Real-Time Global Illumination on Distributed Systems

Mark Charles Magro

Supervisor: Dr Keith Bugeja

Co-supervisor: Dr Sandro Spina

October, 2023

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science.

L-Università ta' Malta
Faculty of Information & Communication Technology

First edition, Friday 13th October, 2023

FACULTY/~~INSTITUTE/CENTRE/SCHOOL~~ Information & Communication Technology

DECLARATION OF AUTHENTICITY FOR DOCTORAL STUDENTS

(a) Authenticity of Thesis/Dissertation

I hereby declare that I am the legitimate author of this Thesis/Dissertation and that it is my original work.

No portion of this work has been submitted in support of an application for another degree or qualification of this or any other university or institution of higher education.

I hold the University of Malta harmless against any third party claims with regard to copyright violation, breach of confidentiality, defamation and any other third party right infringement.

(b) Research Code of Practice and Ethics Review Procedure

I declare that I have abided by the University's Research Ethics Review Procedures. Research Ethics & Data Protection form code ICT-2022-00081.

☑ As a Ph.D. student, as per Regulation 66 of the Doctor of Philosophy Regulations, I accept that my thesis be made publicly available on the University of Malta Institutional Repository.

☐ As a Doctor of Sacred Theology student, as per Regulation 17 (3) of the Doctor of Sacred Theology Regulations, I accept that my thesis be made publicly available on the University of Malta Institutional Repository.

☐ As a Doctor of Music student, as per Regulation 26 (2) of the Doctor of Music Regulations, I accept that my dissertation be made publicly available on the University of Malta Institutional Repository.

☐ As a Professional Doctorate student, as per Regulation 55 of the Professional Doctorate Regulations, I accept that my dissertation be made publicly available on the University of Malta Institutional Repository.

To Mum

Acknowledgements

I am immensely grateful to my mentors, Dr Keith Bugeja and Dr Sandro Spina, for their guidance, patience, and constant encouragement. In no small part thanks to you, although these years were tough they were also enjoyable. Thank you, your help was invaluable. This thesis would not have been possible without you.

I would like to thank my friends and fellow students for their support and for being such a pleasant group to work with. Kevin and Adrian, I am indebted to you for proofreading and going through the technical stuff in this thesis, and for suggesting numerous improvements. Thank you for all the help and encouragement you have given me during my research. Jennifer and Duncan, a heartfelt thank you for lending a helping hand whenever I needed it.

This thesis would also not have been possible without the help of my family, especially Mum who saved me countless hours with her care. Ma, Michelle, Paul and David, I greatly appreciate your patience, support and encouragement throughout my research. Michelle, thanks for proofreading this thesis so thoroughly and for your helpful suggestions.

Abstract

Realistic computer-generated visuals captivate users and provide them with immersive experiences. Synthesising these images in real time requires significant computational power and is out of reach for a wide range of commodity hardware, particularly for mobile devices. Remote rendering solves this problem by computing frames on the Cloud and streaming the results to the client device as video. This solution provides good image quality but introduces latency, which may make applications appear unresponsive, degrading user experience. It may also require significant bandwidth. This work investigates an alternative distributed rendering strategy, where the computational power of the local device is not discarded but instead used to eliminate or reduce latency. Three methods are presented, all using a client-server architecture that splits the rendering pipeline between a powerful remote endpoint and a weaker local device. The first makes use of sparse irradiance sampling on a voxelised representation of the scene. It supports multiple clients and is highly configurable, allowing the use of different interpolation schemes according to the capability of the device; image quality can be reduced to lower reconstruction cost. The second stores radiance in a megatexture and communicates it to the client device, where rendering is performed at a low cost by sampling the megatexture. A coarse megatexture that fits into GPU memory is maintained on the client device and used to provide temporary low-quality output until high-quality server data are received. The third uses the double warping image-based rendering technique to produce novel views from two reference views. The client device also receives irradiance data which it caches in a coarse megatexture. The cache is used in a fallback mechanism that mitigates visual artefacts due to holes in the data. The results show that input lag can be eliminated and bandwidth requirements can be kept low, while retaining a measure of fault tolerance and decent image quality. It is envisaged that distributed methods similar to the ones proposed will gain more traction as the computational capability of commodity devices increases and they can be assigned larger workloads and use more sophisticated algorithms.

Contents

List of Figures

List of Tables

List of Abbreviations

ADR	Asynchronous Distributed Rendering
AVC	Advanced Video Coding
BRDF	Bidirectional Reflectance Distribution Function
BSDF	Bidirectional Scattering Distribution Function
BSSRDF	Bidirectional Surface Scattering Reflectance Distribution Function
BTDF	Bidirectional Transmittance Distribution Function
CGI	Computer-Generated Imagery
CPU	Central Processing Unit
DARM	Device-Agnostic Radiance Megatextures
DARM-V	DARM Virtual Atlas
DSSIM	Structural Dissimilarity
EM	Electromagnetic
FFS	Full-Frame Streaming
GI	Global Illumination
GPU	Graphics Processing Unit
GUI	Graphical User Interface
H.264	AVC
H.265	HEVC
HDR	High Dynamic Range
HEVC	High Efficiency Video Coding
HSH	Hemispherical Harmonics
HVS	Human Visual System

IBR	Image-Based Rendering
IGI	Instant Global Illumination
IMC	Irradiance Megatexture Cache
IR	Infrared
LAN	Local Area Network
MC	Monte Carlo
MSSIM	Mean Structural Similarity
NDC	Normalised Device Coordinates
P2P	Peer-to-Peer
PDF	Probability Density Function
ppwu	pixels per world unit
PSNR	Peak Signal-to-Noise Ratio
PVS	Potentially Visible Set
RAIL	Remote Asynchronous Indirect Lighting
ReGGI	Regular Grid Global Illumination
RR	Remote Rendering
RSM	Reflective Shadow Map
SH	Spherical Harmonics
SIMD	Single Instruction, Multiple Data
UV	Ultraviolet
VIS	Visible Spectrum
VPL	Virtual Point Light
VR	Virtual Reality
VRAM	Video Random Access Memory

List of Publications

Journal Paper

Mark Magro, Keith Bugeja, Sandro Spina, and Kurt Debattista. Cloud-based Dynamic GI for Shared VR Experiences. *IEEE Computer Graphics and Applications*, Volume 40(5), pages 10-25, 2020.

Peer-Reviewed Papers

Mark Magro, Keith Bugeja, Sandro Spina, and Kurt Debattista. Interactive Cloud-based Global Illumination for Shared Virtual Environments. In *2019 11th International Conference on Virtual Worlds and Games for Serious Applications (VS-Games)*, pages 1-8. IEEE, 2019.

Mark Magro, Keith Bugeja, Sandro Spina, Kevin Napoli, and Adrian De Barro. Atlas Shrugged: Device-agnostic Radiance Megatextures. In *Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, VISIGRAPP 2020, Volume 1: GRAPP, Malta, February 27-29, 2020*, pages 255-262. SciTePress, 2020.

1 Introduction

Computer-generated images are derived from a mathematical representation that includes the geometric and material properties of objects, a description of the light sources, and camera parameters. Phenomena such as shadows, reflection, and refraction are reproduced accurately by mimicking the propagation of light and its interaction with materials. In decades of active research, light transport models have become increasingly sophisticated and can now synthesise images that are hard to distinguish from photographs. Noteworthy breakthroughs over the years were the modelling of light using ray optics (Appel, 1968), the realisation that new rays can be fired recursively from where a previous ray intersected a surface (Whitted, 1979), the inclusion of light reflected from all types of materials, not only from perfectly reflective surfaces (Cook et al., 1984; Goral et al., 1984; Kajiya, 1986), and the rendering equation, a mathematical formulation for global illumination (Kajiya, 1986). These achievements established the foundations for the computer graphics we see today.

The rendering equation is a recursive integral equation that can only be solved with numerical methods, typically finite element or Monte Carlo methods. Whichever approach is used, the computation is expensive timewise, with rendering times dictated by factors such as desired quality, scene complexity, output resolution, and the hardware used. A large body of work has been dedicated to speeding up the process with the aim of obtaining convincing solutions in real time, that is, in only a few milliseconds. Images produced and displayed at this rate give the impression of a smooth animation, which is the concept used for television, cinema, and digital screens. There has been a great deal of progress towards achieving this seemingly impossible task, the visuals presented in modern video games being prime examples of the state of the art. The images produced in this domain are not paragons of physical correctness, but they are pretty, plausible, and provide users with immersive and enjoyable experiences. Making these images happen is a combination of powerful hardware, clever rendering strategies, and a great many approximations. For weaker battery-operated mobile devices this technology is provided by distributed rendering, where rendering is performed on powerful cloud-

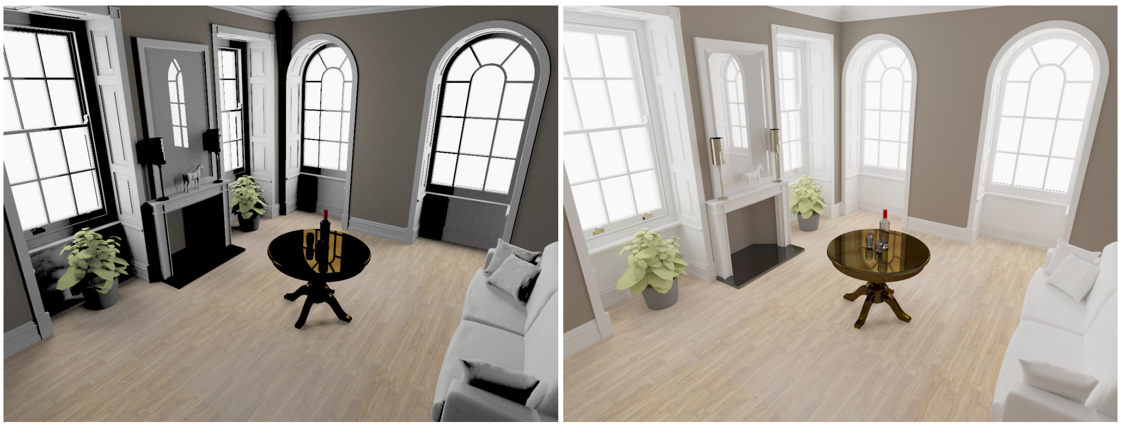


Figure 1.1: Left: Local illumination. Right: Global illumination.

based hardware and streamed to the device as video.

This thesis investigates real-time distributed rendering solutions with the aim of improving user experience over that provided by the video streaming approach, which is by a large margin the dominant form of remote rendering in use today. In particular, the thesis focusses on latency-hiding techniques where input lag is replaced by the arguably more manageable problem of output lag. Emphasis is placed on techniques that are suitable for deployment on mobile devices such as smartphones, tablets, laptops and untethered VR headsets.

1.1 Global Illumination

Consider a surface that is illuminated only by direct light, that is, light that reaches it directly from a light source. Surface points that do not have a direct line of sight to the light source are completely dark. This is called local illumination (Figure 1.1, left). Images produced using this form of illumination tend to appear unrealistic because an important lighting component, indirect light, is missing. Indirect light is light that arrives at a surface by an indirect path, for example by reflection off a nearby surface. Global illumination (GI) takes into account both direct and indirect light, producing more realistic looking images (Figure 1.1, right and Figure 1.2).

Global illumination finds uses wherever a realistic simulation of light transport is required. Special effects are commonplace in the film industry and need to appear authentic to blend in well with the other objects in the frame. Similarly, mixed reality applications require seamless transitions between real and virtual objects. Flight simulators for pilot training and combat simulations used in the military benefit from realistic

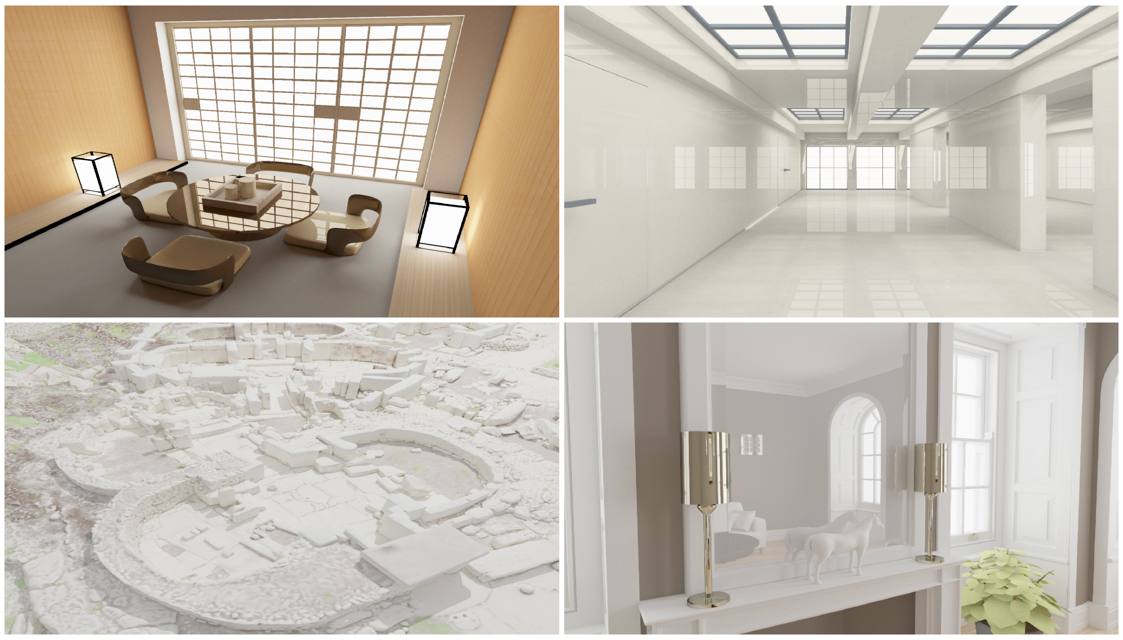


Figure 1.2: Global illumination.

visuals. Global illumination can provide truly immersive experiences for virtual reality applications. Product designers and architects can visualise the finished product beforehand. Realistic interior design walkthroughs help buyers make better choices. Global illumination has also been used in cultural heritage and archaeology to virtually reconstruct structures. Artists and content creators are able to use a more efficient workflow because different lighting conditions can be recreated accurately.

1.2 Real-Time Rendering

Synthesising realistic images requires an enormous amount of computation. In the film industry, when computer-generated imagery (CGI) is used, the aim is to produce images, or *frames*, of the highest possible quality; the time taken to generate a frame is not of primary importance. In these circumstances, offline renderers are used. These renderers use physically based techniques to simulate light transport accurately. The production of a single frame may require minutes, or even hours.

Applications that require high-fidelity images in real time, such as video games, interior design walkthroughs, simulations, and XR (virtual reality and its variants augmented reality and mixed reality), have a very tight time budget in which to produce a frame. At 60 frames per second, the time budget is only 16.67 milliseconds; the time needed to generate a frame may be even less than that if the time budget is shared with the application's logic. These applications use a different breed of renderers from those used in the film industry. These real-time renderers approximate the results of offline renderers and operate extremely efficiently. On high-end machines, high-fidelity images can be produced at a resolution of 1080p or higher, at a rate of 60 frames per second or more. VR applications may require a frame rate of at least 72 Hz to avoid causing cybersickness, where the user experiences sensations of nausea and dizziness.

1.3 Distributed Rendering

Most consumer devices, including desktops, laptops, tablets, smartphones, and untethered VR headsets lack the computation power required to run real-time renderers. There is therefore a need to obtain computation capability from elsewhere. The perfect candidate is the Cloud which can provide computation as a service practically anywhere and to any device. This is the solution used by cloud-gaming operators. Frames are rendered on powerful cloud-based servers and streamed as video to the consumer's device. In this way, operators can provide the service to any kind of device no matter its computational capability. This setup is known as cloud gaming, game streaming, remote rendering (RR), or full-frame streaming (FFS), and the best solutions to date include NVIDIA's GeForce NOW, Sony's PlayStation Now, Microsoft's Xbox Cloud Gaming, and Google's Stadia.

RR works well but it has some drawbacks, most importantly the issue of latency, or *input lag*. This is the delay between the user triggering an action and seeing the result of that action on screen. The delay occurs because the user's action needs to be relayed to the server for processing, after which the rendered frame needs to be streamed back to the user's device where it is decoded and presented. Although this only takes a fraction of a second, it is enough to make for an uncomfortable experience. In video games, for example, a delay larger than 75 milliseconds starts to be noticeable, and if the delay is around 100 milliseconds, a fast-paced action-heavy game becomes practically unplayable (Beigbeder et al., 2004). In latency tests performed in early 2020, the lag for GeForce NOW was measured at 69 milliseconds[1]. Lag measurements can vary wildly because they depend on network speed and the distance to the data centre. Input lag can be experienced even when playing natively (on a local PC), and may be substantial if high graphics quality settings are used. In tests on Google Stadia in late 2019, latency

[1] https://www.pcgamesn.com/nvidia/geforce-now-competitive-mode-latency
Last accessed: 2022-07-29.

values in excess of 200 milliseconds were measured for some configurations[2]. RR may require a high bandwidth and a stable network connection is a must.

1.4 Motivation

An alternative to remote rendering is asynchronous distributed rendering (ADR). In this paradigm, the client device is not treated as a thin client. Communication with a remote server is still necessary, but the processing power of the client device, although small, may be enough to implement strategies where milliseconds of latency can be shaved off, resulting in a better user experience. Furthermore, short network hiccups may be worked around. The main problem with ADR appears to be that it is much more complex to implement than RR. Features that essentially come for free with RR, such as great image quality, sky boxes, dynamic environments, collisions with boundaries, and GUI overlays, may be difficult to achieve in ADR. Moreover, client and server computations need to be merged while keeping the two endpoints synchronised, and the server may need to send more data than in RR (for example, depth data may need to be sent together with the frame). Is the added complexity of ADR worth investing in? What is preferable, reduced lag or perfect image quality?

The amount of processing to distribute to the client depends on the client's capabilities. The least powerful clients can only receive a video stream, decode it, and present it. Only remote rendering is suitable for these clients. A slightly more powerful client could reproject frames using image-based techniques when the camera is moving or rotating; this would compensate for some latency but could introduce visual artefacts. At the next client power level, the client is capable of processing the scene's geometry. At this level, shading can be performed in a number of ways. The client could merely project a texture over the geometry. If texture information (albedo) is available, the client could apply some crude shading to the geometry; this primitive form of shading could be enhanced by information received from the server. More sophisticated clients could shade the geometry using information about the light sources in the scene. Again, the client's shading could then be enhanced with other information received from the server, to include global illumination effects, for example.

[2] https://www.pcgamer.com/heres-how-stadias-input-lag-compares-to-native-pc-gaming *Last accessed: 2022-07-29.*

1.5 Research Questions

Real-time global illumination benefits numerous applications as it produces realistic visuals and provides compelling user experiences. However, it is computationally intensive and far beyond the capabilities of mobile devices such as smartphones and untethered VR headsets. This difficulty may be overcome by distributing the rendering pipeline. However, by doing so, other challenges come to the fore and need to be addressed. Network instability requires fault tolerance mechanisms. A high bandwidth may be required due to the amount of data that needs to be transferred over the network. This issue is further compounded by the high resolutions in use today and by HDR (high dynamic range) requirements, where colour data are stored at a higher quality. A low-latency connection is required for highly responsive applications. Input lag and unsynchronised local and remote illumination computations degrade the user experience.

Past research (Crassin et al., 2015; Bugeja et al., 2018; Mueller et al., 2018; Hladky et al., 2019b) has shown that asynchronous distributed rendering has the potential to bring low-latency high-fidelity rendering to a wide variety of devices. By distributing the rendering pipeline between the Cloud and the client, input lag can be significantly reduced or even eliminated entirely, depending on the capability of the local device. ADR therefore seems to be a perfect fit for the cloud-gaming paradigm, where a high level of interaction is required in real time and low input lag is key. However, instead of investing in ADR, cloud-gaming services use remote rendering, where all rendering takes place on the cloud-end of the pipeline and the client device is effectively a dumb terminal. In our research we would like to identify the sticking point for a transition from remote rendering approaches to ADR and to improve upon existing research to make the adoption of ADR more enticing. Our research questions therefore are:

- Why is asynchronous distributed rendering not viable yet?
- How can it be improved to become viable?

1.6 Research Methodology

Our research focusses on distributed rendering pipelines that are very powerful at the remote end (cloud-based servers) and much less capable at the local end (smartphones and untethered VR headsets). Asynchronous distributed rendering is an area that is being actively researched. In recent years it seems to be gathering momentum. We review methods that use this paradigm and add to the body of knowledge in this area.

Regular Grid Global Illumination (ReGGI)

The method described in Chapter 5, ReGGI, investigates the quality of diffuse indirect illumination reconstructed from a set of extremely sparse illumination samples. The method also investigates efficient reconstruction techniques in order to provide a global illumination solution at a low cost, and aims to eliminate input lag. A spatial partitioning of the scene is used and a number of interpolation schemes are proposed.

Device-Agnostic Radiance Megatextures (DARM)

The strategy used in our work related to radiance megatextures (Chapter 6, DARM) is to represent the entire scene as a megatexture, a large texture atlas. The rendering back end shades the megatexture and communicates the relevant sections of it to the client, where efficient rasterisation techniques are used for image reconstruction. This method also aims to eliminate input lag. Further aims are to study the effectiveness of megatextures at storing scene-wide illumination data, and to improve upon ReGGI by not being limited to diffuse materials only and by having a lower reconstruction cost. A variant of the method was integrated into a popular game engine.

Irradiance Megatexture Cache (IMC)

The method proposed in Chapter 7, IMC, aims to improve image quality over that of DARM while keeping reconstruction costs low and eliminating input lag. The method uses a combination of image-based rendering and local rendering enhanced with remotely computed irradiance. The method makes use of megatextures and is integrated into a popular game engine.

1.7 Thesis Structure

Chapter 2: Background provides context for the terminology and techniques used in this thesis. This chapter includes a detailed description of radiometric quantities and the rendering equation, together with its formulations.

Chapter 3: Real-Time Global Illumination provides a literature review of algorithms that speed up the computation of global illumination and are suitable for real-time application. The categories of algorithms described include those based on caching mechanisms, many-light methods, and others.

Chapter 4: Distributed Rendering introduces rendering on distributed systems and analyses the related literature. A number of cluster-based, grid-based and peer-to-peer methods are described, together with various cloud-based collaborative approaches.

Chapter 5: Regular Grid Global Illumination describes a method that enhances local rendering with remotely computed indirect illumination. The method supports virtual environments that are shared by a number of users.

Chapter 6: Radiance Megatextures proposes a strategy that provides low-cost global illumination by storing scene-wide illumination data in megatextures.

Chapter 7: Irradiance Megatexture Cache presents a method that uses image-based rendering to hide latency and provide high-quality visuals, while employing megatextures to mitigate the impact of missing data.

Chapter 8: Conclusion wraps up the thesis. It reviews this work, summarises the contributions and discusses the limitations of the proposed methods, suggesting avenues for further research.

2 Background

This chapter provides background information for the concepts and technologies mentioned throughout the thesis. The fundamentals of radiometry and photometry, reflectance models, the rendering equation and Monte Carlo integration are necessary for computing global illumination. A brief overview of the rasterisation pipeline is given, together with image-based rendering techniques that provide a cheap way to create novel views of a scene from one or more reference views. The concepts behind gamma correction and tone mapping are outlined to clarify how the perception of the human eye is taken into account when presenting images on a screen. Video compression technologies are used to transfer image data efficiently over computer networks.

2.1 Radiometry and Photometry

Electromagnetic (EM) radiation, or radiant energy, is propagated by photons, tiny massless packets of energy that exhibit both particle and wave properties and travel at the speed of light. Radiometry measures EM radiation in the optical spectrum, the portion of the electromagnetic spectrum that includes ultraviolet (UV) radiation, the visible spectrum (VIS), and infrared (IR) radiation (Figure 2.1). Photometry measures EM radiation

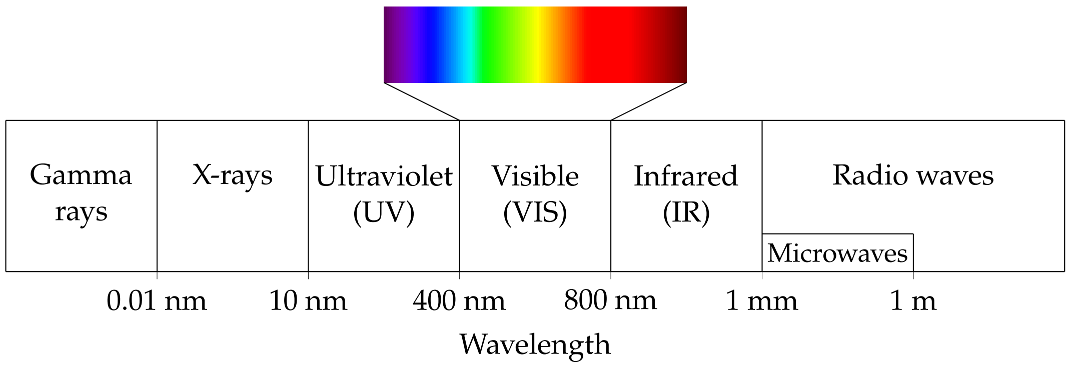


Figure 2.1: The electromagnetic spectrum.

Radiometry			**Photometry**		
Quantity	**Symbol**	**Unit**	**Quantity**	**Symbol**	**Unit**
Radiant energy	Q_e	joule J	Luminous energy	Q_v	talbot T
Radiant flux or Radiant power	Φ_e	watt W	Luminous flux or Luminous power	Φ_v	lumen lm
Irradiance	E_e	$\mathrm{W\,m^{-2}}$	Illuminance	E_v	lux lx or $\mathrm{lm\,m^{-2}}$
Radiosity or Radiant exitance	B M_e	$\mathrm{W\,m^{-2}}$	Luminous exitance	M_v	lux lx or $\mathrm{lm\,m^{-2}}$
Radiant intensity	I_e	$\mathrm{W\,sr^{-1}}$	Luminous intensity	I_v	candela cd or $\mathrm{lm\,sr^{-1}}$
Radiance	L_e	$\mathrm{W\,m^{-2}\,sr^{-1}}$	Luminance	L_v	nit $\mathrm{lm\,m^{-2}\,sr^{-1}}$

Table 2.1: Radiometric and photometric quantities.

as perceived by the human visual system (HVS) and is confined to the visible spectrum. The boundaries of the different types of EM radiation are not as clear cut as they appear in the figure. For example, sensitivity readings of the eye are average values. Since some people are more sensitive to light than others, the visible spectrum dips slightly into the ultraviolet and infrared ranges illustrated, from around 360 nm to 830 nm. In photometry, weights are assigned to different light wavelengths depending on the strength of the eye's response. Since the eye is most sensitive to green, wavelengths within the green "band", approximately from 500 nm to 565 nm, are assigned the highest weights.

Radiometric and photometric quantities are very similar, to the extent that sometimes the same nomenclature and symbol are used for a radiometric quantity and its photometric counterpart. To distinguish between the two, when it is not possible to do so from context, radiometric quantities are prefixed by *radiant* whereas photometric quantities are prefixed by *luminous*. Similarly, radiometric symbols use an "*e*" subscript (for *energy*) and photometric symbols use a "*v*" subscript (for *visual*). Hence we have *radiant flux* (Φ_e) and *luminous flux* (Φ_v), radiant intensity (I_e) and luminous intensity (I_v), and so on. Table 2.1 lists the most commonly used quantities.

The radiometric quantities depend on wavelength too. When referring to a radiometric quantity at a specific wavelength, the "spectral" prefix and the subscript "λ" (the symbol for wavelength) are used. For example, L_λ is spectral radiance, with units $\mathrm{W\,m^{-2}\,sr^{-1}\,nm^{-1}}$. Computing a radiometric quantity therefore requires integrating over the required range of wavelengths; for rendering this would be the range of wavelengths

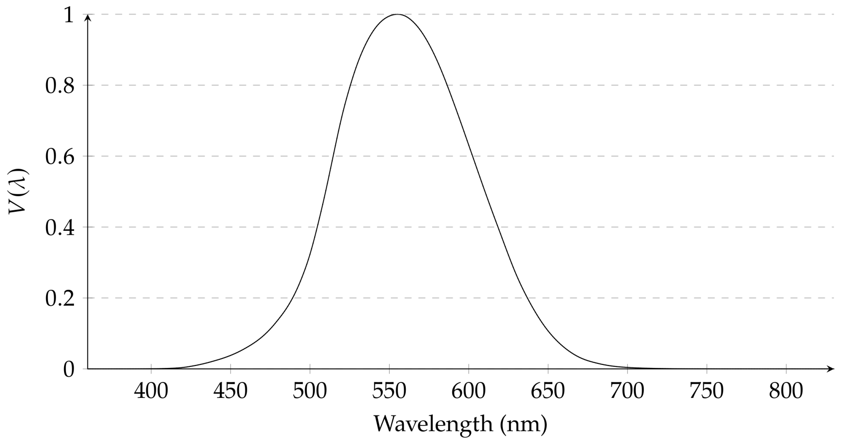


Figure 2.2: The CIE (1931) photometric curve. $V(\lambda)$ is the photopic (day vision) spectral luminous efficiency function.

in the visible spectrum. For clarity, except for the note on converting between quantities in the following paragraph, although the dependency on wavelength is there, we will generally not make it explicit. Instead we will proceed as if we were dealing with monochromatic light, that is, light of only one wavelength (or a narrow band of wavelengths).

Conversion from radiometric quantities to photometric quantities requires the formula

$$X_v = 683 \int_{360}^{830} X_\lambda \, V(\lambda) \, d\lambda. \tag{2.1}$$

X_v is the required photometric quantity and X_λ is the corresponding spectral radiometric quantity. $V(\lambda)$ is the spectral luminous efficiency function, obtained from the CIE[1] photometric curve (Figure 2.2), a bell-shaped curve that has a peak at 555 nm, the wavelength the eye is most sensitive to. The value 683 comes from the fact that one watt of power at the 555 nm wavelength is equivalent to 683 lumens. The integral domain is the range of wavelengths in the visible spectrum. The range used here corresponds to the one tabulated by the CIE for the photometric curve. However, more conservative wavelength ranges such as from 380 nm to 770 nm are often used (McCluney, 2014). Converting from photometric to radiometric quantities is more difficult, and is not always

[1] Commission Internationale de l'Eclairage (the International Commission on Illumination)

possible since the spectral radiometric quantity needs to be known over a larger range of wavelengths, not just for the visible spectrum (Palmer, 2000). In practice, for light sources, converting from luminous flux to radiant power is straightforward. Since the luminous flux and wattage of a light source are provided by the manufacturer, an average value for luminous efficacy (the ratio of lumens per watt) is easily obtained. This value can then be used as an approximation for similar light sources with different wattages. Alternatively, tables of luminous efficacy values for commonly used light sources are readily found. For example, the typical luminous efficacy of tungsten incandescent light bulbs is 15 $\mathrm{lm\,W^{-1}}$ and that for LED lamps is 90 $\mathrm{lm\,W^{-1}}$.

In rendering, illumination is typically computed using the physically based radiometric quantities rather than with perceptual photometric quantities. However, since it may be convenient to specify the brightness of light sources in photometric units, an initial conversion between units may still be required. A notable exception to the rule is the Frostbite rendering engine, which uses photometric units throughout the entire rendering pipeline (Lagarde and De Rousiers, 2014).

Radiant Energy

Radiant energy (Q), measured in joules (J), is the energy transported by light. The energy of a photon is $h\,\nu$, where h is Planck's constant and ν is the photon's frequency. Alternatively, the energy of a photon can be written in terms of the photon's wavelength, λ,

$$Q = h\,\nu = \frac{h\,c}{\lambda}, \tag{2.2}$$

where c is the speed of light.

Radiant Power

Radiant power (Φ), also called radiant flux, is radiant energy per unit time ($\mathrm{J\,s^{-1}}$). It is measured in watts (W) and describes the flow of radiant energy. Higher wattage incorrectly tends to be associated with higher brightness. For example, we would expect a 100 W light bulb to be brighter than a 40 W bulb. However, wattage describes power consumption not brightness. In fact it is perfectly possible that a modern energy-efficient light bulb uses less power but is as bright or even brighter than an old bulb. For physical light sources, for a correct indication of brightness, the photometric counterpart of radiant power, the lumen (lm), should be used instead. In our calculations, however, we will use watts and assume that light sources operate at 100% efficiency.

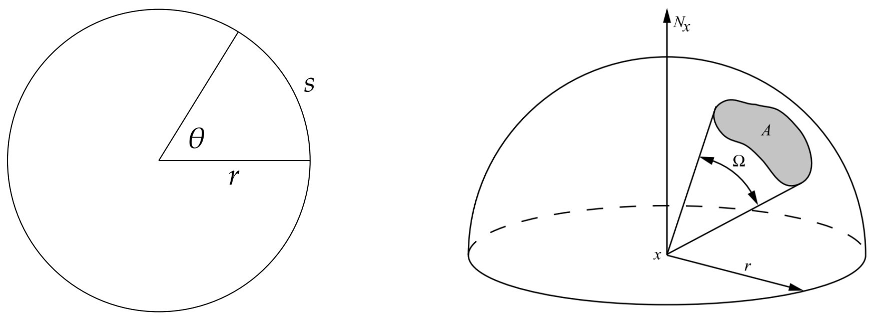


Figure 2.3: Left: Plane angle. Right: Solid angle (Dutré et al., 2006).

Common assumptions in computer graphics are that light travels infinitely fast and that the light in a virtual environment reaches equilibrium immediately, that is, the distribution of light energy is not changing. This is called the steady state assumption. It is very close to what occurs in real life, where equilibrium is reached nearly instantaneously, and it simplifies calculations. Using the steady state assumption, we can make instantaneous measurements of quantities:

$$\Phi = \lim_{\Delta t \to 0} \frac{\Delta Q}{\Delta t} = \frac{dQ}{dt} \text{ [W]}. \tag{2.3}$$

Irradiance

Irradiance (E) is incident radiant power per unit surface area (W m^{-2}). A similar quantity, radiosity (B), is the radiant power leaving a surface per unit surface area (W m^{-2}). If the surface is a light source, the radiant power is emitted. Otherwise, radiant power is reflected by or transmitted through the surface. Some texts refer to radiosity as radiant exitance (M) (Palmer and Grant, 2010; Pharr et al., 2016); we will consider radiosity and radiant exitance to be equivalent too. However, note that there is a measure of ambiguity regarding radiant exitance since it is sometimes defined as the emitted component of radiosity. Notwithstanding this ambiguity, irradiance, radiosity and radiant exitance all have the same units and describe the area density of radiant power.

$$E(\boldsymbol{x}) = B(\boldsymbol{x}) = M(\boldsymbol{x}) = \lim_{\Delta A \to 0} \frac{\Delta \Phi}{\Delta A} = \frac{d\Phi}{dA} \text{ [W m}^{-2}\text{]}. \tag{2.4}$$

These quantities are functions of position $\boldsymbol{x}$. The differential area dA is an infinitesimally small area centred at $\boldsymbol{x}$.

Solid Angle

The next radiometric quantities both refer to a *solid angle*. Recall the relationship between the angle θ subtended at the centre of a circle of radius r by an arc of length s (Figure 2.3, left):

$$\theta = \frac{s}{r} \text{ [rad].} \tag{2.5}$$

θ is called the plane angle and is measured in radians (rad). For an arc of length r, $\theta = \frac{r}{r} = 1$ rad. The plane angle subtended by a circle is $\frac{2\pi r}{r}$ rad $= 2\pi$ rad. The solid angle is the three-dimensional counterpart of the plane angle. It is defined as the angle Ω subtended at the centre of a sphere of radius r by an area A on the sphere's surface (Figure 2.3, right). The solid angle is expressed in steradians (sr) and is calculated as

$$\Omega = \frac{A}{r^2} \text{ [sr].} \tag{2.6}$$

If the area is equal to r^2, $\Omega = \frac{r^2}{r^2} = 1$ sr. The solid angle subtended by a sphere is $\frac{4\pi r^2}{r^2}$ sr $= 4\pi$ sr. Note that the area can take any shape. The solid angle subtended by an arbitrary surface is computed by projecting the surface onto a sphere of radius r, determining the projected area, and then proceeding using Equation 2.6. The r^2 term in the denominator is the familiar inverse square law, where the value of a quantity drops off at a rate proportional to the square of the distance. For a small surface with area A centred around the point $\boldsymbol{x}$, the projected area can be approximated as $A \cos\theta$, where θ is the angle between the surface's normal and the line connecting $\boldsymbol{x}$ to the sphere's centre (Dutré et al., 2006). The solid angle is then computed as

$$\Omega = \frac{A \cos\theta}{r^2} \text{ [sr].} \tag{2.7}$$

Differential Solid Angle

A point P in three-dimensional space can be described in spherical coordinates (r, ϕ, θ) (Figure 2.4, left). r is the distance from the origin O. $\phi \in [0, 2\pi]$ is the azimuthal angle. In right-handed coordinate systems it is measured counter-clockwise from the x-axis on the xy-plane. $\theta \in [0, \pi]$ is the polar angle. It is measured from the positive z-axis towards the line segment OP. Given angles ϕ and θ, a differential area on a sphere of radius r can be constructed by increasing both angles by infinitesimal amounts, $d\phi$ and $d\theta$ respectively (Figure 2.4, right). The arc length intercepted by $d\theta$ is $r\ d\theta$. The arc length intercepted by $d\phi$ is $r\ \sin\theta\ d\phi$, where the $\sin\theta$ term accounts for the fact that the arc lengths intercepted by $d\phi$ are larger close to the zenith (the "equator") and smaller

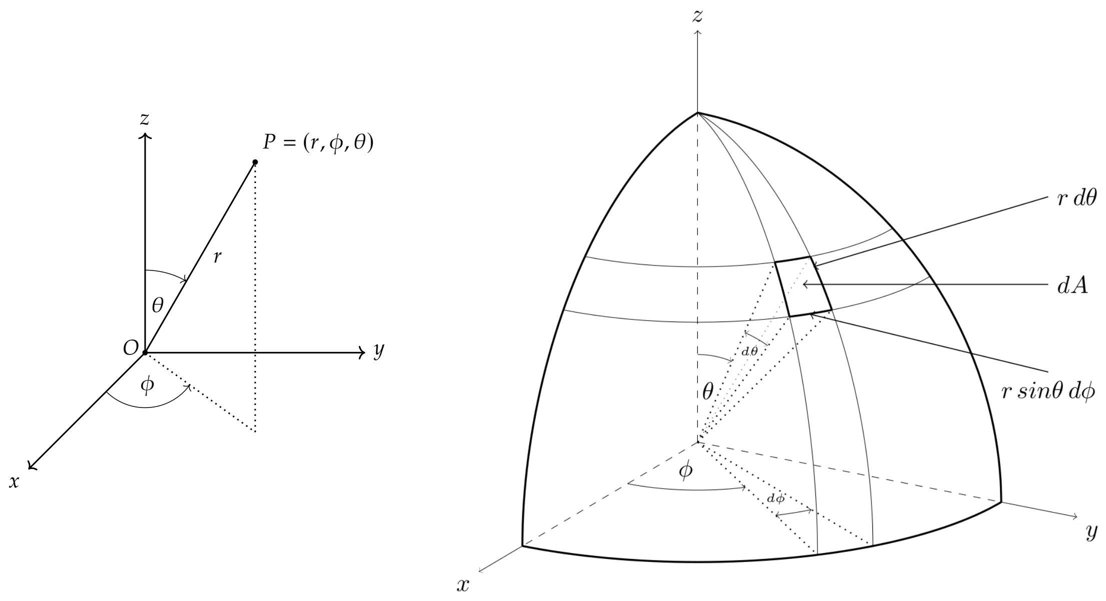


Figure 2.4: Left: Spherical coordinates. Right: The differential solid angle.

close to the poles. The differential area enclosed by two pairs of these arcs is therefore $dA = r^2\ \sin\theta\ d\theta\ d\phi$. Using Equation 2.6, the differential solid angle $d\omega$ subtended at the centre of the sphere is

$$d\omega = \frac{dA}{r^2} = \sin\theta\ d\theta\ d\phi\ \ [\mathrm{sr}]. \tag{2.8}$$

The differential solid angle subtended at the centre of a sphere by an arbitrarily oriented differential area dA that is a distance r away from the centre is

$$d\omega = \frac{dA\ \cos\gamma}{r^2}\ \ [\mathrm{sr}], \tag{2.9}$$

where γ is the angle between the differential area's normal and the direction from the differential area to the centre of the sphere. The $\cos\gamma$ term serves to "foreshorten" the differential area if it is not oriented perpendicularly to the line connecting it to the centre of the sphere.

Radiant Intensity

Radiant intensity (I) is radiant power per unit solid angle. It describes the directional density of radiant power.

$$I = \lim_{\Delta\omega\to 0}\frac{\Delta\Phi}{\Delta\omega} = \frac{d\Phi}{d\omega}\ \ [\mathrm{W\,sr^{-1}}]. \tag{2.10}$$

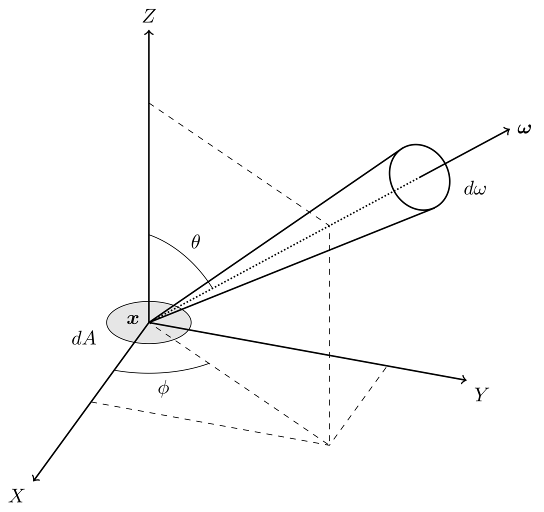


Figure 2.5: Geometric visualisation of radiance.

Radiance

Radiance (L) is the radiometric quantity that is detected by a sensor such as a camera or the eye. It describes the radiant energy within a light ray or a thin pencil of light rays and can be measured at a point in space or on a surface; the terms field radiance and surface radiance are used to distinguish between the two. Radiance is a function of position $\boldsymbol{x}$ and direction $\boldsymbol{\omega}(\phi, \theta)$. It is defined as radiant power per unit solid angle per unit projected area:

$$L(\boldsymbol{x}, \boldsymbol{\omega}) = \lim_{\substack{\Delta\omega \to 0 \\ \Delta A \to 0}} \frac{\Delta\Phi}{\Delta\omega\ \Delta A\ \cos\theta} = \frac{d^2\Phi}{d\omega\ dA\ \cos\theta}\ [\mathrm{W\ m^{-2}\ sr^{-1}}]. \tag{2.11}$$

The terms in the definition are illustrated in Figure 2.5. The illustration applies equally to exitant (outgoing) radiance (radiant power leaving a differential area dA centred at $\boldsymbol{x}$ in the direction $\boldsymbol{\omega}$ within a differential solid angle $d\omega$) as well as incident (incoming) radiance (radiant power arriving on a differential area dA centred at $\boldsymbol{x}$ from the direction $\boldsymbol{\omega}$ within a differential solid angle $d\omega$).

The cosine term accounts for the spread of radiant power over a larger area when the surface is not perpendicular to $\boldsymbol{\omega}$. Consider a beam of light of width l incident on a surface that is perpendicular to the rays' direction (Figure 2.6(a)). The rays hit an area of length l. Now consider the same beam hitting the surface at an angle $\theta \in (0, \frac{\pi}{2}]$ (Figure 2.6(b)). The rays now hit a region of length $\frac{l}{\cos\theta}$, which is larger than l since $\cos\theta \in [0, 1)$, so the energy contained in the rays is more spread out. An area of length

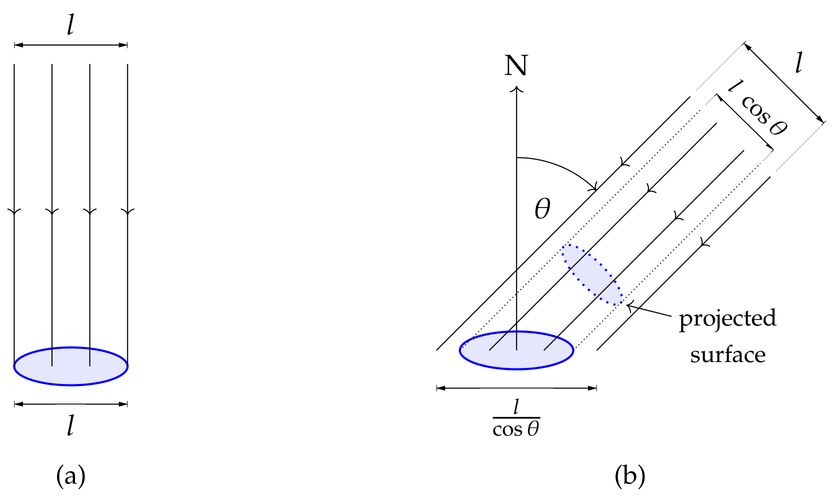


Figure 2.6: Effect of the cosine term in the definition of radiance.

l is also shown, together with its projection. The projected area has a length of $l\ \cos\theta$, which is smaller than l, indicating that fewer rays within the beam hit the area.

Radiance describes the directional and areal distribution of radiant power, making it the most useful radiometric quantity. All the other quantities can be derived from it. For example, from Equation 2.11,

$$L(\boldsymbol{x}, \boldsymbol{\omega})\ \cos\theta = \frac{d^2\Phi}{d\omega\, dA} \tag{2.12}$$

$$= \frac{d}{d\omega}\left(\frac{d\Phi}{dA}\right). \tag{2.13}$$

Integrating w.r.t. solid angle yields irradiance:

$$\int_{\Omega^+} L(\boldsymbol{x}, \boldsymbol{\omega})\ \cos\theta\ d\omega = \int_{\Omega^+} \frac{d}{d\omega}\left(\frac{d\Phi}{dA}\right)\ d\omega \tag{2.14}$$

$$= \frac{d\Phi}{dA} \tag{2.15}$$

$$= E(\boldsymbol{x}). \tag{2.16}$$

Irradiance can therefore be expressed as the sum of cosine-weighted radiance incident from all directions in the hemisphere's solid angle. Similarly, integrating w.r.t. area yields radiant intensity while integrating w.r.t. solid angle *and* area yields radiant power.

Invariance of Radiance

Consider the setup in Figure 2.7. Two points, $\boldsymbol{x}$ and $\boldsymbol{y}$, are a distance r apart. Suppose a quantity of radiant energy leaves from $\boldsymbol{x}$ in the direction of $\boldsymbol{y}$. Denote this direction

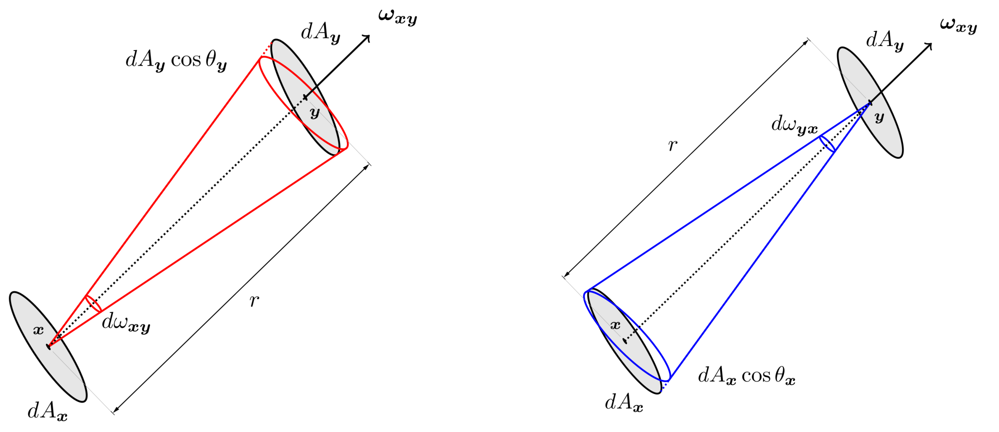


Figure 2.7: Left: Outgoing radiance from $\boldsymbol{x}$. Right: Incoming radiance on $\boldsymbol{y}$.

