High Performance Computer Graphics
Project Report
Interior Parallax Occlusion Mapping

Eliseu Amaro, el6033am-s
Edwin Gustafsson, ed3047gu-s

December 2022

1 Introduction

The goal of our project was to combine two different graphical techniques, Parallax Occlusion Mapping (POM) and Interior Mapping, in order to bring an illusion of depth to interior mapped walls.

POM is a technique that uses height information to simulate a parallax effect with the possibility of self-occlusion to give flat objects the illusion of having more complex geometry. POM can be expanded upon to produce self-shadowing which accentuates the depth effect.

Interior mapping is a technique that maps 5 textures to a flat surface, giving it the appearance of a cuboid room interior that can be seen through the surface.

Interior mapping is featured in many games, such as Insomniac games' 'Spider-Man' (2018). But we can not find an example of it being combined with Parallax Occlusion Mapping for added detail. This fact motivated us to try it for this project.

2 Algorithms

Height and depth may be used interchangeably in this text, as the textures are defined as height maps, but the algorithms use depth values that are the inverse of height: $depth = 1 - height$.

2.1 Parallax Occlusion Mapping

The POM algorithm works by stepping along the view direction and checking for intersections with the height map, similarly to a ray tracing algorithm.

The view vector is translated into ΔUV coordinates, which is the change in texture coordinates when one step is performed and one depth layer is advanced. The size of this step is determined by a predetermined depth layer resolution and the height scale, which defines the apparent distance between the geometry surface and the deepest point of the simulated surface defined by the height map.

A collision is detected when the current depth map value is less than the depth of the current ray position, checked at every step along the ray.

After a collision is detection, the last step is linearly interpolated along to approximate a more accurate collision. The result of this interpolation are the final texture coordinates used to sample albedo, normal, and any other texture map. Code by Victor Gordan was referenced for the base POM algorithm, which helped give us a swift start to the project. [1]

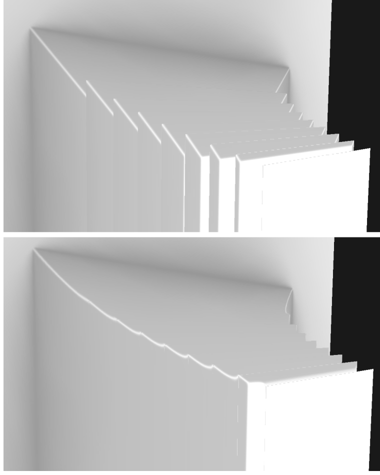

Figure 1: Comparison without and with the last interpolating step.

2.1.1 Hard Shadows

Hard shadows is the naive approach to POM self-shadowing. When the POM algorithm is done and has provided a new set of texture coordinates, we can start from that position in a new ray that is projected along the direction to a light source, and if the ray intersects the height map, it is considered to be in shadow. This approach is not ideal because intersections are checked in discreet steps, which will cause a very obvious aliasing effect when the occluder is very near the shadowed surface.

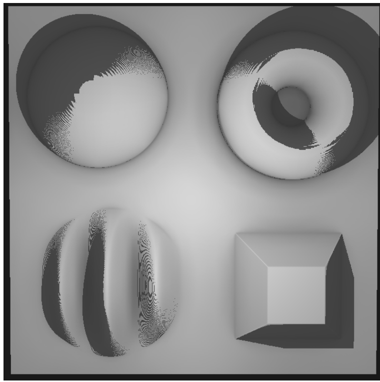

Figure 2: POM with hard shadows

2.1.2 Soft Shadows

Soft shadows are performed as described in 'Dynamic parallax occlusion mapping with approximate soft shadows'.[2]

Soft shadows are handled similarly to hard shadows, except instead of finding the first occlusion, we trace the entire ray until it escapes the surface of the actual geometry. The largest occlusion is recorded along with it's distance to the starting point, the largest occlusion is the point on the ray that is the deepest as compared to the height field. The information of the distance from the occlusion and the size of the occlusion can be used to approximate the penumbra size of the shadow using the equation shown in figure 3.

