Wrocław University of Technology
Faculty of Computer Science and Management

Dissertation

Image-space radiosity lighting method for dynamic and complex virtual environments

mgr inż. Paweł Rohleder

Supervisor: dr hab. inż. Ireneusz Jóźwiak, prof. PWr

Wrocław 2010

Abstract

The main goal of current real-time graphics applications (like computer games, or flight simulators) is to imitate natural phenomena to provide realistic and natural-looking virtual environments. Many lighting techniques such as soft shadows, high-dynamic range effects, surreal lighting or post-processing effects have been developed over the last few years to enhance the quality of rendered image. One of the method which greatly improves the level of realism in computer graphics applications is global illumination.

This dissertation demonstrates very neat and efficient method for calculating global illumination for dynamic and complex scenes in real-time. The proposed solution utilizes image space to compute inter-reflections between different quanta of virtual scene in very fast and robust way. New algorithm, called *ISR* (Image-Space Radiosity) has been introduced which takes advantage of graphics hardware acceleration to provide realistic lighting computations in real-time at very reasonable frame rates. *ISR* is independent from scene complexity and very easily pluggable into any existing rendering pipeline. Compared to previous lighting techniques, *ISR* is faster, more accurate and very well scalable.

Apart from the implementation details of the new algorithm along with the theoretical background and explanation, this work contains quality and performance thorough analysis and evaluation for different cases, like closed and open-spaced virtual environments.

Streszczenie

Głównym celem dzisiejszych graficznych aplikacji komputerowych czasu rzeczywistego (takich jak gry komputerowe, czy symulatory lotu) jest dokładne naśladowanie rzeczywistych zjawisk w celu uzyskania fotorealistycznych wizualizacji scen wirtualnych. Wiele technik obliczeń oświetlenia, takich jak miękkie cienie, surrealne oświetlenie, czy efekty montażowe mających na celu poprawienie jakości generowanego obrazu zostało zaprezentowanych w ostatnich latach. Jedną z metod, która istotnie zwiększa poziom realizmu w aplikacjach graficznych czasu rzeczywistego jest oświetlenie globalne.

Niniejsza rozprawa przedstawia nowy i efektywny sposób liczenia oświetlenia globalnego dla dynamicznych i złożonych scen wirtualnych w czasie rzeczywistym. Proponowane rozwiązanie wykorzystuje przestrzeń obrazu do obliczeń oddziaływań oświetlenia pomiędzy różnymi fragmentami sceny wirtualnej. Pokazano nowy algorytm, nazwany ISR (ang. Image-Space Radiosity), który wykorzystując możliwości nowoczesnych kart graficznych służy do obliczania realistycznego oświetlenia przy bardzo dobrej wydajności. W porównaniu do istniejących technik, ISR jest szybszy, bardziej dokładny i odpowiedni dla różnego rodzaju danych.

Oprócz szczegółowej teoretycznej analizy algorytmu wraz z dokładnym opisem implementacji, niniejsza rozprawa zawiera dokładną analizę i ocenę zaprezentowanego algorytmu dla różnych przypadków, tj. zamknięte i otwarte przestrzenie wirtualne.

To my wife, my family and friends.

Acknowledgments

This dissertation would never have been emerged without the support of many people. First of all, I would like to thank my supervisor, prof. Ireneusz Jóźwiak, for his support and encouraging me through my work. His kindness and optimism made me believe that there are no impossible goals to achieve. I also thank prof. Adam Grzech and prof. Zbigniew Huzar from Wrocław University of Technology for their invaluable comments and remarks during the formation of this thesis.

I would also like to thank Jakub Klarowicz, Maciej Jamrozik and other co-workers from Techland for their knowledge and sharing the ideas. Special thanks go to Nicolas Thibieroz from ATI/AMD for very constructive talks, ideas and support on cutting-edge computer graphics technology.

Finally and the most importantly, I would like to thank my parents Regina and Krzysztof, my brother Maciej, and in particular my wonderful wife Katarzyna for their support during this project. Without their patience and understanding this thesis would never have been possible.

Paweł Rohleder

Wrocław 2010

Table of contents

List of figures

List of tables

List of listings

1. Introduction

This chapter outlines the content of this dissertation and introduces the real-time global illumination problem.

1.1. Overview

Modern graphics hardware give the possibility to simulate open space environments in a very realistic way. Plenty of different approaches of imitating natural environmental effects like: sunlight, water or clouds have been developed during the last decades. One of the most essential phenomena in real-time computer graphics is lighting effect.

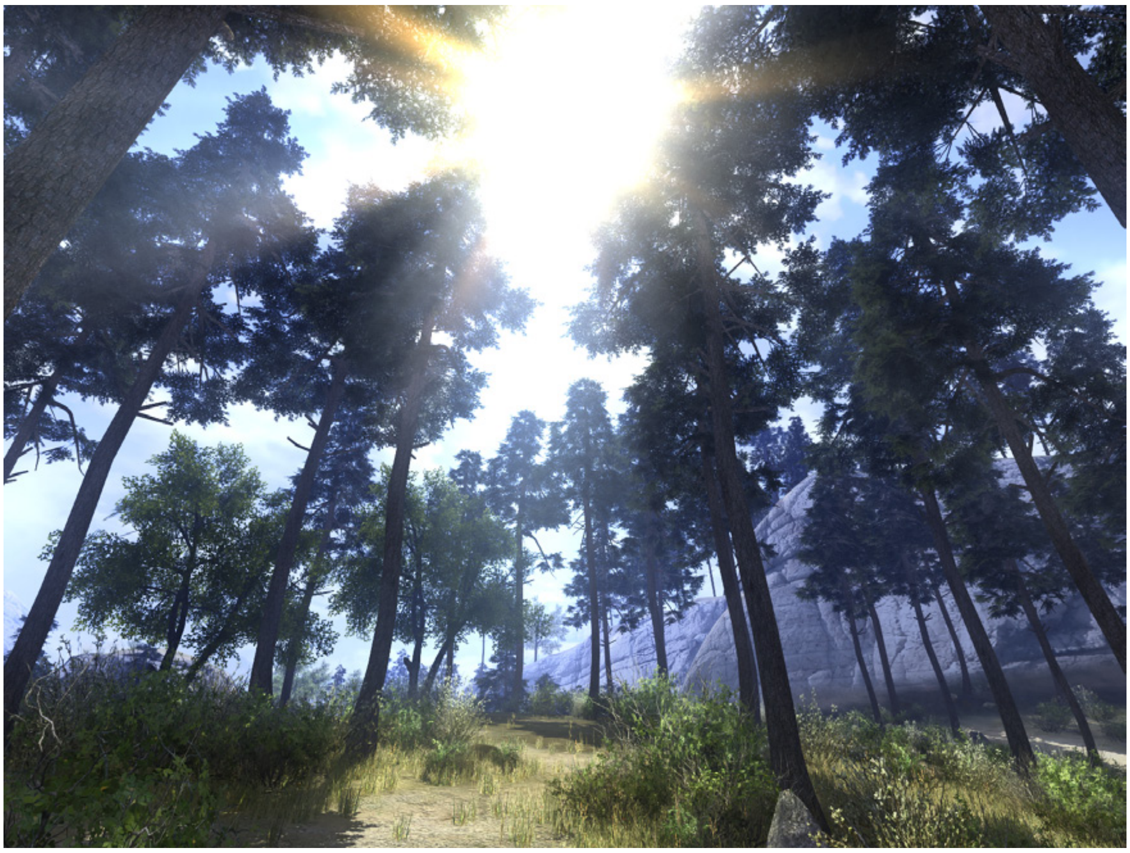


Figure 1.1: Sunlight with volumetric light rays in "Call of Juarez" game

Many lighting techniques such as soft shadows, high-dynamic range effects, surreal lighting or post-processing effects have been developed to enhance the quality of rendered image. One of the method which greatly improves the level of realism in computer graphics applications is a global illumination algorithm called radiosity.

This dissertation demonstrates very neat and efficient method for calculating global illumination for dynamic and complex scenes in real-time. The proposed solution utilizes image space to compute inter-reflections between different quanta of virtual scene in very fast and robust way. New algorithm, called *ISR* (Image-Space Radiosity) has been introduced which takes advantage of graphics hardware acceleration to provide realistic lighting computations in real-time at very reasonable frame rates. *ISR* is independent from scene complexity and very easily pluggable into any existing rendering pipeline. Compared to previous lighting techniques, *ISR* is faster, more accurate and very well scalable.

The presented technique is based on deferred shading algorithm which utilizes multiple render targets technique to generate specified per-pixel information through vertex and pixel shader GPU programs. Obtained data is being used to compute image-space ambient occlusion and simple approximation of global illumination.

Apart from the implementation details of the new algorithm along with the theoretical background and explanation, this work contains quality and performance thorough analysis and evaluation for different cases, like closed and open-spaced virtual environments.

1.2. Radiosity overview

Two major techniques have been extensively studied and developed for modeling the illumination process, ray tracing and radiosity. Ray tracing examines light rays traveling from the camera to the light sources which is computationally expensive. The behavior of light rays going through the virtual world, which can be absorbed, reflected or refracted by the objects on the scene, is modeled by the simple law of optics. More recent and innovative way for lighting calculations is the radiosity method.

Radiosity is a global illumination algorithm used in 3D computer graphics rendering to determine the illumination factor on a scene. The radiosity term is derived from the theory of thermal radiation which relies on computing the amount of light energy transferred among surfaces by tracking the energy flow of light interacting with different materials. Unlike standard shading models which compute only direct illumination (surfaces influenced straightforwardly by the light source), the radiosity method is also taking into account light reflected from the nearby surfaces on the scene (which is called the indirect illumination).

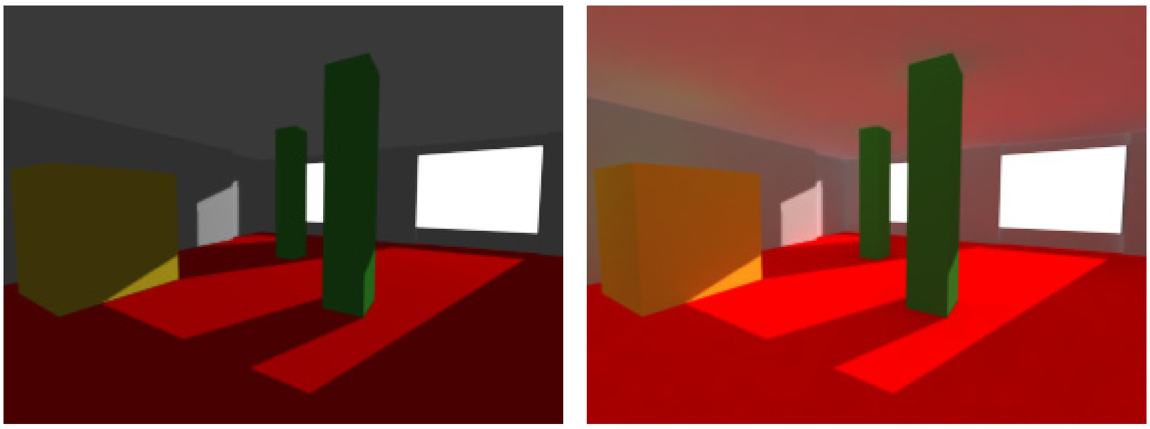


Figure 1.2: Comparison between direct lighting and radiosity (indirect lighting) methods

The images generated using radiosity technique (or generally, using indirect lighting) appear more photo-realistic, however, are computationally more expensive comparing to standard direct lighting methods and much more difficult to obtain in real-time time. In this dissertation we emphasize on the approximation of radiosity technique for real-time computer graphics software. We prove that computationally expensive illumination model can be efficiently simplified and applicable in real-time to provide higher level of realism in computer generated images.

Figure 1.3: Alchemists Laboratory by Jaime Vives Piqueres, rendered with POV-Ray, 2001

1.3. Outline

This dissertation is composed of six chapters. At the beginning we introduce the real-time radiosity problem very briefly. Chapter 2 contains some background information about radiosity as well as related work in the area of global illumination in computer graphics. Subsequently, in chapter 3 we covered detailed description of lighting models in real-time computer graphics applications. We also introduced the problem of real-time radiosity. Chapter 4 elaborates the novel algorithm, called Image-Space Radiosity (ISR) which is followed by the experiments and evaluation placed in chapter 5. Finally, the dissertation concludes with a short summary and the list of possible future work ideas in chapter 6.

2. Background and related work

This chapter introduces the basic terms and previous work related to lighting techniques in real-time computer graphics. It explains the differences between direct and indirect lighting and shows the advantages of using radiosity method.

2.1. Introduction to lighting

Apart from the accurate graphical representation of objects on virtual scene, the lighting effects are the most important in the manner of photorealism. Computer graphics lighting refers to the simulation of light in computer graphics based applications. The term illumination determines the amount of light reaching a surface and is being used in shading process which calculates the color and intensity of light reflected toward the viewer for each pixel representing the surface. The color value depends on the various properties of light source (like position or color) as well as the reflective characteristics of the surface itself.

Basically, lighting effects are described with models that consider the interaction of electromagnetic waves with object surfaces involving the principles of physical laws that describe surface light intensities. Many basic models use empirical simplification of photometric calculations to minimize required computation time. More accurate models, such as the radiosity algorithms which represent the indirect illumination models, determine light intensity by taking into account the propagation of radiant energy between the object surfaces on the scene. Although indirect lighting algorithms provide more photo-realistic representation of virtual scenes, they are often to computationally complex and can not be utilized in real-time computer graphics applications.

In the following section we take a look into basic illumination models which represent direct lighting; afterwards, we discuss more accurate and simultaneously more time-consuming indirect lighting algorithms. Subsequently, we introduce the ambient occlusion term and radiosity approach to real-time computer graphics rendering. The chapter ends with the description of related work in closing with conclusion and problem formulation.

2.2. Indirect lighting

2.2.1. Ray-tracing

Ray tracing [29][75][4] represents the technique that is capable of produce digital images at very high level of photorealism. It utilizes tracing the path of light through pixels in an image plane to evaluate corresponding pixel color value. The algorithm generates better quality images comparing to standard rasterisation rendering methods, but at a greater computational cost which makes the ray-tracing suitable for non real-time computer graphics utilization, like movie industry.

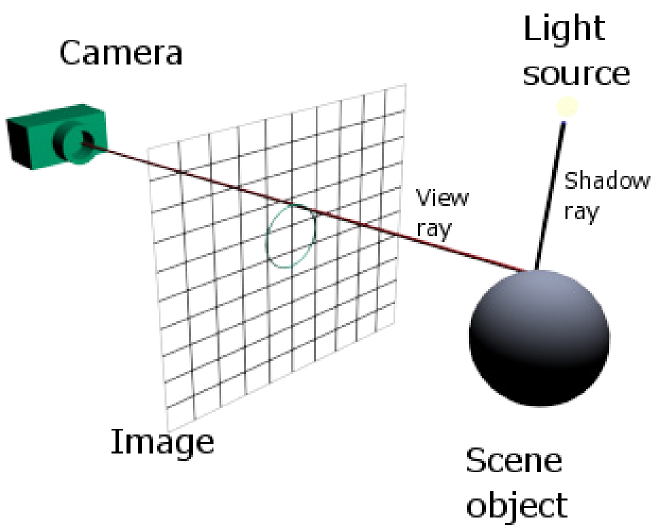


Figure 2.1: Ray-tracing algorithm overview

The idea behind ray tracing is that physically correct images are composed of light rays fired up from the light source and bounced in a scene before hitting the virtual camera. This intuitive approach provides a simple and powerful rendering technique for obtaining global transmission and reflection effects. Ray tracing algorithm is also supports visible surface detection, shadowing effects, transparency or multiple light source illumination.

There are many extensions to the basic ray tracing algorithm which provide very sophisticated and highly photo-realistic images generation. RT, for example, works very well with shiny objects rendering (as presented in the figure below), however it requires considerable computation time to generate, therefore inappropriate for real-time requirements.

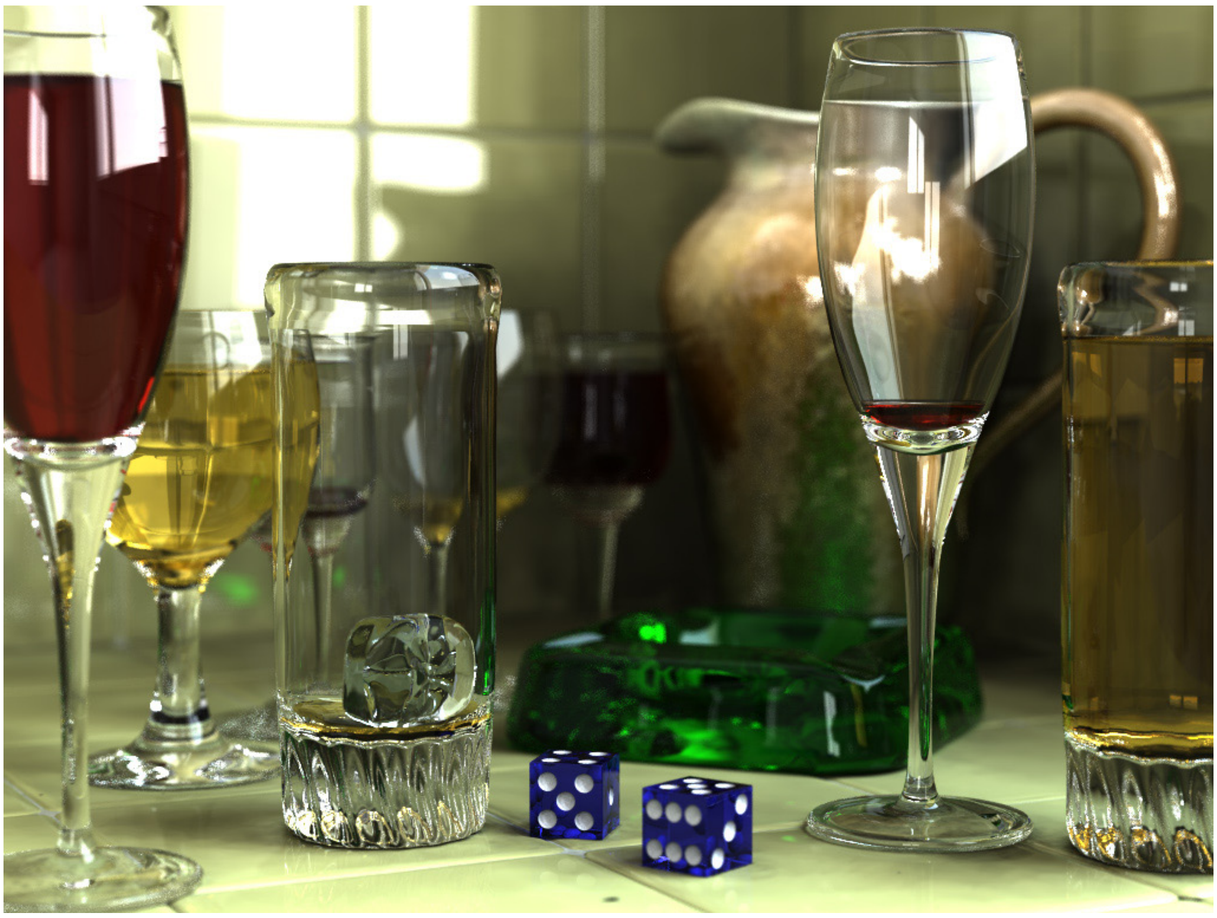


Figure 2.2: Glasses by Gilles Tran, rendered with POV-Ray, 2006

Pseudo-code for basic ray-tracing algorithm looks as follows:

Listing 1: Ray-tracing algorithm pseudo-code

```
for every pixel on the screen
{
   create 3D ray from view position passing through this pixel
   find the nearest intersection point of the ray with scene geometry

   if (intersection not found)
       fill the pixel with background color
   else
   {
       for each light fire a ray from that point to see if it is in the
shadow
       if surface is reflective – generate recursive reflection ray
       if surface is transparent – generate recursive refraction ray
       fill this pixel with the result of shading computation function
   }
}
```

At the very beginning the algorithm generates a ray path passing through the center of each pixel position on the screen. For each pixel ray it tests each surface in the scene to determine if it is intersected by the specified ray. If the surface is intersected, we calculate the distance from the camera to the intersected point – the closest intersection point identifies the visible surface from the given pixel. Subsequently, we produce the reflection (for reflective surfaces) and refraction (for transparent surfaces) rays which are called the secondary rays and we repeat the above procedure for each of them. The recursive ray tracing algorithm terminates when the depth reaches the maximum (user defined) level, or if the ray hits the light source.

The pixel color value which corresponds to the amount of illumination it receives is determined by accumulating the intensity contributions from the ray tracing tree. If no surface is intersected by the current pixel ray, the pixel is assigned the intensity of the background color. If the ray from the corresponding surface point to the light source collides with solid object then the pixel is in shadow and the light does not contribute to its shade.

The great advantage of RT over other rendering methods is its simplicity and legibility. Effects like shadows, reflection, refraction, scattering or ambient occlusion are a natural result of ray tracing algorithms as opposed to scan-line rendering algorithms. Another advantage is that the calculations for each ray can be done separately which means that the algorithm can be easily parallelized.

The most serious disadvantage of RT is the performance due to the heavy computational cost. To perform calculations in the reasonable time the scene information needs to be stored in a space-partitioning data structure, like oct-tree, kd-tree or BSP-tree. Despite of pre-computed scene information the performance is far from real-time, furthermore, the problem with virtual worlds changing dynamically arises.

2.2.2. Radiosity

Radiosity method considers the radiant energy interactions between all surface in the scene by calculation the differential amount of radiant energy (dQ) leaving at each surface point and summing the energy contribution over all surfaces. The radiosity term refers to radiant exitance, which is the power emitted from the surface. For better understanding of what the term radiosity means we look at the basic radiometric quantities explained accurately in [7].

The light can be expressed as a **radiant energy (Q)** which is the energy of electromagnetic waves. Radiant energy derives from the integration of **radiant flux** (radiant power) over time and is measured in Joules. Radiant flux is the total power of electromagnetic radiation which can be expressed as:

$$\Phi=\frac{dQ}{dt} \tag{1}$$

and is measured in Watts.

The radiant flux per unit area at a point on the surface is called **radiant flux density**. The radiant flux density is referred to as **irradiance** when the flux is arriving at the surface from any direction, and **radiant exitance** when the flux is leaving in any direction above the surface.

Irradiance is expressed in Watts per square meter as:

$$E = \frac{d\Phi}{dA} \tag{2}$$

where Φ is the radiant flux arriving at the point and dA is the differential area surrounding the point, whereas radiant exitance is defined similar to irradiance as:

$$M = \frac{d\Phi}{dA} \tag{3}$$

where Φ is the radiant flux leaving at the point and dA is the differential area surrounding the point.

