Πτυχιακή Εργασία

«Implementation of Diffuse Global Illumination using Imperfect Shadow Maps»

Φουναριωτάκης Εμμανουήλ

3060185

Επιβλέπων: Παπαϊωάννου Γεώργιος

ΑΘΗΝΑ, ΔΕΚΕΜΒΡΙΟΣ 2011
# Table of Contents

1. Introduction .......................................................................................................................... 3

2. Related Work .......................................................................................................................... 4
   2.1. Instant Radiosity Methods .............................................................................................. 7

3. Imperfect Shadow Maps ......................................................................................................... 8
   3.1. Introduction .................................................................................................................... 8
   3.2. Scene Representation ...................................................................................................... 8
   3.3. ISMs creation .................................................................................................................. 8
   3.4. Parabolic Mapping ........................................................................................................ 9
   3.5. Pull-Push ........................................................................................................................ 11
       Outlier Rejection ........................................................................................................... 12
   3.6. Multiple Bounces ........................................................................................................ 12
   3.7. Limitations .................................................................................................................... 13
   3.8. View-Adaptive Imperfect Shadow Maps ....................................................................... 13

4. Illumination ............................................................................................................................. 15
   4.1. The Rendering Equation ................................................................................................ 15
   4.2. Instant Radiosity ............................................................................................................ 16
   4.3. Computing Indirect Illumination using Reflective Shadow Maps .................................. 16
       4.3.1. Evaluation ............................................................................................................ 17
           a. Per point evaluation based on spatial proximity ................................................... 17
              Screen Space Interpolation .................................................................................. 18
           b. Evaluation using importance sampling .............................................................. 18

5. Application Architecture ......................................................................................................... 21
   5.1. Overview ........................................................................................................................ 21
       Pre-computation ............................................................................................................. 21
       Runtime .......................................................................................................................... 21
   5.2. Scene tessellation .......................................................................................................... 22
   5.3. Deffered shading ........................................................................................................... 22
   5.4. RSM ............................................................................................................................. 26
   5.5. ISM ............................................................................................................................... 27
   5.6. Development Environment .......................................................................................... 30
   5.7. Application steps' representation ................................................................................. 30

6. Results ................................................................................................................................... 32
   6.1. Room ............................................................................................................................. 32
   6.2. Sponza Atrium ............................................................................................................... 36

References .................................................................................................................................... 38
1 Introduction

Without doubt, the main challenge in the field of computer graphics is that of photorealism. Producing images taking into account all the physical laws applied in nature has been a goal of great importance and thorough research has been directed towards it. A physically accurate representation of a scene depends on a variety of factors including the scene's description detail, the choice of a suitable form to model its object's properties, the quality of the two dimensional images used and so on. Probably the most crucial factor in generating a convincing image is the accurate computation of lighting, i.e. the interaction between objects and light. Light completely determines the way we perceive our surroundings and therefore the precise reproduction of the way it interacts with the world is a necessity in order to fulfill the need for photorealism. The whole set of effects caused by the interaction between light and materials is described by the term global illumination. Global illumination algorithms consider not only light coming directly from a light source but also light reflected by other surfaces in the scene. Effects grouped under this term include reflections, refractions, shadows etc.

Creation of images illustrating global illumination effects is feasible using offline rendering techniques. Path tracing, photon mapping and Radiosity are typical examples of algorithms capable of producing high quality results given the necessary amount of time.

In the field of real time rendering, surface shading from direct lighting (not accounting for shadows) is considered a solved problem today even for scenes of high complexity. The advancements on graphics acceleration hardware allow the fast completion of this task, even for a large number of point light sources (e.g. via deferred lighting). However, direct lighting is only a subcase of global illumination. Results can be convincing only if more global illumination effects are present. (Figure 1.1)

The small portion of time available for the computation of each frame though, does not suffice for achieving full global illumination. The inherent complexity of the problem, which involves the recursive sampling of light over many directions and the calculation of visibility between arbitrary points in the scene, makes current global illumination approaches not fast enough for realtime use or not accurate enough because of the simplifying assumptions used. Thus, fast computation of global illumination, suitable for real time applications remains an open problem.

In this thesis the implementation and results of two established diffuse global illumination variants of the instant radiosity method are presented. The first one, Reflective Shadow Maps is used to compute the light bounced from surfaces lit directly by the light source, which corresponds to the first bounce of indirect lighting. Because this technique does not consider occlusion information for indirect lighting a second technique called Imperfect Shadow Maps is used. This one, enables efficient shadow casting for indirect lighting using a rough approximation of the scene elements.
2 Related Work

Several methods for the computation of indirect illumination have been presented:

Dachsbacher and Stamminger presented Reflective Shadow Maps (RSM)\cite{1} in 2005. The main idea of their method is to extend a shadow map by additionally storing normal, position and reflected flux for each pixel. Thus each pixel of the shadow map can be considered as a virtual point light (VPL) and the stored information (position, normal, flux) can be used to gather illumination from each VPL. Because the huge amount of pixels contained in a typical shadow map of 512x512 makes the evaluation of all VPLs impossible for real time, the authors use a precomputed sampling pattern to reduce the final number of VPLs to a few hundreds. Furthermore, the computation of indirect illumination is done for an image of lower dimension followed by an interpolation in order to produce the final image. Reflective Shadow Maps do not consider occlusion for the secondary light sources and can produce only the first bounce of indirect illumination.

Splatting Indirect Illumination\cite{2} was published by Daschabacher and Stamminger in 2006. This method is based on their previous publication (Reflective Shadow Maps) but the process of gathering the indirect illumination from virtual point lights is transformed into a deferred shading process in order to improve performance. In addition, virtual points lights are sampled from the reflective shadow map by applying an importance wrapping in order to adapt the sampling pattern to the actual flux distribution of the RSM. Indirect occlusion is not computed as in reflective shadow maps and illumination is also gathered only for the first bounce.

Interactive Indirect Illumination using Adaptive Multiresolution Splatting\cite{3} by Nichols and Wyman, adopts the concept of reflective shadow maps in order to create virtual point lights but uses a multiresolution splatting approach to gather illumination from the VPLs thus exploiting the low frequency nature of indirect lighting. In their method, mipmap's storing depth and normal discontinuities are created. Then the original image is divided into a fixed amount of regions and each region is further subdivided if the discontinuity stored in the relevant mipmap exceeds a certain threshold. Consequently, the indirect illumination is rendered to a multiresolution illumination buffer and the results are combined to produce the final image. As in Reflective Shadow Maps and Splatting Indirect Illumination no occlusion is taken into account.

The three methods described above do not compute occlusion for indirect lighting. Imperfect Shadow Maps\cite{4} presented by Ritschel et al. in 2008 use a point based representation of the scene in order to achieve shadowing from virtual point lights. As a preprocessing step a number of points is selected from its object in order to form the scene's point representation. Additional data besides position are stored so dynamic objects
can be supported without re-creating the point set for each frame. In runtime each point is randomly assigned to a virtual point light so a low resolution parabolic shadow map can be drawed for each VPL. For typical settings, the number of points is not sufficient to completely fill the shadow maps so a pull-push pass is applied in order to fill the holes in each shadow map. Thus the illumination for each VPL is gathered taking occlusion into account. Although each individual shadow map is not completely accurate, smooth gradations of indirect lighting tends to mask errors produced by incorrect visibility.

Ward et al. introduced the concept of computing indirect illumination by using interpolation from cached values[5] in 1988. Instead of recalculating irradiance for each pixel the mean value over a small set of computed values is used. Irradiance for each of these values is accurately computed by casting a number of rays from the shaded point.

