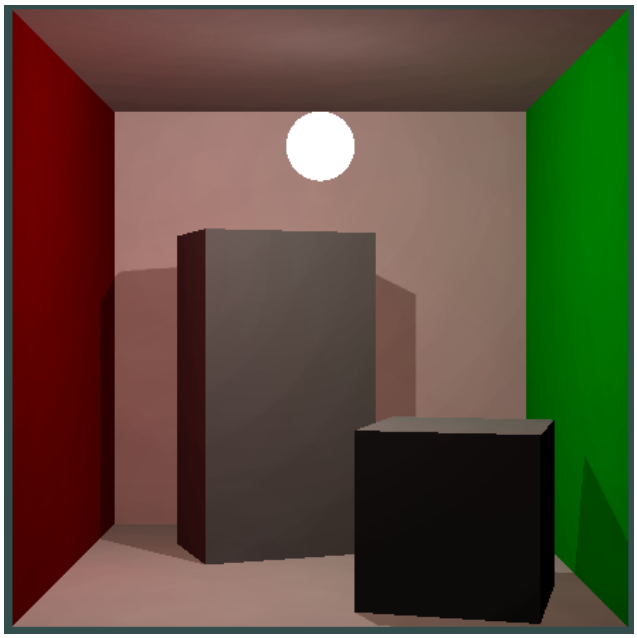
Virtual Point Lights

Emanuele Franchi and Lorenzo Ferrante



Introduction

The aim of this work is to implement the global illumination technique presented by Alexander Keller in his 1997 paper titled Instant Radiosity.

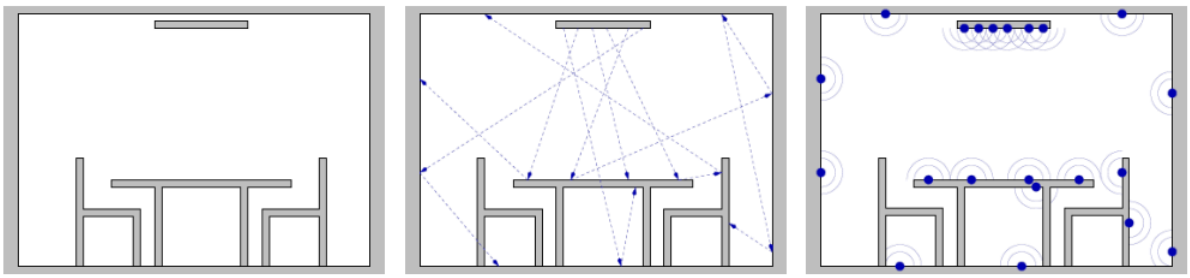
The reason we choose to implement this technique stems from its inherent simplicity. Keller tries to approximate the diffuse global illumination of the scene by shooting off random rays from the light source, spawning virtual light sources where they land, and lighting the scene using these virtual lights, together with the main one. Hence, this technique reduces the problem of computing global illumination to directly lighting the scene using multiple lights, thing that we well know how to do.

When Keller proposed this technique, graphics hardware was still in its infancy. Even using the latest hardware at the time, the achieved rendering rate was in the orders of seconds. This, obviously is far from the requirement we have for real time applications, which is in the orders of milliseconds. Hence, another point of interest for us has been revisiting this technique, with modern graphics hardware and rendering APIs (DirectX 11).

In our work, for simplicity's sake, we have not implemented the full fledged technique proposed by Keller, but a simplified version. During our discussion of the technique, we will mention whenever we deviate from it.

Keller's Instant Radiosity

In this section we illustrate the instant radiosity technique proposed by Keller.



(a) Scene (b) Shooting rays into the scene (c) Using hit points as VPLs

We suppose, without lack of generality, that there is a single light source in the scene. Extending the technique to multiple light sources is trivial. The light source itself is modeled by a light emitting surface.

Let:

- N be the number of particles to start off the light source.
- Let ρ be the scene's mean reflectivity. Note that $\rho \in (0,1)$.

We start by randomly spawning N particles on the light's surface.

$$p_0, p_1, p_2, \ldots, p_{N-1}$$

Each particle spawns a virtual light in its position. This is step zero of the particle simulation. Then, the first $\lfloor \rho N \rfloor$ particles are shot in the scene following random directions. Where they land we spawn a virtual light. This is step one of the particle simulation. Next, the first $\lfloor \rho^2 N \rfloor$ particles bounce off the geometry they hit, continuing their travel. Where they land, we spawn a virtual light. This is step two of the particle simulation. We continue until we reach the first step j such that $\lfloor \rho^j N \rfloor$ is zero.

Each particle stores the amount of light that it carries. When a particle is spawned, the amount of light it carries is derived from the light source. Each time a particle bounces off a surface with diffuse reflectivity ρ_d, the amount of light it carries is attenuated by ρ_d / π. When a particle bounces off a surface, we also spawn a virtual light. The amount of light emitted by this virtual light is the attenuated light carried by the particle.

At the end of the simulation, we will have spawned M virtual light sources

$$l_0, l_1, l_2, \ldots, l_{M-1}$$

each one emitting light

$$L_0, L_1, L_2, \ldots, L_{M-1}$$

For each virtual light l_i, we render the whole scene with shadows using l_i as the **only** point light source, storing the resulting image in the buffer b_i.

When rendering the scene using virtual light l_i as the only point light source, the amount of emitted light L_i is multiplied by $N/\lfloor w \rfloor$. The term $w = \rho^j N$ is used to compensate for the light attenuation applied during the simulation. Here, j is the simulation step in which light l_i was spawned.