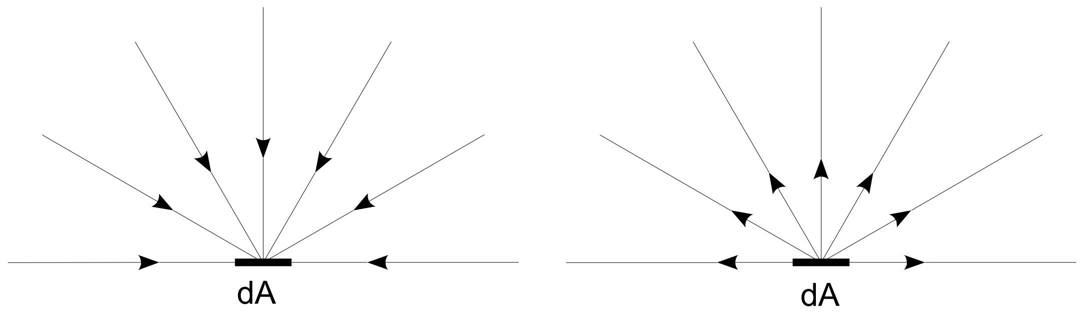


Figure 2.3: Irradiance and radiant exitance at surface point

Radiance is the amount of radiant flux contained in the ray of light arriving at or leaving the point on a surface in the given direction. Radiance is measured in Watts per square meter per steradian and is defined in the following way:

$$L = \frac{d^2\Phi}{d\omega\, dA \cos\theta} \tag{4}$$

where Φ is radiant flux, dA is the differential area surrounding the point and $d\,\omega$ is the differential solid angle of the elemental cone and θ is the angle between the ray and the surface normal.

One of the most vital quantity in radiosity calculations is the **form factor** [31] which related th proportion of energy transmitted which can be transferred to another object. Given two arbitrary oriented patches dE_i emitting some quantity of flux Φ_i and dE_j receiving a portion of emitted flux Φ_{ij} we denote the dimensionless form factor:

$$F_{ij}=\frac{\Phi_{ij}}{\Phi_i} \tag{5}$$

After expanding the above equation (in accordance with [7]) the form factor from a differential area dE_i to another differential area dE_j is given by:

$$dF_{dE_i-dE_j}=\frac{\cos\Theta_i\cos\Theta_j\,dA_j}{\pi r^2} \tag{6}$$

where Φ_i and Φ_j are angles between a line connecting dE_i and dE_j and their respective surface normals n_i and n_j and dA_j is the differential area of dE_j

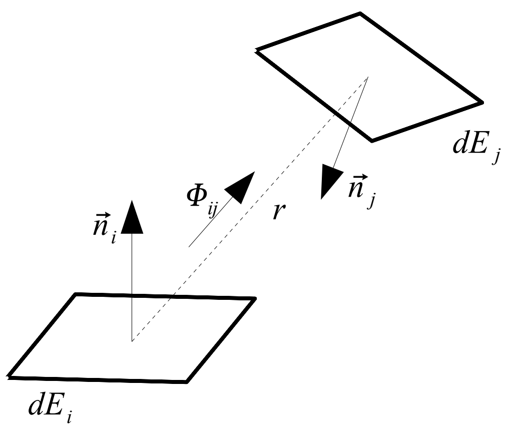


Figure 2.4: Interrelation between two arbitrary patches
(patch dE_j receiving flux Φ_{ij} from patch dE_i)

Thus, the form factor from the finite area patch E_i to another finite area patch E_j is defined as:

$$F_{ij}=\frac{1}{A_i}\int_{A_i}\int_{A_j}\frac{\cos\Theta_i\cos\Theta_j}{\pi r^2}dA_i dA_j \tag{7}$$

Having the radiant exitance and the form factor defined we can formulate the radiosity equation such as:

$$M_i=M_{0\mathrm{i}}+\rho_i\sum_{j=1}^{n}M_j F_{ij} \tag{8}$$

where $M_{0\mathrm{i}}$ and ρ_i stand for initial exitance and the reflectance of patch E_i respectively.

Radiosity method has some limitations comparing to ray-tracing techniques. First of all, the radiosity theory assumes that all surfaces are ideal diffuse reflectors and for specular surfaces and transparent materials the combination with other lighting techniques is required.

Another limitation is the geometry surface representation. Ray tracing can use implicit equations to define curved surfaces, whereas the radiosity model operates on 3D patches which can be modeled by a typically nonuniform polygon meshes. This is not a serious limitation since every surface can be approximated by a polygon mesh with the desired accuracy.

2.3. Rendering equation

A primary challenge in realistic rendering is trying to solve the rendering equation simultaneously introduced into computer graphics by David Immel and James Kajiya in 1986. The rendering equation [45] is an integral equation which denotes the total amount of illumination at specified position as a sum of emitted and reflected radiance under a geometric optics approximation:

$$L_0(x,\Theta)=L_e(x,\Theta)+\int_{\Omega_x} f_r(x,\Theta'\Leftrightarrow\Theta)L_i(x,\Theta')\cos\theta' d\omega_{\Theta'} \tag{9}$$

where:

- L_0 - total amount of illumination (exitant radiance) at x into direction Θ
- L_e - self-emitted light (self-emitted radiance) at x into direction Θ
- L_i - incoming light (incident radiance) at x from direction Θ
- $f_r(x,\Theta'\Leftrightarrow\Theta)$ - bidirectional reflectance distribution function (BRDF) at x for scattering from a direction Θ into direction Θ' (or vice versa)
- x - specified location on a surface
- Θ - outgoing direction at location x
- Θ' - incoming direction over the hemisphere around x
- θ' - angle between Θ' and surface normal at x
- $d\omega_{\Theta}'$ - infinitesimal solid angle containing direction Θ'
- Ω_x - hemisphere of directions above x

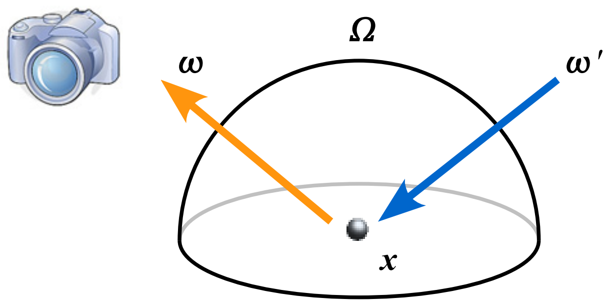


Figure 2.5: A single sphere (S, R) which occludes surface at point x

The reflected light itself is the sum of incoming light from all possible directions multiplied by the surface reflection coefficient (BRDF) and the attenuation of the inward light due to the incident angle. The BRDF function characterizes the reflectance of a surface at specified point x and is defined as a ratio of an outgoing radiance over the incoming irradiance.

The rendering equation (also known as the light transport equation) derives from the physics and the law of conservation of energy and describes the equilibrium distribution of radiance over the given environment.

2.4. Path tracing

In the 1980s, James Kajiya presented a method capable of fully solving the rendering equation which is called path tracing [48][35]. Path tracing is a probabilistic point sampling technique which extends the original ray tracing algorithm by tracing a bunch of ray paths from the camera back to the lights where the light contribution along the path is being evaluated. Although the proposed technique is very accurate and it allows to simulate nearly all effects of the light transport, it is inefficient due to the very large number of rays that need to be traced to avoid visible artifacts in the form of noise emerged in point sampling process.

Pseudo code for path tracing is shown below:

Listing 2: Path tracing algorithm pseudo-code

```
Color TracePath(Ray r, int depth)
{
  if (depth == Maxepth)
     return Color::Black;  /// Maximum depth reached

  Surface s = r.FindNearestIntersectedSurface();
  if (s == NULL)
     return Color::Black;  /// Ray hits nothing

  Material m = Surface.GetMaterial();
  Ray r2;

  r2.Origin = r.GetIntersectionPoint();
  r2.Direction =
     PickRandomDirectionOverHemisphere(r.GetIntersectionNormal());
  float cosa = dot(r2.Direction, r.GetIntersectionNormal());

  Color reflected = TracePath(r2, depth+1);
  float BRDF = m.Reflectance/PI;
  return m.Emmitance + BRDF * cosa * reflected;
}
```

In path tracing a new ray is being randomly generated (within the hemisphere of an intersected point) and traced further, whilst traditional ray tracing samples lights directly when there is a hit with diffuse surface. That means the path may hit many diffuse surfaces before interacting with a light (unlike in RT).

It is possible to trace rays in the opposite direction – from light source to the observer point. This method is called photon tracing and is use to simulate realistic photon behavior in closed environment. Photon tracing is fairly simple to implement using simple ray tracer and also gives the global illumination and radiosity solution for free, but the computation time is very big and inappropriate for real-time computer graphics.

2.5. Photon mapping

Henrik Jensen has developed a two-pass global illumination algorithm called photon mapping [43][42][91] as an alternative to pure Monte-Carlo ray-tracing techniques. In the first pass rays from the light sources and the camera are traced independently until some specified criterion is met. Subsequently, the results are being merged in the second step to produce the radiance value, which is used in realistic simulation of light interaction between objects.

Photon mapping decomposes the lighting calculations from the scene geometry by representing the solution in the spatial data structure called photon map [38] which allows to calculate the rendering equation terms separately. This makes the photon mapping very flexible and expandable for accounting the specified media effects, like sub-surface scattering caustic effects, or diffuse inter-reflection.

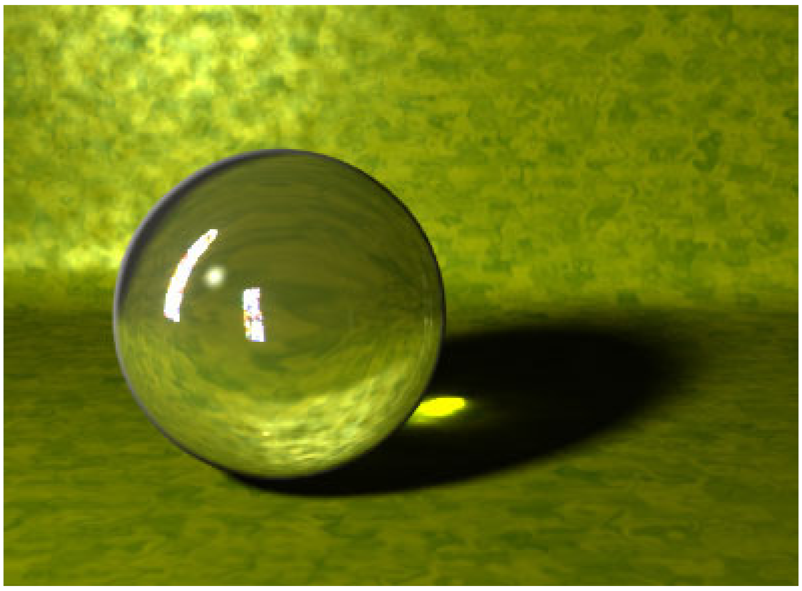


Figure 2.6: A crystal ball with caustic (rendered via photon mapping)

The first pass of photon mapping algorithm creates the photon maps by spreading the overall power of light over photons leaving the light sources. If the specified photon hits the surface it is either reflected, refracted or absorbed and the probability is calculated from the surface's material properties (Monte Carlo method called Russian roulette is used to choose one of the mentioned actions). We continue this step keeping up to date with photon maps until the photon is fully absorbed. Once the photon maps have been populated, the stored information is being used in the second pass (the rendering pass) to obtain the reflected radiance at surfaces and in participating media by gathering n nearest photons taken from the photon map (using the nearest neighbor search function). For the sake of efficiency, the rendering equation is being decomposed into separate factors, like direct illumination, indirect illumination, specular reflection and caustics:

$$L = L_{direct} + L_{specular} + L_{caustics} + L_{indirect} \tag{10}$$

Direct lighting factor can be accurately estimated by tracing a ray from the intersection points to each light source. Specular reflections can be also calculated using ray tracing procedures. The contribution of caustics is being denoted from caustics photon maps directly. Finally, the indirect lighting part comes from the reflected radiance calculated from global photon map.

2.6. Ambient occlusion

Ambient occlusion [47][73] refers to crude approximation of global illumination. In general, ambient occlusion in a specified point in 3D space is the amount of occlusion received due to surrounding occluding objects. This amount of occlusion at point x with a surface normal n is given by the following equation:

$$AO(x,\vec{n}) = \frac{1}{\pi}\int_{\Omega} V(x,\vec{\omega})(\vec{n}\cdot\vec{\omega})\,d\,\omega \tag{11}$$

where $V(x,\vec{\omega})$ represents the visibility function from x along direction $\vec{\omega}$ (returns 1 if occluded and 0 if not) over the hemisphere Ω.

There are many existing ambient occlusion algorithms since this technique greatly improves the level of realism in real-time computer graphics industry. In [47] a method called Ambient Occlusion Fields has been introduced which pre-computes a field in a surrounding space that encodes the occlusion caused by the object. This technique provides very nice results in image quality, however it is dependent of the scene complexity and not suitable for large and complex scenes, like outdoor environments. [73]demonstrates image space approach for high-frequency ambient occlusion which is similar to the method described in this chapter but they do not consider bilateral filtering nor upsampling. Additionally their method includes distant-occluder approach for low frequency ambient occlusion which is also dependent on scene complexity. The idea of bilateral upsampling is borrowed from [76] where real-time soft global illumination technique is being explained.

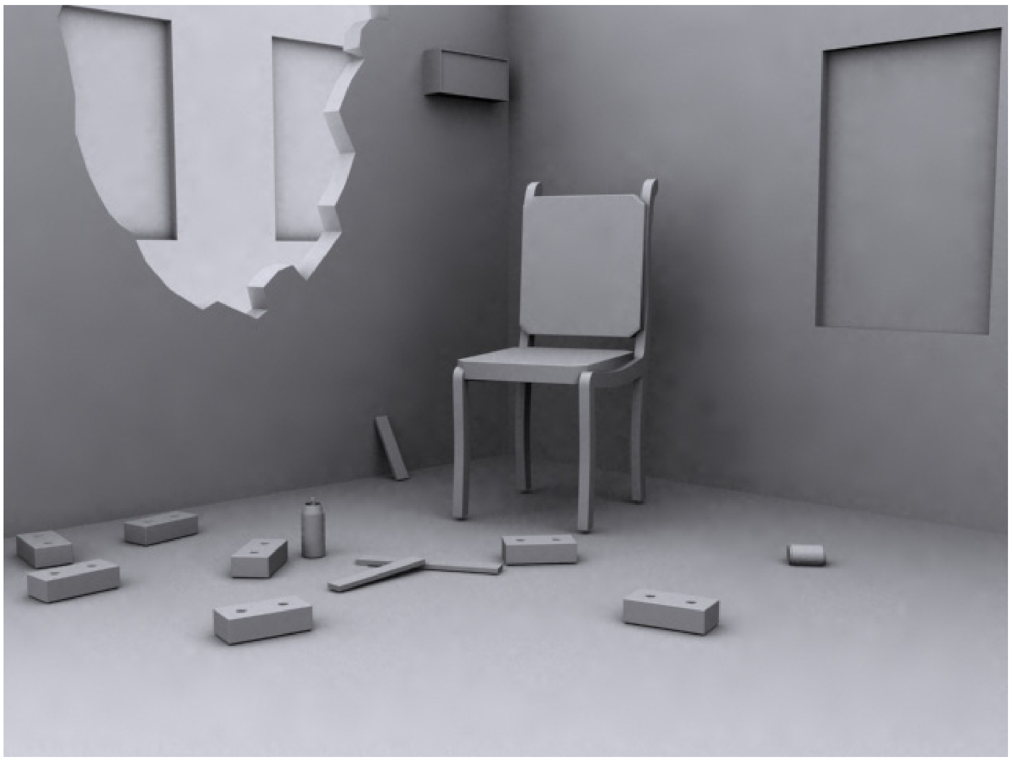


Figure 2.7: Ambient occlusion in old room scene

In this work the amount of ambient occlusion at location x is being approximated by calculating spherical cap delimited by the sphere <S,R> on the unit hemisphere at x and relative position of the occluding sphere <S,R> to x (more detailed description can be found in [73]):

$$AO(S,R,x,\vec{n}) = C(S,R,x) \cdot max(\vec{n} \cdot Sx, 0) \tag{12}$$

The surface area of the spherical cap (Fig 1) can be expressed as:

$$C(S,R,x) = 2 \cdot \pi \cdot h \cdot 1 = 1 - \sqrt{1 - \left(\frac{R}{|Sx|}\right)^2} \tag{13}$$

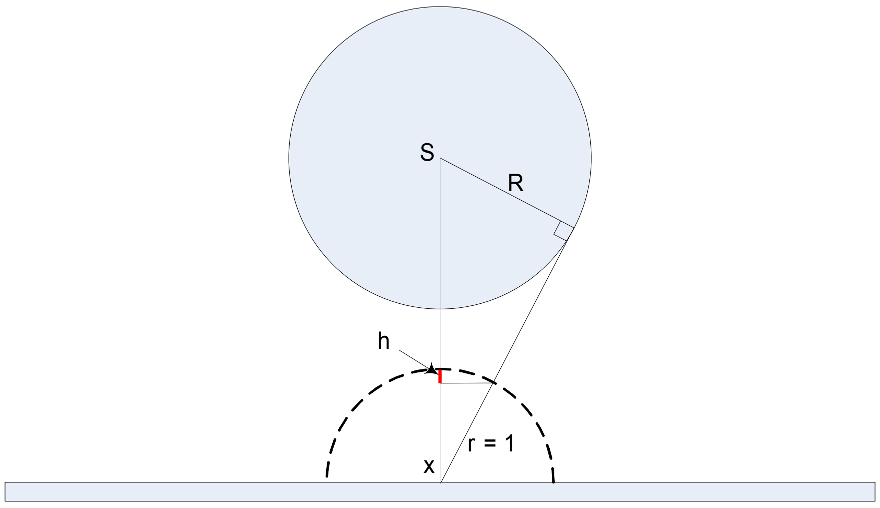


Figure 2.8: A single sphere (S, R) which occludes surface at point x

Final equation for calculating ambient occlusion at location x is given by:

$$AO(S,R,x,\vec{n}) = \left[1 - \sqrt{1 - \left(\frac{R}{|Sx|}\right)^2}\right] \cdot max(\vec{n} \cdot Sx, 0) \tag{14}$$

2.7. Deferred rendering and MRT

Deferred rendering [37] idea was primarily introduced by Michael Deering et all. in the paper entitled The triangle processor and normal vector shader: a VLSI system for high performance graphics. The term deferred shading [59][87] describes the technique which uses intermediate buffers (called g-buffer) storage with screen-space, per pixel information such as diffuse color, normal vector or depth value. A g-buffer is a collection of screen-sized render targets (MRT), which can be generated in a single pass with the modern graphics hardware (significantly reducing the rendering load) . The g-buffer is used then as an input to the shading algorithm (for example a lighting equation) without the necessity of browsing the original geometry (all information required for the calculations at this stage, like position of the pixel in 3D world space, can be extracted from the g-buffer). In this way the algorithm operates only on visible pixels which greatly reduces the computation complexity and memory bandwidth.

The main advantages of the deferred shading approach are the simplification of the rendering pipeline, ease of managing complex shader resources and calculations, and finally simple and robust management of complex lighting resources (like dynamic lighting). The technique is used with great success in modern post-processing rendering effects, like: screen-space ambient occlusion, depth of field [93], motion blur and anti-aliasing. The key drawback of this technique is inability to handle transparent objects on the scene properly. Order-independent transparency can be achieved by using depth-peeling technique [84], but at the cost of performance due to additional drawing batches and g-buffer size.

2.8. Related work

Global illumination is a very extensive area of research, especially in the last few years. Radiosity and ray-tracing based methods are often used for off-line film production rendering [81]. Real-time computer graphics usually utilizes vast approximation of global illumination

techniques, like ambient occlusion. [47] describes a real-time technique for calculating inter-object ambient occlusion by pre-computing an approximation of the occlusion caused by each object in the surrounding space. This information stored in cube map textures is used at run-time for evaluation the shadow cast on the receiving objects. Despite of good results this method suffers from large pre-computation step. It also utilizes big number of textures (cubemap per each mesh) which is not suitable for complex scenes. Finally, this technique does not handle deformable objects (due to pre-computation step) and is limited to rigid body motion.
[73] describes real-time AO approximation as a multi-pass algorithm that consists of two independent and parallel detailed and distant AO methods. First (detailed) technique utilizes image space to approximate AO due to the nearby occluders caused by high surface details, whereas the second approach uses approximated version of the occluder geometry. The high-frequency AO method requires no pre-computation step and is very easily pluggable into any existing rendering pipeline, however it requires a huge amount of surrounding samples to achieve satisfying results. This dissertation extends among other things the idea from [73] using reformulated equation for calculation the amount of occlusion due to the spherical occluder. Image-space ambient occlusion (SSAO) technique has also been extensively studied in [44] and [9].

[13] approximates AO and simple GI by treating polygon meshes as a set of disc-based surface elements which can emit, transmit, or reflect light and can occlude each other. The method works on a per-vertex level and requires huge pre-computation step which eliminates fully dynamic scenes handling. Additionally, the method requires high-tessellated scene geometry to provide good-looking, acceptable results.

[19] proposes to capture the scene geometry causing indirect illumination by an extended shadowmap and to distribute secondary light sources on directly lit surfaces. Rendering of the secondary lights contribution extends their previous work and is being performed in a

deferred shading process which makes rendering time independent from the scene complexity. The approximated indirect lighting does barely exhibit coarse artifacts, however does not produce soft shadows. Additionally, the algorithm operates on final display resolution which suffers from performance issues.

[46] utilizes shadow maps and the accumulation buffer to approximate the indirect lighting over a set of photons traced stochastically from the light source. The algorithm uses the quasi-random walk based on the method of quasi-Monte Carlo integration to generate virtual point lights which simulates indirect lighting. Instant radiosity suffers from far from real-time performance and variance problems due to inadequate VPL sampling like other Monte Carlo methods. [72] extends instant radiosity method by introducing a bidirectional sampler to find relevant virtual point lights for a given point of view in a fast and efficient way. It also utilizes deferred shading approach to optimize the influence of many point lights, however the algorithm still remains inappropriate for interactive rendering and not for real-time applications.
Another work which derives from instant radiosity is presented in [71] where Delaunay triangulation-based algorithms [34] is used for maintaining the VPLs efficiently. Proposed method is capable of rendering single bounce of indirect illumination from static geometry to static and dynamic geometry in real-time. It also lacks indirect shadows caused by the dynamic objects and did not solve the problem of VPLs visual importance for complex scenes.