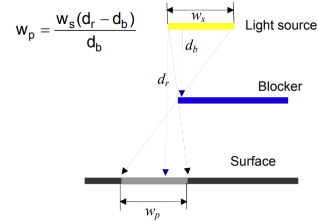

Figure 3: Penumbra size approximation for area light sources, Where w_s is the light source width, w_p is the penumbra width, d_r is the receiver depth and d_b is the blocker's depth from the light source.[2]

The approximated penumbra width is then translated to a shadow factor between 0 and 1 that is multiplied onto the fragment colour. Using the following function:

$$s(w_p) = \begin{cases} 0 & \text{if } 1 - \dfrac{w_p}{0.24 + w_p} < 0 \\ 1 - \dfrac{w_p}{0.24 + w_p} & \text{if } 0 \leq 1 - \dfrac{w_p}{0.24 + w_p} \leq 1 \\ 1 & \text{if } 1 - \dfrac{w_p}{0.24 + w_p} > 1 \end{cases}$$

This function is based on the "fast sigmoid" function. [3] The number 0.24 is changed from 1 because it changes the curve of the function, and in this case increases the contrast of the shadows.

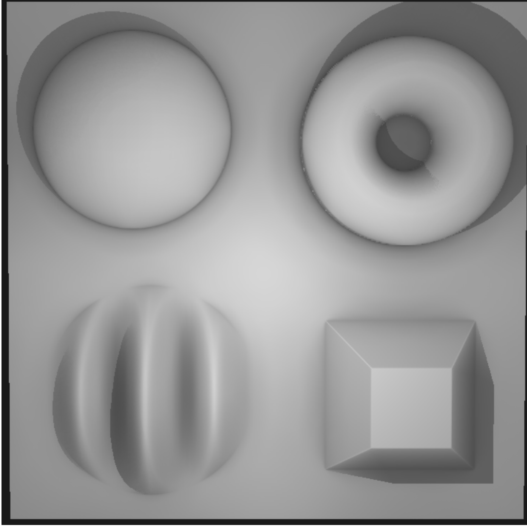

Figure 4: POM with soft shadows

2.1.3 Dynamic Layer Resolution

A system for dynamically adjusting the layer resolution for POM and shadows was developed as an optimization. It is realized as a function with two variables, the ray's angle from the surface normal θ and the distance between the fragment and camera d, and a set of range constants that are interpolated between, L_{min}, L_{max} and $L_{\theta_{min}}$:

$$L = (1 - \cos\theta)(L_{min}\frac{d}{3+d} + L_{max} \cdot (1 - \frac{d}{3+d})) + L_{\theta_{min}} \cos\theta$$

The range constants are set to appropriate values depending on the requirements of the textures used. A high frequency detail in height maps will need a higher resolution to look acceptable than height maps with low frequency shapes.

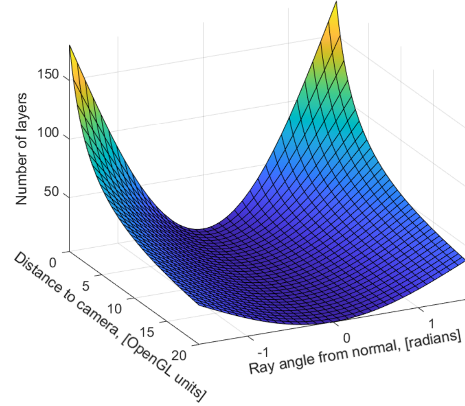

Figure 5: Layer resolution as a function of camera distance and ray angle from normal. Range constants: $L_{min} = 16$, $L_{max} = 180$, $L_{\theta_{min}} = 4$

2.2 Interior Mapping

When rendering dense cities, it becomes impossible to render inner building detail in real time. A major contributing factor to this comes in the presence of windows, which would normally let you peer into the rooms inside, leading to exponentially more detail to be rendered.

Interior mapping has allowed various games to give the illusion of inner spaces to great effect. Nonetheless, prior implementations lacked depth, which we hoped to employ using POM as explained earlier.

In order to keep the implementation simple, and allow this technique to be used in conjunction with POM, the fragment shader was constructed in such a way as to section each wall calculations from one another.

2.2.1 Theory

Floor and ceiling are obtained via the following y-axis or horizontal plane system of equations:

$$\begin{cases} y = D \\ (x, y, z) = P + \dagger \cdot V \end{cases} \quad (1)$$

Where:

- D is equal to the predefined room height;

- P is a known point which is part of the ray traced from the camera to the quad's surface;
- † is some ratio of the ray vector that meets the intersection point of the closest horizontal plane hit after the quad's intersection;

This can be simplified as the following:

$$\begin{cases} y = ceiling_y \\ (x, y, z) = camera_pos + \dagger \cdot (frag_pos - camera_pos) \end{cases} \tag{2}$$

Which solved for †, becomes:

$$\dagger = (ceiling_y - camera_pos.y)/(frag_pos - camera_pos) \tag{3}$$

Finally, we can retrieve which 2D point on the plane located at $y = D$ we hit. This point, when normalized to a specified *room size*, is our texel coordinate.

$$(x, z) = (camera_pos.xz) + \dagger \cdot (frag_pos - camera_pos) \tag{4}$$

Our † y value can then be compared against half the *room's height*, if greater then it's a ceiling, if lower then it's the floor.