Wang et al.[6] presented a similar method implemented entirely on the GPU in 2009. Although his method is able to demonstrate complex effects such as multiple bounces of indirect lighting, caustics and glossy reflections it achieves interactive frame rates only in very simple scenes.

Irradiance Volume[7] introduced by Greger in 1998, maintains a two-level grid containing irradiance points which are used to interpolate irradiance for each pixel. The grid is created in a pre-process step by subdividing the bounding box of the scene into a regular grid. Further subdivision is applied to cells of the grid containing geometry. In runtime the four nearest sample points to the shaded surface location are discovered and a distance-weighted average pre-computed value is used as the surface's point irradiance value.

The work of Nijasure et al.[8] in 2005 is based on the same principle but uses a spherical harmonics representation to store the incident radiance at the vertices of a regular grid. Incident radiance is estimated by rendering a cube map at each grid point sampling the resulting texels and encoding the radiance field as spherical harmonics coefficients. The indirect illumination at the surface points is approximated by interpolating the radiance from the closest grid points. This method supports multiple bounces and indirect occlusion but is very expensive for complex dynamic scenes or many cache points due to the capture and encoding of the cube maps.

Radiance Hints[9] by Papaioannou presented in 2011, combines ideas from the grid-based radiance caching of Nijasure et al.[8] and the Reflective Shadow Maps approach. Similar to Nijasure et al.[8], it stores sparse radiance field values at regular volume locations. Instead of constructing cube maps to sample the visibility and incoming radiance, though, it employs reflective shadow map sampling in one pass for all sampling locations. This allows for a significant performance increase in radiance field computation at the selected points and does not require any additional rendering of the scene’s geometry, as all necessary information is already stored in the RSMs of the light sources. However, since RSMs do not convey any secondary light bounce occlusion information, shadowing is indirectly inferred from the distances of RSM samples and the available image-space depth, resulting in a potential loss of fidelity for secondary occlusion phenomena.

Global Illumination using Photon Maps[10] introduced by Jensen in 1996, illustrates a two pass global illumination method. In the first pass two photons maps are created (caustic photon map, global photon map) by emitting photons from the light sources. For each photon hitting a surface incoming flux is stored in a balanced kd tree. In the second pass the indirect illumination of visible surface points is approximated by gathering the k nearest photons to that point. Photon mapping essentially implements a cached scheme for light path tracing. The approximate incident flux is sampled during a second pass that performs one- or two-level path tracing from the camera, depending on the captured phenomena (caustics or diffuse global illumination respectively).

In McGuire and Luebke[11] the first bounce of the photons is computed using rasterization on the GPU, continues the photon tracing on the CPU for the rest of the bounces and finally scatters the illumination from the photons using the GPU. Since part of the photon tracing still runs on the CPU, a large number of parallel cores are required to achieve interactive frame-rates.
Recently Yao et al. [12] introduced a photon splatting method that calculates ray-surface intersections in the image space of multiple environment maps (with normal and depth information) carefully placed within the volume of the scene by optimizing volume coverage. The method dispenses with ray-polygon intersections by employing the Distance Impostors method proposed in Szirmay-Kalos et al.[13], which performs ray-environment map intersections.

Thiedemann et al. [14] introduced an interactive volume-based global illumination method, where the spatial occupancy and color data are generated by injecting a geometry texture atlas containing point samples of the polygonal geometry. The authors propose an optimized ray marching scheme for intersecting the gathering rays with the volume data. At each hit voxel, the RSMs are looked up to determine its visibility from the source, and subsequently its normal and the incident light. Their method produces fast and high quality first-bounce diffuse global illumination but requires model preprocessing and extra storage for the texture atlas and is sensitive to the point sampling rate and surface deformations. The authors suggest that secondary indirect light bounces should be handled by iterative path tracing but do not provide a solution.

Mavridis and Papaioannou adopted a similar volume population approach[15], where occupancy and direct illumination of the geometry are injected into the volume as vertices of the tessellated geometry. Supplementary points are generated by injecting the view camera and RSM G-buffers into the volume, similar to Cascaded Light Propagation Volumes[16] presented by Kaplanyan and Dachsbacher in 2010. Ray marching at voxel level is subsequently applied to simulated global illumination with multiple bounces at a cost proportional to the volume size and sampling distance. Both methods produce a full-scene occupancy volume and can thus correctly handle secondary light bounce occlusion.

Ritschel et al. [17] extended previous work for screen space ambient occlusion calculation (Shanmugam and Arikan[18]) and introduced a method to approximate the first indirect near field diffuse bounce of the light by only using information in the 2D frame buffer. This method has a very low computational cost but the resulting illumination is hardly accurate since it depends on the projection of the (visible only) objects on the screen.

Kaplanyan and Dachsbacher introduced a volume-based method, the Cascaded Light Propagation Volumes [18], where RSM-generated virtual point lights are injected into a volume texture. Instead of evaluating the low frequency indirect illumination from the VPLs at the surface points though, the method uses an iterative propagation scheme to transfer energy from voxel to voxel, taking into account occlusion caused by blocking voxels. Blocking voxels are marked by storing any available depth information from the camera depth map and the RSMs into a separate occlusion (geometry) volume. The method achieves high performance for a relatively small number of propagation iterations with respect to the volume size, which in most cases is not adequate to model multiple bounces of diffuse inter-reflection. The method also suffers from popping artifacts due to view-dependent occlusion information.

Implicit Visibility and Antiradiance for Interactive Global Illumination[19] by Dachsbacher and Stamper introduced the concept of antiradiance. Besides radiance, a new quantity, antiradiance, is propagated in the same way as normal radiance. Specifically, every surface point emits incident light backwards as "negative light". By iteratively applying this process the incorrect light traversed due to lack of occlusion computation is canceled out.

Crassin et al.[20] presented a new method for interactive indirect illumination capable of computing two light bounces for both lambertian and glossy materials. Their implementation is entirely implemented on GPU and relies on a voxel octree representation of the scene generated from triangle meshes and updated on runtime for dynamic objects. For each light, irradiance values and light directions are stored in the octree. Then, illumination for each point is gathered from the octree by tracing a number of cones over the point's hemisphere.
2.1 Instant Radiosity Methods

A wide group of methods is based on Instant Radiosity, introduced by Keller\[21\]. In the original paper, a fixed number of N particles is emitted from the primary light source and a virtual point light (VPL) is created for each particle-geometry intersection. Each VPL's properties are altered by the material properties of the hit point and then the scene is rendered once for each VPL using explicit occlusion information (shadow mapping). The above procedure is repeated (new particles are traced from some of the initial hit points) thus leading to the computation of multiple bounces of indirect illumination. The most expensive part of this method, as in most indirect illumination solutions, is the computation of occlusion information which has to be done for approximately 2N lights.

Lightcuts\[22\], a more recent method focused on reducing the final number of lights was presented by Walter et al. in 2005. In this method lights are divided into clusters and illumination from each cluster is approximated by a representative light defined for its cluster thus reducing the number of shadow maps computed. Individual lights and light clusters are represented in the form of a binary tree with its leaf representing a light and its interior node representing a cluster. Individual lights are grouped into clusters using criteria such as spatial proximity and similar orientation. A lightcut is chosen locally in order to provide increased accuracy by replacing each node with its children until the error introduced by its node falls below a certain threshold. The set of all the selected nodes defines the lightcut.

A similar approach is presented by Dong in Real Time indirect Illumination with clustered visibility\[23\]. Dong introduces the concept of virtual area light (VAL), an area light built by combining a number of virtual point lights. Each VAL is created by combining VPL with similar positions and orientation (normals). The VPLs' positions and normals are averaged to compute the position and normal of the produced VAL. The VAL's are used only for computing the occlusion for indirect light sources by applying a soft shadow algorithm whereas illumination is computed by using the initial VPLs.

