Department of Informatics
BSc in Computer Science

LIGHT MAP GENERATION USING RAY TRACING

Authors: Gkanatsios Panagiotis, Exadaktylos Theodoros
Supervisor: Papaioannou Georgios

Athens, September 2021

Abstract

Nowadays, ray tracing is the most recognized technique in the department of computer graphics. By simulating the flow of light with rays it produces high visual realistic scenes. However, rendering photorealistic scenes real-time with this technique is an issue that needs to be addressed. This thesis establishes the issues of ray tracing and proposes an implementation that overcomes its flaws. It introduces an offline ray tracing algorithm that samples the global illumination of the scene and encapsulates the outcome on a light map. This method accomplishes high quality lighting in the scenes with fast real-time construction time.

Contents

1 Introduction

1.1 Historical overview

Lightmapping is the process of pre-calculating lighting information and storing it in a texture , a light map. Lightmaps are most efficient if many of the light sources and geometry are static or environmental. They work fine on low-end PCs and mobile devices, as they consume almost no resources at run-time.

Before lightmaps were invented, real-time computer graphics applications relied purely on Gouraud shading technique [GH71]. In the early '90s John Carmack and Michael Abrash were trying to get rid of some of the Gouraud Shading technique disadvantages to enhance the quality of their video-game Quake [MA00]. After that lightmaps became more popular and engines began to combine light maps with other techniques, like ray tracing.

Ray tracing is a rendering technique for sampling paths of light in a 3D scene. It works by simulating actual light rays, using an algorithm to trace the path that a beam of light would take in the physical world. This technique is capable of producing high visual realism and is considered the top standard mechanism for image synthesis in computer graphics.

In physics, ray tracing is a method for calculating the path of waves or particles through a system with regions of varying propagation velocity, absorption characteristics, and reflecting surfaces. The algorithm of ray tracing on computer graphics is deeply inspired by it and the first implementation was in 1968 by Appel [AA68]. Appels algorithm was simple, send one ray for each pixel and if an intersection with an object is found send a ray to the light source to check for shadows. A significant step further was taken by Whitted , in 1980. By taking advantage of the prior algorithm he developed a recursive procedure, with the ray tracing algorithm as its core routine.

Today ray tracing and light maps are often used in computer graphics in a variety of techniques. The first one offers high visual realism and the second one low cost implementation, so the combination of these two techniques is unavoidable.

1.2 Problem Description

Ray tracing is a process that may not be too complex to implement, but there often is a part in its implementation which causes a bottleneck. The most complicated part of the process is the intersection of rays with 3D models, objects that may reside inside the 3D world. The main issue with these intersections is that timewise and in memory, they are very expensive.

The main problem of ray tracing is that it requires all the scene geometry to be stored in memory while the image is being rendered. On the other hand rasterization, discards objects that are not visible on the screen and also triangles that face away from the camera. In ray-tracing, even if an object is not visible in the scene, it might cast shadows on objects visible by the camera, so it needs to be kept in memory until the last pixel in the image is processed. With production scenes containing millions of triangles, the amount of memory that is required to store the data can become a problem.

So we can understand that ray-tracing is useful to produce a more realistic image but at the expense of being slower and using more memory than a program based on the rasterization algorithm. On the other hand, rasterization based algorithms despite their very fast construction stage have their problems too. The major problem of rasterization lies in its difficulty to incorporate global effects efficiently which has a direct effect on the resulting quality. Both rasterization and ray tracing will be discussed further in the next chapter.

So far, the conflicting nature between ray tracing and rasterization methods has resulted in the proposal of numerous techniques that improve either the interactivity in ray tracing, the quality in rasterization, or both, by combining the two approaches. In this thesis a hybrid implantation is introduced, using ray tracing offline to calculate the ambient light of the objects and storing it in a light map to use it afterwards in the rasterization pipeline.

1.3 Thesis Structure

This thesis consists of four chapters: Chapter 2 will include all the basic information that is needed to understand this thesis. From basic maths to rendering and light theory. Chapter 3 will introduce our algorithm, light map generation using ray tracing. At last in chapter 4 the above method will be evaluated through a series of rendered scenes.

2 Background Work

In order to properly visualize and understand the concepts of this thesis, it is necessary to explain and to formalize some of the basics of computer graphics theory and terminology. In this chapter we provide every useful information regarding the understanding of the thesis. Every aspect of mathematical knowledge such as spherical coordinates, planes, vectors, computer graphics theory such as the rendering pipeline, the rasterization of the geometry, the different coordinate systems and computer graphics algorithms that were used and we deem necessary to understand this thesis, will be covered.

2.1 Geometry representation

It is important to clarify how the world is represented in graphics and how the scene we visualize comes into existence. In general, the mathematical world we display in graphics consists of entities describing the geometry of the surface of 3D objects, the volume of the space inside and outside the surfaces, the energy (light) that is transmitted, the materials (substance qualities) the energy interacts with, the spatial relationships of entities and finally the dynamics of all the above (motion) [GP15c]. Usually we are interested in displaying surfaces of 3D objects, which essentially is the interface between one medium and another and where most lighting calculations and events happen. These surfaces form geometric shapes which are placed in the artificial world with a set of variables that define their

position, orientation and scale. The position is defined by a set of coordinates relative to a coordinate system that encompasses the whole 3D world and its origin (0, 0, 0) is usually in the center of the world. Geometric shapes or else geometric objects are divided into smaller portions in the form of polygons so that they can be processed more efficiently. Such polygons are called primitives and usually are triangles, but they can also be quads or other convex shapes. On each primitive we can define a useful set of variables that represent surface properties, such as normal vectors which are used for shading and a material specification. Materials are describing the object's surface to simulate light's behavior when it reaches the surface, hence providing a more realistic visual effect.

Most modern graphics engines and hardware process and organize primitives in the form of triangles and are optimized to handle this type. The triangle is an efficient shape which is always convex and is very convenient and easy to use. From this point on in this thesis the geometric objects will be represented as a set of triangle meshes.

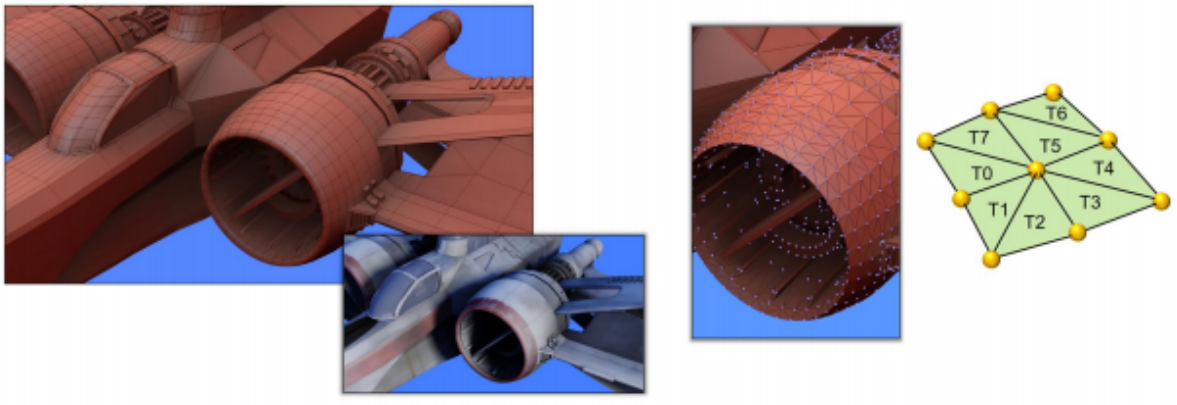

Figure 2.1: From left to right: Modelling of three-dimensional surfaces as polygonal meshes facilitates the efficient image generation of the shapes. Polygonal meshes often consist of triangles, provided as a connected network of vertices.

2.2 Scene representation

The scene that is visualized and simulated with the graphics engine should have a reference point so that the objects can be placed in a defined point inside the scene which at the same time is controllable by the programmer and can be changed at any point either manually or automatically. Defining a global coordinate system with a specifically defined origin point and placing a geometric object to a point relative to the origin is a possible solution, but the necessary lighting calculations require a much more complex solution. In this section we'll discuss the

varying and necessary coordinate systems used in computer graphics to display a scene when working with three dimensions.

Each object in the scene has its own local coordinate system (LCS) with its origin point usually in its center of mass. However, the object's position is expressed in a manner that pays no regard to other objects possibly located in the scene or a global reference point. To better understand the problem, let's assume multiple objects in a scene. These objects will be loaded in an unknown way and will likely overlap with each other. Using this coordinate system the object can be translated (moved around), scaled and rotated, but when applying such transformations, the need for a global reference frame seems clear.

In most 3D applications, each different type of coordinate system is defined with respect to a master coordinate system called the world coordinate system (WCS). It defines the origin and the main x-, y- and z-axes from which all other coordinate systems are defined. The world coordinate system is maybe the most important of all the distinct coordinate systems in the rendering pipeline. It represents a global (or "world"-wide) reference frame for all the geometry present in the scene/world. Hence, when certain transformations are applied to an object, we can pinpoint exactly where the object will be positioned in the scene and its position relative to the rest of the objects, if any.

Since a 3D application's purpose is to simulate the real world, we must take into account that in the real world, what a person sees depends on where they are standing and the direction in which they are looking. That is, we can't make a picture of the scene until we know the position of the viewer, where the viewer is looking and even how the viewer's head is tilted [DE21]. In order to accomplish such an effect, the viewer is attached to their own individual coordinate system, which is known as the eye coordinate system (ECS). In this coordinate system, the viewer is at the origin, (0,0,0), looking in the direction of the negative z-axis; the positive direction of the y-axis is pointing straight up; and the x-axis is pointing to the right. This is a viewer-centric coordinate system and an "egocentric" representation of the world.

However, the viewer can't see the entire 3D world, only the part that fits into the viewport, which is the rectangular region of the screen or other display device where the image will be drawn. We say that the scene is "clipped" by the edges of the viewport. Furthermore, the viewer can see only a limited range of z-values in the eye coordinate system. Points with larger or smaller z-values are clipped away and are not rendered into the image. This is of course not the way that viewing works in

the real world. The volume of space that is actually rendered into the image is called the view volume. Things inside the view volume make it into the image; things that are not in the view volume are clipped and cannot be seen. For purposes of drawing, OpenGL applies a coordinate transform that maps the view volume onto a cube. The cube is centered at the origin and extends from -1 to 1 in the x-direction, in the y-direction, and in the z-direction. The coordinate system on this cube is referred to as clip coordinates.

In the end, when things are actually drawn, there are device coordinates, the 2D coordinate system in which the actual drawing takes place on a physical display device such as the computer screen. Ordinarily, in device coordinates, the pixel is the unit of measure. The drawing region is a rectangle of pixels. This is the rectangle that is called the viewport. A process called “viewport transformation” takes x and y from the clip coordinates and scales them to fit the viewport.

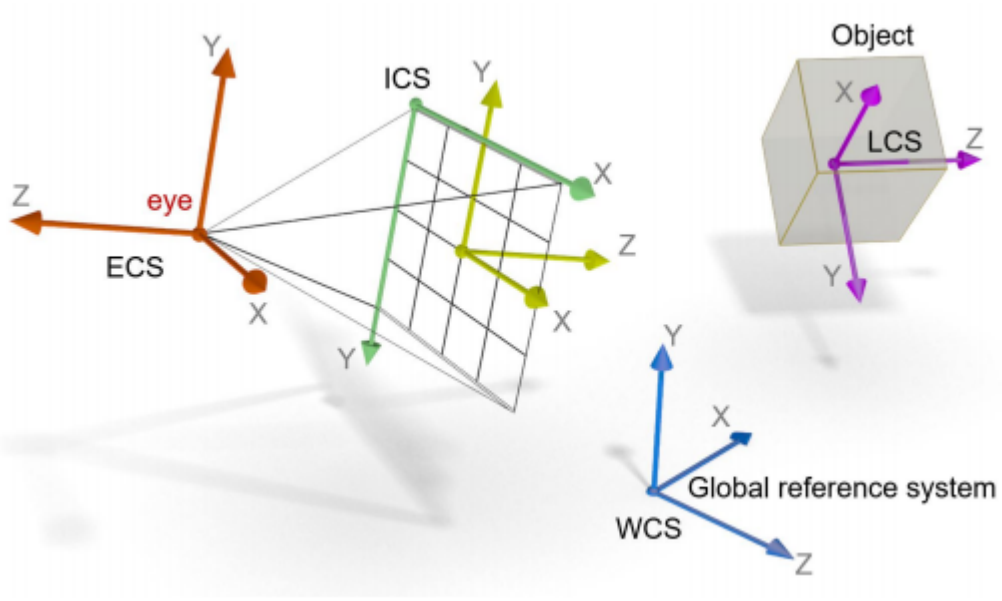

Figure 2.2: The various common coordinate systems that are encountered when working in three dimensions: Local or object space (LCS), the global or world reference frame (WCS), the ego-centric or eye coordinate system of the observer (ECS) and the image space coordinates that the content is projected on (ICS).

2.2.1 Projections

A 3D image can show only a part of the infinite 3D world. The view volume is the part of the world that is visible in the image. The view volume is determined by a combination of information such as where the viewer is located, what direction the viewer is facing, how much of the world the viewer can see and the shape and extent of the region that is in view. Think of the viewer as a camera, with a big

invisible box attached to the front of the camera that encloses the part of the world that that camera has in view. The inside of the box is the view volume. As the camera moves around in the world, the box moves with it, and the view volume changes. There are two general types of projection, perspective projection and orthographic projection [GP15d].

Perspective projection is more physically realistic. It shows a view that you could get by taking a picture of a 3D world with an ordinary camera. In a perspective view, the apparent size of an object depends on how far it is away from the viewer. Only things that are in front of the viewer can be seen. Ignoring clipping in the z-direction for the moment, the part of the world that is in view is an infinite pyramid, with the viewer at the apex of the pyramid, and with the sides of the pyramid passing through the sides of the viewport rectangle. Since an infinite amount of storage and memory does not exist, only objects in a certain range of distances from the viewer can be part of the image. That range of distances is specified by two values, near and far. For a perspective transformation, both of these values must be positive numbers, and far must be greater than near. Anything that is closer to the viewer than the near distance or farther away than the far distance is discarded and does not appear in the rendered image. The volume of space that is represented in the image is thus a "truncated pyramid." This pyramid is named frustum and is the view volume for a perspective projection. The view volume is bounded by six planes, the four sides plus the top and bottom of the truncated pyramid. These planes are called clipping planes because anything that lies on the wrong side of each plane is clipped away.

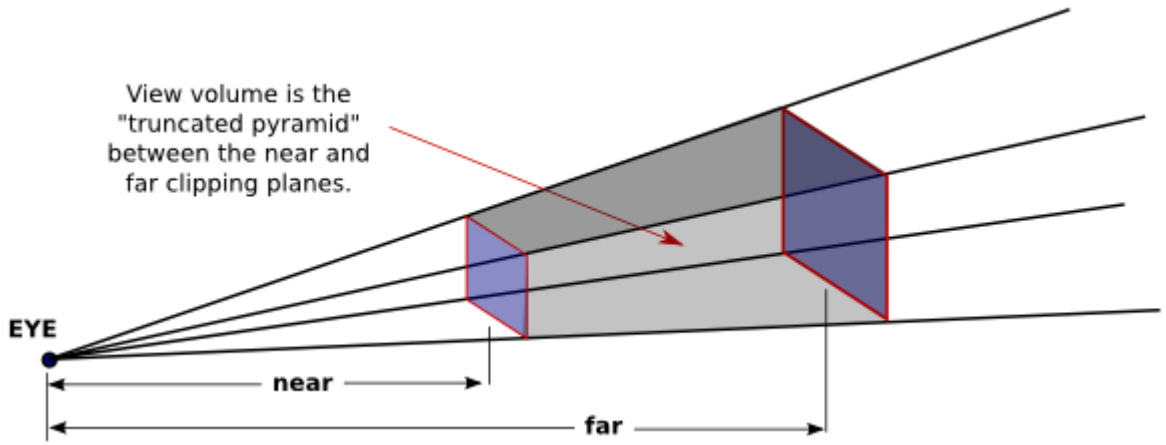

Figure 2.3: Perspective projection visualised. The view volume is bounded by six planes; the four sides plus the top and bottom of the truncated pyramid. [DE21]

In regards to orthographic projection, it is easier to understand: In an orthographic projection, the 3D world is projected onto a 2D image by discarding the *z*-coordinate of the eye-coordinate system. This type of projection is unrealistic in that it is not what a viewer would see. For example, the apparent size of an object does not depend on its distance from the viewer. Objects in back of the viewer as well as in front of the viewer can be visible in the image. In fact, it's not really clear what it means to say that there is a viewer in the case of orthographic projection. Nevertheless, it is considered to be a viewer. The viewer is located at the eye-coordinate origin, facing in the direction of the negative z-axis. Theoretically, a rectangular corridor extending infinitely in both directions, in front of the viewer and in back, would be in view. However, as with perspective projection, only a finite segment of this infinite corridor can actually be drawn. This finite view volume is a parallelepiped that is cut out of the infinite corridor by a near clipping plane and a far clipping plane. The value of far must be greater than near, but for an orthographic projection, the value of near is allowed to be negative, putting the "near" clipping plane behind the viewer.