[18] reformulates the rendering equation to use implicit visibility achieved by introducing new quantity called anti-radiance. Anti-radiance refers to the idea of negative light which is propagated along with the radiance and is stored in directional elements in the scene. Presented approach has some limitations which are increased memory footprint for the directional data structures and excessive subdivision for dynamic scenes. Although described technique is able to compute indirect illumination much faster than traditional methods based

on rendering equation it is still to slow for real-time computer graphics applications like complex 3D FPP games.

Precomputed radiance transfer (PRT) [77][90] offers precomputing complex lighting interactions which are being used during real-time scene rendering. PRT can be utilized to determine diffuse lighting of the scene in dynamically changing lighting environment by computing the illumination of a point as a linear combination of incident irradiance. Spherical harmonics [33][40][3] is the efficient way to encode the light transportation function in PRT method.
The similar technique named precomputed radiance maps (PRM) [80] obtains the indirect illumination caused by multiple scatterings of the light from partial light paths that are precomputed and stored in the preprocessing phase. Both PRT and PRM methods can handle dynamic light sources and give very respectable results, but work in static environment only and suffer from large precomputation step.

[10] presents the real-time radiosity method which does not use PRT or any pre-computed stuff. For the performance reasons it calculates the radiosity equation for each 16 samples per frame and then applies blurring to eliminate noise effect. The technique works similar to light maps which maps every mesh in the scene onto unique texture updated dynamically every frame. In conclusion, presented technique consumes a lot of video memory (single texture per mesh) and is quite slow which is not suitable for complex scenes, like open space environments.

Finally, [78] describes hierarchical indirect lighting computations in screen space. The algorithm is compatible with deferred rendering pipeline and is very fast (it consumes less than 10 ms per frame on GPU for typical settings). Presented technique is very similar to the ISR algorithm examined in this dissertation ([78] has emerged at the final stage of writing this thesis).

2.9. Problem formulation

Global illumination, which simulates global interaction of light with objects in a 3D scene, is a complex problem. Number of different approaches in real-time global illumination techniques have been developed and evaluated over the recent years. Majority of them can handle very simple scenes only and are not suitable for complex and dynamic worlds appearing in cutting-edge computer games and virtual reality simulators. Many of them suffers from huge precalculation step which strongly affects the application's content generation pipeline and requires additional resources like CPU power or memory capacity.

Our main goal is to develop and implement efficient method of calculating real-time radiosity algorithm that overcome the above issues and can be easily applicable in any computer graphics-based application. It is expected to be independent from scene complexity and flexible to handle both dynamic and static scenes equally. Additionally, it should produce real-time images with higher quality comparing to the images illuminated with standard direct lighting techniques.

We have given two images I_A and I_B which refer to the images rendered in real-time using standard direct lighting and rendered off-line using accurate global illumination techniques. We claim that it is possible to render image I_X in real-time which approximates global illumination in real-time producing image quality closer to the model I_B.

Above all, the solution should meet the following requirements:

- *speed*: the algorithm needs to be as fast as possible (suitable for real-time computer graphics-based applications, like computer games or flight simulators);
- *complexity*: the algorithm should be independent from scene complexity and it should work efficient for simple as well as complex virtual environments;

- *dynamics*: algorithm should be able to handle static as well as dynamic virtual scenes. Any pre-rendering or pre-calculating steps are prohibited;
- *quality*: algorithm should produce high-quality-lighting images with visible indirect lighting factor;
- *accessibility*; it is expected that new algorithm works completely in real-time and requires no content generation pipeline changes. It should also be easily implementable in any existing rendering pipeline;
- *user-friendliness*: the algorithm should avoid non-intuitive coefficients/variables and should be easily to configure and manipulate by artists to obtain satisfying results;

3. Terminology and models

This chapter introduces the model and conventions used throughout the dissertation. We also present mathematical problem formulation along with specification details.

3.1. Coordinate system

A Cartesian coordinate system is characterized by three mutually perpendicular axes: x, y and z. This dissertation uses left-handed Cartesian coordinate system with positive x-axis pointed to the right, positive y-axis pointed up and positive z-axis directed away from the viewer. The point v(x, y, z) in 3D space can be expressed as:

$$\vec{v}=\vec{i}\cdot x+\vec{j}\cdot y+\vec{k}\cdot z \tag{15}$$

where **i**, **j** and **k** are unit vectors parallel to the three perpendicular axes.

3.2. Model of 3D virtual scene

3D virtual scene (virtual space) describes the computer-simulated environment which is modeled by triple:

$$\boldsymbol{S}=(\boldsymbol{E},\boldsymbol{L},\boldsymbol{c})$$

where:

- **E** represents a collection of virtual entities **e**; an entity determines an instance of a geometric mesh **g** positioned and oriented in virtual space;
- **L** corresponds to a set of point light sources **l**;
- **c** describes virtual camera orientation in virtual space; **c** is represented by a pair of view (M_v)

and projection (M_p) matrices which determine the virtual observer position in virtual space, viewing direction and field of view;

3.2.1. Geometry

Each object in 3D virtual scene is represented by a geometric mesh build from 3D triangles spread along the vertex array. Model of a single mesh contains:

$$\boldsymbol{g}=(g_v, g_n, g_{uv}, g_{diff}, g_i, g_{mat})$$

where:

- *g_v is the array of vertices (R_3);*
- *g_n is the array of vertex normals (R_3);*
- *g_{uv} is the array of mapping coordinates (R_2);*
- *g_{diff} is the array of diffuse color values (per each vertex);*
- *g_i is the list of indexes which form the triangles;*
- *g_{mat} is the mesh material identifier;*

We define vertex normal as a resultant vector of a sum of the adjacent triangles normal vectors. Vertices of a mesh are positioned relative to the mesh origin (mesh pivot) which we call model space.

3.2.2. Lights

Lights are considered to be spherical point light sources described as:

$$l=(\boldsymbol{l_{pos}}, l_r, l_{pow}, l_c, f_{atten})$$

where*:*

- *l_{pos} is the position of light;*
- *l_r is the light radius;*
- *l_{pow} is the light power multiplier;*
- *l_c is the color emitted by the light;*
- *f_{atten} is the attenuation function of distance.*

The attenuation function for the light **l** any point **P** in 3D virtual scene is defined as follows:

$$f_{atten}(l, P) = saturate(1 - (\frac{|\boldsymbol{l}_{pos} - \boldsymbol{P}|}{l_r})^2) \qquad (16)$$

3.3. Pixels and images

We define **pixel** as a picture element which is the smallest piece of information in a digital image. In this work pixel is considered as a 3-component vector composed from red, green and blue components:

$$\vec{P} = [r, g, b]$$

The luminance [53] of a pixel P is expressed as:

$$lum(P) = 0.27\,P_r + 0.67\,P_g + 0.06\,P_b \qquad (17)$$

Pixel are arranged in a 2-dimensional grid, called image (or pixmap) which refers to the spatially mapped array of pixels. Image is characterized by its dimension and represents data structure which is used to store digital images labeled as $\boldsymbol{I}_{label}$

The difference between image I_1 and I_2 is defined as a difference image D:

$$\forall_{x \in \mathbb{N}, x < w; y \in \mathbb{N}, y < h} D_{12}(x, y) = |I_1(x, y) - I_2(x, y)| \qquad (18)$$

where $I_i(x, y)$ refers to the pixel at location x, y in the image $I_{i..}$

We also define an error factor computed from difference image D which is the mean value of all pixels components:

$$\delta_D = \frac{1}{3\text{wh}} \sum_{x=0}^{w} \sum_{y=0}^{h} D(x,y)_r + D(x,y)_g + D(x,y)_b \tag{19}$$

3.3.1. Pearson product-moment correlation

Pearson product-moment correlation coefficient (PMCC) [63][54] measures the strength of linear dependence (correlation) between two given variables. PMCC is equivalent to dividing the sample covariance between two variables by the product of their sample standard deviations.
For the finite population the PMCC is defined as follows:

$$\rho(X,Y) = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{X_i - \mu_X}{\sigma_X} \right) \left(\frac{Y_i - \mu_Y}{\sigma_Y} \right) \tag{20}$$

where:

$\frac{X_i - \mu_X}{\sigma_X}$, μ_X and σ_X

stand for standard score, population mean and population standard deviation respectively.

3.4. Quality function

In section 2.9. we have formulated the statement about the image quality better to the other one. Let's say we have two images I_1 and I_2 and the model image I_0 each representing the same scene with the view. The mathematical meaning of the quality statement can be expressed by the quality function $f_q(I_{0,} I_{1,} I_2)$:

$D_{01}=|I_0-I_1|$ - difference image between I_0 and I_1;

$D_{02}=|I_0-I_2|$ - difference image between I_0 and I_2.

$$f_q(I_{0,}I_{1,}I_2)=\begin{cases}-1, & \text{if } \delta_{01}>\delta_{02}\\ 0, & \text{if } \delta_{01}=\delta_{02}\\ 1, & \text{if } \delta_{01}<\delta_{02}\end{cases} \tag{21}$$

Quality function f_q returns -1 when I_2 is closer to I_0, 1 when I_1 is closer to I_0 and 0 when the error factors for both I_1 and I_2 are equal.

4. Image-space radiosity algorithm

In this chapter we describe the novel image-space radiosity (ISR) algorithm along with the implementation details. We also present additional effects like image-space ambient occlusion and shadowing through exponential shadow mapping technique to improve the quality of final rendered image.

4.1. Overview

This section reveals the idea of a novel image-space radiosity (ISR) algorithm. First we describe the general rendering pipeline including G-buffer configuration and ISR algorithm idea in step-by-step pseudo-code. In the forthcoming sections we present fast and efficient method for the world position reconstruction from the depth buffer which is extremely vital for ISR algorithm. Afterwards, we explain the ambient occlusion and color bleeding steps along with the implementation details. This part is the core of the ISR method and is covered very attentively. Subsequently, we present exponential shadow map implementation in deferred rendering which can be easy plugged-in into ISR rendering pipeline as an additional step to improve the level of realism in real-time computer generated images. We finish this chapter with a short conclusion and future ideas regarding ISR algorithm.

4.2. Rendering pipeline

ISR algorithm operates on every single frame and does not require any pre-computed data, nor additional content pipeline changes. It works on the fly and is suitable for simulation both simple/static and complex/dynamic virtual environments.

We start by introducing the MRT G-buffer configuration utilized during the geometry rendering pass at the very beginning of the frame generation cycle. In general, we use 4 surfaces during virtual scene rendering pass (as shown in Figure 4.1):

- hardware depth/stencil surface (DS) which is used simply for visible surface determination during geometry rendering phase; stencil buffer is used to mark out background pixels;
- albedo color surface (CLR) containing unlit rendered geometry with basic material properties (textures, diffuse color); alpha channel of this surface holds the radiation mask used during radiosity calculations;
- normal surface (NRM) which stores world space normalized per-pixel geometry normal vectors; each normal component is packed into the range from 0 to 255 (8 bits per component) and can be easily decompressed using the formula:

$$c_{unpacked} = \frac{2\,c_{packed}}{255} - 1 \tag{22}$$

- additional depth buffer (DPT) containing 32-bit floating point linear eye-space z coordinate for fast 3D view/world space position reconstruction (described accurately in section 4.3.).

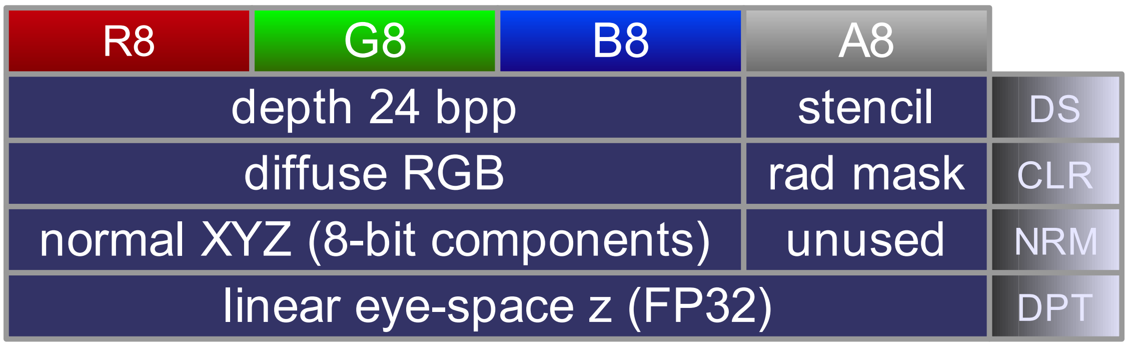


Figure 4.1: MRT G-buffer configuration

After the G-buffer is ready we perform the clear operation which is followed by rendering scene geometry with specified technique. The geometry rendering pass traverses the

scene for the visible objects and renders them into the G-buffer filling the appropriate color and depth-stencil video surfaces. This step depends on the scene complexity and is responsible for visible surface determination (i.e. frustum culling) to perform the rendering pass as efficient as possible. The visibility problem is one of the major issues in the field of 3D computer graphics and is beyond the scope of this dissertation [15][92].
At this point we have color, normal and depth buffer ready for the later use in the post-processing step. Subsequently, we reduce the sampling rate for the sake of performance during ISR/SSAO calculations by downsampling CLR, NRM and DPT surfaces into four times smaller coarse buffers (CLR_4, NRM_4, DPT_4) by averaging four adjacent samples in each buffer.

The next phase is to calculate ISR/SSAO surface with scene radiosity and ambient occlusion information. It is the most significant step of the algorithm and is examined penetratingly in section 4.4. Once the coarse ISR_4 buffer has been determined we perform blurring in both horizontal and vertical directions to get smooth representation of indirect lighting as shown in Fig 4.2. Filtering is done by convolving the image with a Gaussian function [20][30][36].
The final step is per-pixel lighting calculations which produce the ultimate view of the illuminated scene. During this phase, both direct and indirect lighting information (taken from the previous step) are being utilized to generate image with photo-realistic illumination (see section 4.5. for detailed description).

Listing 3: Image-space radiosity (ISR) algorithm pseudo-code

```
RenderFrameWithISR()
{
  /// Setup render targets
  SetupMRTBuffers(CLR, NRM, DPT);

  /// Clear G-buffer
  Clear(color, depth, stencil);

  /// Render scene (geometry) with selected technique
  RenderLevel(technique = “std”);

  /// Downsample G-buffer into 4x coarse buffers
  CLR_4 = DownsampleBuffer(CLR);
  NRM_4 = DownsampleBuffer(NRM);
  DPT_4 = DownsampleBuffer(DPT);

  /// Perform ISR/SSAO calculations on coarse buffers
  ISR_4 = RenderISR(CLR_4, NRM_4, DPT_4);

  /// Blur coarse result buffer
  ISR_4_tmp = BlurHorizontal(ISR_4);
  ISR_4 = BlurVertical(ISR_4_tmp);

  /// Final lighting composition + bilateral upsampling ISR/SSAO
  LGT = Lighting(CLR, NRM, DPT, ISR_4, DPT_4);

  /// Show final buffer (copy into back-buffer)
  Present(LGT);
}
```

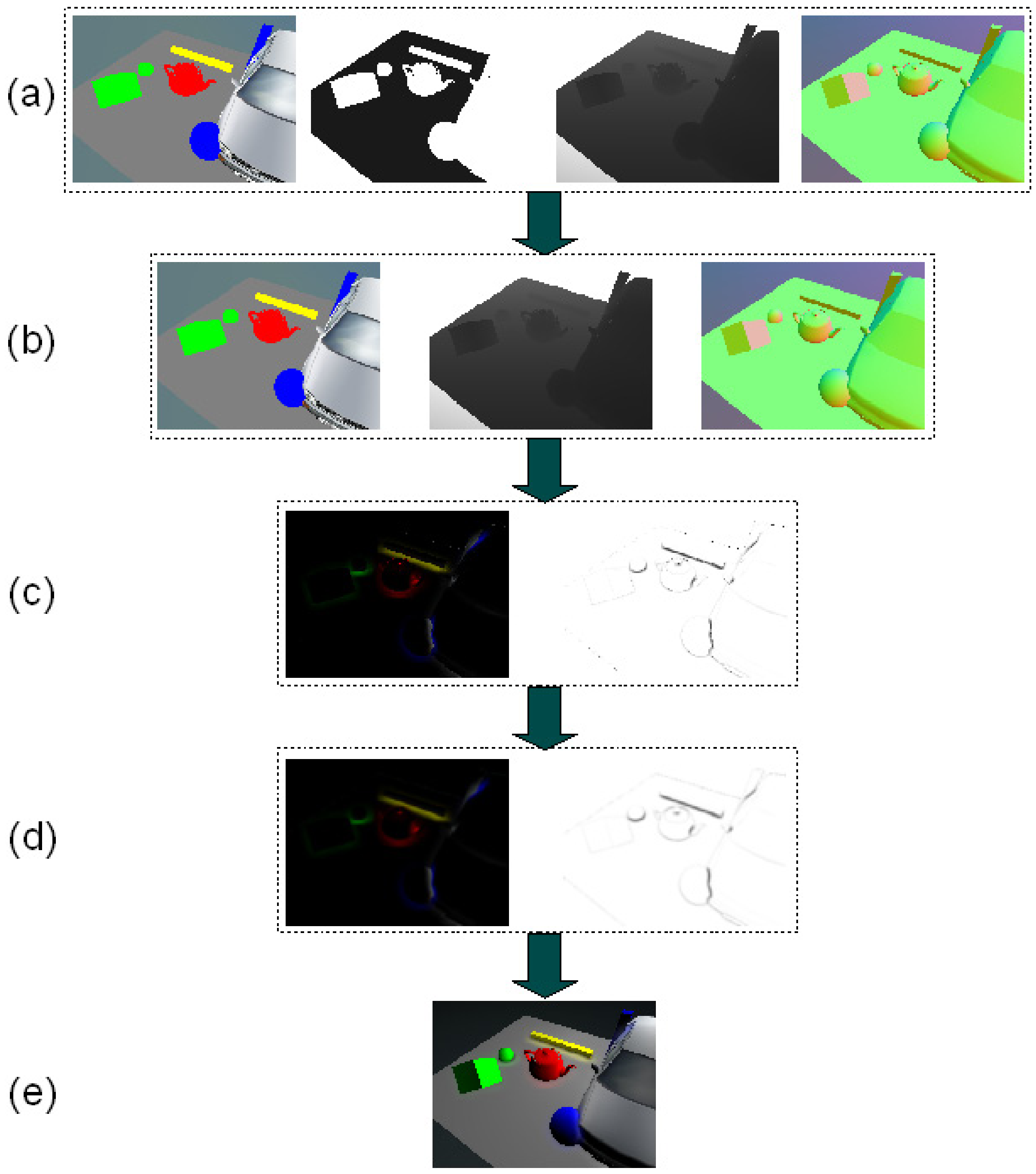
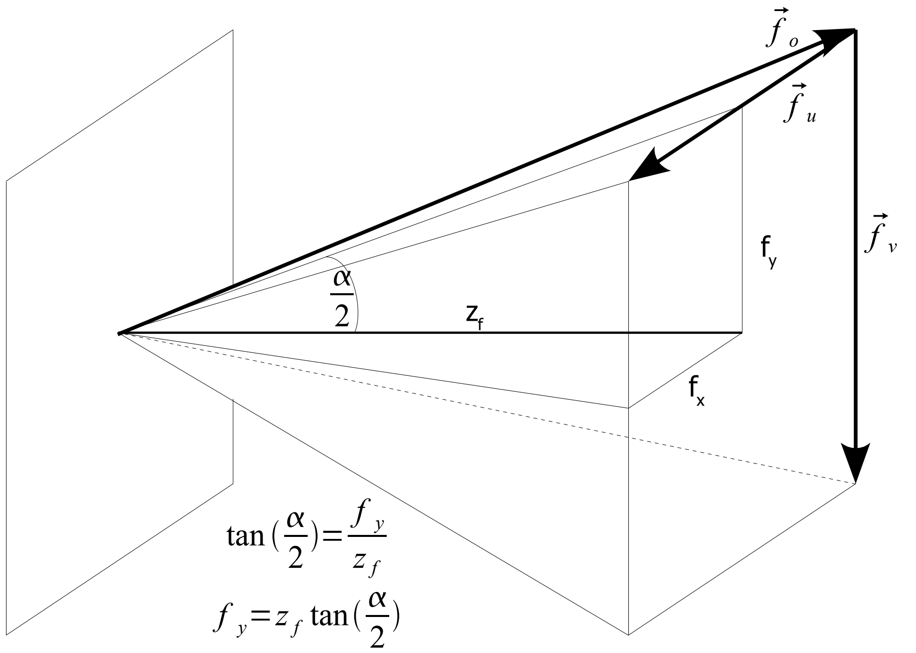


Figure 4.2: ISR rendering pipeline
(a) render G-buffer in MRT pass; (b) downsample into 4 times smaller buffer;
(c) ISR and SSAO generation pass; (d) filtering ISR and SSAO buffers with Gaussian blur;
(e) final composition with direct and indirect lighting components

4.3. Reconstructing world position (linear eye-space z method)

During screen-space ambient occlusion or screen-space color bleeding calculation there is a necessity to retrieve world space coordinates of the adjacent pixels. In other words, we need to reconstruct a specified number (i.e. 24) of world space coordinates per each screen pixel. That means we need to perform the calculations as fast as possible to avoid performance issues.



$\vec{f}_o$
$\vec{f}_u$
$\vec{f}_v$
f_y
$\frac{\alpha}{2}$
z_f
f_x
$\tan(\frac{\alpha}{2})=\frac{f_y}{z_f}$
$f_y=z_f\tan(\frac{\alpha}{2})$

Figure 4.3: Viewing camera frustum

The idea is to calculate specific frustum far plane orientation vectors $\vec{f}_o, \vec{f}_u, \vec{f}_v$ which multiplied by the normalized linear z coefficient will result in the expecting view or world space 3D position.

At first, we need to store linear eye-space z value instead of post-perspective z coordinate during MRT rendering. We normalize this value by dividing it by the frustum far coefficient.

The HLSL shader code looks as follows:

Listing 4: Render depth buffer in MRT pass

```
/// Vertex shader code
float4 pos_in_world = mul(input.Position, WorldMatrix);
float4 pos_in_view = mul(pos_in_world, ViewMatrix);
output.Depth.x = pos_in_view.z * Frustum.w; // (equivalent to:
pos_in_view.z / Frustum.z)

/// Pixel shader code
output.Depth = float4(input.Depth.x, 0, 0, 1);
```

To summarize the depth buffer creation, we store the linear z from camera space divided by the far frustum plane value z_f instead of the post-perspective z from projection space.

Afterwards, in order to get the 3D view/world space position from depth buffer we need to denote the camera far plane point according to the specified screen coordinates. In order to achieve this we need to calculate three vectors $\vec{f}_o, \vec{f}_u, \vec{f}_v$ as shown in figure 4.3.