Another method based on Instant Radiosity is Bidirectional Instant Radiosity\[24\] published by Segovia et al. in 2006. Bidirectional instant radiosity extends instant radiosity in the same way bidirectional photon mapping extends classic photon mapping. Virtual point lights are not only generated from the light but also from the camera by generating paths from the camera's origin (reverse Instant Radiosity). VPL’s are placed at the end of these paths (reverse VPLs). As stated in the paper, using VPLs created from the camera is not the best choice in all cases so this methods combines both classic Instant Radiosity and Reverse Instant Radiosity by creating an equal number of "classic" VPLs and camera VPLs.

One more variant of Instant Radiosity, Incremental Instant Radiosity for Real-Time Indirect Illumination\[25\] was presented by Laine et al. in 2007. Instead of recomputing all VPLs for each frame the authors use a mechanism -based on computational geometry- to determine the validity of each VPL in order to recompute only a small number of VPL's. Their mechanism makes feasible the use of a paraboloid shadow map for each VPL as only a small number of shadow map needs to be created in each frame. The major drawback of this method is that only static geometry is considered when placing virtual point lights. Furthermore the solution is limited to one bounce indirect illumination.
3 Imperfect Shadow Maps

3.1 Introduction

The method of Imperfect Shadow Maps deals with the computation of occlusion in a scene, as previously mentioned. Many indirect illumination methods are based on the concept of virtual point lights for the approximation of light bounces. ISMs’ contribution is the efficient parallel computation of the shadows casted by the VPLs. Because of the increased computation cost, due to by the high number of VPLs required to approximate indirect lighting in a convincing way, accuracy is sacrificed for the sake of efficiency. The choice of approximating the scene by a sparse set of points rather than the actual polygons and the ability to create all the shadow maps in a single pass are the crucial factors which make this method efficient for real-time use even for hundreds of virtual point lights.

3.2 Scene Representation

The first step of this method is performed offline and regards the creation of a point set from the scene, which will be used for the rendering of the shadow maps later. For each object in the scene, a number of points lying on its surface are selected and stored in a buffer. In order to represent the scene as accurately as possible, the points are uniformly distributed across the objects' surface. The number of points selected from each object must correspond to its actual size, so each point is selected from a surface with a probability proportional to the total object surface area. A second concern when building the scene's representation is to allow the use of dynamic objects. To avoid re-buidling the point set in each frame, information on which object each point belongs is also stored, so the point's world space location can be easily found at runtime. Optionally, the normal vector and the texture coordinates for each point could be stored in order to compute more light bounces as described later.

3.3 ISMs Creation

Apart from the process of extracting the necessary points there is no other preprocess step. At runtime, ISM assumes that indirect lighting is described by a set of hemispherical lights (VPLs). The authors use the Reflective Shadow Maps method to compute the VPLs but any other method could be used. As illustrated in section 2, there are many algorithms for determining these light sources like sampling a shadow map (RSM) or shooting rays from the primary light source.

After the secondary light sources have been determined the actual process of creating the shadow map for each VPL takes place. The input of this procedure is the set of points extracted offline and its output is a high resolution texture containing all the shadow maps. Typical dimensions for each individual shadow map are 64x64 or 128x128. Each of the stored points is processed as described below:
Its current world space position is derived using the relevant information saved during the preprocess step.

A random number ranging from one to the total number of VPLs is generated. This number indicates the shadow map where the point will be stored.

The point is transformed into parabolic space, considering the selected VPL’s position as the origin of the projection and the VPL’s normal as the vector defining the front hemisphere.

The final position of the point is computed by applying an offset according to the position of its shadow map inside the output texture. (As mentioned earlier all shadow maps are contained in the same texture.)

The depth value of the point is stored in the texture.

Steps 1-4 are typically accomplished in a vertex shader. Furthermore, parabolic mapping is chosen because of its capability to capture the whole hemisphere as required for hemispherical lights. Of course, different projections can be used.

### 3.4 Parabolic Mapping

As mentioned above, a VPL emits light over the hemisphere defined by its position and its normal so a suitable projection for the shadow map should be used. One possible approach would be to use a cubemap. However, this method requires the projection of the scene five times, one for each side of the cube (except the one facing in the opposite direction), which leads to a high computation cost. A more suitable solution is the use of parabolic mapping. A paraboloid described by the following equation can be seen at the next picture:

\[
f(x, y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2)
\]
Heidrich et al.[26] stated the fact that an orthographic camera facing a reflecting paraboloid of this type can see all the information contained in the hemisphere defined by the point \((0,0,0)\) and the vector \((0,0,1)\) looking at the camera. Thus, any ray travelling towards the origin of the paraboloid is reflected at the direction of \((0,0,1)\) as illustrated in the following figure:

In order to project a 3D point into a set of 2D coordinates we have to find the point on the paraboloid which reflects the incoming ray defined by the 3D point towards the vector \((0,0,1)\).

The first step is to transform the point into the light's space, using a view matrix with the light's position as the origin and its normal as the view vector.

We know the incident vector \(\vec{v}\) and the reflecting vector \(\vec{r} = (0,0,1)\) so we can compute the half vector: \(\vec{h} = \vec{v} + \vec{r}\).

As can be seen in the following picture the half vector and the normal vector only differ in their magnitude:
The general form of the normal vector at a point \( P(x, y, f(x, y)) \) on the paraboloid can be computed by taking the cross product of the tangent vectors. The two tangent vectors can be derived from the paraboloid's equation as follows:

\[
T_x = \frac{\partial P}{\partial x} = (1, 0, \frac{\partial f(x, y)}{\partial x}) = (1, 0, -x)
\]

\[
T_y = \frac{\partial P}{\partial y} = (0, 1, \frac{\partial f(x, y)}{\partial y}) = (0, 1, -y)
\]

Consequently, the normal \( \vec{n} \) for a point \((x, y, z)\) on the paraboloid is:

\[
T_x \times T_y = (x, y, 1)
\]

So, dividing the \( \vec{n} \) vector by its z component gives the normal vector.

### 3.5 Pull – Push

The need for low computation cost enforces the use of a sparse set of points for the shadow maps. Consequently, each individual shadow map will have holes which leads to inaccurate shadowing (Figure 3.2). In order to compensate for this insufficiency, an image-space technique capable of replacing the holes in the shadow maps with approximate depth values, is used. This technique is called Pull-Push[27] and consists of two steps. The input is the texture containing the incomplete shadow maps and the output is a reconstructed version of the input texture with no holes.

In the first step (pull) of Pull-Push, an image pyramid is built in the following way: Using the input texture, a new texture having its x and y dimensions equal to input texture's dimensions divided by two is generated. The value for each pixel of this new texture is set by interpolating the (depth) values of the four closest pixels of the input texture. Then, this process is repeated until a texture of 1x1 dimensions is created. A special case is the treatment of pixels representing holes or pixels which do not participate in any of the shadow maps. Pixels of this kind do not represent depth values so they are not used in the interpolation.
In the second step (push), the holes of each texture are filled using the following method: Each "empty" pixel of a texture is replaced by the interpolated value of the four closest pixels on the texture of the coarser level of the pyramid. For example, the holes of a texture of 32x32 dimensions are filled using values from a texture of 16x16 dimensions.

**Outlier Rejection**

Combining depth values which greatly differ could lead to wrong results when setting missing depth values. Therefore, during the pull phase a second test has to be done in order to ensure that only pixels with close depth values are combined. To that purpose, we first find the pixel holding the minimum value and then each of the rest pixels participates in the interpolation only if its difference from the closest pixel does not exceed a certain threshold. Respectively, during the push phase only values that significantly differ from the values of the input texture are overwritten. The thresholds for the above operations are typically set to 5% of the scene extent and are scaled by $2^l$ where $l$ is the level (l) of the pyramid being processed, having $l=0$ for the coarsest level.