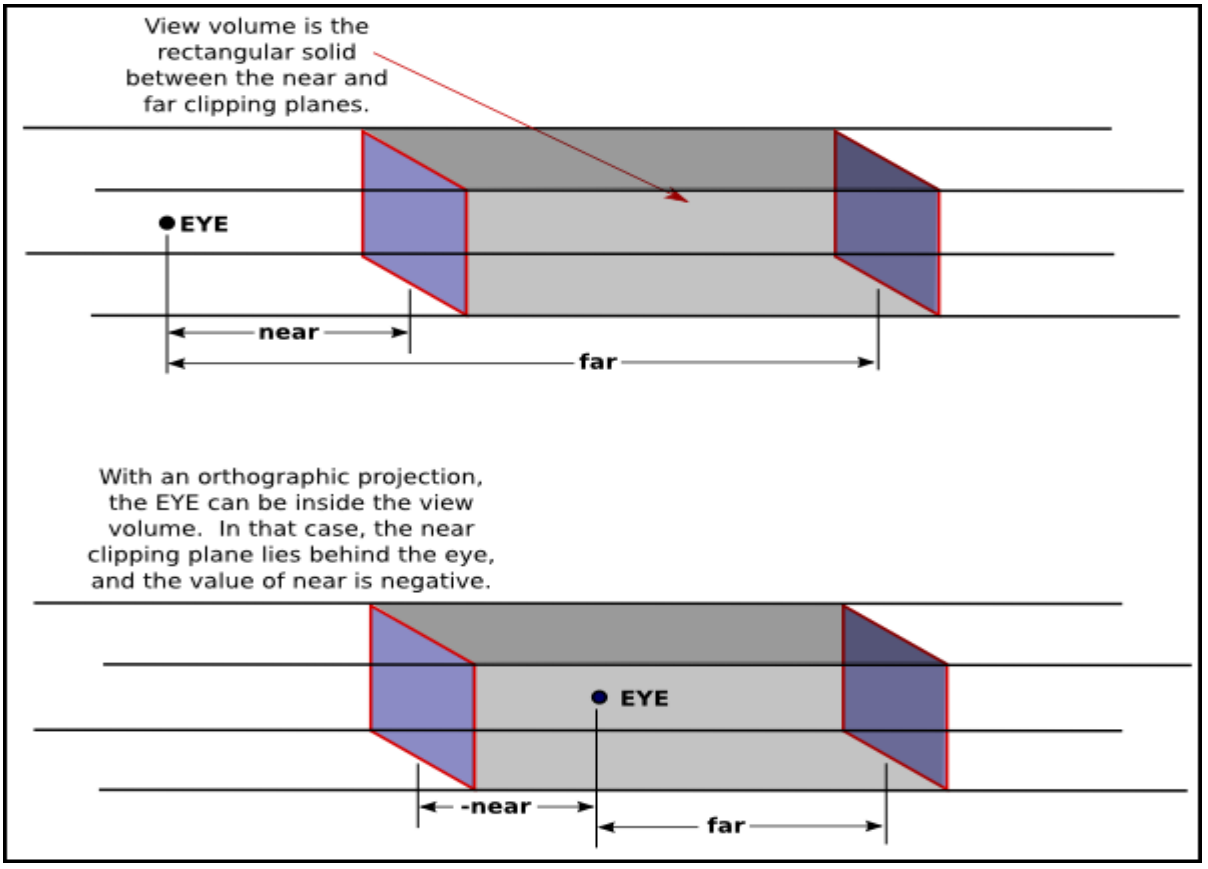

Figure 2.4: Orthographic projection visualised. The viewer is located at the eye-coordinate origin, facing in the direction of the negative z-axis.[DE21]

2.3 Rendering

In computer graphics, a computer graphics pipeline, rendering pipeline or simply graphics pipeline, is a conceptual model that describes what steps a graphics system needs to perform to project a 3D scene to a 2D screen. Instead of the term projecting, the term rendering is usually used.

The concept of rendering is abstract and the rendering process is often implemented via varying methods which give form to the process itself and as a result, they produce the final image. Two implementations we discuss in this thesis are rasterization and ray-tracing, but first we have to understand the concept of rendering.

Rendering is the sequence of steps that we use to create the final image which represents the virtual world either in 2D or in 3D [GP15e]. In other words it is the image synthesis procedure. It is the last stage of the graphics pipeline, hence sometimes we'll use the terms ray-tracing pipeline and rasterization pipeline.

A rendering method can have certain properties that define the resulting image. An important property for a rendering method is the photo-realistic accuracy it can achieve. According to this property, the goal is that the result should not be distinguishable from real photographs which can be done via the simulation of real world lighting phenomena. The counterpart of this property is the non photo-realistic method which is a valid approach where partially explored light simulation is acceptable.

Another commonly used property to describe a rendering method is interactivity. The time needed for an application to produce and visualize a single frame distinguishes a rendering method into two categories; offline and real-time. Rendering methods that fall in the offline category have no time constraints with respect to the target image they are processing in each frame. Usually such methods and systems are associated with algorithms that process images in order to output a high-quality result for a long sequence of consecutive frames. On the other side, real-time algorithms process individual frames in fractions of less than a second [EI19]. Thus, interactivity is defined as the time needed for the production of a frame and how the parameters of the scene such as cameras, lighting effects and geometry can be changed dynamically.

Rendering methods are often used to solve the visibility problem. Visibility consists of being able to tell which parts of 3D objects are visible to the camera. Some parts of these objects can be hidden because they are either outside the

camera's visible area or hidden by other objects. Below, we present the two rendering methods we used in this thesis; ray-tracing and rasterization.

2.3.1 Ray Tracing

Ray tracing as mentioned before 1.1 is a rendering method like rasterization that works with shooting rays into the scene. Ray tracing is one of the most elegant techniques in computer graphics, many phenomena that are difficult or impossible with other techniques are simple with ray tracing, including shadows, reflections, and refracted light [CPC84]. In general, the process involves the tracing of rays throughout the environment to identify the closest ray-objects intersections, without involving projections or scene approximations. On the downside, ray tracing requires that for every ray the entire environment must be traversed.

The main ray tracing system consists of two parts, one construction stage and one ray traversal stage. At the first one the representation of the virtual world is properly indexed and stored into a data structure in order to be efficiently queried for potential intersections. At the second stage, rays are produced from random places on the scene and followed until they hit something. When the ray intersects with an object, local information is extracted and evaluated at that point.

A common algorithm for the second stage of ray tracing is; for each pixel in the image plane a ray is emitted from the camera to the scene. If this ray intersects with objects multiple times, then the closest “hit” to the camera is marked visible. From each closest hit two things have to happen. First, a second ray is generated from the point towards the light source, if it can navigate there without an intersection then

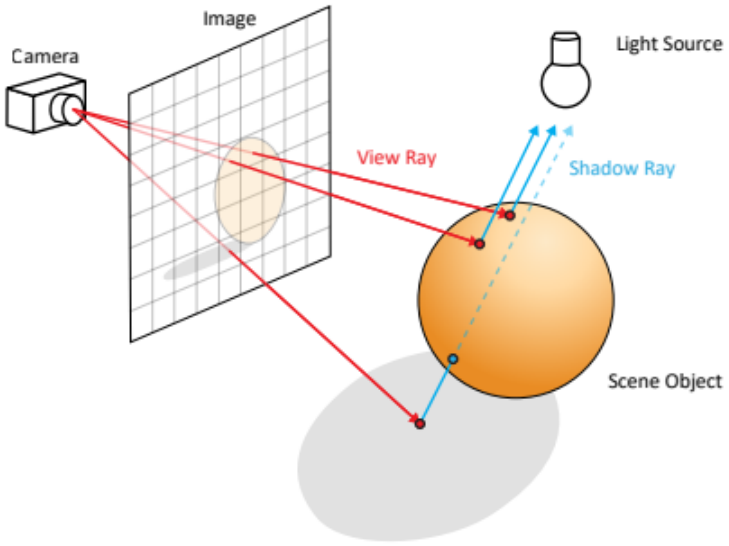

Figure 2.5: Illustrative example of an image-driven ray tracing pipeline. Rays are generated from the virtual camera, crossing the image plane, and traced within the environment. The final color of each pixel is decided based on the intersections of these rays with the scene objects. [KV16]

this point contributes with light, else this point is in shadow. Secondly, the outgoing illumination to the camera has to be calculated, accordingly with the object's material properties. The whole process is visible in Figure 2.5.

This algorithm is standard ray tracing but in this thesis we are using another version that is simpler, path tracing. Fundamentally, the algorithm is integrating over all the illumination arriving to a single point on the surface of an object. With path tracing we can simulate many effects that are useful like ambient occlusion 2.6.

Both processes though, especially when the scene consists of many triangles and objects, can become problematic because both traverse all the objects on the scene for each ray. Though, this process can be accelerated with acceleration structures which will be discussed later 2.3.3.

2.3.2 Rasterization

Computer graphics images are often defined in terms of points on a Cartesian plane which are connected by lines and curves to form polygons and other shapes. Rasterization is the task of taking an image described in such a format and converting it into a raster image, which is a series of pixels, dots or lines, which, when displayed together, create the image which was represented via shapes [MW95]. The rasterization rendering technique is surely the most commonly used technique to render images of 3D scenes and is also the technique used by GPUs to

produce 3D graphics. From section 2.3.1 we can conclude that ray-tracing is the first possible approach to solve the visibility problem. That method is image centric because we shoot rays from the camera into the scene.

Rasterization takes the opposite approach. To solve for visibility, it projects triangles onto the screen meaning it goes from a 3D representation to a 2D representation of that triangle, using perspective projection. This can easily be done by projecting the vertices making up the triangle onto the screen (using perspective projection). The next step in the algorithm is to use some technique to fill up all the pixels of the image that are covered by that 2D triangle.

The rasterization algorithm can be summed up in two loops. The outer loop iterates over all the triangles in the scene. Then, the inner loop iterates over all pixels in the images and finds out if the current pixel is "contained" within the projected image of the current triangle.

This algorithm is object centric because we actually start from the geometry and walk our way back to the image as opposed to the approach used in ray tracing where we started from the image and walked our way back into the scene [HW14].

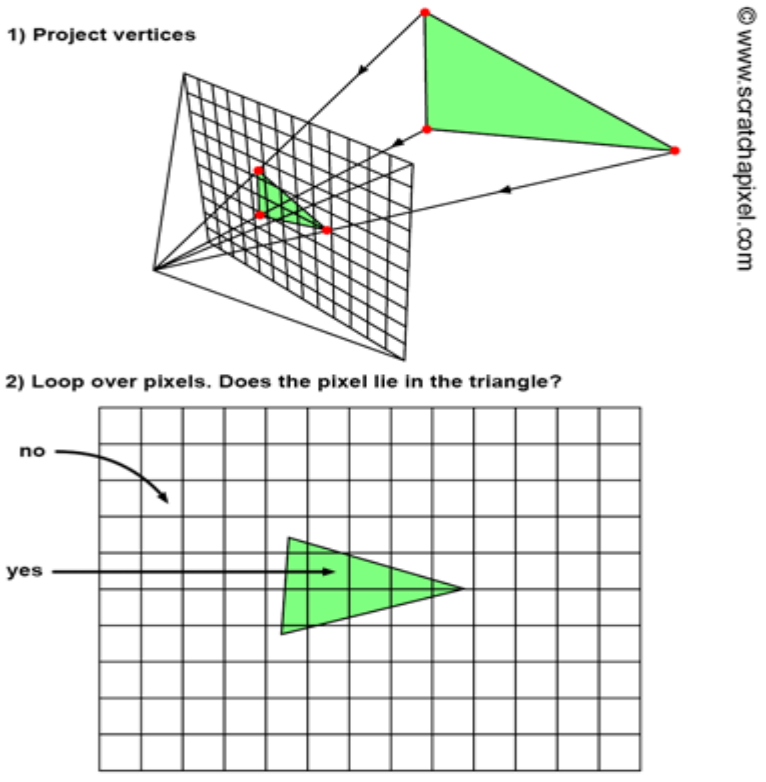

Figure 2.6: rasterization can be roughly decomposed in two steps. We first project the 3D vertices making up triangles onto the screen using perspective projection. Then, we loop over all pixels in the image and test whether they lie within the resulting 2D triangles. If they do, we fill the pixel with the triangle's color.

2.3.3 Accelerated Ray Tracing

It has been established so far that the ray-tracing process is slow and expensive but at the same time produces great results visually. The number of primitives in the scene is what makes ray tracing slow and the fact that for every cast ray, all primitives are checked for an intersection point between them and the ray. A possible solution to make the ray tracing process faster is to reduce the number of primitives each ray will perform the intersection test with. This is done by splitting the primitives into separate clusters and performing intersection tests with the coordinates of the cluster rather than the primitives themselves. The data structures that contain and implement this type of clusters are called acceleration data structures. The principle of acceleration structures is to help decide as quickly as possible which objects from the scene a particular ray is likely to intersect and reject a large group of objects which for certain the ray will never hit.

Spatial Division Hierarchies

Another intuitive and common way of optimizing the ray tracing process and reducing the number of intersection tests is subdividing the 3D space into subregions and checking if they contain geometry that should be ray traced. Two of the methods are the Grid and the use of trees such as kd trees and the octree. Rather than checking rays against all objects in the scene these methods are used to determine if the region through which the ray is passing, is occupied by objects.

Octree

In order to speed up the intersection calculation, academics have implemented divide-and-conquer strategies such as octrees [SS92]. As the name suggests, they form a tree hierarchy where each parent node is succeeded by four or eight children respectively. Trees of that type are usually constructed in a top down manner, starting from the root which holds the bounding box of the scene and recursively partitions space in uniform subregions in every major axis. First the bounding volumes of all the objects in the scene are computed and as a result, the overall scene bounding volume has been computed, which is the result of these volumes combined. The size of the octree can be defined from that scene's overall

volume which is a cube located at the center of the overall volume, whose dimension is the maximum value of any of the volume's extent along the x- y- and z-axis. This cube represents the top node of the octree, its root.

The node insertion routine performs a top-down traversal. If the node in which we want to insert an object is a leaf, and that leaf doesn't contain any object yet, then the object is inserted in this node. If the node already contains an object then the current node is split into eight cells and the object contained by the node as well as the new object are inserted in the cells they overlap. This routine ends either when there's no more objects and primitives to insert or when a user defined maximum depth is reached. In a second step, the octree is traversed in a bottom-up fashion this time. Starting from the leaves, we compute the overall bounding box of the objects they contain. The overall volume of the node above the leaves is then computed from combining its children bounding volumes. This process is repeated until the root is reached. At the end of these two processes, we obtain a representation of the scene as a hierarchy of bounding volumes, where the volumes are grouped based on proximity.

The intersection of the hierarchy is quite intuitive. Starting from the root node, the bounding volume for this node is tested if it's intersected by the ray. Generally, if the bounding volume of a node is intersected by the ray, we move one level down in the tree and test for an intersection with the bounding volumes of this node's children. If the node is a leaf, then ray intersection tests are performed with any of the objects it contains.

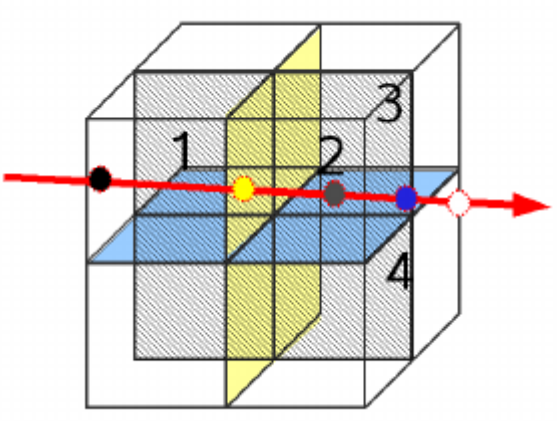

Figure 2.7. Single ray traversing an octree node. The traversal algorithm finds the order of intersection of the X (yellow), Y (blue), and Z (gray) mid-planes. [KAW06]

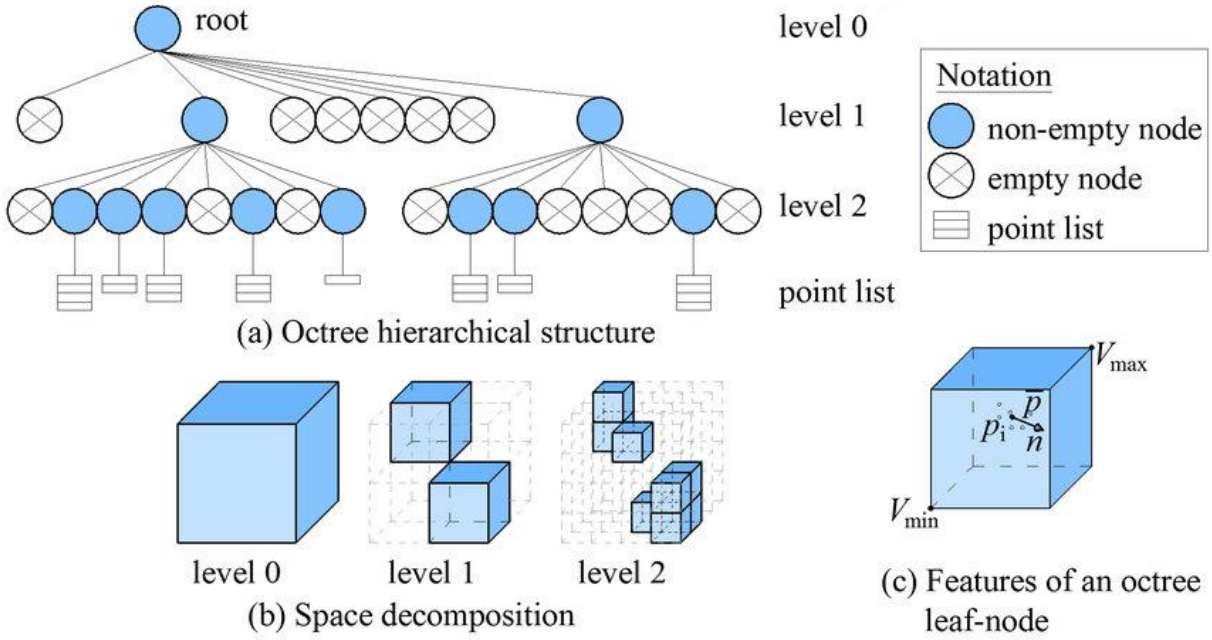

Figure 2.8. Example of octree decomposition [VTLB15]

Grid

The region of space containing the object is subdivided into a regular 3D grid. This method was popularised by Fujimoto et al [FI85]. Triangles of the models are then inserted in the grid's cells they overlap. Some cells will be empty but some others will contain a subset of the model's geometry. This simple but quite efficient idea requires several considerations to be taken into account in order to produce the desired performance. The main issue is related to the cell size itself. Naively setting the grid size might result in the resolution and granularity being too fine or too coarse or even both at the same time in different locations of the scene, with respect to the given geometry.

Testing if a ray intersects the model's geometry is simple. The grid is traversed cell by cell following the ray's direction. If the current cell is occupied by some geometry then the ray intersection tests are performed for this part of the geometry. If the ray intersects any part of the geometry, the traversal can be stopped, otherwise the next cell pierced by the ray is brought onto the surface for testing. This process is repeated until the ray hits an object or until the ray leaves

the grid.