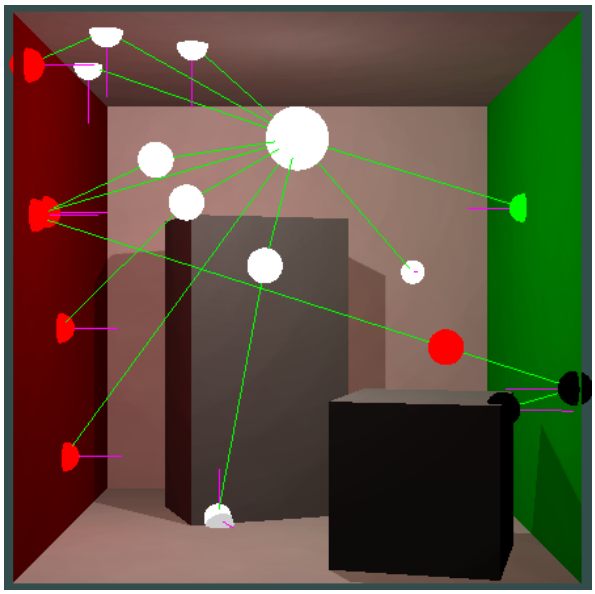
At the end, we sum together all the buffers

$$b_0, b_1, b_2, \ldots, b_{M-1}$$

into the final frame. When being summed, all the buffers are weighted by 1/N.

Our Version of Keller's Instant Radiosity

In this section we discuss our simplified version of Keller's technique.



Particle simulation on the example scene with N=10 and ρ=0.5.

Like Keller, we suppose that there is a single light source in the scene. Unlike Keller, this light source is a point light, instead of a light emitting surface.

We start by spawning N particles, at the point light position.

$$p_0, p_1, p_2, \ldots, p_{N-1}$$

The particle simulation we run is slightly different from Keller's. First and foremost, we don't spawn virtual lights where we have spawned our particles. Then, instead of shooting off the first $\lfloor \rho N \rfloor$ particles into the scene, we shoot all N particles. In the next simulation step, the first $\lfloor \rho^2 N \rfloor$ particles continue their travel into the scene, and so on.

Exactly as Keller's, our particles store the amount of light they carry. At the start, their amount of light is the same as the amount of light emitted by our point light. Each time a particle bounces off a surface, we attenuate its light in the same way as Keller.

Exactly as Keller's, every time a particle bounces off a surface, we spawn a virtual light. Its emitted light is the amount of light carried by the particle, once attenuated.

We render the whole scene:

- Using the point light as the only light source, with shadows.
- For each virtual light, using the virtual light as the only light source, **without shadows**, unlike Keller.

When rendering the scene using the various virtual lights, we still multiply each virtual light's intensity by $N/\lfloor \rho N \rfloor$.

Lastly, exactly as Keller, we sum together all the rendered buffers, each one having weight 1/N.

Implementation

In this section we delve into more technical details of our implementation of instant radiosity.

Particle Simulation

The particle simulation is implemented through a series of ray casts.

At the start, we randomly generate N rays, starting from the point light's position, along random directions. These random directions are chosen by randomly picking a point on the unit sphere, using a uniform distribution.

A particle being shot into the scene is implemented through a ray cast. Given a ray, we iterate over each object in the scene, doing a ray-object intersection test. This intersection test returns the closest point of intersection. We also keep track of the closest point of intersection along the ray, among all objects in the scene. In this way, at the end of the loop, we will have the closest intersection point between the ray and the scene.

The particles also need to bounce off the geometry they hit. To implement this behaviour, when a particle should bounce, we perform another ray cast, starting from the previously found point of intersection. The direction of this new ray cast is given by the reflection of the direction of the previous ray cast, over the object's normal at the intersection point. This means that, when computing the ray-object intersection, we also need to compute the object's normal at the intersection point.

Intersection Tests

In our scene we have only quads and boxes. Thus, we have implemented two types of ray-object intersection test: ray-quad and ray-box.

Ray-Quad

We do the ray-quad intersection test in local space.

Given a world space ray, represented by an origin **o** and a direction **d**, we convert it to local space by applying to both the inverse of the quad's model matrix. We use homogeneous coordinates for both **o** and **d**: for **o** we use w=1, for **d** we use w=0. The w=0 for **d** makes it so that **d** is not affected by translations. This is customary, since we don't want to translate a direction vector; translating it would change the direction.

We start by checking whether the ray intersects the plane π on which the quad lays. The equation to use is

$$\mathbf{p} \cdot \mathbf{n} + s = 0$$

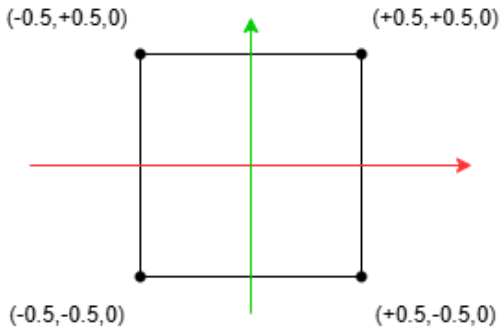
where **p** is a generic point, **n** is the plane's normal and s is the plane's offset from the origin of the reference system. To find the intersection point, we plug the ray equation into the plane equation

$$(\mathbf{o} + t\mathbf{d}) \cdot \mathbf{n} + s = 0$$

By construction, the local space quad lies on plane z=0. Thus we know that **n**=(0,0,1) and s=0.

$$(\mathbf{o} + t\mathbf{d}) \cdot (0,0,1) = 0$$
$$\mathbf{o} \cdot (0,0,1) + t\mathbf{d} \cdot (0,0,1) = 0$$
$$o_z + td_z = 0$$
$$td_z = -o_z$$
$$t = -o_z / d_z$$

If $d_z \neq 0$, then the intersection point exists. Since we have a ray, it must also be that $t > 0$, otherwise the intersection would be behind **o**. If both these conditions are met, then the intersection point **q** between the ray and the plane is given by plugging the t we found into the ray equation.