3.6 Multiple Bounces

The process as described so far, only concerns the first bounce of indirect lighting. Although the actual method does not deal with the creation of the initial VPLs, the authors present an extension of their technique in order to create VPLs for subsequent bounces on the GPU and render them taking occlusion into account. Their method is called Imperfect Reflective Shadow Maps (IRSM) as it combines concepts from ISMs and RSMs.

In order to implement this extension, modifications on the scene's representation structure have to be done. Instead of storing only positions, normals are also stored. We also assume that texture coordinates for each point are available.

Then, to create the VPLs for the second bounce we use the point set in the same way as we did for creating the shadow maps for the first bounce, but instead of storing depth values we store positions, normals and flux. As can be observed, this procedure is identical to Reflective Shadow Maps but points are processed instead of polygons. Now, by sampling the generated texture, using importance sampling or uniform sampling, all the required data (position, normal, flux) for defining the VPLs for the second bounce are available. Of course, the point set is re-used in the traditional way, to compute the occlusion information for the second bounce. By repeating the whole process, the desired number of bounces is computed.
3.7 Limitations

As mentioned in the introductory paragraph, ISM manages to achieve real-time performance by reducing accuracy. The accuracy of the results depends on the ability of the point set to closely approximate the actual scene.

As can be observed, the necessary number of points needed for a sufficient representation is directly affected by the geometric and depth complexity of the scene. As the depth complexity increases, higher resolution shadow maps are required to keep the representation capable of producing accurate shadow maps. On the other hand, having shadow maps of relatively high resolution creates the need for more points in order to have a reasonable region of its shadow map filled.

Therefore, the suitability of this technique becomes greatly reduced when it comes to outdoor environments. For large environments where geometry can be organized in portals (e.g. a building with many rooms), the authors suggest the division of the point set into smaller chunks in order to compute results locally.

Generally, ISMs is a technique which needs a careful selection of parameters such as the dimensions of the shadow maps and the size of the point set, as the scene's characteristics greatly determine which values will give in convincing results.

3.8 View-Adaptive Imperfect Shadow Maps

The use of a constant point density for the approximation of the scene excludes any possibilities of improving the point distribution on a per-frame basis by selecting more points from objects closer to the viewer. This approach bounds the correctness of the results as well as the size and type of scenes that can be handled correctly by ISM. Ritschel et al. presented View-Adaptive Imperfect Shadow Maps [28], an extension of ISM where the point selection scheme is modified in order to support denser geometry sampling for objects being close to the viewer and more rough sampling for distant objects. This way, the indirect shadows casted on close geometry are of higher detail and satisfactory results can be obtained for larger scenes.

To be able to perform this adaptation, information about the whole scene must be available in each frame. The authors use a high resolution texture where the vertices' coordinates of all scene's triangles are placed. To define each triangle's importance at runtime, its solid angle relative to a number of view samples is computed. A view sample represents a randomly selected texel of the image seen by the camera. The next step is to store the importance computed for each triangle in a 1D texture. The \( i^{th} \) texel of the texture stores the importance of the \( i^{th} \) texture. Using this texture, a new one of identical dimensions is built. This new texture contains a cumulative distribution function (CDF) which is computed by storing in the \( i^{th} \) texel the sum of all pixels \( j \) with \( j \leq i \). Then, given a total number of \( N \) points, a triangle is selected for each one based on its importance. To fully determine the point's position on the selected triangle, a pair of random barycentric coordinates, holding the property \( x + y \leq 1 \), is sampled from a two-channel texture.
In order to accurately compute indirect shadows for dynamic scenes a large subset of the point set must be updated in each frame. These changes though, might cause flickering. To prevent this, a lazy approach regarding the update procedure is adopted. More specifically, only a small subset is updated in each frame, about the $\frac{1}{8}$th of the points. This approach introduces an additional advantage which is the reduction of the computation time because of the less number of points processed in each frame. The disadvantage of this approach, is of course the non-optimal point distribution during motion which reduces the accuracy of the solution. However, the result is still better than using traditional ISM.

Figure 3.7: Comparison between uniform and adaptive scene sampling. Only a random view sample was chosen from the framebuffer to guide the sampling.

Figure 3.8: A visual representation of the steps applied in order to compute adaptive ISMs.
4 Illumination

4.1 The Rendering Equation

In order to achieve true Global Illumination we must take into account all interactions between the light and the world objects. In nature, this is an infinite process, as the light continuously travels from one object to another, being absorbed, refracted or reflected each time it hits a surface. To capture an image of the world, what we have to do is to measure the light travelling towards our eye (or camera) from a set of directions. In the context of computer graphics each direction is defined by a ray starting from the camera and passing through a pixel of the two dimensional image. The quantity which describes the amount of light travelling through a point in space toward a specific direction is called radiance. Therefore, to generate our image we must find a way to compute the radiance leaving a point of a surface towards a given direction. An equation capable of accurately computing this quantity, known as the Rendering Equation[29 was] presented in 1986 and is shown below in one of its forms:

\[ L_o(\bar{p}, \bar{v}) = L_e(\bar{p}, \bar{v}) + \int_{\Omega} f(\omega_i, \bar{v}) \ L_o(r(\bar{p}, \omega_i), -\omega_i) \ \cos \theta_i \ \ d\omega_i \]

where:

\[ L_o(\bar{p}, \bar{v}) \] , represents the outgoing radiance from point \( \bar{p} \) towards direction \( \bar{v} \).

\[ L_e(\bar{p}, \bar{v}) \] , is the radiance emitted from point \( \bar{p} \) towards direction \( \bar{v} \) independently of any incoming light, e.g. the light emitted from a shining object.

\[ f(\omega_i, \bar{v}) \] , is a function describing how light coming from direction \( \omega_i \) is reflected towards direction \( \bar{v} \) for point \( \bar{p} \) , known as BRDF.

\[ r(\bar{p}, \omega_i) \] , returns the first point intersected by the ray with \( \bar{p} \) as origin and \( \omega_i \) as direction.

\[ L_o(r(\bar{p}, \omega_i), -\omega_i) \] , is the outgoing radiance from point returned by \( r(\bar{p}, \omega_i) \) towards direction \( \omega_i \). This operator needs recursive evaluation which significantly increases the computation cost.

**Figure 4.1:** The incoming radiance at point \( p \) from direction \( \omega_i \) equals the outgoing radiance towards \( -\omega_i \) from the first hit point.
\( \omega_i \), is the direction of incident light parameterized as \((\theta_i, \phi_i)\)

\( \Omega \), is the hemisphere above p, centered at the normal vector

If we only take diffuse reflections into account the BRDF term remains constant regardless of the incoming and outgoing directions. Therefore the equation becomes:

\[
L_o(\hat{p}, \hat{v}) = L_e(\hat{p}, \hat{v}) + \frac{\rho(\hat{p})}{\pi} \int_{\Omega} L_o(r(\hat{p}, \omega_i), -\omega_i) \cos \theta_i \ d\omega_i
\]

where \( \rho(\hat{p}) \) is the diffuse color (albedo) of the surface.