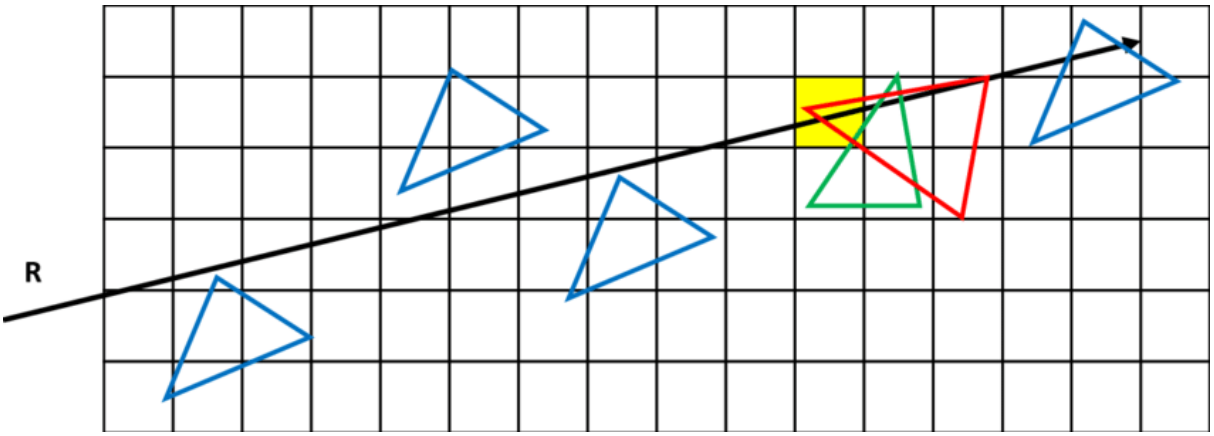

Figure 2.9: Dealing with the intersection in uniform grid structure during ray traversal process. The blue triangles are the triangles that are checked but they're not intersecting with the ray. The green and red triangles are intersecting with the ray but the green one is intersecting outside the yellow box. [RH11]

In regards to the creation of the grid, the first step is to compute an overall bounding box of the grid which is the same as the scene bounding box. An efficient way to do it is to loop over all the objects of the scene and merge their bounding boxes. This way the size of the grid is determined, the next task consists of choosing its resolution (the number of cells in each dimension). Unfortunately there is no formula to compute a resolution which could be proven to be optimum for all the scenes or even a given scene. Some metrics have been developed but most of the time, still relies on a user defined parameter to fine tweak the performance of the scheme. The method used to compute the grid resolution is most of the time based on a metric proposed by Cleary et al [CJ83]. This metric tries to establish some relation between the dimension of the scene, the number of primitives it contains and the overall volume of the scene. If the resolution is too high, the time spent traversing the scene outweighs the benefit of using the acceleration structure. On the other hand if the resolution is too low the cell contains many triangles and the time spent intersecting each triangle contained in the cells doesn't make the grid a much better method than simpler acceleration methods.

Bounding Volume

The simplest method of accelerating ray tracing is a bounding volume. A bounding volume is a volume that contains a certain number of primitives. and it is usually the tightest possible volume; a box in many cases but it could also be a

sphere surrounding the object. The first test is if the ray intersects this volume, and if it doesn't, we know for sure that it can't intersect the geometry contained in this bounding volume. The triangles from this bounding volume can safely be ignored, which saves many ray-triangle intersection tests. Considering that many of the bounding volumes from the scene are likely to fail this test as well, overall, a significant number of calls to the ray-triangle routine can be omitted. If the ray intersects the bounding volume though, all the triangles contained in the volume will have to be tested for an intersection with this ray.

Bounding Volume Hierarchy

An improvement of the previous technique can be done by grouping the bounding volumes into a hierarchy of bounding volumes. This technique is called a Bounding Volume Hierarchy or BVH. Bounding volume hierarchies (BVHs) in general are indispensable tools in computer graphics. They are used to accelerate a multitude of algorithms, including collision detection, frustum culling, and ray tracing [SMSFD20]. Each object has its own bounding volume and by merging them, a larger bounding volume is obtained which represents a larger group of objects. Thus, the testing of these volumes enclosed by these groups can be avoided, possibly rejecting many objects at once. The objects and the volumes have to be grouped by proximity so that an unnecessarily big volume is avoided, meaning it's preferable to create smaller volumes with less primitives and then group them together.

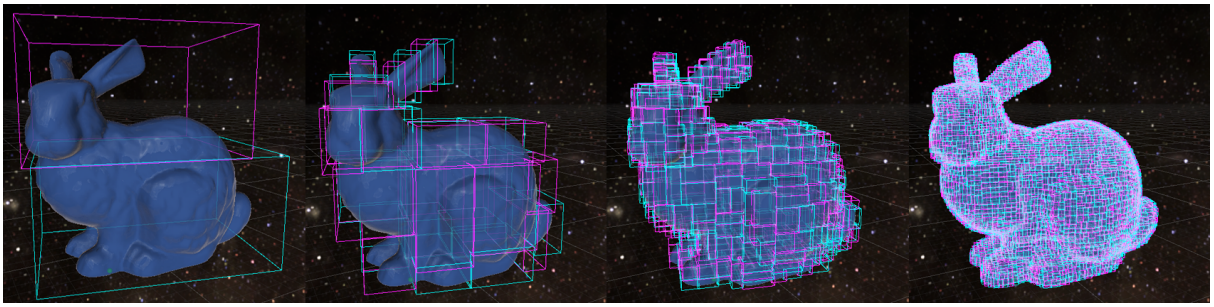

Figure 2.10: BVH visualization of a Stanford Bunny. Cyan boxes are left children, magenta boxes are right children. Images show how a BVH adjusts step by step to the mesh with several depth layers. From left to right: 1, 5, 9, 13. [RW18]

2.4 Light Transport Theory

In 3D scenes light is emitted at the light source and scattered around the environment in a practically infinite number of directions and scattering events. So it is required to have a deep understanding of the physical laws in order to implement an illumination algorithm.

In computer graphics, an important distinction exists between **local** and **global** illumination algorithms based on the number of light bounces in a virtual environment [KV16]. On Local illumination, known also as object oriented lighting, the algorithms take into account only the first bounce of light, so the surfaces are directly lit from the light source. These algorithms are fast to compute but with not such a realistic outcome. The two prevailing models are Phong and Blinn-Phong illumination models.

The Phong model considers that the color of the pixel is the sum of the above explained terms ambient, specular and diffuse lighting [BTP75]. Thus this model takes into account the view angle so that it can calculate the specular lighting and highlight the object for a more realistic view. This shinines is based upon the roughness of the surface.

The second model Blinn-Phong is similar to the Phong model because it uses specular lighting to highlight the object [BL77]. There is one difference between them, in Phong the model constantly calculates the dot product between the viewer and the reflected ray, on the other hand in Blinn-Phong it calculates the dot product of the normal and a halfway vector between the viewer and the light source.

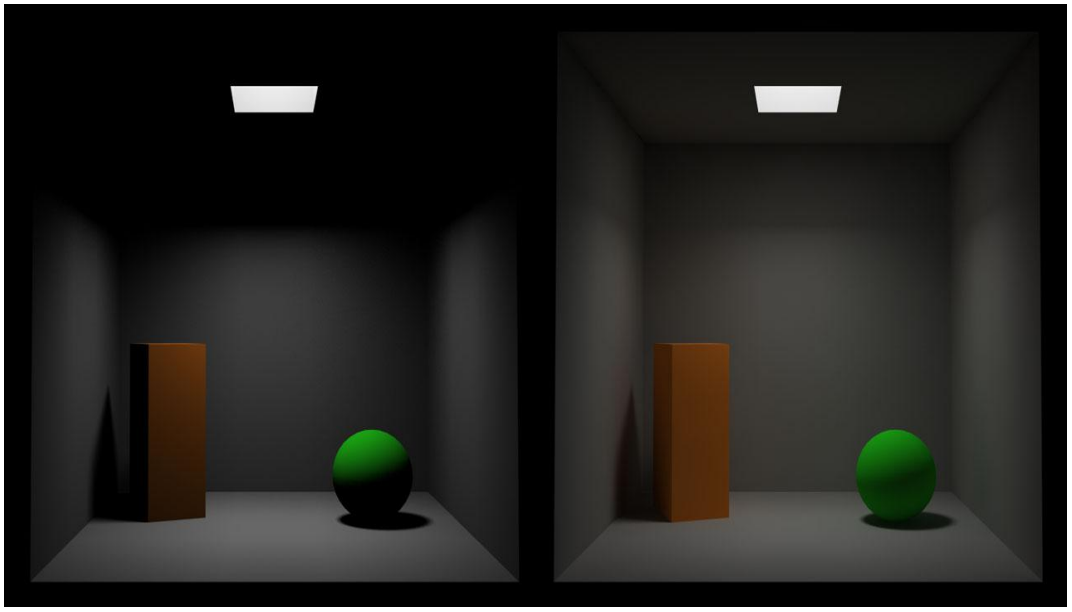

Figure 2.11: a scene without GI (on the left) and the same scene rendered with GI (on the right) [RW21]

Global illumination varies from local illumination on their approach, it takes into account not only light that comes directly from a light source but all the light that travels throughout the scene. It computes all bounces of light, surfaces are lit both directly from the light sources and indirectly through the scattering and transmission of photons within the environment [KV16]. The outcome of global illumination algorithms often appear more realistic. The latter class of algorithms can be considered as a superset that includes the local illumination methods, while the underlying theory is the same.

2.4.1 Basic Lighting

The theory of light transport describes different phenomena by assuming different models of light representation. These are classified as geometric, wave and quantum optics. Each of these models captures different properties of the dual wave-particle nature of light as it interacts with the environment at different levels of detail.

Computer graphics in general and this thesis use the first model. Geometric optics or ray optics introduces phenomena as reflection, refraction, absorption and emittance; it is considered the simplest model. It assumes that light is emitted as rays, in straight lines at infinite speed. This model has proven efficient since it covers some of the most useful optical phenomena of what we see everyday and is easier to simulate.

There are two basic ways to measure light in a scene: radiometry and photometry. Radiometry is the measurement of optical radiation, which is electromagnetic radiation in the frequency range between 3 x 1011 Hz and 3 x 1016 Hz. On the other hand, photometry is the measurement of light, which is defined as electromagnetic radiation detectable by the human eye. The difference between radiometry and photometry is that radiometry includes the entire optical radiation spectrum, while photometry deals with the visible spectrum weighted by the response of the eye [OY00]. All quantities in global illumination calculations are in radiometric units by convention.

2.4.2 Radiometry

Radiometric quantities are used in GI (global illumination) to measure how the photons propagate and interact with the scene.

Radiant Energy (Q): Is the energy emitted from the light sources or reflected from a surface and is transferred through space as photons.

Radiant Flux (Φ): Is the energy flowing through a surface per unit time,

$$\Phi = dQ/dt$$

Irradiance (E): Is the energy flow towards a surface from all incoming directions,

$$E = d\Phi/dA$$

where A is the surface area.

Radiosity (B): is the radiant exitance of a surface and is the same radiometric quantity as irradiance but measured in the outgoing energy flow direction from a surface,

$$B = d\Phi/dA$$

Radiant Intensity (I): is radiant flux per unit solid angle,

$$I = d\Phi/d\omega$$

Radiance (L): is radiant flux per unit area and per unit solid angle,

$$L = d^2\Phi/(dA_\Phi \cdot d\omega) = d^2\Phi/(dA \cdot d\omega \cdot cos\theta)$$

where θ is the angle between the surface normal and the specified direction and A_Φ is the projected area (the area that is perpendicular to the light direction). The amount of the projected area relative to the surface area is given by: $A_\Phi = A \cdot cos\theta$.

All the above quantities compute the radiation for all frequencies of the electromagnetic spectrum. The equivalent quantities that are defined per wavelength are spectral radiant power, spectral irradiance, spectral radiosity, spectral radiant intensity and spectral radiance.

The radiometric quantities described in this section, as well as their units, are summarized in Table 2.1.

Quantity	Symbol	Unit	Definition
Radiant Energy	Q	Joules (J)	Energy of electromagnetic radiation
Radiant Flux	Φ	Watts (J/s)	Electromagnetic energy per unit time
Irradiance	E	W/m^2	Incident radiant flux per unit area
Radiosity	B	W/m^2	Exitant radiant flux per unit area
Radiant intensity	I	W/sr	Radiant flux per unit solid angle
Radiance	L	W/m^2sr	Radiant flux per unit area per unit solid angle

Table 2.1: A list o radiometric quantities

2.4.3 Rendering equation

In global illumination light emitted from the light sources bounces around the scene and undergoes alteration through scattering, absorption, reflection etc. The objective is to estimate as accurately as possible this photon multiplication on the scene. The most fundamental mathematical model in computer graphics was presented by Kajiya [JK86], who formulated light transport in computer graphics in the form of an integral equation, called the rendering equation.

The rendering equation has a physical basis on the law of conservation of energy, which states that the total energy on an isolated system remains constant. To depict this system of energy we use the term of radiance. Radiance is a fundamental quantity in computer graphics and especially in lighting calculations because it represents the amount of light traveling on rays [KV16].

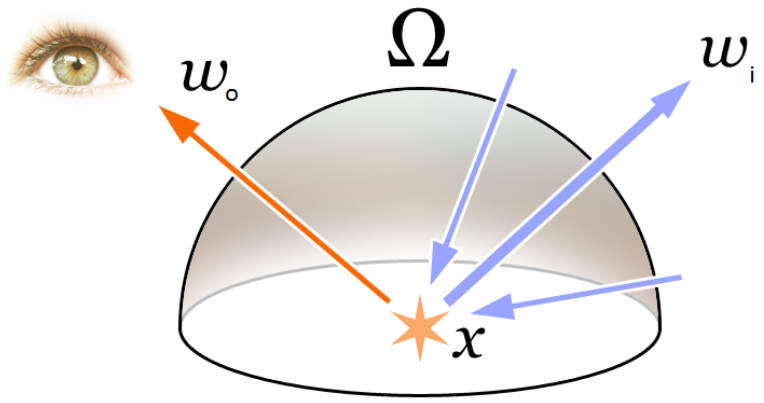

Figure 2.12: The rendering equation describes the total amount of light emitted from a point **x** along a particular viewing direction, given a function for incoming light and a BRDF. [WIKa]

Taking into account the irradiance from all incident directions over the hemisphere above the surface point, the reflected radiance is [GP15a]:

$$L_0(x, \omega_0, \lambda, t) = L_e(x, \omega_0, \lambda, t) + \int_\Omega f_r(x, \omega_i, \omega_0, \lambda, t) L_i(x, \omega_i, \lambda, t)(\omega_i \cdot n) d\omega_i$$

which states that the radiance leaving a surface point x in direction ω_0 is equal to the sum of the emitted radiance L_e from that point in the ω_0 direction and the radiance L_r leaving the surface in the same direction due to light scattering events at x.

BRDF

At last $f_r(x, \omega_i, \omega_0, \lambda, t)$is the BRDF which means Bidirectional reflectance distribution function. BRDF is a scuttering function and it is used to properly represent the material properties of the objects. The BRDF for a particular point depends on the incident and outgoing light directions and is defined as the ratio of the differential outgoing radiance to the differential irradiance:

$$f(x,\ \omega_0,\ \omega_i)\ = \frac{dL(x, \omega_0)}{dE(x, \omega_i)}$$

where ω_0 is the outgoing direction under consideration and ω_i is the light's incoming direction [KV16] .

The simplest body reflectance BRDF and the one that is still widely used in interactive applications is the ideal diffuse, pure diffuse, or Lambertian BRDF [LM60], which states that a lit surface is viewed equally bright from all directions. In this case, the BRDF is constant, equal to ρ(x)/π. The value ρ(x), is the albedo of the surface at x, i.e., the amount of energy being reflected instead of absorbed.

The simplest surface reflectance BRDF is the ideal specular BRDF where the incoming energy is totally reflected in the ideal reflection direction, similar to a perfect mirror reflection. However, the behavior of a more realistic BRDF is characterized by a reflectance lobe in which the scattering of light covers a range of directions, instead of a single one. Surface reflectance BRDF models are known as glossy BRDFs, with the most historically popular being the Blinn-Phong BRDF [BL77].

Other models are based on microfacet theory, which models the surface as a number of tiny mirror-like or ideal specular planar reflectors. Historically, the most widely used isotropic reflectance model has been the Cook-Torrance model [CT82].

In order to actually simulate light effects, the majority of the rendering algorithms have been designed to capture a subset of phenomena. This is transpired because simulating all light transport effects is a challenging and time consuming task even in current production renderers. By excluding certain phenomena, the rendering equation can be simplified significantly. These different algorithms include: ambient occlusion and environment mapping.

2.5 Textures and Texture Mapping

Texture Mapping is a technique and a concept introduced in computer graphics for providing high visual realism in a scene [TMB05]. It has many distinct variations and Light Mapping is one of them. Texture mapping is "applying" a diverse selection of properties onto a surface in order to give the appearance of surface detail that is not actually present in the geometry of the surface. Properties that can be mapped onto a surface include color, transparency, and light reflectivity. The direction of light reflected off bumpy surfaces is what visually reveals the texture, or bumpiness, of the surface. Essentially, a texture map is a 2D image applied (mapped) to the surface of a shape or polygon. They are usually stored in common image file formats and processed by modern graphics engines to produce the final result.

Texture images, or simply textures, can be one-, two-, or three-dimensional and are essentially buffers of pre-calculated data that the mapping function addresses to acquire material values. Just like a common image, textures consist of pixels which are called in short texels. The image is usually mapped to a normalized domain of the parametric space T^D, where D is the dimensionality of the pattern ($D = 2$ for the case of a conventional two-dimensional image). The continuous normalized parameters are called texture coordinates. The mapping of the three-dimensional coordinates to the texture space produces parameters that are either wrapped or clamped to the $[0, 1]$ range. In the rest of this thesis a texture map will refer to a two-dimensional pattern with corresponding texture parameters $u, v \in [0, 1]$. [] The distinct areas/bits of the texture are called uv islands.

A texel contains RGB values and its local coordinates relative to the origin point of the texture. The origin point $(0, 0)$ is usually considered to be the bottom left corner whereas the top right corner is the point $(1, 1)$.