Local space quad.

Things are not over yet: we also need to check whether **q** is inside the quad or not. By construction, any point on the quad must have $x \in [-0.5,+0.5]$ and $y \in [-0.5,+0.5]$. If both conditions are satisfied, then **q** is inside the quad. This is our intersection point.

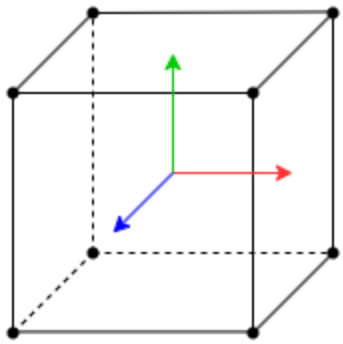
We have found the intersection point in local space. We need to convert it to world space. To do this we just transform it by the quad's model matrix.

We also need to find the object's normal at the point of intersection. In local space, the normal at the point of intersection coincides with the plane's normal: (0,0,1). We then transform it by the quad's normal matrix.

Ray-Box

We do the ray-box intersection test in local space.

We convert the ray to local space, exactly as we did for the ray-quad intersection test.



Local space box.

By construction, the local space box is the AABB with extremes (-0.5,-0.5,-0.5) and (+0.5,+0.5,+0.5). We use the slab method for finding the ray-box intersection.

The box is made out of three slabs. Each slab is defined by two parallel planes. Finding the ray-slab intersection is trivial: we just perform two ray-plane intersections (look at the ray-quad intersection section). If we are not in a degenerate case, these two intersection tests will yield two parameters: t' and t''. Let t_{min} = min(t',t'') and t_{max} = max(t',t''). Thus, the ray-slab intersection will yield as a result an interval $[t_{min}, t_{max}]$ of parameters to plug into the ray equation that represent the set of points laying on the ray that overlap the slab.

Having three slabs, one perpendicular to the x axis, one perpendicular to the y axis, and another perpendicular to the z axis, we have three intervals to work with $[t^x_{min}, t^x_{max}]$, $[t^y_{min}, t^y_{max}]$ and $[t^z_{min}, t^z_{max}]$. The ray intersects the box if and only if

$$[t^x_{min}, t^x_{max}] \cap [t^y_{min}, t^y_{max}] \cap [t^z_{min}, t^z_{max}] \neq \varnothing.$$

If this is the case, let the interval [a,b] be the result of the intersection.

The interval [a,b] is the result of intersecting the ray and the AABB. For us, a valid intersection is one given by a non negative t. Thus, we need to compute $[a,b] \cap [0,+\infty]$. If the intersection is not empty, let $[t_{min},t_{max}]$ be the resulting interval. The intersection point between the ray and the AABB is obtained by plugging t_{min} into the ray equation.

We also want the surface's normal at the intersection point. From the intersection point we found, it is easy to know on which side of the box it lies, and thus our normal in local space.

We use the same logic we did in the ray-quad intersection test to convert our results from local space to world space.

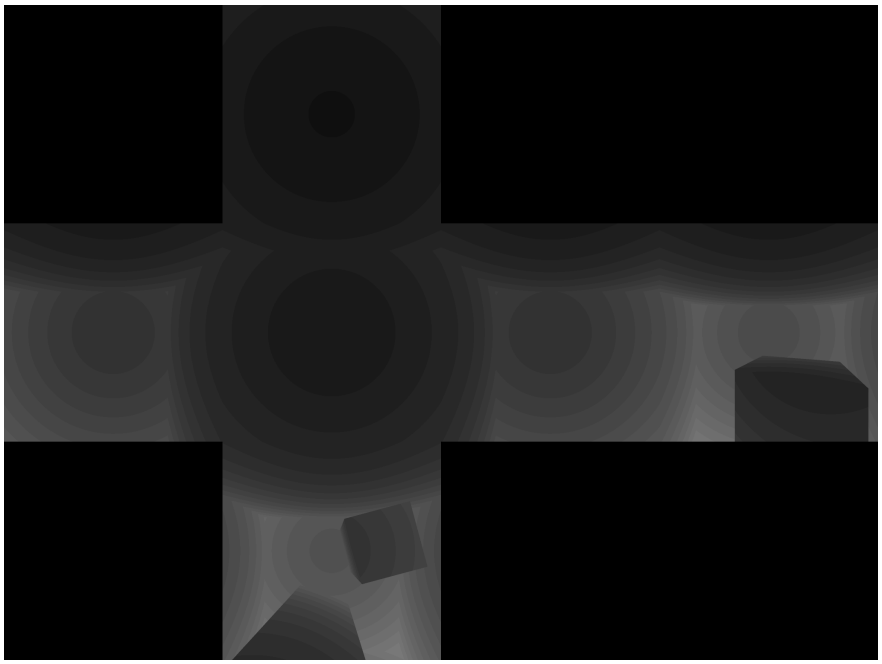
Rendering

Our rendering can be subdivided into three steps:

1. Rendering the cube shadow map of the point light.
2. Rendering the whole scene, with shadows, using the point light as the only light source.

3. For each virtual light, we render the whole scene, without shadows, using the virtual light as the only light source.

Rendering the Cube Shadow Map



Cube shadow map.

The first step consists in rendering the cube shadow map that will be used to render the scene with shadows. To do this, we need to render the scene six times, one for each side of the cube map.

As we know, to render the scene, we need to set up a view matrix and a projection matrix. The projection matrix we use is the same for all six sides of the cube: we use perspective projection with a FOV of 90 degrees and an aspect ratio of 1. The aspect ratio of 1 is trivial: both sides of a cube face have the same length. We use a FOV of 90 degrees so that the six frustums together omnidirectionally capture the scene.