$\boldsymbol{\omega}_{xy}(\phi_y, \theta_y)$. To physically measure the radiance $L(\boldsymbol{x}, \boldsymbol{\omega}_{xy})$ leaving from $\boldsymbol{x}$, we need to use small but finite areas and solid angles instead of points and directions, which are purely mathematical constructs. Hence we define tiny areas centred around $\boldsymbol{x}$ and $\boldsymbol{y}$ - the differential areas dA_x and dA_y respectively - and a thin cone of directions subtended at $\boldsymbol{x}$ around $\boldsymbol{\omega}_{xy}$, the differential solid angle $d\omega_{xy}$. Since

$$d\omega_{xy} = \frac{dA_y \ \cos\theta_y}{r^2}, \tag{2.17}$$

applying the definition of radiance, and rearranging terms,

$$L(\boldsymbol{x}, \boldsymbol{\omega}_{xy}) = \frac{d^2\Phi_x}{d\omega_{xy} \, dA_x \ \cos\theta_x} = \frac{d^2\Phi_x}{dA_x \, dA_y} \frac{r^2}{\cos\theta_x \ \cos\theta_y}, \tag{2.18}$$

where Φ_x is the radiant power leaving $\boldsymbol{x}$. Therefore,

$$d^2\Phi_x = L(\boldsymbol{x}, \boldsymbol{\omega}_{xy}) \, dA_x \, dA_y \, \frac{\cos\theta_x \ \cos\theta_y}{r^2}. \tag{2.19}$$

Similarly, we can obtain an expression for the incoming radiance at $\boldsymbol{y}$ from the direction $\boldsymbol{\omega}_{xy}(\phi_x, \theta_x)$. This is the same direction as $\boldsymbol{\omega}_{xy}(\phi_y, \theta_y)$ but is now expressed relative to dA_x. Since

$$d\omega_{yx} = \frac{dA_x \ \cos\theta_x}{r^2}, \tag{2.20}$$

$$L(\boldsymbol{y}, \boldsymbol{\omega}_{xy}) = \frac{d^2\Phi_y}{d\omega_{yx} \, dA_y \ \cos\theta_y} = \frac{d^2\Phi_y}{dA_x \, dA_y} \frac{r^2}{\cos\theta_x \ \cos\theta_y}, \tag{2.21}$$

where Φ_y is the radiant power arriving at $\boldsymbol{y}$. Therefore,

$$d^2\Phi_y = L(\boldsymbol{y}, \boldsymbol{\omega}_{xy}) \, dA_x \, dA_y \, \frac{\cos\theta_x \ \cos\theta_y}{r^2}. \tag{2.22}$$

If we assume that the space between $\boldsymbol{x}$ and $\boldsymbol{y}$ is a vacuum, no energy is absorbed or scattered between the two points. By the law of conservation of energy, $d^2\Phi_x = d^2\Phi_y$. Therefore, from Equation 2.18 and Equation 2.21,

$$L(\boldsymbol{x}, \boldsymbol{\omega}_{xy})\, dA_x\, dA_y\, \frac{\cos\theta_x\ \cos\theta_y}{r^2} = L(\boldsymbol{y}, \boldsymbol{\omega}_{xy})\, dA_x\, dA_y\, \frac{\cos\theta_x\ \cos\theta_y}{r^2}, \tag{2.23}$$

implying

$$L(\boldsymbol{x}, \boldsymbol{\omega}_{xy}) = L(\boldsymbol{y}, \boldsymbol{\omega}_{xy}). \tag{2.24}$$

This result shows that in vacuum, radiance is invariant along straight lines and is not affected by distance.

Throughput

Rearranging the terms in the definition of radiance (Equation 2.11), we obtain an expression for the differential power propagated by a ray with radiance L as a function of position (at the point $\boldsymbol{x}$ on a differential area dA) and direction (within the differential solid angle $d\omega$ subtended at $\boldsymbol{x}$ around the direction $\boldsymbol{\omega}$):

$$d^2\Phi(\boldsymbol{x}, \boldsymbol{\omega}) = L(\boldsymbol{x}, \boldsymbol{\omega})\ \cos\theta\ d\omega\ dA. \tag{2.25}$$

The total power propagated by a beam of rays through an area A and within a solid angle Ω is then

$$\Phi = \int_A \int_\Omega L(\boldsymbol{x}, \boldsymbol{\omega})\ \cos\theta\ d\omega\ dA. \tag{2.26}$$

If the radiance is constant for all the rays within the beam we can write:

$$\Phi = L \cdot \int_A \int_\Omega \cos\theta\ d\omega\ dA. \tag{2.27}$$

Setting

$$G = \int_A \int_\Omega \cos\theta\ d\omega\ dA \tag{2.28}$$

we obtain

$$\Phi = L \cdot G. \tag{2.29}$$

G is called the geometrical extent or the throughput of the beam of rays. It is a purely geometric quantity that describes the power-carrying capacity of a beam of rays. From Equation 2.28, the differential throughput is

$$d^2G = \cos\theta\ d\omega\ dA. \tag{2.30}$$

Equation 2.25 can now be written more concisely as

$$d^2\Phi(\boldsymbol{x}, \boldsymbol{\omega}) = L(\boldsymbol{x}, \boldsymbol{\omega}) \cdot d^2G \tag{2.31}$$

and radiance can be expressed as differential power per differential throughput:

$$L(\boldsymbol{x}, \boldsymbol{\omega}) = \frac{d^2\Phi(\boldsymbol{x}, \boldsymbol{\omega})}{d^2G}. \tag{2.32}$$

Assuming a uniform unchanging medium and perfect energy conservation, it can be shown that throughput (or differential throughput) is an invariant quantity.

2.2 Reflectance Models

The appearance of a non-emissive material is due to how it interacts with incident light. Some of the light is absorbed and the rest is scattered. Reflection, refraction, dispersion, diffraction, polarisation, fluorescence, and phosphorescence are all forms of scattering. If a material reflects red wavelengths and absorbs all other wavelengths, the material is perceived as having a red colour.

Ideal diffuse materials, called Lambertian materials, reflect light uniformly in all directions. Due to this property these materials are said to be isotropic; they look the same when viewed from any direction. Lambertian materials are theoretical but they are a reasonable approximation for reflection off matte surfaces such as stone walls. Perfectly specular materials reflect or transmit light in a single direction. Their appearance is affected by the view direction and they are said to be anisotropic. These materials are also theoretical. They are used to model perfect mirrors or perfectly translucent surfaces.

The reflectance properties of materials are modelled by a bidirectional surface scattering reflectance distribution function (BSSRDF), S, a four-dimensional function that relates outgoing differential radiance from a point $\boldsymbol{x}$ in direction ω_o to incident differential power from direction ω_i at a point $\boldsymbol{y}$:

$$S(\boldsymbol{x}, \boldsymbol{\omega_o}, \boldsymbol{y}, \boldsymbol{\omega_i}) = \frac{dL_o(\boldsymbol{x}, \boldsymbol{\omega_o})}{d\Phi(\boldsymbol{y}, \boldsymbol{\omega_i})}. \tag{2.33}$$

The BSSRDF is suitable for modelling all kinds of materials, including translucent ones such as paper and skin. In these materials, light enters at a point but exits from a different point. If subsurface light transport is ignored and light is assumed to enter and exit at the same point, a simpler model can be used. This is the bidirectional reflectance distribution function (BRDF), f_r. Given a point $\boldsymbol{x}$ on a surface, the BRDF relates reflected differential radiance dL_o in direction $\boldsymbol{\omega_o}$ to differential irradiance dE from direction $\boldsymbol{\omega_i}$:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega_o}, \boldsymbol{\omega_i}) = \frac{dL_o(\boldsymbol{x}, \boldsymbol{\omega_o})}{dE(\boldsymbol{x}, \boldsymbol{\omega_i})}. \tag{2.34}$$

In this thesis the term BRDF will be used for both reflectance and transmittance. Some texts use the term BSDF (bidirectional scattering distribution function) for this purpose, and indicate reflectance or transmittance with the terms BRDF and BTDF (bidirectional transmittance distribution function) respectively.

For the BRDF to be physically plausible, it needs to satisfy several properties. First, its value cannot be negative:

$$f_r \geq 0. \tag{2.35}$$

Second, the Helmholtz reciprocity condition must hold. This condition states that if the directions passed into the BRDF were reversed, the value of the function would not change:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega_o}, \boldsymbol{\omega_i}) = f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o}). \tag{2.36}$$

Third, energy must be conserved. Therefore, for any incident direction $\boldsymbol{\omega_i}$,

$$\int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_o}, \boldsymbol{\omega_i}) \, \cos\theta_o \, d\omega_o \leq 1,$$

where Ω is the sphere of directions around $\boldsymbol{x}$.

The Lambertian BRDF

The Lambertian BRDF is

$$f_r = \frac{\rho_d}{\pi}, \tag{2.37}$$

where ρ_d is the diffuse reflectance or the *albedo* of the surface and represents the fraction of reflected light. f_r is constant for all points on the surface, for all directions.

The Phong BRDF

Glossy materials exhibit non-ideal specular reflection, reflecting light towards a particular direction. These materials were originally modelled using the Phong reflectance model (Phong, 1975), an empirical (not physically based) model. The Phong BRDF is

$$f_r = \rho_s(\boldsymbol{R} \cdot \boldsymbol{V})^n, \tag{2.38}$$

where $\rho_s \in [0, 1]$ is the specular reflectance of the material, representing the fraction of specularly reflected light, $\boldsymbol{R}$ is the reflection direction, $\boldsymbol{V}$ is the viewer direction and $n \geq 1$ is the specular reflection exponent. For a perfectly reflective material, n would be ∞. If $\boldsymbol{L}$ is the light direction and $\boldsymbol{N}$ is the surface normal, $\boldsymbol{R}$ is calculated by reflecting $\boldsymbol{L}$ about $\boldsymbol{N}$,

$$\boldsymbol{R} = 2\boldsymbol{N}(\boldsymbol{N} \cdot \boldsymbol{L}) - \boldsymbol{L}, \tag{2.39}$$

giving

$$f_r = \rho_s((2\boldsymbol{N}(\boldsymbol{N} \cdot \boldsymbol{L}) - \boldsymbol{L}) \cdot \boldsymbol{V})^n. \tag{2.40}$$

The Blinn-Phong BRDF

The Blinn-Phong BRDF (Blinn, 1977), another empirical model, is a slight improvement of the Phong BRDF. It reformulates the Phong BRDF in terms of the vector $\boldsymbol{H}$, which lies half way between $\boldsymbol{L}$ and $\boldsymbol{V}$,

$$\boldsymbol{H} = \frac{\boldsymbol{L} + \boldsymbol{V}}{|\boldsymbol{L} + \boldsymbol{V}|}. \tag{2.41}$$

The Blinn-Phong BRDF is

$$f_r = \rho_s(\boldsymbol{N} \cdot \boldsymbol{H})^n. \tag{2.42}$$

Microfacet Models

Physically based models are commonly based on microfacet theory, where a surface's microgeometry is represented as tiny planar facets, perfectly reflecting or refracting light. If the surface is smooth, the facets are aligned. If the surface is rough, the facets are misaligned, with the facet normals following some statistical distribution. In general, microfacet models use the principles of geometrical optics and are defined by a Fresnel term F, a facet normal distribution D and a geometry term G, which represents how the facets mask or shadow each other. Some microfacet models, such as the He-Torrance-Sillion-Greenberg model (He et al., 1991), use wave optics. These models are more complete since they can simulate a wider range of surface effects. However, they require more computation and hence microfacet models that use geometrical optics are preferred.

The first physically based microfacet model was the Cook-Torrance model (Cook and Torrance, 1982):

$$f_r = \frac{F}{\pi} \frac{D \cdot G}{(\boldsymbol{N} \cdot \boldsymbol{L})(\boldsymbol{N} \cdot \boldsymbol{V})}. \tag{2.43}$$

Ward (1992) designed a model that supports anisotropic reflection. The Oren-Nayar model is used for rough, diffuse surfaces (Oren and Nayar, 1994). The GGX model of Walter et al. (2007) is used for rough transparent materials such as ground glass or frosted glass. Neumann's model is used for metallic surfaces; it is physically plausible but not physically based (Neumann et al., 1999).

2.3 The Rendering Equation

The rendering equation (Kajiya, 1986) is a mathematical formulation for global illumination. Essentially, $L_o(\boldsymbol{x}, \boldsymbol{\omega_o})$, the radiance leaving a point $\boldsymbol{x}$ on a surface in the direction $\boldsymbol{\omega_o}$, can be broken down into two components, emitted radiance $L_e(\boldsymbol{x}, \boldsymbol{\omega_o})$ and reflected radiance $L_r(\boldsymbol{x}, \boldsymbol{\omega_o})$,

$$L_o(\boldsymbol{x}, \boldsymbol{\omega_o}) = L_e(\boldsymbol{x}, \boldsymbol{\omega_o}) + L_r(\boldsymbol{x}, \boldsymbol{\omega_o}), \tag{2.44}$$

where "reflected radiance" should be interpreted as referring to both reflected and transmitted radiance.

The emitted radiance is non-zero only if the surface is a light source. We can readily compute this value if we know the details of the light source: its shape, power and so on. The evaluation of the reflected radiance is not as straightforward because we need to determine where the light is coming from before it is reflected; we need to locate the original emitter of that light. The light could be coming from anywhere in the hemisphere of directions centred at the point in question. Moreover, the light may not be coming directly from a light source. It may have taken an indirect path by reflecting off other surfaces any number of times. Therefore, we need to take into consideration all the directions that light is coming from, and we also need to repeat all the computations recursively at all the other surface points along the light path. This computation is expressed by the equation

$$L_r(\boldsymbol{x}, \boldsymbol{\omega_o}) = \int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_i(\boldsymbol{x}, \boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i}. \tag{2.45}$$

Ω^+ represents the hemisphere of directions above $\boldsymbol{x}$. $L_i(\boldsymbol{x}, \boldsymbol{\omega_i})$ is the incident radiance arriving at $\boldsymbol{x}$ from the direction $\boldsymbol{\omega_i}$, one of the directions within Ω^+. $f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})$ is the BRDF. θ_i is the angle between $\boldsymbol{\omega_i}$ and the surface normal at $\boldsymbol{x}$. $(\cos\theta_i)^+$ is the foreshortening term, that is, $\cos\theta_i$ clamped to a minimum value of 0.

Incident radiance can be rewritten as the outgoing radiance from another point $\boldsymbol{x'}$ in the scene,

$$L_i(\boldsymbol{x}, \boldsymbol{\omega_i}) = L_o(\boldsymbol{x'}, -\boldsymbol{\omega_i}), \tag{2.46}$$

and $\boldsymbol{x'}$ can be related to $\boldsymbol{x}$ by means of the ray casting operator r which identifies the first surface point hit by a ray originating at $\boldsymbol{x}$ and going in the direction $\boldsymbol{\omega_i}$:

$$L_i(\boldsymbol{x}, \boldsymbol{\omega_i}) = L_o(r(\boldsymbol{x}, \boldsymbol{\omega_i}), -\boldsymbol{\omega_i}). \tag{2.47}$$

By substituting for L_i in Equation 2.45 we get

$$L_r(\boldsymbol{x}, \boldsymbol{\omega_o}) = \int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_o(r(\boldsymbol{x}, \boldsymbol{\omega_i}), -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i}, \tag{2.48}$$

and by substituting for L_r in Equation 2.44 and dropping the subscript from L_o we obtain the hemispherical formulation (or the angular form) of the rendering equation,

$$L(\boldsymbol{x}, \boldsymbol{\omega_o}) = L_e(\boldsymbol{x}, \boldsymbol{\omega_o}) + \int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L(r(\boldsymbol{x}, \boldsymbol{\omega_i}), -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i}. \tag{2.49}$$

The notation can be simplified and expressed more concisely and intuitively by using the transport operator T defined as

$$TL(\boldsymbol{x}, \boldsymbol{\omega_o}) \equiv \int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L(r(\boldsymbol{x}, \boldsymbol{\omega_i}), -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i}. \tag{2.50}$$

Using this operator, Equation 2.49 becomes

$$L = L_e + TL \tag{2.51}$$

$$(I - T)L = L_e \tag{2.52}$$

$$L = (I - T)^{-1} L_e \tag{2.53}$$

where I is the identity operator. Since $(I - T)^{-1}$ can be expanded as a Neumann series,

$$(I - T)^{-1} = 1 + T + T^2 + T^3 + \cdots = \sum_{i=0}^{\infty} T^i, \tag{2.54}$$

Equation 2.51 can be rewritten as

$$L = L_e + TL_e + T^2 L_e + T^3 L_e + \cdots = \sum_{i=0}^{\infty} T^i L_e. \tag{2.55}$$

Equation 2.51 is the operator form of the rendering equation. In the expanded operator form (Equation 2.55) the rendering equation can be readily interpreted as a sum of emitted light (L_e), light that was reflected once (TL_e) and light that was reflected two or more times ($T^2L_e + T^3L_e + \dots$) Interpreting these terms relative to the eye, L_e is light reaching the eye directly from a light source; a light source is directly visible. TL_e is light that bounces off one surface before reaching the eye; a surface that is directly lit is in view. This term corresponds to *direct illumination*. The higher order terms correspond to light that bounces off surfaces two or more times before reaching the eye. This *indirect illumination* is the reason why surfaces that are not directly lit do not appear completely black.

2.3.1 The Area Formulation

The rendering equation (Equation 2.49) can be written as an integral in terms of areas instead of solid angles. Since $d\boldsymbol{\omega_i}$, the differential solid angle subtended at point $\boldsymbol{x}$ in the direction of point $\boldsymbol{x'}$, can be written as

$$d\boldsymbol{\omega_i} = \frac{dA_{\boldsymbol{x'}}\ \cos\theta_{\boldsymbol{x'}}}{||\boldsymbol{x}-\boldsymbol{x'}||^2}, \tag{2.56}$$

the rendering equation becomes

$$L(\boldsymbol{x},\boldsymbol{\omega_o}) = L_e(\boldsymbol{x},\boldsymbol{\omega_o}) + \int_A f_r(\boldsymbol{x},\boldsymbol{\omega_i},\boldsymbol{\omega_o})\ L(\boldsymbol{x'},-\boldsymbol{\omega_i})\ \cos\theta_{\boldsymbol{x}}\ \frac{dA_{\boldsymbol{x'}}\ \cos\theta_{\boldsymbol{x'}}}{||\boldsymbol{x}-\boldsymbol{x'}||^2}\ V \tag{2.57}$$

where $\boldsymbol{x'} = r(\boldsymbol{x},\boldsymbol{\omega_i})$ and A is the area of all the surfaces in the scene. A visibility term V is added to ensure that only those points that are directly visible from $\boldsymbol{x}$ are included in the integral. It is defined as

$$V(\boldsymbol{x},\boldsymbol{x'}) = \begin{cases} 1, \text{if } \boldsymbol{x} \text{ and } \boldsymbol{x'} \text{ are mutually visible} \\ 0, \text{otherwise.} \end{cases} \tag{2.58}$$

Rearranging the terms yields the area formulation

$$L(\boldsymbol{x},\boldsymbol{\omega_o}) = L_e(\boldsymbol{x},\boldsymbol{\omega_o}) + \int_A f_r(\boldsymbol{x},\boldsymbol{\omega_i},\boldsymbol{\omega_o})\ L(\boldsymbol{x'},-\boldsymbol{\omega_i})\ G\ dA_{\boldsymbol{x'}} \tag{2.59}$$

where G is the geometry term

$$G(\boldsymbol{x},\boldsymbol{x'}) = V(\boldsymbol{x},\boldsymbol{x'})\ \frac{\cos\theta_{\boldsymbol{x}}\ \cos\theta_{\boldsymbol{x'}}}{||\boldsymbol{x}-\boldsymbol{x'}||^2}. \tag{2.60}$$

2.3.2 The Path Integral Formulation

Veach (1998) derived another formulation for the rendering equation, expressing it in path space as

$$I_j = \int_\Omega f_j(\bar{x})\ d\mu(\bar{x}). \tag{2.61}$$

The integration domain Ω is the set of light paths of any length (the minimum length of a path is one segment). $\bar{x}$ represents a path. For example, a three-segment path is represented as

$$\bar{x} = \boldsymbol{x_0} \longrightarrow \boldsymbol{x_1} \longrightarrow \boldsymbol{x_2} \longrightarrow \boldsymbol{x_3}. \tag{2.62}$$

x_0 is a point on a light source and x_3 may be a point on a sensor such as the eye. $d\mu(\bar{x})$ is the measure. It is a product measure in this case and is defined for every path of finite length $k, 1 \leq k < \infty$, as

$$d\mu_k(\boldsymbol{x_0} \longrightarrow \boldsymbol{x_1} \longrightarrow \dots \longrightarrow \boldsymbol{x_k}) = dA_{x_0}\ dA_{x_1}\ \dots\ dA_{x_k}. \tag{2.63}$$

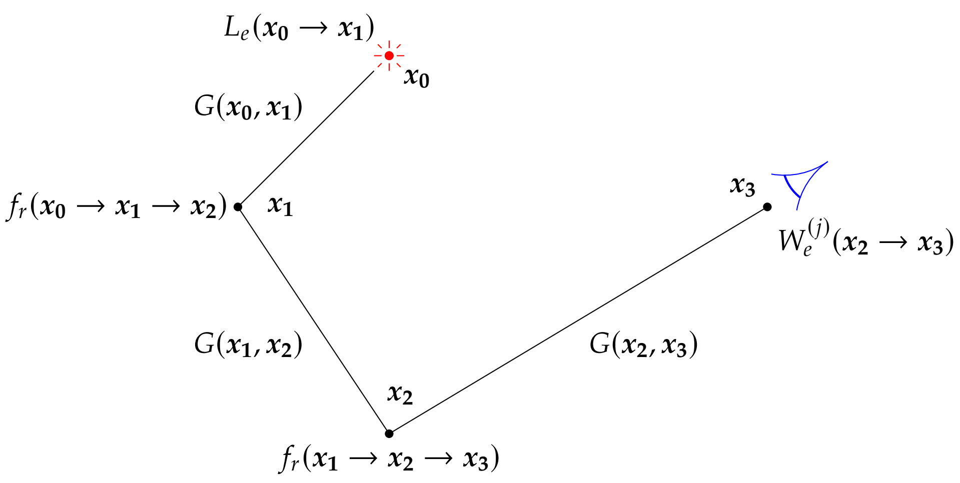


Figure 2.8: The terms of the measurement contribution function.

The integrand f_j is called the measurement contribution function. It is a product of terms. In the three-segment example shown in Figure 2.8 it is defined as

$$\begin{aligned} f_j(\bar{x}) = L_e(x_0 \to x_1) \cdot G(x_0, x_1) \cdot f_r(x_0 \to x_1 \to x_2) \cdot G(x_1, x_2) \cdot \\ f_r(x_1 \to x_2 \to x_3) \cdot G(x_2, x_3) \cdot W_e^{(j)}(x_2 \to x_3). \end{aligned} \tag{2.64}$$

$W_e^{(j)}(x_2 \to x_3)$ represents the importance of the last segment of the path. This term would be zero if the point x_3 was not on a sensor. Using this formulation, light paths are sampled instead of solid angles or areas. Paths are generated randomly and the integral is evaluated with Monte Carlo (MC) methods. The probability density function (PDF) used encapsulates all the random choices taken along the entire path.

The path integral formulation introduces a paradigm shift in the computation of light transport. Instead of solving a recursive integral equation, it requires the computation of an integral. This facilitates the computation to some extent since a variety of mathematical methods for computing integrals can be used. On the other hand, choosing suitable paths is not trivial. This formulation is well-suited for methods such as bidirectional path tracing (Lafortune and Willems, 1993; Veach and Guibas, 1995) and Metropolis light transport (Veach and Guibas, 1997) where the choice of paths used plays a fundamental role in the algorithm's design.

2.4 Monte Carlo Integration

In Monte Carlo integration, the value of an integral is obtained by solving an equivalent problem, the computation of an expected value. The expected value $E(X)$, or equivalently the mean value μ, of a continuous random variable X, is defined as

$$E[X] \equiv \mu \equiv \int_{-\infty}^{\infty} x\, p(x)\, dx, \tag{2.65}$$

where $p(x)$, the PDF of x, satisfies the property

$$p(x) \geq 0,\ \forall x \in \mathbb{R}, \tag{2.66}$$

and the normalisation property

$$\int_{-\infty}^{\infty} p(x)\, dx = 1. \tag{2.67}$$

Moreover, if $g(X)$ is a function of the random variable X, it is also a random variable and

$$E[g(X)] = \int_{-\infty}^{\infty} g(x)\, p(x)\, dx. \tag{2.68}$$

The variance of X is defined as

$$\mathrm{Var}[X] \equiv E[(X - E[X])^2] = E[X^2] - (E[X])^2, \tag{2.69}$$

and similarly, the variance of $g(X)$ is

$$\mathrm{Var}[g(X)] = E[g^2(X)] - (E[g(X)])^2. \tag{2.70}$$

Let $X_1, X_2, ..., X_N$ be N independent and identically distributed random variables with PDF $p(x)$. Let g be a function that operates on the X_i random variables. Using the constants w_i we can construct a function G as a linear combination of the random variables $g(X_1), g(X_2), ..., g(X_N)$,

$$G = \sum_{i=1}^{N} w_i\, g(X_i). \tag{2.71}$$

The expected value of G is

$$E[G] = E[\sum_{i=1}^{N} w_i\, g(X_i)] \tag{2.72}$$

$$= \sum_{i=1}^{N} w_i\, E[g(X_i)]. \tag{2.73}$$

By setting each w_i to $\frac{1}{N}$, G becomes the arithmetic average of the $g(X_i)$ random variables and we obtain

$$E[G] = \frac{1}{N} \sum_{i=1}^{N} E[g(X_i)] \tag{2.74}$$

$$= E[g(X)]. \tag{2.75}$$

Since the mean of G is equal to the mean of $g(X)$, G is said to be an estimator of $E[g(X)]$.

Suppose we needed to evaluate the integral

$$I = \int_{\Omega} f(x)\, dx, \tag{2.76}$$

where Ω is the domain and I is the result of evaluating the integral. Consider the estimator $\langle I \rangle$,

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}, \tag{2.77}$$

where x_i is a sample from the domain, selected with PDF $p(x)$. The expected value of $\langle I \rangle$ is

$$\begin{aligned} E[\langle I \rangle] &= E[\frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}] && (2.78)\\ &= \frac{1}{N} \sum_{i=1}^{N} E[\frac{f(x_i)}{p(x_i)}] && (2.79)\\ &= \frac{1}{N} N \int_{\Omega} \frac{f(x)}{p(x)}\, p(x)\, dx && (2.80)\\ &= \int_{\Omega} f(x)\, dx && (2.81)\\ &= I. && (2.82) \end{aligned}$$

In other words, the mean value of the estimator is equal to the needed quantity. Therefore, by picking a large number of independent random variables $X_1, X_2, ..., X_n$ from the domain using the PDF $p(x)$, we can compute many values for $\frac{f(X_i)}{p(x)}$ and by averaging these values we obtain the mean of the estimator and hence the value of the integral I.

For example, computing direct illumination from an area light using the area formulation of the rendering equation requires evaluating the integral

$$\int_{A} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_e(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, V\, \frac{\cos\theta_{\boldsymbol{x}}\, \cos\theta_{\boldsymbol{x'}}}{||\boldsymbol{x} - \boldsymbol{x'}||^2}\, dA. \tag{2.83}$$

Using Monte Carlo integration, this integral is estimated as

$$\begin{aligned} &\frac{1}{N} \sum_{i=1}^{N} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_e(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, V\, \frac{\cos\theta_{\boldsymbol{x}}\, \cos\theta_{\boldsymbol{x'}}}{||\boldsymbol{x} - \boldsymbol{x'}||^2} \cdot \frac{1}{p(\boldsymbol{x'})} && (2.84)\\ = &\frac{A}{N} \sum_{i=1}^{N} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_e(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, V\, \frac{\cos\theta_{\boldsymbol{x}}\, \cos\theta_{\boldsymbol{x'}}}{||\boldsymbol{x} - \boldsymbol{x'}||^2} && (2.85) \end{aligned}$$

where A is the area of light source and $p(\boldsymbol{x'}) = \frac{1}{A}$ is the probability of choosing point $\boldsymbol{x'}$ on the light source.

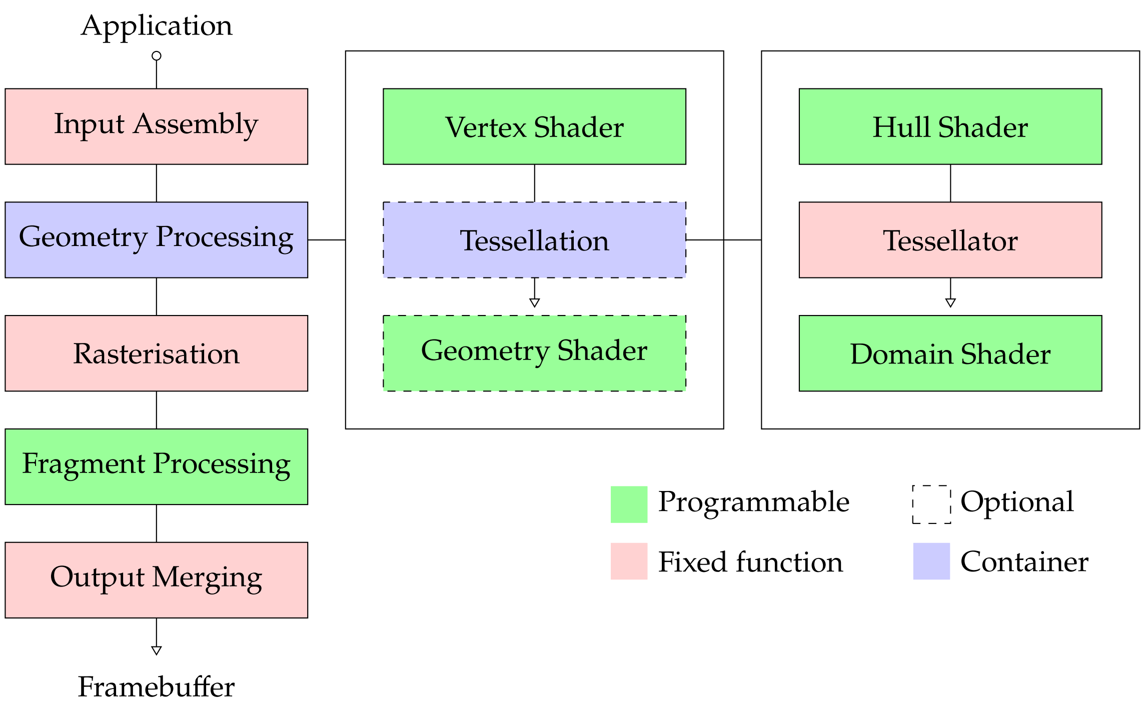


Figure 2.9: The rasterisation pipeline.

2.5 The Rasterisation Pipeline

The rasterisation pipeline is a sequence of stages, some programmable, that is used to render graphics in real time. It operates like an assembly line, with each stage feeding the next. A high-level view of the pipeline is shown in Figure 2.9. A CPU-based application configures the pipeline and triggers execution by issuing a draw call. All the pipeline stages perform highly parallelisable operations and are implemented on the GPU (Graphics Processing Unit). After each run or pass, the end result of the last stage is a framebuffer, a rectangular array of pixel colours that will be presented on the display. Depending on the effects desired, multiple passes of the pipeline may be required to generate the final framebuffer for display.

The application sets up all the information needed to render a scene. A specification of the vertices that make up the geometric primitives within the scene (usually triangles) are stored in vertex buffers together with other per-vertex information called vertex attributes. Typical vertex attributes are texture coordinates and shading normals. These data together with images (textures), material properties, lighting information, and any other data that are needed to produce the final render are passed to the GPU. This setup is performed once before the first run of the pipeline. In the following runs, the application stage updates vertex information, modifies the viewpoint (the camera's position and orientation), processes animations, detects collisions, handles user input, and other CPU-based tasks. The application can also launch compute shaders, to perform general purpose GPU-based work.

The first stage of the pipeline, input assembly, constructs vertices from the provided vertex buffers in the layout required by the next stage. The type of primitives being rendered (typically triangles) is also determined at this stage.

The next few stages are responsible for geometry processing. The vertex shader is the first programmable stage. It is called once per vertex, transforming the vertex from model space (also called local space and object space) to world space using a model matrix M, to eye space (also called camera space and view space) using a view matrix V, to clip space using a projection matrix P,

$$P \times V \times M \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}. \tag{2.86}$$

The most commonly used projections are the perspective and orthographic (parallel) projections. The tessellation and geometry shader stages are optional. They can be used to refine geometry and to create new geometric primitives. Note that tessellation consists of a hull shader, a tessellator, and a domain shader. Optionally, the primitives generated during geometry processing may be output (streamed out) to a buffer; this buffer could then be used as an input for a future run of the pipeline.

In the rasterisation stage, primitives are culled (back face culling and primitive culling) and clipped (primitives falling outside the viewing cube or frustum are discarded, primitives partially outside are split into new primitives). Next, vertices are transformed to normalised device coordinates (NDC) space using perspective division,

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ w_{ndc} \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \\ \frac{1}{w_c} \end{pmatrix}, \tag{2.87}$$

and finally to window space (also called screen space and pixel space) using a viewport transform,

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \end{pmatrix} \xrightarrow[\text{transform}]{\text{viewport}} \begin{pmatrix} x_w \\ y_w \end{pmatrix}. \tag{2.88}$$

The rasterisation stage identifies the pixels that potentially make up the geometric primitive (scan conversion plus scissor test) and interpolates the vertex attributes. This stage produces fragments. A fragment consists of all the data needed to shade a pixel and to test whether the fragment should be discarded. Barycentric coordinates for each pixel are generated and perspective-correct interpolation is performed. The depth value is also interpolated.

The fragment processing stage is programmable. It is called once per fragment and is responsible for shading a pixel using the interpolated data received from the previous stage. Lighting calculations are usually performed in this stage.

The output merging stage performs visibility tests (depth, stencil, etc.) and outputs the final pixel colour by blending with the colour currently in the framebuffer, possibly also applying raster operations (ROPs) in the process. Data from other render targets can also be used for colour blending.

2.5.1 Forward and Deferred Rendering

The rasterisation pipeline operates in object order. This means that all the triangle primitives in a scene are processed in turn. For each triangle, the screen pixels that make it up are identified and shaded. The z-buffer is also populated during this procedure, so if a pixel has a higher depth value than that stored in the z-buffer, calculations for that pixel are avoided. Depending on the depth order of the triangles, a particular pixel may still be processed more than once. This is known as overdraw; avoiding it reduces computation costs.

The lighting calculations used to shade pixels are quite expensive. In forward rendering there is typically a pass for each light source, that is, for each pixel, lighting calculations are performed as many times as there are lights. The implication is that when there are many lights, wasted computation due to the overdraw problem adds up drastically. Deferred shading mitigates the problem by performing visibility and shading in separate passes, in that order. The visibility pass consists of the usual per-primitive pixel-identification process and z-buffer population, but no shading is performed and geometry and material properties are collected at every pixel and stored in G-buffers (geometric buffers) (Saito and Takahashi, 1990). The shading pass uses the information stored in the G-buffers and computes illumination for a pixel only once, saving computation. Note that overdraw is still present because the geometry and material information collected at a pixel may be overwritten multiple times. However, the cost of this form of overdraw is negligible when compared to the savings obtained for lighting calculations.

2.6 Image-Based Rendering

Image-based rendering (IBR) techniques synthesise their output from a set of supplied images rather than from geometric primitives. Nevertheless, if available, geometric information can be used to complement the process. Shum and Kang (2000) classify IBR techniques into a continuum, depending on how much geometric information is used. Three broad categories are identified. At one end of the spectrum are techniques that do not use any geometric information at all. An example for this category is the construction of a panoramic view from a set of images. In the middle of the spectrum are techniques that use implicit geometry, where a correspondence between the given images is supplied. The view morphing technique (Seitz and Dyer, 1996) lies in this category. It requires two images of an object, the projection matrix used for each image, and a mapping from the pixels in the first image to the pixels in the second image that can be constructed semi-automatically. At the other end of the spectrum are techniques that use explicit geometry. This category includes texture mapping and 3D warping, and is the category that we will focus on.

Texture Mapping

Texturing or texture mapping is the process of applying a texture (an image) to a surface. It is a computationally inexpensive way of adding detail to an object. Texture coordinates are added to the information stored per vertex to specify which point on the image should be mapped to that vertex. During rasterisation, intermediate texels (texture pixels) are obtained by interpolation. Texture space is the set of two-dimensional coordinates (u, v) where $u \in [0, 1]$ and $v \in [0, 1]$. However, extensions to texture mapping may give special meaning to components out of these ranges. For example, components greater than 1 typically indicate tiling, where the texture is scaled and repeated for the number of times specified by the component's value.

Perspective-Correct Interpolation

The process of transforming vertices to two-dimensional window space (Section 2.5) includes the perspective division. Assuming a perspective projection, this causes the relationship between pre-NDC space coordinates and window space to become non-linear. Since interpolation of per vertex attributes occurs in window space, care needs to be taken to ensure correctly interpolated results. This is usually taken care of automatically by the API (for example, OpenGL) being used. Some details are provided here as they will be needed in the discussion regarding 3D warping.

A1
A2
A3
A4
B1
B2
B3
B4
C1
C2
C3
C4
D1
D2
D3
D4
Figure 2.10: Perspective correction off (left) and on (right).

Figure 2.10 shows two textured quads that are tilted "into" the page, both using the same texture coordinates and both rendered with the same perspective projection. Texture coordinates were interpolated linearly for the quad on the left, resulting in incorrect results. Figure 2.11 explains the skewed appearance of this quad. The quad is geometrically made up of two triangles, A and B. Expanding each triangle into a parallelogram shows how a different pattern originates for each triangle.

The solution is to interpolate vertex attributes hyperbolically (Blinn, 1992). Once the (x_c, y_c, z_c, w_c) clip space coordinates are obtained, any vertex attributes that need to be interpolated are divided by w_c, and an additional vertex attribute, $1/w_c$, is created. All vertex attributes are then interpolated linearly using the usual mechanism. Finally, in window space, the interpolated attributes are divided by the interpolated $1/w_c$ value. This quotient of linearly interpolated values yields the required hyperbolically interpolated values. The result is shown in Figure 2.10 (right).

3D Warping

Superimposing a Texture onto Geometry

As will become apparent in the following section, a useful image-based rendering technique is the superimposition of a full-screen texture onto a scene's geometry. To project the texture onto the geometry, texture coordinates for a vertex are computed from the vertex's coordinates after they have been transformed into clip space. Computing the perspective division manually on these coordinates transforms them into NDC space.

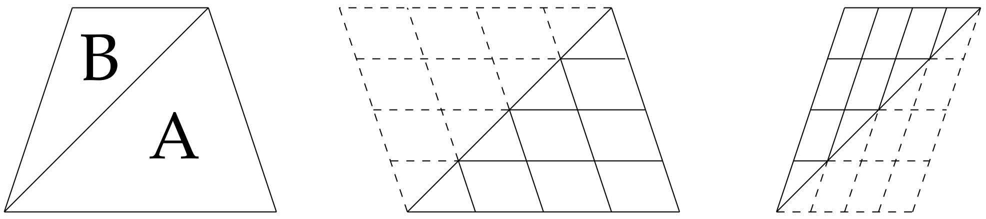


Figure 2.11: Explanation for the appearance of Figure 2.10 (left).

Mapping the x and y NDC components from $[-1, 1]$ to $[0, 1]$ yields the required texture coordinates (u, v) for the vertex.

If the texture is already perspective corrected, this algorithm requires disabling the rendering pipeline's perspective correction. This can be accomplished by multiplying u and v by w_c, the fourth homogeneous coordinate, and using four-component texture coordinates $(u, v, 0, w_c)$. This process cancels out the perspective division. Alternatively, in some APIs, perspective correction can be disabled in an easier way. For example, in OpenGL this is accomplished by specifying the *noperspective* keyword.

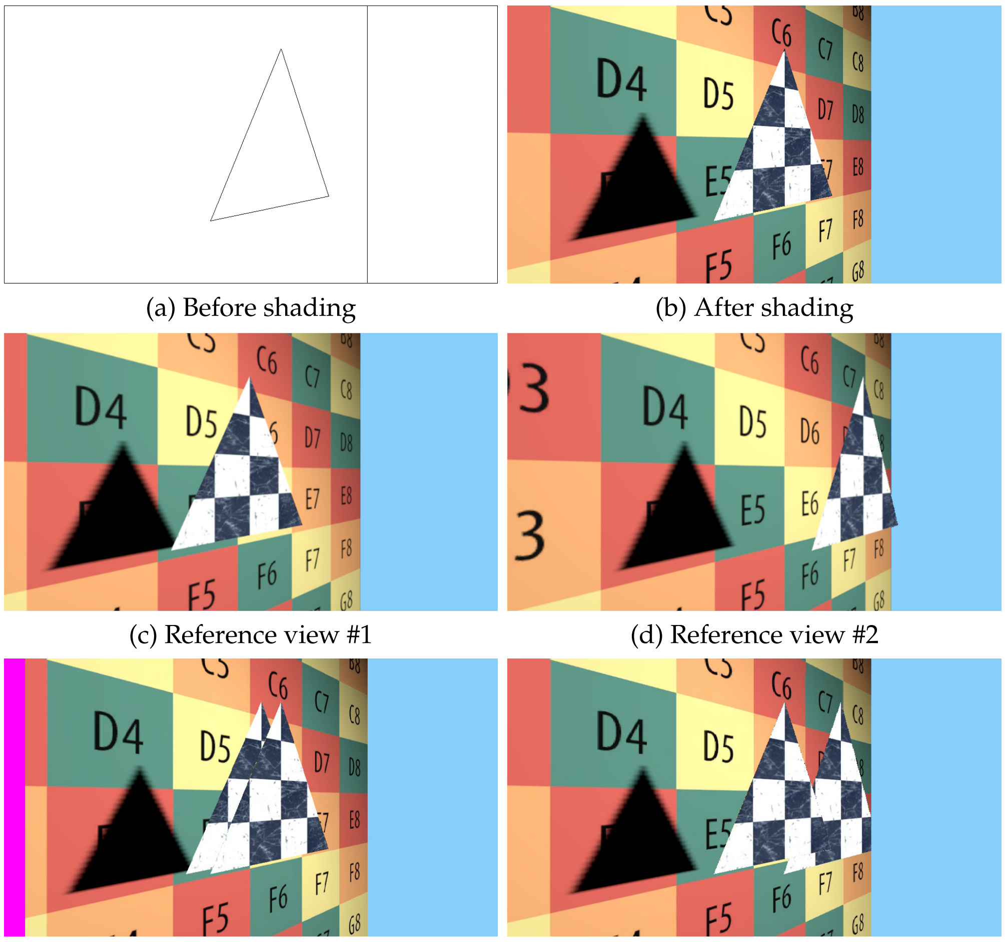
Independent Superimposition

The texture projection technique described in the previous section can be generalised to superimpose a full-screen texture onto geometry that is observed from a different viewpoint. In this case, a different view matrix needs to be used for the texture coordinates. Therefore, instead of using the clip space vertex coordinates computed with the $P \times V \times M$ transform as a starting point, the clip space coordinates are instead computed as $P \times V' \times M$, where V' is the view matrix used to generate the full-screen texture in the first place.

Double Warping

Figure 2.12 illustrates a setup where a scene (a) is shaded to produce (b) from two reference views, View #1 (c) and View #2 (d). The scene consists of a triangle in front of a vertical plane. In the lead up to the required view, the camera was moving sideways, to the left. View #1 is a view of the scene as it appeared a short time in the past. In this view the E5 tile behind the triangle (just below the D5 tile) is mostly occluded, whereas the E7 tile is nearly completely unoccluded. View #2 is a view of the scene obtained by extrapolating camera movement a short time into the future. In this view, the E5 tile is unoccluded whereas the E7 tile is occluded. The required view (b) lies between View #1 and View #2.

3D warping Views #1 and #2 to match the camera angle in (b) yields (e) and (f) respectively. The triangle appears duplicated in (e). Here the triangle on the left is incorrect. That triangle is the result of shading the part of the plane that became unoccluded when warping View #1 to produce (e). The problem occurs because the texture coordinates of the uncovered part of the plane map to the triangle in (c). The magenta region on the left represents missing data. The data needed to shade this region is not present in (c) causing the texture coordinates to be out of range. Two triangles



(a) Before shading (b) After shading

(c) Reference view #1 (d) Reference view #2

(e) Warped view #1 (f) Warped view #2

Figure 2.12: Image-based rendering.

also appear in (f). Here the triangle on the right is incorrect. Texture coordinates for the plane behind this triangle map to the triangle in (d).

Neither of the warped views is by itself sufficient to produce the required view. For instance, it is clear that the E5 tile needs to be obtained from (f) whereas the E7 tile needs to be obtained from (e). By choosing the relevant parts of the image from (e) and (f), for example by comparing depth values for each pixel and choosing the view where the depth value is closer to that of (b), the required view can be reconstructed. Although a perfect reconstruction was obtained in this case, this is not always possible. In complex scenes, new occluders may come into play in the reference views, hiding required data.

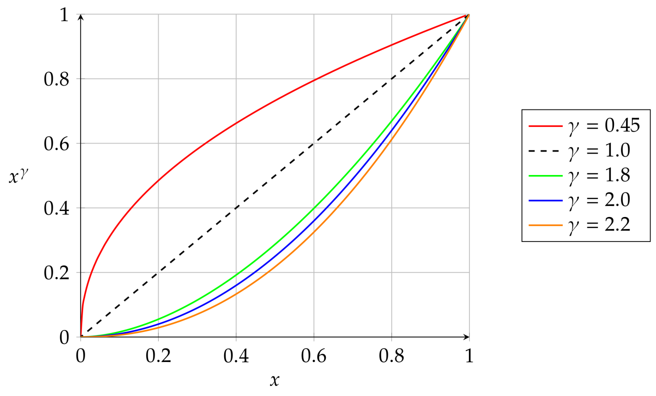


Figure 2.13: Plots for various values of gamma.

Forward and Backward Warping

In forward warping, every pixel in a source image is transformed to a new position, producing a new image. Depending on the transformation, there could be gaps in the new image if no pixels were mapped to these locations. Alternatively, multiple pixels in the source image could map to the same pixel in the new image. Backward warping produces better results. For every pixel in the new image, the corresponding pixel is looked up in the source image by using the inverse transformation. When the source location lies between pixels, interpolation between the neighbouring pixels is used to obtain a colour value.

2.7 Gamma Correction and Tone Mapping

The human eye can distinguish between darker shades better than lighter shades because the relationship between the number of photons entering the eye and perceived brightness is nonlinear (logarithmic). In dim lighting conditions, slightly increasing the amount of light is perceived as a large increase. The same small increase applied in bright lighting conditions is hardly perceived. This property of the human visual system makes it possible to store colour information in assets such as images more efficiently. Colours are converted from linear space (typically linear RGB components) to gamma space (typically nonlinear R'G'B' components, standardised as sRGB) using the formula

$$V_{\text{out}} = V_{\text{in}}^{\gamma}. \tag{2.89}$$

This operation is called gamma encoding when $\gamma < 1$ and gamma decoding when $\gamma > 1$. The value V that is raised to the power of γ is typically luminance (Y) which is computed as a weighted sum of linear RGB components,

$$Y_{709} = 0.2125R + 0.7154G + 0.0721B. \tag{2.90}$$

Gamma encoding redistributes colour ranges favouring darker tones. In this way more bits are used where they are needed most and a wide range of colours can be stored using only 8 bits per component. The γ value used for gamma encoding is typically $1/2.2 \approx 0.45$ (Figure 2.13). JPEG files store colours using a similar mechanism.

For physical correctness, light transport calculations need to be performed using linear data. For example, when computing the contributions from multiple light sources, the individual contributions are simply summed up. Any assets that contain nonlinear colour information therefore need to be converted to linear space before they are used in the calculations. When the calculations are complete and before the framebuffer is sent to the display, another colour conversion is required for historical reasons. On CRT televisions and monitors, brightness was not proportional to the input voltage, but to the input voltage raised to a power of around 2.5 (Devlin et al., 2002). Modern displays are designed to replicate this nonlinearity but with the power (gamma) standardised as 2.2. To nullify this so-called display gamma, pixel colour components are raised to the power of 1/2.2. This is called gamma correction, where an inverse transformation is applied to pixels to cancel out the display gamma.

