detailed surfaces under varying light conditions.

We have described an efficient algorithm for **computing intersections** of the height field profile with rays with high precision.

We presented a algorithm for generating **soft shadows** during occlusion computation.

An **automatic level-of-detail control system** is used by our approach to control shader complexity efficiently.

A benefit of our approach lies in a **modest texture memory footprint**, comparable to normal mapping. It requires only an grayscale texture in addition to the normal map.

Our technique is designed to take advantage of the **GPU programmable pipeline** resulting in highly interactive frame rates.

It efficiently uses the **dynamic flow control feature** to improve resulting visual quality and optimize rendering speed.

Additionally, this algorithm is designed to easily support **dynamic rendering to height fields** for a variety of interesting effects.

Algorithms based on precomputed quantities are not as flexible and thus are limited to the static height fields





SIGGRAPH2005

# References

---

- [Blinn78] Blinn, James F. "*Simulation of wrinkled surfaces*", Siggraph '78
- [Max88] N. Max "*Horizon Mapping: shadows for bump-mapped surfaces*", The Visual Computer 1988
- [Sloan00] P-P. Sloan, M. Cohen, "*Interactive Horizon Mapping*", Eurographics 2000
- [Welsh04] T. Welsh, "*Parallax Mapping with Offset Limiting: A Per Pixel Approximation of Uneven Surfaces*", 2004
- [Wang03] L. Wang *et al*, "*View-Dependent Displacement Mapping*", Siggraph 2003
- [Doggett00] M. Doggett, J. Hirche, "*Adaptive View Dependent Tessellation of Displacement Maps*", Eurographics Hardware Workshop 2000
- [Kaneko01] Kaneko *et al.*, "*Detailed Shape Representation with Parallax Mapping*", ICAT 2001



SIGGRAPH2005

## References (cont.)

---

- [Brawley04] Z. Brawley, N. Tatarchuk, "*Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing*", ShaderX<sup>3</sup>, 2004
- [Oliveira00] M. Oliveira et al, "*Relief Texture Mapping*", Siggraph 2000
- [Policarpo05] F. Policarpo, M. M. Oliveira, J. L. D. Comba, "*Real-Time Relief Mapping on Arbitrary Polygonal Surfaces*", ACM Symposium on Interactive 3D Graphics and Games, 2005
- [Yerex04] K. Yerex, M. Jagersand, "*Displacement Mapping with Ray-casting in Hardware*", Siggraph 2004 Sketch
- [Donnelly05] W. Donnelly, "*Per-Pixel Displacement Mapping with Distance Functions*", GPU Gems2, 2005

# Acknowledgements

---



- Zoe Brawley, *Relic Entertainment*
- Dan Roeger, Abe Wiley, Daniel Szecket and Eli Turner (3D Application Research Group, *ATI Research*) for the artwork
- 3D Application Research Group



**SIGGRAPH**2005

---

**Questions?**