For the view space we simply need to calculate the upper-left corner of camera frustum in view space:

$$\begin{aligned} f_x &= z_f \tan\left(\frac{\alpha}{2}\right) \\ f_y &= aspect \cdot f_x \\ f_z &= z_f \\ f_w &= \frac{1}{z_f} \end{aligned} \qquad (23)$$

where z_f is the camera far plane, α is the vertical field of view and $aspect$ means the screen resolution width to height aspect. We also keep $\frac{1}{z_f}$ in f_w coordinate to avoid division operation in the vertex shader.

Having the upper-left corner of camera frustum we can deduce any point in frustum using $\vec{f}_o, \vec{f}_u, \vec{f}_v$ vectors (see figure 4.3):

$$\vec{v}(x,y)=\vec{f}_o+x\vec{f}_u+y\vec{f}_v \tag{24}$$

therefore 3D view space position has the following form:

$$\vec{v}(x,y)=\vec{f}_o+x\vec{f}_u+y\vec{f}_v=(-f_x,f_y,f_z)+x(2f_x,0,0)+y(0,-2f_y,0)=(xf_x,-yf_y,f_z)$$

For 3D world space position reconstruction we need to transform $\vec{f}_o, \vec{f}_u, \vec{f}_v$ vectors into world space by multiplying them by inverse view matrix M_v^{-1} .

At this point we have everything we need to reconstruct 3D world/view position form depth buffer in a very efficient way.
Pixel shader code using our alternative version is much more simpler now:

Listing 5: 3D view/world position reconstruction from screen space

```
/// p.xy [0..1, 0..1], p.z - sampled linear eye-space z
float3 ScreenSpaceToCS(float3 p)
{
  float2 h = p.xy * 2 - 1;
  return float3(Frustum.xy * h.xy, Frustum.z*p.z);
}


/// p.xy [0..1, 0..1], p.z - sampled linear eye-space z
float3 ScreenSpaceToWS(float3 p)
{
  return CameraPos.xyz + (FrustumO.xyz + FrustumU.xyz*p.x +
     FrustumV.xyz*p.y) * p.z;
}
```

4.4. ISR step

Calculations for the ambient occlusion and radiosity factors in ISR method are being performed completely in screen space. The algorithm takes three input buffers:

- coarse color buffer (CLR_4);
- coarse normal buffer (NRM_4);
- coarse depth buffer (DPT_4);

and produces one output buffer (ISR_4) with illuminated pixels which approximate global illumination. The result buffer is being utilized in the further lighting phase.
ISR denotes indirect lighting bounce as well as ambient occlusion factor in the following way: for every pixel corresponding to 3D world position P with normal n, the direct radiance is being calculated from fixed N neighborhood samples mapped over the surrounding hemisphere, each covering the solid angle of $\frac{2\pi}{N}$ using the radiosity equation in the following form :

$$L_P(X)=\rho_P\frac{2\pi}{N}F_{PX}L_X \tag{25}$$

where $L_P(X)$ is the radiance at point P coming from point X, ρ_P is the reflectance sampled from alpha channel of CLR_4, N is the number of samples, F_{PX} is the form factor between points P and X and finally L_X is the initial radiance at point X sampled from RGB channels of CLR_4. Equation (25) calculates color bleeding from point X into point P and represents derived form of radiosity equation (8).

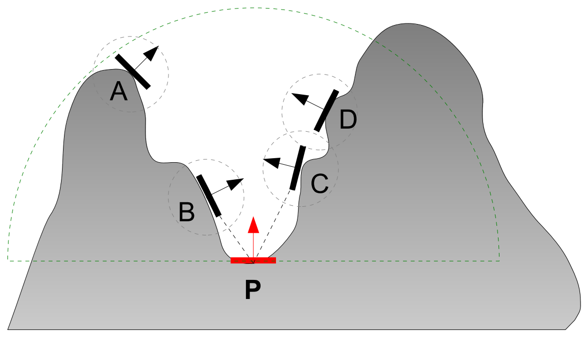


Figure 4.4: Pixels as patches in 3D space

As shown in figure 4.4, each pixel on the screen corresponds to the surface patch in 3D space with normal vector and associated area based on the camera space z component sampled from the depth buffer. This statement facilitates calculation form factors between adjacent pixels which is fundamental in determining the radiance value.
Having pixel P and the neighborhood samples A,B,C,D transformed into world space we can calculate the form factor from equation (6) straightforwardly.

We also denote SSAO at point P by determining the amount of occlusion by treating each neighborhood sample as a spherical occluder to point P. We utilize the equation (14) in the following form:

$$SSAO_P(X)=\frac{2\pi}{N}\left(1-\sqrt{1-\left(\frac{r}{|PX|}\right)^2}\right)\cdot max(\vec{n}\cdot PX,0) \tag{26}$$

where $SSAO_P(X)$ is the amount of occlusion at point P caused by occluder at X, r is the occluder radius based on the distance in camera space sampled from the depth buffer and $\vec{n}$ is the normal at point P sampled from the normal buffer.

Both radiance and ambient occlusion components are being accumulated per pixel in the output ISR_4 result buffer.

Listing 6: Per-pixel ISR step algorithm pseudo-code

```
for each pixel (input TexCoord0)
{
  n0 = decode(sample normal from NRM_4 at TexCoord0);
  z0 = sample z from DPT_4 at TexCoord0;
  p0_ws = ScreenSpaceToWS(TexCoord0, z0);

  ao = 0;
  gi = (0,0,0);

  for (z=0; z<num_samples; z++)
  for (x=0; x<num_samples; x++)
  {
     tx1 = calculate offset for SSAO sample
     tx2 = calculate offset for ISR sample
     n1 = decode(sample normal from NRM4 at tx1);
     n2 = decode(sample normal from NRM4 at tx2);
     z1 = sample z from DPT_4 at tx1;
     z2 = sample z from DPT_4 at tx2;
     c1 = sample color from CLR_4 at tx1;
     c2 = sample color from CLR_4 at tx2;
     p1_ws = ScreenSpaceToWS(tx1, z1);
     p2_ws = ScreenSpaceToWS(tx2, z2);

     /// Calculate ISR factor from equation (25)
     gi += CalculateIsrFactor(p0_ws, p2_ws, n0, n2, z0, z2, c2);

     /// Calculate SSAO factor from equation (26)
     ao += CalculateSsaoFactor(p0_ws, p1_ws, n0, z0, z1);
  }

  return float4(gi.xyz, ao);
}
```

4.5. Lighting phase

After filling the buffer with ambient occlusion and indirect lighting factors we can go through the ultimate per pixel lighting calculations. The most important part of the lighting phase is the upsampling from coarse ISR_4 buffer into the native resolution before the final lighting calculations for the corresponding pixel. The most intuitive bilinear filtering during upsampling causing discontinuities in the receiver geometry visible across the object silhouettes. To address this issue we have changed the interpolation kernel and handles upsampling with a variant of bilateral filtering [85][25][57][61]. For each target pixel we take four samples from the coarse ISR_4 buffer and interpolate them through weights based on their difference with the depth of the corresponding pixel. This approach leads to the contribution of source samples that are similar to the target pixel and greatly reduces discontinuities between distant pixels.

Given the target pixel p at specified location and corresponding 2×2 block of coarse source samples indexed by i=1,2,3,4 we denote the interpolation weights by:

$$w_i' = w_i^b w_i^z \tag{27}$$

where the individual weight factors are defined as following:

$$w_i^b = \{(1-x)(1-y), x(1-y), (1-x)y, xy\}$$

$$w_i^z = \frac{1}{\epsilon + |z_p - z_i|}$$

w_i^b corresponds to the standard bilinear weights determined by the 2D position (x, y) of the target relative to the source samples whereas w_i^z denotes the dimensionless bilateral upsampling weights based on the difference with the depth of the source samples and target pixel.

Final weight factors used for interpolation are being normalized using the formula:

$$w_i = \frac{w_i'}{\sum_{j=1}^{4} w_j'} \tag{28}$$

As mentioned in [76] it is possible that all four unnormalized weights w_i' are nearly zero which means no coarse sample sufficiently matches the target pixel. To address this issue we could perform an additional pass with an accurate shading calculations utilizing the entire list of contributing proxies. This problem however arises at very few pixels along silhouettes and has been neglected throughout this dissertation since no visual artifacts has been perceived.

Once the ISR information has been leveraged, we can compute the final pixel illumination applying the formula as follows:

$$\vec{L_{final}} = \vec{C}_0 k_{dl}\, saturate(L_{SSAO} + 1 - k_{SSAO}) + \vec{L_{ISR}} k_{ISR} \quad (29)$$

where:

- $\vec{C}_0$ is the color sampled from CLR color buffer at the corresponding pixel location;
- k_{dl} is the direct lighting coefficient;
- L_{SSAO} is the weighted ambient occlusion factor;
- k_{SSAO} is the ambient occlusion power (from 0 to 1);
- $\vec{L_{ISR}}$ is the weighted indirect lighting factor (calculated from equation 25)
- k_{ISR} is the indirect lighting power (from 0 to 1).

By changing the k_{SSAO} and k_{ISR} we can change the contribution of ambient occlusion and indirect lighting factors. These factors have been balanced experimentally to get the most convincing results.

Pseudo-code below represents the shader which calculates per-pixel lighting including ambient occlusion and indirect lighting components:

Listing 7: Per-pixel lighting pseudo-code

```
for each pixel (input TexCoord0)
{
  n0 = decode(sample normal from NRM at TexCoord0);
  z0 = sample z from DPT at TexCoord0;
  c0 = sample color from CLR at TexCoord0;
  p0_ws = ScreenSpaceToWS(TexCoord0, z0);

  isr = (0,0,0);
  ao = 0;

  float4 bilinear_weights = CalculateBilinearWeights();
  float4 bilateral_weights = CalculateBilateralWeights();
  float4 weights = bilinear_weights * bilateral_weights;

  coarse_isr = get 2x2 block of ISR_4 source samples;
  isr = accumulate coarse_isr samples using weights factor;

  k_dl = calculate direct lighting using n0;

  if (shadowing enabled)
  {
     sf = CalculateShadowingFactor();
     k_dl = k_dl * sf;
  }

  return c0.xyz * k_dl * saturate(ao+1-k_ssao) + k_isr*isr.xyz;
}
```

The above listing modifies the direct lighting factor when shadowing calculations are being enabled (see the next section for more details).

4.6. Shadowing

Improved shadowing algorithm based on exponential shadow maps [68][69][2][70] has been implemented to increase visual quality of rendered images. ESM derives from shadow mapping technique [52][55][79] and denotes the shadowing factor by evaluating the continuous exponential function of the distance between shadow caster and shadow receiver:

$$f_e(r,o)=\mathrm{e}^{-c(r-o)} \tag{30}$$

where c is a constant coefficient which modifies the difference between the shadow map pixel value and the depth value in the light space (r-o). This approach alleviates the difference between shadowed and lit regions resulting in sort of soft shadowing effect which gives significantly higher quality shadows. The c coefficient can be used to control the level of softness on the border of the shadowed regions.

Comparing to standard shadow mapping technique, ESM introduces shadow map filtering step:

- render shadow map (depth map) from the light point of view;
- **filter (blur) the shadow map (i.e. using Gaussian blur technique);**
- render the scene and determine the shadowed regions taking advantage of exponential function $f_{e.}$

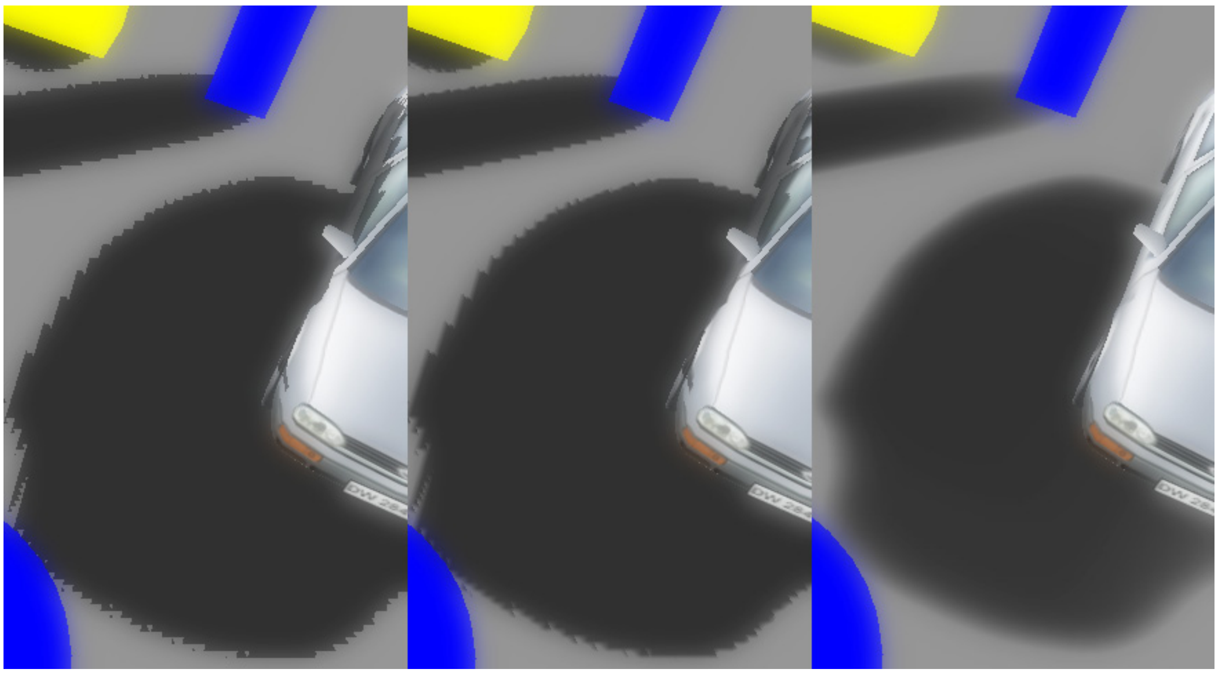


Figure 4.5: Shadowed regions quality differences
starting the left side: a) standard shadow mapping technique; b) PCF technique; c) ESM technique

The shadow map generation process remains the same as for the standard shadow mapping algorithm. The next step is to blur the shadow map using floating point format temporary texture. The results presented in Figure 4.5 are achieved by logarithmic blur utilization explained in [68].

The filtered shadow map is used during deferred rendering full screen quad lighting phase. Each pixel is transformed from screen space into the world space and then into the ligh space to determine the shadow factor through the equation 30. Additionally, percentage cloder filtering (PCF) technique has been applied to reduce aliasing and roughness artifacts on the shadowed region border. The HLSL code below introduces ESM+PCF algorithm implementation details:

Listing 8: ESM shadowing with PCF in deferred rendering pass

```
float4 ls = mul(float4(p0_ws.xyz, 1.0f), LightViewProjMatrix);
ls.xy = float2(0.5f, 0.5f) + 0.5f *  ls.xy;
ls.y = 1.0f - ls.y;

#if SHADOWS_ESM == 0
  float eps = 0.005f;
  shadow_factor = (tex2D(SamTexShadowMap, ls.xy) + eps < ls.z) ? 0.0f:
1.0f;
#else

  /// Calculate bilateral weights
  float2 SM_SIZE = float2(512 ,512);
  float2 unnormalized = ls.xy * SM_SIZE;
  float2 fractional = frac(unnormalized);
  unnormalized = floor(unnormalized);

  float z_bias = 0.0f;
  float zw = ls.z;

  float4 shadow_s;

  shadow_s.x = zw - tex2D( SamTexShadowMap, (unnormalized.xy +
      float2(0,0))/SM_SIZE ) - z_bias;
  shadow_s.y = zw - tex2D( SamTexShadowMap, (unnormalized.xy +
      float2(1,0))/SM_SIZE ) - z_bias;
  shadow_s.z = zw - tex2D( SamTexShadowMap, (unnormalized.xy +
      float2(0,1))/SM_SIZE ) - z_bias;
  shadow_s.w = zw - tex2D( SamTexShadowMap, (unnormalized.xy +
      float2(1,1))/SM_SIZE ) - z_bias;
  shadow_s = max(shadow_s, 0.0f) * saturate(dot(-n0, normalize(p0_ws.xyz-
     SunPos.xyz)));

  float4 shadow4 = exp(-90.0f * shadow_s);

  shadow_factor =lerp(lerp( shadow4.x, shadow4.y, fractional.x ),
                     lerp( shadow4.z, shadow4.w, fractional.x ),
                     fractional.y );
#endif
```

4.7. Discussion

During final per-pixel lighting calculations the ambient occlusion power k_{SSAO} and indirect lighting power k_{ISR} coefficients have been defined to manipulate the intensity of the corresponding factors. For the sake of this dissertation these values have been set to 0.5 resulting in subtle but visible ambient occlusion and indirect lighting contribution.

More intuitive way of 3D view/world space position reconstruction from depth buffer is to multiply the screen-space vector by the inverse of view/view-projection matrices and divide by w coordinate (apply reverse transformation). This solution, however, requires more math operations and turned out to be over than twice slower than the linear eye-space z method described in section 4.3.

ESM method with PCF technique creates very realistic soft shadows as presented in Figure 4.5. For high quality distant shadows a technique called cascaded shadow maps (CSM) [22] can be used which fits properly as an extension to the proposed solution. There is also a possibility to render ESM+PCF shadows into the separate screen-sized buffer and perform full screen filtering to get shadows even more smoother.

5. Experiments and evaluation

In this chapter we present the practical approach of the ISR algorithm through experiments on different kinds of virtual scenes. At the very beginning we describe the schedule of the experiments and also introduce the tools and packages that have been utilized. Subsequently, we conduct the experiments on different types of virtual scenes and present the results.

5.1. Tools

This section introduces tools and libraries utilized to examine the ISR algorithm. We start by describing POV-Ray tool which has been used to generate very realistic model images. Next section presents 3DSMAX with MentalRay tool which also generates high quality realistic images applicable in the movie industry. 3DSMAX is also used for managing and exporting 3D virtual scene into V-Engine tool described in the consecutive section. Finally, we introduce author's Imalyzer tool used for evaluating generated images and present brief description of Perl dynamic programming language utilized for conducting the experiments.

5.1.1. POV-Ray

POV-Ray [60] (The Persistence of Vision Ray-tracer) is a computer graphics application for creating photo-realistic computer-generated images and animations using ray tracing techniques. The main renderer reads the scene description stored in human-readable script and produces very high quality images with realistic with advanced lighting, reflections, shadows and other effects. The most important features of POV-Ray are:

- easy to use scene description language;
- very high quality output images (up to 48-bit color);

- different light types, Phong and specular highlighting for realistic lighting;
- **inter-diffuse reflection (radiosity) for realistic global illumination effect;**
- photon mapping for reflections, refractions and caustics;
- natural phenomena effects like atmosphere, ground fog and rainbow;
- particle media for clouds, dust, fire or steam modeling.

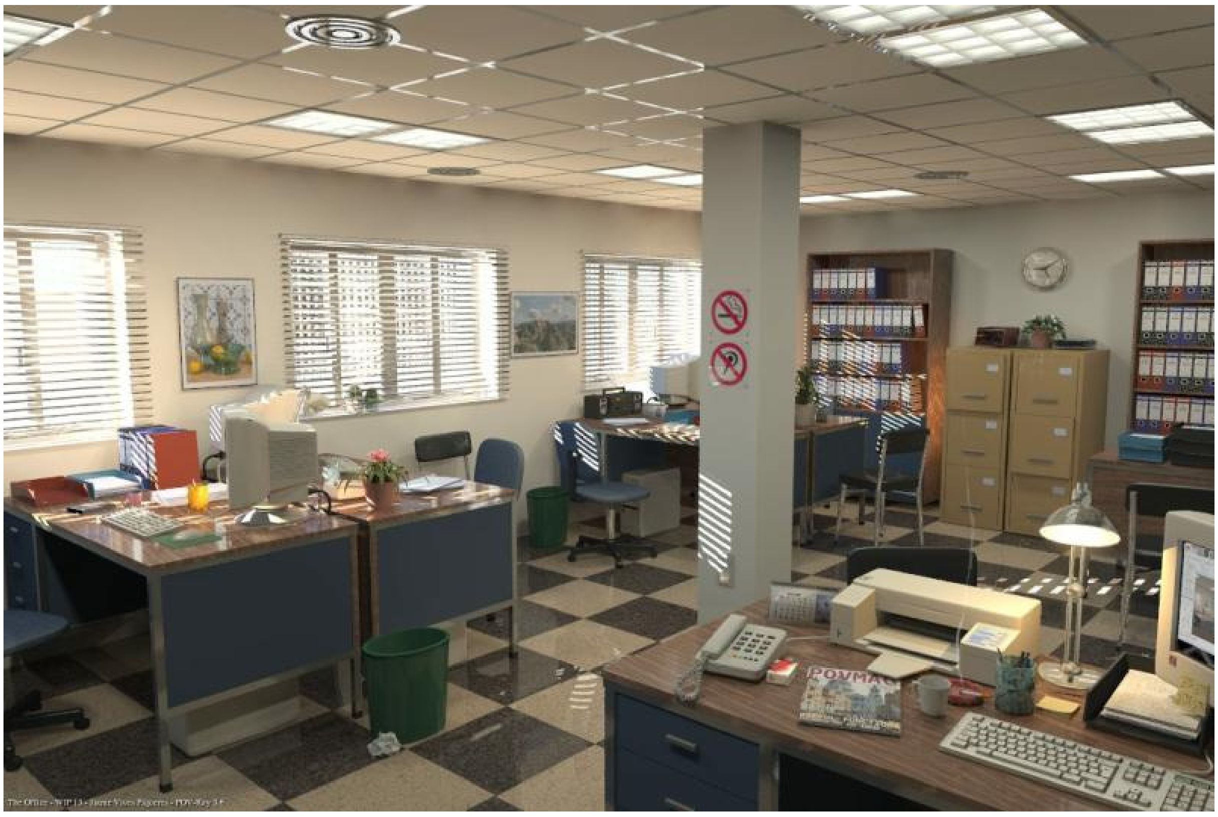


Figure 5.1: Office by Jaime Vives Piqueres, rendered with POV-Ray, 2004

The most important feature is the ability of accurate radiosity calculations via ray tracing method. The full feature list, description and tutorials can be found in POV-Ray documentation page [60].

Note: all POV-Ray renderings in this dissertation were made on the POV-Ray version 3.7 beta 32 .

5.1.2. 3DSMax and MentalRay

3D Studio MAX® is a professional modeling, animation and rendering commercial package developed by Autodesk® Media and Entertainment [8]. 3DSMAX is intended to produce high quality, photo-realistic images used in the creation of top-selling computer games and award-winning film and video content. The application has a built-in off-line high quality rendering plug-ins, such as Mental Ray [51] by Mental Images, or RenderMan [58] developed by Pixar Animation Studios.