Contrary to what we do with the projection matrix, we need six view matrices, one for each side of the cube. This is obvious: each face of the cube faces a different direction. We use the look-at technique: the camera position is the point light position, the camera forward and up vectors depend on the cube face we are rendering.

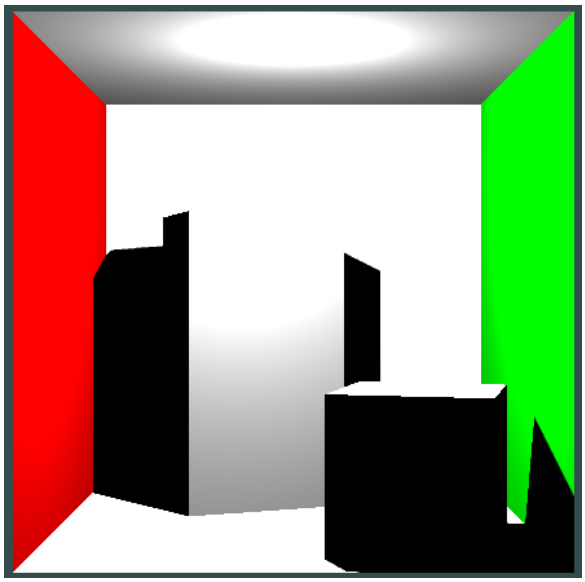
The vertex shader we use performs the usual computation of the clip space vertex position. It also outputs the world space vertex position.

The pixel shader we use only writes to the depth buffer. The written depth is the world space distance between the camera's position and the fragment's position. Before actually writing this

distance as depth, we need to scale it accordingly. This is because depth values get clipped on the range [0,1]. Obviously, our distance doesn't necessarily lie in this range. What we can do is divide this distance by the value of the far plane used for building the projection matrix. This scales it into the range [0,1].

Writing the world space distance as depth, instead of the actual fragment's depth, allows us to implement a straightforward way of doing comparisons with the shadow map samples, when evaluating the lighting model.

Rendering the Point Light with Shadows



Scene lit from the point light, with shadows.

On the application side, there is nothing interesting about rendering the scene and lighting it using the point light. We simply loop over each object and issue a draw call for it. The only thing to note is that we directly write to the default color and depth buffers; we don't use auxiliary buffers to accumulate the contributions of the various lights.

The vertex shader we use performs the usual computation of the clip space vertex position. It also outputs the world space vertex position and the vertex normal, transformed by the normal matrix.

It's in the pixel shader where the fun happens. Let:

- ρ_d be the object's diffuse color.
- **n** be the fragment surface normal.
- **l** be the world space vector from the fragment's position towards the point light's position.
- **L** be the light emitted by the point light.

The fragment color is computed as follows

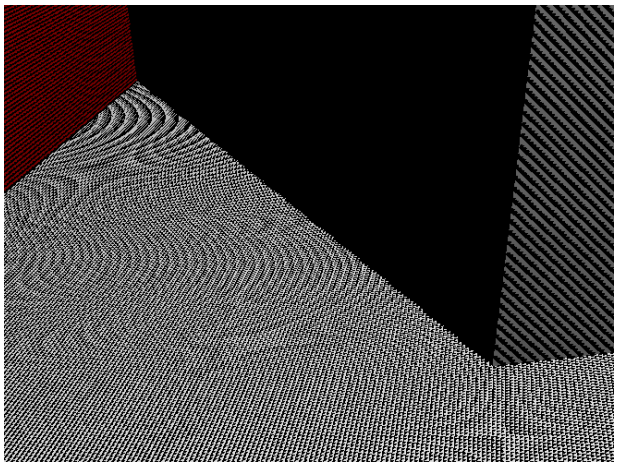
$$1/N * \rho_d/\pi * (1 - s) * \mathbf{L} * \max(\mathbf{n} \cdot \mathbf{l}, 0)$$

where:

- N is the number of particles we have used for the particle simulation.
- s ∈ [0,1] is the shadow factor.

The interesting part is the way in which we compute the shadow factor. We first consider the vector -**l**, that is, the world space vector going from the point light toward the fragment. We use -**l** to sample the shadow cube map, reading back distance d. We scale this distance by the value representing the far plane used for the light's projection matrix. This undoes the mapping to [0,1] we did when we built the shadow cube map. Now d is the proper world space distance between the light and its closest point along direction -**l**. Then, we compare d with the length of -**l**, which is the world space distance between the light and the fragment. If ||-**l**|| > d, then the fragment must be shadowed, thus s=1. If ||-**l**|| ⩽ d, then the fragment must be lit, thus s=0.

This is not the full story. There are two problems we need to address: **shadow acne** and **jagged shadows**.

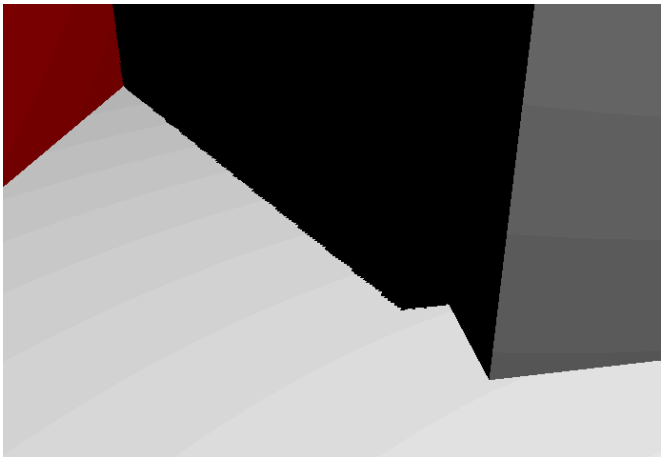


Example of shadow acne.