Allowing for adaptation to different lighting conditions, the human visual system is sensitive to a wide range of luminance values, from around $10^{-6}\,\mathrm{cd\,m^{-2}}$ (or $\mathrm{lm\,m^{-2}\,sr^{-1}}$) to $10^{8}\,\mathrm{cd\,m^{-2}}$ (Hood and Finkelstein, 1986). By contrast, the typical range of luminance values that could be displayed on CRT displays was from $1\,\mathrm{cd\,m^{-2}}$ to $100\,\mathrm{cd\,m^{-2}}$. The ratio between the highest and lowest luminance values, 100:1 in this case, is called the dynamic range. Modern displays have a much higher dynamic range but this may still not be enough to match the dynamic range of the eye. Tone mapping tries to produce a perceptually correct image on a display that has a limited dynamic range. It converts a range of colours to a smaller range. For example, data specified in a high dynamic range with 10 bits per colour component (HDR10) is converted to a low dynamic range with 8 bits per colour component. Simple tone mapping can be performed using the function

$$L_{\text{out}} = \frac{L_{\text{in}}}{L_{\text{in}} + 1}, \tag{2.91}$$

which scales down higher luminance values by larger amounts, and maps values to within $[0, 1)$ which can then be scaled to the required range (Reinhard et al., 2002).

2.8 Video Compression

Video compression formats are designed to transfer streams of images efficiently, maintaining good image quality while using little bandwidth. The most widely used format is H.264, also called AVC (Advanced Video Coding). H.264 is pretty much ubiquitous and is hardware-accelerated even on low-powered devices such as smartphones. H.265, or HEVC (High Efficiency Video Coding), is H.264's successor. It compresses images more efficiently, thus requiring less bandwidth than its predecessor. However, it is more computationally demanding which makes it less suitable for mobile devices. VP9 (originally used by YouTube) and its successor AV1 (used by YouTube, Netflix, Facebook, Twitch, and others) are open (non-proprietary) formats competing with HEVC. In our work we primarily use H.264 to support a wide range of devices while keeping computation requirements low. For this reason we will focus on H.264 for the rest of the discussion. However, note that most concepts are common to all video compression formats.

H.264 supports a number of lossy and lossless encoding schemes. Many schemes are typically identified by a trio of values such as 4:2:0, 4:4:4, and so on, but not all schemes can be identified in this way. Colour information for the lossy formats is stored as three components, one component for luminance, Y, representing brightness (this is essentially a greyscale image, but it is usually called a "black-and-white" image) and two components for chrominance, U and V, representing colour differences. Colours are stored in this way and not as the usual RGB triple to take into account the human visual system. This helps achieve better compression ratios; the same concept is used for the lossy JPEG file format. The eye is more sensitive to brightness variations than to colour differences, so luminance is stored at a higher resolution than chrominance. U and V are also referred to as Cb and Cr respectively:

$$U = Cb = B' - Y' \tag{2.92}$$

$$V = Cr = R' - Y' \tag{2.93}$$

The prime symbol indicates that the value is gamma corrected.

Consider the layout in Figure 2.14 which illustrates the encoding of an 8×8 pixel image. Each cell in the top 8×8 grid contains a luminance value indicating that luminance is stored at full resolution. Each cell in the 4×4 grid at the bottom contains a pair of U and V values. This indicates that chrominance is stored at quarter resolution (half the height and half the width). One UV cell is used to provide colour information for four Y cells. All four cells at the top left corner of the top grid use the same UV values, the values stored in the the top left cell of the bottom grid. Each Y, U, and V value requires one byte of storage. Therefore the storage requirements for the 8×8 pixel image are

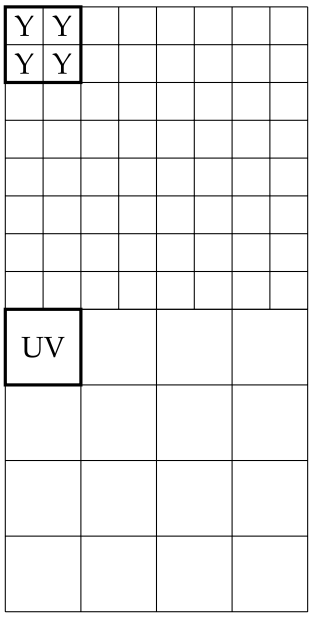


Figure 2.14: The 4:2:0 encoding scheme.

$8 \times 8 + 4 \times 4 \times 2 = 64 + 32 = 96$ bytes. If the image is encoded using an RGB triple for each pixel (one byte per component), the storage requirements would be $8 \times 8 \times 3 = 192$ bytes. Encoding the image as YUV therefore takes 50% less storage. After the encoding process, the data is compressed resulting in an even smaller size.

This encoding scheme is referred to as 4:2:0. The general representation is **J:a:b**. **J** identifies the size of a reference region (**J** pixels wide by 2 pixels high). The reference region represents a subset of pixels within the image (aligned to a 4-pixel boundary horizontally and to a 2-pixel boundary vertically). One luminance value is always associated with one cell in the reference region. **a** is the number of chrominance samples in the first row of the reference region. **b** is the number of changes for the chrominance samples between the two rows of the reference region. Figure 2.15 illustrates some configurations for the chrominance samples. Different colours indicate different chrominance samples. The reference region on the left contains two samples in the first row, and the same samples are retained in the second row, so there are zero changes; hence **2:0**. The central region contains two samples in the first row and both change in the second row, so there

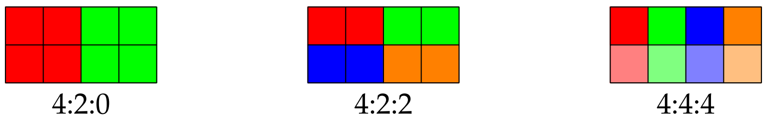


Figure 2.15: Reference regions specified with the J:a:b designation.

are two changes; hence **2:2**. The region on the right indicates lossless encoding since chrominance is stored at full resolution too. There are four samples in the first row, and all change in the second row (hence **4:4**), making eight samples in all, that is, one sample per pixel.

The compression process is extremely involved and sophisticated, and yields high compression ratios. Within the same frame, pixels are subdivided into blocks and matched in various ways with the other blocks. Moreover, compression is also performed temporally, between consecutive frames. Frames are categorised into three main types. I-frames (intra-coded frames) are "key frames". They are self contained and have no dependencies on previous or future frames, so they can be quite large. P-frames (predicted frames) contain deltas from previous frames. B-frames (bidirectional predicted frames) contain deltas from both previous and future frames. B-frames store information very efficiently and are particularly useful when a video stream can be buffered before being presented. For real-time streams, no buffering is used, so B-frames are avoided. Occasionally frames are "dropped", meaning they do not reach the destination. When this happens, errors in video stream quickly add up and visual artefacts are displayed. I-frames serve to correct this situation. Since they are self contained, they effectively reset the error to zero. Typically an I-frame is sent every so often for this purpose, for example once every 90 frames for a video stream playing at 60 frames per second.

H.264 is capable of encoding and compressing many different resolutions. It supports images with a fourth "alpha" component, and also components larger than 8 bits. Unfortunately it is only suitable for encoding and compressing image data. Transferring arbitrary data such as depth values is problematic. However, this has not stopped researchers from coming up with clever schemes which encode arbitrary data as image data, with varying degrees of success. Pece et al. (2011) encode 16-bit depth data into three 8-bit colour channels, taking into account the transformations performed by the video encoder. The decoded depth values are mostly close to the original values, but there are errors at depth discontinuities. Nenci et al. (2014) break down the depth data into multiple channels and transmit them over multiple H.264 streams. Their method trades bandwidth for accuracy.

H.264 has a daunting number of configuration parameters and can be tweaked and fine-tuned for specific situations. To simplify its setup, a number of profiles containing sets of commonly used parameters are defined in the standard. One particularly useful parameter is the target bandwidth. H.264 is capable of adjusting the effectiveness of its compression algorithm to respect the required target quite well. Visual artefacts manifest if the target bandwidth is too low for a particular frame or resolution.

2.9 Summary

This chapter described concepts, techniques and technologies that are referred to throughout this thesis. A thorough description of radiometric quantities was followed by an introduction to reflectance models. The rendering equation and several of its formulations were discussed together with some notes on Monte Carlo integration. The rasterisation pipeline and a number of techniques related to image-based rendering were outlined. The chapter concluded with a discussion on gamma correction and tone mapping, and a section on video compression which focussed on H.264.

3 Real-Time Global Illumination

Algorithms that synthesise images by replicating the behaviour of light accurately while also taking into consideration the nuances of indirect illumination are too slow for real-time applications. Nevertheless, through clever strategies and approximations, images that closely resemble those produced by offline methods can be generated in a fraction of the time. This chapter provides an overview of several of these methods, focussing on recent and promising approaches. Where appropriate, a description of the original slow techniques from which the faster methods are derived is included.

3.1 Caching

This section contains approaches that cache illumination data such as irradiance for later reuse. Caching algorithms are characterised by the type of information stored, where the information is stored, and how the information is then used for shading (Zhao et al., 2019).

Irradiance

The irradiance cache (Ward et al., 1988) was introduced as an optimisation for stochastic ray tracing (Cook et al., 1984). At the time, hardware support for ray tracing was still 30 years away (NVIDIA's RTX GPUs appeared in 2018), and although path tracing (Kajiya, 1986) was more efficient than distributed ray tracing it was still an expensive technique for computing reflection off diffuse surfaces.

Perfectly specular surfaces such as mirrors are the easiest kind of surface to evaluate. Their appearance depends on the viewer's direction. To shade a specific point on the surface, only a single sampling ray in the reflected viewer direction is required (Figure 3.1). Conversely, the evaluation of illumination on diffuse surfaces is the most difficult. The appearance of these surfaces does not depend on the viewer's direction and a substantial number of rays (possibly a hundred, or even more) are required to

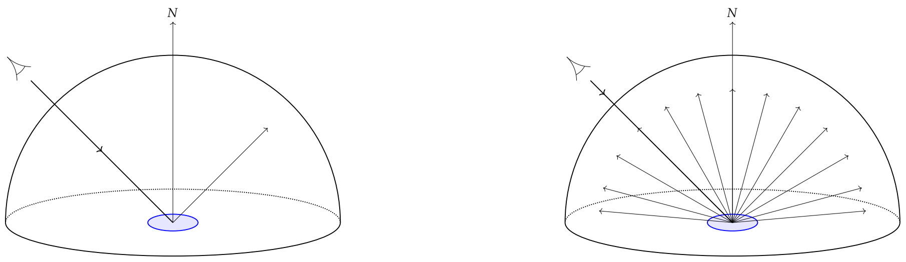


Figure 3.1: Reflection on perfectly specular (left) and ideal diffuse (right) surfaces.

sample the entire hemisphere of directions. Diffuse surfaces do have the advantage however, that light reflected off them is low frequency, that is, it is smooth and does not change abruptly. With this property in mind, Ward et al. (1988) designed a strategy by which the expensive hemisphere sampling mechanism is only needed occasionally, for a relatively small percentage of surface points. Whenever this computation occurs, the resulting irradiance value is stored in an irradiance cache, a spatial data structure such as an octree. A radius of influence is computed for every record in the cache. When an irradiance value for a surface point is needed, the cache is queried for any nearby points. If any records that include the point in question within their radius of influence are found, an irradiance value is computed efficiently using interpolation or extrapolation. Otherwise the full computation takes place. This procedure is illustrated in Figure 3.2. Values E_1 and E_2 are previously cached values. Irradiance at A is computed as the weighted average of E_1 and E_2. Irradiance at B is extrapolated from E_2. A new irradiance value is calculated for point C and cached.

The irradiance cache method uses an interesting mechanism for computing the radius of influence for a specific surface point. The curvature of the surface and the approximate distance of the surface point from other surfaces are taken into consideration. Higher surface curvature and closer proximity to other surfaces result in a smaller radius of influence. With this strategy, the full irradiance evaluation is computed more often

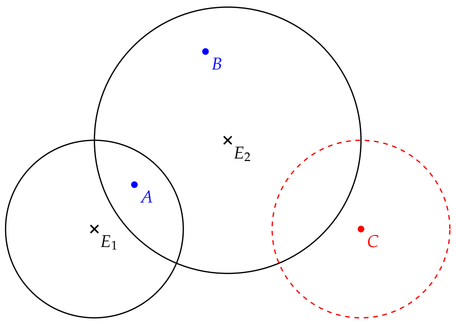


Figure 3.2: Interpolation and extrapolation using the irradiance cache.

where it is needed most. A visualisation of the surface points where the full evaluation takes place would show sparse points on large flat surfaces and densely packed points at the edges of objects. The method was later extended by Ward and Heckbert (1992) by introducing irradiance gradients. In the original method, irradiance values were interpolated linearly. In the extension, the gradient between irradiance samples is taken into consideration. Rotational and translational gradients are computed and used in the interpolation calculation, producing smoother results.

On a single processor, the irradiance cache speeds up the computation of indirect diffuse illumination over stochastic ray tracing by an order of magnitude (Ward et al., 1988). Since the cache is accessed frequently, at least once per pixel for queries and occasionally for creating new entries, parallelising the algorithm generates a large amount of contention for a shared cache, limiting the effectiveness of the implementation and only yielding modest speed-ups. Debattista et al. (2006) proposed an alternative strategy for computing global illumination using a parallel irradiance cache, which they implemented on a distributed cluster of computers. They split the computation of global illumination into two components: indirect diffuse lighting and all other illumination categories. They designated only a small proportion of the available machines to compute the indirect diffuse component in parallel. In this way, these machines had less work to do since they were only computing part of the complete solution, and contention was reduced since the number of machines was small. To further reduce contention, these machines maintained a local copy of the irradiance cache, periodically synchronising with each other. Simultaneously, the other machines in the cluster computed the second component. Combining the results from both machine groups produced the complete global illumination solution. This approach obtained speed-ups ranging from 6.9% to 15.5% over the next fastest parallel irradiance cache implementation.

The irradiance cache is populated with information obtained from the surfaces visible in the current camera view. Brouillat et al. (2008) proposed a hybrid method to seed the irradiance cache from surfaces throughout the entire scene. They used a photon mapping (Section 3.3) pass to gather scene-wide lighting information and populated the cache from that information. The scene can then be rendered using the irradiance cache instead of performing the expensive final gather pass in photon mapping. In this way, quick scene previews can be obtained.

Kán and Kaufmann (2013) used a modified irradiance cache together with ray tracing and rasterisation techniques to compute illumination in a mixed-reality setup, where virtual objects are rendered onto a real-world environment. A mixed-reality scene is particularly expensive to render since the global illumination solution needs to be computed twice. A geometric model of the real scene is acquired and rendered. The scene is then re-rendered with the virtual objects included in the model. Finally, the two solutions are combined with a real image of the scene to produce the end result. The irradiance cache helps to speed up the process.

Whereas the irradiance cache samples points on surfaces, the irradiance volume (Greger et al., 1998) samples points in space, within the scene's volume. The irradiance volume is targeted at diffuse surfaces and is designed to approximate indirect irradiance efficiently in detailed environments that are mostly static except for a few small dynamic objects. The volume is constructed in a precomputation stage by partitioning the scene into a bi-level grid. Starting off from a regular grid, any cells that contain geometry are subdivided once more to produce a finer sub-grid. Radiance is sampled at all the cell vertices in a set of directions and used to compute the irradiance distribution at that point. At runtime, irradiance at any point and direction in the scene is queried and used to compute radiance.

Radiance

Lambertian (ideal diffuse) surfaces reflect light uniformly in all directions and hence their appearance does not depend on the view direction. Reflected radiance for these surfaces is readily computed if the amount of irradiance (a directionless quantity) impinging on the surface is known. This fact was used to good effect by the irradiance cache (Ward et al., 1988). On the other hand, the appearance of glossy surfaces depends on the view direction, so an irradiance cache cannot be used without modification to accelerate the computation of global illumination for these surfaces. In a method derived from the irradiance cache, Krivánek et al. (2005) added support for low-frequency (not very shiny) glossy BRDFs. The new method stores directional incident radiance instead of irradiance and is accordingly named radiance caching. As in the irradiance caching method, radiance cache records are stored in an octree.

Spherical harmonics (SH) form an orthonormal basis for functions on the sphere and are often used in computer graphics to represent BRDFs and environment maps as they can store directional information efficiently. If the information is only required for a hemisphere, a common use case, an even better representation can be achieved with hemispherical harmonics (HSH) (Gautron et al., 2004). In both cases, the information is stored as a number of coefficients; the representations are impractical for storing high-frequency information because a large number of coefficients (and hence high storage requirements) would be needed. As in photon mapping (Jensen, 1996), the evaluation of the rendering equation is split into several parts: direct illumination, perfectly specular, ideal diffuse, low-frequency BRDFs and high-frequency BRDFs.

The method proceeds as follows. As a preprocessing step before the rendering starts, all the BRDFs used in the scene are analysed. The BRDFs suitable for radiance caching are identified and their representation in HSH is computed. To determine BRDF suitability, the user specifies a maximum error, n_{max}. If no HSH representation of order $n < n_{max}$ is sufficient for the specified error, the BRDF is deemed not suitable. During the rendering stage, at every ray-surface intersection, the HSH representation of the BRDF at the point of intersection is retrieved. If a representation is not available, the indirect glossy and indirect diffuse terms are computed using Monte Carlo importance sampling and irradiance caching respectively. Otherwise, these terms are computed from the radiance cache. Either the incident radiance is interpolated from nearby radiance cache records or a new radiance cache record is created. Radiance interpolation is carried out by interpolating the coefficients and the interpolation quality is enhanced by using translational gradients. SH rotation is used to align the local coordinate frames at the interpolation point and at the cached radiance point. Once the incident radiance is obtained it is used together with the BRDF to compute outgoing radiance. Since incident radiance at a point is represented as a vector of SH or HSH coefficients, and the scene BRDFs are also represented in the same basis, the illumination integral evaluation reduces to a dot product of the interpolated incident radiance coefficients and the BRDF coefficients. In this manner, outgoing radiance is computed.

A limitation of radiance caching is that it cannot handle caustics. Moreover, both irradiance caching and radiance caching are susceptible to artefacts produced by ray leaking due to imperfections in the scene model; neighbour clamping (Krivánek et al., 2006) is used to mitigate these issues.

Vardis et al. (2014) cache radiance as a luminance-chrominance pair, with the perceptually less-important chrominance component stored at a lower resolution (using fewer SH coefficients). This allows storing luminance in higher-order spherical harmonics while retaining the same memory budget. Zhao et al. (2019) cache outgoing radiance to facilitate support for highly glossy surfaces and glossy-to-glossy interreflection effects.

Radiosity

Even though computer-generated imagery for the film industry is not constrained by tight time budgets, decreasing the rendering times of offline renderers saves on costs. Moreover, during production, quick previews are useful to confirm that the lighting will appear correct in the final render. Tabellion and Lamorlette (2004) were the first to use global illumination in a full-length animated feature (Shrek 2). They cached both irradiance and the result of direct illumination in a hybrid method designed to speed up final render times and to improve the workflow of artists by providing them with a fast feedback loop. Irradiance caching was enhanced to support non-diffuse surfaces in an approximate lighting model. Direct illumination was optionally precomputed and stored in texture maps to speed up final gathering by avoiding calls to complex surface shaders. Low-resolution texture maps were used to reduce noise.

Pixar, a subsidiary of Walt Disney Studios, used a rendering system called Reyes (Cook et al., 1987) for many years, producing famous feature films such as Cars (2006) and Monsters University (2013). In Reyes, surfaces are broken down into small patches and each patch is tessellated into micropolygons. The vertices of the micropolygons in a patch are called a micropolygon grid. Shading is performed in full only for the micropolygon grids and obtained by interpolation at all other points. Christensen et al. (2012) observed that when the geometry is detailed and complex surface shaders are used, most of the rendering time is spent on shading the points hit by rays rather than obtaining the hit points themselves. Shading includes launching shadow rays towards the light sources to evaluate direct illumination and calling complex surface shaders. Recall the rendering equation,

$$L(\boldsymbol{x}, \boldsymbol{\omega_o}) = L_e(\boldsymbol{x}, \boldsymbol{\omega_o}) + \int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L(r(\boldsymbol{x}, \boldsymbol{\omega_i}), -\boldsymbol{\omega_i})\, (\cos\theta_i)^+\, d\boldsymbol{\omega_i}. \tag{3.1}$$

Let $\boldsymbol{x}$ be a non-emissive point so the $L_e(\boldsymbol{x}, \boldsymbol{\omega_o})$ term can be dropped. Referring to the point $r(\boldsymbol{x}, \boldsymbol{\omega_i})$ as $\boldsymbol{x'}$ for brevity and expressing $L(\boldsymbol{x'}, -\boldsymbol{\omega_i})$ as a sum of emitted and reflected radiance yields:

$$\begin{aligned}
L(\boldsymbol{x}, \boldsymbol{\omega_o}) &= \int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, (L_e(\boldsymbol{x'}, -\boldsymbol{\omega_i}) + L_r(\boldsymbol{x'}, -\boldsymbol{\omega_i}))\, (\cos\theta_i)^+\, d\boldsymbol{\omega_i} && (3.2)\\
&= \int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_e(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, (\cos\theta_i)^+\, d\boldsymbol{\omega_i} && (3.3)\\
&\quad + \int_{\Omega^+} f_r(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_r(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, (\cos\theta_i)^+\, d\boldsymbol{\omega_i} \\
&= \text{direct illumination} && (3.4)\\
&\quad + \text{indirect illumination.}
\end{aligned}$$

Furthermore, diffuse and specular surfaces can be computed separately too:

$$
\begin{aligned}
L(\boldsymbol{x}, \boldsymbol{\omega_o}) &= \int_{\Omega^+} (f_{\text{diffuse}}(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o}) + f_{\text{specular}}(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o}))\, L_e(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i} \qquad (3.5)\\
&\quad + \int_{\Omega^+} (f_{\text{diffuse}}(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o}) + f_{\text{specular}}(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o}))\, L_r(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i}\\
&= \int_{\Omega^+} f_{\text{diffuse}}(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_e(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i} \qquad (3.6)\\
&\quad + \int_{\Omega^+} f_{\text{specular}}(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_e(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i}\\
&\quad + \int_{\Omega^+} f_{\text{diffuse}}(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_r(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i}\\
&\quad + \int_{\Omega^+} f_{\text{specular}}(\boldsymbol{x}, \boldsymbol{\omega_i}, \boldsymbol{\omega_o})\, L_r(\boldsymbol{x'}, -\boldsymbol{\omega_i})\, (\cos\theta_i)^+ \, d\boldsymbol{\omega_i}\\
&= \text{direct diffuse illumination} \qquad (3.7)\\
&\quad + \text{direct specular illumination}\\
&\quad + \text{indirect diffuse illumination}\\
&\quad + \text{indirect specular illumination.}
\end{aligned}
$$

Christensen et al. (2012) reduced rendering times for Reyes by splitting up the rendering equation into these components, computing the view-independent parts (the diffuse components) separately and caching the results. In this way, part of the expensive computation is computed only once and reused many times. Whenever a value cannot be computed from the cache, the full computation is performed for the patch's entire micropolygon grid. The method was used both for final renders and for previews, obtaining speed-ups that ranged from 3 to more than 30.

3.2 Precomputation

By precomputing lighting, large speed-ups are possible at runtime. Although this strategy has limited use for dynamic scenes, preprocessing the static parts of the scene may still be effective at reducing computation costs.

The precomputed radiance transfer (PRT) (Sloan et al., 2002) class of methods compute partial light transport in a preprocessing stage. Light transport is computed as a transfer function at each vertex of an object. A transfer function is a matrix that includes the BRDF and visibility, and is stored using spherical harmonics. The transfer functions used by Sloan et al. (2002) convert incident low-frequency lighting that is assumed to originate from distant environment maps to transferred (outgoing) radiance over the

Expression	Description
L	light
E	eye
D	diffuse scattering event
S	specular or glossy scattering event
X|Y	X or Y (logical OR)
X∗	zero or more occurrences of X
X+	one or more occurrences of X
X?	zero or one occurrences of X
()	parentheses, used for grouping
(LDS∗E) | (LS∗E)	classic ray tracing
LD∗E	classic radiosity
L(D|S)∗E	global illumination

Table 3.1: Heckbert's light transport notation.

surface of the object. At runtime the functions are used to compute effects such as self-shadowing and reflections of the object's surface onto itself. Since the light transport is partially already computed, these effects can be produced in real time. At runtime the object that stores PRT can be viewed from arbitrary locations and the scene illumination can be dynamic. The method is particularly well-suited for complex models such as faces. The method was later extended to support deformable (non-rigid) objects (Sloan et al., 2005).

3.3 Photon Mapping

This category comprises multi-pass bidirectional methods where light distribution is first computed, and then gathered in a final pass.

Photon mapping (Jensen, 1996) is designed to optimise Monte Carlo ray tracing methods such as stochastic ray tracing and path tracing. It is particularly efficient in the computation of caustics, areas of concentrated reflected or refracted light such as those observed around glass objects or at the bottom of swimming pools. In light transport notation (Heckbert, 1990), caustics are produced by light paths of type LS+DE. This notation is summarised in Table 3.1. It is designed to describe concisely the path a ray or photon takes as it propagates away from a light source, in terms of the scattering events occurring along the path. For example, a path of type LDE describes the direct illumination of a diffuse surface, where a ray originating at a light source is scattered diffusely once (D) before reaching the eye. To produce caustics, light needs to be scattered

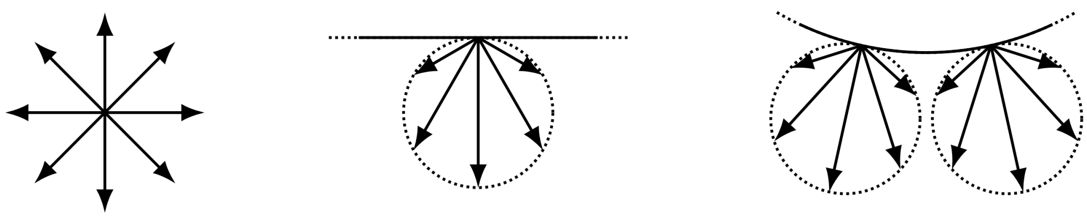


Figure 3.3: Photon emission from a point light (left), a diffuse rectangular area light (centre), and a diffuse spherical light (right). Adapted from Jensen (2001).

at least once by a specular or glossy surface (S+), then by a diffuse surface (D) before finally reaching the eye, hence LS+DE.

In photon mapping, rays or photons are fired from the light sources in a first pass, a technique known as light ray tracing (Arvo, 1986), and from the eye in a second pass as is usually done in ray tracing methods. A large number of photons are fired in the first pass, typically around 200K to 500K for small scenes. Figure 3.3 illustrates photons fired from various light sources. The radiant power of the light source is split among all the photons fired from it. For diffuse area lights photons are fired using a cosine-weighted distribution, favouring directions close to the surface normal. This compensates for the foreshortening term in the calculations at the surface point hit by the photon, helping to obtain the global illumination solution more efficiently. The photons are followed around the scene as in path tracing, and every time a surface is hit, radiant power and directional information are stored in photon maps. These data structures are internally represented as balanced *k*-d trees for efficient search operations.

Photons are fired in two waves, a high-quality wave consisting of a high density of photons and a low-quality wave. The high-quality wave is aimed at specular objects in the scene and stores information in a caustics photon map. This map is used to compute caustics during the second pass. This information is directly visualised, hence the need for it to be of high quality. The low-quality wave is fired throughout the scene and the information obtained is stored in a global photon map. In the second pass this information is used to compute diffuse indirect illumination. The low quality of this information is beneficial for the final image since it introduces some blurring which serves to smoothen the high frequency noise that is usually visible in Monte Carlo ray tracing algorithms.

In the second pass the integral part of the rendering equation (the reflected light) is decomposed into components by splitting both the incoming radiance and the BRDF:

$$L_i = L_{\text{light}} + L_{\text{diffuse}} + L_{\text{specular}} \tag{3.8}$$

$$f_r = f_{\text{diffuse}} + f_{\text{specular}}, \tag{3.9}$$

where f_{diffuse} represents ideal diffuse and slightly glossy surfaces, and f_{specular} represents highly glossy and perfectly specular surfaces. The resulting components are light paths of types L(D | S)E (direct illumination), L(D | S)+SE (specular reflection), LS+DE (caustics), and LD+DE (diffuse indirect reflection). Caustics are computed directly from the caustics photon map. If an approximate evaluation is deemed sufficient, direct illumination and diffuse indirect reflection are computed directly from the global photon map. A more accurate evaluation is required for these terms when a surface point is viewed directly or by reflection off a specular surface, and when light bounces between surfaces that are in close proximity so that colour bleeding effects are reproduced correctly. Although not used directly for accurate evaluations, the information in the photon maps is still useful because it can guide the selection of a new ray direction and also reduce the number of shadow rays needed. Evaluation of the specular reflection term does not make use of the photon maps but it is cheap because the solid angle that needs to be sampled for highly reflective materials is small, so only a few new rays are required.

The original photon mapping algorithm speeds up the rendering of caustics but rendering times are still in the region of some minutes. The first and last ray path segments between a light source and the eye are the most expensive parts of the algorithm. The first bounce of the first pass (from the light) requires tracing and evaluating a large number of photons. The first bounce of the second pass (from the camera) needs to be evaluated many times for each pixel. McGuire and Luebke (2009) observed that photon mapping could be accelerated by using rasterisation techniques for computing the two expensive endpoints if point light sources and a pinhole camera model are assumed. Their algorithm, image space photon mapping, rendered Full HD frames at up to 26 Hz. CPU-based ray tracing was used for computing the middle segments of the ray paths.

When the illumination is dominated by caustics or when light paths of type LSDSE are important, such as when rendering the glass casing of a light source, millions of photons may be required to obtain accurate results. However, photon map sizes are constrained by the amount of available memory. Progressive photon mapping (Hachisuka et al., 2008) circumvents these issues. An initial ray tracing pass identifies the visible points in the scene, either directly or through specular reflections, and stores them in a photon map. An unlimited number of photon tracing passes follow. In each of these passes, thousands of photons are fired from the light sources as in standard photon mapping. These photons do not add more entries to the photon map but only update the radiance estimates for nearby stored points. This strategy limits the size of the photon map and progressively refines it. An image can be rendered after every photon tracing pass. The method is effective at computing radiance estimates at individual points. Hachisuka and Jensen (2009) inserted a stochastic ray tracing pass after each photon tracing pass to randomly generate new hit points within a region of interest. This extension enables the computation of average radiance estimates over a region, thereby supporting effects such as depth of field and motion blur.

In the methods of Hachisuka et al. (2008) and Hachisuka and Jensen (2009), statistics about each point or region need to be maintained. Knaus and Zwicker (2011) showed that the photon tracing passes can run independently of each other, thereby allowing for parallel implementations and removing the need to maintain statistics. Their method, probabilistic progressive photon mapping (PPPM) can also be applied for rendering in the presence of participating media. Evangelou et al. (2020) produced a GPU-based implementation of PPPM using only rasterisation techniques (that is, without ray tracing) that runs at interactive rates.

3.4 Many-Light Methods

Many-light methods are inspired by the idea that indirect illumination can be simulated by a large number of point lights scattered throughout the scene. This concept, first used in the instant radiosity method (Keller, 1997), effectively created a new category of rendering algorithms.

Instant radiosity computes direct and indirect reflections off diffuse surfaces by utilising point lights. Each light source is replaced by a number of point lights, called virtual point lights (VPLs), distributing the intensity of the original light source between the newly created lights. For area lights, the VPLs are distributed over the light source as well. As in light tracing (Arvo, 1986) and photon mapping (Jensen, 1996), a light path is traced from each VPL throughout the scene for a number of bounces. A new VPL is created at each path vertex and its intensity is computed. Only a fraction of all the light paths are allowed to proceed beyond the first bounce. Similarly, only a fraction of the remaining light paths proceed beyond the second bounce, and so on. Using rasterisation techniques, the scene is rendered multiple times, from the point of view of each VPL. Each of these renders produces a shadow map (Williams, 1978), a depth buffer from the point of view of the light source. The shadow maps are stored in an accumulation buffer (Haeberli and Akeley, 1990), effectively combining them and producing the final image.

Modern hardware can compute hundreds of shadow maps per second. However, since thousands of VPLs are needed to simulate high-quality indirect illumination, in its original form instant radiosity cannot be regarded as a real-time rendering algorithm. Improvements to the algorithm include generating only a few VPLs per frame and progressively updating the results. Computing the best placement for VPLs is a difficult problem and is the subject of some research. An advantage of instant radiosity is that it does not exhibit the noise inherent in Monte Carlo ray tracing. The algorithm's calculations contain a weak singularity due to the inverse-square law and high intensities (manifested as bright patches of light) result when points are in close proximity to VPLs. These artefacts can be mitigated by clamping values to a desired range. However, this results in darker images as some energy is lost from the system. Modern variants of the algorithm have mechanisms that compensate for the lost energy.

A shadow map captures the surfaces directly visible from a light source. If there is only one light source, the points stored in the shadow map are exactly the points where the first and most important bounce of indirect light occurs. Dachsbacher and Stamminger (2005) use this observation to extend shadow maps into reflective shadow maps (RSMs) where information about reflected light is stored at each pixel in the map. The pixels are then treated as *pixel lights*, small light sources that illuminate the scene indirectly. The method assumes diffuse surfaces and that there is a single delta light source (point light or spot light) or directional light source in the scene. Moreover, occlusion is ignored during the evaluation of indirect illumination.

The information stored at each pixel is captured using multiple render targets and consists of depth, world space position, surface normal, and reflected radiant power. Computing indirect illumination at a surface point requires evaluating and summing up the irradiance contributions from the pixel lights. Due to the large number of pixels in the RSM (typically hundreds of thousands), evaluating the irradiance from all the pixel lights would be too expensive for interactive rendering. Instead, the most relevant few hundred pixel lights for a surface point are determined by projecting the point onto the RSM and randomly selecting pixel lights from those in the vicinity. To further reduce the number of evaluations, a three-pass approach is used. In the first pass, indirect illumination is computed for a low-resolution image. In a full-resolution second pass, indirect illumination for each pixel is obtained by interpolation from the low-resolution samples if they are deemed suitable based on normal similarity and world space proximity. The final full-resolution third pass evaluates indirect illumination in full for any remaining pixels. In general only the pixels at the edges of objects require the full evaluation.

Instant radiosity supports dynamic geometry and lights but requires many passes. Although the original RSM method only supports a single simple light source, it requires fewer passes and provides a rough approximation for one-bounce indirect light in dynamic scenes. For simple scenes at low resolution (512×512) frame rates between 4 Hz and 28 Hz were obtained.

Visibility or occlusion queries, that is, determining whether there is a direct line of sight between two points, is an expensive part of the GI computation. While Dachsbacher and Stamminger (2005) completely avoided these calculations for indirect illumination, Ritschel et al. (2008) compute and use approximate visibility instead. Possibly inaccurate visibility information is obtained from a low-quality point-based representation of the scene and stored in low-resolution shadow maps called imperfect shadow maps (ISMs). The generation of ISMs is fast and many can be created in each frame. Due to the sparse scene representation there may be missing depth values in an ISM. This is corrected using a pull-push mechanism. An image pyramid is created for the ISM with the finest image at the bottom and progressively coarser images at higher levels; each level stores an image at a quarter of the resolution of the image in the level just below it. In the pull stage, the pyramid is processed from bottom to top, that is, from the finest level to the coarsest level. The coarser image at each level is populated by interpolating valid depth values from the finer version below it. In the push phase, the pyramid is processed from the coarsest to the finest level, interpolating from the coarse level to fill missing data in the finer level. With only two coarse levels, this method fills in most holes. After generating VPLs, an ISM is created for each one. The ISMs are paraboloid shadow maps (Brabec et al., 2002) that cover the entire hemisphere of directions. During the evaluation of indirect illumination for a point, visibility queries are performed efficiently as for regular shadow maps, by computing the position of the point relative to the light and comparing the resulting depth value with the depth value in the ISM. Using the speed-up provided by ISMs for computing indirect illumination, Ritschel et al. (2008) obtained a frame rate of 12 Hz for a large scene at a resolution of 1920×1600. The method supports full dynamic scenes and multiple indirect light bounces.

Several methods reduce computation costs by grouping VPLs into clusters, thereby reducing their numbers (Prutkin et al., 2012). Lensing and Broll (2013) take the opposite approach. Instead of reducing the number of VPLs they reduce the number of surface points that require shading, and then interpolate the computed illumination over the rest of the surface.

3.5 Light Probes

These methods rely on screen-space techniques and image-based lighting, and can also be considered to be a form of caching. Due to their efficiency they are commonly used in video games.

Computing the rendering equation at every point in a scene is impractical for real-time applications. A much more efficient alternative is to compute lighting sparsely, only at a few locations. These locations and the lighting information stored in them (typically irradiance or radiance) are called light probes. Lighting at other locations is derived from the probes cheaply, for example by interpolation. Quite convincing results can be generated in this way. It is common for probe locations to be selected manually, and for probes to be added or removed using trial and error, both time-consuming processes. Selecting the best locations for the probes and determining how many probes are sufficient to reproduce lighting plausibly are subjects of active research. Probes may be specialised too. For example, probes may store information that is useful for computing indirect lighting and shadows, another set of probes may be used to compute reflections, and so on.

McGuire et al. (2017) use *light field probes*, storing radiance and visibility information. The probes are automatically placed in a regular grid and are populated in a precomputation step that takes one to two minutes; at this stage the environment is static. At runtime, the information in the probes is queried in real time and used to produce view-dependent global illumination for both static and dynamic objects. The method supports detailed "centimetre-scale" geometry. A light field ray tracing algorithm is used. The geometry information stored in the probes is used to reduce light and shadow leaks. The method automatically supports area lights and soft shadows.

Majercik et al. (2019) extend irradiance probes to encode the scene's dynamic indirect irradiance field, where occlusion is also taken into account. The method is called *Dynamic Diffuse Global Illumination* (DDGI) and builds upon the work of McGuire et al. (2017). The method produces good image quality at a fraction of the cost of offline path tracing methods. The probes are again placed on a regular grid. However, rather than populating the probes in a precomputation step, they are dynamically updated at runtime in every frame, producing more accurate global illumination for dynamic scenes. A number of optimisations were applied to the method in Majercik et al. (2020). A system using continuously updated probe states was devised in which only those probes that are judged as contributing to the final image are used, increasing performance. Probes are also pruned by a 3D "window" positioned around the camera. Multiple probe volumes at different resolutions, similar to those used in Kaplanyan and Dachsbacher (2010) are used too.

Vardis et al. (2021) use a simple method to place light probes automatically. A large number of probes are initially placed in the scene automatically, using a regular grid or some other strategy, and the radiance field is computed. The probes are connected to form an undirected graph. A number of evaluation points are then placed around the scene at strategic locations, for example close to dynamic geometry. Incident radiance is computed at the evaluation points from the nearest k neighbouring probes identified from the graph. Next, the number of probes is reduced in an iterative process. A probe is removed, and incident radiance at all the evaluation points is recomputed. The error in the new results is recorded, and the probe is added to the graph again. When all the probes have been processed, the probes that contribute the least according to certain illumination criteria are pruned.

3.6 Voxels

A voxel (volume pixel), the three-dimensional equivalent of a pixel, stores information at a point in three-dimensional space. It is typically visualised and rendered as an axis-aligned cube on a regular grid. However, since a voxel can store any kind of information it can be rendered in an arbitrary way, for example as a sphere or as a textured polygon. This flexibility, combined with their regular structure which makes them easy to manipulate, makes volumes (voxel models) particularly well-suited for representing finely detailed materials such as fur and hair or amorphous phenomena such as clouds and smoke. For these cases, using many tiny geometric primitives instead produces aliasing artefacts that cannot be countered with traditional antialising methods (Kajiya and Kay, 1989). Voxels are also useful to represent distant models and to speed up operations such as collision detection.

The methods in this category use a voxelised representation of the scene to compute global illumination. The representation simplifies the geometry and provides a spatial subdivision of a volume that is convenient for storing and propagating light information. The easiest way to implement the voxelisation process is by a CPU-based approach. However, this is only useful if voxelisation is needed only once and can be performed as a precomputation. For speed, GPU-based approaches using rasterisation, ray tracing, or compute (general-purpose GPU computation) are necessary.

Kaplanyan (2009) achieved single-bounce diffuse global illumination for CryEngine 3 in a tight time budget of 3.3 ms per frame. A set of VPLs was generated using a reflective shadow map, clustered based on their intensities, and used to seed a spatial subdivision structure representing the distribution of light in the scene. The volume representation used, called a light propagation volume (LPV), was a coarse $32 \times 32 \times 32$ regular 3D grid with each cell containing two bands of spherical harmonic coefficients. Intensity is propagated iteratively throughout the volume, which is then used for lighting the scene. In each iteration, intensity is propagated to the six neighbouring cells in the axial directions. For a neighbouring cell, the incident radiant power on each face is computed, transformed into outgoing intensity and stored at the centre of the cell. Cascaded light propagation volumes (Kaplanyan and Dachsbacher, 2010) are an extension of the method, using multiple nested grid levels to support level of detail and large scenes. Interpolation between grid levels is used at the boundaries to ensure smooth transitions. Using three cascaded LPVs, real-time rendering rates (58 FPS) were attained for a large scene at a resolution of 1280×720.

In the voxel cone tracing method of Crassin et al. (2011), parts of the voxel representation are recreated in real time, in each frame, to support dynamic scenes. As in Kaplanyan and Dachsbacher (2010), the voxelised representation consists of a set of nested grids. The entire structure resides in GPU memory as a hierarchical voxel octree. The leaves of the octree hierarchy are seeded with incoming radiance from the light sources; the higher levels are populated by filtering these radiance values. The structure is used to compute visibility and indirect illumination, supporting both diffuse and glossy reflections. Indirect illumination is estimated by ray tracing through the octree hierarchy. A ray represents a cone of directions, with a tight cone for specular materials and a wide cone for diffuse materials. At every step along the ray the cone radius is calculated, and a light estimate is computed from the level in the octree hierarchy that matches that cone radius and accumulated. At a resolution of 1024×768, for a scene containing diffuse and specular materials, one dynamic object and one moving light, the method obtained a frame rate of 11 Hz. Sugihara et al. (2014) used voxel cone tracing only for computing visibility and estimated indirect illumination with RSMs that are split into layers (layered reflective shadow maps, LRSMs).

Since detailed voxel models may require more memory than is available, data may need to be streamed in from disk. Aiming to show that voxels can be a viable rendering primitive even for an exorbitant number of voxels, Crassin et al. (2009) developed an out-of-core voxel rendering method that achieved frame rates between 20 and 90 Hz with scenes containing billions of voxels. The volume is represented as an octree where each node contains a pointer to a small voxel grid (typically 32^3) called a brick. Tree nodes and bricks are stored in a tree node pool and a brick pool respectively, both 3D textures. The volume is mirrored on the CPU to allow easy modifications to the volume. The GPU is then updated with any changes. Laine and Karras (2010) also investigated the use of voxels as a geometric primitive. They developed an out-of-core method using a sparse voxel octree representation and presented an efficient ray casting algorithm for this data structure. In their tests, primary rays were cast at a rate of 107 million rays per second for triangles and at 122 million rays per second for voxels.

3.7 Summary

This chapter provided a literature review of algorithms designed to accelerate various aspects of the global illumination computation (Table 3.2). Caching methods obtain speed-ups by performing expensive computations infrequently, reusing or interpolating from previous results whenever possible. Precomputation methods perform costly computations in a preprocessing stage and use the results at run-time. Photon mapping is used to compute caustics efficiently. Many-light methods simplify the global illumination computation by replacing it with the evaluation of many point lights scattered throughout the scene. Light probes operate similarly to caching methods; they typically store sparse illumination information in space rather than on surfaces. Voxels are versatile; some of their uses include speeding up data structure traversal, and storing and propagating illumination data.

Category	Method
Caching	Ward et al. (1988), Ward and Heckbert (1992) Debattista et al. (2006) Brouillat et al. (2008) Kán and Kaufmann (2013) Greger et al. (1998) Tabellion and Lamorlette (2004) Christensen et al. (2012) Krivánek et al. (2005)
Precomputation	Sloan et al. (2002), Sloan et al. (2005)
Photon Mapping	Jensen (1996) McGuire and Luebke (2009) Hachisuka et al. (2008) Hachisuka and Jensen (2009) Knaus and Zwicker (2011) Evangelou et al. (2020)
Many-Light Methods	Keller (1997) Dachsbacher and Stamminger (2005) Ritschel et al. (2008)
Light Probes	McGuire et al. (2017) Majercik et al. (2019), Majercik et al. (2020) Vardis et al. (2021)
Voxels	Kaplanyan (2009), Kaplanyan and Dachsbacher (2010) Crassin et al. (2011) Sugihara et al. (2014) Laine and Karras (2010) Crassin et al. (2009)

Table 3.2: Global illumination algorithms.

4 Distributed Rendering

This chapter provides an overview of distributed rendering systems, focussing on cloud-based approaches and user interactivity. We briefly introduce the class of distributed systems dedicated to computation, describing hardware organisation and system architectures. We discuss the main challenges in interactive distributed rendering and review the relevant literature.

4.1 Distributed Computing

A distributed system is a collection of networked computation nodes. A node may be a software process or a hardware device of any capability, from a powerful supercomputer to a cheap microprocessor. Van Steen and Tanenbaum (2017) define a distributed system in terms of two characteristic features, namely that the nodes operate independently of each other, and the abstraction of the system as a single entity:

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

There are three main classes of distributed systems. These are distributed computation, distributed information (the prime example is the Web), and pervasive systems such as the Internet of Things (IoT). Since we are interested in computation for rendering we will focus on the distributed computation class.

A machine may be upgraded to a higher computational capability by improving its individual components. The number of computation cores could be increased, or its CPU replaced with one that has a higher clock speed. This is known as *vertical* (or *up*) *scaling*. Due to advances in component miniaturisation, fitting multiple cores on a single chip is common practice, with current high-end processors having 16 cores and a maximum clock frequency of around 5 GHz. However, due to physical limits and heat dissipation issues, further increasing the computation power of the system can realistically only be achieved by adding more machines and linking them up over a network so they can collaborate, thereby creating or expanding a distributed system. This is referred to as *horizontal* (or *out*) *scaling*.

4.1.1 Organisation

The hardware for distributed computing systems can be organised in many ways, from a simple two-node setup where data is exchanged over a direct connection or a LAN, to a large number of geographically dispersed nodes communicating over the Internet. In high-performance distributed computing, where a number of computers cooperate to accomplish a task, the hardware is organised into clusters or grids. The Cloud, with its vast computational resources, can be regarded as the next step up from grids, but it also provides general-purpose computation as a utility.

Clusters A cluster, also called a server farm (or a render farm if the computation is specific to rendering) can be thought of as a virtual supercomputer. It is a collection of similar computers running the same operating system and connected by a LAN. All the hardware is privately controlled by an entity. This is the typical setup used for cryptocurrency mining, where large computation resources are needed. Miners distribute the workload over arrays of connected processors. Render farms have been used for many years in the film industry. To produce high quality visual effects (VFX), a single frame may require hours of rendering time. To be able to complete the task within a reasonable time frame, render farms may consist of thousands of machines. Clusters typically use the master-worker communication model.

Grids There are two classes of grids. Computational grids are made up of heterogeneous nodes, use dedicated powerful computational resources, and are connected by high-speed networks. Multiple organisations pool their resources for collaboration in large-scale computation endeavours, creating a virtual organisation of sorts. These systems can consist of thousands of machines scattered over a large geographic area, country-wide or even globally. The Worldwide Large Hadron Collider Computing Grid (WLCG) (Bird, 2011) is an example of such an international collaboration.

The second grid class is the desktop grid, where computer users contribute their machine's idle computer cycles to process a job, a tiny part of a much larger task. Desktop grids can be Internet-based or LAN-based. In the latter case they are also called local desktop grids. Famous examples of Internet-based desktop grids, also known as supercomputing projects, are the SETI@home[1] (Anderson et al., 2002) program, which is currently in hibernation, the still-active Great Internet Mersenne Prime Search[2] (GIMPS), and Folding@home[3], an ongoing protein-simulation project used to study and design treatments for diseases such as cancer, Alzheimer's, and COVID-19. The framework used in these cases is known as volunteer or public-resource computing. Users enrolled in these schemes install software which uses the Internet to communicate with a command centre and receive job details. When the job is completed, the results are communicated back, where they are validated and merged with the results from other volunteers. Internet-based desktop grids have also been used for distributed rendering, for example, the now-defunct Big and Ugly Rendering Project (BURP)[4].

LAN-based desktop grids (Litzkow et al., 1987) are well-suited for research labs and small institutions. Networked workstations in academic and research and development environments are typically underutilised and are available approximately 70% of the time (Mutka and Livny, 1987). By using the computation power of these machines when they would otherwise be idle, a large amount of computing power can be obtained essentially for free.