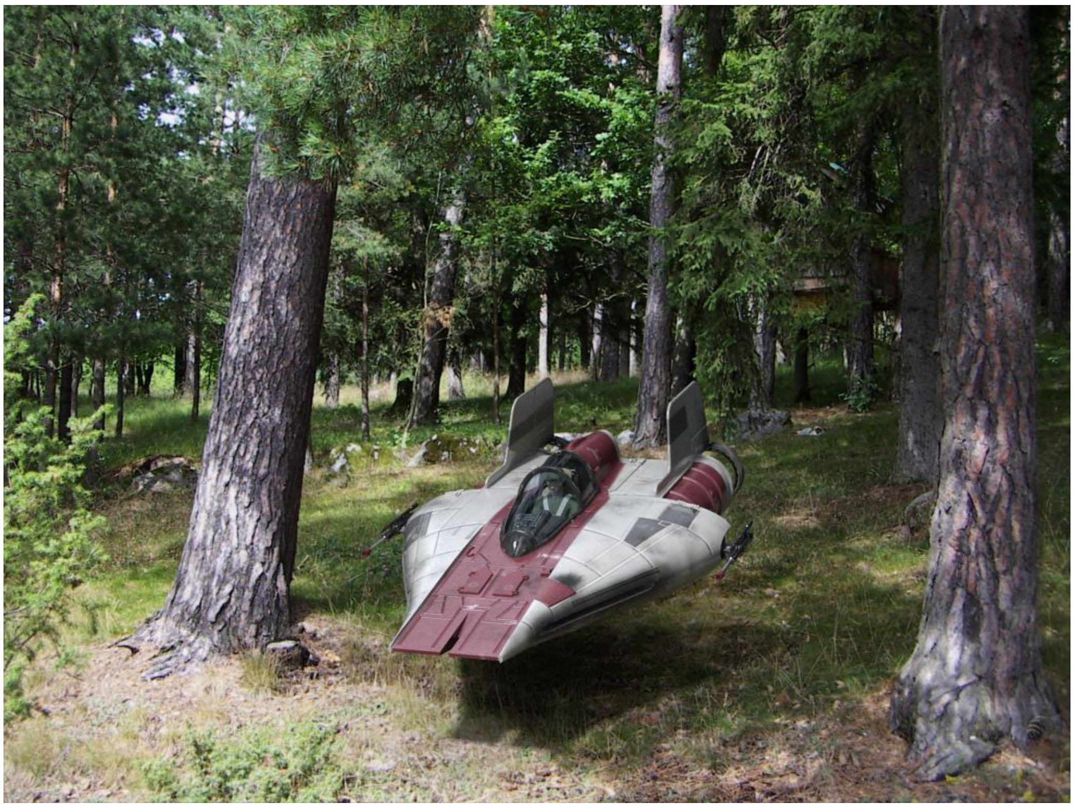


Figure 5.2: A-Wing in the forest rendered with MentalRay

Mental ray is a 3rd party application which produces high quality, photo-realistic computer images using ray tracing method. The most important future of mental ray is the ability of high performance parallelized rendering on multi cores and across render farms

which significantly improves the computation time. The tool has a capability to simulate any combination of diffuse, glossy and specular light reflection and transmission. It also provides support for caustics and physically correct simulation of global illumination utilizing photon maps.
Mental ray has been employed in the several recent films, including *Hulk, Matrix Reloaded, Matrix Revolutions, or Star Wars Episode II: Attack of the clones.*

5.1.3. V-Engine

V-Engine is an object-oriented graphics rendering engine written in C/C++, fully designed and implemented by the author for the sake of testing and verifying various 2D/3D graphics algorithms. The rendering architecture is designed to be simple, flexible and easily applicable via Lua scripts giving the ability to control the whole rendering process from the user level. Since the engine has a long list of features (see below) and it is well optimized, it can be used as a basis for making video games or other real-time computer graphics application.

V-Engine was used to audit the ISR method from the practical point of view. The key features of V-Engine are as follows:

- *neat and robust object-oriented architecture based on easily implementable interfaces;*
- *clean and well designed C++ implementation of engine classes;*
- *Direct3D 9 renderer implemented via easily extensible and flexible platform independent rendering system (can be easily extended to OpenGL, Direct3D 10/Direct3D 11, or software rendering library support) [1];*
- *support for multi-threaded rendering tasks via separate render queues;*
- *dynamic visibility determination through frustum culling, occluders and portals;*

- *powerful compositor system which allows to control every step in the rendering pipeline via simple and easily configurable LUA [41] scripts;*
- *example post-processing scripts, like: screen-space ambient occlusion, depth of field, bloom, tone-mapping, motion-blur, sepia, radial-blur, glass, tiling and ISR;*
- *highly customizable geometry support via different vertex declarations and geometry streams (integration with 3DSMax exporter tool);*
- *support for all modern per-pixel lighting and rendering techniques including bump-mapping, parametrized Phong lighting, virtual displacement mapping, deferred shading dynamic lighting, etc.;*
- *billboarding for 3D-sprite graphics;*
- *support for sky-boxes, sky-domes, sky-planes and layered clouds for natural atmospheric effects simulation;*
- *flexible terrain system based on height maps with level-of-detail support for fast and efficient large-scale terrain rendering;*
- *dynamic shadowing using real-time generated exponential shadow maps for smooth shadows;*
- *powerful material and shader system with HLSL support allows to manage and modify materials in real-time without recompiling the code;*
- *support for different texture formats, like: PNG, DDS, JPEG, BMP and DDS; support for different texture types: 1D, 2D, volumetric textures and cube maps;*
- *resource management through virtual file system which allows to load data from memory, HDD, or ZIP archive files transparently;*
- *rendering to texture support allows for real-time reflections and static scene captures;*
- *extensible event system for efficient inter-system communication;*
- *scripting support via LUA scripts;*
- *flexible, object-oriented GUI system architecture [64] with basic predefined widgets and container controls;*

- *built-in POV-Ray scene description language exporter.*

5.1.4. Imalyzer

Imalyzer (Image Analyzer) is an author's application for comparing the quality of the specified image files. The program operates on pixel color values and produces error calculations between two input images. Imalyzer has also the ability to calculate Pearson product-moment correlation coefficient which is utilized to verify the quality of the corresponding image with relation to the model image.

5.1.5. Perl

Perl [89][74] is a high-level, general purpose dynamic programming language which has been developed by Larry Wall since 1987. The overall structure of Perl is borrowed from C, which means it is a procedural language with variables, expressions, statements, control structures and subroutines.

Perl is mostly used for text manipulation and reporting but it also handles wide range of tasks including system administration, web development, network programming games and GUI development.

5.2. Experiment course

In this section we describe the complete experiment process applied for the ISR algorithm evaluation. First we present the main concept along with the experiment pipeline. Subsequently, we go through the detailed description of each step of the experiment process and finally we summarize the observations and reveal the results.

5.2.1. The idea

The main purpose of the experiment is to compare the quality of real-time rendered images using both standard direct lighting technique and ISR technique for different types of virtual scenes. Both images are compared against the model image rendered off-line using advanced time consuming ray-tracing algorithm. The image that is less different from the model image (has a higher similarity level) is considered to have higher quality.

We use the following notation throughout the experiment process:

- I_{DL} stands for the image rendered in real-time with standard direct lighting technique;
- I_{ISR} stands for the image rendered in real-time with ISR algorithm;
- I_{POV} stands for the model image rendered using POV-Ray application.

For evaluation purposes we have prepared a dozen of different virtual scenes representing diverse environments starting from simple Cornell radiosity box, through simple objects and indoor rooms, corridors to open space wild-west scenarios with terrain, trees, vegetation, characters, animals and objects like wagons, boxes, cans, debris, fences, etc. The variety of virtual scenes has a significant meaning during the experiment since we wanted to be sure that we have performed the tests on as much general input as possible.
Additionally, we have created an input script file containing the list of camera views for each scene. Script file can be created via V-Engine simulator manually, or automatically by drawing random camera view parameters. The main idea is to generate the different scenarios for the comparisons between direct lighting and ISR techniques.

Collected input scenes along with the script containing the list of camera views go to the V-Engine application which for each case generates the following:

- snapshot image with standard direct lighting method;

- snapshot image with ISR lighting algorithm applied;
- POV-Ray input script with corresponding camera view parameters and 3D scene information.

The snapshots are being stored in PNG files (bitmap image format with lossless data compression) for further evaluation. POV-Ray scripts are being passed through the POV-Ray renderer application to produce model images corresponding to the V-Engine snapshots. Finally, Imalyzer tool compares the generated images to perform image quality analysis.

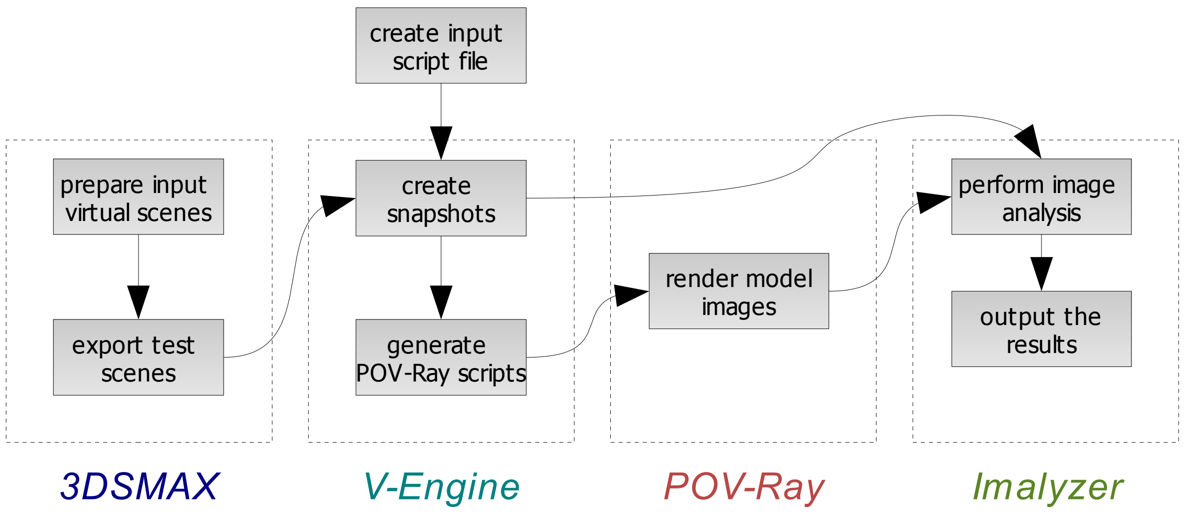


Figure 5.3: General experiment pipeline

The figure above demonstrates the general experiment pipeline which will be discussed more accurately in the following sections.

5.2.2. Test sets

Different types of test scenes have been carefully selected for the ISR algorithm evaluation. We have created several simple scenes, like group of colored primitives, office, garage scene or Cornell radiosity box which are widely used to determine the accuracy of rendering software. Additionally, we have randomly selected almost three hundred test cases from commercial game “Call of Juarez” to verify our algorithm in true, existing project.

Call of Juarez [14] is a FPP Western-themed shooting PC/XBox360 game developed by Techland [83] and published in 2007 by Ubisoft Entertainment [86]. The game is loosely based on a number of Western movie hits from sixties and early seventies. Many game reviewers appreciated its game-play and graphics focused on high level of realism, breathtaking environments and fantastic natural phenomena effects introducing great Western atmosphere.

Figure 5.4: Call of Juarez in-game screenshot

Variety of different locations and scenarios makes Call of Juarez game a very good ISR algorithm test case (world of COJ is dynamic, complex and fulfilled with diverse scenarios, like indoor house interiors, outdoor mountainous terrain covered with complex vegetation, etc.).

Total input test sets consist of 80 pictures taken on simple scenes (like Cornell radiosity box, office, garage) and 147 screenshots grabbed from different scenarios of COJ game (Western themed town, saloon, jail, forest, farm).

5.2.3. Collecting snapshots

V-Engine application gives the ability to load and visualize any 3D virtual scene prepared in 3DSMAX. The program in its interactive free-camera mode allows to define the list of camera views by simply moving and rotating the camera around the scene and marking samples for off-line rendering. After collecting snapshots we can run the program in special mode which retrieves the camera views and render images utilizing both standard direct lighting and ISR lighting techniques. The whole process is very similar to taking the same photograph with different camera settings.

Additionally, each snapshot generation produces POV-Ray readable script for the reconstruction of exact take using precise ray-tracing method. V-Engine outputs radiosity settings, camera view parameters, point and directional lights and the scene geometry.

5.2.4. POV-Ray rendering

POV-Ray renderer application takes the V-Engine output script and generates the model image using accurate ray-tracing techniques for the radiosity calculations. The example script file looks as follows:

Listing 9: Fragment of a POV-Ray script exported by V-Engine

```
global_settings {
radiosity {
brightness 2.4
pretrace_start 0.08
pretrace_end 0.01
count 250
error_bound 0.25
recursion_limit 2
}
}

light_source {
<2e+006,2e+006,2e+006> rgb 1.3 shadowless parallel point_at <0,0,0>}

camera {
location <1.64971,3.74113,-2.69074>
direction <-0.337043,-0.764329,0.54973>
up <-0.399505,0.644827,0.651609>
right <1.1367,0,0.696916>
look_at <-1.72072,-3.90216,2.80656>
angle 75.1782
}
background { color rgb < 0.0, 0.0, 0.0 > }

object {
// element name: Plane01
mesh2 {
vertex_vectors {
25
<-3,0,-3>,  <-3,0,-1.5>,  <-3,0,0>,  <-3,0,1.5>,  <-3,0,3>,  <-1.5,0,-3>,  <-1.5,0,-1.5>,  <-
1.5,0,0>,  <-1.5,0,1.5>,  <-1.5,0,3>,  <0,0,-3>,  <0,0,-1.5>,  <0,0,0>,  <0,0,1.5>,  <0,0,3>,
<1.5,0,-3>,  <1.5,0,-1.5>,  <1.5,0,0>,  <1.5,0,1.5>,  <1.5,0,3>,  <3,0,-3>,  <3,0,-1.5>,  <3,0,0>,
<3,0,1.5>, <3,0,3>, }
normal_vectors {
25
...
```

The radiosity settings used for POV-Ray rendering have the following meaning:

radiosity parameter	value	notes
brightness	**2.4**	initial brightness factor for the scene
pretrace_start	**0.08**	controls the radiosity pre-trace gathering step
pretrace_end	**0.01**	
count	**250**	number of rays that are sent out whenever a new radiosity value has to be calculated
error_bound	**0.25**	the fraction of error tolerated; a compromise between rendering speed and the final image quality
recursion_limit	**2**	integer value which determines how many recursion levels are used to calculate diffuse inter-reflection; the upper limit is 20

Table 1: POV-Ray radiosity configuration block

The radiosity parameters have a significant impact on the rendering time and have to be chosen very carefully. The *recursion limit* has been set to 2 which means that only ambient occlusion factor and the 1st bounce of indirect lighting should be calculated (ISR method during this experiment falls under the same limitations). *Count* and *error bound* parameters strongly affects the rendering time and the level of indirect lighting approximation and have been chosen experimentally to obtain satisfying results.
All rendering have been performed in 1024x768 pixels resolution with anti-aliasing option disabled.

5.2.5. Output image analysis

The final stage of the experiment is to compare the real-time rendered snapshots according to the model image generated by POV-Ray renderer application. Imalyzer takes two images as an input and outputs calculated mean error as well as the Pearson correlation coefficient. The tool has also the ability to produce difference images which depict the negative of the absolute error between corresponding images.

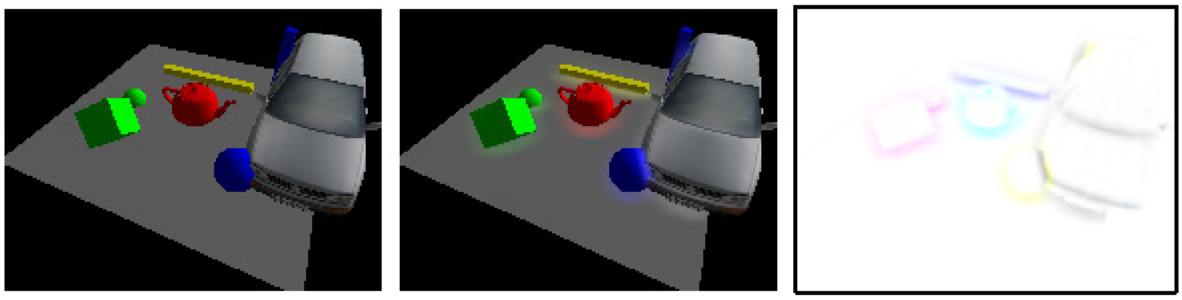


Figure 5.5: Imalyzer input images (a,b) and the output negative difference image (c)

Figure above presents the image generated with direct lighting method (a), the image generated with ISR method (b) and the corresponding negative of the absolute error (c). In this case we have visualized the negative of the isolated ISR factor.

5.3. Results

This section presents the results obtained with ISR algorithm described in chapter 4. Our algorithm has been tested in various, mostly game typical, simple and complex scenes described in section 5.2.2. First we reveal the performance results and then we perform quality and perception analysis.

5.3.1. Performance

All renderings in this dissertation were performed on Intel 2.33 GHz Core 2 Quad and ATI/AMD Radeon HD 4870 graphics board. The output screen resolution is 1024x768 pixels whereas the coarse ISR_4 buffer is 512x384 (4 times smaller). Since our method performs all computations purely on GPU, the CPU has no impact on real-time rendering performance (only POV-Ray off-line renderings were performed on CPU).

ISR algorithm works completely in image space which means it does not depend on the scene complexity therefore it might be used to calculate lighting for simple as well as complex scenes analogically with the same computational workload. Table 2 shows the timing values for the experiment render phase; direct lighting and ISR calculations were done on GPU at real-time framerates (see the average FPS) whereas POV-Ray ray-traced images were calculated on CPU.

	total rendering time [s]	average frame time [s]	average FPS
direct lighting	**1,64**	**0,00722**	**138,41463**
ISR	**2,02**	**0,00889**	**112,48761**
POV-Ray	**168070**	**740,39648**	**0,00135**

Table 2: Performance results for 227 rendered images

ISR rendering time values are slightly different for each image due to the variable scene rendering view (from 200 to 500k triangles per frame) which is beyond the lighting calculation phase. Nevertheless, the minimum FPS is over 90 and the average FPS is 112 (as the table 2 shows) which indicates that ISR meets one of the main requirements – it works in real-time.

The results in table 2 were obtained for 48 samples per pixel in ISR calculations. We have tested the performance impact for a different number of samples per pixel.

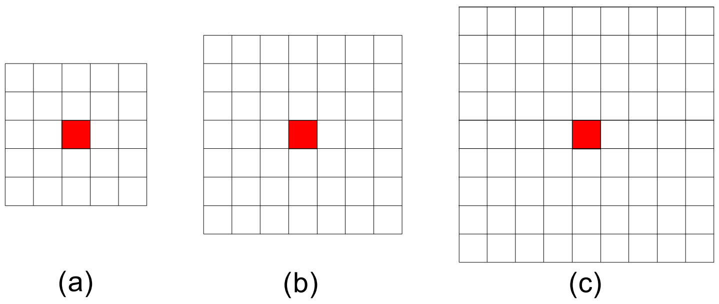


Figure 5.6: Different number of samples per pixel: (a) 24; (b) 48; (c) 80

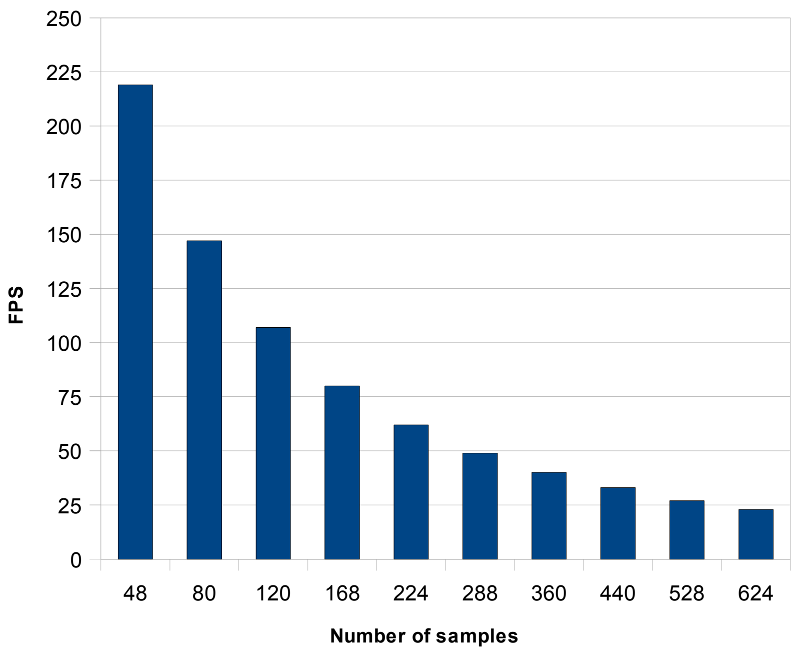


Figure 5.7: Performance costs for the different number of per-pixel samples

Figure 5.7 presents different performance costs for medium complex scene (~20k triangles) for different number of neighborhood samples used in ISR calculations. We have noticed that the quality changes are slightly visible for over than 80 samples per pixel - mostly because of low impact between distant pixels (bigger distance in 2D screen space goes together with bigger distance in 3D world space). All further experiments are established with number of 80 samples per pixel.

ISR calculation time for each image is constant and can be denoted from the table 2 as:

$$t_{ISR} = \frac{T_{ISR} - T_{DL}}{number\ of\ images} = \frac{2{,}02\,s - 1{,}64\,s}{227} = 1{,}674\,ms$$

which does not introduce noticeable performance hit.

The interesting fact is that ISR method works on a single GPU over than 83000 times faster than ray-tracing method on quad-core CPU.

5.3.2. Quality analysis

As mentioned before, we have tested the quality of ISR rendered images by comparison to the model image generated with accurate and time-consuming ray-tracing method. Figure 5.8 presents the mean error obtained using both ISR and DL techniques. The experiments evinced that for all 227 sample images ISR produced smaller error comparing to DL technique. Figure 5.9 shows the percentage quality gain for each individual image (the gain is positive in 100% cases). ISR improves the quality in average of $\mu = 5{,}88\%$ $(\mu_{1/2} = 7{,}18\%)$ which positively exceeds the initial expectations.