The major cause of shadow acne is the fact that one shadow map texel may get sampled by many fragments, each one generally having a different distance from the light source, compared to the sampled distance. Thus, some of these fragments will be shadowed and others will be lit. This is what gives origin to the banding effect of shadow acne. Moreover, the difference in distance gets bigger, the more the light vector deviates from the surface normal.

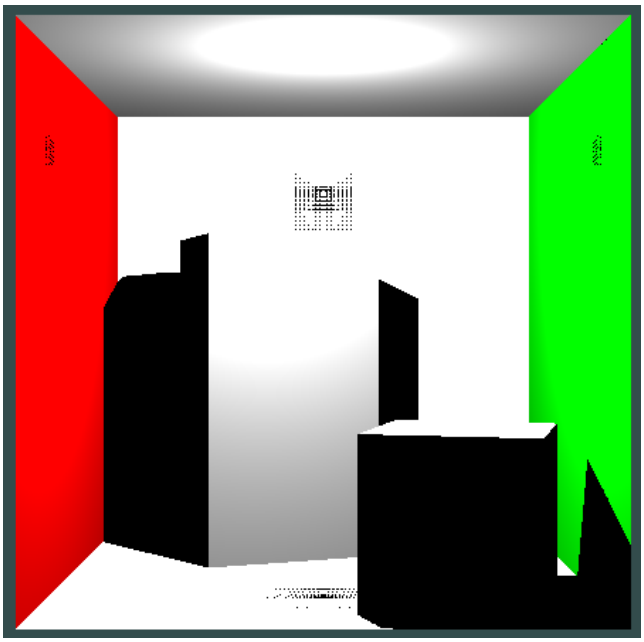
To fix shadow acne, we either have to bias d or ||-**l**||. We choose to bias ||-**l**||. This means that, instead of comparing ||-**l**|| and d, we compare ||-**l**||-bias and d.

The simplest thing we can do is use a static bias. The more shadow acne we have, the more we increase the bias. With high enough values for the bias, it becomes apparent that shadows start to shift from the objects that cast them. This is what's called **peter panning**. Thus, choosing the right bias is an act of balancing between avoiding the shadow acne and having as little peter panning as possible.



Example of peter panning.

Using a static bias may be too limiting. This is even more so when we observe that the severity of shadow acne may be different across a model, due to different surface normals and/or light direction vectors. In this case, using a static bias, we need to choose a high enough value to cover the most severe cases of shadow acne, even though for most of the mesh a smaller value would have been sufficient. From here comes the idea of a **dynamic bias**.



Example of only using the dynamic bias.

We compute a dynamic bias value, based on the angle between **l** and **n**. The more **l** and **n** are aligned, the less the bias we need. The more **l** and **n** deviate, the higher the bias we need. It is easy to see that we get what we want by taking 1 - **n** . **l**.

It is interesting to note that we can't just use a dynamic bias. When **l** and **n** are parallel, the bias would be zero. In this case we would still have shadow acne. Thus, we combine both a static bias b_s and a dynamic bias b_d. We compute the bias as

$$\text{bias} = b_s + b_{dmax}b_d$$

where b_{dmax} is used as a scaling factor for b_d; it represents the maximum allowed dynamic bias.

Example of jagged shadows.

The other problem we face are jagged shadows. The cause is the limited resolution of shadow maps. Resolution that, in general, is not high enough for shadows to look naturally smooth. From anti-aliasing techniques, we know that jaggies can be mitigated by smoothing them using some sort of filter. A very common filtering technique for shadow maps would be percentage-closer filtering, **PCF** for short. PCF works by taking multiple samples of the shadow map, using them for the shadow comparisons, and then computing the shadow factor as a percentage of the number of samples that pass (or fail) the comparison.

Example of PCF.

Let:

- S be the number of cube shadow map samples we take.

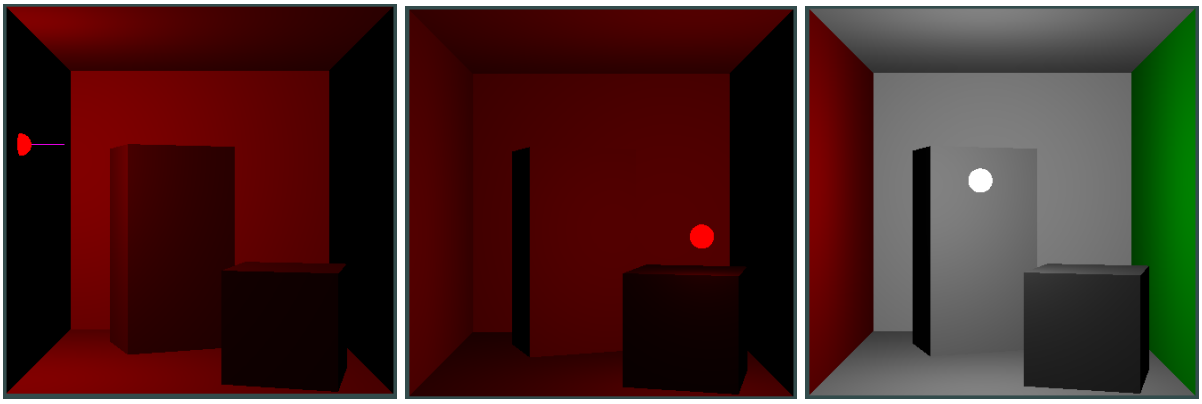
- $\mathbf{o}_0$, $\mathbf{o}_1$, $\mathbf{o}_2$, …, $\mathbf{o}_{S-1}$ be S random offset vectors.
- off be a scaling factor for the random offset vectors.