### 4.2 Instant Radiosity

The recursive factor of the rendering equation and the need to evaluate the incoming radiance over a full hemisphere make the equation expensive for real time use. Instant Radiosity approximates the incoming radiance for each surface point by a set of particles computed on the CPU. Each particle is created by shooting a ray from the light source and finding the first hit point. To enable the evaluation of more light bounces new rays can be traced from each initial particle based on a russian roulette scheme[33]. Therefore, the integral is replaced by a sum and the rendering equation is transformed as follows:

\[
L_o(\hat{p}, \hat{v}) = L_e(\hat{p}, \hat{v}) + \sum_{i=0}^{N} \frac{\rho(\hat{p})}{\pi} L_i
\]

where \( L_i \) is the outgoing radiance from the surface point i which was hit by a particle.

The derived formula is not sufficient if we do not take shadows from each of these particles into account. To that purpose, Keller in the original paper, treats each particle as a point light which is rendered using normal shadow mapping. He computes the final image by rendering the scene in an accumulation buffer, once for each particle. Today, by taking advantage of the modern hardware it is possible to render all particles in a single pass using techniques such a deferred shading which operates directly on the two-dimensional image.

### 4.3 Computing indirect illumination using Reflective Shadow Maps

The method used by the authors of ISM for the computation of indirect illumination which is also the method used in this thesis is based on Reflective Shadow Maps.

Although there are more than one ways to use RSMs for indirect lighting the underlying concept of a Reflective Shadow Map remains the same. A Reflective Shadow Map is a way to create the particles (VPLs) emitting the first bounce of indirect light and it is based on the fact that all light emitted from the primary light sources is reflected from surfaces directly visible to it. So in order to create the VPLs all the information of the surfaces visible to the light must be acquired. All this information can be easily and efficiently computed using a shadow map.
Taking the light's position as the position of the camera one or more shadow maps (depending on the light's type) are drawn and each of their pixels represents a location reflecting direct light.

A reflective shadow map must provide all the required information for a VPL, therefore rendering only a single shadow map giving us each pixel's world space position does not suffice. To capture the complete information multiple render targets must be used. For each pixel-VPL we must know its position in order to compute the light transfer coefficient between the VPL and the shaded point during the shading process, its normal vector, as VPLs are represent hemispherical lights and the flux of the indirect light, therefore at least two render targets are needed for the two latter parameters and a third one if we want to store position directly instead of recomputing it from the depth buffer of the shadow map. As we can see, unlike in classic Instant Radiosity, the whole process can be implemented on the GPU.

After the Reflective Shadow Maps have been created, we have to decide on how to use them for applying indirect lighting. As mentioned earlier, each pixel of an RSM reflects light from the primary source so it is a candidate VPL. However, using a typical shadow map of 512x512 results in tens of thousands of VPLs, which is not a manageable number even for modern hardware. Therefore, a subset of pixels has to be selected, suitable of achieving both convincing results and high frame rates.

4.3.1 Evaluation

a. Per point evaluation based on spatial proximity

The best possible set of VPLs for each surface point are those contributing the most at its illumination. The contribution of a VPL to a point depends on their distance and on the orientation of their normals. The approach presented in the original RSM paper builds a set of VPLs for each point by trying to select the pixels which are closer to it.

Figure 4.2: The three components of a Reflective Shadow Map in the middle. The position buffer, the normal buffer and the flux buffer. The first image shows the depth buffer and the last one the final result.

Taking the light's position as the position of the camera one or more shadow maps (depending on the light's type) are drawn and each of their pixels represents a location reflecting direct light.

A reflective shadow map must provide all the required information for a VPL, therefore rendering only a single shadow map giving us each pixel's world space position does not suffice. To capture the complete information multiple render targets must be used. For each pixel-VPL we must know its position in order to compute the light transfer coefficient between the VPL and the shaded point during the shading process, its normal vector, as VPLs are represent hemispherical lights and the flux of the indirect light, therefore at least two render targets are needed for the two latter parameters and a third one if we want to store position directly instead of recomputing it from the depth buffer of the shadow map. As we can see, unlike in classic Instant Radiosity, the whole process can be implemented on the GPU.

After the Reflective Shadow Maps have been created, we have to decide on how to use them for applying indirect lighting. As mentioned earlier, each pixel of an RSM reflects light from the primary source so it is a candidate VPL. However, using a typical shadow map of 512x512 results in tens of thousands of VPLs, which is not a manageable number even for modern hardware. Therefore, a subset of pixels has to be selected, suitable of achieving both convincing results and high frame rates.

4.3.1 Evaluation

a. Per point evaluation based on spatial proximity

The best possible set of VPLs for each surface point are those contributing the most at its illumination. The contribution of a VPL to a point depends on their distance and on the orientation of their normals. The approach presented in the original RSM paper builds a set of VPLs for each point by trying to select the pixels which are closer to it.

Figure 4.3: Two virtual point lights, \( x_q \) and \( x_p \) in the left image. In the right image, their position \( p \) and \( q \) in the shadow map created from the light's viewpoint.
The selection of nearby pixel-lights is based on the assumption that points which are close in world space will also be close in the shadow map. Therefore, the process is accomplished by projecting each surface point on the shadow map and then selecting pixels according to their distance from the projected point on the shadow map. This assumption may lead to the selection of lights not actually close to the point and besides this, the light's normal orientation is not taken into account, therefore it is possible to include unimportant lights in the selection. However, it is most likely that all important lights for a point will lie close to it in the shadow map and will be selected. To quickly select nearby lights a pre-computed sampling pattern is used (Figure 4.5).

**Figure 4.4:** These images illustrate the idea that points close in world space are likely to be close in the shadow map. However only \( x_2 \) contributes to the illumination of point \( x \), as the other three lights do not have a suitable orientation.

**Screen space interpolation**

Although only a subset of VPLs from the RSM is used for each point, the computation cost for all pixels is still too expensive for real time use. One way to reduce the cost is by exploiting the nature of indirect lighting. As indirect illumination consists of low frequency gradations, the evaluation of a VPL group for an image of lower resolution would suffice for the accurate representation of indirect lighting. For the final image, interpolating values from the coarser image could effectively provide the a convincing result.

So the first step is to render an image from the camera view, but of lower resolution than the final and then evaluate the RSM as described above for all pixels. When rendering the final image, what we have to do is sample the four surrounding pixels of the coarser image and use their interpolated value. It is understood that the interpolated value will successfully represent the actual value of the final image only if the pixel of the final image and the relevant pixels of the coarse one present a satisfying geometric similarity, i.e. they have a similar position and normal. To ensure this, the position and normal of the target pixel are compared against the positions and normals of the corresponding samples. A similarity between the target pixel and 3 or 4 of the samples ensures the suitability of the samples therefore the interpolated value is used. Otherwise, the target pixel is marked as not computed. In a subsequent pass the indirect illumination for all marked pixels of the final image is computed directly.

**b. Evaluation using importance sampling**

A different approach includes the selection of a subset of lights for the shading of all points, instead of choosing a different subset for each point. As each indirect light mainly contributes to its nearby surfaces, using the method of deferred shading yields satisfactory results in terms of quality and speed.
To use deferred shading, properties such as the world space position, normal and material have to be stored for each rasterized point. In a subsequent pass a geometric shape covering the light's affecting area is drawn. For each pixel, the information stored earlier is read and combined with the light's properties available from the RSM, in order to compute the indirect lighting for the point. The cost of the deferred pass is partially based on the region covered by each geometric shape therefore using a shape tightly enclosing the region affected by the light reduces the load put on the fragment shader which is usually the bottleneck.