The same method can be used for a vertical plane on the *x-axis*, forming the left and right walls. And then again, for a vertical plane on the *z-axis*, retrieving us the back wall.

2.3 Light scattering

The light scattering implementation follows a ray, traced from the light position to the camera's position. An inverted opacity layer of the window's texture allows us to know which rays hit the *"glass"* of the window.

We start by painting these hit points, and then follow the direction of the light ray, painting this *"hit plane"* again, and again at a known interval rate with a diminished intensity each step of the way.

2.4 Combination

To combine interior mapping with POM, a function was created that performs POM based on the following input parameters:

1. vec2: Texture coordinates of the interior wall, supplied by the interior mapping algorithm.
2. vec3: Normal vector of the interior wall.
3. vec3: View vector, transformed to interior wall space with a rotation matrix.
4. float: Height scale, for individual height scales for each wall.
5. int: Wall index, a numerical representation for which wall texture should be sampled.

This function returns UV coordinates produced by POM for the sampled interior wall.

Soft shadows were implemented with a similar function, inputting UV coordinates retrieved from POM, and using the vector to the light source instead of the view vector. The function returns a shadow factor.

For each wall, a view matrix transform is applied, in order to translate us into each wall's space. Then, since we know our UV coordinates, and normals, we can use each of the above functions to apply the parallax occlusion mapping technique as well as soft shadows unto the wall in question.

3 Results

The textures used to demonstrate our shader were assembled using assets sourced from Quixel Megascans, issued under a personal licence. The results are demonstrated on a single quad, using 6 albedo maps, 6 normal maps, 5 height maps, and 1 opacity map. All textures are 2048 pixels by 2048 pixels.

Figure 6: Interior mapping with normal mapping only.

Figure 8: Interior mapping with POM and soft shadows.

Figure 7: Interior mapping with normal mapping only, window wall hidden.

Figure 9: Interior mapping with POM and soft shadows, window wall hidden.

3.1 Performance comparison

The following performance metrics were taken from *nvtop*, a command line GPU process monitoring tool (`https://github.com/Syllo/nvtop`). Uses NVIDIA's Management Library (NVML) API for its information gathering.

3.1.1 Testing rig

- **OS** Arch GNU/Linux
- **Kernel** 6.1.1-zen
- **CPU** Intel i7-12700H (20) @ 4.6GHz
- **GPU** NVIDIA Geforce RTX 3070 Ti Laptop

3.1.2 Testing parameters

- Baseline wall's were created from their respective heightmaps, sampled on a 320x320 grid in Blender, no textures applied, and basic lighting
- POM used a minimum of 1 layer, and a maximum of 256 layers
- Light scattering used 42 layers

3.1.3 Data

Baseline statistics show very high GPU utilization (92%), with a reported power usage of 141W.

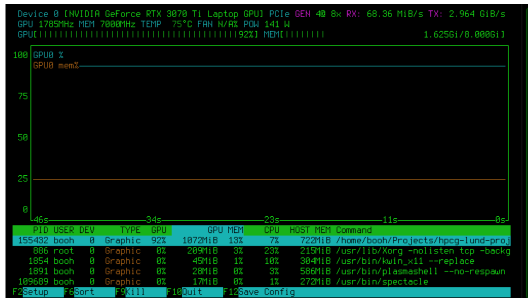

Figure 10: Baseline room with actual geometry

Interior mapping by itself shows relatively low GPU utilization (42%), with a reported power usage of 36W. While the rendered output is comparatively simple in detail, it also shows that we've still got a lot of headroom.