Instead of reading only one sample off the shadow cube map, we read S samples. Sample i is given by the vector

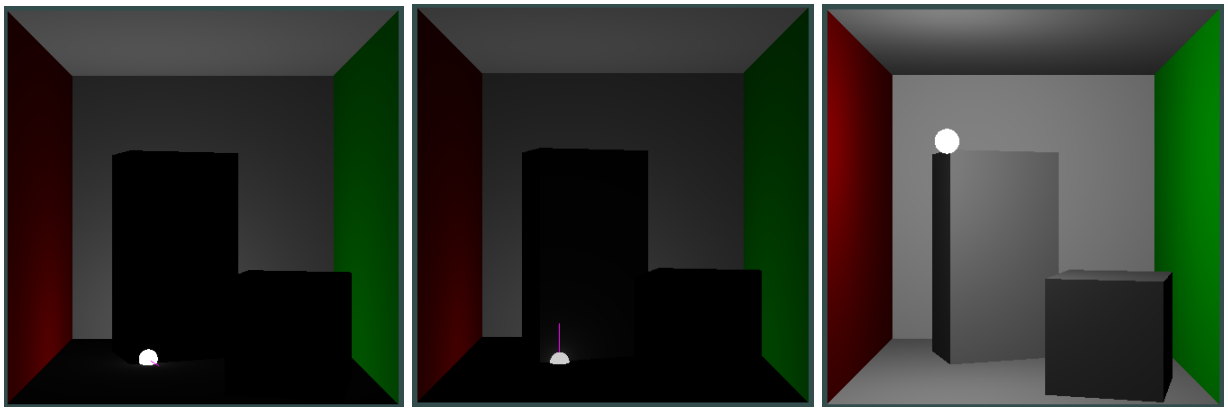
$$-\mathbf{l} + off * \mathbf{o}_i$$

For each sample we read, we compare it with ||-**l**||-bias. Every time the comparison tells us that the fragment should be shadowed, we increment s by one. After comparing all the S samples, we just need to normalize s. We do it by dividing it by S.

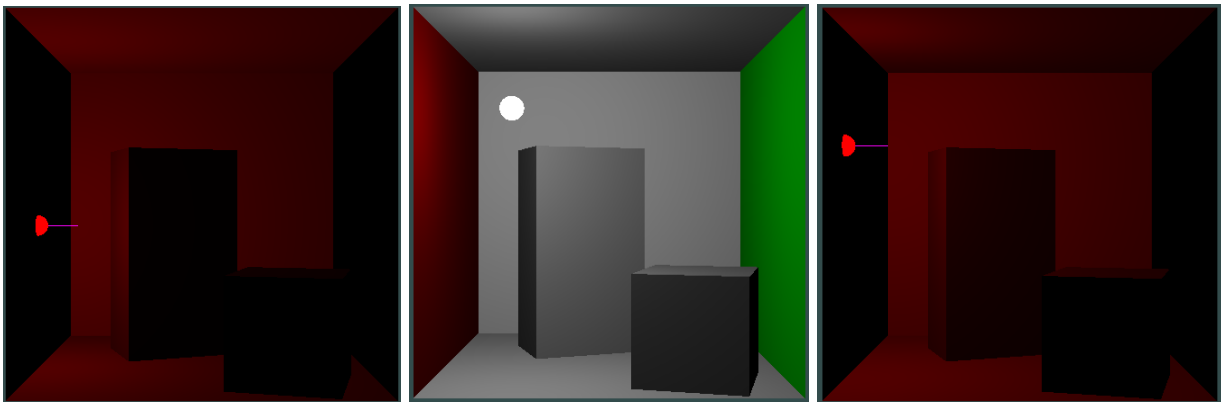
Rendering the Virtual Lights without Shadows



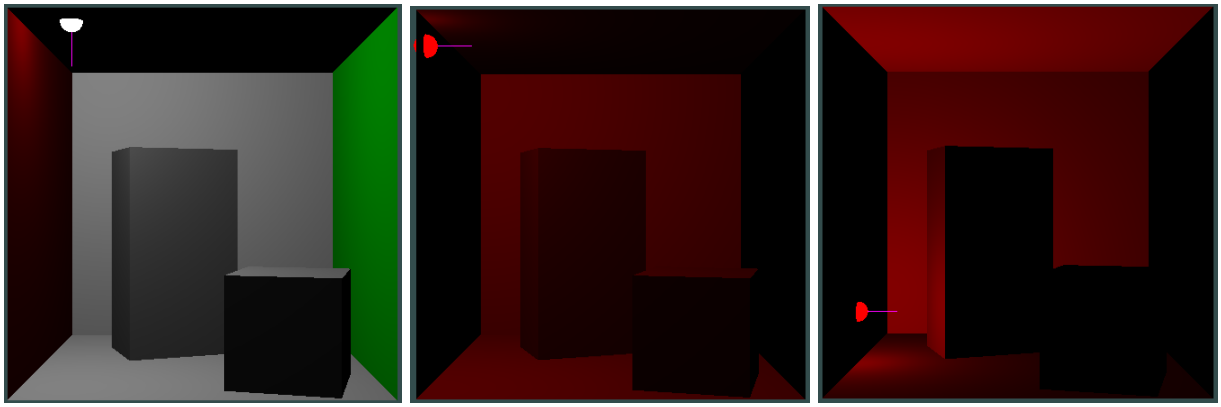
Left: Virtual Light 1 - Center: Virtual Light 2 - Right: Virtual Light 5.
The missing virtual lights have been skipped because they were irrelevant.