The (u, v)-coordinate data can be dynamically calculated for each vertex or can be retrieved from a data structure along with the rest of the mesh information. In both cases, unless the texture coordinates have been explicitly assigned by the user, the$u$and$v$parameters have been deduced from the Cartesian coordinates of the vertices with the help of a texture-coordinate generation function.

A texture-coordinate generation function provides a simple mapping from the Cartesian domain to the bounded normalized domain in texture space. Most common functions perform the mapping in two steps. First the arbitrary Cartesian

coordinates are mapped on a predetermined “supplementary” surface embedded in space, whose shape can be represented parametrically. Then this surface’s parameters themselves are normalized to represent the texture coordinates.

The simplest (u, v)-coordinate generation function is planar mapping. Consider illuminating the surface of an object by shooting parallel rays from a video projector that displays the texture map but with the provision that rays pass through the surface and also illuminate the hidden sides. Planar mapping uses a plane as an intermediate parametric surface. The Cartesian coordinates are parallelly projected on the plane, and the parametric representation of the projected points is used as a set of texture-coordinate pairs.

In regards to performance and storing space optimization, there exist several optimizations for texture mapping. The optimization that will be discussed in this thesis is called baking [VG17]. Baking is the term used to describe the action of running processes and expensive calculations offline, meaning not real-time, and storing the results into textures. These textures are then used just like any other texture. They are being processed in a similar way and applied onto surfaces to portray certain details. Baking is most commonly used for lightmaps and it saves significant process power and space due to it being done only once and not per frame.

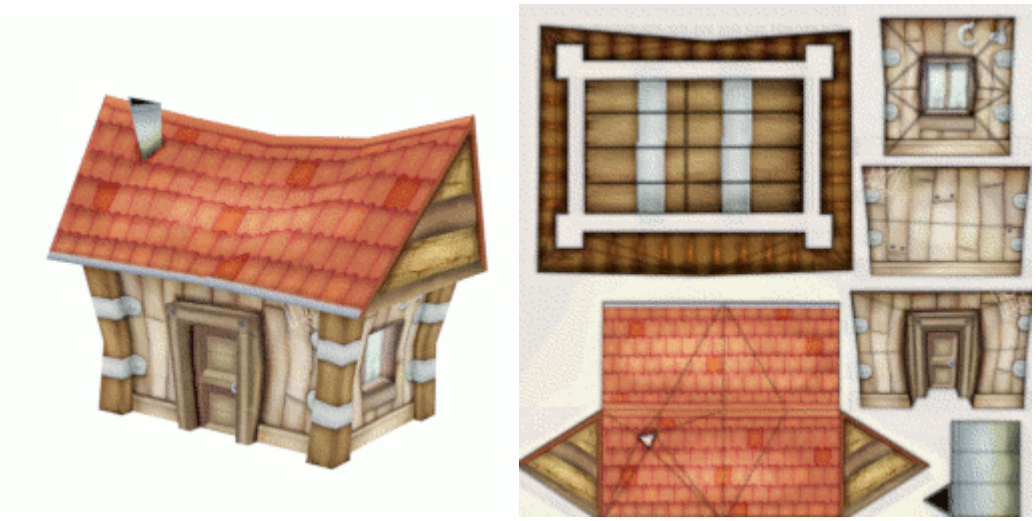

Figure 2.13. Left image is the 3D mode with the texture map applied. Right image is the texture map. UV islands are separated by white space.

2.5.1 Texture Atlases

It is common in practical applications of texture mapping for the need for a bijective and unique mapping of a surface's parts and their corresponding texels in the texture domain to arise. A texture atlas is a surface parameterization where connected parts of the object's surface or otherwise charts, are each mapped onto contiguous regions of the texture domain [GP15f]. Atlases are usually used in situations when one, texel values are dependent on their position on the surface, two, the texture is used as storage for information regarding the geometry's properties and three, there's a constant need for information of the contents of the atlas, from both directions. One of the most frequent uses of texture atlases is in the use of textures to store precomputed lighting effects. These textures are called lightmaps and will be discussed later in detail.

There have been several methods proposed to generate a texture atlas. A model is first split into charts and then each chart is parameterized in a way so that the corresponding part of the object's surface is mapped onto a 2D surface, thus ensuring a unique bijection between the cartesian coordinates of the chart and the parameters of the surface [TPPP07]. To successfully generate an atlas without some of its charts overlapping, a certain packing procedure must be used, to assure the uniqueness of the bijection. There are four criteria that need to be satisfied by each of the packing methods. One, texture distortions and artifacts should be minimized, two, texel distribution over the surface should as even as possible, three, the continuity and conformity of mapping along the charts should if possible be ensured and four, the area coverage of the charts should be maximized and at the same time the number of separate charts should be minimized. This is the third and final step of the generation procedure and it's essentially a variation of the NP-complete "bin packing problem". A popular approach to generate an atlas while conforming to the above four criteria is by cutting the surface into regions, which are called polypacks, and map each one to a plane with as little distortion as possible. A brief overview of the polypacks approach is presented below.

Polypacks

The first step is the surface segmentation. The surface is segmented in such a way that each surface region (polygon cluster or polypack) is mapped to a plane with as little distortion as possible. The easiest way to do this is to cut the surface into areas of connected polygons that face the same primary half-axis and use the

corresponding plane, perpendicular to the primary axis, as the projection plane for the planar mapping [TPPP07]. The problem with this naive segmentation method is that for relatively complex surfaces and models with creases or irregular curved regions, there are too many polypacks produced which are also relatively small. The number of charts in the final atlas should be minimized as much as possible, because first, the unused space in the atlas texture is increased with the number of atlas elements and second, the seams between adjacent textured patches tend to be noticeable as the texture parameterization changes in scaling or direction across the surface. The solution to this is a modified version of the planar mapping method in order to produce fewer polygon clusters and alleviate the problem.

After splitting and parameterizing the surface, which is not assumed to be watertight or manifold in general, each cluster is mapped to the normalized parametric domain $([0, 1],[0, 1])$. In order for the texels of the final atlas texture to be uniformly distributed among the atlas charts and avoid stretching the texture, each chart needs to be assigned a bitmap, whose aspect ratio $r(i)$ matches the aspect ratio of the planar projection of the ith polygon cluster, prior to packing. This atlas element does not need to be a power-of-two sized bitmap [LPRM02]. As the elements will be packed to fill up the atlas texture, we only need to worry about the size of the elements.

Next, the texture packing is performed by recursively partitioning the final atlas texture, into areas that will eventually host the atlas elements. The most common method performs a nonuniform binary partitioning of the image space that results in an unbalanced kd-tree.

It must be mentioned that the atlas generation procedure faces certain issues. First and foremost, as the number of charts increases, so does the unused space, so charts should be tightly packed to ensure some guard space. In order to avoid using texture values from neighboring elements in the final packed atlas texture, the texture coordinates of the polygon clusters need to be pulled toward the inside of the area they occupy in texture space. Let's call "active texels" the texels actually mapped to the polygons after shrinking the texture coordinates. The unused pixels left in the atlas element outside the active texels are flagged as "sand texels" and they are the ones who act as a guard space. Second, Texel area coverage must be as close to uniform as possible to avoid stretching and to ensure proper and proportional scale of charts in packs. [GP15f]

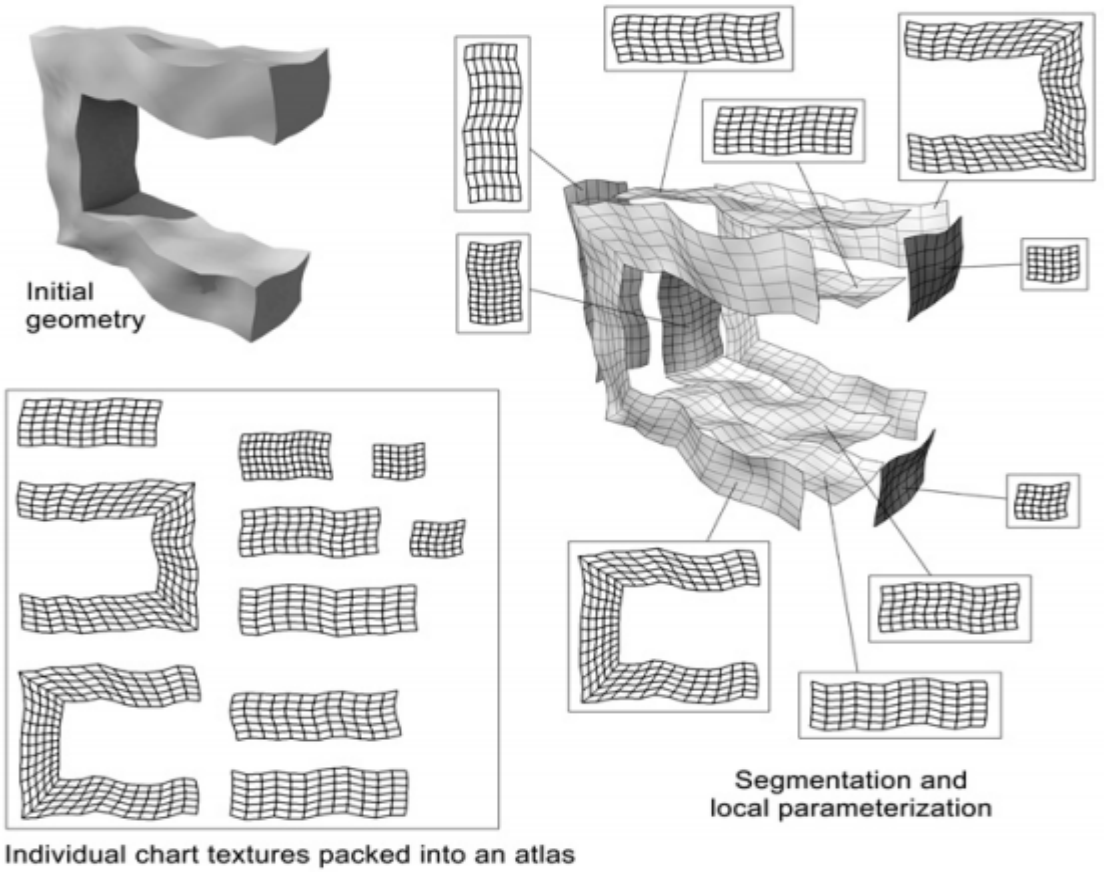

Figure 2.14: The parameterization of a surface as an atlas texture [GP15f]

2.5.2 Light Mapping

The most common use of atlas textures is the process of storing precalculated lighting into a texture. All incident rays of light, regardless of the angle, are saved in the texture. When the model is being drawn onto the screen, instead of using complex calculations to compute lighting and lighting effects, it is possible for this texture to be used, assuming the scene is mostly static and the contribution of its surrounding, possibly dynamically changing, models to its overall illumination is minimal.

To be more specific, light-mapping is the process of pre-calculating the brightness of surfaces in a scene. It stores the information it calculates in a texture or lightmap, as it's usually called, for later use. Lightmaps allow you to add global illumination, shadows, and ambient lighting at a relatively low computational cost.

Lightmaps are composed of lumels, in analogy with texels in Texture Mapping. Smaller lumels produce a higher resolution lightmap, providing finer lighting detail at the price of reduced performance and increased memory usage.

The techniques in which a lightmap can be created vary. A common way is to treat the surrounding environment as one big light source. This is generally accomplished by manipulating a cubemap environment map 2.5.3 such that we can directly use it in our lighting equations: treating each cubemap texel as a light emitter. This way can effectively capture an environment's global lighting and general feel, giving the 3D objects a better sense of belonging in their environment.

2.5.3 Environment Mapping

Environment mapping, also called reflection mapping, is an efficient image-based lighting technique for approximating the appearance of a reflective surface by means of a precomputed texture. The basic idea is, assuming the environment is far away and the object does not reflect itself, the reflection at a point can be solely decided by the reflection vector. Usually this information is discreetly encoded on a set of images and it's convenient to store them in cube maps [GP15b]. Environment illumination is expressed mathematically as:

$$Lr(x, \omega_o) = \int_{\Omega} f(x, \omega_o, \omega_i) L_{map}(\omega_i)(n_x \cdot \widehat{\omega}) d\omega$$

where L_{map} is the environment map.

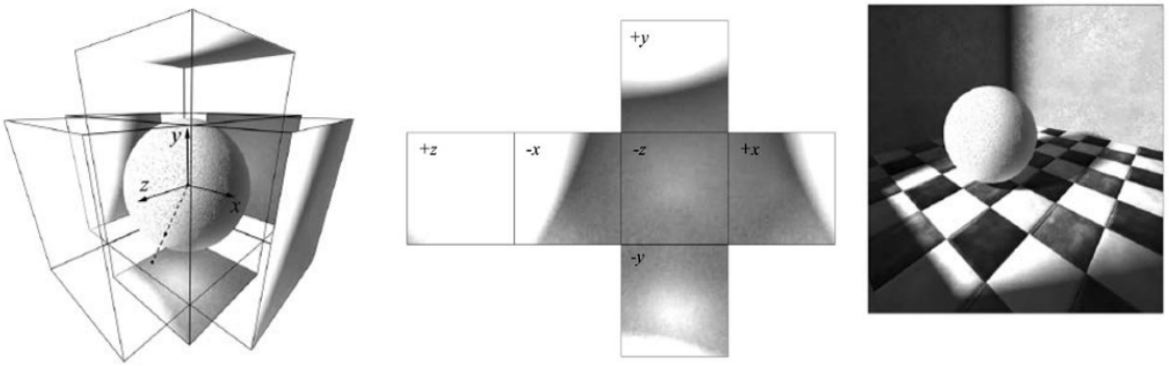

Figure 2.15: A Environment map implemented into a cubemap [GP15b]

The first technique that was used was sphere mapping in which a single texture contains the image of the surroundings as reflected on a spherical mirror but it has been surpassed by cubemaps. It was introduced in 1986 by Ned Green

[GN86], in which the environment is projected onto the six faces of a cube and stored as six square textures as shown in figure 2.15.

Cube maps have many applications in computer graphics, perhaps one of the most advanced is skyboxes. It creates pre-rendered panoramic sky images which are then rendered by the graphical engine as faces of a cube at practically infinite distance with the view point located in the center of the cube. In a lot of scenes skyboxes produce the global illumination, so many times we have to sample the cubemaps to create the lighting of an object.

2.5.3.1 Cube Map Sampling

When a Cube Map is sampled the ray/vector (x, y, z) has to be converted to the face and texture coordinates (u, v). First the algorithm has to find where the ray is pointing so that it can sample the right face of the cube map. It compares the absolute values of (x, y, z) and the sign of the biggest axis. With these known, as shown in the figure 2.16, it can find the u,v accordingly. For example if the biggest absolute value axis is $y$and $y$is negative, according to the figure below $u$is equal to

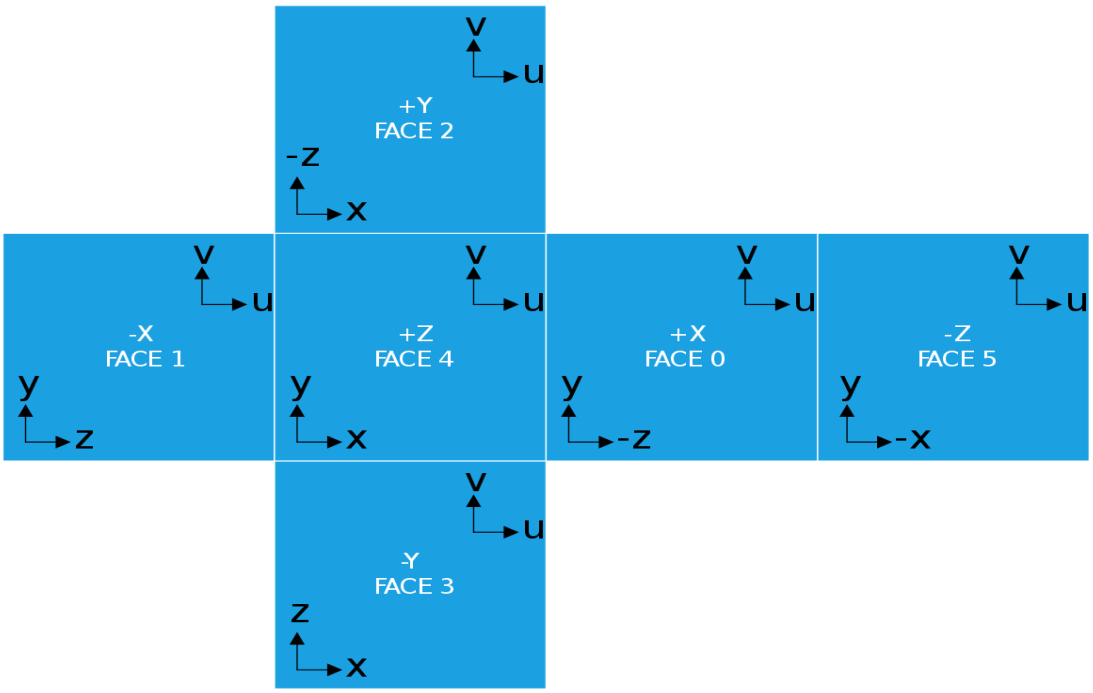

Figure 2.16: This illustration shows how a cube map is indexed and addressed.[WIKb]

x and $v$to z. The last step of the algorithm is to convert the u,v that we found from range (-1,1) to (0,1) so it can correctly sample the texture. This is done using these equations:

$u = 0.5 * (uc/maxAxis + 1)$

$v = 0.5 * (uv/maxAxis + 1)$

where uc and uv are the u,v values in (-1,1) range.

2.6 Monte Carlo Methods

The most popular methods for solving the rendering equation are based on Monte Carlo Integration approaches. The idea is trivially simple: random samples are chosen over the integration domain, the integrand is evaluated for each sample and, finally, the integral is approximated based on the statistical mean of these evaluations.

Monte Carlo Integration

Assume we need to evaluate the integral I of a function f over a domain Ω:

$$I = \int_{\Omega} f(x)dx.$$

The idea of Monte Carlo integration is to approximate I by drawing N samples over the integration domain, where each sample is weighted by a probability density function:

$$\widehat{I} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}$$

where $\widehat{I}$ is the estimator of the target function and x1, ..., xN are independent and identically distributed random variables.