Since we are using a subset of lights for all points, care should be taken so that the chosen samples best describe the indirect lighting distribution, i.e. having more samples in areas of higher flux. To that purpose, importance sampling[31] by hierarchical warping can be used. By hierarchical warping a number of uniformly distributed samples \( s_i = (x_i, y_i) \), \( x_i, y_i \in [0,1) \) are recursively transformed to follow the function's distribution. The warping is performed independently for each dimension so in the case of a two-dimensional texture the process is executed twice for each level of warping. Given the initial point set, the first step is to compute the total flux of each half along one of its dimensions e.g. for the vertical one we compute \( \Phi_{\text{top}} \) and \( \Phi_{\text{bottom}} \). Then, to adjust the distribution according to the percentage of the flux found in each half, each point's respective coordinate is scaled as seen below:

\[
y'_i = \begin{cases} 
  y_i/(2\Phi_r) & \text{if } y_i < \Phi_r \\
  (1+y_i-2\Phi_r)/(2(1-\Phi_r)) & \text{otherwise}
\end{cases}
\]

where:

\[
\Phi_r = \frac{\Phi_{\text{top}}}{\Phi_{\text{top}} + \Phi_{\text{bottom}}}
\]

**Figure 4.5:** The sampling pattern used for sampling indirect light sources for each point. As nearby light are considered more important the sampling density decreases with the distance to the center. The radius of its sample represents its weight which increases as the distance to the center grows.

**Figure 4.6:** The result of screen space interpolation: The red areas represent pixels were interpolation was not possible due to normal or position discontinuities. For the largest part of the scene however interpolation from a low-res image was possible.
The weight of each sample is also readjusted so that the sum of all weights remains the same:

\[ w'_i = \begin{cases} 
  \frac{w_i}{2\Phi_r} & \text{if } y_i < \Phi_r \\
  \frac{w_i}{2(1-\Phi_r)} & \text{otherwise}
\end{cases} \]

The process is repeated for the horizontal dimension, separately for each row.
5 Application Architecture

5.1 Overview

Pre-computations
At initialization time the point scene representation used for the rendering of the ISM is created. Each triangle is passed through the geometry shader and a number of points lying on its surface are stored in a vertex buffer. Each triangle can be tessellated up to 3 times in order to improve the accuracy of the representation.

Runtime
The steps executed in each frame are the following:

1. First, the G-Buffers are created. This involves rendering the position, normal and diffuse color for every point seen by the camera. The G-Buffers are used later for the shading of direct and indirect lighting.

2. Creation of the RSMs. The RSMs are not only used for sampling the VPLs but also for computing the shadow map of the primary light sources. Therefore, they are computed before the shading of direct lighting.

3. Shading of direct lighting. In this step the information from the G-Buffers and the RSM are used to compute the effect of direct lighting and save it to the backbuffer. For every pixel, its position, normal and color properties are retrieved and used in conjunction with the shadow map to shade the point. For every light, a mesh representing the area influenced by the light is rendered leading to the evaluation of the shading equation for each pixel covered. The type of mesh depends on the light's type and characteristics.

4. Steps 4 and 5 involve the computation of indirect lighting. At step 4 the Imperfect Shadow Maps are created. Information about the VPLs have already been computed at step 2 so each point of the scene representation is assigned to an indirect source in order to create the shadow maps of the VPLs.

5. Shading of indirect lighting. The VPLs as well as their shadow maps have been determined so the only step left is the computation of indirect illumination. To that purpose, a mesh of appropriate size and shape is rendered, as in step 3, and for each point the mesh covers the indirect lighting in computed and added upon the existing color value.
5.2 Scene Tessellation

As mentioned above the initialization step includes the subdivision of each triangle up to three times in order to increase the representation's accuracy. This procedure is implemented at the geometry shader by a loop simulating the recursive subdivision function listed below. For each final triangle four points are stored in a vertex buffer: its three vertices and its centre. The data stored for its point are its local space coordinates and the id of its object. Due to restrictions on the total number of float elements a geometry shader can output, the subdivision is limited to three levels.

```c
void tesselated(triangle tri, uint level)
{
    if (level == 0)
        output(tri.v0, tri.v1, tri.v2, tri.centre)
    else
        c0 = tri.edge(0).centre
        c1 = tri.edge(1).centre
        c2 = tri.edge(2).centre
        tessellate(tri.v0, c0, c2, level-1);
        tessellate(tri.v1, c1, c0, level-1);
        tessellate(tri.v2, c2, c1, level-1);
        tessellate(c0, c1, c2, level-1);
}
```

5.3 Deferred shading

Deferred shading[31] is technique for applying the illumination from a number of lights in the scene, where the computation of lighting is performed using information previously stored in textures, instead of directly processing the scene's geometry. The basic idea of deferred shading is to decouple the required amount of time for computing the illumination from the geometric complexity of the scene. In classic (or forward) shading the illumination is computed independently for each object as shown below:

For each object:
    Render mesh, applying all lights in one shader
This leads to $M \times L$ complexity where $M$ is the number of objects and $L$ is the number of lights. By decoupling lighting from geometry only a single pass is required in order to store the relevant data, which is a relatively cheap process. After that, illumination from each light is directly applied to its area of influence, leading to $M + L$ complexity:

- **For each object:**
  - Render to multiple targets
- **For each light:**
  - Apply light as a 2D postprocess

To save all the required information in one pass, multiple render targets have to be used. The scene is drawn from the camera's viewpoint and for each visible pixel, position, normal and material attributes are stored, each one in a separate buffer. Storing the position attribute can be omitted as it can be recomputed from the depth buffer in the pixel shader. These buffers are usually called G(ometry)-buffers and as long as they have been filled the objects are no longer required.

**Figure 5.2:** To create the G-Buffers the geometry is submitted to the GPU. For each rasterized point multiple data are stored in the same pixel position, but in different render targets.

Finally, the lighting is applied as a two-dimensional post-process. For each light, a polygon is drawn with the G-buffers passed as input textures. The polygon can be either a simple quadrilateral enclosing the area to be illuminated or a more precise representation of the influenced area, in order to avoid needless pixel shader executions. For each pixel covered by the polygon, the corresponding attributes are retrieved from the G-buffers and they are used to compute the shaded color of the surface by evaluating a shading function.

As mentioned above, the main advantage of deferred shading is that the procedure of shading is performed only once for each pixel so there is a known upper bound at the rendering cost. This results in reduced computation cost for a wider group of processes related to shading e.g. the normal transformation when a normal mapping technique is used. Additionally, the direct determination of the influenced pixels for each light by the use of an appropriate mesh, eliminates the need to detect which lights affect each object which is a mandatory task in forward shading in order to reduce computations.
Besides its advantages deferred shading also comes with some limitations. Firstly, the G-buffers significantly increase the amount of memory required. Another disadvantage of deferred shading concerns the rendering of transparent objects. Since only one object per pixel can be stored in the G-Buffer transparent objects cannot be handled and are usually rendered in the end using a forward renderer approach.

In this application three G-Buffers are created. The first one is used to store the position of each visible point, the second is used for the normal vectors and the last one stores the corresponding diffuse color.

Normal Buffer:

![Normal Buffer Image](image)

**Figure 5.3:** The render target containing the normal data for each visible point. Each point's normal is scaled to [0,1] and saved using 16 bits for each of the three coordinates.

Diffuse Buffer:

![Diffuse Buffer Image](image)

**Figure 5.4:** The diffuse color buffer created using 8 bits for each color channel.
Position Buffer:

Figure 5.5: The position buffer which is the most memory-consuming as 32 bits are dedicated for each coordinate.

To perform the shading, meshes tightly enclosing the area to be illuminated are rendered. The application supports three types of lights: directional, spotlight, and omnidirectional. A directional light affects the entire scene so a full screen quadrilateral is used. A spotlight is described geometrically by its position, its direction vector, which indicates the direction of the emitted light, an angle $\Phi_{\text{max}}$ defining the maximum angle the light reaches, and a scalar value defining its maximum range. Therefore, its area of influence is best represented by a cone. To account for different angles or maximum range values a source may have, a unit cone is initially loaded and transformed analogously to the source's characteristics. An omnidirectional point light source has a position and a radius of influence. Therefore a unit sphere is loaded and scaled according to the source's radius so it can enclose its area of influence.