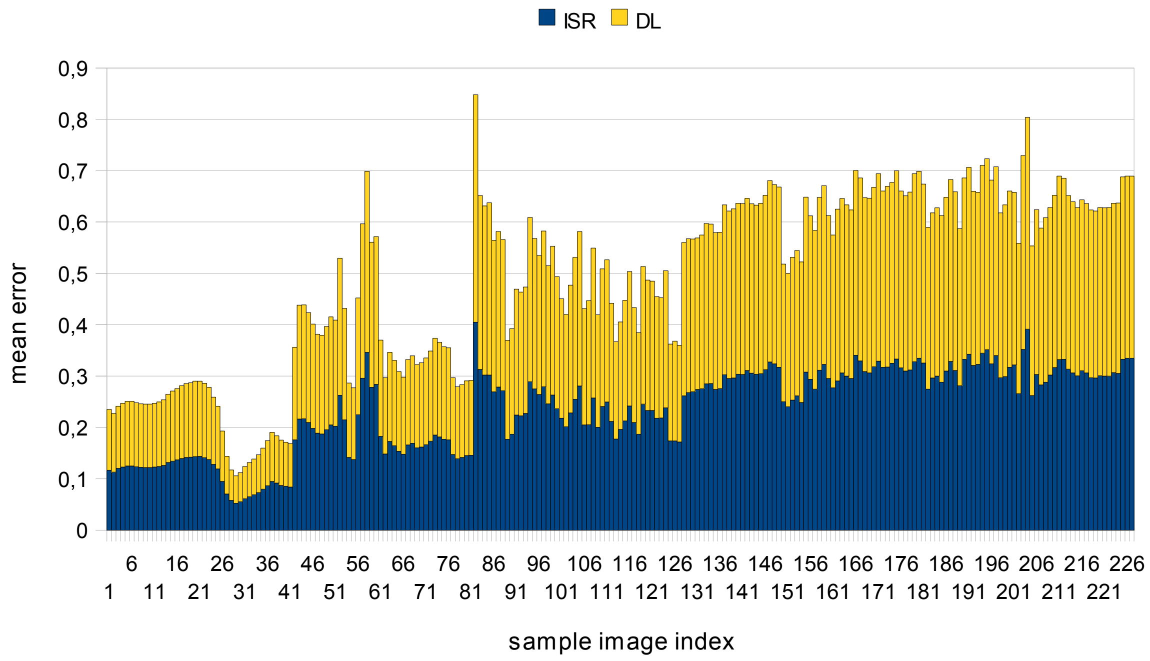


Figure 5.8: Mean error for direct lighting and ISR techniques

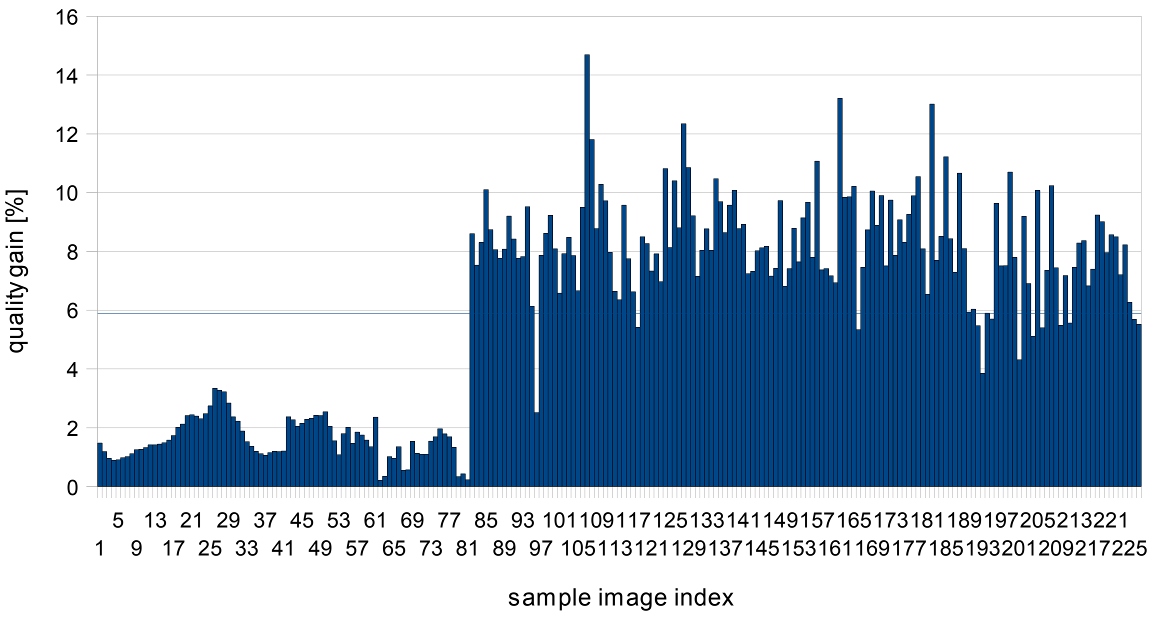


Figure 5.9: Percentage quality gain for each image sample

In our experiments we have also observed that the quality gain is smaller for the simple scenes (first 80 image samples). The reason for this is the smaller interrelation between patches on the simple scenes caused by the bigger polygons on the scene. Quality gain for the simple (in our case up to 25k triangles) is in the rank from 0,21% to 3,34% whereas for the complex scenes (from 25k to 500k) from 2,51% to 14,69%. Figure 5.13 demonstrates the images with the biggest (over 12%) improvement in the quality for complex scenes.

We have also examined the Pearson correlation coefficient to compare linear dependence between DL and ISR techniques. As shown in figure 5.10 there are some cases when the ISR correlation coefficient is worse than the DL correlation coefficient (the values below 0). We have demonstrated the best and the worst cases for the correlation coefficient gain in figures 5.11 and 5.12 respectively. They also show the corresponding images obtained by POV-Ray during ray-tracing phase. The negative correlation gain cases show that the indirect lighting bounce has too much impact and should be decreased. After tweaking the indirect lighting power coefficient k_{ISR} (from 0.5 to 0.1) we were able to achieve positive correlation gain for the specified images.

In overall, without any tweaking, the average Pearson correlation coefficient increased from $\rho=0{,}8867$ to $\rho=0{,}8870$ to ISR method advantage.

Since ISR algorithm operates on individual pixels we have also examined the pixel level efficiency: there were 178 520 064 pixels in 227 images at 1024x768 resolution. 176 824 189 ISR generated pixels were closer to the pixels taken from model images, which means that **99,05%** of all pixel values have been improved.

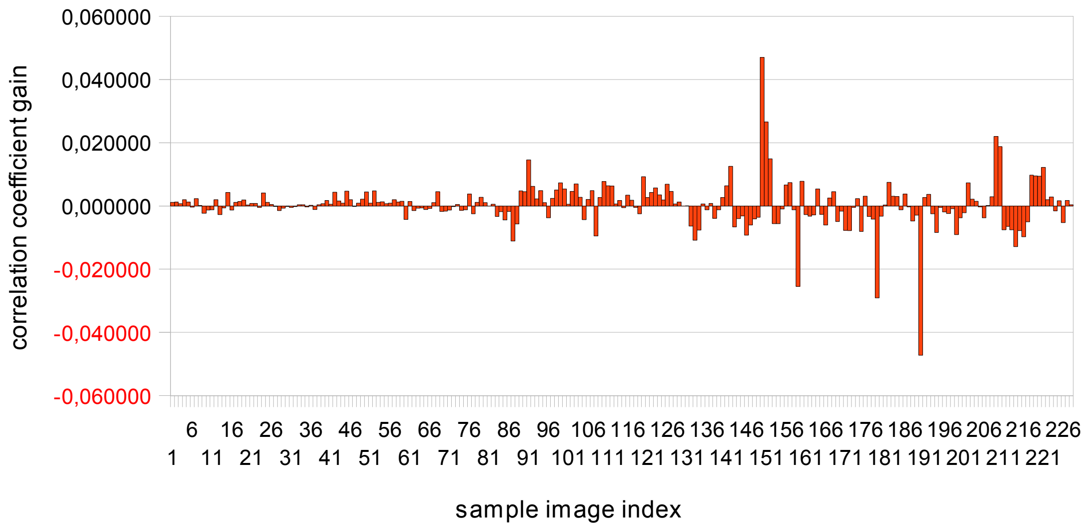


Figure 5.10: Percentage quality gain for each image sample

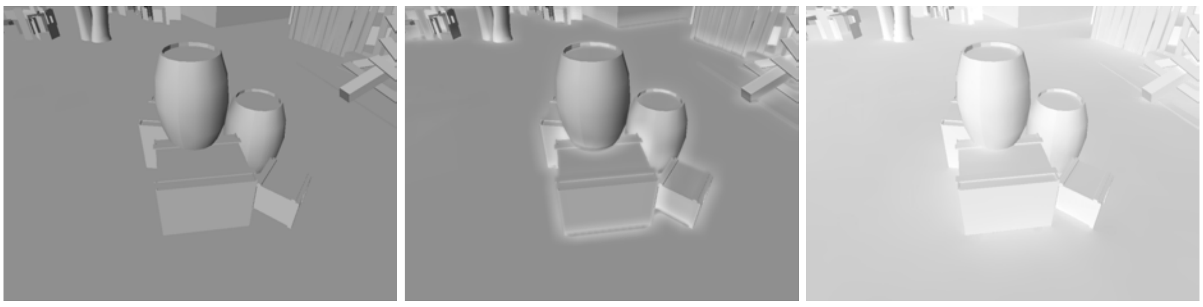


Figure 5.11: Best case (positive) correlation coefficient gain (image 150)

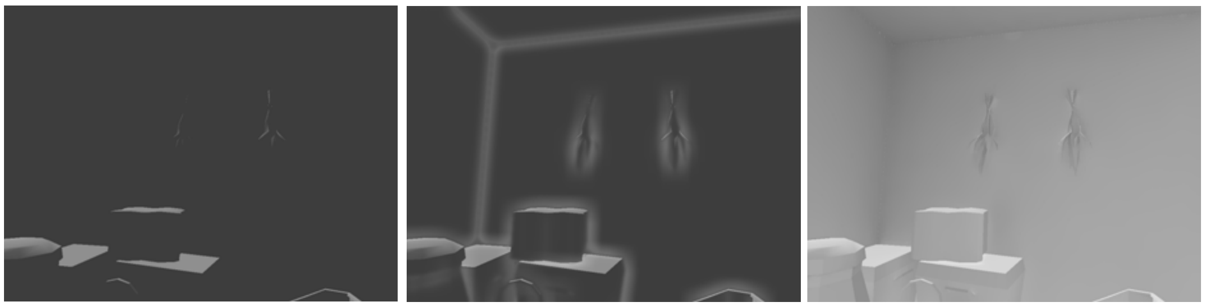


Figure 5.12: Worst case (negative) correlation coefficient (image 190)

5.3.3. Perception aspect

Apart from the mathematical point of view we can observe perceptual quality improvements with ISR method which help in better understanding 3D shape of the scene. Figures 5.13, 5.14 and 5.15 show the difference between flat and uniform surfaces lit with the DL method (left side) and much more credible surfaces lit with the ISR method (right side). We have noticed that ISR algorithm brings out more details mostly due to the lighting transport calculations between adjacent patches. If the local geometry is sufficiently represented by the nearby pixels it receives more indirect illumination and becomes more distinct as shown in the figures below. Ambient occlusion term provides better perceiving of true distances between objects in 3D space which is noticeable especially on the images representing indoor scenes (figures 5.14 and 5.15).

Figure 5.16 shows color bleeding between objects with different material properties. It also demonstrate the ISR technique in cooperation with exponential shadow maps technique described in section 4.6.

5.3.4. Discussion

Since method described in this dissertation works completely in the image space it is not limited to static objects only. ISR does not perform any off-line pre-calculations and computes lighting equations each frame from scratch which means it can handle fully dynamic scenes with object transformation, morphing, skinning, skeletal animation, or any other mesh geometry deformation.

ISR technique has also been tested on equivalent nVidia GeForce GTX 260 graphics board and the results were very similar in terms of performance and quality.

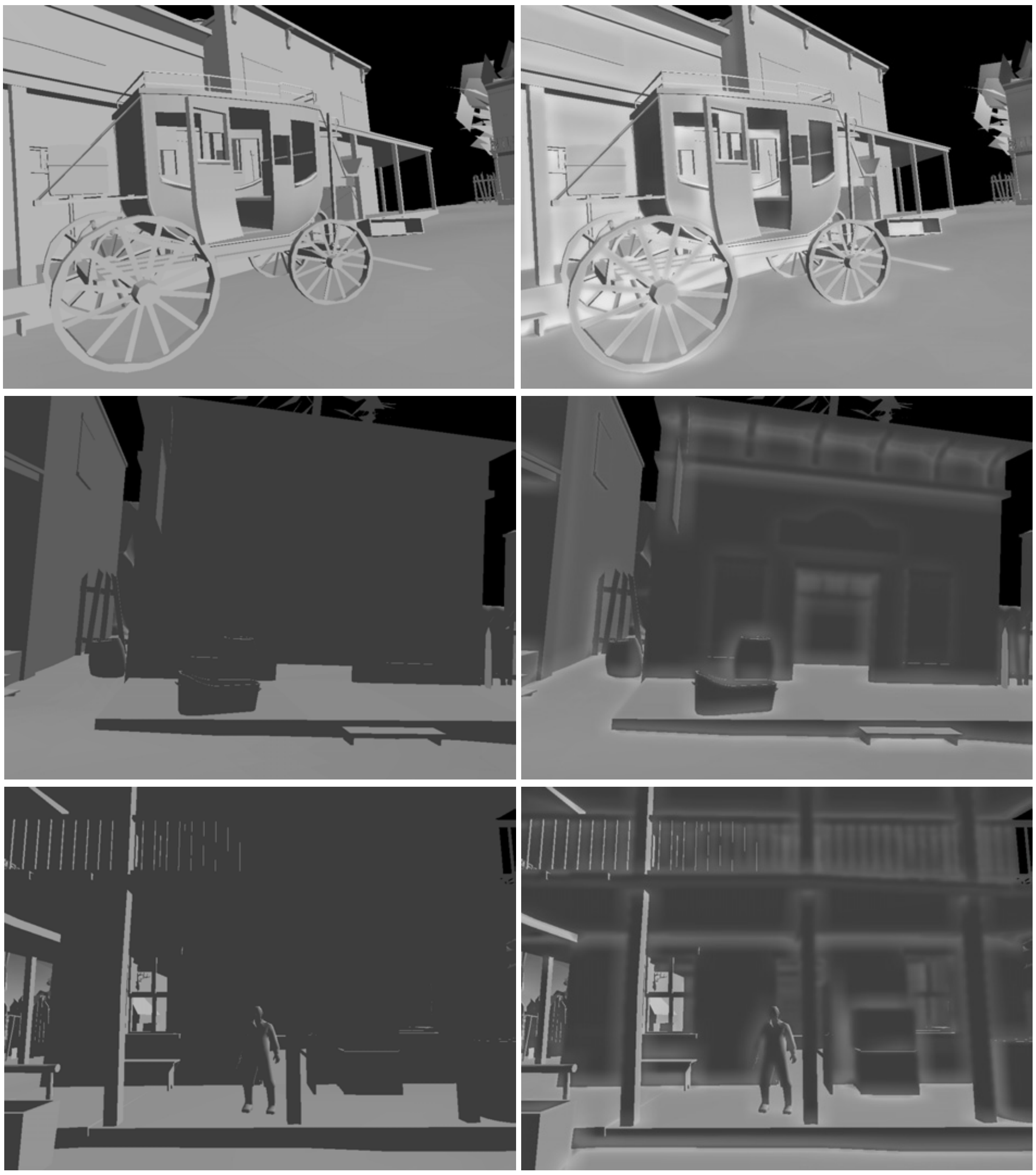


Figure 5.13: Comparison between direct lighting (left) and ISR (right) techniques for open-space environment scenes

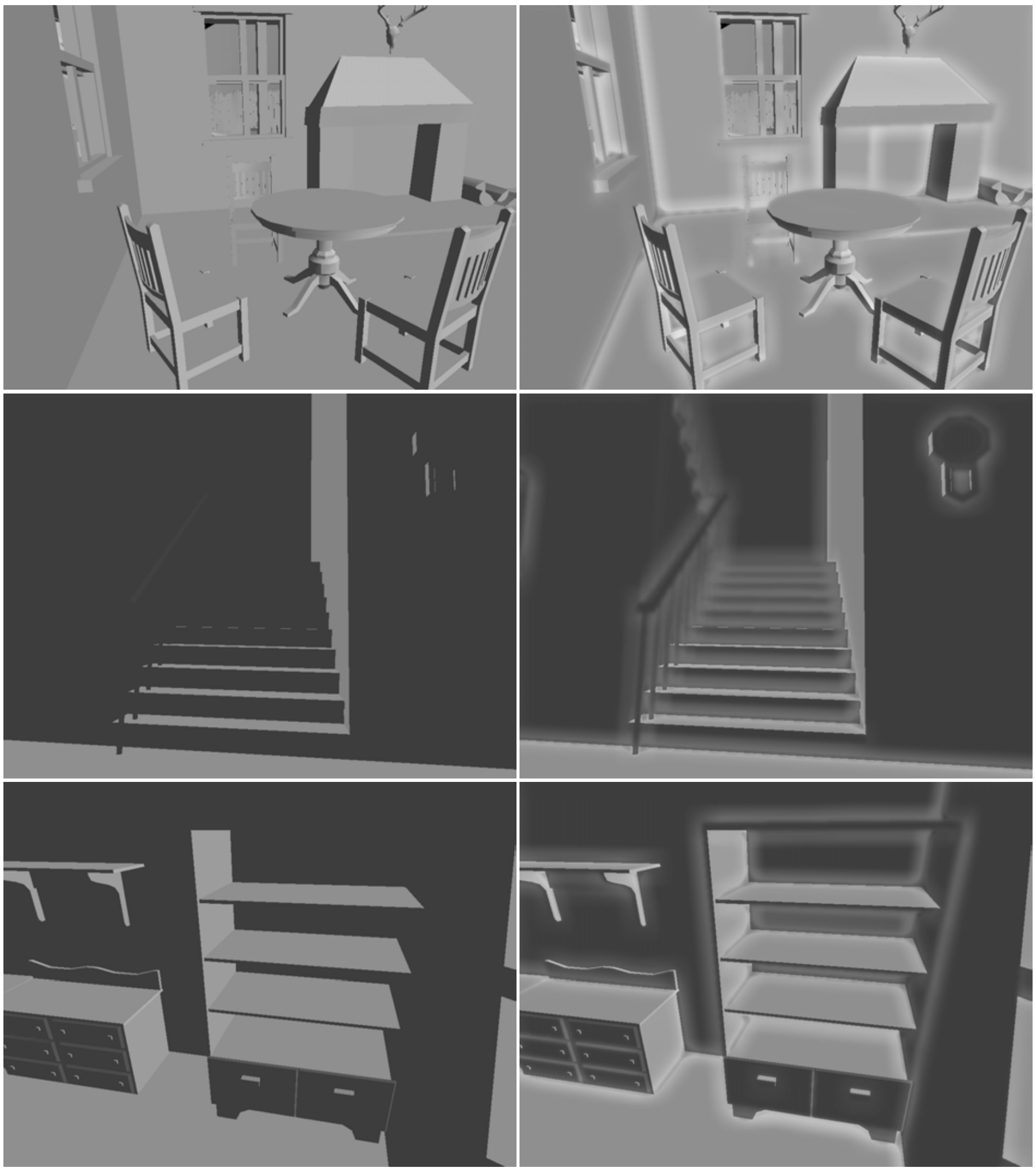


Figure 5.14: Comparison between direct lighting (left) and ISR (right) techniques for indoor scenes (room)

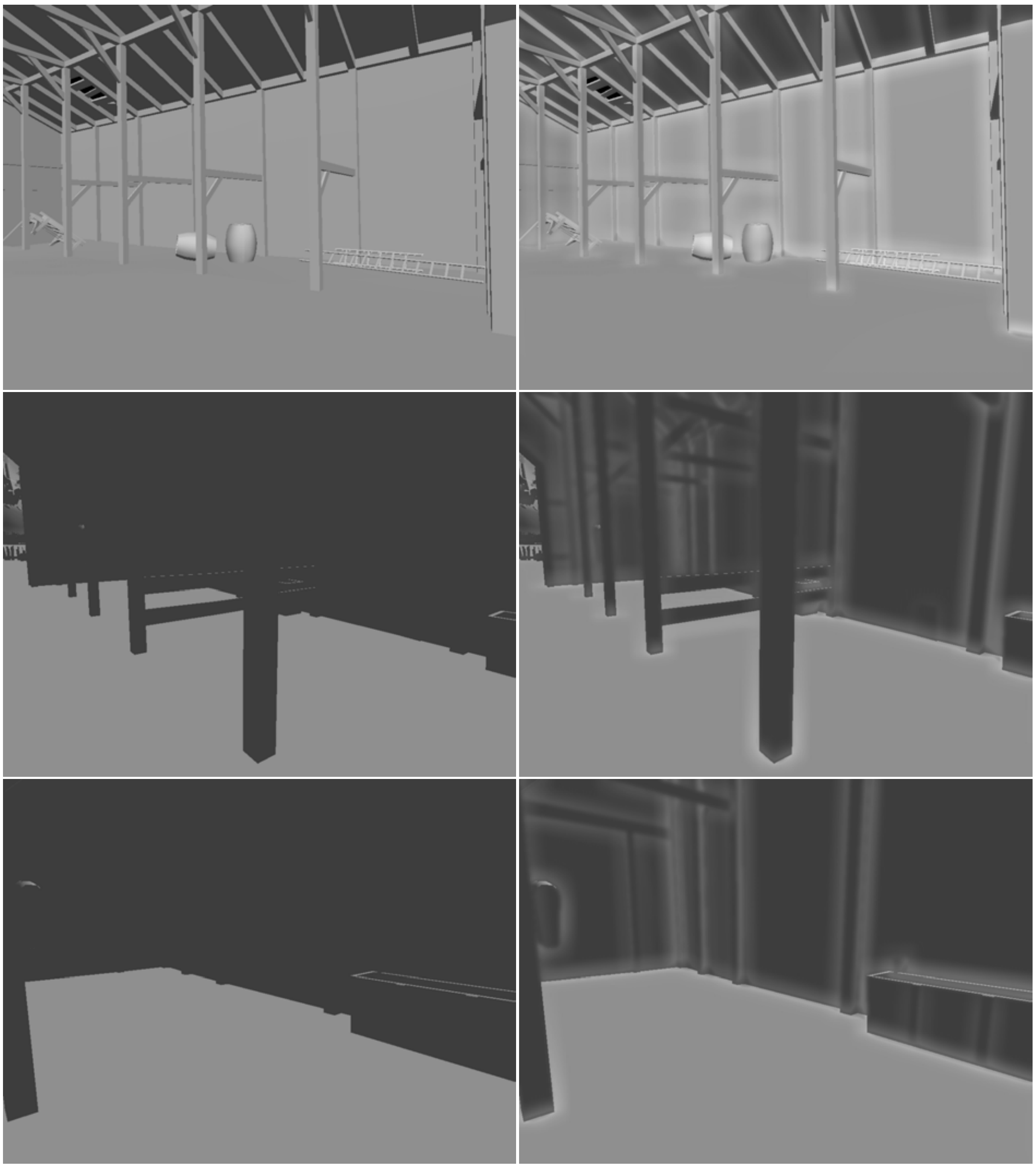


Figure 5.15: Comparison between direct lighting (left) and ISR (right) techniques for indoor scenes (barn)