About the probability density function, the monte carlo method is based on probability theory, that associates random outcomes of an experiment with a certain probability. The random variables are divided in two categories: discrete random variables, where the set of possible outcomes is finite, or countably infinite and

continuous random variables where they are defined over a continuous domain. Since we are interested in the rendering equation, we restrict our discussion to continuous random variables. The expected value of a continuous random variable x, or the weighted mean, defined over a domain Ω, is:

$$E[x] = \int_{\Omega} xp(x)dx$$

where p(x) is the probability density function, or PDF, describing the probability of occurrence of each outcome of the random variable.

Monte Carlo approaches suffer from high variance, a major source of variance is due to the choice of the probability density function. The simplest PDF is the uniform probability distribution, where each sample has equal probability of being selected over the integration domain. That could lead to high variance if a function contains high peaks in several regions. The most efficient PDF would be one that would follow the characteristics of f. The technique of reducing the variance based on the choice of an alternative distribution function that matches the form of the integrand is called **importance sampling**. A common practice in illumination algorithms is the distribution of samples based on one or more of the functions comprising the rendering equation, a cosine weighted term, the incident light, or the material properties of the underlying surfaces. But the choice of PDF is very important because with an improper PDF could lead to even higher variance [WJ08].

Ambient Occlusion

Ambient occlusion belongs in the Monte Carlo techniques that trie to simulate how exposed each point in a scene is to ambient light. It sends rays to sample nearby geometry that blocks or occludes light.

AO methods are distinguished between object-space and image-space ones. Object-space methods produce high quality, stable results, but depend on the scene complexity and are typically more expensive to compute. Image-space methods produce reasonably convincing results and offer bounded rendering times as they do not depend on scene complexity. They use information stored in the camera G-buffer, a series of textures containing information (depth, normal, reflectivity, etc.) from a specific viewpoint, as sampled for image generation. These methods can be executed in real-time and support fully dynamic environments, but are prone to

undersampling, view-dependent artifacts and accuracy limited to near-field geometry [KV16].

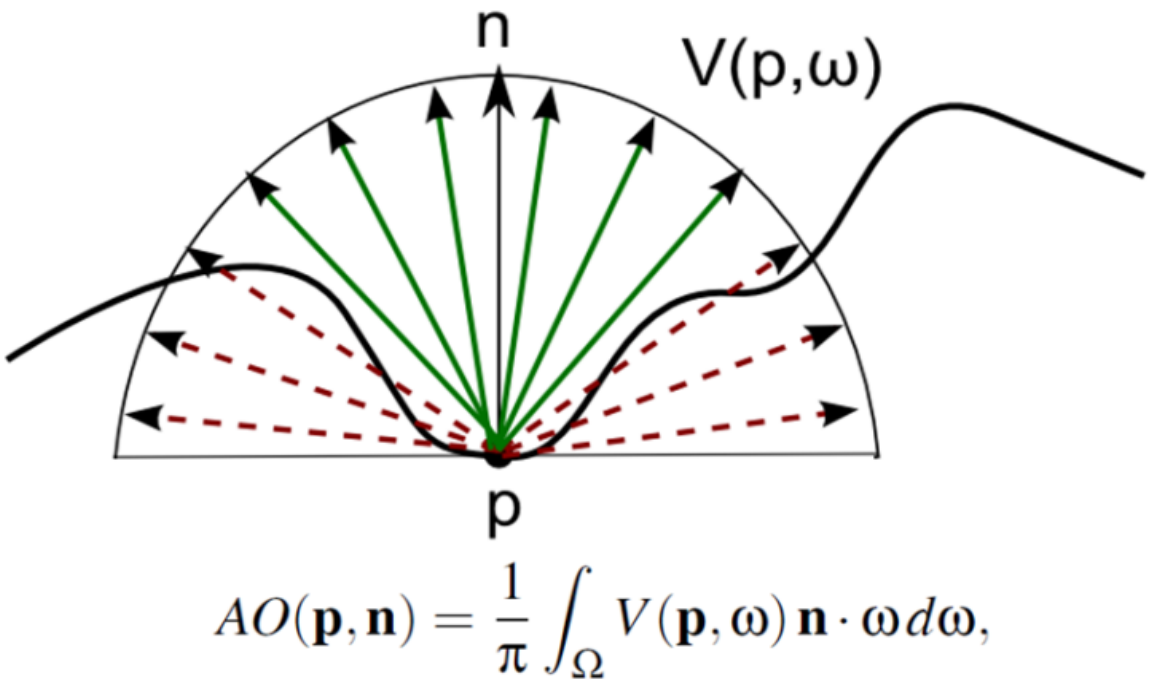

Figure 2.17: Ambient occlusion sampling hemisphere [AV18]

In most cases, especially lately with ray traced ambient occlusion(RTAO), ambient occlusion is calculated by constructing a hemisphere of rays emanating from a point on the surface in all directions and checking for intersections with other objects. Rays that reach the background or sky increase the brightness of the surface, while the rays that cross other objects do not add brightness. As a result, points surrounded by a large amount of geometry are displayed as darker ones, and points with a small amount of geometry in the visible hemisphere are displayed as lighter ones. How that sampling is implemented is explained below.

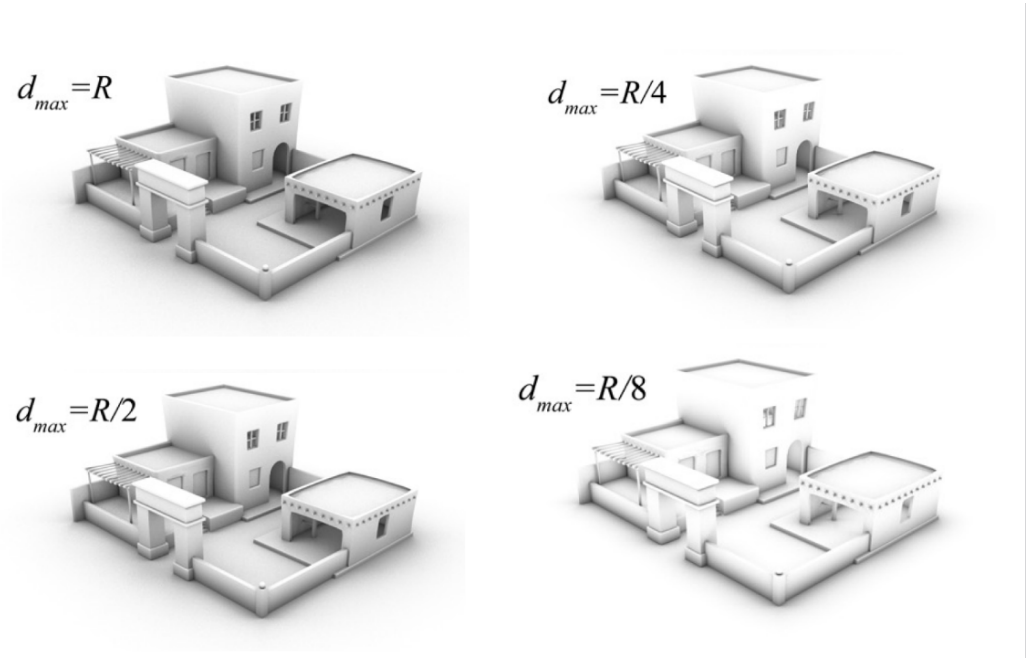

Figure 2.18 : Ambient Occlusion With different max radius [GP15b]

2.6.1 Sampling Ambient Occlusion

Ambient Occlusion belongs in a class of algorithms that depend on random sampling to achieve an outcome, the Monte Carlo methods. AO theory states that the sampling should emanate from a hemisphere **Ω**, however it is computationally impossible to sample the environment from every possible direction, that's why we take a finite number of directions/samples/rays.

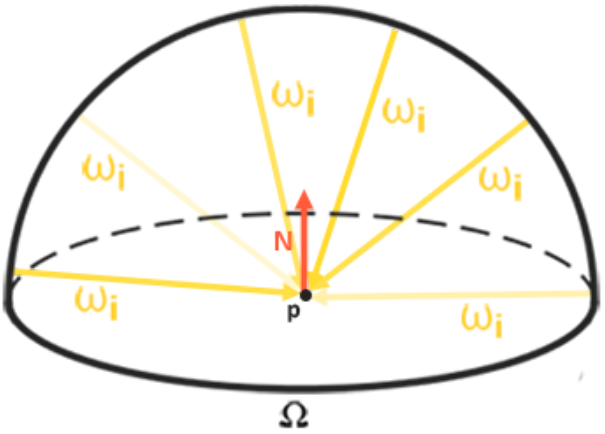

Figure 2.19: hemisphere **Ω** oriented along **N** [OP14]

To approximate the number of directions on the hemisphere we have to use spherical coordinates. Spherical coordinates are another way to define points and

vectors. In this coordinate system, the position of a point or direction and length of a vector are defined by two angles theta(θ) and phi(φ) and a radial distance(r). Theta is used to sample around the ring of the hemisphere from 0 to π radians and phi from 0 to 2π radians to sample the increasing rings of the hemisphere.

To go from cartesian to spherical coordinates the following equations are used:

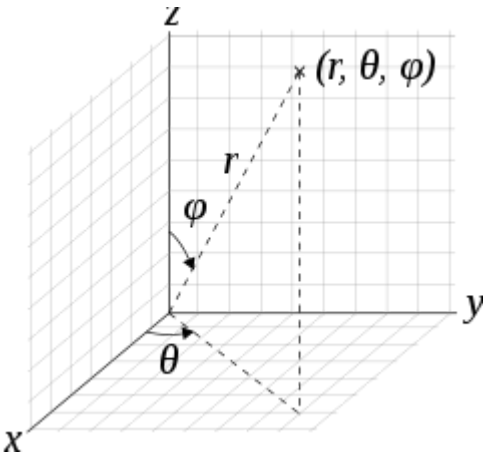

Figure 2.20: Spherical Coordinates [WIKc]

$$x = r\,sin(\theta)\,cos(\phi)$$
$$y = r\,cos(\theta)$$
$$z = r\,sin(\theta)\,sin(\phi)$$

These equations are valid only when the y-axis is the up vector, often in shading the z-axis is the up vector so y and z equations should be swapped.

2.6.2 Ambient Occlusion Light Equation

In this implantation a simplified rendering equation is used called reflectance equation:

$$L_o\left(p, \omega_o\right) = \int_{\Omega} F_r\left(p, \omega_i, \omega_o\right) L_i\left(p, \omega_i\right) n \cdot \omega_i \, d\omega_i$$

This equation describes the outgoing radiance from a point $L_o\left(p, \omega_o\right)$. Where **n** is the normal of the surface and $L_i\left(p, \omega_i\right)$ is the irradiance of the scene. To obtain the irradiance it sums all the incoming radiance, hence the integral sign in front of the equation. The dot product $n \cdot \omega_i$ is there to take into account the angle of incidence angle of the light ray. If the ray is perpendicular to the surface it will be

more localized on the lit area, while if the angle is shallow it will be spread across a bigger area, eventually spreading across too much to actually be visible.

The equation is simply representing the outgoing radiance given the incoming radiance weighted by the cosine of the angle between every incoming ray and the normal to the surface. At last $F_r\left(p, \omega_i, \omega_o\right)$ represents the BRDF, this function takes as input position, incoming and outgoing ray, and outputs a weight of how much the incoming ray is contributing to the final outgoing radiance.

Lambert's BRDF sets $F_r\left(p, \omega_i, \omega_o\right) = \frac{c}{\pi}$ where *c* is the surface colour. So the equation changes to:

$$L_o\left(p, \omega_o\right) = \frac{c}{\pi}\int_{\Omega} L_i\left(p, \omega_i\right) n \cdot \omega_i \, d\omega_i$$

Since it integrates in spherical coordinates the formula has to be changed to reflect that:

$$L_o\left(p, \omega_o\right) = \frac{c}{\pi}\int_{\Phi}\int_{\Theta} L_i\left(p, \theta_i, \varphi_i\right) cos(\theta_i) sin(\theta_i) d\theta_i d\varphi_i$$

Note that (due to the general properties of a spherical shape) the hemisphere's discrete sample area gets smaller the higher the zenith angle θas the sample regions converge towards the center top. To compensate for the smaller areas, we weigh its contribution by scaling the area by$sin\theta$[MA13]. At last the double integral is solved by applying a Monte Carlo estimator on each one, this leads to the following discrete equation that we can finally use:

$$L_o\left(p, \theta_o, \varphi_o\right) = \frac{c}{\pi}\frac{2\pi}{N_1}\frac{\pi}{2N_2}\sum^{N_1}\sum^{N_2}\left(p, \theta_i, \varphi_i\right) cos(\theta_i) sin(\theta_i) =$$

$$L_o\left(p, \theta_o, \varphi_o\right) = \frac{\pi c}{N_1 N_2}\sum^{N_1}\sum^{N_2}\left(p, \theta_i, \varphi_i\right) cos(\theta_i) sin(\theta_i)$$

2.7 Intersection Algorithms

In this chapter we'll make a brief overview of two essential algorithms that were used in this thesis to produce the final results. These algorithms are ray intersection algorithms and refer to the process of cast rays hitting either the triangles of objects or hitting boxes containing triangles en masse. These boxes were used as part of acceleration structures which will be introduced and explained later.

2.7.1 Ray-triangle intersection

There are a lot of different algorithms about ray-triangle intersection, each one with its own advantages. It is very difficult to compare the speed of all these different implementations because it depends on many factors such as the way they are implemented and the type of the scene. One of the most frequently used algorithms is Möller–Trumbores implementation. It was introduced by Tomas Möller and Ben Trumbore in their paper Fast, Minimum Storage Ray/Triangle Intersection in 1997 [MT97].

Barycentric Coordinates

This particular algorithm utilizes the barycentric coordinates of triangles, which is a coordinate system that can be used to express the position of any point located on the triangle with three scalars. That means in a triangle **T** with **v1, v2, v3** as its vertex, in order to compute the position of this point **P** using barycentric coordinates we use the following equation

$$P = t_1 v_1 + t_2 v_2 + t_3 v_3$$

where t_1, t_2, t_3 are the barycentric coordinates, three real numbers $t_1, t_2, t_3 \geq 0$ such that $t_1 + t_2 + t_3 = 1$.

The barycentric coordinates are proportional to the area of the three sub-triangles defined by P, the point located on the triangle, and the triangle's vertices (v1, v2, v3), which leads to the following formulas:

$$t_1 = \frac{v_2 P v_3}{v_1 v_2 v_3}$$

$$t_2 = \frac{v_1 P v_3}{v_1 v_2 v_3}$$

$$t_3 = \frac{v_2 P v_1}{v_1 v_2 v_3}.$$

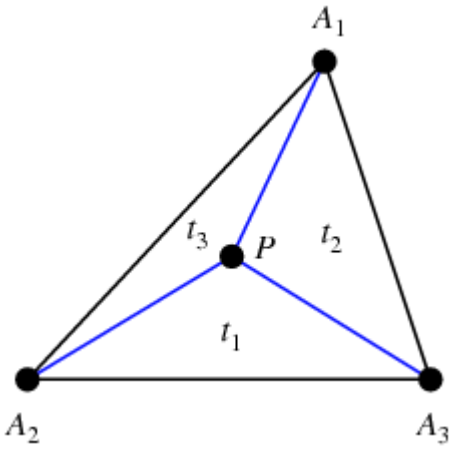

Figure 2.21: Barycentric coordinates .[WW]

In order to calculate the above triangle areas we compute the parallelogram that has as edges the two vectors. For example for the $v_2 P v_3$ area a parallelogram is created with $v_2 P$ vector as the two parallel sides and $P v_3$ as the other two. To compute the area of a parallelogram, simply compute its base, its side and multiply these two numbers together scaled by sin(θ), where θ is the angle subtended by the vectors $v_2 P$ and $P v_3$.

So the triangle area is calculated : $v_2 P v_3 = \frac{\left|\left|(v_2 - P)\right|\right| * \left|\left|(v_3 - P)\right|\right| sin(\theta)}{2}$

At last one of the cross product properties of vectors is that the magnitude of the product equals the area of a parallelogram with the vectors for sides. So the above equation transforms into:

$$v_2 P v_3 = \frac{\left|\left|(v_2 - P) * (v_3 - P)\right|\right|}{2}$$

where the length of the cross product $(v_2 - P) * (v_3 - P)$ can be easily calculated.

Algorithm

The Möller-Trumbore algorithm takes advantage of the parameterization of **P** and the intersection point in terms of barycentric coordinates. The parametric equation is $P = O + tD$ where t is the distance from the ray's origin to the

intersection **P**. As mentioned above $t_3 = 1 - t_2 - t_1$ so the barycentric equation can be written like this:

$$P = t_1 v_1 + t_2 v_2 + (1 - t_2 - t_1) v_3 =$$

$$t_1(v_1 - v_3) + t_2(v_2 - v_3) + v_3$$

If P on the parametric equation is replaced with the above equation we get

$$O + tD = v_3 + t_1(v_1 - v_3) + t_2(v_2 - v_3).$$

In order to not confuse t with t_1, t_2 we change t_1=u and t_2=v.

$$O - v_3 = -tD + u(v_1 - v_3) + v(v_2 - v_3).$$

So now on the right side of the equation all the unknowns are gathered t,u and v. The above equation can be rearranged on the left side into a row-column vector multiplication:

$$\begin{bmatrix} -D & (v1 - v3) & (v2 - v3) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - v3 \qquad \text{(Equation 1)}$$

This is the simplest possible form of matrix multiplication. You just take the first element of the raw matrix (-D, v1-v3, v2-v3) and multiply it by the first element of the column vector (*t*,*u*,*v*). Then you add the second element of the raw matrix multiplied by the second element of the column vector. Then finally add the third element of the raw matrix multiplied by the third element of the column vector.

The handy property of barycentric coordinates is that when transformations are applied on the triangle u, v will not change. We have an equation that's defined by u, v and t where $u, v <= 1$and u,v can't be lower than 0.

Cramer's rule

To solve the equation with three unknowns , Möller and Trumbore use a technique which is known in mathematics, as the Cramer's rule. Cramer's rule says that if the multiplication of a matrix M (three numbers disposed in a horizontal manner) by a column vector X (three number disposed in vertical manner) is equal to a column vector C, then it is possible to find Xi (the ith element of the column vector X) by dividing the determinant of Mi by the determinant of M. Mi is the matrix formed by replacing the ith column of M by the column vector C.