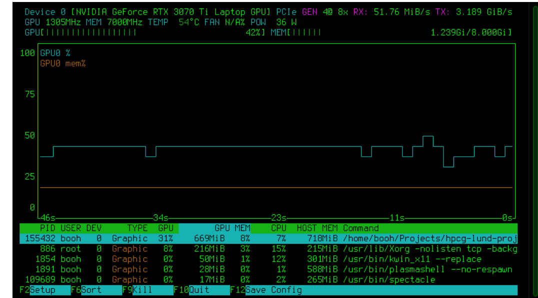

Figure 11: Interior mapping only

Interior mapping with parallax occlusion mapping done for each of its 5 walls still shows a relatively low GPU utilization (51%), with a reported power usage of 74W. Impressive results considering that we've essentially as good of an output as shown under our baseline. While geometric detail is comparable, texture work is also presented. Contrary to baseline.

Judging solely by power usage, the target baseline was twice as less efficient, proving the potential of this technique for real time rendering applications.

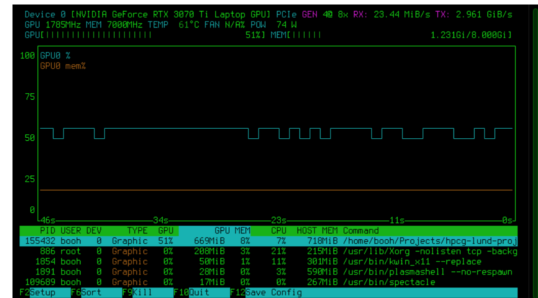

Figure 12: Interior mapping + POM

Upon enabling soft shadows, GPU utilization grew by 10%, reporting a power usage of 81W. Nonetheless, improvements in image depth perception more than make up for it.

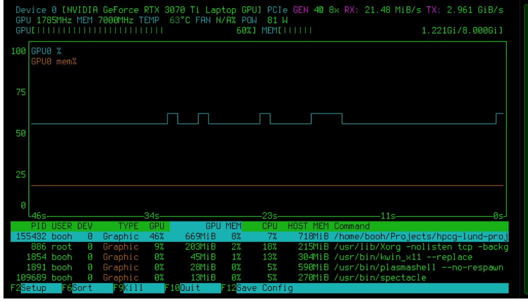

Figure 13: Interior mapping + POM + soft shadows

Light scattering further bumped GPU utilization by 10%, making it hover around 70% GPU utilization. Power usage is now reported at 93W.

Its effect further grounds the scene, to great effect. However, the power creep begins to suggest that further fidelity improvements will lead to diminishing returns in overall performance.

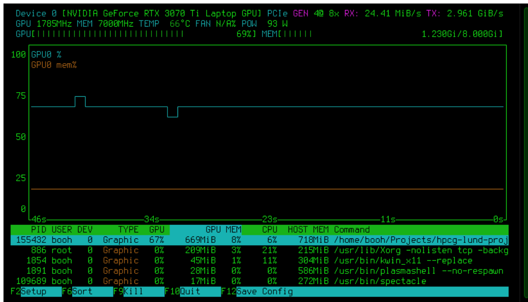

Figure 14: Interior mapping + POM + soft shadows + light scattering

When gathering these results, it became clear that our light scattering implementation required further tweaking.

The number of samples used was notably diminished, from 100 to just 32. Decay rate, sample density and weight was lowered as a result. Light intensity got halved as well. In the end, similar results were obtained but the performance benefits were substantial. Goes to show that simple knob adjustments go a long way when using this technique.

4 Discussion

We are very satisfied with the result of our algorithm visually. The performance should be further investigated before concluding if this method would be suitable for use in games. A limiting factor would probably be the amount of textures used, as our method requires albedo, height and normal textures for each wall, this results in a large memory footprint.

It is important to note that this method requires the boundary condition of a height value equal to 1 for the interior wall height textures, this stops repetition of the texture from being visible, an effect commonly seen with POM.

4.1 Potential improvements

Currently, some direction vectors and rotation matrices are hard-coded, these could be improved to depend on the orientation of the geometry surface so that surfaces of any orientation produces correct results.

In addition, if this technique were to be coupled with parallax corrected cube maps for the room's interior mapping, graphical artifacts seen on the wall's edges could be mitigated. If not entirely bypassed. These artifacts come from the fact that we're creating intrusions unto the wall, instead of extruding it, making connecting edges a difficult endeavour.

References

[1] Victor Gordan. Opengl tutorial 28 - parallax occlusion mapping. `https://youtu.be/LrnE5f3h2SU`, 2021. Accessed on November 29, 2022.

[2] Natalya Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. 2006.

[3] Antti Huima. Fast sigmoid algorithm. `https://stackoverflow.com/questions/10732027/fast-sigmoid-algorithm`, 2012. Accessed on December 5, 2022.