Since desktop grids consist of volatile resources, algorithms running on these systems need to be fault tolerant. The same job may be scheduled to run on more than one machine for redundancy. Jobs may fail and may need to be resubmitted. Checkpointing, where the state of a job can be saved, and the job resumed later, possibly on another machine, is an often-used strategy.

Cloud Computing The Cloud provides computation as a service and is used for a myriad of applications. In the last decade or so, remote rendering, with the remote end being cloud-based has become immensely popular, particularly since it is capable of bringing gaming to mobile platforms and other weak devices at a low cost. Shi and Hsu (2015) define a remote rendering system as a network-connected pair of devices, with frames produced at one end on the server device and displayed at the other end on the client device. To allow interactivity, the system must accept user input on the client and forward it to the server, where it is acted upon by the rendering application running there. When used for gaming, this setup is known as cloud gaming or game streaming.

Film studios are also opting to out-source their render farms to operate on cloud resources such as Amazon Web Services (AWS).

[1] https://setiathome.berkeley.edu/
[2] https://www.mersenne.org/
[3] https://foldingathome.org/
[4] `http://burp.boinc.dk/`

To provide better quality of service (QoS) to customers, edge computing is often used in conjunction with cloud-based services. It is a way to bring parts of a service closer to the consumer, for example, content delivery networks (CDNs). Fog computing, in which a number of small servers are placed between the Cloud and edge devices, close to the latter, further improves the system. The idea of fog computing is to provide more intelligent data flow between edge devices and cloud servers. If part of the data processing can be performed at the fog layer, cloud servers need to be contacted less often, making the system more efficient.

4.1.2 System Architectures

The system architectures used on distributed systems can be categorised as centralised (client-server or master-worker), decentralised (peer-to-peer), or hybrids of the two. In centralised architectures, there is a main node that the other nodes communicate with. An example of a centralised client-server system is Video on Demand (VoD) where clients request videos from the server. In decentralised architectures, the nodes are functionally equivalent. They communicate directly between themselves and there is no main node. Blockchain technology, Skype, and Spotify are all peer-to-peer systems.

Client-Server The most common centralised architecture is the client-server request-reply model. In the synchronous variant, the client issues a request and waits until the reply arrives. In the asynchronous alternative, the client does not depend on the reply to be able to continue. It issues a request and immediately continues performing other work. The reply, or replies, are received later, at an indeterminate time. Asynchronous distributed rendering is a collaborative client-server setup in which both endpoints perform part of the rendering, even though the server is usually much more powerful computationally than the client. Due to the imbalance in capabilities, most of the work is assigned to the server. The challenge in this setup is to seamlessly introduce remotely computed data into the client's rendering pipeline. The data communicated to the client may be in the form of textures; this is referred to as texture streaming.

Master-Worker Clusters and grids typically make use of the master-worker paradigm, where a controlling machine enlists the help of multiple workers. This paradigm involves the scheduling of many small and similar tasks. Tile-based load balancing is a master-worker subclass.

Peer-to-Peer Peer-to-peer (P2P) strategies are collaborative methods that do away with a central authority. Peers communicate directly with each other, potentially reducing both computation and latency. P2P networks have a high degree of fault tolerance. Challenges in this area include discovery (detection of new or non-responsive peers), synchronisation and reaching consensus, and the timely sharing of data.

Hybrids Hybrid architectures use a combination of architectures. An example is the BitTorrent file-sharing software. It starts off as client-server when a specific file is requested, then uses peer-to-peer to obtain fragments of the file from multiple nodes.

4.2 Distributed Rendering

4.2.1 Challenges

The primary challenges in interactive distributed rendering systems are latency and bandwidth. Secondary challenges are synchronisation, scalability, and fault tolerance. Other typical challenges or features of distributed systems such as replication and security have their counterparts in distributed rendering too. Caching can be regarded as a form of replication, and as discussed in Section 3.1 is useful in many ways, for example stored results can later be used for interpolation or extrapolation. Caching previously rendered results can also be used to improve image quality, for example with smoothing functions or progressive updates. Koller et al. (2004) suggest a scheme that protects detailed models from piracy, where rendering is performed entirely at the secure remote end and only less-detailed models are communicated to the unsecured client end. Issues such as video compression efficiency (performance and compression ratios), image quality, and support for various resolutions and frame rates are derived challenges related to the computational capability of the hardware and to bandwidth.

Latency Beigbeder et al. (2004) studied the effects of packet loss and latency on the first-person shooter (FPS) game Unreal Tournament 2003. Players were mostly unaffected by packet losses up to 5%. Latencies of 100 ms were noticeable and significantly affected shooting accuracy. Claypool and Claypool (2006) confirmed the latency results, and set latency thresholds for several types of games, beyond which player performance degrades. They set thresholds of 100 ms for first-person shooter and car racing games, 500 ms for sports games and role playing games, and 1000 ms for real-time strategy games and simulations. These studies show that interactive response times should not exceed 100 ms when a high level of interactivity is required, such as in fast-paced first-person shooter video games. Slower-paced video games can make do with higher latencies, but although performance may not decrease, the game may start feeling sluggish well before these thresholds. Choy et al. (2012) broke down interactive response times into client processing, network latency, and server processing, optimistically setting a value of 20 ms for the combined client and server processing. They conducted a study on cloud-to-user latency in the US on Amazon's Elastic Compute Cloud (Amazon EC2), a component of Amazon Web Services, and found that the infrastructure was capable of providing a network latency of 80 ms or less to fewer than 70% of the 2,504 users in the test. They calculated that by adding a small number of edge servers to the existing infrastructure, user coverage would increase by an additional 28%, thereby accounting for nearly all the population.

Bandwidth Reducing bandwidth directly reduces costs. Image data can be compressed efficiently using encoders such as H.264 or HEVC. Stengel et al. (2021) use lossless encoding to obtain better image quality, at the cost of bandwidth. Distributed rendering solutions may also need to communicate other kinds of data such as floating-point depth data, which may not be compressed as efficiently as image data.

Synchronisation In collaborative systems, the nodes of a distributed rendering pipeline need to be kept synchronised. This is problematic as synchronising the system clocks at the nodes to the high degree of accuracy required (a few microseconds) is not possible. This is a well-known problem in distributed systems. Distributed solutions use logical clocks (Lamport, 1978), or dead reckoning (Pantel and Wolf, 2002).

Scalability Scalability refers to the ability of a system to manage more resources or clients efficiently. When a group of machines are working together towards a common goal, such as in a computational cluster, increasing the performance of the system simply by adding more machines is a desirable goal. The system needs to be designed carefully so that as it becomes larger and more complex, its management and the coordination tasks involved do not themselves become a bottleneck and slow the system down; this would have the opposite of the intended effect. In a centralised system, the ability of the server to handle multiple clients with sublinear scaling may result in increased profit margins for the provider and significant cost savings for both the provider and the clients.

Fault Tolerance When communicating over an inherently unreliable network such as the Internet, the ability to overcome short intermittent failures is a must. The same holds when operating over Wi-Fi or mobile networks; a poor signal may result in dropped packets and a general communication slowdown. Ideally the system would tolerate these situations to some extent and recover from them. In master-worker and peer-to-peer architectures, fault tolerance can also refer to the detection of non-responsive nodes or the management of untimely delivery of results. The system may tolerate these hitches gracefully and bring the main task to completion by reassigning a failed job to a different node, or by introducing a measure of redundancy by assigning the same job to multiple nodes.

4.2.2 Clusters

Interactive distributed ray tracing Wald et al. (2001b) developed a fast software ray tracer, RTRT (real-time ray tracing), by using SIMD (Single Instruction, Multiple Data) instructions via Intel's SSE[5] to operate on packets of four rays in parallel, and by optimising cache usage. Parallel operations include scene traversal based on an axis-aligned BSP (binary space partitioning) tree[6], ray-triangle intersection, and shading. Triangle intersection data and shading data are stored separately, while aligning the data to cache lines. Read-only data is stored separately from read-write data to reduce cache line invalidation. To exploit cache coherence and improve performance, the four rays within a packet are preferably chosen to have roughly the same origin and propagate in the same general direction. Primary rays benefit most from this scheme, but it is also effective for shadow rays and secondary rays. RTRT obtains large performance speed-ups of 11 to 15 over the Rayshade and POV-Ray ray tracers.

The computations needed for ray tracing are trivially (or embarrassingly) parallel and apply well to distribution. Wald et al. (2001a) modified the RTRT system to run on a cluster of seven dual Pentium IIIs to render complex models consisting of up to 50 million triangles. Without the SIMD code, as it was still being adapted for the distributed system, they reported three to five frames per second at a resolution of 640×480; they estimated around double these frame rates once the SIMD code was in place. The scene data was several gigabytes large. To avoid replicating this data on all the nodes, it was only stored on the master machine. The main challenge then was for the worker machines to access the shared scene data efficiently. In a preprocessing stage on the master, a high-level BSP tree is used to subdivide the scene adaptively until the tree nodes approximately contain a configurable number of triangle primitives. At this point each node represents a small, self-contained voxel of space. Another low-level BSP tree

[5] Streaming SIMD Extensions
[6] An axis-aligned BSP tree is also called a *k*-d (*k*-dimensional) tree.

within each voxel further subdivides the space. Each voxel is stored in a file and is around 250 KB before compression. The high-level BSP tree now only stores references to the voxels. It is communicated to each worker where it is used to keep track of the locally available voxels. Demand-driven load balancing is achieved by subdividing the image into 32 $\times$ 32-pixel tiles. When workers request tiles, the system tries to assign the same tiles to the same workers. This promotes better performance by maximising cache reuse on each worker. Workers buffer an extra tile to be able to continue working while a request for another tile completes. The workers also request any required voxels and cache them locally. The spatial subdivision performed in the preprocessing stage is required to achieve interactive rendering rates, but it also limits the system to static environments.

Instant Global Illumination (IGI) Wald et al. (2002) developed a distributed global illumination algorithm on a cluster of dual-processor commodity PCs. The method is a hybrid of several algorithms. Direct lighting, reflection and transmission are computed using fast (coherent) ray tracing (Wald et al., 2001b). Photon mapping (Jensen, 1996) is used for caustics and a variation of instant radiosity (Keller, 1997) is used to compute indirect illumination. Although GI algorithms use ray-based methods, distributed ray tracing (Wald et al., 2001a) cannot be used to compute indirect illumination due to fundamental differences between ray tracing and GI. At the desired frame resolution, the hardware used was capable of tracing 27 rays (ray segments not ray paths) per pixel at a frame rate of only 1 Hz. Such a small budget of rays may be sufficient for ray tracing, but is inadequate for estimating GI using standard Monte Carlo integration. The variance would be too large and the resulting images would be extremely noisy. Furthermore, GI algorithms are less suitable for distribution than ray tracing algorithms. Typically, GI algorithms obtain speed-ups by using data structures such as photon maps or caches. However, accessing and synchronising these data structures over a network nullifies the performance gains they provide. These constraints were overcome by using randomised quasi-Monte Carlo integration, interleaved sampling (Keller and Heidrich, 2001) and a discontinuity buffer.

Monte Carlo methods estimate a quantity from independent random samples, and the error can be estimated easily. In quasi-Monte Carlo (QMC), the samples are obtained in a more uniformly distributed way, causing the convergence rate for the estimate to be faster than in Monte Carlo, hence saving on computation while also reducing variance (noise). However, obtaining an estimate for the error is difficult. Randomised quasi-Monte Carlo remedies this problem by randomising the QMC samples (Owen, 1998).

To compute indirect illumination with instant radiosity, a number of VPLs are created in every frame and sampled from each pixel. The number of VPLs cannot be large as it would degrade performance. Even with a moderate number of VPLs, sampling them all from each pixel is also impractical. On the other hand, if all pixels sample the same small set of VPLs, aliasing artefacts would appear and the quality of the illumination would be poor. Similarly, good quality caustics require a large number of caustic photons, and the same drawbacks apply. To work around these issues, the image is split into small $m \times n$-pixel tiles, where typically $m = n = 3$ or $m = n = 5$. In every frame, a moderate number of VPLs and caustic photons are generated, say ≈100 VPLs from ≈25 light paths, and 500 to 1000 caustic photons per light source. The VPLs and caustic photons are divided into $m \cdot n$ sets. These sets are used for interleaved sampling, a cross between regular and irregular sampling, where the cells of a regular grid are sampled irregularly. Each pixel within a tile is assigned a different set of VPLs and photons from the caustics map. This arrangement is efficient, and although the same small number of VPLs and caustic photons are reused many times (once per tile), aliasing artefacts are replaced by structured noise, resulting in better image quality.

The discontinuity buffer, which also uses an $m \times n$-pixel area, is used to reduce variance, removing noise. The buffer stores information at each pixel, such as the distance to the surface point associated with that pixel, the surface normal, and the computed irradiance. Using this information, the pixel's irradiance is averaged with that of any neighbouring pixels that are determined to be associated with similarly oriented nearby surface points. This procedure smoothens the noise. Matching $m \times n$ areas need to be used for the combination of interleaved sampling and the discontinuity buffer to generate good results.

The load is balanced over a number of workers with each tile rendered by one worker. By using low-discrepancy sequences for random number generation, the workers can independently generate the same sets of VPLs and caustic photons, without needing to communicate with the master. Photon mapping is only used to visualise caustics (only a caustics photon map is used; the global photon map is unused). Since the photons related to caustics are typically highly localised, only small parts of the data structure need to be stored.

IGI was later refined by Benthin et al. (2003). Scalability bottlenecks were removed, significantly improving performance. The core algorithms were reimplemented using SIMD instructions to be able to use packets of four rays, exploiting coherence as in Wald et al. (2001b). Due to interleaved sampling, where a different set of VPLs is used for each pixel within a tile, constructing packets of primary rays from neighbouring pixels results in incoherent shadow rays. Primary rays were instead grouped by the interleaving set used to obtain coherent shadow rays. This resulted in a small (10% to 20%) reduction in performance for primary rays, but the overall performance increased due to the more significant gains obtained for shadow rays. The different stages in the ray-based pipeline were also reorganised. Instead of processing each packet of rays independently from the others, for each VPL set within a tile all primary rays were processed first and their results stored, then all shadow rays, then all secondary rays. This strategy was employed to maximise coherence and obtain further speed-ups on CPU architectures that support larger ray packets.

Photon mapping was dropped as it was deemed to have an excessive processing cost. The computations related to the discontinuity buffer were offloaded to the clients. This slightly increased client processing but lowered the server load and reduced network communication with the server significantly. Tone mapping was also delegated to the clients. The results were communicated back to the server and combined there. The server employed dynamic tone mapping, where the parameters used for tone mapping are updated in every frame for better visual quality. The parameters were communicated to the clients and used for the next frame. The frame of latency for tone mapping introduced in this way is negligible. The revamped system obtained speed-ups between 2.5 and 3.2 over IGI at a resolution of 640×480, and speed-ups up to 8.0 at 1600×1200.

4.2.3 Grids

Patoli et al. (2009, 2008) implemented a render farm on a desktop grid made up of seven machines within the University of Sussex. The machines were a mixture of single core and multi-core machines. The Condor (Litzkow et al., 1987) scheduling software was used, which includes a small monitoring software component that is installed on every machine to detect processor idleness, and makes the machine available for use on the grid, or immediately suspends its use on the grid. Rendering software was also installed on each machine, and a job submission tool was developed. Good speed-ups were obtained, with desktop grid-based rendering being 88% faster compared to rendering on a single-core machine.

Aggarwal et al. (2012) used a desktop grid for interactive high-fidelity rendering. Interactivity introduces time constraints, prohibiting the use of several fault tolerance strategies such as checkpointing and job resubmission. The authors used a special image reconstruction scheme based on quasi-random sampling to overcome this difficulty. Typically, a frame is subdivided into tiles, and each tile is submitted as a job to a worker. This is problematic because if a job fails, an entire region of the frame (a whole tile) would be missing. In this case, data reconstruction is not possible and a partial image would be immediately noticeable and jarring to the viewer. The authors work around this issue by instead subdividing a frame into sets of pixels that are chosen quasi-randomly, and submit these sets as jobs. Quasi-random pixel selection has two benefits. First, missing pixels due to a failed job can be reconstructed from the neighbouring pixels. Second, pixels are chosen in a seemingly random and irregular way, avoiding structured noise artefacts in the reconstructed image in case of missing data. The system waits for all the jobs belonging to a frame to complete. If a frame is still incomplete after a heuristically chosen time, missing pixels are reconstructed from an appropriately chosen group of neighbouring pixels. The system was tested on a desktop grid made up of 48 dual-core Linux machines and 8 quad-core Windows machines, connected over a 100 Mbps Ethernet LAN. A complex 861K-polygon scene rendered at around 3 Hz. A frame rate of up to 10 Hz was obtained for simpler scenes.

4.2.4 Peer-to-Peer

Bugeja et al. (2014a) use a peer-to-peer (P2P) network for collaborative rendering. The method is designed for a setup in which a number of peers are participating in the same virtual environment and each peer needs to render their immediate surroundings. If the peers are completely independent, when different peers access the same location, either contemporarily or at different times, they would perform the same computation. This redundancy may be removed if the peers communicate with each other, sharing their results. This frees up computation cycles, potentially enabling lower rendering times while improving the rendering quality. To keep track of computation results, a shared global state is maintained. Whenever a peer modifies the global state, the changes need to be propagated through the network to the other peers. Moreover, the order in which the updates are applied may be important. To ensure consistency in the global state, proper sequencing and merging of the updates is essential.

Significant changes in a peer's internal state result in observable events that are applied to the global state. Since the physical clocks on the peers are not guaranteed to be perfectly synchronised at all times, for proper ordering of events vector (logical) clocks and timestamps are used instead. Observable events are timestamped and propagated using a strategy based on epidemiology, where a peer updates or "infects" another peer chosen at random. Each peer maintains a list of peers that it knows about. The list is limited to a maximum number of peers and is initialised when the peer joins the network (to join in the collaboration, a peer needs to know at least one other peer on the network). When a peer updates another, they also merge their lists. In this way a peer learns about the other peers on the network.

As a case study, the irradiance cache (Ward et al., 1988) was adapted for P2P on a network consisting of eight peers. All machines ran the same operating system, had the same amount of memory and all were equipped with quad-core processors. However, the machines had a mixture of computational capability (presumably different microprocessor generations and clock speeds) to simulate an Internet-based P2P setup. Static scenes were used for the tests and the rendering was CPU-based. Significant speed-ups were obtained. When a peer joined a network with an already populated irradiance cache, two- to five-fold speed-ups were measured in rendering times, relative to the rendering times obtained by the peer rendering everything itself. When all the peers joined a network simultaneously, the average speed-up was less (1.17×). Better average speed-ups were obtained when the peers joined the network at staggered intervals (1.24× for 60-second intervals and 1.4× for 120-second intervals).

Serious games are useful tools in fields such as education and healthcare where they present an engaging game-like front while helping in teaching and therapeutical sessions. Bugeja et al. (2014b) developed a method based on instant radiosity by which users or peers sharing a virtual environment in a serious game, possibly using weak devices such as tablets and smartphones, may cooperate to precompute diffuse illumination to enhance visual realism. One of the peers is designated as the master while the other peers take on worker roles. The master subdivides the scene into sets of unique geometry vertices and assigns sets to the workers. Each worker computes irradiance at the given vertices from a number of VPLs that it generates. Workers need to generate exactly the same VPLs to ensure that no discontinuities in the illumination appear when the master merges the individual results. This is accomplished by using seemingly random but actually deterministic low-discrepancy sequences instead of pseudo random numbers during VPL tracing, so that each worker samples the same points on the light sources and computes the same directions. This mechanism avoids data transfers between the peers to synchronise VPLs. Each worker communicates its results back to the master and waits to be assigned a new task. When all the scene's vertices have been processed, the master communicates the computed irradiance values to all the peers. This completes the precomputation. During the game, when the peers render the environment, the irradiance stored at the vertices is interpolated and used to generate an indirect illumination estimate at a low cost. The method can also be used for geometry that is generated procedurally but deterministically. The method was tested on two tablets, obtaining a rendering rate of 11 to 31 frames per second.

		User Interaction	
		Restricted	**Unrestricted**
3D Model Data	**Static**	Remote visualisation	Virtual environment walkthrough
	Dynamic	Remote rendering of 3D video, animation	Cloud gaming

Figure 4.1: Classification of specialised remote rendering systems with sample applications. Adapted from Shi and Hsu (2015).

4.2.5 Remote Rendering

Shi and Hsu (2015) classify specialised remote rendering systems into four categories, by the type of user interaction allowed and the type of the 3D model data (Figure 4.1). We are interested in the combination that is used for cloud gaming, that is, unrestricted interaction, where the user has control over the camera, and dynamic 3D models. Cloud gaming, or game streaming, uses a dual-streaming client-server model. The video game is executed remotely on a cloud-based server and the game's output is communicated to a client device as a video stream. In this way, games can be played anywhere and on any device, from desktops to smartphones. The device is essentially a dumb terminal displaying video and continuously relaying input events back to the server.

Cloud-gaming services allow users to play games that are computationally intensive or have special requirements, such as GPU-accelerated ray tracing, without incurring the cost of buying new hardware or upgrading their equipment. Users never need to patch or update their games since this is automatically done on the server. Providers benefit by reaching more users and by the reduction in distribution costs. This model has proved immensely popular as it is easy to implement. There are numerous providers, including big names such as NVIDIA GeForce NOW, PS NOW, Google's Stadia, Microsoft's Xbox Cloud Gaming, and Amazon's Luna. These services are susceptible to bandwidth fluctuations and therefore require a stable network connection. The major players in the game-streaming market are indicated in Table 4.1 together with their bandwidth requirements.

Similarly to how YouTube videos are streamed, all services use adaptive bitrate streaming to adjust image quality depending on the bandwidth. The gaming experience

Table 4.1: Bandwidth requirements.

Service	Owner	Bandwidth (Mbps)	Resolution	FPS
Playstation Now	Sony	5	720p	60
GeForce NOW	NVIDIA	25 - 50	1080p	60
GeForce NOW	NVIDIA	15 - 25	720p	60
GeForce NOW	NVIDIA	10 - 15	720p	30
Xbox Cloud Gaming	Microsoft	10	720p	60
Stadia	Google	35	4K HDR	60
Stadia	Google	20	1080p HDR	60
Stadia	Google	10	720p	60

is arguably affected more by network latency than by degraded image quality. Network latency is the time taken for a packet of data to propagate from the client to the server and back. Lower latency implies more responsiveness; a typical value for low latency is 25 ms, whereas 100 ms is high. When the user provides some input, the effects of that input are expected to be visible on screen immediately. If the effects are only visible after a noticeable delay, say 150 ms, due to network latency and local and remote processing, the gaming experience is greatly affected, to the extent that the game may become unplayable. Due to these issues, although streamed games may be for the most part playable, they are not a complete replacement for local gaming, especially for experienced gamers.

4.2.6 Cloud-Based Collaborative Approaches

Multiplayer online games use a client-server model. All rendering and a portion of the game logic are executed on the client. The global state of the game is maintained and rectified by a remote central authority, the server, and propagated to all the players. Although the rendering itself is not distributed, it is controlled or affected by state changes received from the server. The schemes used for multiplayer online gaming and cloud gaming (Section 4.2.5) are polar opposites. In multiplayer online gaming all rendering is performed locally, whereas in cloud gaming all rendering is performed remotely. An amalgamation of the two methods, a dual stream model where the client and the server collaborate to produce the final rendered output has the potential to reduce the latency and bandwidth requirements in cloud gaming. Admittedly, such a system is far more complex to implement than the current remote rendering model. This section considers different approaches based on this idea.

Scalable remote rendering In remote rendering, all rendering is performed on the server. This limits the ability of the server to scale with respect to the number of simultaneously connected clients. Pająk et al. (2011) improve server scalability by offloading some of the server's workload onto the client device. The biggest expense on the server is due to shading, especially if ray tracing is used. To reduce this cost the server renders low-resolution frames. For each frame the server also obtains a set of high-resolution attribute buffers cheaply using deferred rendering (Deering et al., 1988). The attribute buffers consist of depth and motion data (motion vectors between the previous frame and the current frame). The attribute buffers are streamed together with the frame data to the client, where they are used for 3D warping, generating new scene views without having the geometric representation of the scene.

An edge-encoding scheme is used for the attribute buffers. Edges are identified as discontinuities in the buffers and stored at high precision, whereas low precision is used for the rest of the data. This sparse edge representation can be encoded efficiently and is amenable to good compression ratios. The client reconstructs high-resolution depth and motion data from the edge representation using a push-pull mechanism. The push stage serves to fill holes by reducing resolution. The pull stage propagates the filled holes back to the higher resolution image. Any remaining pixels are filled by averaging values from their neighbours. The high-resolution final image is reconstructed from the low-resolution frame and the reconstructed depth and motion data using spatio-temporal upsampling (Herzog et al., 2010).

Outatime Lee et al. (2015) use the standard cloud-gaming setup but attempt to reduce latency by monitoring user input patterns to predict input events in the near future. These predictions are communicated to the server, which generates multiple frames accordingly and streams them along with their depth information to the client where they are either used (potentially adjusted by 3D warping) or discarded if the predictions were incorrect. The system is effective at reducing latency but has a negative effect on bandwidth since multiple frames are transmitted.

CloudLight In the CloudLight system (Crassin et al., 2015), three global illumination algorithms were adapted for use on a distributed rendering pipeline. In each case indirect illumination is computed on the remote server and if the client is powerful enough, direct illumination is computed locally. The three algorithms used were cone-traced voxels, path-traced irradiance maps and photon mapping. The experiments were performed on a wide variety of client devices, categorised as low-powered ones (smartphones and some tablets), medium-powered ones (some tablets, head-mounted displays, laptops and some PCs), and high-powered ones (gaming PCs). Since the CloudLight paper was published, smartphone capabilities have increased and high-end smartphones can also be placed into the medium-powered device category. The experiments also sought to determine whether the server and the CloudLight infrastructure could scale to handle up to 50 simultaneously connected clients, while reusing computations in order to amortise costs between the clients.

Distributing the voxels algorithm between the server and the client device turned out not to be viable since the huge amount of data that needed to be sent to the client required bandwidths between 1.9 Gbps and 6.7 Gbps for the scenes used. The authors therefore rendered full frames on the server and streamed them as video to the client. This brought the mean bandwidth requirements down to between 3 Mbps and 6 Mbps and a maximum reported bandwidth of 15 Mbps. Since no computation was performed on the client device, the voxels algorithm was used for the least powerful clients. The system scaled well. One server GPU was capable of handling five clients at 30 fps and 25 clients at 12 fps.

The photon mapping algorithm produced the highest quality results but it required an excellent network connection and a very powerful client. The mean bandwidth requirements were between 16 Mbps and 43 Mbps. Due to the high bandwidth requirements the system scaled well up to around 30 simultaneous clients where the bandwidth limit was reached. Beyond this point, network congestion issues and increased latency became evident.

The best all-round results were obtained when using irradiance maps which encode indirect illumination data and are transmitted to the client as a texture stream. This method had very low bandwidth requirements (1.7 Mbps at most), was suitable for moderately powerful clients and scaled particularly well, efficiently handling all 50 available clients simultaneously. The downside of this method is that it requires laborious manual parametrisation of the scenes. Due to this limitation, the experiments could only be performed on three out of the six test scenes.

Kawahai Mobile platforms are computationally weak when compared to desktop systems. Games designed for these platforms may have to avoid using highly detailed models entirely, or may be forced to render at low frame rates. It is also common for games to allow the user to turn rendering features on or off, to cater for a variety of computation capabilities. Cuervo et al. (2015) argue that the gaming experience on mobile platforms can be improved by offloading some computation to a remote server. Two methods are presented. In the first, the client renders less-detailed geometry that is enhanced on the fly by data streamed from the server. The data contains differences between a low-quality frame and its high-quality counterpart; streaming these deltas is more bandwidth-efficient than streaming high-quality frames directly. The second method simulates the I-frames and P-frames that are used in video streaming. I-frames are key frames, large packets of data that contain an entire high-quality frame. P-frames are much smaller than I-frames. They contain deltas from the previous frame. The client renders high-quality frames (I-frames) at a slow rate and receives high-quality updates as P-frames from the server. By combining the two frame types as video decoders do, the client can reconstruct and present high-quality output continuously at a high frame rate. An advantage of this setup is that the system is fault tolerant. If a network outage occurs, visual quality suffers but the user can continue playing. This collaborative client-server approach can be utilised instead of standard cloud gaming saving bandwidth. The authors state that the system can obtain high-quality visuals using a sixth of the bandwidth needed for the thin client approach.

Proxy-guided image-based rendering Reinert et al. (2016) use image-based rendering to hide latency in a method targeted at weak mobile devices such as untethered VR headsets. To mitigate disocclusion artefacts the server sends two views of the scene as suggested by Mark et al. (1997), together with their depth maps and camera parameters. The views use a wide 180° hemispherical field of view. The depth maps contain inverted linear values to have higher precision for nearby objects, quantised to eight bits. The second view contains surfaces that are not visible in the primary view; surfaces already present in the primary view are stripped away with a depth peeling technique, avoiding redundancy and reducing transmission cost. The camera for the second view is placed at a constant offset from the primary camera (1.5 m in front, 1.8 m above, and tilted down 15 degrees). The offset was determined heuristically by analysing player movement in a number of scenes. Only vertical and forward/backward translation and tilt rotation angle are considered in the heuristic; sideways translation and rotation were excluded from the study. I would argue that these types of movements are too important to be sidelined, since looking to the left or right (sideways rotation) is common when exploring a virtual environment. Moreover, although moving towards rather than away from objects is the norm, the player hardly ever passes *through* objects, going around them (sideways translation) instead. The depth peeling technique has the disadvantage that it uncovers geometry that will never be visible, such as the ground sublayer. To save on computation the client renders proxy (simplified) geometry that is received in advance, shading each pixel from one of the received views. The correct view is selected by computing depth errors for the two views and testing against a threshold value. Using IBR the client generates novel views at high frame rates.

Remote Asynchronous Indirect Lighting (RAIL) Bugeja et al. (2018) developed a distributed rendering pipeline similar in concept to CloudLight (Crassin et al., 2015). Indirect illumination, the computationally intensive part of the pipeline, is delegated to the Cloud and direct illumination and image reconstruction are performed on the local device, which could be a low-powered device such as a tablet or a high-powered device such as a gaming PC.

In a preprocessing stage, the server generates a point cloud representation of the scene and partitions it into a regular grid with most cells containing 10 points or less. The server regularly samples indirect illumination at these points using a many-light method, progressively updating samples if the scene has not changed. The grid and the samples within it are transmitted to a client and are updated asynchronously. Clients reconstruct indirect illumination for static geometry using information in the grid, interpolating between the samples as necessary. The grid speeds up the interpolation process which makes use of nearest neighbour searches. For dynamic geometry, a coarser indirect lighting approximation is used, by means of a second regular grid superimposed over the scene called the ambient grid. Instead of points, each ambient grid cell contains an ambient term that is computed as a weighted mean from the point cloud samples that lie within the region occupied by the cell. Cells within the ambient grid that happen to be empty are populated by a propagation mechanism. When illuminating geometry using the ambient grid, an ambient occlusion computation is also performed to scale the illumination appropriately, so that for example, a heavily occluded point receives less light than a less occluded point. The reconstruction process on the client also makes use of smoothing functions to reduce flicker and to give the impression that indirect illumination updates are occurring at the same rate as the direct illumination updates.

RAIL has very low bandwidth requirements, less than 3 Mbps for the worst case. The solution only supports diffuse indirect illumination but is resolution invariant and supports deformable geometry. Moreover, the indirect lighting representation on the server is view independent and can be used by multiple clients. The system scaled well when tested with 24 clients and the image quality was moderately good.

On the desktop, frame rates exceeding 60 Hz were obtained at a resolution of 2560×1440. On the tablet, a frame rate of 25 Hz was obtained at a resolution of 1024×768. The system effectively eliminated input lag but increased output lag with one of the smoothing functions. At 6 Hz, indirect light updates, which require a request from the client, are quite slow. By replacing the mandatory requests from the client by continuous, periodic server updates, the frequency of the updates should improve. However, in all likelihood this would have a significant negative impact on bandwidth requirements which is one of the strongest features of the method.

Shading Atlas Streaming (SAS) Mueller et al. (2018) designed a system that streams high-quality visuals to untethered VR headsets, while minimising perceived latency. As in CloudLight (Crassin et al., 2015) and RAIL (Bugeja et al., 2018), it is a remote rendering approach that splits the rendering pipeline between the server and the client, but it does so in a different way. All the shading is performed on the server using rasterisation and the data is transferred to the client as a texture stream using standard video compression techniques. The client renders the scene using the received textures and is able to generate images for slightly shifted viewpoints. SAS uses motion prediction to estimate new viewpoints and transmits the resulting data to the client. This helps reduce artefacts caused by parts of the scene becoming unoccluded when the viewpoint changes.

Frequent updates from the server are normally welcomed, especially if the illumination changes rapidly, because they improve the synchronisation between the client and the server. The authors note that less frequent updates may be beneficial. If the server has more time to prepare a frame, the quality of the frame is better. Since less data is sent to the client, less bandwidth is used. Moreover, the client's workload is reduced since fewer frames result in less cycles dedicated to video decompression or to the updating of local data structures; this is particularly significant if the client is computationally weak.

SAS computes a potentially visible set (PVS) of patches, which are made up of two or three adjacent triangles and sometimes, in the worst case, a single triangle. The patches are shaded and packed into an atlas, a large texture. The texture is compressed using H.264 video compression and streamed to the client using RTP over UDP. Vertex, triangle and texture coordinate information is also transmitted but in a separate channel, over TCP. The vertex and triangle data are only transmitted once, the first time they are needed. Patches in the atlas may be relocated; the updated texture coordinates are transmitted whenever this happens. As in DARM, since the data is sent to the client over two independent channels, proper synchronisation of the channels is essential. To help in this regard, I-frames are sent regularly, every 5 frames. Relocations within the atlas only occur at this time. Unlike DARM, the atlas is not part of a bigger megatexture, therefore no mapping information needs to be sent to the client. In SAS, efficient packing of patches (one, two or three adjacent triangles) is achieved by subdividing the atlas into square superblocks which are made up columns of equal widths. The columns are further subdivided into blocks of equal widths but different heights. Patches are transformed into blocks and placed in the atlas in a block with the most closely matching dimensions. This indicates why single-triangle patches should preferably be avoided, because they waste some space.

SAS is fast, and the client frame rate, which is unlocked, maintains a steady 120 Hz. However, the method cannot display realistic light phenomena. The authors give suggestions on how effects such as transparency can be added, but the proposed solutions are typical of rasterisation techniques, complex and not physically correct. SAS can be regarded as a real-time rendering method but it is not appropriate for global illumination.

Tessellated Shading Streaming (TSS) Hladky et al. (2019b) aim to bring high-fidelity visuals to smartphones, tablets and untethered head-mounted displays while minimising latency. The method employs a client-server architecture. The back-end employs rasterisation techniques and streams textures to the client device. The novelties of the system include the use of hardware-accelerated tessellation to create a high-quality representation for triangles in a texture atlas. The triangles stored in the atlas are obtained by computing a potentially visible set of triangles from four reference views around the location of interest; this ensures that novel views rendered from nearby locations are complete and hole free (Hladky et al., 2019a).

On the server, shading information for a triangle primitive is obtained by sampling the triangle at various points. The number of samples used is proportional to the on-screen size of the triangle. The location of the samples is determined by tessellating the triangle, creating new vertices around and inside the triangle. The triangle is sampled at these vertices. The triangles are cut along a subsection and unfolded into an L-shape. The quad structure of the samples produced in this way enables the use of bilinear interpolation between the samples. A mapping function is constructed between the barycentric coordinates of a sample and the position of the sample within the L-shape. The L-shape also allows for efficient packing of the samples within an atlas; every other L-shape is flipped so that the L-shapes fit together perfectly. Triangles that are nearly equilateral are well-suited for conversion into an L-shape. Triangles shaped like long slivers are processed differently, using a method called oversampling which also uses tessellation. When all the triangles are processed, the atlas is compressed using JPEG encoding and transmitted to the client.

The method is not suitable for adaptation to techniques that benefit from progressive updates (such as instant radiosity or path tracing) which reduce the rendering time for a single frame but allow updates to accumulate over frames, improving image quality. Since the atlas is regenerated for every frame, a history of samples is not maintained and therefore progressive updates cannot be used. Bandwidth usage is high; a total of 44 Mbps is needed, split as 4 Mbps for meta data and 40 Mbps for the compressed atlas. This does not make it particularly well-suited as a cloud-based service.

Distributed dynamic light probes Stengel et al. (2021) stream irradiance data. Direct illumination is computed on the client. Diffuse global illumination is computed using irradiance volumes on the server and streamed losslessly to the client. Their prototype is capable of streaming thousands of irradiance probes per second but requires up to 50 Mbps of bandwidth when streaming at 60 frames per second. Since the system computes irradiance rather than full frames, the computation can be amortised between several simultaneous users. This approach splits the graphics pipeline into a view-dependent part, computed on the client, and a view-independent part, computed on the server. Visibility information is also streamed. Reliable UDP over ENet is used. The irradiance probes use an octahedral mapping to encode directional light information. Each probe also stores mean distance and mean squared distance; these are used to produce visibility weights during shading. High dynamic range video compression using HEVC is used. A potentially visible set of probes is estimated by rendering a spherical view at the client's position. Full volume updates occur at 10 to 30 Hz.

4.3 Summary

This chapter started off by briefly introducing distributed computing, describing various hardware setups and system architectures. The challenges of distributed rendering were then outlined. A literature review of distributed rendering methods (Table 4.2) followed. The methods covered include those tailored for clusters, grids, and peer-to-peer systems. The remote rendering paradigm was discussed, after which special focus was given to cloud-based collaborative approaches, all of which use a client-server architecture. The methods in this last group support a wide variety of client devices and include cases where the server scales to support a number of simultaneously connected clients.

Category	**Method**
Clusters	Wald et al. (2001a)
	Wald et al. (2002)
	Benthin et al. (2003)
Grids	Patoli et al. (2009, 2008)
	Aggarwal et al. (2012)
Peer-to-Peer	Bugeja et al. (2014a)
	Bugeja et al. (2014b)
Cloud-Based Collaborative	Pająk et al. (2011)
	Lee et al. (2015)
	Crassin et al. (2015)
	Cuervo et al. (2015)
	Reinert et al. (2016)
	Bugeja et al. (2018)
	Mueller et al. (2018)
	Hladky et al. (2019b)
	Stengel et al. (2021)

Table 4.2: Distributed rendering methods.

5 Regular Grid Global Illumination

This chapter describes a distributed rendering pipeline that enables a limited form of global illumination at interactive rates on weak devices. In Section 2.3, the rendering equation was expressed as the infinite series

$$L = L_e + TL_e + T^2 L_e + T^3 L_e + \dots \tag{5.1}$$

where the first term is the emitted light, the second term is the direct illumination and the higher order terms are the indirect illumination. If light sources are limited to point lights and only diffuse materials are used, direct illumination can be computed with rasterisation techniques quite efficiently and accurately even on low-powered devices such as smartphones. The estimation of indirect illumination is vastly more computationally intensive. If this part of the rendering equation is computed on more capable hardware and the result propagated to the weak device, global illumination can be reconstructed on the device itself by combining the local and remote computations.

The proposed method, Regular Grid Global Illumination (ReGGI), makes use of the fact that plausible indirect illumination can be computed from sparse irradiance samples using interpolation (Ward et al., 1988). Irradiance samples are computed at run-time using the concept of virtual point lights (VPLs) from instant radiosity (Keller, 1997), and are progressively refined over time to enhance image quality. Smoothing functions are employed to improve temporal coherence, reducing flickering artefacts.

ReGGI is introduced in Section 5.1 with a high-level overview. A detailed description follows in Section 5.2, where the method's architecture is presented and the workings of the various components are explained. The performance of the method, the image quality it provides, its scaling capability, and its bandwidth requirements are evaluated in Section 5.3.

5.1 Introduction

ReGGI distributes the computation of global illumination between the Cloud and the local device. It requires no scene annotations, has low bandwidth requirements and completely eliminates input lag, but may produce output lag, where indirect light trails behind direct light. In general, image quality is close to that produced by instant radiosity (Keller, 1997).

ReGGI makes use of a regular grid to spatially subdivide the scene. Indirect lighting in the form of irradiance is stored in grid cells that contain geometry. The grid is constructed and populated on the remote end of the pipeline, the server. Multiple clients sharing a virtual environment can connect to the server. Each client maintains a local copy of the grid, updating it with data received from the server. The clients reconstruct global illumination by interpolating indirect illumination from the grid and adding it to direct illumination that is computed locally. The grid enables quick nearest-neighbour lookups for the interpolation algorithms. The contributions of the method are:

- A scalable cloud-based global illumination solution that requires little bandwidth and no precomputation and is suitable for weak devices such as smartphones.
- Elimination of input lag.
- Amortisation of server-side computation over multiple connected clients.

5.2 Method

The server voxelises the scene onto a *world grid* and gathers up to two indirect light samples in every cell that contains geometry. A snapshot of the world grid is maintained on the client, in CPU memory, and is populated asynchronously with data received from the server. In addition, the client creates a *sampling grid*, a subgrid of the world grid, in GPU memory, and periodically updates it from the world grid. The dimensions of the sampling grid are configured on the client itself. It represents the neighbourhood of the player and moves with the player throughout the scene. The sampling grid avoids excessive CPU-to-GPU data transfer since only the data within its bounds is transferred instead of the entire world grid data. The resolution of the world grid needs to be chosen with care to balance image quality, reconstruction and bandwidth requirements. A higher resolution world grid produces high quality images but equates to a higher

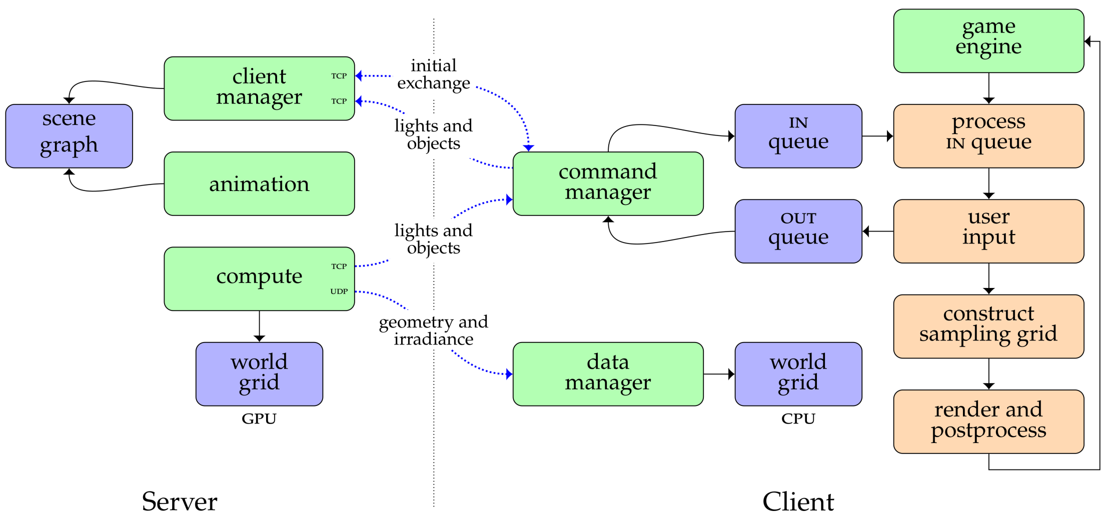


Figure 5.1: ReGGI architecture.

bandwidth cost and higher computation and reconstruction costs on the server and client respectively.

The architecture of the method is shown in Figure 5.1. The server and the client both have a local copy of the scene description. At startup, the server loads the scene description, creates the world grid data structure and voxelises the scene (Section 5.2.1) to seed the world grid with geometry data. For static scenes, where the geometry does not change, this is the only time that voxelisation is performed. The world grid is also seeded with illumination data at this point (Section 5.2.2 and Section 5.2.3). The server launches its *client manager*, *animation* and *compute* threads and waits for clients to connect.

The client manager thread handles client connections and reliable two-way communication with the clients. When clients connect, an initial exchange takes place where they receive information about the world grid and the dynamic lights and objects in the scene. After the initial exchange, clients use the reliable channel to communicate updates related to any lights and objects they control; the server updates its scene graph with this information. The animation thread handles server-controlled modifications to lights and geometry and updates the scene graph accordingly. The compute thread, outlined in Figure 5.2, performs the main server processing. It applies any modifications to the scene graph, voxelises the scene if any geometry has changed (Section 5.2.1), gathers irradiance (Sections 5.2.2, 5.2.3 and 5.2.4) and communicates geometry and illumination information to any connected clients (Section 5.2.5).

When a client starts it loads the scene description and connects to the server, triggering an initial exchange. During the exchange, the client receives meta scene information

```
1: procedure COMPUTE
2:     loop
3:         ApplySceneUpdates
4:         VoxeliseScene
5:         procedure GATHERIRRADIANCE
6:             GenerateVPLs
7:             AccumulateContributions
8:             RefineSamples
9:         end procedure
10:        procedure TRANSFERDATA
11:            PrepData // reap, encode, compress
12:            SendData // network transfer
13:        end procedure
14:    end loop
15: end procedure
```

Figure 5.2: The server's *compute* thread.

which it uses to create its world grid. The client launches its *command manager* and *data manager* and informs the server that it can start receiving updates, also specifying the UDP port to which the updates should be sent. The command manager is used for reliable two-way communication with the server. The client uses *commands* to notify the server of client-initiated changes to light sources, geometry or the camera. Commands are placed in an OUT queue; the command manager retrieves them and communicates them to the server. The command manager also receives commands from the server, specifically notifications of server-initiated changes to light sources or geometry, and places these commands in an IN queue for later consumption by the client. The data manager receives voxelisation and illumination data from the server. These updates are not reliable in the sense that packets may be dropped, duplicated or arrive out of order. The data manager decompresses and decodes the data and updates the world grid.

The client computes direct lighting using rasterisation methods and reconstructs global illumination as outlined in Figure 5.3. The reconstruction is performed in a shader that runs as a postprocessing effect on the GPU. In every frame, the sampling grid is constructed and copied to the shader. For every pixel, the corresponding grid cell and a number of neighbouring cells are identified (line 3). Irradiance is computed by interpolating from the values in these cells (line 4). The interpolation is actually computed at a low resolution and upscaled when merging with the locally computed data. Indirect lighting is obtained by combining albedo and irradiance (line 5). The diffuse BRDF is obtained by dividing the albedo by π. Multiplying the BRDF by the irradiance produces the radiance value for indirect light. Finally, direct and indirect light

```
1: procedure ReconstructImage($G_a$, $G_d$, $G_n$)
2:     for all p ∈ pixels do
3:         $C \leftarrow$ GetCells($G_d$, p)
4:         $E \leftarrow$ ComputeIrradiance($C$, p, $G_d$, $G_n$)
5:         $I_i \leftarrow$ ComputeIndirect($G_a$, E)
6:         GI $\leftarrow$ Merge($I_d$, $I_i$)
7:         TGI $\leftarrow$ ToneMap(GI)
8:     end for
9: end procedure
```

Figure 5.3: Image reconstruction on the client. G_a, G_d and G_n are the *albedo*, *depth* and *normal* geometry buffers respectively. E is the irradiance, I_d and I_i are the direct and indirect illumination respectively. GI is the global illumination image, and TGI is its tone-mapped counterpart.

are merged with a simple addition to obtain global illumination (line 6). Tone mapping (line 7) completes the reconstruction procedure.

5.2.1 Scene Voxelisation

On the server, the powerful back-end of the pipeline, indirect light is sampled throughout the entire scene. The scene's volume is partitioned into a regular grid made up of cubical cells called the *world grid* and a crude approximation of the scene's geometry is computed. The resolution of the grid is configured by specifying the number of cells along one of the principal axes or along the principal axis where the scene extents are largest (*DimX*, *DimY*, *DimZ*, or *MaxDim* respectively). Figure 5.4 shows a visualisation of three grid resolutions where the number of cells is specified along the x-axis.