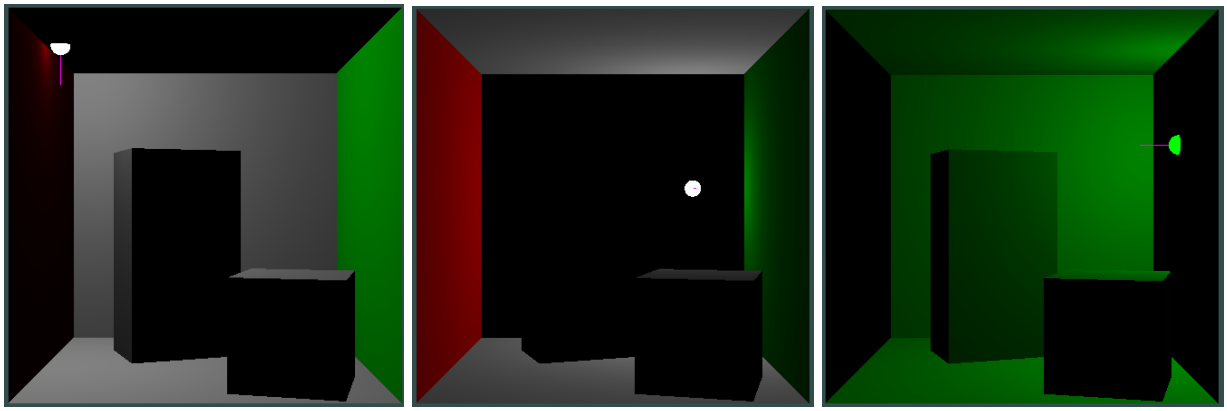
Left: Virtual Light 6 - Center: Virtual Light 7 - Right: Virtual Light 8.



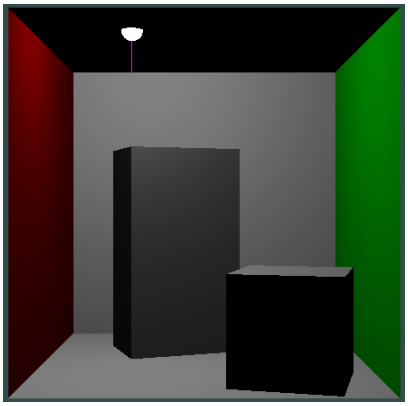
Left: Virtual Light 9 - Center: Virtual Light 10 - Right: Virtual Light 11.



Left: Virtual Light 12 - Center: Virtual Light 13 - Right: Virtual Light 14.



Left: Virtual Light 15 - Center: Virtual Light 16 - Left: Virtual Light 17.



Virtual Light 18.

For every virtual light, we render the scene, without shadows, using the light as the only light source. The vertex shader we use is the usual, and the pixel shader is the same as the one we used for rendering the scene with the point light, obviously without the shadow mapping part.

The interesting thing here is how we accumulate the various renderings of the scene, using only one color buffer and one depth buffer.

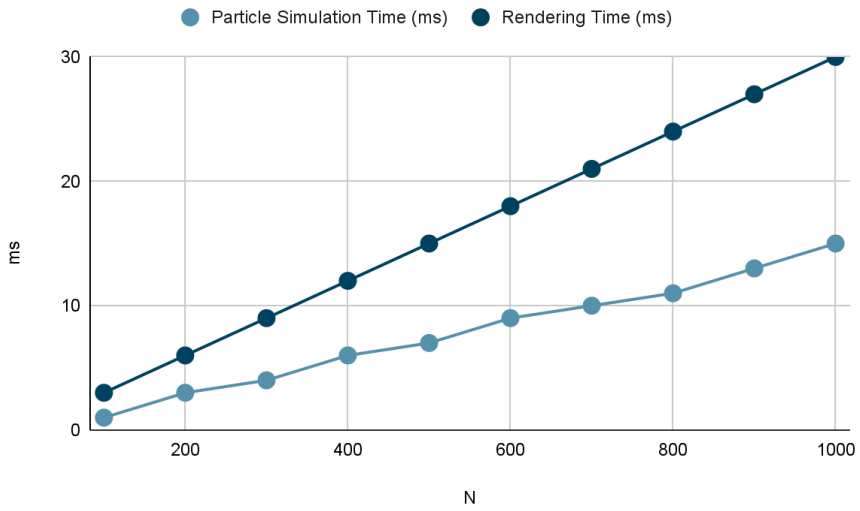
Since we have already rendered the scene, lighting it with the point light, the color buffer has already been populated. Then, before rendering the scene with the virtual lights, we configure the output merger stage: instead of writing the fragment's color to the color buffer, we sum the fragment's color with the color in its corresponding texel of the color buffer. In this way, we are summing the weighted (we divide by N inside the pixel shader) contributions of the various light sources all together.

This is a good idea, but there is a catch. What do we do with the depth buffer? This buffer has already been populated when the scene was rendered for the point light. If we were to clear it every time we render a virtual light, we would get incorrect results. Imagine this scenario. We issue a draw call for object A, which passes the depth test, and thus its fragments get summed to the contents of the color buffer. Then we render object B, which occludes A. B passes the depth test, and its fragments get summed to the contents of the color buffer. The color buffer now contains incorrect results. This is because we must only sum the colors of the fragments that we know for certain are visible to the camera, once the whole scene has already been rendered.

From the previous discussion, it is obvious that we cannot clear the depth buffer. What if we leave things as is? If we keep the contents of the depth buffer, we already know which fragments are already visible: exactly those whose depth is equal to the depth in the corresponding depth texel. This is what we want. We are on the right track. But we still get incorrect results. This is because the comparison function used by default for depth testing is "less than". This means that the fragments that pass the depth test are only those whose depth is less than the corresponding depth in the depth buffer; otherwise they get discarded. Since in the depth buffer we are storing the depths of the closest fragments to the camera, all the

fragments we render cannot pass the depth test, and thus get thrown away. In this case, we don't write anything to the color buffer. We fix this by changing the comparison function with "equal to". Now, the fragments that pass the depth test are only those whose depth is equal to the corresponding depth in the depth buffer. The fragments that pass the depth test are thus exactly those fragments that are visible to the camera and thus we sum their colors to the corresponding contents of the color buffer.