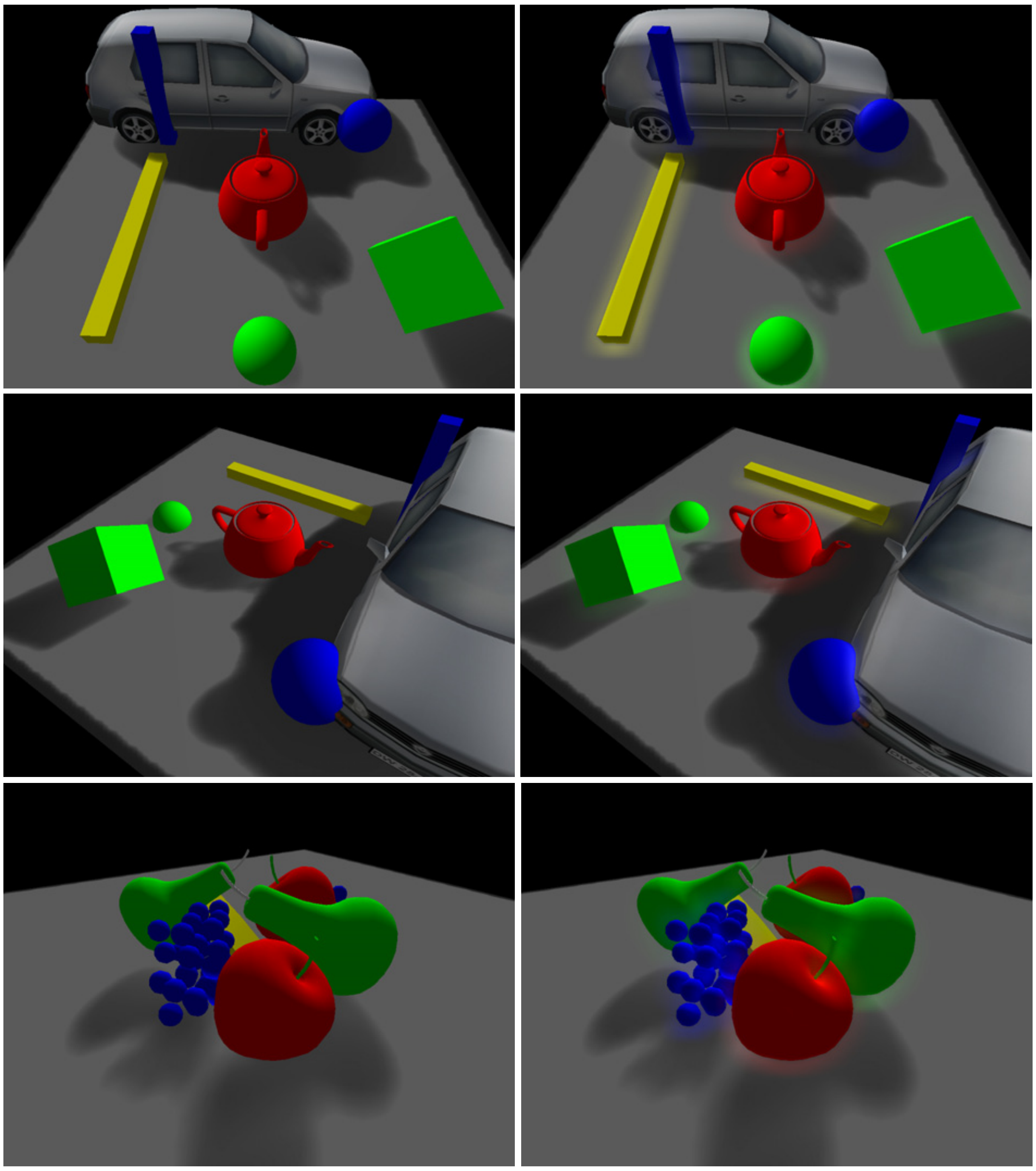


Figure 5.16: Comparison between direct lighting (left) and ISR (right) techniques for simple scenes with soft shadows

6. Conclusions

This chapter contains the final summary of the dissertation along with the conclusions. We also present the ideas of possible extensions and future work.

The novel image-space radiosity (ISR) algorithm along with the implementation details and comprehensive evaluation has been examined. We have shown that ISR algorithm models light transport more accurately than standard direct lighting techniques at real-time frame rates. ISR computes the local radiosity taking interrelations between nearby pixels under consideration which is used to imitate ambient occlusion and global illumination distribution.

Let's have a look at the pros and cons of the proposed solution:

Pros:

- **algorithm is independent on the scene complexity;** the most vital advantage of ISR algorithm is that it works the same for the simple scenes with only a few objects (like the Cornell box) and for the complex dynamic outdoor scenes with millions of visible polygons. The algorithm operates on screen-space and does not depend on the number of objects that need to be illuminated.
- **algorithm works on visible pixels only;** standard forward rendering performs lighting calculations for every primitives sent to the GPU which can be very inefficient, especially when the overdraw rate is high. ISR utilizes deferred rendering and computes each pixel illumination factor **only once** which is the most efficient way of lighting calculations;
- **proven to be fast and efficient for complex and dynamic scenes (as well indoor as outdoor scenes);** as described in the previous sections the algorithm works fast for both simple indoor and complex outdoor scenes (does not depend on how the scene is

being modeled).

- **requires no content generation pipeline changes (done completely in real-time);** unlike the SH-based and lightmap-based lighting models ISR computes the lighting transport completely in real-time and does not require any pre-calculated data which means that the content generation pipeline remains unchanged;
- **easily implementable in any existing rendering pipeline (especially when using deferred rendering technique)**; the algorithm works completely as a post-processing effect and putting it into the existing rendering pipeline is very straightforward. Majority of graphics systems and engines perform post-processing phase as a separate step defined in external script files which means that implementing ISR can be done without touching the line of code;
- **very well scalable into the upcoming hardware and software technology (compute shaders in Direct X 11);** compute shader is being added to the upcoming Direct X 11 API to take advantage of the massive parallel power in today's GPUs; advanced post-processing techniques like ISR or SSAO can be efficiently enriched and extended in the upcoming graphics software and hardware;
- **more accurate approximation of rendering equation than direct lighting method;** ISR light model computes light emitted and reflected from the nearby geometry (indirect lighting), not only the direct illumination.
- **requires no additional memory, nor CPU cost;** The algorithm is being calculated completely on the GPU. It requires additional buffers (render targets) which can be reused for other post-processing effects, like motion blur, depth of field, fog, etc

Cons:

- **works in screen-space (quality artifacts similar when using SSAO technique);** ISR technique introduces artifacts visible mostly on object silhouette edges (it is hard to correctly blur out the noise without interfering with depth discontinuities). The bilateral upsampling has been applied to overcome this issue, however it does not solve to problem completely (there is a noticeable halo effect around the objects).

- **view dependency and calculations the light transport between visible pixels only;** in our model only pixels which are visible emits and reflects the indirect illumination in the scene. It can cause strange behavior, like no indirect lighting in the certain conditions (i. e. the view direction is parallel to the primitive plane which means there are very few pixels visible causing inaccurate indirect lighting).
- **works locally – only the nearby pixels influence the corresponding one;** for the sake of performance we are limited to use the small number of surrounding samples to calculate the light transport. To overcome this issue we propose the solution with variable number of samples based on the corresponding pixel depth (pixels which are close to the camera needs to be handled more accurately than the far ones). It is also possible to experiment with different filter kernels when sampling the neighboring pixels. Additionally, we can extend the sample quantity on the newer graphics hardware as we noticed very good performance there (~200 FPS).
- **calculations must be performed on coarse, diminished buffers;** for the sake of higher performance the lighting transport calculations are being made on the diminished buffers which needs to be finally upsampled into the native resolution. Upsampling may cause some artifacts on the silhouette edges (bilateral upsampling step is performed to overcome this issue). As mentioned before, the newer hardware handle better post-processing operations in the term of speed and it is possible to perform ISR calculations on full native resolutions.

We present a few ideas and improvements to overcome the itemized disadvantages:

- **depth peeling;** depth peeling [84] can be used to extract a number of layers which representing the front-most pixels that are peeled away after each pass. The main idea is the use of two depth tests for each pair of layers to determine to current front-most layer and to exclude previously peeled layers. Afterwards, we can calculate lighting transport via ISR for each layer and combine the results at the final stage which will bring us more information about the patches emitting and reflecting the light on the scene. The main drawback of this approach is the fact is requires a pass per layer which can lead to the performance issues (especially when complex objects are being rendered);

- **wider FOV for better indirect lighting on border pixels;** we can also extend the camera field of view to grab more pixels at the screen border. This approach brings the more accurate lighting calculations on the screen borders, however, it requires rendering the scene in higher resolution (some kind of super-sampling [39]) which affects the performance and consumes more video memory;
- **additional cameras;** different camera views may be used to handle occluded regions of the scene. As proposed in [62] different camera positions can be useful for polygons which are viewed at an acute angle. For the sake of performance, lower resolution buffers may be sufficient for handling color bleeding from occluded regions.
- **increase number of light bounces**; the ISR algorithm is not limited to calculate only the ambient occlusion factor and one bounce of indirect illumination. We can very easily extend our method by successive iterating the lighting transport calculations step to obtain multiple bounces of indirect illumination.

To summarize, the final product of this dissertation is an extensively evaluated real-time radiosity algorithm which can be easily implemented into any 3D real-time computer graphics application. ISR can be applicable in computer games, flight simulators, movie industry, virtual building visualizations, CAD systems, GUI systems, medical diagnostics, multimedia systems, or measurement data visualizations.

A. List of abbreviations

AO	Ambient Occlusion
BF	Bilinear Filtering
BRDF	Bidirectional Radiance Distribution Function
BSP	Binary space partitioning
CAD	Computer Aided Design
CPU	Central Processor Unit
CSM	Convolution Shadow Maps
DL	Direct Lighting
ESM	Exponential Shadow Maps
FOV	Field of view
FPP	First Person Perspective
FPS	Frames per second
GI	Global Illumination
GPU	Graphics Processor Unit
GUI	Graphical User Interface
HLSL	High-Level Shader Language
ISR	Image-space radiosity
MRT	Multiple Render Targets
PCF	Percentage Closer Filtering
PMCC	Pearson product-moment correlation coefficient
PNG	Portable Network Graphics
PRM	Precomputed radiance maps
PRT	Precomputed radiance transfer
PS	Pixel Shader
RT	Ray tracing
SH	Spherical Harmonics
SSAO	Screen-space ambient occlusion
UML	Unified Modeling Language
VPL	Virtual Point Light
VR	Virtual Reality
VS	Vertex Shader
VSM	Variance Shadow Maps

B. Notations

Symbols used throughout this dissertation:

M_w	*world matrix*
M_v	*view matrix*
M_p	*projection matrix*
M_c	*combined matrix (compound of world, view and projection matrices)*
I_{POV}	*model image*
I_{ISR}	*ISR image*
I_{DL}	*Image rendered with standard direct lighting technique*
D	*difference image*
f_q	*quality function*
k_a	*ambient light power factor*
L_a	*ambient light color*
k_d	*diffuse light power factor*
L_d	*diffuse light color*
k_s	*specular light power factor*
L_s	*specular light color*
$saturate(x)$	*function which clamps value x into range [0..1]*
$lum(p)$	*function which calculates the luminance of pixel p*
μ_X	*population mean*
$\mu_{1/2X}$	*Median of variable X*
σ_X	*population standard deviation*
$\mathbb{N}$	*natural number set*
$\mathbb{R}$	*real number set*
$V(x, \vec{\omega})$	*visibility function from x along direction* $\vec{\omega}$
Q	*radiant energy*

Φ	*radiant flux*
E	*irradiance*
M	*radiant exitance*
F_{ij}	*form factor between patches i and j*
A_i	*Area of patch i*
ρ_i	*reflectance of patch i*
λ	*Wavelength of light*

We also explain UML [27][56] symbols used in UML diagrams:

name : type	*name and type of attribute (i.e. string, float, 3D vector, or 4x4 matrix)*
+Method()	*public class method*
IClassName	*classes begin with „I" stands for interfaces*
VClassName	*classes begin with „V" stands for class implementations*
◆—	*class aggregation*
▲	*class inheritance*

Table 3: UML symbols and definitions

C. Derivation of radiosity equation

The rendering equation is the fundamental transport equation which describes the light transport in a three-dimensional scene. It is a recursive integral equation which is in practice very difficult to solve analytically (numerical techniques need to be used). This supplement reveals the theory behind the radiosity equation derived from rendering equation utilized in the dissertation entitled “*Image-space radiosity lighting method for dynamic and complex virtual environments*”.

The light transport in environments exhibiting general light emission and scattering introduced by Kajiya [45] is given by:

$$L_0(x,\Theta)=L_e(x,\Theta)+\int_{\Omega_x} f_r(x,\Theta'\Leftrightarrow\Theta)L_i(x,\Theta')\cos\theta' d\omega_{\Theta'} \tag{31}$$

where:

- L_0 - total amount of illumination (exitant radiance) at x into direction Θ
- L_e - self-emitted light (self-emitted radiance) at x into direction Θ
- L_i - incoming light (incident radiance) at x from direction Θ
- $f_r(x,\Theta'\Leftrightarrow\Theta)$ - bidirectional reflectance distribution function (BRDF) at x for scattering from a direction Θ into direction Θ' (or vice versa)
- x - specified location on a surface
- Θ - outgoing direction at location x
- Θ' - incoming direction over the hemisphere around x
- θ' - angle between Θ' and surface normal at x
- $d\omega_{\Theta}'$ - infinitesimal solid angle containing direction Θ'
- Ω_x - hemisphere of directions above x

Radiance is the amount of radiant flux per unit projected area per unit solid angle, measured in Watt per square meter per steradian:

$$L=\frac{d^2\Phi}{d\omega dA^{\perp}}=\frac{d^2\Phi}{d\omega dA\cos\theta} \tag{32}$$

where:

- Φ - radiant power (flux) measured in Watt
- $dA^{\perp}$ - unit area projected perpendicular to the direction of normal incident angle

The average radiosity B_i (measured in Watt per square meter) emitted by a patch i in such an environment is defined by [24] as the following:

$$B_i = \frac{1}{A_i} \int_{S_i} \int_{\Omega_x} L_0(x, \Theta) \cos\theta \, d\,\omega_{\Theta} \, dA_X \tag{33}$$

where:

- A_i - surface area of patch i
- dA_x - differential area at a point x
- S_i - surface of patch i (set of points)

The rendering equation (31) simplifies in purely diffuse environment (self-emitted radiance BRDF do not depend on directions Θ and Θ') into the form:

$$L_0(x) = L_e(x) + \int_{\Omega_x} f_r(x) \, L_i(x, \Theta') \cos\theta' \, d\,\omega_{\Theta'} \tag{34}$$

Diffuse surfaces reflects light in a uniform way over the entire reflecting hemisphere which means that BRDF is constant for all directions Θ . For such a pure Lambertian surfaces we denote BRDF as:

$$f_r(x) = \frac{\rho(x)}{\pi} \tag{35}$$

where reflectance $0 \leqslant \rho(x) \leqslant 1$ represents the fraction of incident energy that is reflected at the surface.

Incident radiance $L_i(x, \Theta')$ corresponds to the exitant radiance $L_0(y)$ emitted by the point y visible from x along the direction Θ' , so changing the integration (34) over the hemisphere Ω_x into integration over all surfaces S in the scene yielding an integral equation in which no directions appear:

$$L_0(x)=L_e(x)+\rho(x)\int_S G(x,y)\,L_0(y)\,dA_y \tag{36}$$

where:

- $\rho(x)$ - reflectivity at point x
- $G(x,y)$ - geometric radiosity kernel

The geometric radiosity kernel:

$$G(x,y)=\frac{\cos\theta_x\cos\theta_y}{\pi r_{xy}^2}\,vis(x,y) \tag{37}$$

which stands for the interrelation between x and y with additional visibility function between these locations (the *vis* function returns 1 if the point x “sees” the point y and 0 otherwise).

Interrelation between the radiant exitance B and the radiance L for flux leaving Lambertian surface is explained in [7] (pp. 26-27) and can be expressed as:

$$B=\pi L \tag{38}$$

Multiplication with π both sides of the equation (35) leads to:

$$B(x)=B_0(x)+\rho(x)\int_S G(x,y)\,B(y)\,dA_y \tag{39}$$

where B and B_0 stands for radiant exitance and self-emitted radiosity respectively.

Now, the equation (33) simplifies into:

$$B_i = \frac{1}{A_i} \int_{S_i} B(x)\, dA_X \qquad (40)$$

To solve integral equations like (32), the Galerkin method [21][24] might be used which yields a continuous operator problem into a discrete form. As explained in [24], we can project the both sides of (32) onto a set of basis functions and equate the resulting coefficients. We approximate $B(x) \approx B'(x) = \sum_i B_i' \psi_i(x)$ with a constant basis function $\psi_i(x) = \begin{cases} 1, x \in S \\ 0, x \notin S \end{cases}$ for each patch i which drives into the classical radiosity system of linear equations:

$$B_i' = B_{0\mathrm{i}} + \rho_i \sum_j F_{ij} B_j' \qquad (41)$$

where F_{ij} stands for the patch-to-patch form factors defined as:

$$F_{ij} = \frac{1}{A_i} \int_{S_i} \int_{S_j} G(x, y)\, dA_y\, dA_x \qquad (42)$$

The above equation must typically be solved using numerical methods (there are no practical analytic solutions for this equation).

D. V-Engine-based simulator

V-Engine development kit is an object-oriented graphics rendering engine written in C/C++ for real-time computer graphics applications. ISR software extensively uses V-Engine to provide simulation environment for image-space radiosity algorithm evaluation and testing. Attached CD-ROM disc contains binaries and full source code of V-Engine utilized during experiments with ISR algorithm.
The full list with V-Engine features has been included in the dissertation on pages 58-59.

Directory structure

ISR application consists of the following modules:

- **Isr/** - main application folder
  - **bin/** - contains binary (executable) files
    - **data/** - contains data files (models, material scripts, shaders, post-processing scripts, textures)
    - **exp/** - contains ISR experiments result files (**final** folder contains all result images)
  - **scripts/** - contains automatic build scripts (cleaning temporary files, batching POV-Ray scripts, etc.)
  - **SDK/** - contains 3rd party middle-ware libraries utilized by V-Engine
  - **simple_app/** - ISR test application source code
  - **vengine/** - V-Engine library source code
    - **inc** – library interface (C++ header files)
    - **src** – library implementation (C++ source code)

Run **Isr/bin/start.cmd** to start ISR test application.

Controls

ISR test application loads the default 3D scene (with car and simple primitives) and gives the possibility to move the camera around the scene arbitrarily:

- **W, S, A, D** – camera movement (forward, backward, left side, right side)
- moving the mouse – camera rotation (changing *yaw* and *pitch angles*)

The default display mode is the ISR lighting mode with ESM shadows enabled. Use the following keys to change the display mode:

- **F2** – default mode (ISR with ESM enabled)
- **F3** – ISR only (without any shadows)
- **F4** – ISR disabled (direct lighting with ESM shadows)
- **F5** – ISR disabled (direct lighting without ESM shadows)

Additional post-processing effects:

- **F6** – „bloom" effect
- **F7** – „depth-of-field" effect
- **F8** – „glass" effect
- **F9** – „radial_blur" effect
- **F11** – „sepia" effect
- **F12** – „tiles" effect

Source code remarks

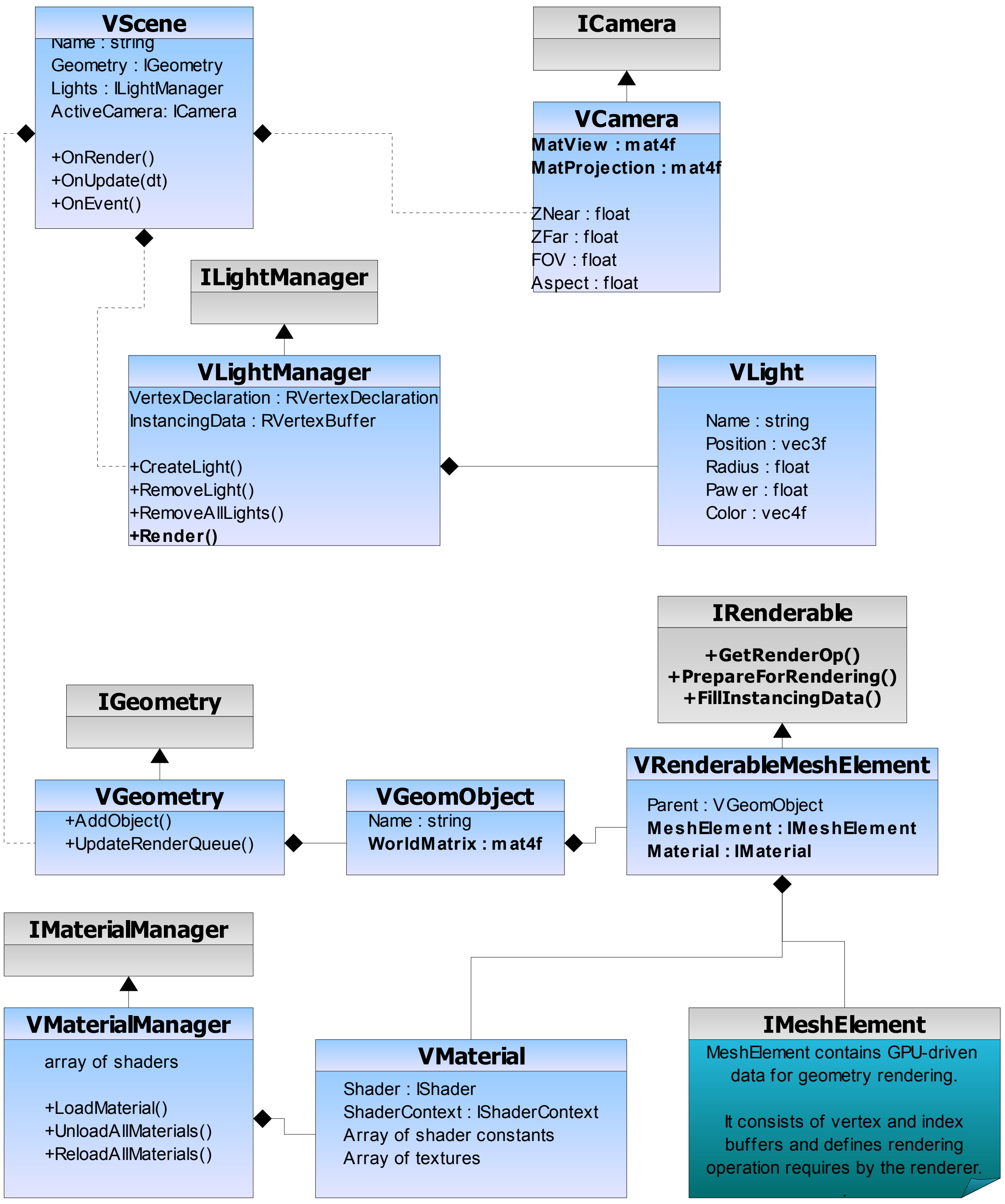
CD-ROM disc contains full ISR application and V-Engine library source code. The most vital source code snippets are:

- **Isr/simple_app/main.cpp** – test application source file; it handles camera controller, scene management, rendering loop (post-processing effects) defined in Lua script file

(***Isr/bin/pp.lua***);

- **Isr/bin/pp.lua** – post-processing effects script file; it defines how the single frame should be rendered for different rendering modes (with ESM shadows enabled/disabled, etc.); it contains two main procedures:
    - **OnCreate()** - called once at startup (creates required resources, like render targets);
    - **OnRender(tech)** – called every frame during the application run (tech argument stores current rendering technique); when ISR is enabled it calls RenderWithISR procedure which realizes ISR algorithm presented in the dissertation; shaders for ISR technique are being stored in ***Isr/bin/data/pp.fx*** *file;*
- **Isr/bin/data/pp.fx** – GPU shaders library; the most important shaders are:
    - ***QuadISR*** – calculate indirect illumination and store into the temporary buffers (ISR step, chapter 4.4);
    - ***QuadLighting*** – full-screen calculations with indirect illumination (lighting phase, chapter 4.5);
    - **QuadDirectLighting** – full-screen direct illumination with MRT buffers utilization

All the most vital source code listings have been extensively commented and described in the dissertation.