The presence of geometry within a grid cell is determined by firing a number of regularly spaced rays within it, first between the front and back faces (parallel to the

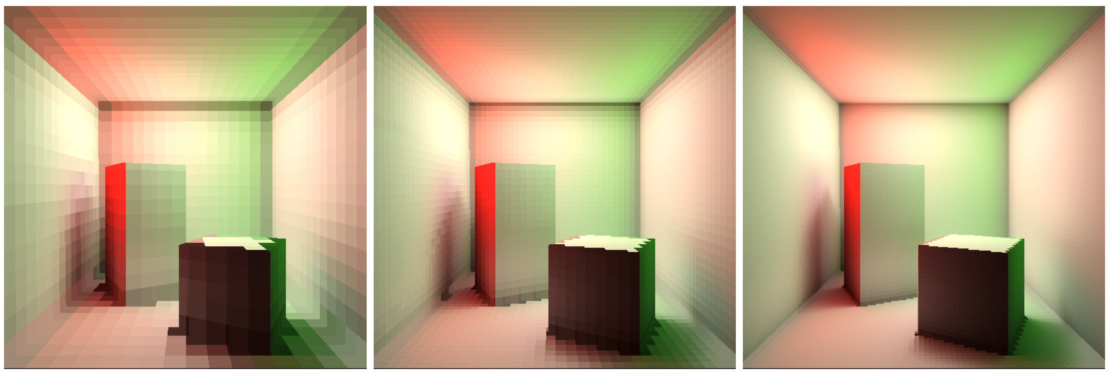


Figure 5.4: Grid visualisation. From left: *DimX* = 16, 32, 64.

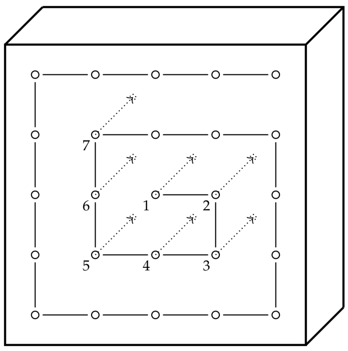


Figure 5.5: The spiral sequence of rays fired during geometry detection, shown here with the first 7 rays out of a possible 25 ($n = 5$), between the front and back faces of a cell.

z-axis) in both directions, then in a similar way between the bottom and top faces and between the left and right faces. In each of these six sets of rays, the first ray is fired from the centre of the face to the centre of the opposite face; the following rays are fired in a spiral pattern, spiralling outwards from the face centre as indicated in Figure 5.5. The maximum number of rays fired from a particular face is n^2, where n is an odd number. This results in a maximum of $6n^2$ rays per cell. For low grid resolutions, for example when using a *MaxDim* value of 50 for a large scene, we set n equal to 9, resulting in 486 rays per cell in the worst case. Such a large number of rays per cell reduces the chances of missing geometry within the cell. If the grid resolution is high, n can be set to a much lower value. In scenes with dynamic geometry, where very fast voxelisation is required, a small value of n may be needed, possibly resulting in undetected geometry. Each ray is tested for closest-hit intersections. Hit points whose surface normal is in the same general direction as the ray (the dot product between the two directions is nonnegative) are interior points of an object and are ignored. When the first hit point is obtained, the cell is marked as a geometry cell, the hit point is stored as the cell's primary representative point and processing moves on to the next face. A new hit point is categorised as either having a normal compatible to that of the cell's primary representative point or not. In our evaluation, we consider two normals to be compatible if the dot product between them is ≥ 0.1. The first hit point with an incompatible normal becomes the cell's secondary representative point. Geometry detection within a cell terminates early when a secondary representative point is detected. Otherwise, processing for a cell terminates when the configured number of rays is reached; such a cell either has a single representative point or no representative points at all. The surface normals at the representative points are also stored.

We originally planned to store only one sample per cell, reasoning that we could simply increase the resolution of the grid to capture more detailed geometry. However, for very thin objects the required grid resolution becomes too high to be practical. Even worse, representing both faces of an infinitely thin object such as a plane using a single sample per cell is not possible, no matter what the grid resolution is. Using a maximum of two samples per cell mitigates the first problem and solves the second.

The surface points hit are categorised by normal direction and sorted according to proximity to the centre of the cell. If there is at least one hit point, the cell is marked as a geometry cell. If there are multiple hit points, only a maximum of two points are selected; the two points are chosen so that the surface normals are in a generally opposing direction. This enables the storage of illumination information for very thin surfaces that can be viewed from either side. Points closer to the centre of the cell are preferred; located towards the middle of the cell, they are likely to better represent multiple surface points within the cell. The selected points are called the representative points of the cell; indirect light will be sampled at these points.

5.2.2 VPL Generation

Photons are fired from the light sources and the resulting light path is followed for several bounces, creating a virtual point light (VPL) at each path vertex. Each of these light paths is allowed to proceed for three segments, after which the path is terminated at random using Russian roulette (Arvo and Kirk, 1990). VPLs are not created at the origin of the path, on the light source itself, because only indirect illumination needs to be estimated. However, the radiance at the origin of the path needs to be calculated. This radiance value will be attenuated at every path vertex and assigned to the VPL at that point. In general, if the radiance of a light source is L_{light}, the radiance at the origin of the associated light path is

$$L_0 = \frac{L_{\text{light}}}{p_{\text{lightPos}} \cdot p_{\text{light}}}, \tag{5.2}$$

where p_{lightPos} is the probability of picking a random point on the light ($p_{\text{lightPos}} = 1$ for point lights) and p_{light} is the probability of picking that particular light ($p_{\text{light}} = 1$ if there is a single light source in the scene). At the next vertex, where the first surface interaction occurs, the radiance is calculated as

$$L_1 = f_r \, \frac{L_0 \cdot s}{p_{\text{dir}}}, \tag{5.3}$$

where $f_r = \frac{\rho_d}{\pi}$ for diffuse surfaces, s is a scaling factor and p is the probability of firing a ray from the previous vertex in the direction of the current vertex. The value of s

depends on the type of light source. For point lights and spherical lights, $s = 1$. For quad, triangular and directional lights, $s = (\cos\theta)^+ = \max(0, \cos\theta)$. For spot lights, the *falloff* value is used, calculated as in Pharr et al. (2016). At all the other vertices the radiance is calculated as

$$L_k = f_r \, L_{k-1} \, \frac{(\cos\theta)^+}{p_{\text{dir}}}, \; \forall k >= 2, \tag{5.4}$$

where L_{k-1} is the radiance at the previous vertex, θ is the incident angle relative to the current vertex and $(\cos\theta)^+ = \max(0, \cos\theta)$. If N is the number of VPL light paths, the radiance L_{vpl} assigned to a VPL is actually $\frac{L_j}{N}$, $\forall j >= 0$. This makes the contribution of each light path equally important.

5.2.3 Gather

A ray-based implementation of instant radiosity is used to simulate indirect diffuse illumination. The representative points in the geometry cells store irradiance. A VPL contributes towards the irradiance at a representative point P if there is a clear line of sight between the VPL's position P_{vpl} and P. This occlusion test is performed using shadow rays as in recursive ray tracing. A VPL's contribution E is computed using the area form of the rendering equation,

$$E = G \, V(P_{\text{vpl}}, P) \, L_{\text{vpl}}, \tag{5.5}$$

where the geometry term G is defined as

$$G = \min\left(\frac{(\cos\theta_P)^+ (\cos\theta_L)^+}{d^2}, \text{c}\right) \tag{5.6}$$

and the visibility term V is defined as

$$V(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} 1, \text{if } \boldsymbol{x} \text{ and } \boldsymbol{y} \text{ are mutually visible} \\ 0, \text{otherwise.} \end{cases} \tag{5.7}$$

θ_P is the incident light angle at P, θ_L is the angle the outgoing light ray makes with the normal at P_{vpl}, d is the distance between P and P_{vpl} and c is a user-defined constant used to clamp large values of G due to the weak singularity produced by the d^2 term in the denominator.

5.2.4 Progressive Refinement

In every iteration of the rendering loop on the server, new VPLs are created and the irradiance samples are updated progressively, accumulating the values and averaging

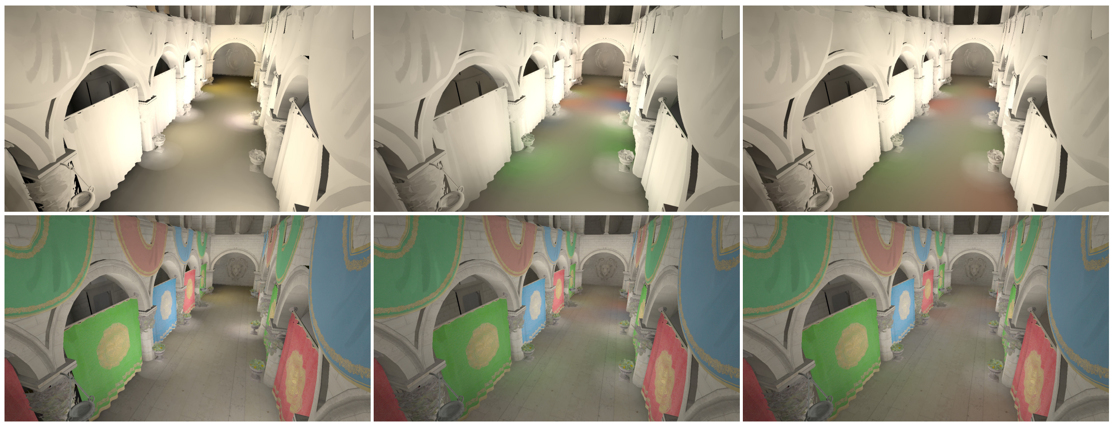


Figure 5.6: The scene, which is illuminated by 3 spotlights facing the floor, is progressively refined by maintaining a running average of irradiance estimates. From left: frames 1, 4 and 64 (16, 64 and 1024 VPLs respectively). Top: irradiance. Bottom: indirect illumination.

them (see Figure 5.6). In this way, a small number of VPLs can be created in every iteration, enabling quicker completion of the iteration and hence shorter rendering times. If the scene has remained static since the last iteration (no changes in geometry or light sources have occurred), the new iteration improves the quality of the samples by updating them rather than overwriting them. New irradiance values are combined with the old values using Welford's method (Welford, 1962), maintaining a running average,

$$M_1 = x_1, \tag{5.8}$$

$$M_k = M_{k-1} + \frac{x_k - M_{k-1}}{k}, \ \forall k > 1, \tag{5.9}$$

where M_k is the new mean, M_{k-1} is the old mean, x_k is the new sample and k is the total number of samples. If the scene changes, the accumulation of values restarts. When this occurs, the number of samples is not reset to zero. It is set to one and the value of the current mean is retained, becoming M_1, the first mean. This reduces the weight of the mean so that further updates will still have a significant effect on it, while smoothening the transition to new values.

5.2.5 Data Transfer

All connected clients are provided with irradiance updates continuously rather than on request as in RAIL (Bugeja et al., 2018). Only modifications to the samples at the representative points are communicated, making the method particularly efficient for static scenes. Since the sample deltas become smaller and smaller as the solution converges, after a while the deltas either become zero or they become smaller than a threshold value. In either case no further updates are sent. The data is first compressed by a data-aware scheme and then more generally using zlib set to the *best speed* compression level. In the data-aware scheme, normals and (indirect) irradiance values are encoded as 16-bit float triples since this is sufficient precision for these kinds of values.

To avoid fragmentation and reassembly processing on routers along the network path from the server to the clients, packets are limited to a maximum data payload size of 1,472 bytes. The packets are sent over UDP and no measures to ensure delivery are performed. The information associated with each cell consists of the cell's 30-bit integer index, two bit flags, the normal at the hit point and the irradiance value. One of the flags indicates the presence of a sample within the cell and is required to support dynamic geometry. The second flag identifies the sample (a cell can contain at most two samples). A zero irradiance value for a cell does not necessarily indicate that the cell does not contain geometry; the cell may simply not be illuminated in any way. To save on bandwidth, the actual positions of the samples are not transmitted. At the client, the samples are assumed to be located at the centre of the cell.

Since the application of received updates is somewhat costly on the client side, the updates can be accumulated and applied periodically *en masse*, according to some configured update interval. For powerful clients, such as laptops, the updates would normally be applied immediately. This is equivalent to specifying an update interval of 0 ms. For low-powered devices, an update interval of 125 ms (effectively a maximum of 8 batch updates per second) is usually used.

5.2.6 Smoothing

Clients maintain a cache of the world grid on the CPU. This cache is updated asynchronously with incoming data from the server. To reduce flickering and improve temporal coherence, two smoothing algorithms are used, as in RAIL (Bugeja et al., 2018). The first smoothing function weights the old and new irradiance values before combining them:

$$E_i = w_{\text{old}} E_{\text{old}} + w_{\text{new}} E_{\text{new}} \tag{5.10}$$

where $w_{\text{old}}, w_{\text{new}} \in [0, 1]$ and $w_{\text{old}} + w_{\text{new}} = 1$. w_{old} and w_{new} are typically set to 0.5. High values for w_{old} result in smoother transitions but the system is less responsive to quick changes in illumination. The second smoothing function makes use of the estimated time between updates, T; this value is continuously revised as the updates arrive. The irradiance value E for a cell is computed as

$$E = E_i + \frac{t}{T}\left(E_{i+1} - E_i\right) \tag{5.11}$$

where E_i and E_{i+1} are the irradiance values from the last two updates and t is the time since the last update. If $t > T$, t is set equal to T. Usually, direct illumination is computed at a faster rate than indirect illumination. The second smoothing function breaks down a single indirect illumination update into several smaller updates, causing indirect light to appear to update at the same rate as direct light. However, this increases output lag as instead of applying the full update immediately, it is applied in steps.

5.2.7 Interpolation

On the world grid, all the irradiance samples are stored at the centres of geometry cells. Irradiance values at intermediate points are obtained by interpolating from neighbouring samples. To improve performance, clients use a *sampling grid*, a subset of the world grid that encompasses the immediate surroundings of the client's camera. Unless otherwise stated, the sampling grid is implemented as a compute buffer.

Trilinear Interpolation

Interpolation algorithms such as trilinear interpolation require the presence of valid values on either side of an intermediate point, in all three dimensions. This requirement may not be satisfied at all locations in the sampling grid. Since not all cells contain geometry, samples may be missing in certain regions. Moreover, the presence of a sample does not imply that it is usable for a particular intermediate point. The sample's normal and the normal at the intermediate point need to be compatible; otherwise, the interpolated value computed would be incorrect. Due to these issues, standard trilinear interpolation cannot be used. In this section we present two adaptations of the trilinear interpolation algorithm that can still operate when these issues are encountered.

Variant 1 (TL1) To work around gaps in the data, the standard trilinear interpolation algorithm is modified by introducing the notion of invalid values which are handled as follows. Let a and b be reference points on either side of an intermediate point p and let a, b and p be collinear. If a and b both contain valid values, the value of p is obtained by interpolation as usual. If only one of a or b contains a valid value, p is assigned that value. Otherwise, both a and b contain invalid values and p is marked invalid as well. In this way the algorithm can proceed to completion. If the end result of the algorithm

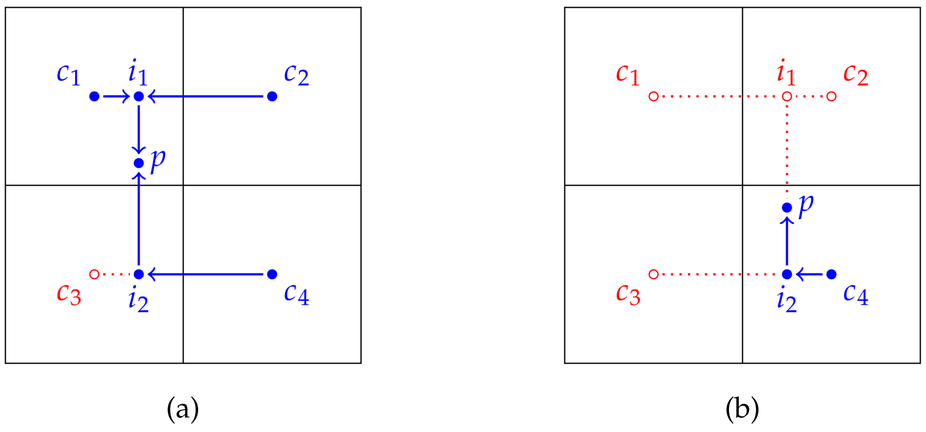


Figure 5.7: Visualisation of TL1 in 2D showing valid values as solid circles and invalid values as circle outlines. The value at point p needs to be calculated. (a) Interpolating c_1 and c_2 yields i_1. Interpolating c_3 and c_4 is not possible; i_2 is assigned the value at c_4. Interpolating i_1 and i_2 yields p. (b) i_1 is assigned an invalid value, i_2 is assigned the value at c_4 and p is assigned the value at i_2.

is an invalid value, an irradiance value of zero is assigned to the intermediate point. Figure 5.7 illustrates how the algorithm works in 2D.

Variant 2 (TL2) This variant is faster than the first but results in reduced image quality. The sampling grid is implemented as a 3D texture instead of a compute buffer. The 3D texture only contains irradiance values; no information about normals is stored, implying that normals are unused during the interpolation. Each cell in the sampling grid is populated with at most one irradiance sample. If the corresponding world grid cell contains two samples, a sample for the sampling grid is generated by choosing the sample with the larger magnitude. An alternative heuristic would be to use the average of the two samples; an indication of the results obtained in this way is displayed in Table 5.4. When using this variant, the server removes gaps in the data in the world grid by creating virtual geometry cells around the detected geometry cells. Virtual geometry cells only have one representative point located at the centre of the cell. The irradiance sample for a virtual geometry cell is duplicated from one of the neighbouring geometry cells and is chosen as follows. All the neighbouring geometry cells are processed in turn. For each representative point in the cell, the direction vector towards the centre of the virtual geometry cell is computed and compared to the representative point's normal. At the end of the process, the cell containing the normal with the closest match is selected.

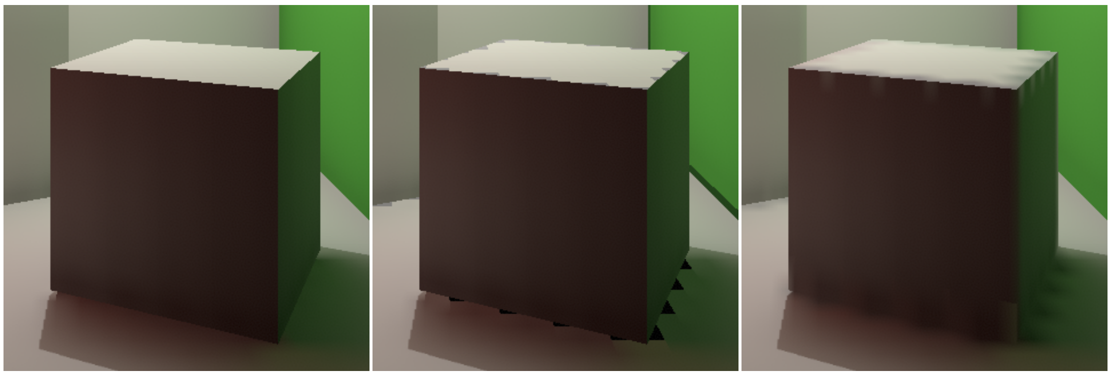


Figure 5.8: Trilinear interpolation. Left: TL1 (up to two samples per cell, normal checks enabled). Centre: One sample per cell, normal checks enabled. Right: No normal checks.

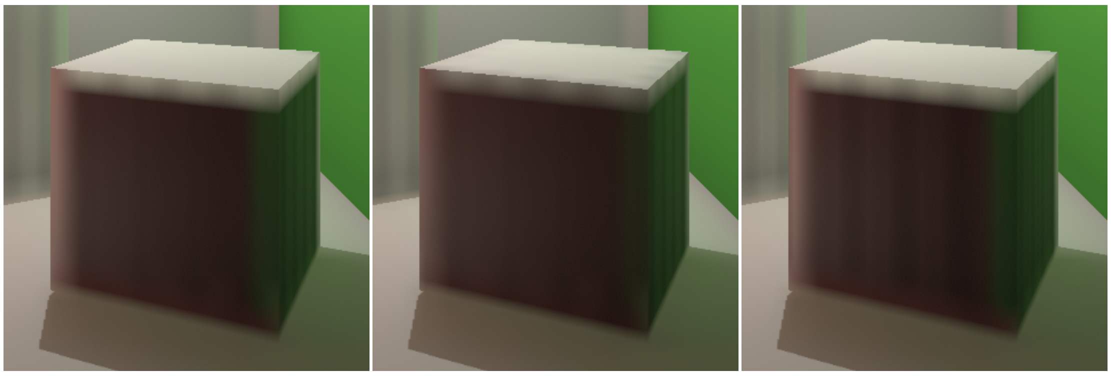


Figure 5.9: Trilinear interpolation. Left: TL2 (using the sample with the larger magnitude, virtual cells enabled). Centre: Using an average of two samples, virtual cells enabled. Right: Using the sample with the larger magnitude, no virtual cells.

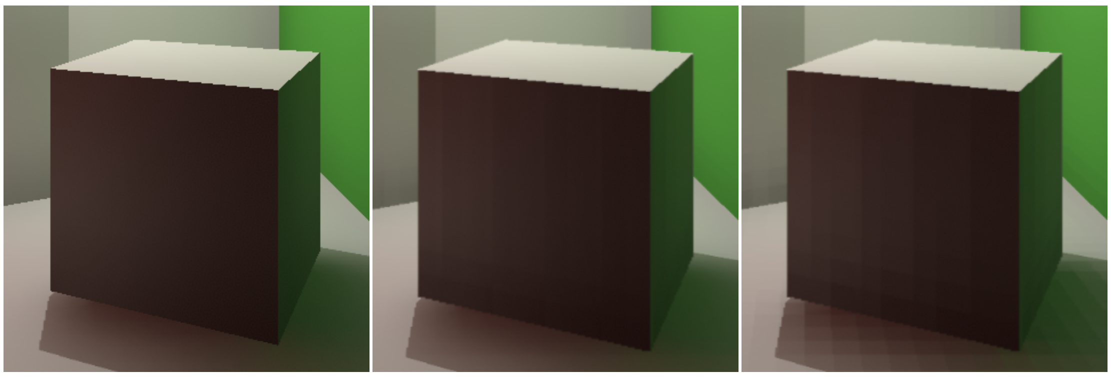


Figure 5.10: Modified Shepard variants. Left: MS1. Centre: MS2(25). Right: MS2(8).

Figure 5.8 illustrates TL1 (left) and the effect of disabling some of the algorithm's features. The image at the centre illustrates the effect of using a single sample per cell instead of a maximum of two; there are not enough samples and artefacts appear at the edges. The image on the right illustrates the effect of disabling normal checks. This causes the algorithm to make use of the first cell sample only; if there is a second sample it is always ignored. The artefacts now appear fuzzy. Figure 5.9 illustrates TL2 (left). To produce the image at the centre, the algorithm was modified slightly. A single sample was calculated for cells that originally had two samples by averaging the samples. The result is similar to that produced by TL2 but is slightly darker. The image on the right illustrates the effect of not using virtual geometry cells; the interpolation is less smooth.

Modified Shepard's Method

Variant 1 (MS1) The best-quality trilinear interpolation variant (TL1) may return poor results and produce artefacts if there are too few usable values to interpolate from. An alternative method that increases the likelihood of more valid samples is interpolation by inverse distance weighting (IDW). In IDW, data samples in the vicinity of a point are weighted before they are combined. More weight is assigned to closer samples. IDW methods can be programmed to have a wider reach, making the use of more samples possible. We adapt one such method, modified Shepard's method (Franke and Nielson, 1980), in the same way as for trilinear interpolation, discarding unsuitable samples. Given N scattered data points $\boldsymbol{x}_k$, $k = 1, \dots, N$, and corresponding data values $f(\boldsymbol{x}_k)$, the general form of Shepard's method for obtaining an interpolated value $\tilde{f}(\boldsymbol{x})$ at $\boldsymbol{x}$ is

$$\tilde{f}(\boldsymbol{x}) = \frac{\sum_{k=1}^{N} w_k(\boldsymbol{x}) f(\boldsymbol{x}_k)}{\sum_{k=1}^{N} w_k(\boldsymbol{x})} \tag{5.12}$$

where $w_k(\boldsymbol{x})$ is the weighting function $w_k(\boldsymbol{x}) = d(\boldsymbol{x}, \boldsymbol{x}_k)^{-p}$, $p \in \mathbb{R}$, $p > 0$ and $d(\boldsymbol{x}, \boldsymbol{x}_k) = \|\boldsymbol{x} - \boldsymbol{x}_k\|$ is the Euclidean distance. In modified Shepard, the weighting function is changed to be

$$w_k(\boldsymbol{x}) = \left(\frac{\max(0, R - d(\boldsymbol{x}, \boldsymbol{x}_k))}{R d(\boldsymbol{x}, \boldsymbol{x}_k)} \right)^p \tag{5.13}$$

where only data points within a radius of influence R are considered. The usual value for p is 2, with larger values assigning more importance to nearer points and smaller values assigning more importance to points further away. When using MS1, a grid radius of 2 is used to search for valid samples and since the data is very sparse, a p value of 0.5. We set R equal to twice the length of a cell edge.

Table 5.1: Scene details.

Scene	Triangles	Grid Resolution	Real Cells	Virtual Cells	Two-Sample Cells
S_1	34	$32 \times 32 \times 32$	6,189	7,032	557
S_2	331,191	$50 \times 13 \times 32$	6,106	5,984	2,321
S_3	262,265	$50 \times 21 \times 31$	13,368	12,926	8,472
S_4	66,927	$50 \times 24 \times 23$	13,854	8,180	3,881
S_5	21,038	$55 \times 15 \times 100$	12,000	20,410	5,565
S_6	75,268	$40 \times 32 \times 17$	5,589	6,792	3,212
S_7	37,169	$44 \times 16 \times 50$	10,000	10,770	5,900
S_8	204,050	$38 \times 16 \times 64$	10,980	11,394	7,737

Variant 2 (MS2) In this variant the number of neighbour lookups is reduced by only considering the cells in approximately the same plane as the point in question. Moreover, if the number of valid samples reaches a certain configurable value, the neighbour search is terminated early. Due to this early termination strategy, a balanced sample set from all around the cell of interest is needed; this is achieved by ordering the neighbouring cells in a spiral pattern.

The modified Shepard variants are illustrated in Figure 5.10. MS2(25) and MS2(8) refer to a maximum of 25 or 8 valid samples respectively.

5.3 Evaluation

The method was tested over Wi-Fi on C_1, a smartphone (Snapdragon 835 SoC) and C_2, an Oculus Quest. The server was a Core i9-9900K machine with an RTX 2080 Ti GPU. The server was written in C++ and made use of the GPU's RT cores via OptiX 6.0. The clients were created with Unity using C# scripts; a native module written in C was used for network communication. The clients and the server were in close geographic proximity; however, for the smartphone tests, network communication was routed via an intermediate cloud-based machine located approximately 2,500 kilometres away. This resulted in an effective client-to-server distance of around 5,000 kilometres. The Oculus Quest tests were performed locally.

Information about the scenes used for the tests is presented in Table 5.1. The scenes are shown in Figure 5.11. Real geometry cells are cells that were found to contain geometry. Virtual cells are extra geometry cells that were created for the TL2 interpolation method. Only real geometry cells can contain two samples. Virtual geometry cells al-

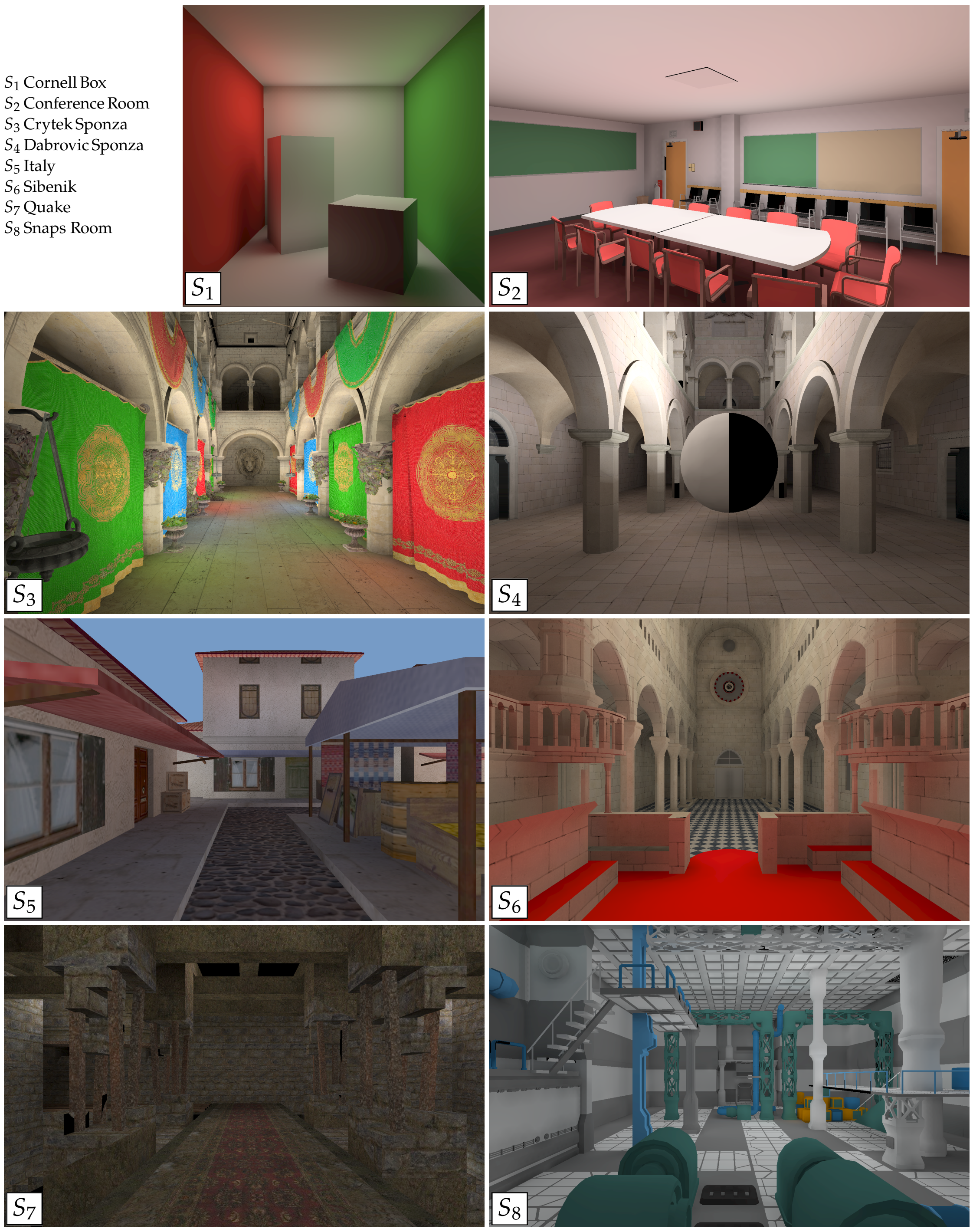


Figure 5.11: The scenes used for the evaluation, shown here with enhanced gamma. All scenes rendered using ReGGI and lit by indirect light only. Scenes S_1–S_4 and S_6 obtained from McGuire's Computer Graphics Archive (https://casual-effects.com/data). Scene S_5 is by Valve Corporation and is licensed under CC BY-NC-ND 4.0. Scene S_7 is from Quake 3 Arena. Scene S_8 was created using the Unity3D Snaps prototyping package.

Table 5.2: Client performance. Values marked with an asterisk were obtained at an eye texture resolution scale of 0.7.

Client	Scene	Direct (fps)	GI (fps)			
			MS1	MS2 25 / 8	TL1	TL2
C_1	S_1	60	26	34 / 57	54	60
C_1	S_2	60	12	21 / 30	31	36
C_1	S_3	60	8	23 / 43	42	54
C_1	S_4	60	11	28 / 52	49	60
C_1	S_5	60	11	30 / 53	49	59
C_1	S_6	60	9	25 / 42	39	53
C_1	S_7	60	8	27 / 45	39	56
C_1	S_8	60	10	23 / 40	40	47
C_2	S_1	72 / 72*	12	12 / 17	17 / 31*	30 / 60*
C_2	S_2	29 / 39*	3	6 / 9	10 / 13*	17 / 31*
C_2	S_3	37 / 61*	3	6 / 10	10 / 14*	20 / 44*
C_2	S_4	67 / 72*	3	7 / 10	12 / 16*	26 / 57*
C_2	S_5	57 / 72*	4	10 / 13	10 / 19*	23 / 50*
C_2	S_6	46 / 72*	3	6 / 9	10 / 14*	22 / 48*
C_2	S_7	52 / 72*	3	6 / 10	10 / 15*	24 / 53*
C_2	S_8	38 / 58*	3	5 / 9	10 / 12*	21 / 42*

ways contain exactly one sample. All the tests were performed at the native resolution of the clients i.e. Full HD (1920×1080) for C_1 and 1440×1600 *per eye* for C_2. In addition, for C_2, image reconstruction at a scale factor of 0.7 (the *XRSettings.eyeTextureResolutionScale* value in Unity) was also evaluated. The performance of the clients was evaluated by monitoring their frame rates when using the interpolation methods presented. Image quality was evaluated by comparing against ground truth images produced using instant radiosity. The effect of grid resolution on image quality was investigated. A large number of images generated from walkthroughs in several test scenes were also tested, and the results for the different interpolation algorithms compared. For the bandwidth tests, the server was configured to generate approximately 10 indirect illumination updates per second by specifying an appropriate number of VPL paths.

5.3.1 Image Reconstruction

Client performance for all the interpolation methods is summarised in Table 5.2. *Direct* indicates the frame rate when computing direct light only. *GI* indicates the frame rate for the full global illumination solution, where image reconstruction and merging is included. MS2 was configured to use a maximum of either 25 or 8 valid samples. The irradiance texture was sampled at 2:1 for these tests. The resolution at which irradiance is interpolated affects performance significantly. For example, for S_3 on C_1

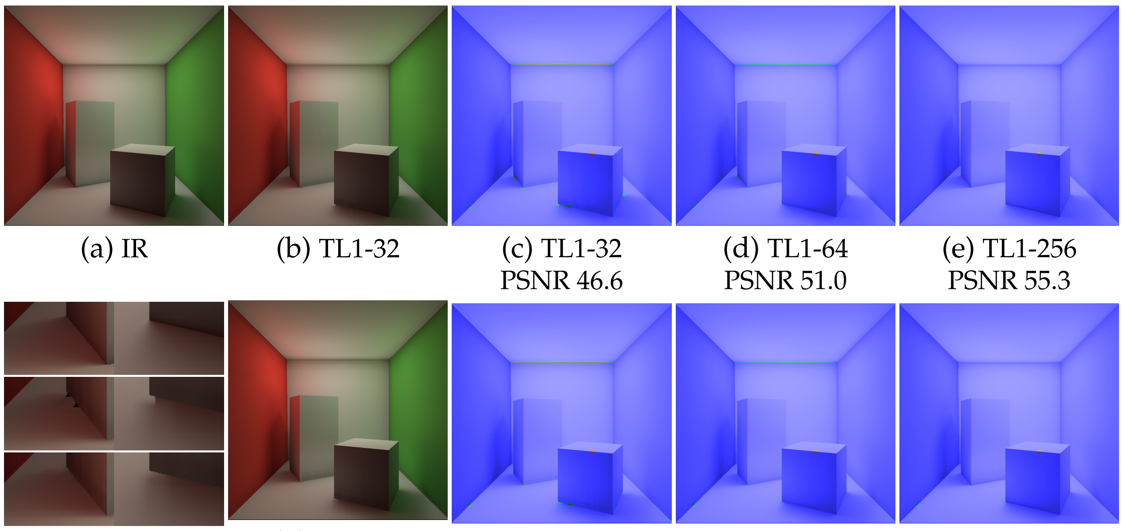
Table 5.3: Effect of grid resolution on image quality (PSNR for instant radiosity vs ReGGI, indirect illumination only).

S_1 **(Cornell Box)**			S_3 **(Crytek Sponza)**			S_6 **(Sibenik)**		
MaxDim	**TL1**	**MS1**	***MaxDim***	**TL1**	**MS1**	***MaxDim***	**TL1**	**MS1**
8	33.3	33.0	25	22.0	24.3	40	22.4	24.3
16	40.7	39.8	50	24.6	25.7	50	22.2	24.4
32	46.6	45.9	100	24.9	25.9	100	24.4	25.7
64	51.0	50.3	200	27.7	28.0	200	25.7	26.8
256	55.3	55.2	400	29.1	29.2	400	27.2	27.5

(the smartphone), the performance of TL1 goes down from 42 fps to 14 fps when the irradiance texture is sampled at 1:1 and up to 60 fps at 3:1. TL1 always performs better than MS1 due to a smaller number of nearest-neighbour lookups. In fact, the performance of TL1 is very close to that of MS2(8). It is always the case that performance improves as the number of required sample lookups decreases. For the modified Shepard methods, MS2(8) performs best, then MS2(25), then MS1. Similarly, TL2 performs better than TL1. Although both clients use the same Snapdragon 835 hardware, C_2 (the Oculus Quest) performed much worse than C_1. One of the reasons for the reduced performance is the higher resolution on the Oculus. This fact is highlighted by the values marked with asterisks in the table; these are the results obtained when the eye texture resolution scale was set to 0.7. In our implementation, we used deferred rendering, real-time shadows, multipass VR rendering, and applied GI as a postprocessing effect. All these techniques negatively impact performance.

5.3.2 Image Quality

All the image quality tests were performed on indirect illumination. When sampling irradiance at 2:1 or even at 3:1, the Peak Signal-to-Noise Ratio (PSNR) values varied by less than 0.1. Sampling irradiance at a lower resolution and then upsampling it to the native resolution during image reconstruction somewhat smoothens the result. In several cases, when using TL1, this had a beneficial effect on the PSNR values, increasing them marginally. For the results in Table 5.3 and Table 5.4, the viewpoints in Figure 5.11 were all rendered at Full HD except for S_1 where a resolution of 1024×1024 was used. Table 5.3 shows the effect of grid resolution on image quality with respect to instant radiosity as the ground truth. Image quality improves rather slowly for the larger scenes and the grid resolution required to achieve PSNR values in the high twenties or more is too large to be practical for our method. High grid resolutions usually equate to a large number of



(a) IR (b) TL1-32 (c) TL1-32 PSNR 46.6 (d) TL1-64 PSNR 51.0 (e) TL1-256 PSNR 55.3

(f) Detail (g) MS1-32 (h) MS1-32 PSNR 45.9 (i) MS1-64 PSNR 50.3 (j) MS1-256 PSNR 55.2

Figure 5.12: Image quality comparison against instant radiosity as the ground truth, indirect illumination only; difference images produced with HDR-VDP-2.2.2. TL1-n and MS1-n refer to the *MaxDim* value for the grid resolution used. In (f) the images at the top are details from (a); the middle and bottom images are details from (b) and (g) respectively. Although TL1 is closer to the ground truth than MS1, it produces artefacts at the base of the tall box.

irradiance samples, which negatively affects a number of stages in our pipeline, including server computation, bandwidth requirements and the time to process the incoming data and update the world grid on the client. Table 5.4 compares the various interpolation methods against instant radiosity as the ground truth. In S_7 although MS2(8) obtained

Table 5.4: Image quality by interpolation method (PSNR for instant radiosity vs ReGGI, indirect illumination only). TL2* refers to a TL2 variant where an average of two samples is used instead of the sample with the larger magnitude.

Scene	***MaxDim***	**TL1**	**TL2**	**TL2***	**MS1**	**MS2(25)**	**MS2(8)**
S_1	32	46.6	23.8	24.5	45.9	45.8	37.5
S_2	50	26.2	24.2	23.0	30.5	29.3	28.2
S_3	50	24.6	25.3	22.8	25.7	25.2	23.3
S_4	50	23.3	20.2	20.6	23.9	23.2	21.5
S_5	100	18.7	20.3	19.6	20.0	20.5	20.7
S_6	40	22.4	21.3	22.6	24.3	23.7	23.3
S_7	50	29.9	24.9	27.3	31.9	30.3	29.9
S_8	64	21.9	21.1	19.5	24.8	23.1	22.7

Table 5.5: Image quality (instant radiosity vs ReGGI, indirect illumination only). PSNR and MSSIM values are averages over paths of 256 keyframes.

Scene	PSNR	MSSIM
S_2	32.8	0.93
S_3	33.6	0.94
S_4	28.2	0.93
S_5	28.9	0.94
S_6	29.0	0.92

a PSNR value equal to TL1, artefacts were present. Figure 5.12 highlights the differences between TL1 and MS1 in S_1. Although both TL1 and MS1 obtain very good PSNR values, with TL1 scoring better than MS1, TL1 produces artefacts when there are not enough samples for effective interpolation.

For a thorough comparison of ReGGI against instant radiosity as the ground truth, numerous images were generated with both techniques and the PSNR and Mean Structural Similarity (MSSIM) metrics calculated. The ReGGI images were produced at the grid resolutions specified in Table 5.1 using MS1 with the exponent value set to 0.5. The images were rendered at key points along paths throughout several scenes. At every one of these points, the two techniques produced Full HD indirect light images using the number of VPL paths specified in the top part of Table 5.8. The PSNR and MSSIM values were computed for each pair of images and recorded. Finally, the PSNR and MSSIM values for each set of images were averaged. The results are summarised in Table 5.5.

5.3.3 Amortisation

Table 5.6 illustrates how the solution scales and the potential for amortising costs when several clients share a virtual environment. The gather time includes VPL generation and irradiance gathering; the send time includes packet construction, compression and the system call that pushes the packet onto the network. Headless clients were used for these tests. These are simulated clients that do not perform any processing and do not produce any output. They connect to the server, perform the initial exchange and then act like sinks, discarding all the data received. The client connections were staggered, a new client appearing every 20 seconds; the tabulated values are the average gather and send times over each 20-second period. Static scenes were used to ensure that the scene state (lights and geometry) was always the same at any time. However, the server was configured to always send the entire scene data to all the clients instead of just the deltas. This simulated a worst case scenario for dynamic scenes. The same number of VPL

Table 5.6: Amortisation (gather / send times in milliseconds, 10 updates per second for each scene).

Scene	VPL Paths	Clients			
		1	2	4	8
S_1	5300	92 / 2	92 / 5	95 / 2	94 / 4
S_2	1900	92 / 5	94 / 3	95 / 3	95 / 5
S_3	850	92 / 9	89 / 13	90 / 13	92 / 16
S_4	1850	87 / 6	91 / 6	88 / 9	94 / 8
S_5	6000	94 / 16	98 / 11	100 / 11	98 / 14
S_6	2100	87 / 6	89 / 6	89 / 5	92 / 6
S_7	15700	92 / 9	94 / 9	95 / 8	94 / 11
S_8	1300	92 / 8	95 / 10	92 / 12	92 / 14

paths as for the bandwidth tests were used so that the server produced approximately 10 indirect illumination updates per second. Since every update was computed only once and the results reused for all the clients, the gather time stayed approximately constant as more clients connected. In general the send time increased marginally with the number of clients.

5.3.4 Data Transfer

Table 5.7 shows the minimum, maximum and average compression ratios obtained for the illumination data sent by the server. Although each packet is compressed individually, zlib (DEFLATE) at the best speed compression level delivers quite good compression. Dynamic lights or objects were used during these tests, simulating continuously changing conditions, preventing illumination convergence.

Table 5.8 indicates the average bandwidth requirements for the test scenes when

Table 5.7: Compression ratios.

Scene	Min	Max	Avg
S_1	1.56	2.15	1.90
S_2	1.34	2.74	1.73
S_3	1.07	2.22	1.48
S_4	1.41	3.53	1.81
S_5	1.40	5.54	1.80
S_6	1.26	2.74	1.80
S_7	1.40	1.93	1.68
S_8	1.28	2.61	1.77

Table 5.8: Bandwidth. The world grid resolution for S_8^* is 29×13×50.

Scene	VPL Paths	Bandwidth (Real Cells Only) (Mbps)	Bandwidth (Real + Virt. Cells) (Mbps)
S_1	5300	4.88	8.34
S_2	1900	4.90	7.18
S_3	850	5.26	7.57
S_4	1850	6.79	9.95
S_5	6000	8.85	12.83
S_6	2100	5.46	8.05
S_7	15700	9.93	13.01
S_8	1300	11.20	15.85
S_8	3800	5.68	7.66
S_8^*	1300	7.14	10.70
S_8^*	3800	3.59	5.25

using only the actual geometry cells (MS1, MS2 and TL1) and when virtual geometry cells are used too (TL2). The server was configured to send approximately 10 indirect illumination updates per second by specifying an appropriate number of VPL paths. The measurements were obtained by averaging the amount of data sent by the server over a 200-frame period (frames 101 through 300). When virtual geometry cells are used, more data needs to be sent by the server for each update, resulting in considerably more bandwidth requirements. Moreover, due to the larger number of cells, more processing was required on the server. This resulted in the frequency of the updates dropping by one, to 9 Hz for all the scenes except S_5, where the update frequency dropped to 8 Hz. In static scenes, as the global illumination solution converges, fewer cells require updates. The size of the updates drops with time and eventually no more updates are required. In the bandwidth tests, a continuously moving light source was used in all the scenes, ensuring a steady stream of updates.

The system is malleable and can be tailored for specific bandwidth requirements. A lower grid resolution equates to fewer cells, fewer irradiance samples and less data to transfer, resulting in lower bandwidth at the cost of image quality. Increasing the number of VPL paths reduces bandwidth and simultaneously improves image quality because updates are sent at a lower rate (and hence less data is transferred) and the samples are of higher quality. This increases output lag and is not suitable for dynamic scenes that contain fast-moving lights or objects. These scenes require more frequent, lower-quality updates at an increased bandwidth cost. Using S_8 as a test case, increasing the number of VPL paths to 3800 results in an update frequency of approximately 5 Hz and cuts bandwidth cost by half to 5.68 Mbps. S_8^* in the bottom two rows of Table 5.8 indicates that a world grid resolution of $29 \times 13 \times 50$ is used. At this grid resolution, with approximately half the number of cells, the bandwidth cost drops to 7.14 Mbps. Using both strategies together (less frequent but higher quality updates together with a lower grid resolution) results in a bandwidth cost of 3.59 Mbps.

5.4 Discussion

The client computes direct illumination itself and also handles user input. Moreover, indirect illumination is introduced into the client asynchronously. Due to these features, input lag is completely eliminated. The image quality results show that by sparse illumination sampling with grid resolutions that are neither too coarse nor too fine, global illumination effects can be reproduced quite effectively. Bandwidth results are reasonable but can be reduced by keeping the server updated with the client camera's position and view direction. In this way the server could transmit data related only to the region of interest instead of the entire scene. Image reconstruction costs were too high for the weaker device and image quality had to be sacrificed to achieve high frame rates. This indicates that lower-cost interpolation algorithms are needed. The excellent server scaling results indicate that there is great potential for amortising costs between clients.

High grid resolutions are not practical because they negatively affect server computation and bandwidth. Especially for larger scenes, an adaptive spatial data structure would be a better choice than a full grid. In this way detailed geometry and the corresponding illumination data could be stored at a higher resolution than for other parts of the scene. However, any alternative spatial data structure chosen would need to support efficient nearest-neighbour queries so as not to increase reconstruction cost on the client.

5.5 Summary

The method presented in this chapter is a viable solution for providing dynamic diffuse indirect lighting to weak devices such as smartphones. The client eliminates input lag by computing direct illumination and responding to user input locally. It reconstructs indirect illumination by interpolating sparse irradiance samples received from a remote server. Combining the direct and indirect illumination components yields the global illumination solution. The method has low bandwidth requirements and its high configurability enables the balancing of characteristics such as image quality and reconstruction cost. The server produces irradiance samples using a method based on instant radiosity, and stores them on a regular grid spanning the entire scene. The server exhibits good scaling with multiple clients, making the method well-suited for computing indirect illumination in a shared virtual environment.

6 Radiance Megatextures

The distributed rendering pipeline described in this chapter provides weak devices with global illumination at low cost, with no constraints on the types of light sources and materials used. The method, Device-Agnostic Radiance Megatextures (DARM), computes all illumination remotely, on the server. The illumination data is stored in a large texture atlas called a megatexture, the relevant parts of which are streamed to the client as textures using video compression. The client is aware of the scene geometry and handles user input locally. This completely eliminates input lag and improves the responsiveness of the client device. The reconstruction cost on the client is minimal as only texture sampling using rasterisation is required.