For example, when we have a system like this:

$$\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix}$$

then the values x,y,z can be calculated by this formula:

$$x = \frac{\begin{bmatrix} d1 & b1 & c1 \\ d2 & b2 & c2 \\ d3 & b3 & c3 \end{bmatrix}}{\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix}} y = \frac{\begin{bmatrix} a1 & d1 & c1 \\ a2 & d2 & c2 \\ a3 & d3 & c3 \end{bmatrix}}{\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix}} z = \frac{\begin{bmatrix} a1 & b1 & d1 \\ a2 & b2 & d2 \\ a3 & b3 & d3 \end{bmatrix}}{\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix}}$$

Implementation

Denoting $E_1 = v_1 - v_3$, $E_2 = v_2 - v_3$ and $T = O - v_3$in equation 1. The solution is obtained by using Cramer's rule.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D,\ E1,\ E2|} \begin{bmatrix} |T,\ E1,\ E2| \\ |-D,\ T,\ E2| \\ |-D,\ E1,\ T| \end{bmatrix}$$ (Equation 2)

where |A, B , C| is the determinant of the matrix that has A, B, C as its columns. It is known from linear algebra that the determinant of a 1x3 matrix is: $|A,\ B,\ C| = -(A \times B) \cdot C = -(C \times B) \cdot A$.

So equation 2 can be written:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E2) \cdot E1} \begin{bmatrix} (T \times E1) \cdot E2 \\ (D \times E2) \cdot T \\ (T \times E1) \cdot D \end{bmatrix}$$

With this equation it's easy to compute the barycentric coordinates.

Now after that the implementation of the algorithm is quite simple. The cross product of E1 and E2 gives the normal of the triangle. And it is known that if the dot product of the ray direction D and the triangle normal is 0, the triangle and the ray are parallel (and therefore there is no intersection).We first compute u and reject the triangle if u is either lower than 0 or greater than 1. If we successfully pass the computation of*u*, then we compute *v* and apply the same tests (there's no intersection if *v* is lower than 0 or greater than 1 and if u+v is greater than 1). At this point we know the ray intersects the triangle and we can compute ***t***.

2.7.2 Ray-box intersection

The ray box intersection algorithm refers to the part where a ray traverses through an acceleration data structure and more specifically a bounding volume hierarchy. It is very common for BVH algorithms to be implemented with axis-aligned bounding boxes. or else AABBs. Such boxes are called this way because they are aligned with the axis of the coordinate system that is being used. The algorithm described in this section is proposed by Smits et al. 2002 [SB02] and further optimized by Williams et. al. 2005 [WBMS05].

A common thing used in mathematics is the analytical equation of a line, which is $y = mx + b$, where m is called the slope or gradient and is responsible for the orientation of the line and b corresponds to the point where the line intersects the y-axis. The ray can also be expressed with the following equation: $O + D * t$, where O corresponds to the origin of the ray, D is the direction and the parameter t can be any real value (negative, positive or zero). By changing t in this equation, any point on the line can be defined by the ray's position and direction. To represent an axis-aligned bounding volume, only two points representing the minimum and maximum bounds of the box are needed.

The bounds of the volume define a set of lines parallel to each axis of the coordinate system which can also be expressed using the line equation. For instance the line equation for the x component of the bounding volume's minimum bound can be written as: $y = B0x$, where $B0x$in this instance corresponds to the x coordinate of the minimum bound. The point where the ray intersects this line can be found according to this equation: $Ox + tDx = B0x$, which can be solved by reordering the terms: $t0x = (B0x - Ox) / Dx$.

The x component of the bounding volume's maximum bound can be used in a similar way to compute$t1x$. When the values for t are negative, the box is behind the ray. Finally, applying the same technique to the y and z components will have as a result at the end of this process, a set of six values indicating where the ray intersects the box planes parallel to the x, y and z axis.

If the ray is parallel to an axis it won't intersect with the bounding volume plane for this axis. The problem at this stage is to find which of the above six values correspond to an intersection of the ray with the box, if the ray intersects the box at all. This can be easily found by operating a simple test on each of the computed

t values. Using figure 2.22 as an example in 2D, the ray first intersects the planes defined by the minimum bound of the box in two places: $t0x$and $t0y$. However, intersecting these planes doesn't necessarily mean that these intersecting points lie on the cube, if they don't lie on the cube, obviously the ray doesn't intersect the box. Finding which one of these two points lie on the cube is done by comparing their values. We choose the point which value for t is the greatest.

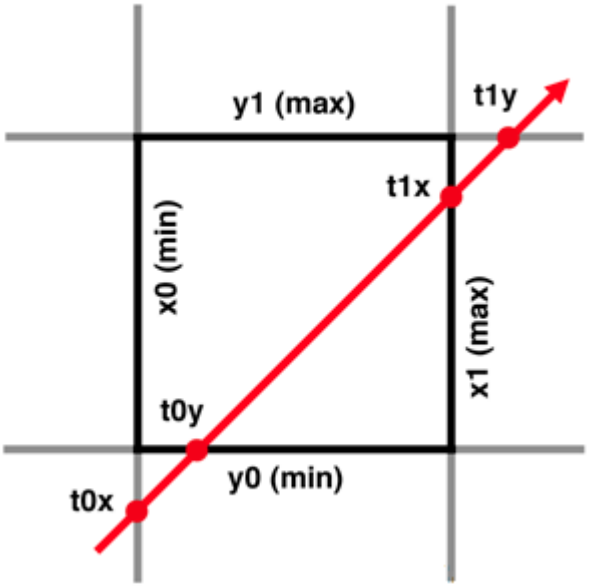

Figure 2.22: A ray intersecting a box in four points.

The process to find the second point where the ray intersects the box is similar, however in that case, the t values computed using the planes defined by the maximum bound of the box will be used and the point which the t value is the smallest will be selected.

The ray doesn't alway intersect the box. Such cases can be identified using simple comparisons between the t values. The ray misses the box when $t0x$ is greater than $t1y$ and when $t0y$ is greater than $t1x$.

The next step is to extend the technique to the 3D case by computing t values for the z component and compare them to the t values we have computed so far for the x and y components.

Finally, there is one last step which needs to be addressed to avoid possible mistakes. Usually in the code the direction of the ray is checked to see if it is positive or negative. This leads to an if-else statement that can be executed in a wrong manner by the compiler due to possible divisions by zero. When the direction of the ray is zero it causes a division by zero. This should be handled properly by the

compiler which shall return +∞. The problem happens when the value for the ray direction is minus zero. When the ray direction is negative, we expect the else statement to be executed, but in the case where the ray direction is minus zero, the first if-block will be executed instead because in IEEE float point, the test -0 = 0 returns true. The values that describe the intersection points will be set to +∞ and -∞ (instead of -∞/+∞) and the code will fail to detect an intersection. Replacing the ray direction by the inverse of the ray direction will fix this problem. In the eventuality of testing the intersection of the ray against many boxes, we can save some time by pre-computing the inverse direction and re-using it later in the intersection method. The sign of the ray direction can also be pre-computed, which will be another factor that will help in saving time. These optimizations not only make the algorithm run more efficiently but also make it more robust.

3 Light Map Generation using Ray Tracing

In this chapter we analyze the problem with ray tracing algorithms and review our implementation of light map generation using ray tracing as a solution. We present an offline cpu ray tracing system that implements a technique that captures the direct irradiance and encapsulates the outcome on a light map, for each object separately. This chapter is organized as follows, 3.1 states the problem that our thesis was based on and what was our contribution, 3.2 is a brief mention of our implementation and lastly 3.3 contains a more detailed view of it.

3.1 Problem statement

Ray tracing is a very useful technique and as mentioned in section 2.3.1, it consists of two broad stages, which we explore in this thesis. Nonetheless, ray tracing is bound by complications concerning time consumption and memory usage. However visually appealing the results of the ray tracing technique might be, it is unsuitable for real-time rendering. The computational bottleneck lies on a fundamental stage of the ray tracing technique; the ray traversal. Specifically, the entirety of the scene's geometry is stored in memory when a ray travels through the scene. Therefore, in order to find intersection points with the geometry, each ray is put through an intersection test with every triangle. These tests take a significant amount of time to complete, which approximate about 90% of the total time the whole lightmapping process took to complete.

As mentioned in the previous chapter, acceleration data structures provide an efficient way to accelerate the ray tracing process, specifically targeting the ray traversal stage. Contrary to the usefulness they may provide, they can present to be problematic in some circumstances. Due to the overall increased computational cost

in order to construct the acceleration structure, it is suited for static or partially evolving environments. When the environment is changing, the acceleration data structures should be rebuilded and as a result, this halts the overall performance making it prohibitive for real-time rendering especially in high-detail scenes.

3.1.1 Our Approach

In this work, we present a case study of ray tracing using lightmaps to offset the limitations of its real-time applications. Lightmaps are used as a way of offline rendering and to portray the results onto the screen. They are projected onto 3D models and objects to examine the results, with respect to their photo-realistic properties, in a realistic 3D world. By taking the process of calculating global illumination offline, we relieve the stage of our work that runs real time from too much workload per frame. Moreover, we present an effective way of building and using acceleration structures to further improve the ray tracing process. Finally, the contributions of this thesis briefly are:

- A comprehensive study which emphasizes the problems of ray tracing and the necessity of offline rendering.
- An offline implementation of a ray traced technique that captures the direct irradiance in order to generate photorealistic lightmaps.
- An effective and efficient way of building and using acceleration structures.

3.2 Method Description

In this chapter we will present a brief overview of our framework, to obtain some basic knowledge about it.

In a previous chapter 2.3.3 we discussed the importance of acceleration structures in ray tracing and how mandatory they are in the implementation of the technique. In this thesis we establish a simple BVH structure based on hierarchically -arranged bounding boxes. Three steps constitute this phase, the first one is sorting all the vertices in ascending order. The next step is splitting the vertices in parts in order to create the clusters. The last part is creating the bounding boxes for these clusters and keeping them in a list for later use. The difference in performance when using the bvh and when not is substantial.

After the BVH is created and all the objects of the scene are ready, the program enters the next step which is acquiring the coordinates and the normals of

each object from the GPU to CPU. This phase is composed of a pass through the shaders, where the framebuffer is bound to two textures, so as an outcome we get two textures/buffers that contain the normals and the coordinates of each object. It will be further explained below.

Ray tracing and its various implementations were widely discussed in the background section. In our case study, we do not implement a full path tracing solution for estimating surface irradiance, but rather a simpler direct sampling of the environment map using the ambient occlusion technique presented in section 2.6. Having access to the coordinates and normals of the objects, a hemisphere is created for each coordinate to sample the scene with rays. Each ray traverses through the scene; if an intersection with a bounding box is found, the ray searches for intersections with triangles inside the cluster. Assuming that an intersection occurs, this ray is terminated. In case that no intersections took place, the ray samples the cubemap 2.5.3.1.

Finally, to create the light map; for each ray the ambient occlusion light equation is used 2.6.2 and the outcome is stored in a texture/buffer.

3.3 Algorithmic Details

Now that we established the basic idea of what our thesis depicts, we will analyze our implementation in greater detail.

3.3.1 BVH Construction

As mentioned before in 3.2, the first step is creating the acceleration structure. First, the coordinates of all the objects are sorted in ascending order using a sorting algorithm similar to quicksort. Every axis is divided into four parts as shown in figure 3.1. Because the scenes in this thesis contain a small amount of triangles breaking each axis in four is adequate. If scenes with more triangles are used it might be more efficient to break them into more parts.

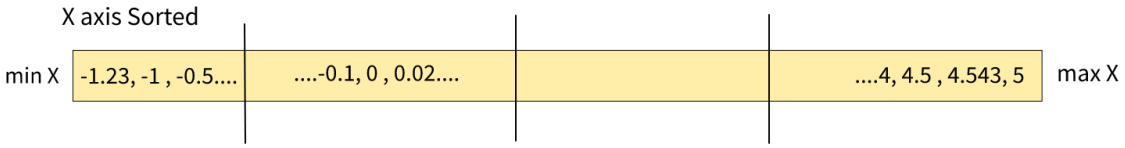

Figure 3.1: Sorting the axis and splitting them in 4 parts

This is done to break the scene in multiple clusters of triangles. For each axis there are five values that define them, the first one is the minimum value on the axes, the last one is the maximum and the others are between them. These different values from every axis are combined with the values from the other axes, to create multiple pairs of minimum point and maximum point. These pairs, known as pmin and pmax, define the first bounding boxes that are used to create the clusters. For every pair of pmin,pmax that are now in memory, it traverses through the triangles of the scene. If the center of the three vertices of the triangle is inside the area that is defined by pmin,pmax, the triangle is inserted on the cluster. The last step on the bvh implementation is creating a new bounding box for every cluster. That means the min and max vertex is found(pmin,pmax), stored in memory for later use.

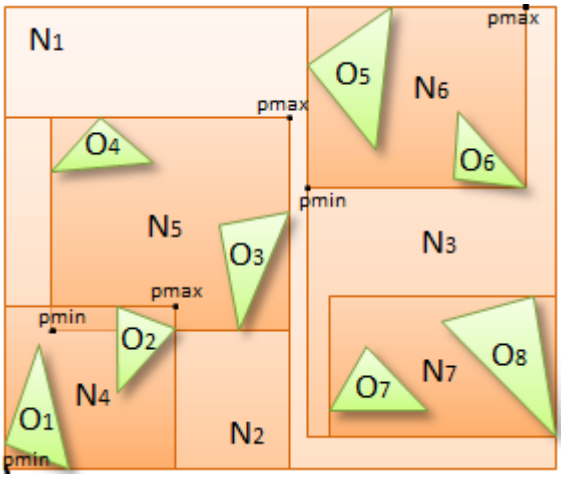

Figure 3.2: BVH and pmin pmax [TK12]

3.3.2 Coordinates Transfer

In order to begin the process of ray traced diffuse irradiance, it's essential to learn from which points the rays will launch. Normally, the sampling should occur at every vertice in the scene; but this is computationally impossible. So another method was implemented, a hybrid method using the gpu and the shaders to send the coordinates inside a texture. These are known as geometry images where cartesian coordinates are coded as colors and saved on the atlas texture. It captures geometry as a simple n x n array of [x, y, z] values, where n is the size of the texture [GU02, SAND03].

More precisely, in order to print values in a texture the framebuffer has to be bound at it. For each object two buffers are created to represent the textures in the CPU, one for the normals and one for the coordinates. There are two things that must be done before entering the shaders. First the viewport must be set to represent the texture's size. Another thing that's necessary is using an orthographic projection matrix, because the screen has to take the shape of the texture.

After that we begin the process. For each vertex in the vertex shader, the normals and the world coordinates are multiplied by the model matrix of the object, in order to be in world space. We transfer the normals, coordinates and also the texture coordinates of the vertex in order to use it afterwards. In the fragment shader these information are printed/placed on each texture respectively, at the position of the texture coordinates of the vertex. The results is shown in figure 3.3

Figure 3.3: texture with coordinates inside/size 256x256

Inside every texel in the above texture there is a coordinate that represents a point in the scene. These will be the starting points in our ray tracing algorithm.

3.3.3 Ray Tracing

Ray tracing is the main part of this thesis. After the object creates the two textures, for every proper coordinate inside, sampling rays are created towards the scene.

In order to sample on a hemisphere a tangent space has to be created. Tangent space is a space that's local to the surface of a triangle. Where the axis is the Normal, Tangent and Bitangent, or commonly known as TBN matrix figure 3.4.

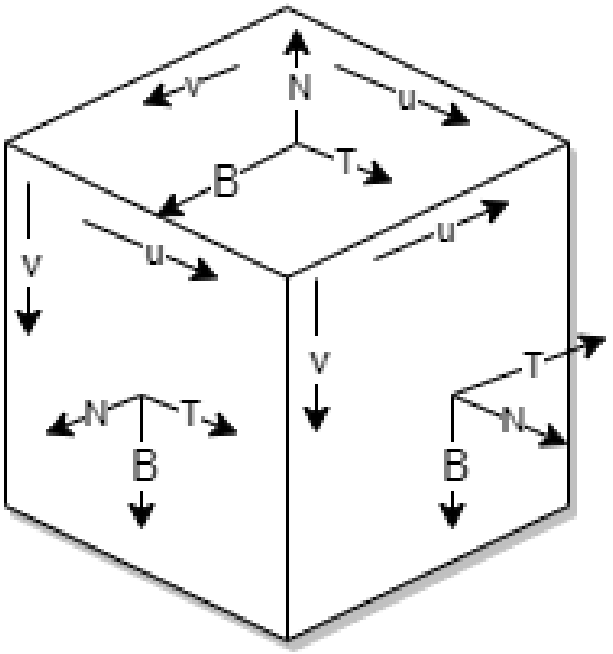

Figure 3.4: TBN Matrix [BR13]

The tangent is equal to the cross product of the normal and the up vector (0,1,0) and the bitagent with the cross product of the normal and the tangent. Unless the up vector and the normal are parallel, thus tangent and bitangent equal the vectors (1,0,0) and (0,0,1) accordingly.

As it was discussed before 2.6.1, sampling the hemisphere Ω from every possible direction is computationally impossible. In order to do that a number of rays are created using spherical coordinates. A nested loop is created, one for phi and one for theta; that as we mentioned above, value from 0 to 2π and 0 to π respectively. To approximate the number of rays that will be sent at the scene, a variable is defined as a step in the loops. The sample rays usually are approximately two hundred fifty for each coordinate.

After transforming the spherical coordinates of the ray to cartesian 2.6.1, they have to be changed from tangent to world space. This is accomplished by multiplying x of the vector with the tangent, y with bitangent and z with the normal. All that were mentioned above is shown briefly in the pseudocode below:

Algorithm Sampling Loop

Input(*Normal,tangent,bitangent****)***
Output(*sample Ray****)***

step <- *0.2*

for *(phi=0; phi<2π; phi=+ step)*
    for *(theta=0;theta<π/2;theta=+ step)*

        tangent Sample <- *(sin(theta) * cos(phi), sin(theta) * sin(phi), cos(theta))*
        sample Ray <- *tangent Sample.x * tangent + tangent Sample.y * bitangent + tangent Sample.z * Normal*
...

Now that the ray is fully constructed the traversal stage begins.