Figure 5.6: A cone mesh covering the area illuminated by a spot light. The final dimensions of the cone has to be determined at runtime according to the light's cut-off angle, range etc.

Figure 5.7: A sphere representing an omnidirectional light located at the center of the room.
The Reflective Shadow Map for each light consists of three render targets, one storing the position for each visible point, one storing the normal of the surface and one storing the reflected flux. Furthermore the depth buffer used when creating a RSM is also used as the shadow map of the primary light source when computing the direct lighting. All render targets are of 512x512 dimensions. In the case of an omnidirectional light the whole set of directions originating from the light's position must be evaluated, so a cubemap is used instead of a single plane.

After the rendering of the RSM buffers the importance sampling step takes place. The reflected flux texture is bound as input and the output is a texture holding the warped positions of the samples. An initial uniform distribution is assumed for the samples and the warping process is executed independently for each pixel of the render target. Only the first level of warping is applied, as shown below:

```c
float2  points[100];
float    weights[100];

float top_y = 0.0f;
float bottom_y = 0.0f;

// Consider 100 pixel lights lying in a 10x10 grid
int num_rt = 100;

// Compute the warping for the vertical (y) dimension
for (int i=0; i<num_rt; i++)
{
    for (int j=0; j<num_rt; j++)
    {
        float x = (i+0.5f) / (float)num_rt;
        float y = (j+0.5f) / (float)num_rt;

        points[i*num_rt+j] = float2(x,y);
        weights[i*num_rt+j] = 1.0f;

        float3 val = rsm_color.Sample(tex_sampler, float2(x,y)).xyz;
        if (j <= 5)
            top_y += (val.x + val.y + val.z) / (float)3.0;
```

Figure 5.8: The VPLs, indicated by yellow color.

Figure 5.9: The light has the same position and orientation as in the left image, but the right wall of the room is now black instead of red. Consequently, no VPLs are placed there as importance sampling is used.
else  
    bottom_y += (val.x + val.y + val.z) / (float)3.0;
}
}

float f_r = top_y / (float)(top_y + bottom_y);
float light_num = 100;

// Adjust positions and weights
for (int i=0; i<light_num; i++)
{
    if (points[i].y < f_r)
    {
        points[i].y = points[i].y / (float)(2.0f * f_r);
        weights[i] = weights[i] / (float)(2.0f * f_r);
    }
    else
    {
        points[i].y = (1.0f + points[i].y - 2.0f * f_r) / (float)(2.0f * (1.0f - f_r));
        weights[i] = weights[i] / (float)(2.0f * (1.0f - f_r));
    }
}

float left_x = 0.0f;
float right_x = 0.0f;

// Continue with the horizontal dimension
for (int i=0; i<num_rt; i++)
{
    for (int j=0; j<num_rt; j++)
    {
        float3 val = rsm_color.Sample(tex_sampler, points[i*num_rt+j]).xyz;
        if (i <= 5)
            left_x += (val.x + val.y + val.z) / (float)3.0f;
        else
            right_x += (val.x + val.y + val.z) / (float)3.0f;
    }
}

f_r = left_x / (float)(left_x + right_x);
for (int i=0; i<light_num; i++)
{
    if (points[i].x < f_r)
    {
        points[i].x = points[i].x / (float)(2.0f * f_r);
        weights[i] = weights[i] / (float)(2.0f * f_r);
    }
    else
    {
        points[i].x = (1.0f + points[i].x - 2.0f * f_r) / (float)(2.0f * (1.0f - f_r));
        weights[i] = weights[i] / (float)(2.0f * (1.0f - f_r));
    }
}

int index = (int)(pin.pos.x - 0.5f) * (float)num_rt + (pin.pos.y - 0.5f);
pout.res = float4(points[index].x, points[index].y, weights[index], 1.0f);
return pout;

5.5 ISM

As mentioned above the first step of the ISM procedure involves the creation of the point set. The previously
described geometry shader process export points into a vertex buffer which is used as input at runtime.

To create the ISMs at runtime we bound the vertex buffer as input and a large texture of 2048x2048
resolution which will store the ISMs as the render target. The purpose of the vertex shader is to compute the
relative position of each point in the output texture and also its respective depth. The first step is to randomly
assign each point to a virtual point light. To that purpose, a texture containing random values is passed as input. This texture is randomly sampled and the value is scaled to the number of virtual point light in order to determine the VPL’s id. The position and normal render targets of the RSM are also passed as input and after the VPL id has been determined they are sampled in order to transform the point to parabolic space:

```cpp
output.position = mul(WS_position, light_view_matrix);
output.position = output.position / OUT.position.w;
float L = length(output.position.xyz);
output.position = output.position / L;
output.position.z = output.position.z + 1;
output.position.x = output.position.x / OUT.position.z;
output.position.y = output.position.y / OUT.position.z;
output.position.z = (L - near_plane)/(far_plane-near_plane);
output.position.w = 1;
```

To perform the pull-push pass 12 textures are created. The texture chain of the pull phase is created by successively switching the input and output textures, i.e. starting with the N-size texture as the input and the \( \frac{N}{2} \)-size texture as the render target, then the previous render target as the input and the \( \frac{N}{4} \)-size texture as the output and so on until the 1x1 texture is bound as render target. This procedure is performed inside a loop.

Pseudocode for the pixel shader executed is shown below:

```cpp
// pixel coordinates in [0,1]
float px = pixel_coords.x / (input_texture_dimension / 2.0f);
float py = pixel_coords.y / (input_texture_dimension / 2.0f);
float cnt_offset = 0.5f / (float)(i_input_dim);
// Sample four nearest samples of input texture
float tl = tex_input.Sample(px-cnt_offset, py-cnt_offset);
float tr = tex_input.Sample(px+cnt_offset, py-cnt_offset);
float bl = tex_input.Sample(px-cnt_offset, py+cnt_offset);
float br = tex_input.Sample(px+cnt_offset, py+cnt_offset);

if (tl + tr + bl + br >= 4.0f)
{
    return 1.0f;
}

float scale = pow(2, i_level);
float threshold = threshold_val / scale;

float minimum = min(tl, tr);
minimum = min(minimum, bl);
minimum = min(minimum, br);
float sum = 0.0f;
uint valid = 0;
if (tl < 1.0f && abs(tl - minimum) < threshold) { sum += tl; valid++;
if (tr < 1.0f && abs(tr - minimum) < threshold) { sum += tr; valid++;
if (bl < 1.0f && abs(bl - minimum) < threshold) { sum += bl; valid++;
if (br < 1.0f && abs(br - minimum) < threshold) { sum += br; valid++;

float d = sum / (float)valid;
return d;
```
The push phase is performed in the same fashion but the textures are used in the reverse order. Pseudocode for pixel shader executed follows:

```c++
// finer level pixel center in [0,1]
float f_px = pin.pos.x / (float)(i_input_dim_push * 2.0f);
float f_py = pin.pos.y / (float)(i_input_dim_push * 2.0f);

float v = tex_flags.Sample(tex_sampler, float2(f_px, f_py)).r;

// coarse level pixel size
float c_psize = 1 / (float)i_input_dim_push;

// coarse level pixel pos
int c_cx = f_px / c_psize;
int c_cy = f_py / c_psize;

// coarse level pixel center
float c_px = c_cx + 0.5f;
float c_py = c_cy + 0.5f;

// coarse level pixel center in [0,1]
c_px /= (float)i_input_dim_push;
c_py /= (float)i_input_dim_push;

float diff = tex_input_push.Sample(tex_sampler, float2(c_px, c_py));

// coordinates of surrounding pixels
float2 f_cnt_0;
float2 f_cnt_1;
float2 f_cnt_2;
float2 f_cnt_3;
float w_0 = 9/(float)16;
float w_1 = 3/(float)16;
float w_2 = 1/(float)16;
float w_3 = 3/(float)16;

f_cnt_0 = float2(c_px, c_py);

// Sample (coarser) input texture
float v0 = tex_input_push.Sample(f_cnt_0).r;
float v1 = tex_input_push.Sample(f_cnt_1).r;
float v2 = tex_input_push.Sample(f_cnt_2).r;
float v3 = tex_input_push.Sample(f_cnt_3).r;

float sum = 0.0f;
float w_sum = 0.0f;
uint valid = 0;
if (v0 < 1.0f) { sum += v0; w_sum += w_0; valid++; }
if (v1 < 1.0f) { sum += v1; w_sum += w_1; valid++; }
if (v2 < 1.0f) { sum += v2; w_sum += w_2; valid++; }
if (v3 < 1.0f) { sum += v3; w_sum += w_3; valid++; }

float scale = pow(2, i_level);
float threshold = threshold_value / scale;
if (valid == 0)
    return 1.0f;
```