A variant of the DARM method, DARM Virtual Atlas (DARM-V), was integrated into Unity, a popular game engine. DARM-V uses the same machinations as DARM but the texture atlas is virtual, that is, it is not backed up by any form of storage. The texture that is streamed to the client is created directly from the framebuffer. Moreover, whereas in DARM the streamed texture is composed of fixed-size tiles, in DARM-V the streamed texture can contain tiles of different size. This level-of-detail mechanism allows fine partitioning of the virtual texture atlas into logical tiles, while allowing tiles to have a larger footprint in the streamed texture.

The chapter is structured as follows. Section 6.1 contains background information on virtual texturing, parametrisation and packing strategies. DARM is detailed in Section 6.2 and evaluated in Section 6.3. Section 6.4 describes DARM-V. Section 6.6 summarises the chapter, discussing the method and its limitations and suggests ideas for future work.

6.1 Introduction

Sparse Virtual Textures

Vast, extremely detailed, non-repeating textures bring a high level of realism to applications such as video games, flight simulators, and virtual reality. However, the size of video memory (VRAM) limits the use of a large number of high quality assets. Operating systems use virtual memory to solve a similar problem. Memory is paged in or out, from or to disk, as needed. Page faults stall processes until the needed portion of memory is paged in. Using the same stalling mechanism in an interactive graphics application is unacceptable as it would immediately destroy the user's experience. Sparse virtual textures, or *megatextures*, to use the term coined by their inventor John Carmack, avoid stalling by using a lower quality variant of the texture until the high quality version is available.

Megatextures were first used in the video game Quake Wars, which was released in 2007. Barrett (2008) provided insight and a reference implementation of the mechanism used (Figure 6.1). The megatexture, which may be extremely large (for example, 256K by 256K pixels), and its mipmaps, are partitioned into square tiles. Tiles that are currently visible, and optionally tiles that may become visible shortly, are placed into a smaller texture called the physical texture. A second texture called the pagetable texture is also constructed at the same time. The pagetable texture has the same dimensions as the megatexture but in tiles rather than in pixels. The pagetable texture contains mappings

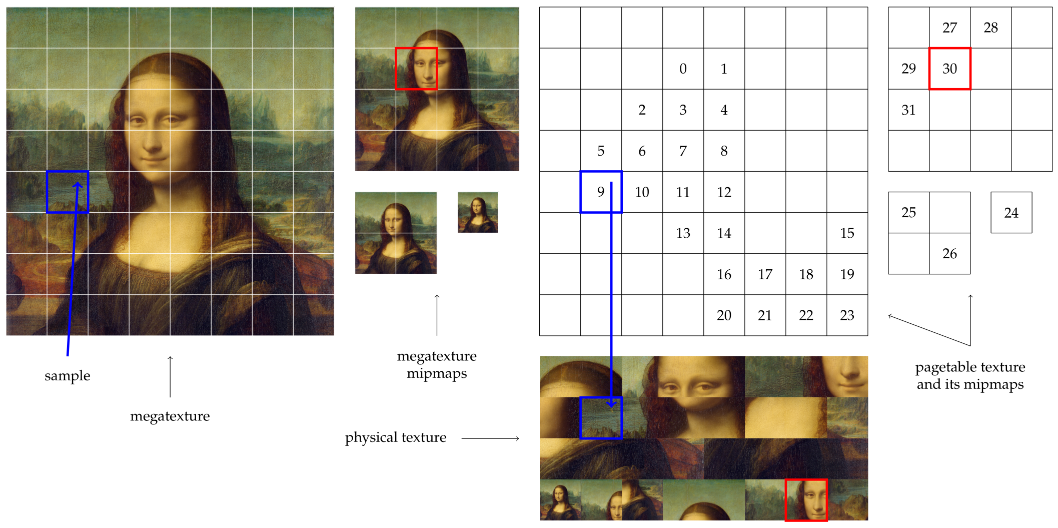


Figure 6.1: Sparse virtual texturing. Adapted from Barrett (2008).

from megatexture coordinates to physical texture tiles, and with some calculations the correct pixel within the physical texture tile can also be identified. Using the mechanism is simple but requires two texture lookups instead of one, first into the pagetable texture and then into the physical texture.

Mittring (2008) provided practical megatexture implementation details from his experience with Crytek's implementation in CryEngine. An important subtopic is related to the management of the physical texture, which is also referred to as the tile cache. Since the space in the tile cache is limited, a strategy for replacing and evacuating tiles is needed. The currently cached tiles and their level of detail are tracked with a quadtree and a least-recently-used (LRU) cache strategy is used. Van Waveren (2009) details the challenges encountered when trying to parallelise the virtual texture implementation in the video game Rage to obtain a 60 Hz frame rate. They opted to refactor large portions of their game engine into many small jobs that have well-defined inputs and outputs and are stateless (independent from one another). Each job is designed to perform a task that can be completed in the time that it takes to render a single frame (16 ms). Van Waveren (2012) also discusses the implementation of an efficient virtual texturing system implemented entirely in software. Schwartz et al. (2013) use sparse virtual textures to stream bidirectional texture functions (large and highly detailed material representations) to the GPU while keeping the memory footprint small.

Parametrisation and Packing

As with standard texturing, virtual texturing requires parametrising (unwrapping) 3D models and packing the 2D results efficiently into a texture atlas. The procedure often involves decomposing the 3D model into a set of charts that are then parametrised separately. This may then give rise to visible seams at the chart borders. By necessity, the 2D representation is a deformed version of the original 3D model. Parametrisation algorithms strive to avoid excessive deformations as these may cause visual artefacts when the texture is sampled. To reduce these artefacts, algorithms try to find a balance between preserving as much as possible the angles of the triangles making up the model while also respecting the relative sizes of the triangles. A common packing strategy is to construct a bounding rectangle around each chart, and place it within the rectangular confines of the atlas using a heuristic. Algorithms are forced to use approximative packing because optimal rectangle packing is known to be NP-hard (Korf et al., 2010). xatlas[1] uses either random placement or brute force placement. stb_rect_pack[2] uses the

[1] `https://github.com/jpcy/xatlas`
[2] `https://github.com/nothings/stb/blob/master/stb_rect_pack.h`

Skyline Bottom-Left algorithm (Jylänki, 2010).

Lévy et al. (2002) presented an automatic texture atlas generation method that minimises angle deformations and stretching, and avoids triangle flipping. Their method creates larger and hence fewer charts, reducing artefacts due to discontinuities. The charts can have complex borders; their Tetris-inspired packing algorithm supports this kind of chart and stores the charts more efficiently than other methods that use strategies based on bounding rectangles. Ray et al. (2010) use a method based on grid-preserving parametrisations together with a postprocessing step to generate seamless texture atlases.

DARM

When using a megatexture, a primary consideration is the amount of detail to store within it. If a single megatexture texel is taken to represent an area of 1m^2, a large 3D space can be represented by a small megatexture that easily fits in memory, but it would not be possible to see any small objects at all. At the other extreme, if a single texel is taken to represent an area of 1mm^2, zooming in on any location within the scene would reveal a great amount of detail (assuming correspondingly detailed textures) but the storage requirements for such a megatexture may be too large to fit on disk. It is therefore important to strike a balance between amount of detail and megatexture size. The measure used in this thesis to determine the amount of detail is *pixels per world unit* (ppwu). For the test scenes, the ppwu values used were between 32 and 128.

DARM, the method proposed in this chapter, eliminates input lag and produces good image quality but suffers from exposure artefacts while the camera is moving. This is mitigated by a coarse megatexture that is maintained on the client. The method has low bandwidth requirements and achieves high frame rates (between 36 and 50 Hz at Full HD) on the mid-range smartphone used in the tests. The contributions of the method are:

- A novel distributed rendering pipeline for high-fidelity graphics based on radiance megatextures.
- A network-based out-of-core algorithm that circumvents VRAM limitations without sacrificing texture variety.
- Automatic precomputation for texture atlas generation.
- A client-side coarse cache that mitigates artefacts due to missing data and makes the system robust to network fluctuations.

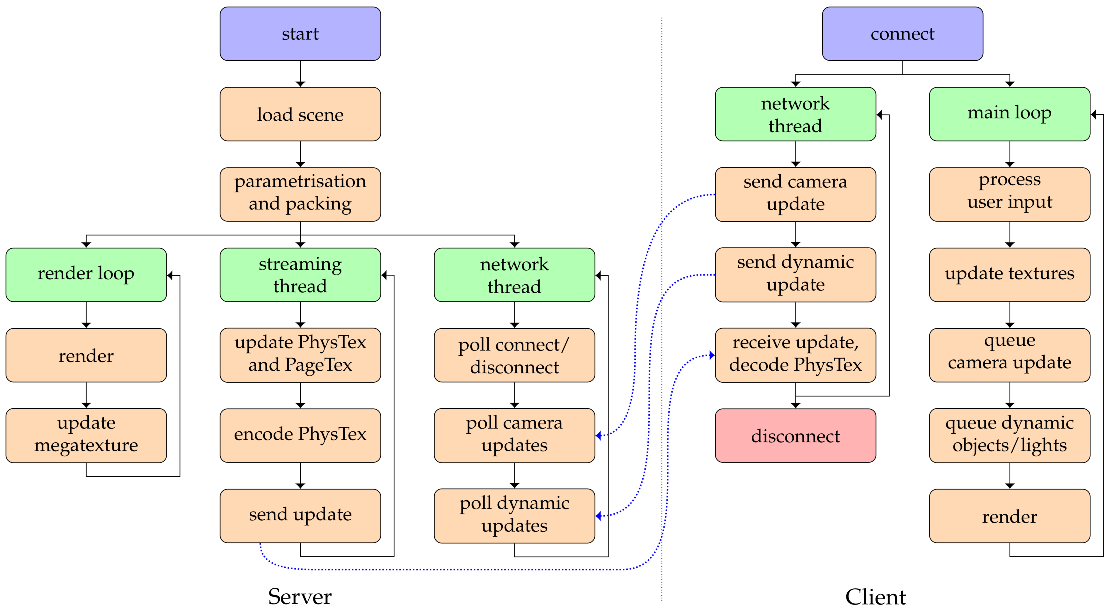


Figure 6.2: DARM architecture.

6.2 Method

Figure 6.2 illustrates the processes executing on the server and the client and the data flow between them. The server loads the scene, parametrises and packs it into a megatexture, and splits processing into the render loop which executes on the main thread, a streaming thread and a network thread. The megatexture is progressively updated after every rendered frame. The streaming thread continuously creates a physical texture and a pagetable texture from the megatexture and streams them to a connected client. The network thread manages client connections and receives camera updates and notifications about dynamic events occurring on the client (dynamic objects and lights). When a client connects, it receives meta information about the megatexture such as its dimensions in tiles, the camera's initial position and orientation, video decoder parameters, and scene information. The latter consists of a set of triangle primitives (vertices and texture coordinates into the megatexture) together with meta information associating objects with the primitives and identifying whether objects are static or dynamic. The client also receives a list of dynamic lights, for the express purpose of user manipulation.

Figure 6.3 illustrates part of the processing that occurs on the server. Given a scene description, such as the one for the scene shown on the left, the server parametrises the scene into a texture atlas (centre), a process that projects all the surfaces in the scene onto a plane without any overlapping parts. The server then renders the scene, shading

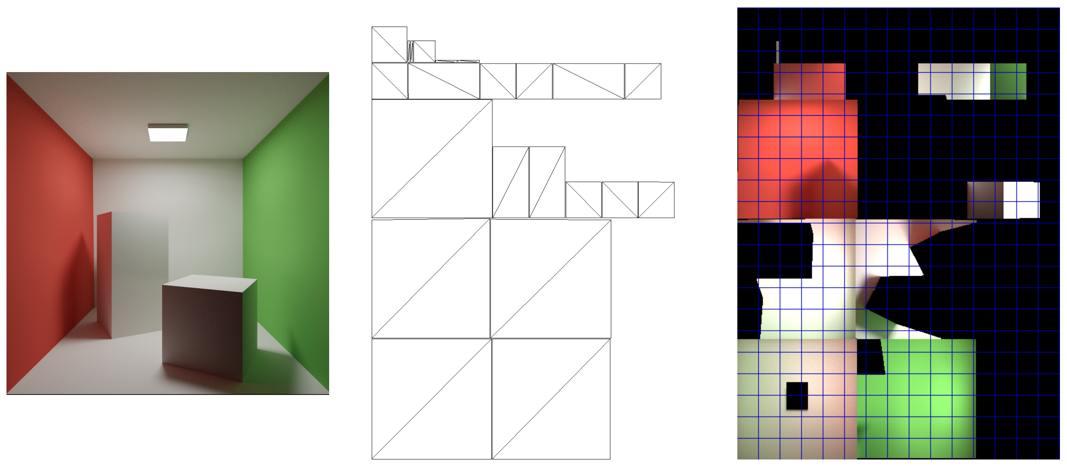


Figure 6.3: Left: The Cornell Box scene. Centre: Parametrised scene. Right: Shaded atlas.

the texture atlas in the process (right). The texture atlas or megatexture contains the entire scene and can be extremely large. In the figure, the megatexture is shown with a superimposed tiling arrangement. This visualisation is used to indicate that tiles are selected from the megatexture and used to construct a smaller *physical texture* which is transmitted to the client.

6.2.1 Parametrisation

Since large scenes can produce a texture atlas with high storage requirements, an algorithm that produces a parametrisation with little wasted space between triangle primitives benefits the subsequent packing process. Ideally, many triangles that are edge-connected in 3D would remain connected after parametrisation. Therefore, instead of creating a triangle soup from all the scene's objects, and feeding the entire set of triangles to the algorithm, we parametrise object by object. We use Least Squares Conformal Maps (LSCM) (Lévy et al., 2002) for parametrisation, either through the Computational

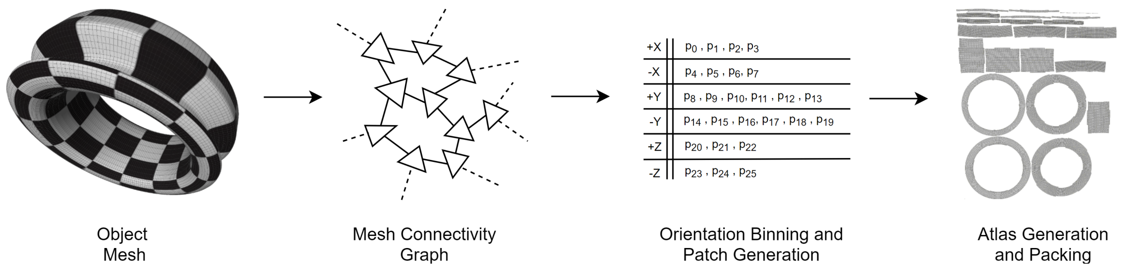

+X p_0, p_1, p_2, p_3
-X p_4, p_5, p_6, p_7
+Y $p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}$
-Y $p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{19}$
+Z p_{20}, p_{21}, p_{22}
-Z p_{23}, p_{24}, p_{25}
Object Mesh
Mesh Connectivity Graph
Orientation Binning and Patch Generation
Atlas Generation and Packing

Figure 6.4: Parametrisation process from object space to texture space.

Geometry Algorithms Library (CGAL) (Fabri et al., 2000) or through xatlas (Young, 2020).

When using LSCM through CGAL some preprocessing is required, due to robustness issues. Each object is broken down into patches of adjacent (connected) triangles, and each patch is parametrised separately. This process, together with the final packed arrangement, is illustrated in Figure 6.4. An edge connectivity graph for each shape is constructed. Each node in the graph corresponds to a triangle and the connections identify common edges between triangles. The triangles are sorted into six bins, according to the maximal component of the face normal along the positive and negative x-, y- and z-axes. If two connected triangles are placed in different bins, the connection is broken. At the end of this process the bins contain a number of patches which are then fed into the algorithm and parametrised. When using LSCM through xatlas the procedure is simpler. We just pass in each object separately, as an independent mesh consisting of the 3D vertices of the triangle primitive making up the object.

6.2.2 Packing

The LSCM-CGAL parametrisation procedure converts each 3D patch supplied as an input into a set of 2D patches called a chart. Using CGAL we generate an oriented bounding box for each chart, determine its principal axis and rotate it as necessary so that it is aligned with the x- or y-axis, whichever is closer. The bounding box is translated so that the minimum x and y coordinates are located at the origin. Finally, since the parametrisation may have used a different scaling for different 3D patches, the chart is rescaled so that the total area of the triangles making it up is equal to the total area of the original unparametrised triangles. This ensures a consistent relative scale between all charts. Before the charts are packed into an atlas (using their rectangular bounding boxes), the dimensions of the atlas in world units are calculated. The width of the atlas is estimated as follows. For perfect packing, with no wasted space at all, the area of the atlas would be equal to the total area of all the charts. Assuming a square-shaped atlas, we calculate its width as the square root of the area. For packing, we use a simple bottom-left strategy. The charts are sorted by decreasing bounding box height and processed in that order, virtually placing them side by side in the atlas. We calculate how many charts fit in a "row", then move up, start a new row, and repeat. In this way a value for the height of the atlas is obtained.

The resolution of the atlas in pixels per world unit is determined by a configuration parameter q. Multiplying the dimensions of the atlas by q we obtain the dimensions of the atlas in pixels. At a later stage, the atlas will be partitioned logically into square

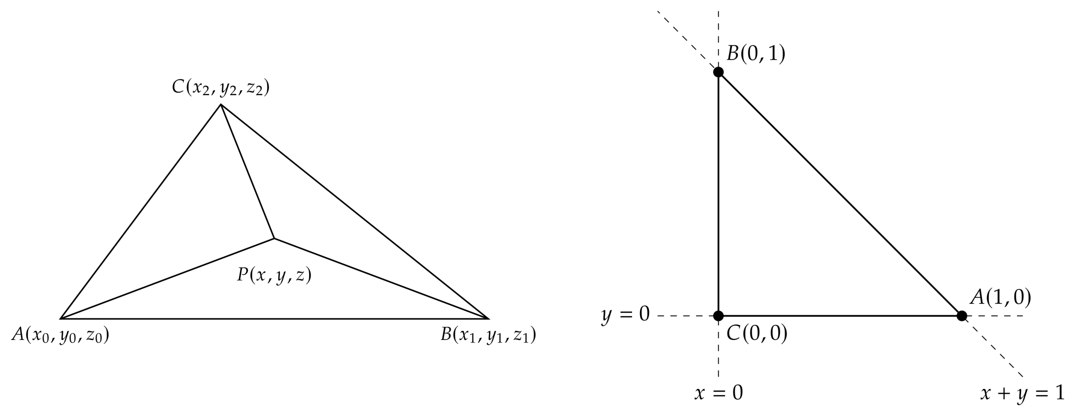


Figure 6.5: Left: Barycentric coordinates. Right: The reference triangle.

tiles; another configuration parameter determines the width of a tile in pixels. The pixel dimensions of the atlas are rounded up to the nearest tile. The dimensions of the atlas are also rounded up to be square in shape to match the shape of texture space, and to the nearest power of two. This is needed for correct sampling from the atlas. The charts are now packed into the atlas using the bottom-left strategy described above. The atlas coordinates for the vertices of all parametrised triangles are computed at this stage and normalised to [0, 1]. This produces the texture coordinates for all the triangles. As in Section 6.2.1, the packing procedure for the LSCM-xatlas combination is less involved. xatlas accepts a "texels per unit" parameter that is equivalent to our pixels per world unit, and performs packing automatically. We only need to round up the atlas's dimensions and normalise the parametrised coordinates. In our tests, the packing strategy used by xatlas typically yields better (higher) atlas occupancy, hence we use LSCM-xatlas as our default parametrisation and packing method.

Finally, storage for the atlas is allocated. A memory-mapped file is used since the atlas may not fit in memory. At this point there are two representations for each triangle in the scene, the original triangle and the version residing in the atlas. An affine mapping between the original triangle's barycentric coordinates and the texture coordinates within the atlas is constructed (Section 6.2.3). This mapping is needed during the rendering pass to populate the atlas with shading information.

6.2.3 Triangle Mapping

Any point of a 2D or 3D triangle, including the vertices, the edges, and all the internal points can be uniquely identified using barycentric coordinates $\alpha, \beta, \gamma \in [0,1], \alpha + \beta + \gamma = 1$. Due to the relationship between these coordinates, any two of them suffice to obtain the third. For example, given β and γ, the third coordinate is readily computed as $\alpha = 1 - \beta - \gamma$. One way of computing the barycentric coordinates of an arbitrary point $P(x, y, z)$ of a triangle ABC (Figure 6.5, left) is by computing ratios of triangle areas:

$$\alpha = \frac{\text{Area of } \triangle PBC}{\text{Area of } \triangle ABC} \qquad \beta = \frac{\text{Area of } \triangle PCA}{\text{Area of } \triangle ABC} \qquad \gamma = \frac{\text{Area of } \triangle PAB}{\text{Area of } \triangle ABC}. \tag{6.1}$$

P's Cartesian coordinates can be retrieved from its barycentric coordinates using:

$$x = \alpha x_1 + \beta x_2 + \gamma x_3 \qquad y = \alpha y_1 + \beta y_2 + \gamma y_3 \qquad z = \alpha z_1 + \beta z_2 + \gamma z_3. \tag{6.2}$$

The vertices of $\triangle ABC$ can be obtained by setting two barycentric coordinates to zero (the third is by necessity one). In the example shown, the barycentric coordinates of vertices A, B, and C are $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ respectively.

Consider the two-dimensional triangle R whose vertices have the Cartesian coordinates $(1, 0)$, $(0, 1)$, and $(0, 0)$ (Figure 6.5, right). $\triangle R$ can be defined as the three half planes

$$x \geq 0, \tag{6.3}$$

$$y \geq 0, \tag{6.4}$$

$$x + y \leq 1. \tag{6.5}$$

From Inequality 6.5, $x \leq 1 - y$, and since the maximum value of the right-hand side occurs when y is zero, we get $x \leq 1$. By similar reasoning we also obtain $y \leq 1$. By setting $z = 1 - x - y$, Inequality 6.5 can be written as $z \geq 0$. Since the maximum value of $1 - x - y$ is 1, occurring when both x and y are zero, we also have $z \leq 1$. Combining all these constraints, we have $x, y, z \in [0, 1]$ and $x + y + z = 1$ which are equivalent to the constraints for the barycentric coordinates α, β, γ described earlier. This establishes an equivalence between the barycentric coordinates of any triangle and the Cartesian coordinates of $\triangle R$. We will call $\triangle R$ the reference triangle.

An affine mapping ϕ can be constructed between a general point $\boldsymbol{r}(r, s)$ of $\triangle R$ and a general point $\boldsymbol{t}(x, y)$ of an arbitrary two-dimensional triangle T, with the vertices A, B, and C of $\triangle R$ mapping to the vertices $A(x_0, y_0)$, $B(x_1, y_1)$, and $C(x_2, y_2)$ of $\triangle T$ respectively. The affine mapping takes the form

$$\phi(\boldsymbol{r}) = \boldsymbol{M} \cdot \begin{pmatrix} r \\ s \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \tag{6.6}$$

where

$$\boldsymbol{M} = \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix}. \tag{6.7}$$

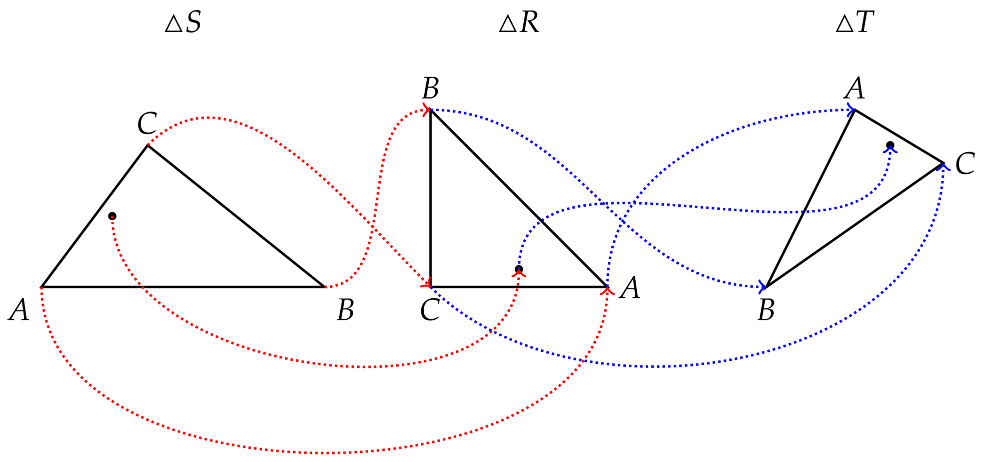


Figure 6.6: Mapping $\triangle S$ to $\triangle T$ via the reference triangle R.

Combining the equivalence between the barycentric coordinates of an arbitrary triangle S and the reference triangle R, and the mapping from $\triangle R$ to another arbitrary two-dimensional triangle T, Equation 6.6 can be written in terms of the barycentric coordinates of $\triangle S$ as

$$\phi(\alpha,\beta) = \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix} \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}. \tag{6.8}$$

This construction, visualised in Figure 6.6, is effectively a mapping from $\triangle S$ to $\triangle T$. We can quickly verify that the vertices of $\triangle S$ map to the correct vertices of $\triangle T$, but the mapping works for any point in $\triangle S$:

$$\phi(1,0) = \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_0 - x_2 + x_2 \\ y_0 - y_2 + y_2 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \tag{6.9}$$

$$\phi(0,1) = \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 - x_2 + x_2 \\ y_1 - y_2 + y_2 \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \tag{6.10}$$

$$\phi(0,0) = \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 + x_2 \\ 0 + y_2 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}. \tag{6.11}$$

6.2.4 Shading

Shading of the atlas can be performed by any rendering method. Rasterisation, recursive ray tracing and path tracing are just a few possible options. For our experiments we used ray-based techniques. The scene is rendered as normal, computing the radiance for each pixel in the frame buffer. Some meta information is retained for each pixel, specifically the index of the related triangle, a flag describing the triangle's material (specular or not) and the barycentric coordinates of the hit point. With this information, the mapping mentioned in Section 6.2.2 can be used to locate the corresponding pixel in the atlas. As in ReGGI, radiance values for non-specular materials are updated progressively using Welford's method (Welford, 1962). Specular materials are updated progressively too but when the viewpoint changes the updates are reset. Changes in the scene (objects or lights) invalidate the entire atlas.

6.2.5 The Physical and Pagetable Textures

The currently visible tiles are determined from the atlas pixel coordinates obtained in Section 6.2.4. A small *physical texture* which we shall refer to as *PhysTex* is constructed from these tiles. A second texture, the *pagetable texture* or *PageTex* is created to map texture coordinates within the megatexture to texture coordinates within *PhysTex*. The dimensions of *PageTex* are equal to the dimensions of the megatexture in tiles rather than in pixels. For example, if the megatexture is made up of 256^2 tiles, *PageTex* is 256^2 pixels, where each pixel contains two floating-point values (*PageTex* is not a texture in the usual sense; it does not contain image data). *PhysTex* and *PageTex* are both communicated to the client and it is important that they are synchronised at all times, otherwise the wrong tile may be retrieved from *PhysTex*.

When tiles are placed into *PhysTex*, we attempt to preserve temporal coherence as much as possible to enable better compression by the stream encoder. With this in mind, tiles are not relocated unless absolutely necessary. If tiles that were already in *PhysTex* are still visible, their position is retained. On occasion there may not be enough space within *PhysTex* to store all the currently visible tiles. When this happens, tiles that are no longer visible (if any) are removed. The tiles are evicted with a least recently used policy. When there are too many visible tiles to fit into *PhysTex*, some tiles do not make it into the texture and are not communicated to the client, resulting in visible artefacts. A low quality megatexture called the coarse texture or *CoarseTex* is used on the client to mitigate this issue.

6.2.6 Client Rendering

The client updates its coarse version of the server's megatexture, *CoarseTex*, whenever it receives a *PhysTex* update. It computes a low quality representation of each tile present in *PhysTex*, and stores the tile at its proper place in *CoarseTex*. Rendering is performed using rasterisation techniques. A simple vertex shader transforms vertices with the usual

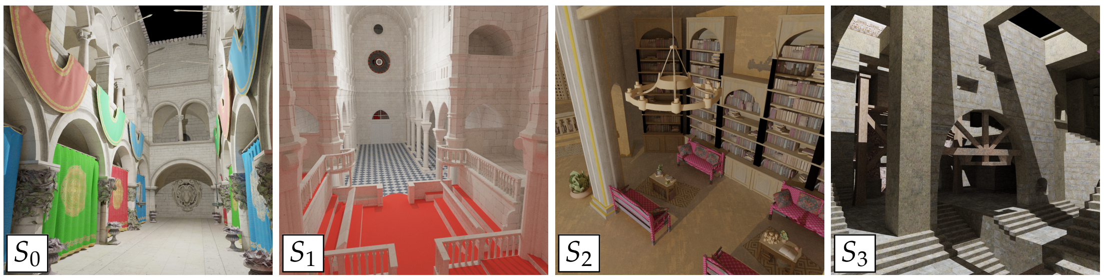


Figure 6.7: The scenes used for the evaluation.

model, view and projection matrices and forwards texture coordinates to the fragment shader. The fragment shader makes use of *PhysTex*, *PageTex* and *CoarseTex*. Since the texture coordinates on the client are relative to the megatexture, two texture-sampling operations are needed. *PageTex* is sampled to obtain texture coordinates into *PhysTex*. Then *PhysTex* is sampled to obtain the shading information. If the required tile in *PhysTex* is missing, *CoarseTex* is sampled instead.

6.2.7 Communication

PhysTex and *PageTex* are communicated to the client continuously, over TCP. *PhysTex* is encoded as an H.264 video stream. Only modifications to *PageTex* are sent, to minimise bandwidth requirements. When possible, clients decode the video stream using hardware. The camera and dynamic lights and objects are controlled by the client; the server is informed of any of these changes over UDP.

6.3 Evaluation

Table 6.1: Scene details.

Scene	Name	Triangles	Patches
S_0	Crytek	262,265	14,116
S_1	Sibenik	75,268	9,046
S_2	Sun Temple	542,629	73,519
S_3	Quake	36,949	9,053

The suitability of DARM for delivering responsive global illumination to a variety of resource-constrained client devices was evaluated through a number of experiments. The four scenes used in the tests are shown in Figure 6.7. Table 6.1 lists the number of triangles and the resulting number of parametrisation patches generated for each scene.

Table 6.2: Atlas configurations.

Scene	Quality (ppwu)	Occupancy (%)	Size (GB)
S_0	128	50.40	178.65
S_0	64	50.14	44.70
S_0	32	50.04	11.20
S_1	128	42.80	53.56
S_1	64	42.50	13.42
S_1	32	42.32	3.36
S_2	128	53.78	58.15
S_2	64	52.98	14.60
S_2	32	52.18	3.68
S_3	128	80.95	56.42
S_3	64	80.82	14.11
S_3	32	80.69	3.53

A high-end desktop equipped with a Core i9-9900K CPU, an RTX 2080 Ti GPU, 32 GB of DDR4 RAM and an SSD hard disk acted as the server. The client devices used were an Ultrabook (a mini-laptop), a Raspberry Pi 4, a smartphone and a laptop (see Table 6.6).

6.3.1 Server Performance

Table 6.2 shows the effect of increasing the megatexture quality q (in pixels per world unit) on the occupancy of the megatexture and its storage requirements. For each scene, the occupancy stayed approximately constant or marginally improved with quality. Our parametrisation and packing strategy achieved ≈50% occupancy for S_0 and S_2 and ≈42% occupancy for S_1. A high occupancy (≈80%) was obtained for S_3 presumably because this scene is mostly made up of rectangular surfaces. As q increases, the size of the megatexture increases, going beyond the available 32 GB of RAM.

The quality setting q has a big impact on the time time taken by the server to update the megatexture and to build the physical texture *PhysTex* which is transmitted over the network. Table 6.3 shows the mean and standard deviation for *PhysTex* build times (in ms) for q values of 32, 64 and 128. When q is 32 or 64, the megatexture fits or mostly fits in memory and build times are reasonable or very good. When q is 128, build times are high, due to paging. The encoding times for *PhysTex* were on average 9.89 ms and 3.72 ms for all scenes at *PhysTex* resolutions of 2048 × 2048 and 1024 × 1024 pixels respectively. The number of visible tiles also affects build times since for more tiles more work needs

Table 6.3: Physical texture build times (ms).

Scene	μ			σ		
	32	64	128	32	64	128
S_0	30	61	150	18	12	30
S_1	32	57	136	16	11	52
S_2	46	41	123	4	17	29
S_3	24	46	94	13	13	9

to be done.

6.3.2 Image Quality

The image quality obtained with the method is illustrated in Figure 6.8 (centre). The path-traced ground truth rendered on the server is displayed on the left. The physical texture communicated to the client is shown on the right with magnified areas.

The effect of q on image quality can be seen in Figure 6.9. The image quality for a specific camera viewpoint in each of the scenes was compared against path-traced

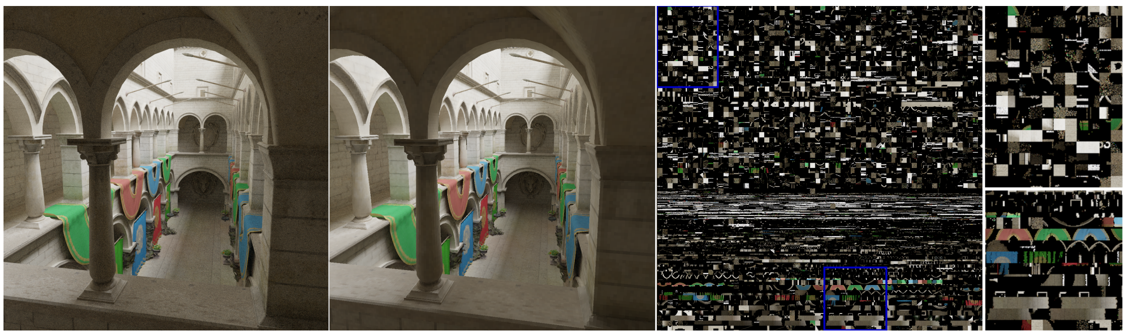


Figure 6.8: From left: Path-traced reference, DARM, the physical texture with details.

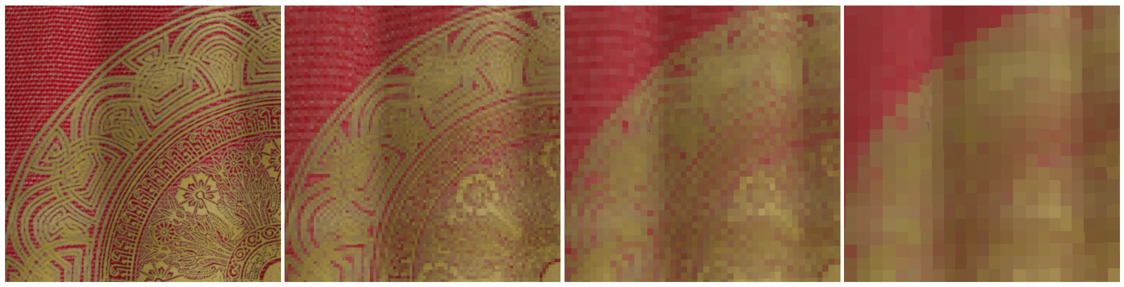


Figure 6.9: From left: Path-traced image followed by DARM images with different quality settings (128, 64 and 32 pixels per world unit).

Table 6.4: Image quality.

Scene	**PSNR**	**MSSIM**
S_0	29.8574	0.9831
S_1	32.2339	0.9857
S_2	30.9392	0.9852
S_3	29.2495	0.9858

ground truth using the Peak Signal-to-Noise Ratio (PSNR) and the Mean Structural Similarity (MSSIM) metrics (Table 6.4). In all cases, q was set to 32.

6.3.3 Network Results

Table 6.5 shows the mean and standard deviation for the bandwidth requirements of the test scenes when using 1024^2-pixel and 2048^2-pixel physical textures. The results were measured over pre-set 20-second walkthroughs. The bandwidth measurements are affected by changes to the tile layout in *PhysTex*. When the layout does not change substantially, the video-encoding algorithm is able to obtain good compression ratios, minimising the amount of data that needs to be communicated to the client.

6.3.4 Client Performance

Client performance (see Table 6.6) was measured on the same walkthroughs as for the network tests. In these tests, server updates were fixed at 10 updates per second. *PhysTex* resolution was 2048×2048 and *CoarseTex* quality was set to 8 ppwu. Performance was excellent on the laptop and the smartphone, while the Ultrabook returned average results. Very low frame rates were experienced on the Raspberry Pi 4, possibly because software video decompression was used on this device. On the other devices, video decompression was hardware-accelerated.

Table 6.5: Bandwidth (Mbps) for 1024^2- and 2048^2-pixel physical textures.

Scene	μ		σ	
	1K	2K	1K	2K
S_0	2.08	1.83	1.35	1.43
S_1	3.38	4.92	1.92	2.82
S_2	5.41	9.40	1.71	1.88
S_3	7.11	9.29	3.90	5.59

Table 6.6: Client performance at 1080p.

Client	Scene	FPS
C_0	S_0	31
	S_1	28
Ultrabook	S_2	16
Intel Core i7-5500U	S_3	28
C_1	S_0	7
	S_1	8
Raspberry Pi 4	S_2	7
	S_3	8
C_2	S_0	50
	S_1	48
Smartphone (Android)	S_2	36
Snapdragon 835	S_3	50
C_3	S_0	204
Laptop	S_1	181
Intel Core i7-6700HQ	S_2	182
GTX970M GPU	S_3	190

6.4 DARM-V

A variant of the DARM method was integrated into the Unity game engine. The variant, DARM-V, differs from DARM in a number of cardinal ways. After parametrising the scene, the parametrisation is partitioned into fixed-size logical tiles as in DARM, but the texture atlas is virtual in the sense that no space whatsoever is reserved for it, neither in memory nor on disk. Instead, in each frame, the physical texture is constructed directly from the framebuffer by mapping logical tiles onto physical texture tiles. Furthermore, physical texture tiles have a variable size. They are scaled according to the number of hits received by the logical tiles. The state and structure of the physical texture (available space, tile locations and tile scales) are maintained by a quadtree.

DARM-V was designed as an experiment to determine what sort of image quality could be obtained without a physical megatexture. Nevertheless, the concepts used have a modicum of novelty. The contributions of the method are:

- The concept of a virtual texture atlas to reduce memory requirements and avoid stalls due to slow data retrieval from disk-based storage.
- A novel level-of-detail method for mapping texture atlas tiles to scaled physical texture tiles.
- The integration of the method into a popular game engine.

6.4.1 Method

The dimensions in pixels of the physical texture are specified (typically 1024^2, 2048^2, or 4096^2). The physical texture is streamed to the client, therefore its size directly affects bandwidth. The physical texture's size also affects image quality since a larger size can accommodate more tiles and possibly larger tiles too. The desired number of logical tiles (for example, 256^2) for partitioning the parametrisation is also specified. The server regularly performs processing on all logical tiles, therefore more logical tiles directly relate to decreased server performance. The relationship between the number of logical tiles and image quality is not so straightforward because when a logical tile is mapped to the physical texture it may be scaled up or down, depending on the circumstances at that time. If a tile is scaled up but there are not enough screen space pixels to populate it, image quality will degrade. As a general heuristic, the number of logical tiles should scale with the size of the scene.

At startup, the scene is parametrised and triangle mappings are generated, from the barycentric coordinates of the scene's triangles to the parametrised triangles, which are in texture space (all coordinates between $[0, 0]$ and $[1, 1]$). Parametrisation is performed shape by shape with xatlas and packed into texture space with stb_rect_pack. A quadtree, the "Space Partitioner", is also initialised at this time. In an early iteration of the method, the quadtree supported dynamic subdivision and merging. However, these operations were too costly and were replaced by a preconfigured subdivision of the quadtree, with dynamic subdivision and merging disabled. The default preconfigured subdivision for a quadtree partitioning a 2048^2-pixel space consists of 12 quadtree levels with a number of tiles at each level. From level 0, where the tile size is 2048^2 pixels, to level 11, where a tile is a single pixel, the number of tiles in this default configuration are 0, 1, 4, 8, 16, 32, 256, 1K, 2K, 8K, 64K, and 128K respectively. A set of tile pixel thresholds specifies the relationship between the number of pixels that fall into a logical tile and the size of the corresponding physical texture tile.

Every frame starts with an info camera gathering the triangle ID and the barycentric coordinates at each pixel. The main camera then renders the scene. DARM-V operates as a postprocess, where a number of compute shaders are executed. The logical tile corresponding to each screen pixel is identified and a counter for each logical tile is updated. Once the hit counts have been obtained, a tiling strategy vacates, places, or rescales physical texture tiles and creates a mapping between logical tiles and physical texture tiles. The physical texture is now populated. Using the triangle mappings computed at startup, every screen pixel is processed in turn. The corresponding logical tile is identified (the same procedure as when obtaining tile hit counts). Using the mapping created by the tiling strategy, the current pixel's colour is assigned to the appropriate pixel within a physical texture tile. Since a number of pixels within a physical texture tile may not be set, a simple inpainting kernel is executed next, filling in missing data by averaging colours from four neighbouring pixels. Following this procedure, the physical texture is read back from the GPU and transmitted to the client together with pagetable data for each tile by which the client can retrieve pixels correctly from the physical texture. This data consists of an integer triple (x, y, scale). (x, y) are the coordinates in pixels of the tile's bottom left corner within the physical texture. The scale refers to the size of the tile (a scale $\times$ scale pixel block).

6.4.2 Evaluation

DARM-V was targeted at Unity's High Definition Render Pipeline (HDRP) due to the capability of this framework to produce extremely high quality visuals. Figure 6.10 illustrates the image quality obtained for one of the HDRP demo scenes. When the camera is stationary, some flickering artefacts may manifest but the overall image quality is reasonably good. However, during camera movement, artefacts due to exposure (missing data due to disocclusions and at the edges of the screen) greatly diminish image quality.

Mitigating these issues by maintaining a coarse megatexture on the client as in DARM requires some attention. Since the tiles in the physical texture have a variable size, a huge number of tiles (tens of thousands) can be accommodated within it. Copying all these tiles into the coarse megatexture would slow down the client significantly. However, since a large number of tiles are miniscule (1×1 texels or 2×2 texels), a better strategy would be to only copy tiles larger than a certain size.

6.5 Discussion

The results obtained by DARM are promising. However, since the size of the physical texture is limited, a better strategy is needed for populating it fairly, while reducing the amount of thrashing. Tiles are often vacated, only to be required again immediately. This causes a high amount of memory paging and produces performance bottlenecks on the server, potentially impacting image quality. These performance drops are not experienced on the client due to the use of the coarse megatexture. In our prototype the megatexture only contained a single level of detail (no mipmaps). Support for multiple levels of detail would have solved or at least greatly reduced the amount of thrashing.

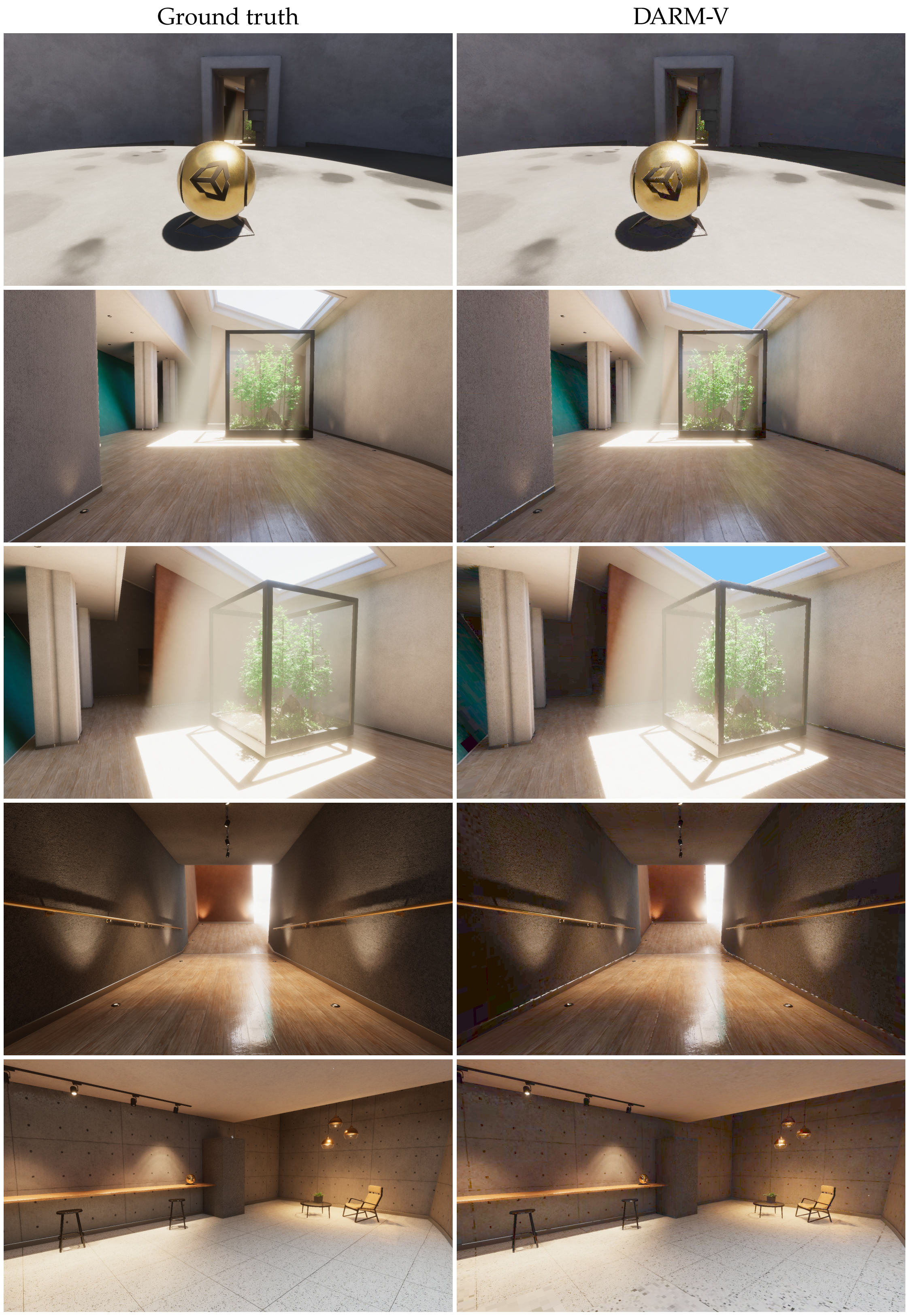


Figure 6.10: Stationary image quality. PSNR values (from top): 34.8, 25.6, 24.9, 28.5, 28.1.

Server updates are communicated over TCP. Although this ensures reliable delivery, retransmissions have an impact on delivery times. Using a network protocol that allows for dropped packets, such as RTP over UDP, would improve performance once the hurdles created by this approach are overcome. First, synchronising frames in the video stream containing the physical texture with pagetable texture data that is sent on a separate channel is challenging. Second, if the physical texture is delivered but the associated pagetable data is dropped, a recovery mechanism would be needed. The implementation is also unoptimised. For example, populating the physical texture is performed as a sequential process; parallelising this part would improve performance.

As it stands, DARM-V is really only suitable for quick scene previews. Except for static snapshots, its image quality is poor, mainly due to two factors. First, flickering visual artefacts are distracting and degrade the user experience. Second, when moving around in a scene, there are many artefacts due to missing data. Both of these issues can be greatly mitigated by maintaining a small low-quality megatexture cache that fits entirely in video memory either only on the client, or preferably on both client and server. This would improve image quality as it would enable support for progressive updates and temporal smoothing functions, while also reducing the effects of missing data. Progressive updates and smoothing functions can be implemented relatively easily; maintaining and synchronising megatexture caches requires substantial work. Since the tiling strategy used was quite complex for a GPU implementation, processing was split between the CPU and the GPU, reducing performance. Implementing the tiling strategy completely on the GPU is challenging but absolutely doable. On the plus side, the concept of a megatexture that does not use full-resolution data is interesting and merits further development.

6.6 Summary

This chapter presented a method based on sparse virtual textures and video streaming technologies. A powerful back end renders a scene and updates a megatexture with the results in real time. Relevant portions of the megatexture are then streamed to a less powerful client device. The client reconstructs images at low cost by using the megatexture portions received to shade objects instead of performing the illumination computation itself. The client is aware of the scene's geometry and responds to user input locally, eliminating input lag. To mitigate output lag, which occurs while data from the server is in transit, the client caches the data it receives in a coarse megatexture. The method has low bandwidth requirements and produces good image quality, with no restrictions on the types of materials used in the scene.