In this thesis there is no hierarchical structure on the bvh, because the scenes consist of a small amount of triangles. Thus the ray checks the intersection with every bounding box that is on memory, unless it hits a triangle. If the ray intersects a bounding box 2.7.2, it investigates if it intersects any triangles inside the bounding box 2.7.1. In case it does hit a triangle the process for this tay is terminated and black color is added on the variable that represents the light. On the other hand if the ray does not intersect with anything the sampling of the cubemap is initiated 2.5.3.1. The color that was extracted from the cubemap sampling is added on the same variable multiplied by cos(theta) and sin(theta) 2.6.2.

As shown in figure 3.5, there are three possible outcomes for each ray. Ray a intersects a bounding box and the triangles inside it. Ray b intersects bounding boxes but not the triangles inside and last ray c that does not intersect with anything. At the last two outcomes the ray samples the cubemap, on the other hand on ray a, the point is on shade so black is added to the color value.

The color for each coordinate is calculated when the above process is completed for every ray.

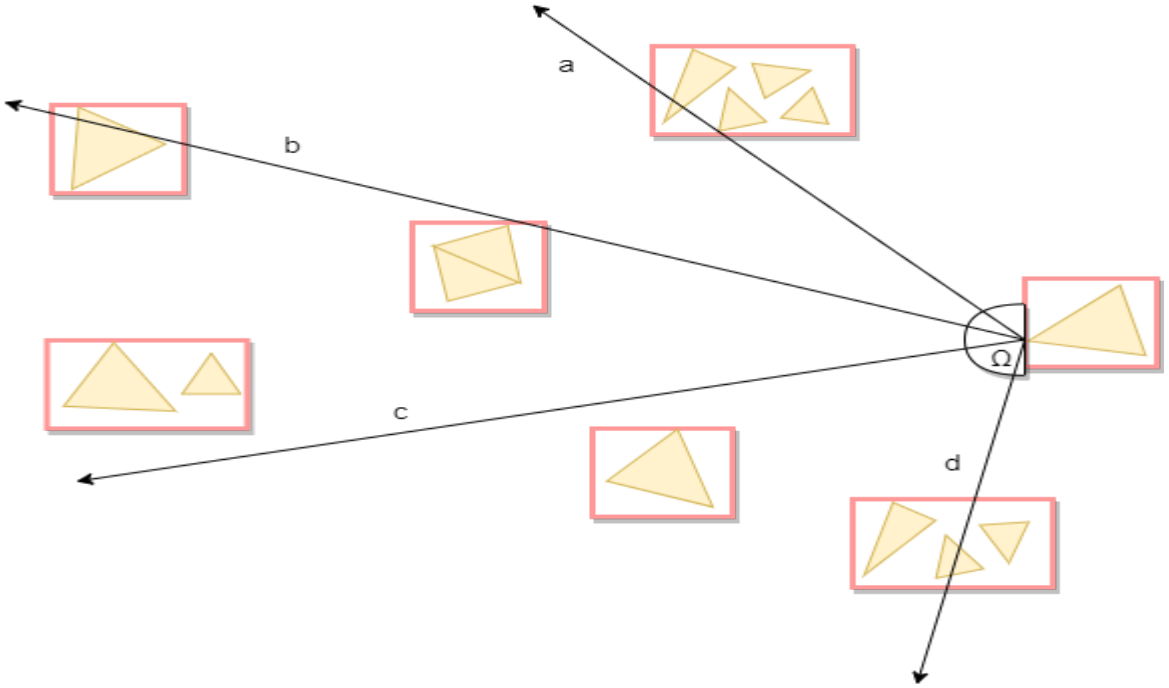

Figure 3.5: Rays Sent from a point towards the scene

3.3.4 Light Map

There are two steps needed in order to create the light map. First, after every ray is finished the color must be averaged and multiplied by π, as the equation states 2.6.2. The derived color is placed in the exact same position as the coordinates on a different buffer. That means when placing the x,y,z values of the color variable on r,g,b respectively, we also insert one on the alpha channel. This will help us on the next step that is padding up the texture.

Padding

After the lighmap is successfully created and stored, it is important to make corrections to avoid possible problems.

One problem that might occur is color leaking. Color leakage can occur due to the technique that was used to create the lightmap and the final result may have lumels with the wrong color or no color at all. Thus, not only affecting the result visually but also making it harder to process it programmatically. Color leakage is described with this term due to the possibility of spreading in neighboring lumels making the visual result worse.

Apart from color leakage, another common problem is that there may be empty areas inside of the lightmap. These empty areas usually form around UV islands. These areas do not affect the processing part in any way but have a visual impact on the final result.

These two problems have a common solution. Padding or else dilation, is the process of filling the empty areas of the lightmap with similar lumels. The color of the empty lumel is filled with the average color of its nine neighbors as shown in the algorithm below. Game engines have the option to create a texture where each texel has four values instead of only the three RGB ones. The fourth one is the alpha value which defines the opacity of the texel, but in this case, it can be used as a binary boolean flag with values “zero” and “one”. The value “zero” means the lumel is empty and respectively the value “one” means the lumel has a color value. Thus, determining which lumel is empty and which of the neighbors should be factored into the average, is implemented in this way.

It is important to mention that there are not any techniques that will guarantee there will be no color leakage or empty areas, but taking measures against these potential problems is always advised.

Algorithm Padding

Input (texture)

```
ColorValue <- (0,0,0)
NumberOfNeighbors <- 0

For all the texels of the texture
    If its Empty
        for each Neighbor that isn't empty
            ColorValue <- NeighborColor
            NumberOfNeighbors +1
        If NumberOfNeighbors >1
            texel <- ColorValue / NumberOfNeighbors
.......
```

Figure 3.6: The result of the padding process.[WIKd]

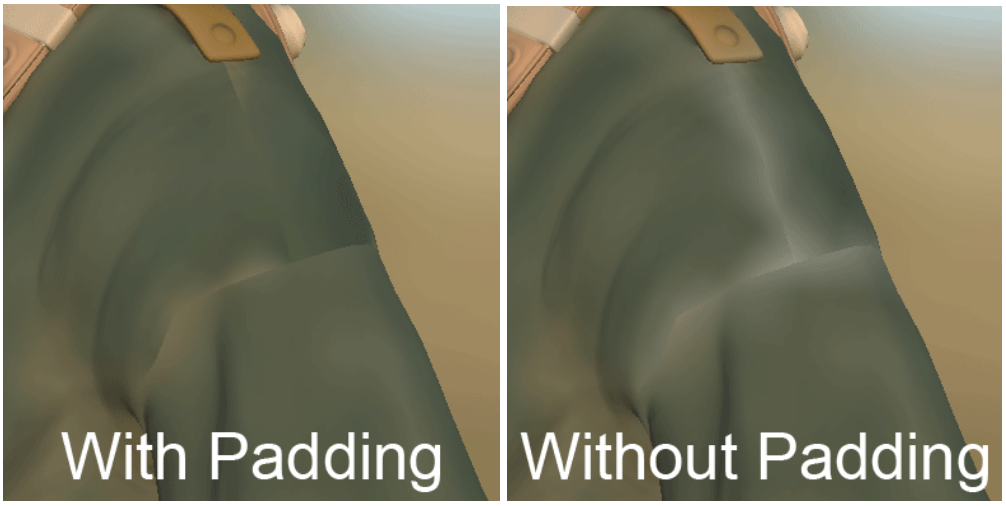

Figure 3.7: The visual results of the padding process onto a 3D model.[ADOB]

Outcome

When this process is finished the light map is created. This is implemented for every object on the scene and stored in memory to use it real-time. An example of a ray traced light map is shown in the figure below.

Figure 3.8: The outcome Light map of our algorithm

4 Evaluation and Examples

In this chapter our framework will be evaluated. In order to do that a series of rendered scenes will be presented. The system used for running the tests consists of an intel i3 9100f with a GeForce GTX 1050 ti and 16 GB RAM. Simple scenes with not many triangles were used to present the technique. In table 4.1 we present the objects that will be in our scenes and how many triangles compose them. There will be various examples of different objects, or same objects with different light map sizes to compare the differences. Unless specified in the sampling part, two hundred and fifty six rays will be created to sample the hemisphere.

	Wall	**Cannon**	**Fire-hydrant**	**Storage-room**	**Apartment**	**Wheelbarrow**	**Cannon Mount**	**Door**
Triangles	4999	1350	4634	1500	504	1700	766	5772

Table 4.1: Primitive count for every object presented

Every object is rendered inside a cubemap/skybox in order to sample it. For each scene presented we will mention the rendering time and also small problems that might occurred in the creation of the light map.

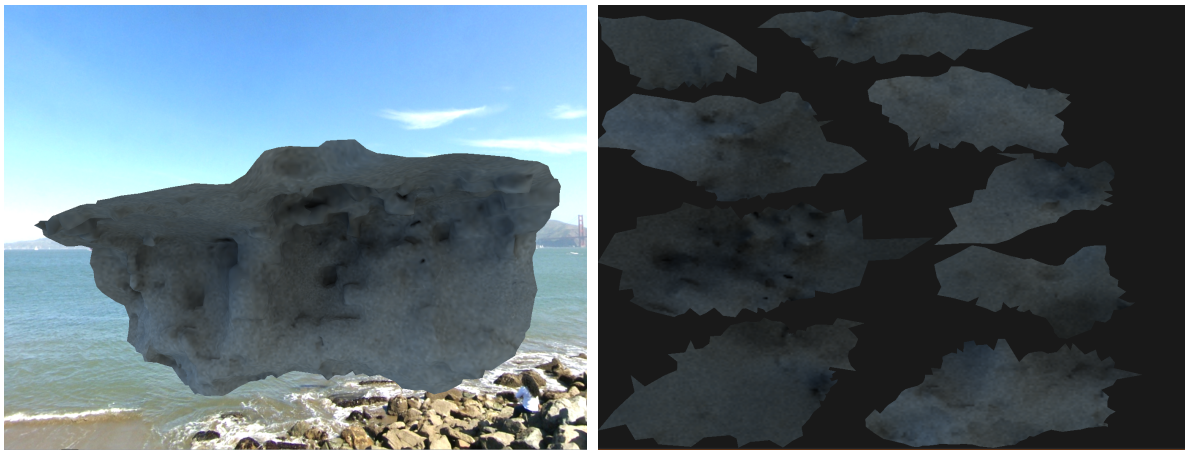

Figure 4.1: Wall. The light map size was 256x256 and was rendered in 4:20.

Figure 4.5: Fire hydrant rendered three times with different light map sizes 128x128, 256x256, 512x512. The render times were 1:00, 2:30 and 11:00 respectively. The difference in quality and smoothness is visible.

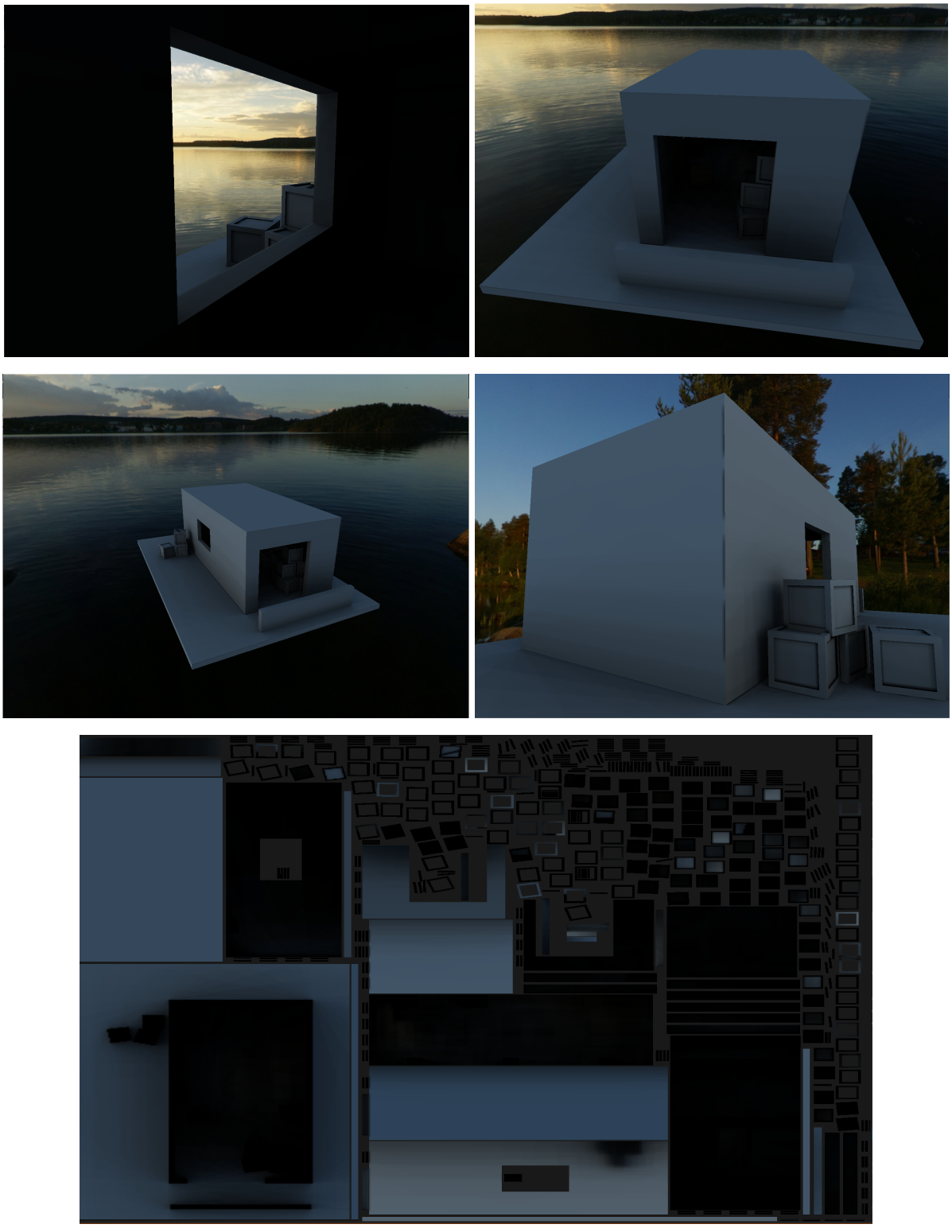

Figure 4.2: Storage room. Light Map size 512x512 and was rendered in 6:10. In the fourth picture there is a small mistake in the corner of the building, this is due to the sampling of near charts in the atlas. See figure 4.3.

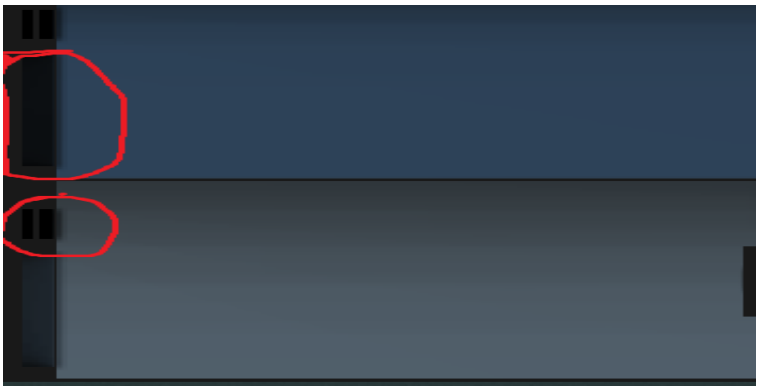

Figure 4.3: The corner of the building samples the nearby texels and black color appears on the wall because of the object that is next to it, also known as color leakage.

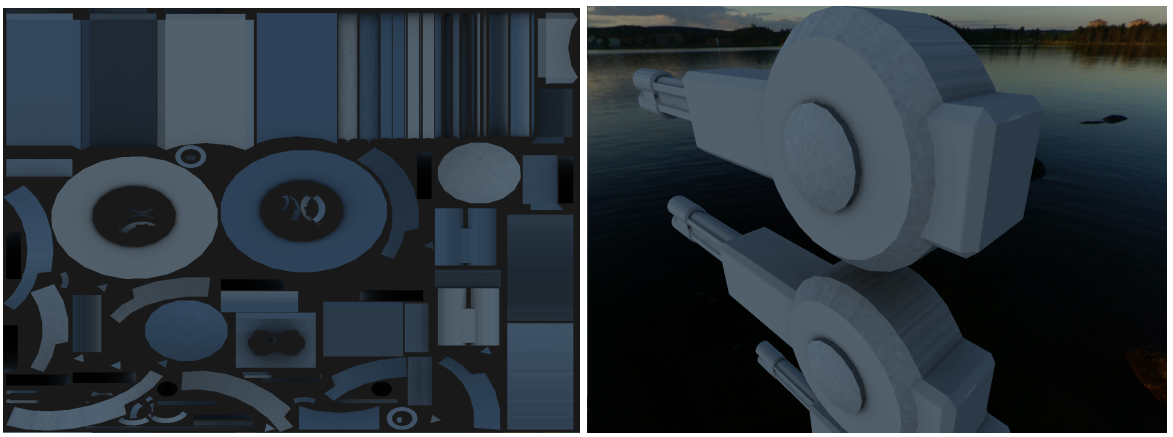

Figure 4.4: Four Cannons. Light Map size 256x256 and was rendered in 23:30.