Performance



There are mainly two points of interest regarding the performance of instant radiosity: the particle simulation and the various renderings of the scene.

Asymptotic Time Complexity for the Particle Simulation

Here we study the simulation's complexity. Since the simulation is done through a series of ray-object intersections, we measure its complexity by counting how many intersection tests we perform.

We start with N particles. We shoot them into the scene. Each particle being shot is implemented as a ray-object intersection test, for every object in the scene. Then, the first $\lfloor \rho N \rfloor$ particles continue their travel, and so on, until we reach simulation step j such that $\lfloor \rho^j N \rfloor$ is zero.

Having B objects in the scene, the number of ray-object intersection tests performed by simulation step 0 are B * N. The number of ray-object intersection tests performed by simulation step 1 are B * $\lfloor \rho N \rfloor$. And so on. Thus, the total number of tests is

$$B * N + B * \lfloor \rho N \rfloor + B * \lfloor \rho^2 N \rfloor + B * \lfloor \rho^3 N \rfloor + \dots$$

This sum can be written as

$$\sum_{k=0}^{j-1} B\lfloor \rho^k N \rfloor \leq \sum_{k=0}^{\infty} B\lfloor \rho^k N \rfloor = B \sum_{k=0}^{\infty} \lfloor \rho^k N \rfloor \leq B \sum_{k=0}^{\infty} \rho^k N = BN \sum_{k=0}^{\infty} \rho^k = BN \frac{1}{1-\rho}$$

Hence, the simulation's time complexity is

$$O(BN\frac{1}{1-\rho})$$

Improving the Particle Simulation

In our case, we opted for a brute force approach, when doing ray casts. However, for complex scenes, it would not be viable to do so. Surely, a spatial data structure should be used to speed up such queries. A BVH seems the most appropriate solution. There is also ample literature on using a BVH to speed up ray casts.

Mind that, for instant radiosity, we don't really need to find the precise intersection point between a ray and an object's mesh. This means that we don't really need to go at a mesh primitive level. Stopping at the object's OBB would be enough for most cases. For the intersection, we just need a position and a normal that are good enough.

Asymptotic Time Complexity for the Rendering

Here we study the complexity of rendering our scene. When talking about rendering, a good metric to use is the number of draw calls performed to render the frame. A draw call is an expensive operation since it requires activating the entire graphics pipeline.

Suppose we have B objects in the scene. For rendering the shadow cube map, we render the scene six times, each time drawing all the B objects in the scene. This takes B * 6 draw calls.

For rendering the scene lit by the point light, we submit B draw calls.

For rendering the scene lit by each virtual light source we submit M * B draw calls.

At the end, the number of performed draw calls is

$$6B + B + MB = (7 + M)B$$

Hence, the rendering time complexity is

$$O(MB)$$

Improving the Rendering

In our case, we took to the letter the idea of rendering the scene for each light source. At the end of the day, this is what Keller states.

Nonetheless, things can be made much faster by decreasing the number of draw calls. We can do so by batching together, into a single pixel shader invocation, the computation of more than one contribution to the lighting from the virtual lights.

We would do this by uploading to the pixel shader, not a single virtual light, but m virtual lights altogether. Then, when running the shader, we would loop on the uploaded virtual lights, and for each one, compute its contribution to the lighting. The sum of these contributions would then be divided by N.

This brings the number of draw calls to $(7 + \frac{M}{m})B$, which is still $O(MB)$, but the multiplicative constant may be much lower, depending on the size of m. This depends on how big of a buffer we can pass to the pixel shader. Usually we have a guaranteed 128MB of data for this purpose. Supposing we use 64 bytes to represent a virtual light, which is more than enough, m could roughly be two millions.

Considerations and Future Work

It has been very fun for us to work on this project. Implementing instant radiosity has allowed us to dip our toes into global illumination algorithms, without being overly complex. Moreover, despite its simplicity, we still managed to achieve good looking results. As it is, we have shown the feasibility of the technique for real time scenarios, at least for simple scenes. Its feasibility has yet to be discussed for more complex scenes. We have also illustrated how to implement this technique using a modern graphics API (DirectX 11).

We would like to suggest two routes, for readers interested in further developing this work.

The first route consists in implementing better logic for the placement of the virtual lights. In particular, we only care about virtual lights that influence the part of the scene we are looking at.

The second route stems from the following observation: the more a particle bounces, the more the light it carries gets attenuated. Experimentally, even a small number of bounces almost entirely disperses all the particle's light. This means that the virtual lights spawned from further bounces are mostly irrelevant. One could even ignore bounces and just stop at the first surface point that the particle hits. Finding these points can be done by rendering the scene from the point light's POV: this is the idea behind reflective shadow maps.

Bibliography

Keller, Alexander. "Instant radiosity." Proceedings of the 24th annual conference on Computer graphics and interactive techniques. 1997.

Appendix A: Libraries

The libraries used for this project are:

- DirectX 11, feature level 11.0.
- DXGI, v1.3.
- Dear imgui, v1.91.9b.
- DirectX Toolkit's Simple Math, March 2025 release.

Appendix B: Test Machine Specs

These are the test machine's specs:

- CPU: AMD Ryzen 5 5600X 6-cores.
- GPU: Nvidia GeForce RTX 3060 Ti.
- RAM: 32GB 3000MHz.