Depending on the dimensions of the scene, and on the desired image quality, the server's megatexture may be extremely large and portions of it may be paged out to disk. This may result in slow data retrieval times. An experimental second method described in this chapter avoids this problem by not creating any storage for the megatexture. Instead, the data that is streamed to the client is obtained directly from the framebuffer. The result is a method good for quick previews, with reasonably good quality images produced when the camera is stationary, and poor image quality while the camera is moving.

7 Irradiance Megatexture Cache

Image-based rendering (IBR) techniques operate on the premise that obtaining a new image by transforming an existing one is computationally less expensive than rendering a new image from scratch. One such technique, 3D warping (McMillan and Bishop, 1995b), makes use of a reference view of a virtual environment together with depth data; novel views are generated from this information at a cost depending on image resolution rather than on the geometric complexity of the scene. The method proposed in this chapter, Irradiance Megatexture Cache (IMC), uses 3D warping and draws upon ideas from ReGGI (irradiance streaming) and DARM (megatextures), combining these techniques in a novel way. IMC improves image quality over ReGGI and DARM, and is integrated into the Unity game engine. The goals of the method are similar to those of the methods described in the previous chapters, namely hiding latency while keeping bandwidth use down and computation cost low in order to achieve real-time global illumination on low-powered devices.

This chapter is structured as follows. Section 7.1 introduces the method. Each component of the method is described in detail in Section 7.2. The method is evaluated in Section 7.3 and possible extensions and future work are discussed in Section 7.4. Section 7.5 summarises and concludes the chapter.

7.1 Introduction

When generating new views using 3D warping, the central problem is that of missing data (Shum and Kang, 2000). This occurs when the scene needs to be viewed from a particular vantage point and portions of the scene are not present in any of the available reference views. A sizeable chunk of the work in IMC is dedicated to reducing and compensating for the artefacts produced by holes in the data.

Missing data can be categorised into two main types: (a) disocclusions caused by camera translation and (b) parts of the scene that were previously out of frame coming into view at the screen periphery due to sideways camera movement (strafing) or rotation.

IMC addresses these two categories separately by using a foveated view for image reconstruction. Geometry in the central part of the screen is shaded by projecting the best-fitting pixels from two reference views. This strategy reproduces the server's lighting calculations, complex as they may be, on the client at minimal cost. Geometry at the periphery of the screen is shaded cheaply on the client device itself by using server-computed irradiance, cached and progressively updated in a megatexture. Client-side shading is also used to back up reconstruction within the central region. If a required pixel is not available in either of the two reference views, or the pixel is judged to have a large error, the method falls back to using client-side shading.

IMC is intended for static scenes, and difficult scenes with many small occluders defeat the algorithm. In general, IMC softens the impact of visual artefacts and produces good quality images. User input is processed locally, effectively eliminating input latency. Output latency is reduced by the method's strategies of predicting camera movement and low-cost local rendering. The method is well-suited for moderately capable smartphones, tablets, and laptops. The contributions of the method are:

- A novel low-cost and low-latency image reconstruction strategy for resolving newly visible parts of the scene at the periphery of the screen, using a progressively updated irradiance megatexture cache.
- A novel low-bandwidth collaborative rendering technique that constructs a seamless foveated view combining dual-view 3D warping from server-generated reference views and locally reconstructed lighting using server-supplied irradiance data.
- A highly configurable method that is integrated into a widely used game engine.
- The concept of a "laggy" camera that is effective at mitigating disocclusion artefacts when the direction of movement is inverted abruptly.

7.2 Method

The architecture of the method is illustrated in Figure 7.1. The operations performed by the server and the client can be broken down as follows:

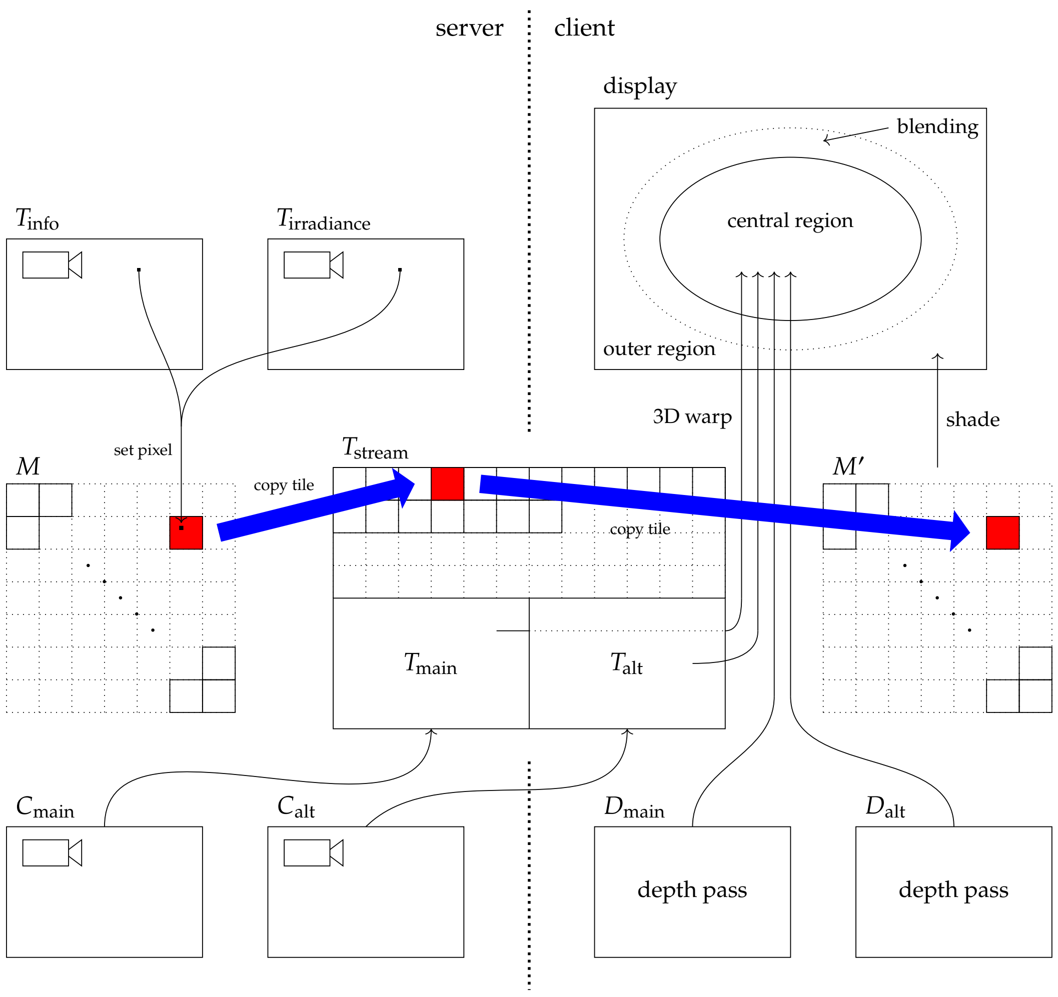


Figure 7.1: IMC architecture.

Server

1. In a precomputation stage, scan the scene to obtain geometry, material and texture information, and parametrise the scene. Communicate this information to the client when it connects.

2. Update the main camera, C_{main}, to match the client's camera, C_{client}, or to slightly lag behind it. Compute the viewing parameters of a second camera, C_{alt}, by using C_{client}'s translational and rotational speeds to extrapolate position and orientation a short time into the future.

3. Render the scene using an info pass and an irradiance pass, generating an info texture, T_{info}, and an irradiance texture, $T_{\text{irradiance}}$, respectively. T_{info} contains the

same per-pixel information as in DARM: an object ID, a primitive ID, and two barycentric coordinates. These passes both use the main camera's viewpoint.

4. Render the scene from C_{main} and C_{alt}, placing the results into two quadrants of a texture, T_{stream}, that will be streamed to the client device. These two subtextures are T_{main} and T_{alt} respectively.
5. Populate an irradiance megatexture, M, from T_{info} and $T_{\text{irradiance}}$.
6. Compute hit counts for the tiles in M. Sort the tiles in descending hit count order. Copy the first n tiles (configurable) into the unused half of T_{stream}.
7. Compress (H.264) T_{stream} and send it to the client together with viewing parameters for each of the two viewpoints and the tile IDs for the n megatexture tiles contained within it.

Client

1. Update a local irradiance megatexture, M', from the tiles received within T_{stream}.
2. Generate depth buffers D_{main} and D_{alt} corresponding to the two sets of viewing parameters received.
3. Render the geometry using a foveated view.
    - Construct the central region by 3D warping T_{main} and T_{alt} onto the current camera view. Choose each pixel either from T_{main} or T_{alt} depending on the closest depth match between the current depth buffer, D_{local}, and D_{main} and D_{alt}. If D_{main} and D_{alt} are both too different from D_{local}, shade the pixel with the procedure used for the outer region (the next item).
    - Construct the outer region from the locally stored textures (the albedo), sampling irradiance from M'. A diffuse BRDF is assumed.
    - Blend colours where the central and outer regions meet to produce a seamless result.

7.2.1 Server Architecture

The server uses the Unity[1] game engine (version 2021.3.2f1) and the Universal Rendering Pipeline (URP). It is made up of several Unity C# scripts and a native plugin (a C++ DLL).

[1] `https://unity.com/`

The plugin uses ENet[2], a UDP-based network communication layer, and the NVIDIA Video Codec SDK[3] for hardware-accelerated video encoding.

Unity's scriptable renderer features are used for the info pass and to blit the main camera's output into T_{stream}. The info pass consists of a set of legacy vertex, geometry, and fragment shaders. All the geometric objects in the scene are set up to use a material whose shader implements custom lighting via Unity's Shader Graph; this is needed to be able to compute irradiance. Compute shader kernels are used to clear and calculate megatexture tile hit counts, populate the irradiance megatexture, and copy megatexture tiles into T_{stream}. Tile sorting is performed by Buffer Sorter[4], another compute component that implements bitonic merge sort (Batcher, 1968).

7.2.2 Precomputation

On startup, the server scans the Unity scene graph, extracting all the information needed to duplicate it on the client, including node IDs and the parent-child relationships between the nodes. Game objects with mesh renderers are processed further to obtain vertices, texture coordinates, triangles, transforms, bones and joints, material information (currently texture scale/tiling and offset), and texture data. Since animation clips cannot be extracted at runtime, an editor extension was implemented to extract them and save them to disk while offline, for later readback at runtime.

The scene is parametrised using xatlas[5]. Instead of providing a triangle soup from all the shapes in the scene as input, an atlas is constructed for each shape. These subatlases are then packed into one large atlas using stb_rect_pack[6], a rectangle-packing algorithm. We have not studied the benefits of this strategy in any detail but we suspect that when computing hit counts for the megatexture tiles (Section 7.2.4) this results in fewer tiles. This is beneficial because only a limited number of tiles can be transmitted to the client in every frame, and it could save the client some computation since fewer tiles need to be copied to the client's megatexture cache. An affine map for each triangle in the scene is constructed as described in DARM (Section 6.2.3), mapping barycentric coordinates from the source 3D triangle to the triangle's 2D representation within the parameterisation.

[2] `http://enet.bespin.org/`
[3] `https://developer.nvidia.com/nvidia-video-codec-sdk`
[4] `https://github.com/EmmetOT/BufferSorter`
[5] `https://github.com/jpcy/xatlas`
[6] `https://github.com/nothings/stb/blob/master/stb_rect_pack.h`

7.2.3 Selecting the Reference Views

Mark et al. (1997) used two reference images and their z-buffers (Catmull, 1974; Straßer, 1974) to generate novel views, compensating for viewpoint translation and rotation with the image warping algorithm of McMillan and Bishop (1995a). Using a second reference view makes it possible to reduce disocclusion artefacts. Since 3D warping is performed twice, once for each image/z-buffer pair, the method is known as double warping. For every pixel in the reconstructed image, one of the two reference images is chosen. The treatment of the reference frames as meshes allows the use of a connectedness metric and a confidence value (a ratio of projected solid angles) to select the best frame. Reference view locations are chosen using future prediction, but no prediction is performed for view directions. A large field of view is used for the reference frames. Given a reference image, Shi et al. (2009, 2010) propose algorithms to search for the optimal viewpoint for a second reference image. Reinert et al. (2016) also use two reference views, but with a constant displacement for the second camera (see Section 4.2.6).

Some methods use more than two reference views, but this may be expensive on the server and during image reconstruction on the client. Hladky et al. (2019a) construct a PVS of triangles from four reference views. Popescu and Aliaga (2006) construct a depth discontinuity occlusion camera (DDOC), a non-pinhole camera model that stores extra samples around depth discontinuities. The camera generates a distorted reference image from six reference images. Since there is effectively only one reference image, this method may be useful in streaming scenarios to save bandwidth. However, extra information would also need to be transferred to reconstruct images, and the reconstruction process is too complex for weak devices.

IMC uses double warping and is most similar to Shi et al. (2009, 2010). We call our two reference views the *main* view and the alternate, *alt*, view. The camera for the main view is placed and oriented to match the last known client camera details. Since the client camera may have moved since the last update, the main camera usually lags slightly. Optionally, the main camera may be configured to lag behind even more. Not matching the main camera perfectly with the client's camera, but instead purposely introducing more lag has a useful benefit. If the client abruptly inverts the direction of movement, the slightly shifted reference view enables better mitigation of disocclusion artefacts. The position and orientation of the alt camera are extrapolated from the last known client camera parameters to a short time in the future using the client camera's translational and rotational speeds.

The client camera's movement speed is computed on the client by keeping track of position and time deltas between movement-related input events. Similarly, the client

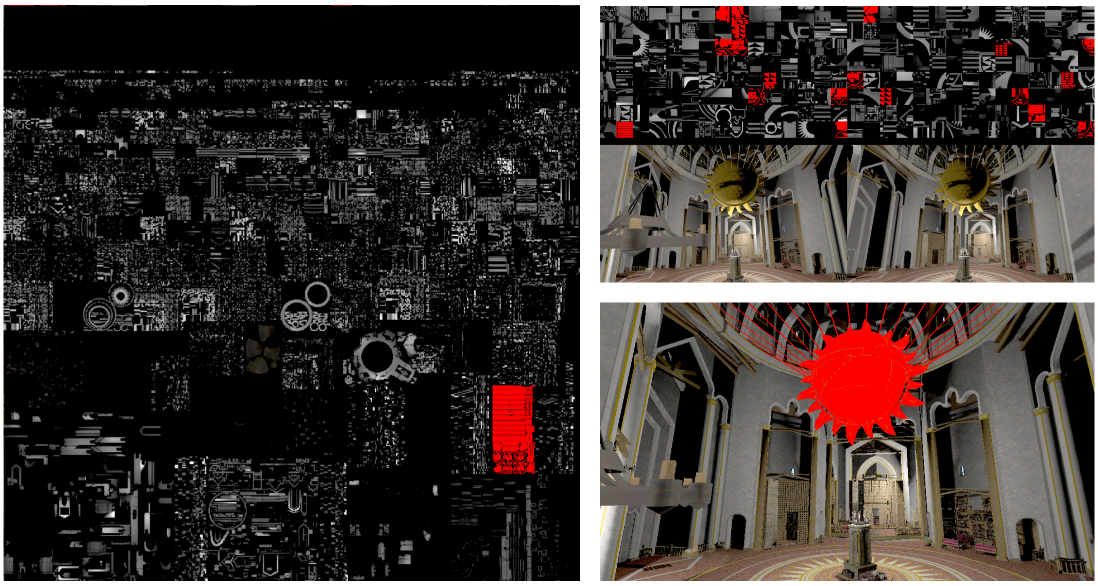


Figure 7.2: Left: The irradiance megatexture, with the Sun Temple's sun object highlighted in red. Top-right: The visible parts of the sun object are communicated to the client as tiles within a streamed texture. Bottom-right: Client view.

camera's rotational speed is computed by using spherical coordinates and calculating the rate of change of the ϕ (azimuthal) and θ (polar) angles. These three values are communicated to the server in every camera update together with the new camera position and the new view direction.

The server keeps track of the client camera's last known movement direction. Using the received client camera position and movement speed, together with the last known movement direction and a configurable "lag time" value, the server can compute the position of the main camera. If the lag time is zero, the main camera's position will match the last known client camera position. The alt camera's position is computed in a similar way, using a configurable "prediction time" value. For the main camera, the server uses the last known client camera view direction. For the alt cam, the rates of change of the viewing angles, together with a configurable "view direction prediction time" value, are used to obtain a predicted view direction. The server calculates the rotation angles around the y-axis and the camera's "right" direction, stores them as quaternions and applies them to the camera's forward direction.

7.2.4 Populating the Irradiance Megatexture

Apart from the main and alt camera passes, the server performs two other passes. These are an *info* pass and an *irradiance* pass; both passes are obtained using the main camera parameters for position and orientation. The info pass is equivalent to that used in DARM. It stores the following per-pixel information in a texture: the object ID, the primitive ID, and two barycentric coordinates for the primitive (the third barycentric coordinate is derived from these two). Recall that an affine mapping was constructed for each primitive in a precomputation step; each mapping transforms barycentric coordinates from a 3D primitive to the parametrised representation of the same primitive within a coarse irradiance megatexture (Figure 7.2, left) that resides entirely in GPU memory. After rendering all four passes, the server uses the affine mappings within a compute shader to update the irradiance megatexture from the framebuffer produced by the irradiance pass.

Ideally, the irradiance megatexture cache would only store indirect irradiance, while direct light would be computed on the device itself. This would have a number of benefits. It would allow the client to have a higher degree of independence from the server, correctly reflecting changes to illumination during short network stutters. Since diffuse indirect illumination is low frequency, it could be stored more coarsely, saving bandwidth and memory. Indirect illumination may lag behind direct illumination to some extent. It has been suggested that up to half a second of latency in indirect illumination may not be so perceptible as to bother users (Crassin et al., 2015). Notwithstanding these benefits, IMC focusses on a low reconstruction cost and avoids lighting computations on the client device. For this reason the irradiance megatexture stores "combined" irradiance, that is, a mixture of direct and indirect irradiance.

7.2.5 Constructing T_{stream}

T_{stream} is a full-screen texture that is reconstructed in every frame (Figure 7.3). It contains the main and alt views, and selected tiles from the irradiance megatexture. The main and alt views are both generated at a quarter resolution and occupy one quadrant of T_{stream} each. The remaining half of T_{stream} is reserved for irradiance megatexture tiles. The tiles are square in shape and the tile size is configurable. By default 64×64-pixel tiles are used. For efficiency, tile positions for the maximum number of tiles that can be housed in T_{stream} are precomputed.

A compute shader identifies the currently visible parts of the megatexture by generating a hit count for every logical tile in the megatexture. The info texture is processed,

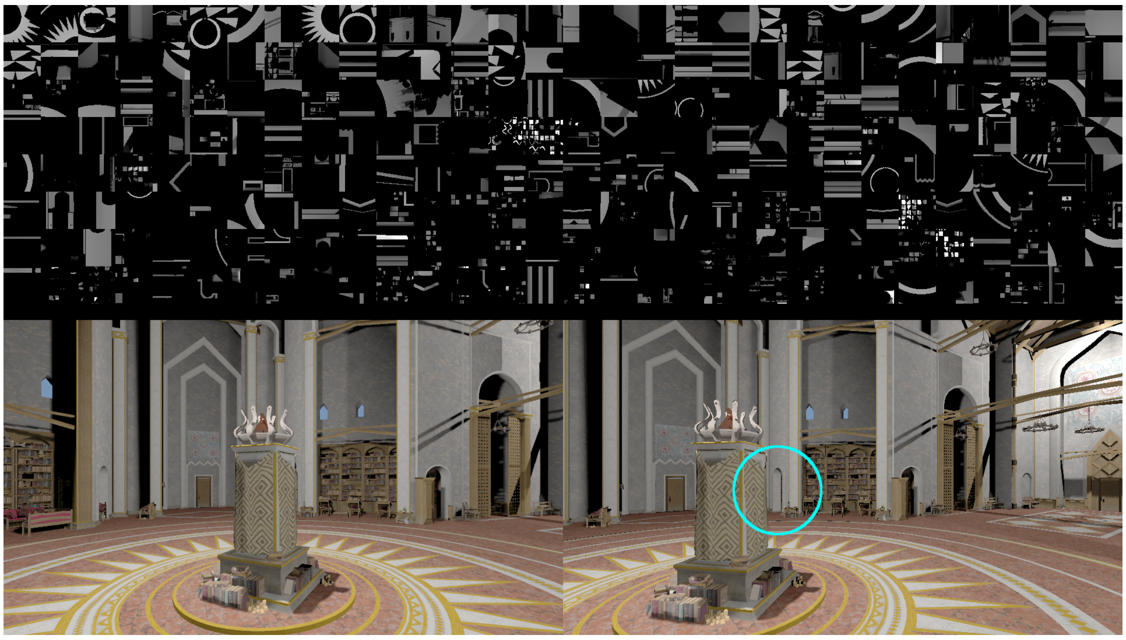


Figure 7.3: T_{stream}. The top half of the texture is made up of irradiance megatexture tiles. The bottom-left quadrant contains the main view. In the instance shown, the client's camera is moving to the right. The alt view (bottom-right quadrant) shows that this movement will disocclude a doorway behind the central pillar (circled).

and by using the same affine mappings as in Section 7.2.4, the tile associated with each pixel is identified, and a counter for that tile is incremented. The tiles are sorted in descending hit count order on the GPU, and the first few tiles (configurable) are copied into T_{stream}.

The way the two views and the megatexture tiles are packed into T_{stream} is pretty much arbitrary. The packing strategy can be modified easily. For example, if a higher quality main view is required, and the quality of the alt view is not that important, more space within T_{stream} could be reserved for the main view and the dimensions of the alt view could be reduced. The maximum number of tiles (for given tile dimensions) that would fit in the remaining space would then be calculated and their positions precalculated accordingly. T_{stream} itself could be enlarged or reduced in size as needed.

7.2.6 Compression and Communication

T_{stream} is compressed as H.264 video. The method uses two hardware-accelerated alternatives for encoding, both based on NVIDIA's NvEncoder. T_{stream} can be read back asynchronously onto the CPU and passed to NvEncoderCuda, or it could be encoded directly on the GPU using NvEncoderD3D11. Metadata needs to be sent with each video frame, specifying the server's view parameters for the main and alt views. RTP (Real-time Transport Protocol) was used in a first implementation. However, synchronising metadata with specific frames was problematic; switching to ENet provided more control. ENet is a tried-and-tested UDP-based networking library that was used in several games. It is robust and extremely simple to use.

Irradiance data is sent as an image, where pixels have intensity levels between zero and one. Since irradiance values can exceed this range, the values are scaled down by a multiple of 10; the values are scaled back up on the client. Although this forced irradiance values to be clamped to a maximum value of 10, it was sufficient for most use cases. The image data contained in T_{stream} is not gamma corrected since this is not recommended for H.264 video streaming. Gamma correction is applied on the client before presenting.

7.2.7 Client Architecture

Two versions of the client are implemented, using OpenGL and OpenGL ES. The OpenGL version was tested on Windows 10, whereas the OpenGL ES version was tested on Android 10 and 11. The client operates on two threads, the main thread and the network thread. The main thread contains the rendering loop. The network thread sends camera updates to the server, and receives T_{stream} together with the associated metadata. In general the main thread runs at a much higher rate than the network thread. When the client receives an update, it decodes T_{stream}, queues the resulting data and signals to the main thread that an update is available. The main thread checks for an update in every loop. If one is available, it updates its shaders with the view parameters received for the main and alt views, copies T_{stream} onto the GPU, and updates the irradiance megatexture cache.

On Windows, a zero-latency video decoder implemented using FFmpeg's AVCodec is used. Complete encoded frames are always passed to the parser and AVCodecParserContext's PARSER_FLAG_COMPLETE_FRAMES flag is set. In this way, the decoder always returns the decoded frame immediately. The Android version of the client is entirely implemented in C++ using the Native Development Kit (NDK) except for one small Kotlin source file in which soft keyboard input is enabled and key events are dispatched. This version of the client is slightly more complex due to the lack of low-latency decoding. When using AMediaCodec, decoded frames are only output after queueing up a number of encoded frames. To associate a decoded frame with its metadata, a presentation time is assigned to encoded frames when calling *AMediaCodec_queueInputBuffer()*. When retrieving decoded frames with *AMediaCodec_dequeueOutputBuffer()*, the presentation time previously set is retrieved too.

7.2.8 Image Reconstruction

The method uses a simple foveated view where the screen is split into a central region and an outer region. The next sections describe image reconstruction for each of these regions.

Central Region

The central region uses IBR (double 3D warping). Depth information is used to choose between the two reference views. Some related studies transmit depth or disparity, a value related to $\frac{1}{z}$ (Reinert et al., 2016) but this increases bandwidth cost and may require quantisation. In IMC depth is computed on the client itself.

The client computes depth passes for the main and alt views and since it also renders the scene itself, it also has its own z-buffer. For each fragment, the client's depth value is compared to the other two depth values to determine which view (main or alt) the current pixel colour should be obtained from. The main and alt depth values are warped to match the client's view and all three depth values are converted from log to linear quantities. The view is chosen depending on which depth value is closest to the client's depth value. The comparison is biased by a tiny amount (0.001) in favour of the main view to avoid z-fighting.

By selecting individual pixels in this way, disocclusion artefacts are greatly reduced. If the client camera moves and rotates in a predictable way, for nearby views there are essentially no disocclusion artefacts. Artefacts due to abrupt changes in movement direction are mitigated by the laggy main camera.

Outer Region

The outer region is reconstructed from the material and texture information received at startup, together with information sampled from the irradiance megatexture cache. Since the client has the geometry information for the entire scene, reconstruction is extremely simple. For each fragment, irradiance is sampled from the megatexture cache, converted to linear colour space, scaled by an order of magnitude (to reverse the scaling applied on the server) and clamped to a maximum value of 10. Similarly, the albedo is sampled and converted to linear colour space. Assuming diffuse surfaces, the BRDF is computed as $\frac{\text{albedo}}{\pi}$. A radiance value is obtained by multiplying the BRDF with irradiance. Conversion to gamma space completes the process.

Colour Blending

Small artefacts at the border between the central and outer regions are removed by colour blending. An intermediate region is defined between the central region and the screen edges. The intermediate region has the same shape as the central region but is extended using a configurable value. Colour blending for a particular pixel within the intermediate region is computed by linearly interpolating between the pixel's RGB value obtained from the selected reference view and the pixel's RGB value computed locally using the irradiance megatexture cache.

Fallback

During the reconstruction of the central region, there are two instances where a fallback mechanism is used to determine a pixel's colour. If the camera is moving or rotating too fast, the views received from the server may both be unusable. This is detected by checking warped texture coordinates for values that are out of bounds. When this occurs, the affected pixel is shaded with client-side rendering (the procedure used for reconstructing the outer region). This mechanism is also used when useable views are received from the server but the depth error for the chosen view is deemed to be too large. The depth error is the difference between the depth value from the chosen view and the depth value in the client's depth buffer.

Smoothing

Flickering artefacts may appear in the outer region when a megatexture pixel is used for shading multiple pixels that have different lighting conditions. This typically occurs at the edges of objects and is due to the way the irradiance megatexture is populated on the server. Multiple different irradiance values may be stored in the same megatexture pixel, each time overwriting the previous value. The flickering is greatly mitigated by applying a smoothing function on the server via a compute kernel. When updating the megatexture, instead of overwriting irradiance values, weights are assigned to the old and new values which are then combined:

$$E = w_{\text{old}} E_{\text{old}} + w_{\text{new}} E_{\text{new}} \tag{7.1}$$

where $w_{\text{old}}, w_{\text{new}} \in [0, 1]$ and $w_{\text{old}} + w_{\text{new}} = 1$. The default weight values are $w_{\text{old}} = 0.9$ and $w_{\text{new}} = 0.1$.

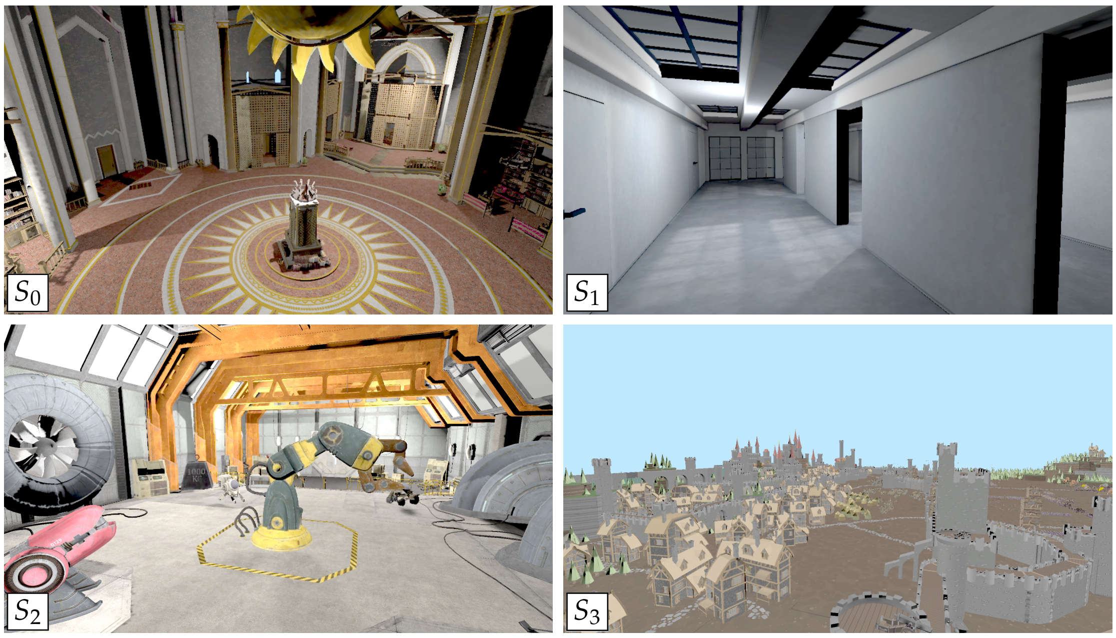


Figure 7.4: The scenes used for the evaluation. Details in Table 7.1.

7.3 Evaluation

The method was tested on a laptop (Core i7-7700HQ, NVIDIA GeForce GTX 1070 with Max-Q Design, Intel HD Graphics 630) and a smartphone (Snapdragon 835 SoC). The scenes used for the tests are shown in Figure 7.4. Scene details are provided in Table 7.1. “Occupancy” refers to the coverage obtained in the texture atlas when the scene is parametrised.

7.3.1 Output Latency

In this section the effectiveness of the latency-hiding strategies used are measured. Input latency is eliminated since action on camera movement is taken on the client itself. We are therefore concerned with measuring output latency, the delay between a user input

Table 7.1: Scene details.

Scene	Name	Triangles	Occupancy
S_0	Sun Temple	543K	82.67 %
S_1	VR Gallery	10K	56.81 %
S_2	Robot Lab	472K	82.78 %
S_3	Battle	7,251K	82.64 %

event and obtaining the expected image on the screen. Two categories of input events are considered, events that trigger camera movement and events that directly affect illumination such as turning a light source on or off. The methodology used to measure output latency is as follows:

Step 1 During a short live walkthrough, a timestamp and camera parameters (position, orientation, translational and rotational speeds) are recorded for every frame. During the walkthrough an easily distinguishable camera action is performed, such as starting to move or rotate, or changing direction. This action will serve to identify the starting time for measuring latency. At the end of the walkthrough the recorded information is saved to a file in a human-readable format.

Step 2 The frame corresponding to the distinguishable action performed in Step 1 is located in the walkthrough file. A textual indicator is manually inserted into the file at this location.

Step 3 The walkthrough is played back in a loop for a number of times (with scripted camera motion). During the looping walkthrough the screen is recorded. The screen-recording software used (NVIDIA ShadowPlay) was previously configured to record at a quality that has no impact on the client's frame rate, which is always displayed on screen along with a frame counter. A red dot is displayed when an indicator frame is encountered. When measuring latency for an illumination event, a procedure that sends a notification to the server is also started in this frame. The server will toggle a specified light source on or off when it receives the notification. Upon commencement of each loop, the red dot is cleared.

Step 4 The screen recording is played back frame by frame and analysed; a note is taken of the frame numbers when the red dot appears and when the action triggered at that instant is displayed on screen. The difference between the two frame numbers yields the output latency in frames. In the tests the frame rate was always fixed at 60 Hz. Converting the output latency from frames to milliseconds is therefore a simple matter of multiplying by 16.67 ms.

The results are displayed in Table 7.2. Every non-zero value in the table is an average from 10 tests performed on the Robot Lab scene. Output latency was measured using our own remote rendering implementation and IMC, in three configurations, Loopback, Wired, and Wi-Fi. In Loopback, the client and the server executed on the same machine.

Table 7.2: Output latency comparison between remote rendering (RR) and IMC.

Method	**GPU Readback**	**Event**	**Loopback (ms)**	**Wired (ms)**	**Wi-Fi (ms)**
RR	Async	Camera	88 (25 + 63)	100	115
IMC	Async	Camera	**0**	**0**	**0**
RR	Direct	Camera	82 (43 + 39)	98	102
IMC	Direct	Camera	**0**	**0**	**0**
RR	Async	Light	82 (20 + 62)	105	117
IMC	Async	Light	160 (42 + 118)	187	205
RR	Direct	Light	77 (37 + 40)	105	113
IMC	Direct	Light	97 (65 + 32)	112	133

In the other two configurations the client device was the laptop, connecting to the server over a wired connection and over Wi-Fi. In the Loopback column non-zero timings are broken down into a sum of server and client components. For example in the first row the output latency was 88 ms. The event was displayed on the server after 25 ms and on the client 63 ms later.

Transferring frames (the framebuffer in remote rendering and T_{stream} in IMC) or any other kind of data from the GPU to the CPU is a notoriously expensive operation, the larger the data the slower the transfer. In our architecture, retrieved frames are placed in a queue. A worker thread consumes the queued frames, encodes them as video if they are not already encoded, and sends them to the client over the network. Two GPU readback mechanisms, *Async* (asynchronous readback) and *Direct* (direct encoding) were tested and compared. *Async* has negligible impact on the server's rendering thread but the nature of the mechanism introduces several extra frames of latency. Furthermore, since hardware-accelerated video encoding is used, dequeued frames are sent back to the GPU for processing, after which the smaller encoded frame is read back. *Direct* avoids this ping-ponging between the GPU and the CPU. Here the video encoder operates directly on the framebuffer or texture already in video memory, and then retrieves the smaller encoded version. The downside of this mechanism is that the video encoder needs to block the server's rendering thread; this may have a slight impact on the server's frame rate. On the other hand, this ensures that the server's rendering thread and the video encoder are always automatically synchronised, avoiding any flow control issues if rendering and encoding operate at different speeds.

As expected, output latency always decreased when a faster medium was used. In every test the lowest latency was experienced with Loopback (memory), then on a wired connection, then over Wi-Fi. *Direct* was always equal or faster than *Async*; the speed improvement was marginal in RR but huge in IMC for the illumination tests. In fact in these tests, when *Direct* was used, IMC latencies were close (within 20 ms) to those in RR; when *Async* was used, IMC latencies were nearly double those in RR.

In RR, for each medium the output latencies were pretty much the same in all tests (the largest difference was 15 ms, which is less than one frame at 60 Hz). In IMC, output latency was completely eliminated when considering only camera movement. The expected action always appeared on screen in the same frame as the indicator. The same results (no latency) were always obtained in both the central and outer regions of the screen for different reasons. In the central region, the double 3D warping strategy coupled with camera extrapolation for the alternate view was effective at hiding latency. Latency due to camera movement in the outer region is never experienced since this region is rendered on the client.

In the outer region, visual artefacts manifesting as bright or dark regions that stayed on the screen permanently were observed on rare occasions. Lighting is reconstructed on the client using texture information received when the client first connects to the server and irradiance information that is continuously updated and cached in a megatexture. Since in each frame, only the megatexture tiles that have the highest hit counts make it into T_{stream}, tiles with a low hit count may never be included, causing the artefacts. Moving towards the affected regions increases the hit counts for these tiles, at which point they are included, the megatexture is updated and the artefacts disappear. IMC supports sending a larger number of megatexture tiles, over a number of frames. This mitigates the issue but increases output latency. The artefacts appear so rarely that this strategy is rarely used, if ever.

7.3.2 Image Quality

The quality of the client's reconstructed frames was evaluated by comparing against ground truth using the perceptual DSSIM metric (Loza et al., 2006). The DSSIM implementation used was version 3.2.0 and was obtained from GitHub[7]. Due to the method's real-time nature, performing the comparison at runtime is not possible since it slows down the system too much and introduces significant latency. Instead, the state of the system is recorded during a live walkthrough. With the information gathered, the walkthrough can be reproduced faithfully, frame by frame, at a later time, without time budget or latency concerns. During the replay, ground truth images and the corre-

[7] `https://github.com/kornelski/dssim/releases`

Table 7.3: Mean DSSIM values for the lossy T_{stream} plots in Figure 7.5.

Scene	DSSIM
Sun Temple	0.0436
VR Gallery	0.0056
Robot Lab	0.0830
Battle	0.1367

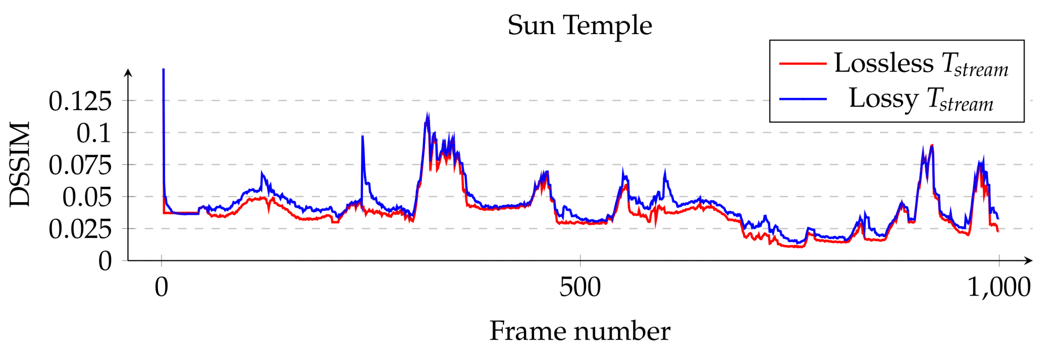
sponding reconstructed frames are extracted. The process is broken down into three steps.

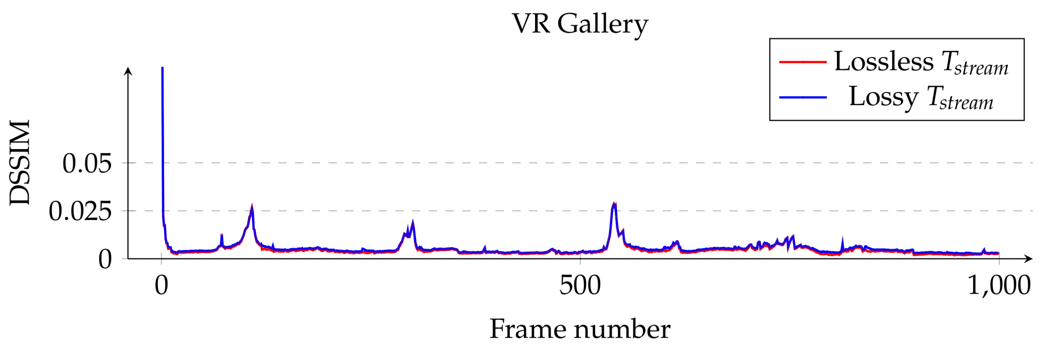
Step 1 A recording feature on the client saves meta information while the user moves around in the scene. The data recorded for every frame consists of a timestamp, the client camera's parameters (position, orientation, translational and rotational speeds), information about the two server views that were used to reconstruct the frame (the position and orientation of the server's main and alt cameras), and the tile IDs of any irradiance megatexture tiles that were updated in the frame before reconstruction. This is the only step that executes at runtime. It saves the data in memory and is extremely lightweight. At the end of the recording, the walkthrough data is saved to disk as a text file.

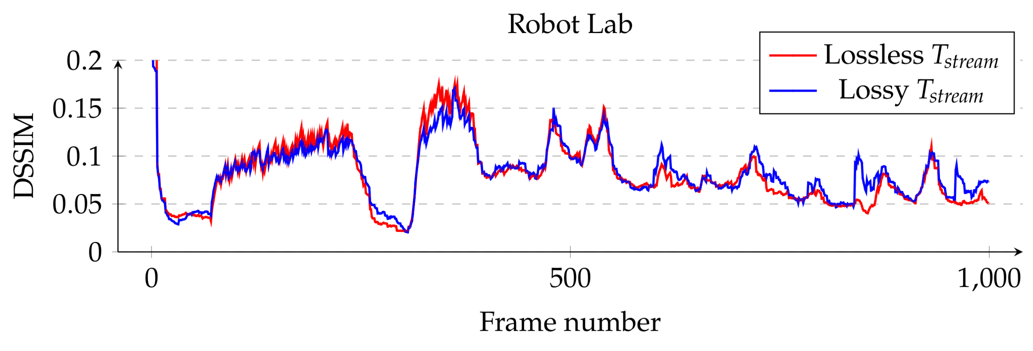
Step 2 The server is run in a special standalone playback mode, without a connected client. In this mode, for every frame recorded in the walkthrough, the server generates T_{stream} and the related ground truth image and saves them to disk. The ground truth image is a full resolution view using the client camera's parameters.

Step 3 The client is run in a special playback mode where it connects to the server only to receive scene information. The walkthrough file and the stream textures generated by the server in Step 2 are all loaded into memory. Using this data, the client produces a reconstructed frame for every entry in the walkthrough, perfectly regenerating the frames of the original live walkthrough of Step 1. The reconstructed frames are saved to disk.

Using this methodology, image quality was evaluated for a number of walkthroughs featuring various scenes and move sets (slow, fast, and stationary, moving forwards, backwards, sideways, and around objects, and rotating the view direction vertically and horizontally). The irradiance megatexture tile sizes were always 64×64 pixels and the server was configured to send at most 240 tiles in every T_{stream}. A laggy main camera







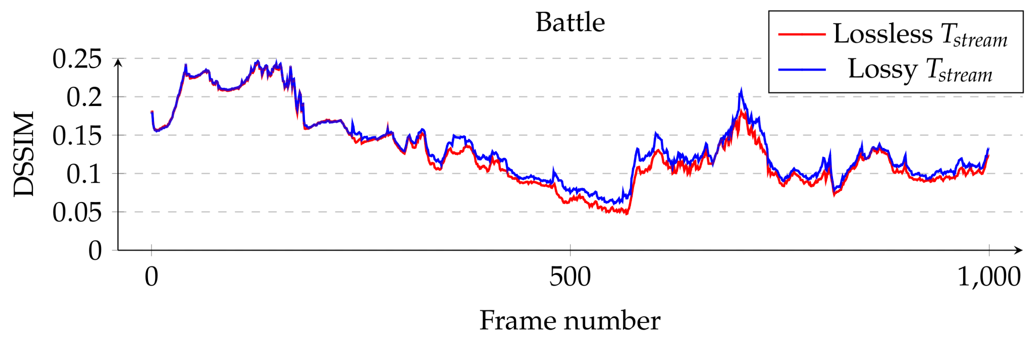


Figure 7.5: Image quality (DSSIM).

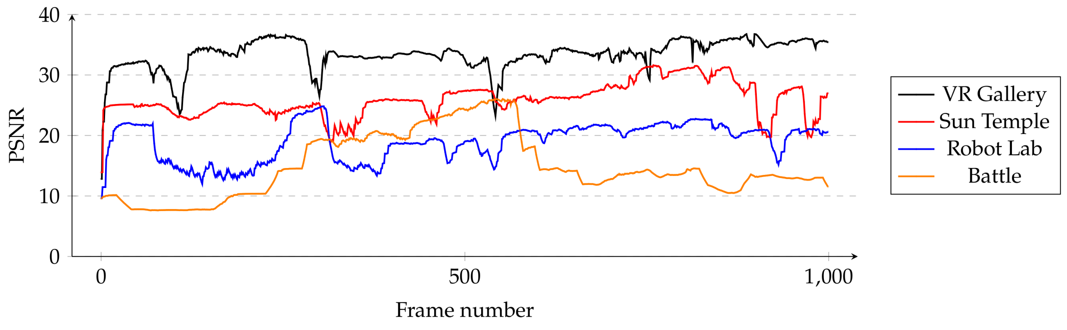


Figure 7.6: Image quality (PSNR).

was not used in these tests; the server's main camera parameters were always set to the last known client camera parameters. DSSIM results are displayed in Figure 7.5. As expected, a slight reduction in image quality is due to the lossy (H.264) compression used for T_{stream} (the blue line). The image quality obtained when using a lossless T_{stream} is shown as a reference (the red line). The mean DSSIM values for Figure 7.5 are listed in Table 7.3. Image quality results for the walkthroughs using the PSNR metric are displayed in Figure 7.6.

The results indicate good image quality, with DSSIM values below 0.1 for most of the walkthroughs for all scenes except for S_3 (Battle). Since this scene is extremely large, the irradiance stored in the megatexture is greatly undersampled and particularly coarse. This contributed towards the relatively high DSSIM values registered when compared with the other scenes. However, the main reason for these results is due to the fact that IMC does not reproduce sky boxes or background colours on the client; a fixed solid background colour is always used on the client. Comparing sky box or background colour pixels will invariably result in mismatches. The Battle scene is the only open-air scene, with a portion of the sky visible on screen throughout most of the walkthrough. At the start of the walkthrough, the sky features prominently as the camera zooms down to ground level from a high altitude. This explains the downward trend that appears in Figure 7.6.

To some extent, the mean DSSIM result obtained for the Robot Lab scene can be compared to results obtained by other researchers. Reinert et al. (2016) use the same scene but the exact walkthrough details are not available; their "Kawahai" method obtained the best score, 0.0543, from the various methods tested. IMC scored a mean DSSIM of 0.0830 for this scene. The result obtained by IMC is slightly worse (by less than 0.03). Nevertheless, we are encouraged by the fact that our walkthroughs were

Table 7.4: Client performance.

	Desktop (ms)	**Laptop (ms)**	**Smartphone (ms)**
Rendering (avg.)	2.0*	2.7*	19.3
Rendering (min.)	0.4*	0.7*	6.4
Rendering (max.)	5.8*	17.1*	39.4
Megatexture update (avg.)	0.5 / 2.2*	2.5*	N/A
Megatexture update (min.)	0.3 / 1.7*	2.5*	N/A
Megatexture update (max.)	1.9 / 3.5*	2.7*	N/A
Depth passes (avg.)	0.4	0.8	5.6
Depth passes (min.)	0.3	0.8	3.7
Depth passes (max.)	0.4	0.9	9.3

* Compute shader used for megatexture updates.

performed with unseeded irradiance megatextures and with reference views at quarter resolution. Improving these two aspects is straightforward and should result in an immediate increase in image quality.

7.3.3 Client Performance

Table 7.4 indicates timings for rendering entire frames and parts of a frame on each of three devices for the Robot Lab scene. The client's resolution was always 1080p. Average, minimum, and maximum values are shown. *Rendering* covers one iteration of the entire render loop (including megatexture population and the depth passes); the timings were obtained from 12,000 frames. *Megatexture update* refers to the copying of the megatexture tiles received within T_{stream} into the client's megatexture cache. These values were obtained from 1,000 frames. This step was implemented using OpenGL texture copying functions and with a compute shader. Surprisingly, the compute shader implementation (marked with an asterisk) was slower. For this reason, a compute shader implementation was not used on the smartphone. Values for the megatexture updates are missing for the smartphone because timing part of a frame was problematic on this device. The timings recorded were too low (practically zero) to be correct.

On the desktop, there is what appears to be a contradiction between the average rendering time (2.0 ms) and the average megatexture update time using the compute shader implementation (2.2 ms) because the former includes the latter. This situation can occur because megatexture updates do not occur in every frame. On a powerful machine such as the desktop, where frame rates of hundreds or even thousands of Hz can be obtained when VSync (vertical sync) is disabled, megatexture updates only occur on a small percentage of the total number of frames. This results in the average rendering time being dominated by frames where no megatexture updates occur. *Depth passes* (values also obtained from 1,000 frames) includes the two depth passes (for the main and alt views) performed whenever an update from the server is received. All operations are very cheap on the desktop and the laptop.