UML diagram with the most vital V-Engine components

E. Perl test script

```
# POV-Ray executable location
$POV_APP="D:\\Program Files\\POV-Ray for Windows v3.7\\bin\\pvengine.exe";
# POV-Ray command line parameters
$POV_PARAMS="-d -w1024 -h768 /exit";
# Location of scripts (directory)
$POV_SCRIPTS="exp\\out";
# Input script
$ISR_SCRIPT="exp\\isr.script";

# Imalizer executable location
$IMALIZER="Imalizer.exe";

$t_gather = 0;
$t_pov = 0;
$t_imalizer = 0;

if (@ARGV < 1)
{
  $t_gather = 1;
  $t_pov = 1;
  $t_imalizer = 1;
}
else
{
  for ($i=0; $i<=$#ARGV; $i++)
  {
    if ($ARGV[$i] =~ /gather/) { $t_gather = 1; }
    if ($ARGV[$i] =~ /pov/) { $t_pov = 1; }
    if ($ARGV[$i] =~ /imalizer/) { $t_imalizer = 1; }
  }
}

print "gather = $t_gather\n";
print "pov = $t_pov\n";
print "imalizer = $t_imalizer\n";


# Store start time
sub PerfStart
{
  $time_0 = time;
}

# Calculate time duration
sub PerfEnd
{
  $time = time - $time_0;
  print "Done in $time seconds\n\n";
}

# Capture ISR images and gather POVRay scripts
sub GatherData
{
  PerfStart();
  system("simple_app.exe script=$ISR_SCRIPT job=1");
  PerfEnd();
}
```

```
# Render POVRay scripts
sub RenderPOV
{
  PerfStart();
  # get scripts from the specified directory
  @scripts=`dir $POV_SCRIPTS\\*.pov /A:A /B /S`;

  # get number of pov scripts
  $qty = scalar(@scripts);
  print "Found $qty POV-Ray scripts\n";

  # script counter
  $idx = 1;

  # loop through all scripts
  foreach $file (@scripts)
  {
     print "Processing $idx of $qty...\n";
     system("\"$POV_APP\" $file $POV_PARAMS");
     $idx++;
  }

  PerfEnd();
}

# Convert POVRay generated BMPs into PNGs and remove BMPs
sub CleanupPOV
{
  PerfStart();

  printf "Converting from BMP->PNG...\n";
  system("bmp2png.exe -E exp\\out\\*.bmp");

  printf "Removing trash files...\n";
  system("del /Q exp\\out\\*.bak");

  PerfEnd();
}

# Imalizer comparisions
sub ImalizerTest
{
  PerfStart();

  open FILE, ">res.txt";

  # Get image list with names matching isr*01.png
  @pngs=`dir $POV_SCRIPTS\\isr_*_01.png /A:A /B /S`;

  # Image counter
  $f_counter=0;
  # Correlation 0 (mean)
  $mc0=0;
  # Correlation 1 (mean)
  $mc1=0;

  $mm0=0;
  $mm1=0;

  # for each image
  foreach $file1 (@pngs)
  {
    # Get isr*02.png and isr*03.png corresponding files
    $base = $file1;
    $file2 = $file1;
    $file3 = $file1;
    $base =~ s/(.*_)(.*)(_01)(\.png.*\n)/\2/i;
    $file2 =~ s/(.*)(_01)(\.png)/\1_02\3/i;
    $file3 =~ s/(.*)(_01)(\.png)/\1_03\3/i;

    # Launch Imalizer test (store results in test0, test1)
    $test0=`$IMALIZER compare $file1 $file3`;
    $test1=`$IMALIZER compare $file2 $file3`;

    # Parse correlation and mean error values
    $c0=$test0; if ($c0 =~ m/(.*)(correlation = )(.*)/) { $c0=$3; }
    $m0=$test0; if ($m0 =~ m/(.*)(mean_error = )(.*)/) { $m0=$3; }

    $c1=$test1; if ($c1 =~ m/(.*)(correlation = )(.*)/) { $c1=$3; }
    $m1=$test1; if ($m1 =~ m/(.*)(mean_error = )(.*)/) { $m1=$3; }

    # Print data

    $result ="$base $c0 $m0\t$c1 $m1";

    if ($c1>$c0)
    { $result .= "\tc:OK"; }
    else
    { $result .= "\tc:failed"; }

    if ($m1<$m0)
    { $result .= "\tm:OK\n";}
    else
    { $result .= "\tm:failed\n"; }

    print $result;
    print FILE $result;

    $mc0 += $c0;
    $mc1 += $c1;
    $mm0 += $m0;
    $mm1 += $m1;
    $f_counter++;
  }

  $mc0 /= $f_counter;
  $mc1 /= $f_counter;
  $mm0 /= $f_counter;
  $mm1 /= $f_counter;

  print "$mc0 vs. $mc1\n";
  print "$mm0 vs. $mm1\n";

  close FILE;

  PerfEnd();
}


#--------------------
# Main test procedure
#--------------------
if ($t_gather==1) { GatherData(); }
if ($t_pov == 1) { RenderPOV(); CleanupPOV(); }
if ($t_imalizer == 1) { ImalizerTest(); }
```

F. Glossary

This chapter presents Polish translations of the most important terms used throughout the dissertation:

albedo	*odbicie promieniowania przez powierzchnię*
ambient lighting	*oświetlenie otoczenia*
ambient occlusion	*blokowanie światła otoczenia*
bilinear filtering	*filtrowanie dwuliniowe*
color bleeding	*rozpływanie się kolorów*
deferred shading	*cieniowanie odroczone*
depth peeling	*zdzieranie głębokości*
depth-of-field effect	*efekt głębi obrazu*
diffuse lighting	*oświetlenie rozproszone*
edge anti-aliasing effect	*efekt niwelowania aliasingu na krawędziach*
emissive light	*światło emitowane*
exponential shadow maps	*wykładnicze mapy cieni*
global illumination	*oświetlenie globalne*
image-space radiosity	*metoda energetyczna w przestrzeni obrazu*
light attenuation function	*funkcja osłabiania światła*
light maps	*mapy światła*
motion blur effect	*efekt rozmycia w ruchu*
octree	*drzewo ósemkowe*
pixel shader	*program cieniujący piksele*
post-processing effect	*efekt montażowy*
radiance	*współczynnik promienności*
radiosity	*metoda energetyczna*
rendering equation	*równanie renderingu*
screen-space ambient occlusion	*okluzja otaczająca w przestrzeni obrazu*
specular lighting	*oświetlenie zwierciadlane*
stencil buffer	*bufor szablonu / bufor maskujący*
surface patches	*płaty powierzchni*
vertex shader	*program cieniujący wierzchołki*

Bibliography

[1] ANGEL E., Interactive Computer Graphics: A top-down approach using OpenGL, 3rd edition, Addison-Wesley, 2003;

[2] ANNEN T., et all., Exponential Shadow Maps, Proceedings of Graphics Interface, 2008, available on-line at: http://www.mpi-inf.mpg.de/~tannen/papers/gi_08_esm.pdf

[3] ANNEN T., KAUTZ J., DURAND F., SEIDEL H-P., Spherical harmonics gradients for mid-range illumination, Eurographics Symphosium on Rendering, 2004;

[4] APEL A., Some techniques for machine rendering of solids, AFIPS conference proceedings 32, pp. 37-45, 1968;

[5] ARVO J., GPU Programming Gems 2, Program of Computer Graphics, Cornell University, Ithaca, New York, ISBN 0-12-064481-9;

[6] ASHDOWN I., Photometry and Radiometry, Heart Consultant Ltd., October 2002;

[7] ASHDOWN I., Radiosity: A programmer's perspective, Heart Consultants Ltd., 2002;

[8] AUTODESK, 3ds Max, Modelling and Rendering tool, available on-line at http://www.autodesk.com/3dsmax

[9] BAVOIL L., SAINZ M., Image-space horizon-based ambient occlusion, ShaderX^7 Programming Book, 2009;

[10] BERENGUIER L., Real-time Radiosity on GPU, available on-line;

[11] BOULANGER K., Real-time realistic rendering of nature scenes with dynamic lighting, PhD. thesis, 2008;

[12] BRATEL S., Deferred rendered radiosity method from first person perspective, Master

thesis in computer graphics, Goteborg, Sweden, 2007;

[13] BUNNELL M., Dynamic Ambient Occlusion and Indirect Lighting, GPU Gems, 2005;

[14] CALL OF JUAREZ, Call of Juarez western shooting game developed by Techland, http://www.coj-game.com/, April 2009;

[15] CHEN S., GORDON D., Front-to-back display of BSP trees, IEEE Computer Graphics & Algorithms, 1991, pp 79-85;

[16] CHRISTENSEN P., BATALI D., An irradiance atlas for global illumination in complex production scenes, Eurographics Symphosium on Rendering, 2004;

[17] COHEN M., WALLACE J., Radiosity and Realistic Image Synthesis, Academic Press Professional Inc., 1993, ISBN 0-12-178270-0;

[18] DACHSBACHER C. et all., Implict Visibility and Antiradiance for Interactive Global Illumination, Conference on Computer Graphics and Applications, 2007;

[19] DACHSBACHER C., STAMMINGER M., Splatting Indirect Illumination, Symphosium on Interactive 3D Graphics, Redwood City, California, 2006;

[20] DAVIES E., Machine Vision: Theory, Algorithms and Practicalities, Academic Press, 1990, pp. 42-44;

[21] DELVES L. M. and MOHAMED J. L., Computational methods for integral equations, Cambridge University Press, 1985;

[22] DIMITROV R., Cascaded Shadow Maps, NVIDIA Corporation Technical Document, August 2007;

[23] DOBASHI Y., YAMAMOTO T., NISHITA T., Radiosity for point-sampled geometry, Computer Graphics and Applications, 2004;

[24] DUTRE P., BALA K., BEKAERT P., Advanced Global Illumination, 29th International

Conference on Computer Graphics and Interactive Techniques, 21-26 July 2002, San Antonio, Texas USA;

[25] ELAD M., On the origin of the bilateral filter and ways to improve it, IEEE Transactions on Image Processing, vol. 11 no. 10, October 2002;

[26] EVANS A., Fast approximations for lighting of Dynamic Scenes, Media Molecule Ltd., Siggraph 2006;

[27] FOLDOC, Unified Modeling Language, 2002, available on-line at: http://foldoc.org/index.cgi?query=UML&action=Search, April 2009;

[28] GARREIN K., Lighting Techniques for Real-time 3D Rendering, Hogeschool, West-Vlaanderen, 2001-2002;

[29] GLASSNER A., An introduction to Ray-Tracing, Academic Press, 1989;

[30] GONZALEZ R., WOODS R., Digital Image Processing, Addison Wesley Publishing Company, 1992, p. 191;

[31] GORAL C., TORRANCE K., GREENBERG D., BATTAILE B., Modeling the Interaction of Light between Diffuse Surfaces, Computer Graphics, Volume 18, Number 3, July 1984;

[32] GRAY K., Microsoft DirectX9 Programmable Graphics Pipeline, Microsoft Press, 2003;

[33] GREEN R., Spherical harmonics lighting: the gritty details, Sony Computer Entertainment, January 2003, available on-line at: http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf, April 2009;

[34] GUIBAS L., KNUTH D., SHARIR M., Randomized incremental construction of Delaunay and Voronoi diagrams, Algorithmica 7, pp. 381-413, 1992;

[35] HANRAHAN P., Monte Carlo path tracing, Computer Graphics: Image Syntesis Techniques, lecture, 2001;

[36] HARALICK R., SHAPIRO L., Computer and Robot Vision, Addison Wesley Publishing Company, 1992, Vol. 1, Chapter 7;

[37] HARGREAVES S., HARRIS M., Deferred shading, Games Developer Conference 2004, available on-line at: http://www.talula.demon.co.uk/DeferredShading.pdf, April 2009;

[38] HAVRAN V., BITTNER J., HERZOG R., SEIDEL H.-P., Ray Maps for Global Illumination, Eurographics Symphosium on Rendering, 2005;

[39] High-Resolution Anti-Aliasing, Technical brief: High-resolution anti-aliasing through multisampling, may 2006;

[40] HOBSON E.W., The theory of spherical harmonics and ellipsoidal harmonics, Chelsea Pub. Co., 1955;

[41] IERUSALIMSHY R., CELES W., DE FIGUEIREDO L.E., LUA scripting language, available on-line at http://www.lua.org/, April 2009;

[42] JENSEN W., Global Illumination using Photon Maps, Rendering Techniques'96, Proceedings of the 7th Eurographics Workshop on Rendering, p. 21-30, 1996;

[43] JENSEN W., Realistic image synthesis using Photon Mapping, A.K. Peters Ltd., Massachusetts, 2001;

[44] KAJALIN V., Screen-space Ambient Occlusion, ShaderX^7 Programming Book, 2009;

[45] KAJIYA J., The Rendering Equation, Computer Graphics, 20(4): 143-150, August 1986, ACM Siggraph'86 Conference Proceedings;

[46] KELLER A., Instant Radiosity, International Conference on Computer Graphics and Interactive Techniques, ACM Press/Addison-Wesley Publishing Co., 1997;

[47] KONTKANEN J., LAINE S., Ambient Occlusion Fields, Symphosium on Interactive 3D Graphics and Games, ACM Siggraph 2005;

[48] LAFORTUNE E., WILLEMS Y., Bi-directional path tracing, Proceedings of Compugraphics'93, Alvor, Portugal 1993, p. 145-153;

[49] LENGYEL E., Mathematics for 3D Game Programming & Computer Graphics, Charles River Media, 2003, ISBN 978-1584500377;

[50] McREYNOLDS T., Advanced Graphics Programming Techniques Using OpenGL, Silicon Graphics, SIGGRAPH'99 Course, 1999;

[51] MENTAL IMAGES, Mental Ray renderer, available on-line at http://www.mentalimages.com/

[52] MICROSOFT MSDN LIBRARY, ShadowMap Sample, 2009, available on-line at: http://msdn.microsoft.com/en-us/library/bb147372(VS.85).aspx

[53] MICROSOFT, The Direct X Software Development Kit, DXSDK Documentation, March 2009, available on-line at: http://www.microsoft.com/downloads/details.aspx?FamilyID=24a541d6-0486-4453-8641-1eee9e21b282&displaylang=en, April 2009;

[54] MOORE D., Basic practice of statistics, 4th edition, WH Freeman Company, pp. 90-114, August 2006;

[55] NVIDIA DEVELOPER ZONE, Real-time shadow algorithms and techniques, 2008, available on-line at: http://developer.nvidia.com/object/doc_shadows.html

[56] OMG, UML Specification v1.1, OMG Document ad/97-08-11, available on-line at: http://www.omg.org/cgi-bin/doc?ad/97-08-11, April 2009;

[57] PARIS S., KORNPROBST P., TUMBLIN J., DURAND F., A gentle introduction to bilateral filtering and its applications, ACM SIGGRAPH, 2008;

[58] PIXAR ANIMATION STUDIOS, RenderMan, information available on-line at http://www.pixar.com/, April 2009;

[59] POLICARPO F., FONSECA F., Deferred shading tutorial, available on-line at: http://bat710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2007/Deferred_Shading_Tutorial_SBGAMES2005.pdf, April 2005;

[60] POV-Ray, The Persistence Vision of Raytracer, avaliable on-line at: http://www.povray.org/, April 2009;

[61] RAMPONI G., A rational edge-preserving smoother, Proceedings of the International Conference on Image Processing, vol. 1, pp. 151-154, Washington D.C., 1995;

[62] RITSCHEL T., GROSCH T., SEIDEL H., Approximating Dynamic Global Illumination in Image Space, Symphosium on Interactive 3D Graphics, ACM, 2009;

[63] RODGERS J. L., NICEWANDER W. A., Thirteen ways to look at the correlation coefficient, The American Statistician, 42(1) pp. 59-66, February 1988;

[64] ROHLEDER P., Advanced GUI System for Games, ShaderX^7 Programming Book, Course Technology, 2009;

[65] ROHLEDER P., Fast and efficient real-time water rendering using advanced GPU programming techniques, ICYR 2006, Zielona Góra;

[66] ROHLEDER P., JAMROZIK M., Sunlight with volumetric light rays, ShaderX^6 Programming Book, Course Technology, 2007;

[67] ROHLEDER P., Real-time ambient occlusion for dynamic and complex scenes, ISAT 2008, Szklarska Poręba, 2008;

[68] ROHLEDER P., Wykładnicze Mapy Cieni, Software Developer's Journal, 02.2009;

[69] SALVI M., A conceptually simpler way to derive exponential shadow maps, June the 12th, 2008, available on-line at:

http://pixelstoomany.wordpress.com/category/shadows/exponential-shadow-maps/

[70] SALVI M., Rendering Filtered Shadows with Exponential Shadow Maps, ShaderX6, Course Technology, 2008

[71] SARANSAARI H., LAINE S., KONTKANEN J., LEHTINEN J., AILA T., Incremental Instant Radiosity, ShaderX^6 Advanced Rendering Techniques, Course Technology, 2007;

[72] SEGOVIA B. et all., Bidirectional Instant Radiosity, Eurographics Symposium on Rendering 2006;

[73] SHANMUGAM P., ARIKAN O., Hardware Accelerated Ambient Occlusion Techniques on GPUs, Symphosium on Interactive 3D Graphics, Proceesings of the 2007 Symphosium on Interactive 3D Graphics and Games, 2007;

[74] SHEPPARD D., Beginner's Introduction to PERL, available on-line at: http://www.perl.com/pub/a/2000/10/begperl1.html, October 2000;

[75] SHIRLEY P., MORLEY K., Realistic Ray-Tracing, 2nd edition, A.K. Peters, 2001;

[76] SLOAN P., GROVINDARAJU N., NOWROUZEZAHRAI D., SNYDER J., Image-based proxy accumulation for real-time soft global illumination, Proceedings ot the 15th Pacific Conference on Computer Graphics and Applications, 2007;

[77] SLOAN P., KAUTZ J., SNYDER J., Precomputed Radiance Transfer for Real-Time Rendering in dynamic, Low-Frequency Lighting Environments, ACM Transactions on Graphics, ACM Press, 2002;

[78] SOLER C. et al., Hierarchical Screen-Space Indirect Illumination for Video Games, INRIA, Rapport de recherche, December 2009;

[79] SZABO M., Hardware generated shadows, CESCG'2004, April 2004;

[80] SZESCI L., SZIRMAY-KALOS L., SBERT M., Interactive Global Illumination with

Precomputed Radiance Maps, ShaderX^6 Advanced Rendering Techniques, Course Technology, 2007;

[81] TABELLION E., LAMORLETTE A., An approximate global illumination system for computer generated films, ACM Transactions on Computer Graphics 23, pp. 469-476, 2004;

[82] TATARCHUK N. at all, Advanced Real-Time Rendering in 3D Graphics and Games, SIGGRAPH 2006, August the 1st;

[83] TECHLAND SP. Z O.O., Electronic entertainment and software developer, publisher and distributor, http://www.techland.pl, April 2009;

[84] THIBIEROZ N., Robust Order-Independent Transparency via Reverse Depth Peeling in Direct X 10, ShaderX6 Programming Book, 2007;

[85] TOMASI C., MANDUCHI R., Bilateral filtering for gray and color images, 6th International conference on Computer Vision, pp. 839-846, Bombay, India, 1998;

[86] UBISOFT, Ubisoft Entertainment - French computer and video game publisher and developer, http://www.ubi.com/, April 2009;

[87] VALIENT M,. Deferred rendering in Killzone 2, Guerrilla, Develop Conference, Brighton, July 2007;

[88] WALD I., Real-time Raytracing and Interactive Global Illumination, PhD. Thesis, Saarland University, 2004;

[89] WALL L., CHRISTIANSEN T., ORWANT J., Programming Perl, 3rd edition, O'Reilly, July 2000;

[90] WANG R., ZHU J., HUMPHREYS G., Precomputed Radiance Transfer for Real-Time Indirect Lighting using A Spectral Mesh Basis, Eurographics Symphosium on Rendering, 2007;

[91] WATERS Z., Photon mapping, tutorial available on-line at:

http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html

[92] WINTER A., An investigation into real-time 3D polygon rendering using BSP trees, 1999, available on-line at: http://www-compsci.swan.ac.uk/~csandrew/papers/disser.pdf

[93] ZHOU T., CHEN J., PULLEN M., Accurate Depth of Field Simulation in Real-Time, Computer Graphics Forum, Volume 26 (2007), 1 pp. 15-23;