// normalize weights
w_0 /= w_sum;
w_1 /= w_sum;
w_2 /= w_sum;
w_3 /= w_sum;

float d = w_0 * v0 + w_1 * v1 + w_2 * v2 + w_3 * v3;
d /= w_sum;

diff = abs(d - diff);

// if there exists a value and it's not far from the
// coarse depth values do not replace
if (v < 1.0f && diff < threshold)
   return v;

return d;

Another option for the pull-push phase would be to create only two high resolution textures, one to contain all textures of even dimensions and one for all textures of odd dimensions.[32].

5.6 Development environment

The application was developed using the C++ programming language and the DirectX 10 graphics API and tested on a desktop computer using a double core CPU running at 2.13Ghz and an ATI Radeon HD 4850 GPU.

5.7 Application Steps' Representation

The process of creating the point representation of the scene. The geometry is passed through the vertex shader and through the geometry shader where it is tessellated. Finally a number of points is written to a vertex buffer using stream-out.

Creation of the RSM buffers. The geometry is passed through the vertex and pixel shaders and three render targets plus the depth buffer are written.
The first step of deferred shading is performed by submitting the geometry and filling the G-Buffers as previously described.

All the shadow maps for the indirect light sources are stored in the same texture. The previously computed RSM buffers (specifically the position and normal data) are needed as input in order to provide information during the parabolic transformation of each point.

The final pass of the application concerns the computation of indirect illumination which is added to the existing direct illumination stored in the frame buffer. The ISM and the RSM textures are needed to compute illumination and occlusion for each VPL.

The direct lighting is computed using the G-Buffers and the depth buffer computed at the RSM step as input. The result is stored directly to the frame buffer.
6 Results

The application was tested using two scenes, the Room scene and the Sponza Atrium scene (Figure 6.3). The first scene consists of around 47000 triangles (was tessellated in an editing program) and the second one is made up from around 262000 triangles.

6.1 Room

The Room scene was tested under various parameter combinations. Specifically it was tested using one, two and three levels of tessellation and 16, 32 and 64 pixels of resolution per imperfect shadow map. Furthermore, 256 and 400 VPLs were used.

Using one level of tessellation results in ~760000 points stored for the ISM procedure which means around 3,000 points per VPL, if 256 VPLs are used. Respectively, two levels of tessellation give ~3000000 points or around 11700 points per VPLs and finally three levels of tessellation give 46000 points per VPL. The corresponding numbers for 400 VPLs are 3000, 7500 and 30000 points respectively.
The Room scene with both direct and indirect illumination but without occlusion for the VPLs is shown in Figure 6.1. In Figure 6.2 ISMs are also used. The scene was rendered using 256 VPLs, 16x16 ISMs and two levels of tessellation at 14 fps.

The following pictures show the Room scene under various different parameters. For each row the tessellation level is increased for one to two and finally three and for each column the ISM resolution changes from 16 to 32 and then to 64 pixels:
While the ISMs' resolution mostly affects the quality of the result, the speed is mainly determined by the quantity of points used for the creation of the ISMs. For one level of tessellation, the application runs at about 19-20 fps. For two levels, around 14-15 fps are achieved and for three levels of tessellation the frame rate drops at 7-8 fps.

Although increasing the ISMs' resolution from 16x16 to 32x32 or 64x64 allows to capture higher depth complexity, it also comes with the drawback of reducing the region of a shadow map occupied by points, as more points are required to maintain the same ratio. In this example, where the scene has a low depth complexity, increasing the dimension of each ISM texture led to worse results as can be seen in the following pictures, especially at the highlighted areas:

Figure 6.4: 16x16 ISM

Figure 6.5: 64x64 ISM

Figure 6.6: 16x16 ISM

Figure 6.7: 64x64 ISM
Regarding the impact of the level of tessellation, it was higher for 64x64 ISMs' where there was a relatively higher need for points, but not so intense for ISMs' of 16x16 or 32x32 resolution:

As can been observed, there are areas where light leaking occurs. To account for this, different thresholds were tested for the ISM sampling phase. Applying a relatively small value as threshold eliminates the light leaking as in Figure 6.7, but gives a coarser result in overall. For low enough values the opposite effect happens, i.e. pixels near the shadow area are incorrectly shadowed. As the bias increases, the shadows become of finer detail but leaking is also more probable to occur. The application of a PCF filter further improves the result as shown in the next two figures:

The scene was also rendered for the same combinations of ISM resolution and level of tessellation but with 400 VPLs. The visual result was similar to the previous images and a reduction of 3-4 fps at the frame rate also occurred.
6.2 Sponza Atrium

The Sponza Atrium scene was rendered using about 1.000,000 points, i.e. around 4000 points per VPL, 32x32 resolution for the ISMs and 256 virtual point lights. No further tessellation was used.

The following three pictures show direct, indirect and combined illumination using the above parameters and a spotlight located at the top of the scene, running at 8 fps.

![Figure 6.12: (a) only direct illumination, (b) only indirect illumination, (c) the combined result of (a) and (b)](image)

The next pictures show a spot light placed at the center of the hallway and oriented towards the right wall. The right picture also shows the 256 VPLs used for the indirect illumination:

![Figure 6.13: A spotlight at the center illuminates the right wall.](image)

![Figure 6.14: The VPLs created from the RSM procedure.](image)

The majority of the VPLs emit light towards the center of the scene, but the existence of the colored curtains should prevent most of it from reaching the center of the scene. The other part of the scene is rendered at the next two pictures. Rendering the scene without using ISMs is shown at the left image where most of the surfaces incorrectly receive indirect lighting. At the right image, occlusion for the VPLs is enabled and only a small portion of light reaches this part of the scene compared to the left picture.
The last two pictures show direct and combined illumination caused by an omnidirectional point light located at the center of the scene, running at 5 fps. The use of a point light is more expensive than the use of a spotlight because of the six shadow maps required for the computation of its occlusion information. This also affects the indirect lighting computation as six RSM buffers are required.

**Figure 6.15:** The hallway of the previous image is at the right. The left part of the scene should not receive any lighting as the hallway is separated from the rest of the scene by the curtains.

**Figure 6.16:** An ISM is computed for each VPL. Therefore, only a small portion of light enters the main hall.

**Figure 6.17:** Only surfaces directly visible from the light source receive lighting.

**Figure 6.18:** Parts of the scene not directly visible from the light source are now lit, as the first bounce of indirect lighting is also computed.
References