7.3.4 Bandwidth

Bandwidth readings were obtained every second over periods of 120 seconds. The values were queried on the client from the ENet library, which keeps track of the total data sent and received. Figure 7.7 illustrates two sets of bandwidth readings per scene for server-to-client data. The bit rate for video compression was configured to 8 Mbps for the first set (marked in blue) and to 20 Mbps for the second set (marked in red). In all scenes, bandwidth readings for client-to-server data were pretty constant, fluctuating between 0.05 Mbps and 0.06 Mbps.

Server Updates

The setup parameters for the video encoder are configurable. In our standard setup and for these tests, the video encoder was configured to use a bit rate of either eight or 20 Mbps with a frame rate of 60 and a GOP (Group of Pictures) size of 120. The packet structure for one server-to-client update is shown in Figure 7.8. An update consists of 76 bytes of metadata together with video-compressed data (T_{stream}) and a variable number of four-byte integers containing megatexture tile IDs. The metadata consists of 16 bytes for packet structure (type, payload length, video data length and number of tile IDs), four bytes that are used for measuring the RTT (frame marker) and 56 bytes containing parameters for the main and alt cameras.

The irradiance megatexture is intended to be coarse and for most scenes storing tile IDs as four-byte integers is overkill. Using two-byte integers instead would save bandwidth while still allowing for a 64K-tile megatexture (256×256 tiles). The tile IDs could be compressed for even more savings, although this would add extra computation on the client side. To save bandwidth and client processing, the maximum number of megatexture tiles to include in T_{stream} is configurable. Moreover, tiles can also be sent in blocks so that as much of the megatexture as needed can be communicated to the client over a number of frames. In all tests a single block with a maximum of 240 megatexture tiles was used.

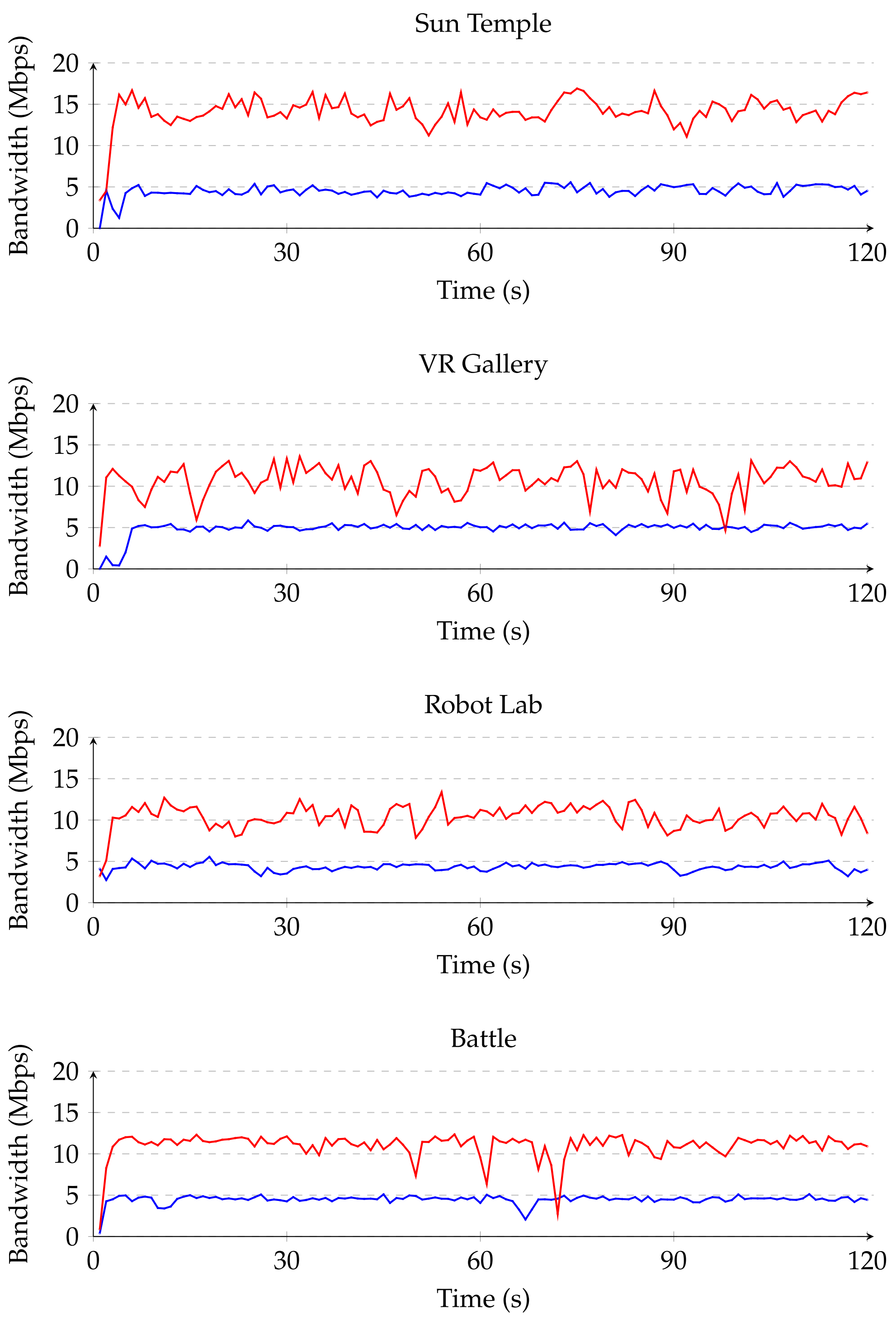


Figure 7.7: Server-to-client bandwidth for configured bit rates of 8 Mbps (blue) and 20 Mbps (red).

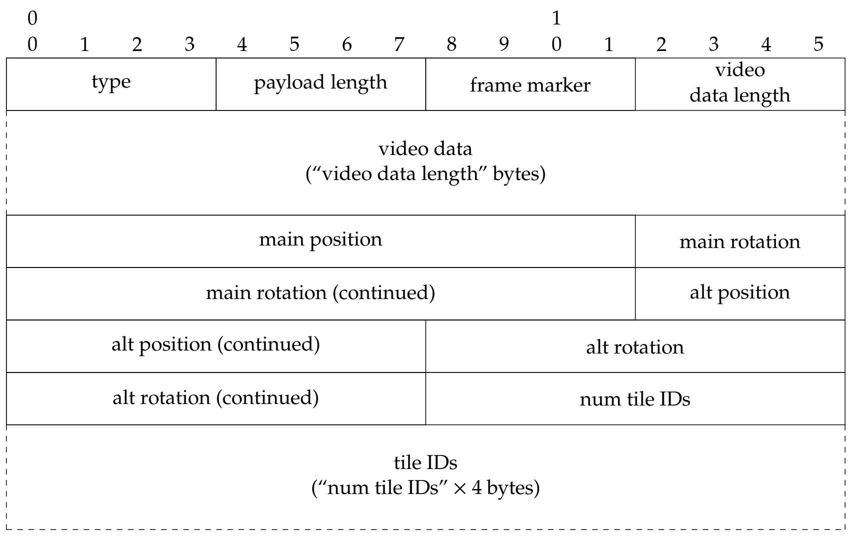


Figure 7.8: Packet structure for server updates. The numbers at the top represent bytes (for example one "row" is 16 bytes long and the "type" field is four bytes long).

Client Updates

The client communicates its camera's parameters (position and forward direction, translational and rotational speeds) to the server whenever the position or forward direction changes (this is checked once per frame) and at least four times a second. The packet size for one client-to-server update is fixed at 40 bytes (Figure 7.9). Four bytes are for packet structure (type), 24 bytes describe the camera's position and forward direction, four bytes are for translational speed and 8 bytes are for the rotational speeds for θ, the polar angle, and ϕ, the azimuthal angle.

7.3.5 Client Storage Requirements

The irradiance megatexture is designed to contain a coarse representation of the irradiance in the scene. It is not backed up by any storage and should fit comfortably in memory. The storage requirements for each scene are summarised in Table 7.5. Since

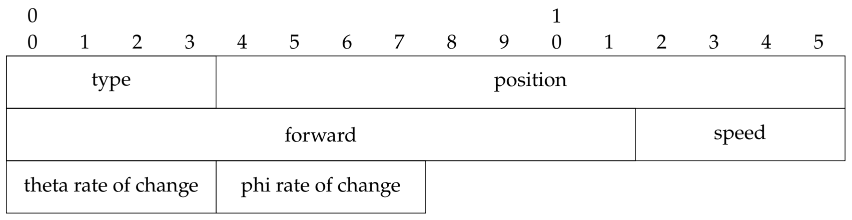


Figure 7.9: Packet structure for client updates.

Table 7.5: Client storage requirements. Megatexture dimensions are in tiles. One tile is 64×64 texels. A texel takes up four bytes.

Scene	**Name**	**Megatexture Dimensions**	**Size (MB)**
S_0	Sun Temple	64×64	64
S_1	VR Gallery	16×16	4
S_2	Robot Lab	64×64	64
S_3	Battle	128×128	256

low-frequency indirect lighting can be reproduced plausibly from sparse irradiance samples, as shown in our results for ReGGI in Chapter 5, even coarser megatextures may be sufficient for scenes S_0 and S_2. On the other hand, huge scenes such as S_3 may require much larger megatextures. To avoid exceeding the limitations on texture sizes imposed on mobile devices (for example the maximum texture size allowed on the smartphone used in the tests is 16K × 16K texels), the method can be extended to use texture arrays instead, with relatively little effort. Note that since tile IDs are communicated as four-byte integers, a maximum megatexture size of 64K × 64K tiles is implied. This limitation may be worked around by modifying the tile-addressing scheme.

7.4 Discussion

IMC has low bandwidth requirements. Three out of the four scenes tested used less than 15 Mbps, while the fourth (Sun Temple) used less than 20 Mbps. These low readings were in part obtained by sacrificing some image quality (the two reference views are communicated at quarter resolution) and in part by not communicating any depth information for the two views. Instead, depth is computed on the client device itself (also at quarter resolution) increasing reconstruction cost. If bandwidth is cheap, the inverse approach could be used, that is, using more bandwidth to transfer higher resolution reference views and their depth information to obtain better image quality and lower reconstruction cost. An alternative approach would be to use simplified geometry on the client, as in Reinert et al. (2016), to lower the cost of computing depth information on the client device itself.

Several methods capture reference views using a wide camera angle. We have shied away from using a similar strategy for two reasons. First, the deformed views obtained in this way could negatively impact image quality. Second, fast camera rotation speeds could defeat this mechanism. However, investigating this approach could be beneficial as it could provide the means to populate the irradiance megatexture ahead of time, reducing the exposure problem at the edges of the screen.

IMC requires scene parametrisation on the server at startup. Depending on the scene this could be practically instantaneous (VR Gallery) or it could take a while (150 s for the Battle scene). Assuming static geometry, an easy fix would be to compute the parametrisation only once, save it to disk, and reload it at startup. Another drawback is that the scene data, possibly hundreds of megabytes (uncompressed), needs to be transferred to the client device when it connects. With good compression the amount of data transferred can be drastically reduced. Moreover, the data could be transferred only once and stored locally for future use. The biggest drawbacks of the method are the lack of support for dynamic geometry and general BRDFs. Supporting other BRDFs in IBR is possible, as shown by several researchers, for example Cabral et al. (1999) and Sinha et al. (2012). Dynamic geometry is a hard problem for IBR techniques, and in a distributed IBR approach such as IMC it is particularly problematic as near-perfect synchronisation of the two endpoints is required for good results. Non-IBR techniques are definitely more suitable for dynamic geometry.

In DARM, megatextures were used to store or cache radiance. IMC uses megatextures to cache irradiance. These methods show that megatextures are effective at storing scene-wide illumination data. The data stored can be fine or coarse, can be progressively refined, and allows for a measure of fault tolerance during short network outages. Furthermore, in adaptations of these methods, the megatexture data computed on the server could be used to support multiple clients sharing the virtual environment. The versatility of the megatexture representation is promising. For future work, supporting level-of-detail in the megatexture is a priority, after which we would like to experiment with storing illumination data using other representations (for example, spherical harmonics) to support view-dependent illumination.

7.5 Summary

The method described in this chapter, IMC, proposes collaborative rendering between the client and the server, with the primary aim of reducing input latency. The client needs to be capable of processing the same vertices and triangle primitives as the server. However, computation is reduced by using simple shaders. Complex lighting calculations, arguably the most expensive part of the pipeline, are eliminated. This is achieved by reconstructing frames using image-based methods and lighting from an irradiance megatexture cache that is continuously and progressively updated by the server. The method does not support dynamic geometry and assumes diffuse materials. It produces good image quality, uses little bandwidth, is tolerant to network stability issues, and is integrated into the Unity game engine.

8 Conclusion

The computational expense of global illumination is too high for a broad spectrum of commodity hardware, from desktops to laptops and tablets, to smartphones and untethered VR headsets. On these devices, high-fidelity visuals at elevated frame rates can instead be obtained by distributed rendering. One solution is remote rendering, also known as full-frame streaming, which requires a stable network connection, may use substantial bandwidth, and generates lag. Notwithstanding these issues, which may increase costs for users and degrade their experience, this is the solution used by all cloud-gaming operators. This choice makes perfect business sense as it provides numerous benefits for the providers, such as the elimination of piracy and immediate support for any kind of device, it is easy to implement, and produces good image quality.

This thesis investigated alternative distributed rendering strategies with the aim of enhancing user experience over that provided by remote rendering. The methods presented all eliminate input lag and target weak to moderately powerful devices, providing varying levels of image quality. The system presented in Chapter 5 (ReGGI) splits the rendering pipeline into two. Direct illumination is computed on the device itself, while sparse irradiance samples are computed on the server through a voxelised representation of the scene and communicated to the client device. Using interpolation, indirect illumination is reconstructed on the client and combined with the locally computed lighting to obtain global illumination. Chapter 6 describes a method (DARM) where the server stores computed radiance in a high-quality megatexture and communicates the relevant portions of it to the client device, where a coarse version of the megatexture that fits in memory is maintained. The reconstruction cost on the client is minimal as no lighting is computed locally and shading only requires texture sampling from high-quality megatexture tiles when these are available and from the coarse megatexture otherwise. A variant of the method, DARM-V, uses a virtual texture atlas without any backing storage, sacrificing image quality to reduce storage requirements and to avoid stalls due to slow data retrieval from a disk-based megatexture. Chapter 7 details a technique (IMC) that combines megatextures and image-based rendering. The server computes and caches irradiance samples in a small coarse megatexture that fits in memory, and synchronises the megatexture with a copy maintained on the client. The server also updates the client with two slightly different views of the scene. Using a foveated view, the client reconstructs the central part of the frame using double warping, and the outer part from the megatexture and the locally stored albedo. The megatexture also serves as a backup for any missing data in the central part of the frame.

ReGGI is suitable for providing plausible diffuse global illumination in shared virtual environments. DARM provides a combination of good image quality and low reconstruction cost, with no limitations as to the types of materials used, whereas DARM-V is useful for quick scene previews. IMC is limited to diffuse materials but provides the best image quality. All methods except for DARM-V can tolerate short network stutters.

8.1 Contributions

The main contribution of this thesis is to add credence to the idea that local computation on commodity devices, although insignificant when compared to that of high-end desktops, can contribute towards enhancing user experience in distributed rendering systems. The methods presented all eliminate input lag and use asynchronous distributed rendering strategies to provide real-time global illumination to a wide range of devices.

8.1.1 Regular Grid Global Illumination (ReGGI)

Ward et al. (1988) showed that a good approximation for indirect diffuse illumination can be obtained by interpolation from irradiance samples stored at surface points around the scene. Greger et al. (1998) computed diffuse global illumination by sampling field irradiance stored sparsely on a bi-level grid. Inspired by these observations, ReGGI superimposes a regular grid of configurable density over the scene and identifies up to two representative points in grid cells that contain geometry. In every frame, VPLs (Keller, 1997) are fired from the light sources and used to compute and progressively update irradiance at all the points stored on the grid. Indirect illumination is then produced by interpolating irradiance from these points and combined with locally computed direct illumination.

The results show that sparse samples on medium-resolution grids can reproduce global illumination effects such as colour bleeding effectively. The best interpolation method we tested with, modified Shepard's method (Franke and Nielson, 1980), was computationally too expensive for the smartphone and the Oculus Quest. Frame rates between 7 and 47 were obtained using trilinear interpolation on the smartphone, depending on the scene and the amount of downsampling used. Image quality had to be sacrificed on the Oculus Quest to obtain usable frame rates, indicating that image reconstruction needs to be optimised further to lower its cost. The system scaled excellently in tests with up to eight simultaneous clients since indirect illumination was computed only once and reused.

List of Contributions

- A scalable cloud-based global illumination solution that requires little bandwidth and no precomputation and is suitable for weak devices such as smartphones.
- Elimination of input lag.
- Amortisation of server-side computations over multiple connected clients.

8.1.2 Device-Agnostic Radiance Megatextures (DARM)

Megatextures were created to store large detailed terrains with unrepeated features, to improve the realism of virtual environments. DARM uses megatextures to store the results of global illumination computed on a powerful back end. Portions of the shaded megatexture that correspond to the currently visible portions of the scene are communicated to a client device. This allows the client to shade the scene at a low cost, simply by sampling from the megatexture, while supporting all kinds of materials. The client maintains a small coarse version of the megatexture, which it uses to create novel views from previously communicated data until newer and more detailed data is received from the server.

The method was tested with megatextures of various qualities, using a pixels-per-world-unit (ppwu) quality metric. The resulting megatextures ranged from 3 GB to 179 GB. For megatextures up to or slightly larger than 32 GB, which either fit entirely in memory or only needed a small amount of paging, server performance was smooth and efficient. For larger megatextures, when memory paging was needed, noticeable slowdowns were observed. Bandwidth requirements were low, with mean values always less than 10 Mbps, and good image quality was obtained, with PSNR values at least 29 and MSSIM values around 0.98. Due to the low reconstruction cost, frame rates were better than in ReGGI; frame rates between 36 and 50 Hz were attained on a smartphone.

List of Contributions

- A novel distributed rendering pipeline for high-fidelity graphics based on radiance megatextures.
- A network-based out-of-core algorithm that circumvents VRAM limitations without sacrificing texture variety.
- Automatic precomputation for texture atlas generation.
- A client-side coarse cache that mitigates artefacts due to missing data and makes the system robust to network fluctuations.

8.1.3 DARM Virtual Atlas (DARM-V)

Large disk-based megatextures caused drops in server performance in DARM. In a variant of the method, DARM-V, we studied the effects of doing away with megatexture storage altogether, and only used a logical partitioning of the scene's parametrisation to identify which portions of the scene were currently visible. This enabled efficient population of the physical texture, directly from the framebuffer. A coarse megatexture was not used on the client as in DARM, so no progressive updates or smoothing functions were possible. This led to poor image quality when moving around the scene, but reasonably good image quality when stationary. The method also developed a novel level-of-detail technique for megatextures. Typically, megatexture and physical texture tiles are of the same size. In DARM-V, megatexture tiles are purely logical constructs, not tied to any number of pixels. This allows the size of tiles within the physical texture to be scaled as necessary, depending on the screen-space dimensions of the objects in that part of the megatexture.

List of Contributions

- The concept of a virtual texture atlas to reduce memory requirements and avoid stalls due to slow data retrieval from disk-based storage.
- A novel level-of-detail method for mapping texture atlas tiles to scaled physical texture tiles.
- The integration of the method into a popular game engine.

8.1.4 Irradiance Megatexture Cache (IMC)

Although good image quality was obtained by DARM, it is inferior to that obtained by remote rendering because of two reasons. Megatexture tiles may contain data that are of a lower quality than needed, and it is often the case that the physical texture is not large enough to contain all the needed tiles. On the other hand, a full frame is equivalent to a perfectly packed physical texture, containing the exact amount of pixels needed for that frame. Image-based rendering techniques can warp frames to create novel views but have to contend with the exposure problem (Shi et al., 2009) where there is missing data due to disocclusion or new parts of the scene coming into view as the camera moves. IMC combines image-based rendering and megatextures to create novel views while mitigating the exposure problem. A coarse megatexture that caches server-computed irradiance allows the client to render parts of a frame at low cost. Double warping is used to reduce disocclusion artefacts, and is backed up by the aforementioned client-side rendering which also counteracts exposure at the edges of the screen.

IMC eliminates input lag and output latency due to camera movement. Output latency for changes in illumination is close to that experienced in remote rendering, on average 20 ms worse. Better image quality than in DARM was obtained, with DSSIM values of 0.083 or less for three out of the four scenes tested. For the large outdoor fourth scene, irradiance was undersampled, reducing image quality. Furthermore, the lack of sky box support increased dissimilarity when comparing against ground truth. A higher (worse) DSSIM score of 0.14 was obtained for this scene. The average frame generation time on the smartphone was 18.6 ms (a frame rate of 54 Hz). At an average of around 15 Mbps, bandwidth requirements were low.

List of Contributions

- A novel low-cost and low-latency image reconstruction strategy for resolving newly visible parts of the scene at the periphery of the screen, using a progressively updated irradiance megatexture cache.
- A novel low-bandwidth collaborative rendering technique that constructs a seamless foveated view combining dual-view 3D warping from server-generated reference views and locally reconstructed lighting using server-supplied irradiance data.
- A highly configurable method that is integrated into a widely used game engine.

- The concept of a "laggy" camera that is effective at mitigating disocclusion artefacts when the direction of movement is inverted abruptly.

8.2 Findings and Insight

Devices that lack the computational capability to synthesise high-fidelity visuals at high frame rates can instead receive the fully rendered frames as a video stream using remote rendering. However, remote rendering is reliant on a stable network, and lag that exceeds a certain threshold is a deal breaker in some applications (Beigbeder et al., 2004). When the conditions of network stability or low latency are not met, the distributed rendering strategies proposed in our research are effective at improving user experience over that provided by the streaming approach. The methods presented confirm the findings of earlier studies. Distributed rendering pipelines can eliminate input lag, produce high image quality, keep bandwidth requirements low, and have the potential to cut costs for both providers and clients through sublinear server scaling (Bugeja et al., 2018; Crassin et al., 2015). Output lag due to changes in illumination is less perceivable for low-frequency indirect illumination (Crassin et al., 2015) and smoothing functions can be used to increase the perceived update rate for this kind of illumination (Bugeja et al., 2018). Image-based rendering techniques can eliminate output lag due to camera movement (Mark et al., 1997) and produce high image quality. Our work also shows that megatextures, even coarse ones, are suitable for storing scene-wide illumination data, and that network stutters can be tolerated with caching strategies.

Good image quality was obtained in most methods. The best image quality was obtained with IMC, the method that makes use of image-based rendering techniques. The various strategies used yielded a mixture of reconstruction costs with corresponding frame rates; proxy geometry, as in Reinert et al. (2016), could be used to improve client performance while simultaneously limiting concerns about piracy of detailed models (Koller et al., 2004). All methods showed good support for dynamic illumination. We also experimented (empirical observations only) with a small number of dynamic objects. Support for dynamic geometry is problematic in distributed rendering methods due to the difficulty of synchronising objects between the distributed pipeline's endpoints. ReGGI obtained plausible shading for dynamic geometry. In this method, high-frequency illumination (direct illumination) is computed on the client device itself; synchronisation with the remote endpoint is not required for this part of the computation. The client receives low-frequency indirect illumination, so output lag is less perceivable. In DARM and DARM-V, the client receives radiance, which includes high-frequency direct illumination; output lag manifesting as trailing shadows was immediately evident and distracting. Initial tests on IMC indicated that the method was not compatible with dynamic geometry.

Asynchronous distributed rendering is significantly more complex to implement than remote rendering. Particularly problematic is the synchronisation of locally and remotely computed data; this causes output lag which is detrimental to the user experience and hinders support for dynamic scenes. It is clear that hiding output lag effectively is indispensable for increasing the viability of ADR. Methods that require substantial processing power in order to reduce output lag may gain traction as the performance of commodity devices improves. Hybrid RR-ADR solutions could also become a possibility, using RR with the best image quality when some input lag is acceptable, and ADR with simpler graphics when input lag needs to be reduced as much as possible. It would be interesting to see how users would respond to such an experience.

8.3 Limitations and Future Work

Supporting large scenes in ReGGI requires using a coarse regular grid since server performance is inversely proportional to the number of grid cells. However, coarser grids reduce image quality. Using a sparse spatial subdivision structure such as an octree would make nearest-neighbour searches more complex but would save space and enable support for larger scenes. Server processing can be reduced by only computing lighting for cells close to the player(s). Optimising the interpolation methods would produce higher frame rates making the method more suitable for VR environments.

DARM can be improved with a better paging strategy to avoid the stalls that occur when a large megatexture is used. Making better use of the limited space in the physical texture using a level-of-detail mechanism would limit the amount of thrashing and improve image quality. DARM-V would greatly benefit from caching coarse megatexture data as is done on the client in DARM and at both ends of the distributed pipeline in IMC.

The image quality in IMC can be immediately improved by using more bandwidth to stream a larger texture containing the main view (or both views) at a higher resolution, possibly even at full resolution. Furthermore, the megatexture cache is currently populated with global (both direct and indirect) irradiance. If direct illumination is rendered on the client device itself, the megatexture would only contain indirect irradiance. For low frequency lighting, this data could be stored at a lower resolution, saving space or making space for high frequency lighting data.

8.4 Final Remarks

Distributed rendering solutions bring realistic high-quality visuals to all kinds of devices, including the relatively weak and increasingly ubiquitous mobile ones. The computational capability of these devices will increase over time. However, physical constraints prohibit these devices from being able to generate high-fidelity visuals on their own steam for the foreseeable future. It can be safely said that distributed rendering is here to stay, and therefore building upon today's techniques with the aim of providing users with better experiences is a worthwhile endeavour. This thesis contributes to the existing body of knowledge by presenting a number of real-time global illumination methods targeted at this domain. ReGGI provides approximate global illumination from extremely sparse irradiance samples and scales well with multiple clients. DARM allows devices to reconstruct high-fidelity visuals at a low cost, while DARM-V improves server performance. IMC uses concepts from DARM to improve image quality while retaining a low reconstruction cost. These methods address a number of important challenges faced in distributed rendering and will hopefully inspire further research in this area.

References

Aggarwal, V., Debattista, K., Bashford-Rogers, T., Dubla, P., and Chalmers, A. High-fidelity Interactive Rendering on Desktop Grids. *IEEE Comput. Graph. Appl.*, 32(3):24–36, 2012.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. SETI@home: An Experiment in Public-resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

Appel, A. Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 37–45. ACM, 1968.

Arvo, J. Backward Ray Tracing. In *Developments in Ray Tracing, Computer Graphics, Proc. of ACM SIGGRAPH 86 Course Notes*, pages 259–263, 1986.

Arvo, J. and Kirk, D. Particle Transport and Image Synthesis. *ACM SIGGRAPH Computer Graphics*, 24(4):63–66, 1990.

Barrett, S. Sparse Virtual Textures. Game Developers Conference, 2008. Available: `http://silverspaceship.com/src/svt/`. [Accessed April 5, 2023].

Batcher, K. E. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 307–314, 1968.

Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J., Agu, E., and Claypool, M. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003®. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, pages 144–151, 2004.

Benthin, C., Wald, I., and Slusallek, P. A Scalable Approach to Interactive Global Illumination. In *Comput. Graph. Forum*, volume 22, pages 621–630. Wiley Online Library, 2003.

Bird, I. Computing for the Large Hadron Collider. *Annu. Rev. Nucl. Part. Sci.*, 61:99–118, 2011.

Blinn, J. F. Models of Light Reflection for Computer Synthesized Pictures. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, pages 192–198, 1977.

Blinn, J. F. Hyperbolic Interpolation. *IEEE Comput. Graph. Appl.*, 12(4):89–94, 1992.

Brabec, S., Annen, T., and Seidel, H.-P. Shadow Mapping for Hemispherical and Omnidirectional Light Sources. In *Advances in Modelling, Animation and Rendering*, pages 397–407. Springer, 2002.

Brouillat, J., Gautron, P., and Bouatouch, K. Photon-driven Irradiance Cache. In *Comput. Graph. Forum*, volume 27, pages 1971–1978. Wiley Online Library, 2008.

Bugeja, K., Debattista, K., Spina, S., and Chalmers, A. Collaborative High-fidelity Rendering over Peer-to-peer Networks. In *EGPGV@EuroVis*, pages 9–16, 2014a.

Bugeja, K., Debattista, K., Spina, S., and Chalmers, A. High-fidelity Graphics for Dynamically Generated Environments Using Distributed Computing. In *2014 6th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pages 1–8. IEEE, 2014b.

Bugeja, K., Debattista, K., and Spina, S. An Asynchronous Method for Cloud-based Rendering. *Vis. Comput.*, pages 1–14, 2018.

Cabral, B., Olano, M., and Nemec, P. Reflection Space Image Based Rendering. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 165–170, 1999.

Catmull, E. E. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. The University of Utah, 1974.

Choy, S., Wong, B., Simon, G., and Rosenberg, C. The Brewing Storm in Cloud Gaming: A Measurement Study on Cloud to End-user Latency. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, page 2. IEEE Press, 2012.

Christensen, P. H., Harker, G., Shade, J., SCHU-BERT, B., and Batali, D. Multiresolution Radiosity Caching for Efficient Preview and Final Quality Global Illumination in Movies. *Pixar Technical Memos*, pages 1–10, 2012.

Claypool, M. and Claypool, K. Latency and Player Actions in Online Games. *Communications of the ACM*, 49(11):40–45, 2006.

Cook, R. L. and Torrance, K. E. A Reflectance Model for Computer Graphics. *ACM Transactions on Graphics*, 1(1):7–24, 1982.

Cook, R. L., Porter, T., and Carpenter, L. Distributed Ray Tracing. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 137–145. ACM, 1984.

Cook, R. L., Carpenter, L., and Catmull, E. The Reyes Image Rendering Architecture. *ACM SIGGRAPH Computer Graphics*, 21(4):95–102, 1987.

Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. Gigavoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pages 15–22, 2009.

Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. Interactive Indirect Illumination using Voxel Cone Tracing. In *Comput. Graph. Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

Crassin, C., Luebke, D., Mara, M., McGuire, M., Oster, B., Shirley, P., and Wyman, P.-P. S. C. CloudLight: A System for Amortizing Indirect Lighting in Real-time Rendering. *J. Comp. Graph. Tech.*, 4(4), 2015.

Cuervo, E., Wolman, A., Cox, L. P., Lebeck, K., Razeen, A., Saroiu, S., and Musuvathi, M. Kahawai: High-quality Mobile Gaming Using GPU Offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 121–135, 2015.

Dachsbacher, C. and Stamminger, M. Reflective Shadow Maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 203–231, 2005.

Debattista, K., Santos, L. P., and Chalmers, A. Accelerating the Irradiance Cache through Parallel Component-based Rendering. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, pages 27–35, 2006.

Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. *ACM SIGGRAPH Computer Graphics*, 22(4):21–30, 1988.

Devlin, K., Chalmers, A., Wilkie, A., and Purgathofer, W. Tone Reproduction and Physically Based Spectral Rendering. In *Eurographics (State of the Art Reports)*, 2002.

Dutré, P., Bala, K., and Bekaert, P. *Advanced Global Illumination*. A K Peters, 2006.

Evangelou, I., Papaioannou, G., Vardis, K., and Vasilakis, A. A. Rasterisation-based Progressive Photon Mapping. *Vis. Comput.*, 36(10):1993–2004, 2020.

Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., and Schönherr, S. On the Design of CGAL a Computational Geometry Algorithms Library. *Softw: Pract. Exp.*, 30(11):1167–1202, 2000.

Franke, R. and Nielson, G. Smooth Interpolation of Large Sets of Scattered Data. *International Journal for Numerical*

Methods in Engineering, 15(11):1691–1704, 1980.

Gautron, P., Krivánek, J., Pattanaik, S. N., and Bouatouch, K. A Novel Hemispherical Basis for Accurate and Efficient Rendering. *Rendering Techniques*, 2004:321–330, 2004.

Goral, C. M., Torrance, K. E., Greenberg, D. P., and Battaile, B. Modeling the Interaction of Light Between Diffuse Surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 213–222. ACM, 1984.

Greger, G., Shirley, P., Hubbard, P. M., and Greenberg, D. P. The Irradiance Volume. *IEEE Comput. Graph. Appl.*, 18:32–43, 1998.

Hachisuka, T. and Jensen, H. W. Stochastic Progressive Photon Mapping. In *ACM SIGGRAPH Asia 2009 Papers*, pages 1–8. ACM, 2009.

Hachisuka, T., Ogaki, S., and Jensen, H. W. Progressive Photon Mapping. In *ACM SIGGRAPH Asia 2008 Papers*, pages 1–8. ACM, 2008.

Haeberli, P. and Akeley, K. The Accumulation Buffer: Hardware Support for High-quality Rendering. *ACM SIGGRAPH Computer Graphics*, 24(4):309–318, 1990.

He, X. D., Torrance, K. E., Sillion, F. X., and Greenberg, D. P. A Comprehensive Physical Model for Light Reflection. *ACM SIGGRAPH Computer Graphics*, 25(4):175–186, 1991.

Heckbert, P. S. Adaptive Radiosity Textures for Bidirectional Ray Tracing. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 145–154, 1990.

Herzog, R., Eisemann, E., Myszkowski, K., and Seidel, H.-P. Spatio-temporal Upsampling on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 91–98, 2010.

Hladky, J., Seidel, H.-P., and Steinberger, M. The Camera Offset Space: Real-time Potentially Visible Set Computations for Streaming Rendering. *ACM Transactions on Graphics*, 38:1–14, 2019a.

Hladky, J., Seidel, H.-P., and Steinberger, M. Tessellated Shading Streaming. In *Comput. Graph. Forum*, volume 38, pages 171–182. Wiley Online Library, 2019b.

Hood, D. C. and Finkelstein, M. A. Sensitivity to Light. In *Handbook of Perception and Human Performance*, volume 1, chapter 5. Wiley, 1986.

Jensen, H. W. Global Illumination Using Photon Maps. In *Rendering Techniques '96*, pages 21–30. Springer, June 1996.

Jensen, H. W. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

Jylänki, J. A Thousand Ways to Pack the Bin - A Practical Approach to Two-dimensional Rectangle Bin Packing, 2010. Available: `https://github.com/juj/RectangleBinPack/blob/master/RectangleBinPack.pdf`. [Accessed April 5, 2023].

Kajiya, J. T. The Rendering Equation. *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 20(4):143–150, August 1986.

Kajiya, J. T. and Kay, T. L. Rendering Fur with Three-Dimensional Textures. *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, 23(3):271–280, 1989.

Kán, P. and Kaufmann, H. Differential Irradiance Caching for Fast High-quality Light Transport Between Virtual and Real Worlds. In *IEEE International Symposium on Mixed and Augmented Reality*, pages 133–141. IEEE, 2013.

Kaplanyan, A. Light Propagation Volumes in CryEngine 3. *ACM SIGGRAPH Courses*, 7(2), 2009.

Kaplanyan, A. and Dachsbacher, C. Cascaded Light Propagation Volumes for Real-time Indirect Illumination. In *Proceedings of the 2010 Symposium on Interactive 3D Graphics*, pages 99–107. ACM, 2010.

Keller, A. Instant Radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 49–56. ACM Press/Addison-Wesley Publishing Co., 1997.

Keller, A. and Heidrich, W. Interleaved Sampling. In *Eurographics Workshop on Rendering Techniques*, pages 269–276. Springer, 2001.

Knaus, C. and Zwicker, M. Progressive Photon Mapping: A Probabilistic Approach. *ACM Transactions on Graphics*, 30 (3):1–13, 2011.

Koller, D., Turitzin, M., Levoy, M., Tarini, M., Croccia, G., Cignoni, P., and Scopigno, R. Protected Interactive 3D Graphics via Remote Rendering. *ACM Transactions on Graphics*, 23(3):695–703, 2004.

Korf, R. E., Moffitt, M. D., and Pollack, M. E. Optimal Rectangle Packing. *Ann. Oper. Res.*, 179(1):261–295, 2010.

Krivánek, J., Gautron, P., Pattanaik, S., and Bouatouch, K. Radiance Caching for Efficient Global Illumination Computation. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):550–561, 2005.

Krivánek, J., Bouatouch, K., Pattanaik, S. N., and Zara, J. Making Radiance and Irradiance Caching Practical: Adaptive Caching and Neighbor Clamping. *Rendering Techniques*, 2006:127–138, 2006.

Lafortune, E. P. and Willems, Y. D. Bi-directional Path Tracing. In *Proceedings of the 3rd International Conference on Computational Graphics and Visualization Techniques (CompuGraphics '93)*, pages 145–153, 1993.

Lagarde, S. and De Rousiers, C. Moving Frostbite to Physically Based Rendering. In *SIGGRAPH 2014 Conference*, 2014.

Laine, S. and Karras, T. Efficient Sparse Voxel Octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8): 1048–1059, 2010.

Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978. doi: 10.1145/359545.359563.

Lee, K., Chu, D., Cuervo, E., Kopf, J., Degtyarev, Y., Grizan, S., Wolman, A., and Flinn, J. Outatime: Using Speculation to Enable Low-latency Continuous Interaction for Mobile Cloud Gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165, 2015.

Lensing, P. and Broll, W. Efficient Shading of Indirect Illumination Applying Reflective Shadow Maps. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 95–102, 2013.

Lévy, B., Petitjean, S., Ray, N., and Maillot, J. Least Squares Conformal Maps for Automatic Texture Atlas Generation. In *ACM Transactions on Graphics*, volume 21, pages 362–371. ACM, 2002.

Litzkow, M. J., Livny, M., and Mutka, M. W. Condor - A Hunter of Idle Workstations. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1987.

Loza, A., Mihaylova, L., Canagarajah, N., and Bull, D. Structural Similarity-based Object Tracking in Video Sequences. In *2006 9th International Conference on Information Fusion*, pages 1–6. IEEE, 2006.

Majercik, Z., Guertin, J.-P., Nowrouzezahrai, D., and McGuire, M. Dynamic Diffuse Global Illumination with Ray-traced Irradiance Fields. *J. Comp. Graph. Tech.*, 8(2), 2019.

Majercik, Z., Marrs, A., Spjut, J., and McGuire, M. Scaling Probe-based Real-time Dynamic Global Illumination for Production. *CoRR*, 2020. Available: `https://arxiv.org/abs/2009.10796`. [Accessed April 5, 2023].

Mark, W. R., McMillan, L., and Bishop, G. Post-rendering 3D Warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pages 7–16, 1997.

McCluney, W. R. *Introduction to Radiometry and Photometry*. Artech House, 2014.

McGuire, M. and Luebke, D. Hardware-accelerated Global Illumination by Image Space Photon Mapping. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 77–89. ACM, 2009.

McGuire, M., Mara, M., Nowrouzezahrai, D., and Luebke, D. Real-time Global Illumination using Precomputed Light

Field Probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–11, 2017.

McMillan, L. and Bishop, G. Head-tracked Stereoscopic Display Using Image Warping. In *Stereoscopic Displays and Virtual Reality Systems II*, volume 2409, pages 21–30. SPIE, 1995a.

McMillan, L. and Bishop, G. Plenoptic Modeling: An Image-based Rendering System. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pages 39–46, 1995b.

Mittring, M. Advanced Virtual Texture Topics. In *ACM SIGGRAPH 2008 Games*, pages 23–51. ACM, 2008.

Mueller, J. H., Voglreiter, P., Dokter, M., Neff, T., Makar, M., Steinberger, M., and Schmalstieg, D. Shading Atlas Streaming. *ACM Transactions on Graphics*, 37(6):1–16, 2018.

Mutka, M. W. and Livny, M. Profiling Workstations' Available Capacity for Remote Execution. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1987.

Nenci, F., Spinello, L., and Stachniss, C. Effective Compression of Range Data Streams for Remote Robot Operations using H.264. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3794–3799. IEEE, 2014.

Neumann, L., Neumannn, A., and Szirmay-Kalos, L. Compact Metallic Reflectance Models. In *Comput. Graph. Forum*, volume 18, pages 161–172. Wiley Online Library, 1999.

Oren, M. and Nayar, S. K. Generalization of Lambert's Reflectance Model. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, pages 239–246, 1994.

Owen, A. B. Monte Carlo Extension of Quasi-Monte Carlo. In *1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274)*, volume 1, pages 571–577. IEEE, 1998.

Pająk, D., Herzog, R., Eisemann, E., Myszkowski, K., and Seidel, H.-P. Scalable Remote Rendering with Depth and Motion-flow Augmented Streaming. In *Comput. Graph. Forum*, volume 30, pages 415–424. Wiley Online Library, 2011.

Palmer, J. M. Radiometry and Photometry: Units and Conversions. In *Handbook of Optics: Volume III: Classical, Vision, & X-ray Optics*, pages 7–1. McGraw-Hill, 2000.

Palmer, J. M. and Grant, B. G. *The Art of Radiometry*. SPIE Press, Bellingham, 2010.

Pantel, L. and Wolf, L. C. On the Suitability of Dead Reckoning Schemes for Games. In *Proceedings of the 1st Workshop on Network and System Support for Games*, pages 79–84, 2002.

Patoli, M. Z., Gkion, M., Al-Barakati, A., Zhang, W., Newbury, P., and White, M. An Open Source Grid Based Render Farm for Blender 3D. In *2009 IEEE/PES Power Systems Conference and Exposition*, pages 1–6. IEEE, 2009.

Patoli, Z., Gkion, M., Al-Barakati, A., Zhang, W., Newbury, P., and White, M. How to Build an Open Source Render Farm Based on Desktop Grid Computing. In *International Multi Topic Conference*, pages 268–278. Springer, 2008.

Pece, F., Kautz, J., and Weyrich, T. Adapting Standard Video Codecs for Depth Streaming. In *EGVE/EuroVR*, pages 59–66, 2011.

Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2016.

Phong, B. T. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, 1975.

Popescu, V. and Aliaga, D. The Depth Discontinuity Occlusion Camera. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 139–143, 2006.

Prutkin, R., Kaplanyan, A., Dachsbacher, C., et al. Reflective Shadow Map Clustering for Real-time Global Illumination. In *Eurographics (Short Papers)*, pages 9–12, 2012.

Ray, N., Nivoliers, V., Lefebvre, S., and Lévy, B. Invisible Seams. In *Comput. Graph. Forum*, volume 29, pages 1489–1496.

Wiley Online Library, 2010.

Reinert, B., Kopf, J., Ritschel, T., Cuervo, E., Chu, D., and Seidel, H.-P. Proxy-guided Image-based Rendering for Mobile Devices. In *Comput. Graph. Forum*, volume 35, pages 353–362. Wiley Online Library, 2016.

Reinhard, E., Stark, M., Shirley, P., and Ferwerda, J. Photographic Tone Reproduction for Digital Images. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 267–276, 2002.

Ritschel, T., Grosch, T., Kim, M. H., Seidel, H.-P., Dachsbacher, C., and Kautz, J. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Transactions on Graphics*, 27(5):1–8, 2008.

Saito, T. and Takahashi, T. Comprehensible Rendering of 3-D Shapes. *Computer Graphics*, 24(4):197–206, August 1990.

Schwartz, C., Ruiters, R., and Klein, R. Level-of-detail Streaming and Rendering using Bidirectional Sparse Virtual Texture Functions. In *Comput. Graph. Forum*, volume 32, pages 345–354. Wiley Online Library, 2013.

Seitz, S. M. and Dyer, C. R. View Morphing. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 21–30, 1996.

Shi, S. and Hsu, C.-H. A Survey of Interactive Remote Rendering Systems. *ACM Computing Surveys (CSUR)*, 47(4):1–29, 2015.

Shi, S., Jeon, W. J., Nahrstedt, K., and Campbell, R. H. Real-time Remote Rendering of 3D Video for Mobile Devices. In *Proceedings of the 17th ACM International Conference on Multimedia*, pages 391–400, 2009.

Shi, S., Kamali, M., Nahrstedt, K., Hart, J. C., and Campbell, R. H. A High-quality Low-delay Remote Rendering System for 3D Video. In *Proceedings of the 18th ACM International Conference on Multimedia*, pages 601–610, 2010.

Shum, H. and Kang, S. B. Review of Image-based Rendering Techniques. In *Visual Communications and Image Processing 2000*, volume 4067, pages 2–13. SPIE, 2000.

Sinha, S. N., Kopf, J., Goesele, M., Scharstein, D., and Szeliski, R. Image-based Rendering for Scenes with Reflections. *ACM Transactions on Graphics*, 31(4):1–10, 2012.

Sloan, P.-P., Kautz, J., and Snyder, J. Precomputed Radiance Transfer for Real-time Rendering in Dynamic, Low-frequency Lighting Environments. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 527–536, 2002.

Sloan, P.-P., Luna, B., and Snyder, J. Local, Deformable Precomputed Radiance Transfer. *ACM Transactions on Graphics*, 24(3):1216–1224, 2005.

Stengel, M., Majercik, Z., Boudaoud, B., and McGuire, M. A Distributed, Decoupled System for Losslessly Streaming Dynamic Light Probes to Thin Clients. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pages 159–172, 2021.

Straßer, W. *Fast Display of Curves and Surfaces on Graphic Display Devices [Schnelle Kurven- und Flächendarstellung auf grafischen Sichtgeräten]*. PhD thesis, TU Berlin, 1974. Available: `https://diglib.eg.org/handle/10.2312/2631196`. [Accessed April 5, 2023].

Sugihara, M., Rauwendaal, R., and Salvi, M. Layered Reflective Shadow Maps for Voxel-based Indirect Illumination. In *High Performance Graphics*, pages 117–125, 2014.

Tabellion, E. and Lamorlette, A. An Approximate Global Illumination System for Computer Generated Films. *ACM Transactions on Graphics*, 23:469–476, 2004.

Van Steen, M. and Tanenbaum, A. S. *Distributed Systems*. Maarten van Steen, 2017.

Van Waveren, J. id Tech 5 Challenges - From Texture Virtualization to Massive Parallelization. *Talk in Beyond Programmable Shading Course, SIGGRAPH*, 9:5, 2009.

Van Waveren, J. Software Virtual Textures. Technical report, id Software LLC, 02 2012.

Vardis, K., Papaioannou, G., and Gkaravelis, A. Real-time Radiance Caching using Chrominance Compression. *J. Comp. Graph. Tech.*, 3(4), 2014.

Vardis, K., Vasilakis, A. A., and Papaioannou, G. Illumination-driven Light Probe Placement. In *Eurographics 2021 Posters*. Eurographics Association, 2021.

Veach, E. *Robust Monte Carlo Methods for Light Transport Simulation*. Stanford University, 1998.

Veach, E. and Guibas, L. Bidirectional Estimators for Light Transport. In *Photorealistic Rendering Techniques*, pages 145–167. Springer, 1995.

Veach, E. and Guibas, L. J. Metropolis Light Transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 65–76, 1997.

Wald, I., Slusallek, P., and Benthin, C. Interactive Distributed Ray Tracing of Highly Complex Models. In *Eurographics Workshop on Rendering Techniques*, pages 277–288. Springer, 2001a.

Wald, I., Slusallek, P., Benthin, C., and Wagner, M. Interactive Rendering with Coherent Ray Tracing. In *Comput. Graph. Forum*, volume 20, pages 153–165. Wiley Online Library, 2001b.

Wald, I., Kollig, T., Benthin, C., Keller, A., and Slulassek, P. Interactive Global Illumination using Fast Ray Tracing. Rendering Techniques (2002), 15–24. In *Proceedings of the 13th Eurographics Workshop on Rendering Techniques*, volume 28, pages 15–24. Eurographics Association, 2002.

Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. Microfacet Models for Refraction Through Rough Surfaces. *Proceedings of the Eurographics Symposium on Rendering Techniques*, 2007:18th, 2007.

Ward, G. J. Measuring and Modeling Anisotropic Reflection. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pages 265–272, 1992.

Ward, G. J. and Heckbert, P. S. Irradiance Gradients. Technical report, Lawrence Berkeley Lab., CA (United States); Ecole Polytechnique Federale, 1992.

Ward, G. J., Rubinstein, F. M., and Clear, R. D. A Ray Tracing Solution for Diffuse Interreflection. *ACM SIGGRAPH Computer Graphics*, 22(4):85–92, 1988.

Welford, B. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3):419–420, 1962.

Whitted, T. An Improved Illumination Model for Shaded Display. In *ACM SIGGRAPH Computer Graphics*, volume 13, page 14. ACM, 1979.

Williams, L. Casting Curved Shadows on Curved Surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, pages 270–274, 1978.

Young, J. xatlas. `https://github.com/jpcy/xatlas`, 2020. [Accessed April 5, 2023].

Zhao, Y., Belcour, L., and Nowrouzezahrai, D. View-dependent Radiance Caching. In *Graphics Interface*, pages 22–1, 2019.