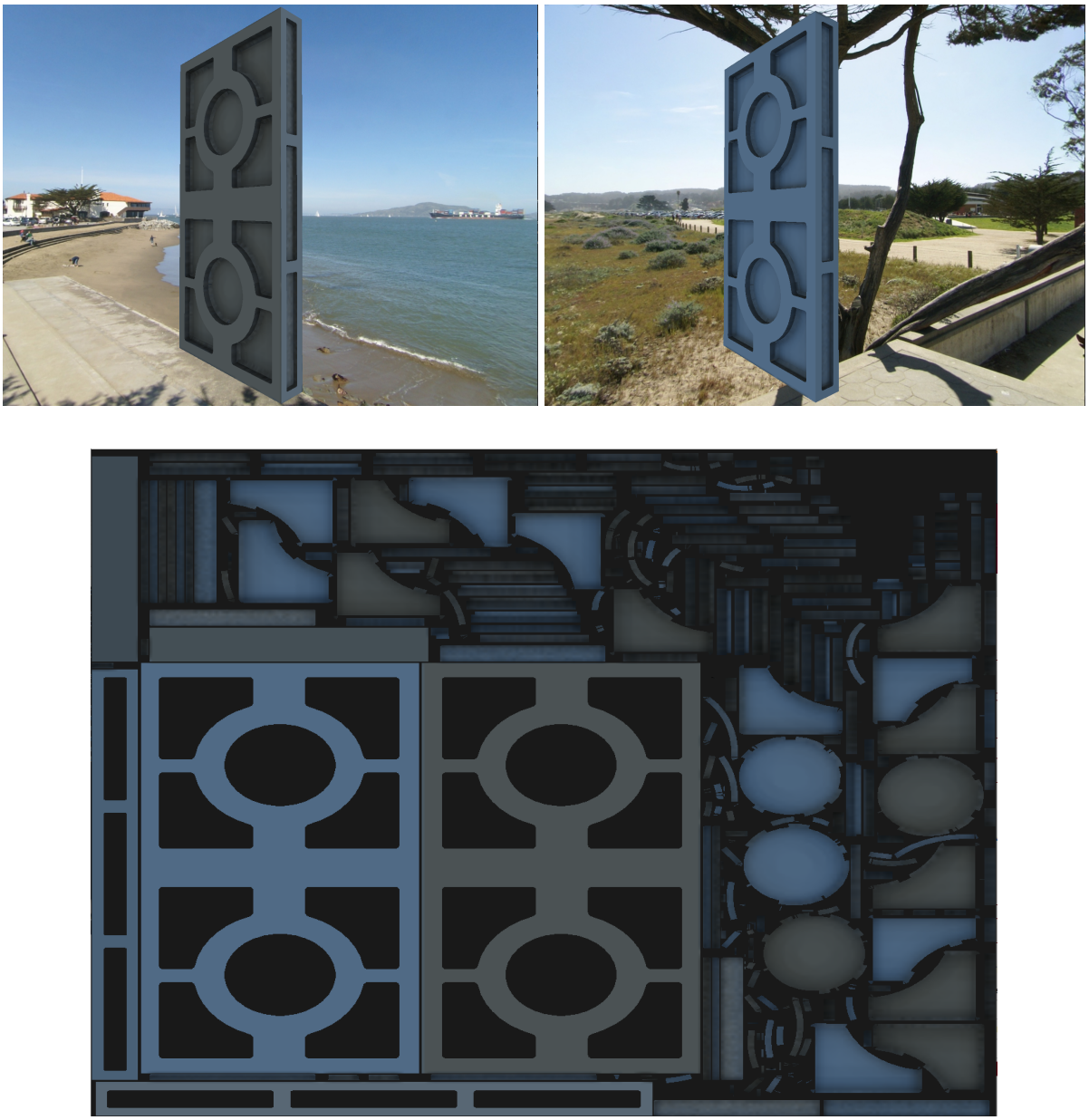

Figure 4.5: Door. Light map size 256x256 and rendered in 4:00.

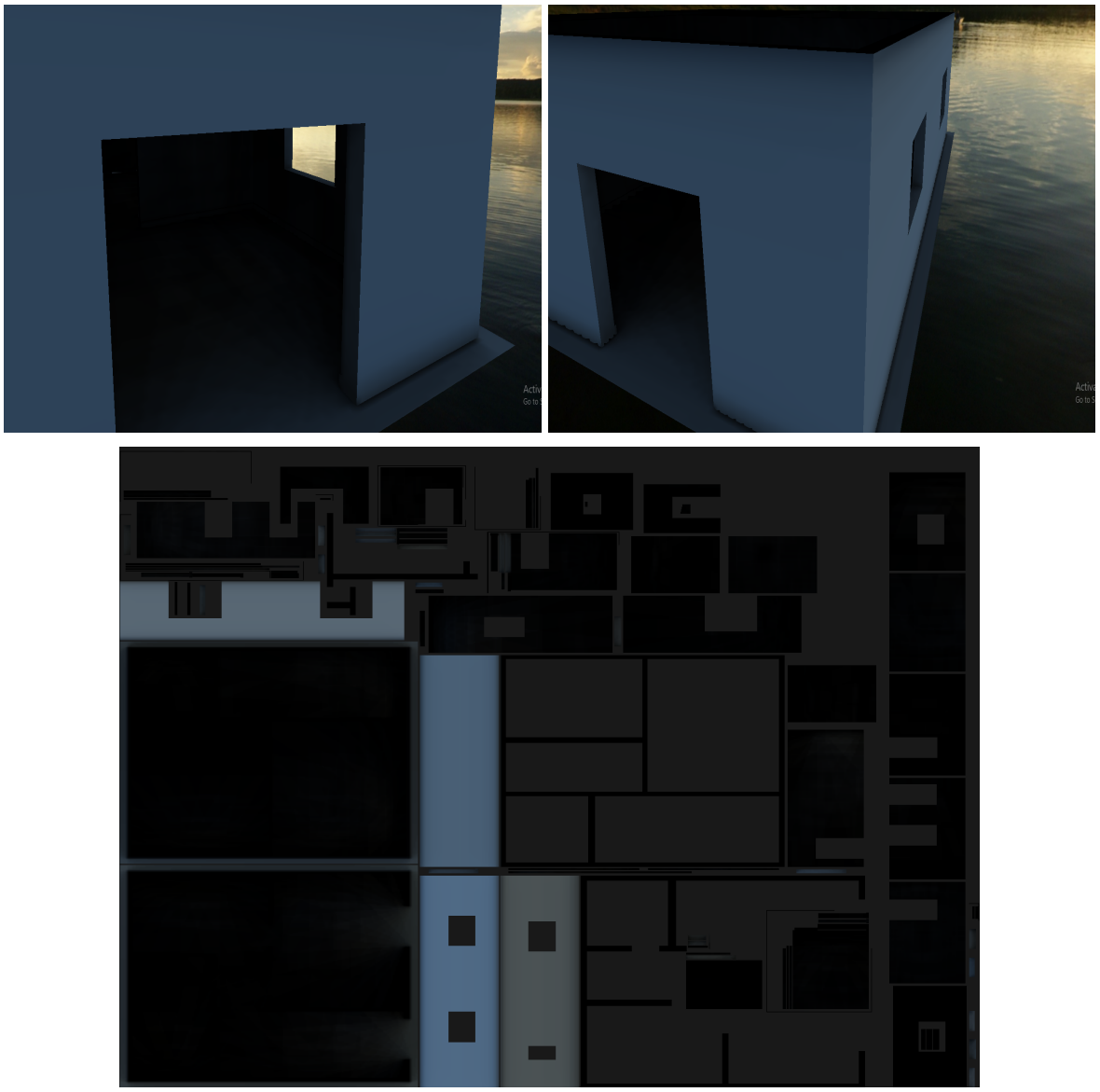

Figure 4.6: Apartment. Light map size 512x512, rendered in 1:20.

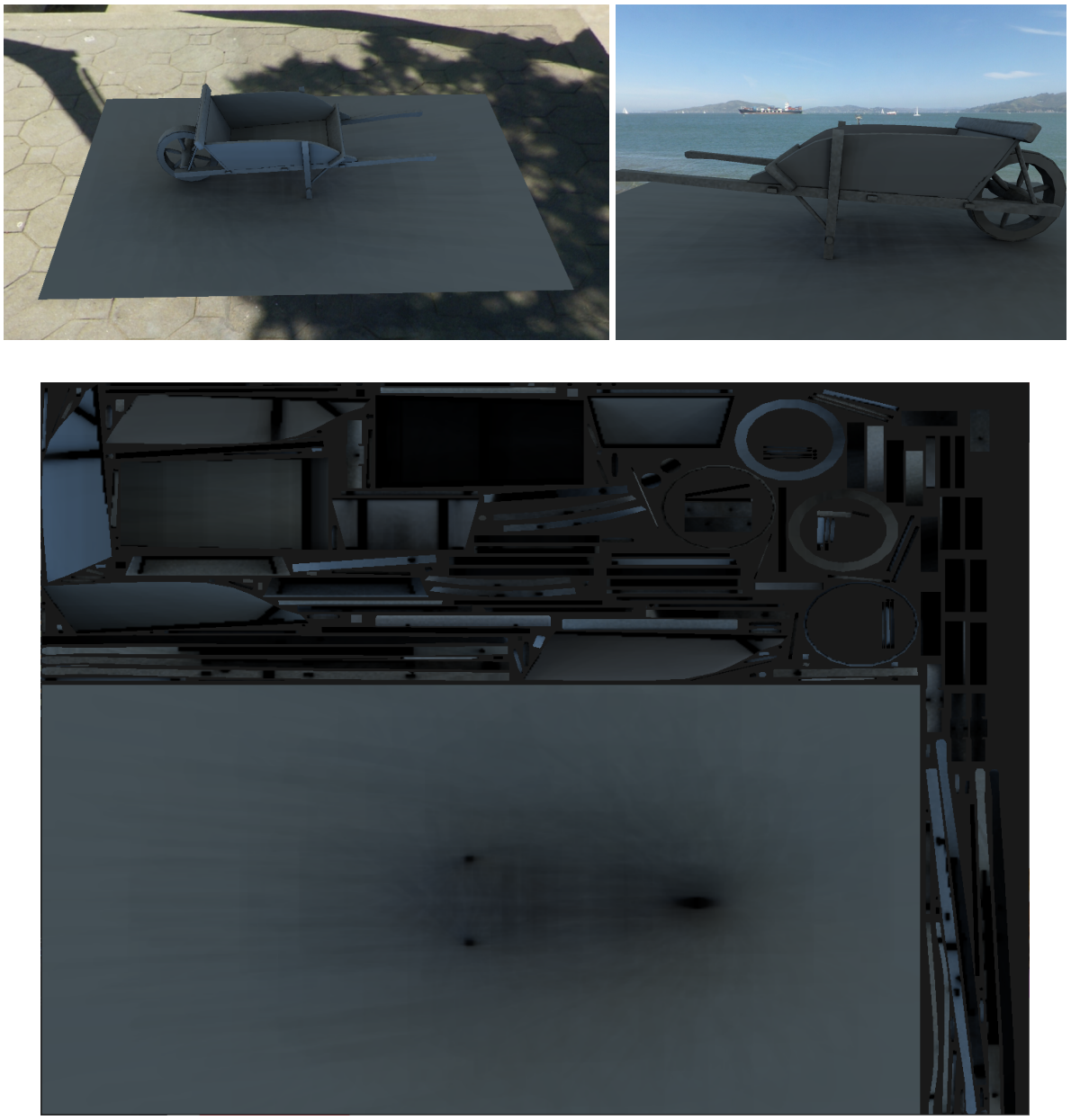

Figure 4.7: Wheelbarrow. Light map size 512x512, rendered in 7:00.

Conclusion

In this thesis, we presented a ray tracing algorithm that captures the ambient light and bake it in a light map. We will briefly make an overview of our thesis and provide parts that may need further optimizations.

We demonstrated a simple version of a bvh structure in order to speed up our ray tracing algorithm and reduce memory usage. Furthermore, we introduced the geometry images in order to represent an approximation of the objects vertices; thus, reducing memory usage. Finally, we established a ray traced direct irradiance algorithm using the geometry images as starting points, in order to encapsulate the result on a light map. By taking this process offline we drastically reduce time and memory usage in real time rendering, while maintaining quality in high levels. This method, because of its texture format, can be combined with other techniques that encapsulate other light phenomena.

In conclusion, improvements can be made in order to optimize our algorithm. By creating a hierarchical structure on our bvh we could reduce time creating the light map and support scenes with a bigger amount of triangles. Furthermore, in order to prevent color leakage more research should be focused on our padding algorithm and on atlas generation, in order to avoid small miscalculations.

Bibliography

[AA68] Arthur Appel. Some techniques for shading machine renderings of solids. In Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[ADOB] Adobe Substance 3D painter. Padding. Last modified 2021/06/23 URL: https://substance3d.adobe.com/documentation/spdoc/padding-134643719.html?fbclid=IwAR3A2TIpaH9ylnwPN2r95rBfClQ18CGi29Hrf--o01dnp55TBB1_NEzQsGk

[AV18] arviVR. November,07 2018 Ambient Occlusion: An Extensive Guide on Its Algorithms and Use in VR. URL: https://vr.arvilab.com/blog/ambient-occlusion

[BL77] James F. Blinn. Models of Light Reflection for Computer Synthesized Pictures. In: Proceed-ings of the 4th Annual Conference on Computer Graphics and Interactive Techniques. SIG-GRAPH '77. San Jose, California: ACM, 1977, pp. 192–198. DOI: 10.1145/563858.563893

[BR13] Basler, Robert, editor. *Three Normal Mapping Techniques Explained For the Mathematically Uninclined*. 22 November 2013. URL: https://www.gamedeveloper.com/programming/three-normal-mapping-techniques-explained-for-the-mathematically-uninclined

[BTP75] Bui Tuong Phong, Illumination for computer generated pictures, Communications of ACM 18 (1975). University of Utah.

[CJ83] Cleary, John G., et al. "Design and analysis of a parallel ray tracing computer." *Graphics Interface*. Vol. 83. 1983.

[CPC84] Robert L. Cook, Thomas Porter, Loren Carpenter (1984). " Distributed Ray Tracing". Computer Division Lucasfilm Ltd.

[CT82] R. L. Cook and K. E. Torrance. A Reflectance Model for Computer Graphics. In: ACM Trans.Graph. 1.1 (Jan. 1982), pp. 7–24. ISSN: 0730-0301. DOI: 10.1145/357290.357293

[DE21] David J. Eck. August 2021 Introduction to Computer: Graphics Section 3.3 Projection and Viewing. Hobart and William Smith Colleges.

[EI19] Evangelou Iordanis. September 2019. Rasterization-Based Ray Tracing. Department of Informatics MSc in Computer Science Athens University of economics and Business

[FI85] Fujimoto A., Iwata K. (1985) Accelerated Ray Tracing. In: Kunii T.L. (eds) Computer Graphics. Springer, Tokyo. https://doi.org/10.1007/978-4-431-68030-7_4

[GH71] Gouraud, Henri (1971). *Computer Display of Curved Surfaces*, *Doctoral Thesis* (Thesis). University of Utah.

[GN86] Greene, N (1986). "Environment mapping and other applications of world projections". *IEEE Comput. Graph. Appl.* **6** (11): 21–29. doi:10.1109/MCG.1986.276658.

[GP15a] Georgios Papaioannou. 2015 Light Transport Foundations.Aueb Computer Graphics course

[GP15b] Georgios Papaioannou. 2015 Environment samling.Aueb Computer Graphics course. pages: 19,11

[GP15c] Georgios Papaioannou. 2015 Geometry Representation.Aueb Computer Graphics course

[GP15d] Georgios Papaioannou. 2015 Viewing and Projections.Aueb Computer Graphics course

[GP15e] Georgios Papaioannou. 2015 Rendering Pipelines.Aueb Computer Graphics course

[GP15f] Georgios Papaioannou. 2015 Game Graphic Techniques.Aueb Computer Graphics course

[GU02] X. Gu, S.J. Gotler and H. Hoppe. Geometry Images. Transactions on Graphics (Proc. SIGGRAPH 02),21(3).2002.

[HW14] Hu, Wei et al. “Ray tracing via GPU rasterization.” The Visual Computer 30 (2014): 697-706.

[JK86] James T. Kajiya.The Rendering Equation. In:Proceedings of the 13th Annual Conferenceon Computer Graphics and Interactive Techniques. SIGGRAPH '86. New York, NY, USA:ACM, 1986, pp. 143–150.ISBN: 0-89791-196-2.DOI:10.1145/15922.15902

[KAW06] Knoll, Aaron & Wald, Ingo & Parker, Steven & Hansen, Charles. (2006). Interactive Isosurface Ray Tracing of Large Octree Volumes. 115-124. 10.1109/RT.2006.280222.

[KC03] Keshav Channa. 21 July 2003. Light- Mapping Theory and Implementation. URL: https://www.flipcode.com/archives/Light_Mapping_Theory_and_Implementation.shtml

[KV16] Konstantinos Vardis. Efficient Illumination Algorithms for Global Illumination In Interactive and Real-Time Rendering. PhD thesis, Department of Informatics, Athens University of Economics and Business, 2016.

[LM60] J. H. Lambert. Photometria, sive, De mensura et gradibus luminis, colorum et umbrae.Augsburg, Germany: V. E. Klett, 1760

[MA00] Michael Abrash. Quake's lighting model: Surface caching. Blue's News Ramblings in Real Time.April 24, 2000

[MA13] Marco Alamia."Article-Physical Based Rendering". Coding Labs.27/05/2013. URL: http://www.codinglabs.net/article_physically_based_rendering.aspx

[MT97] Fast, Minimum Storage Ray/Triangle Intersection. Möller & Trumbore. Journal of Graphics Tools, 1997.

[MW95] Michael F. Worboys (30 October 1995). *GIS: A Computer Science Perspective*. CRC Press. pp. 232–. ISBN 978-0-7484-0065-2.

[OP14] Joey de Vries. 2014. LearnOpenGl. Diffuse Irradiance. URL: https://learnopengl.com/PBR/IBL/Diffuse-irradiance

[OY00] Ohno, Y. (2000), Radiometry and Photometry: Review for Vision Optics, OSA Handbook of Optics Book Chapter (14), OSA Handbook of Optics, Volume III Visual Optics and Vision, (Accessed September 4, 2021)

[RH11] Rachmadi, Reza & Hariadi, Mochamad. (2011). GPU-Based Ray Tracing Algorithm Using Uniform Grid Structure. Jurnal Ilmiah Ilmu Komputer UPH. 7. 111-116.

[RW18] Roman Wiche. Apr 11,2018. How to create awesome accelerators: The Surface Area Heuristic. URL: https://medium.com/@bromanz/how-to-create-awesome-accelerators-the-surface-area-heuristic-e14b5dec6160

[RW21] REDWay3D. Redsdk books Rendering with RedSdk. Software rendering of 3D datasets/ Global illumination. Documentación generated 04/02/2021. URL: http://www.downloads.redway3d.com/downloads/public/documentation/bk_re_sw3d_gi.html

[SB02] Smits B. October 21, 2002 Volume 15, Number 1. Efficient bounding box intersection. Ray tracing news. URL: http://www.realtimerendering.com/resources/RTNews/html/rtnv15n1.html#art4

[SMSFD20] Ströter, Daniel & Mueller-Roemer, Johannes & Stork, André & Fellner, Dieter. (2020). OLBVH: octree linear bounding volume hierarchy for volumetric meshes. The Visual Computer.

[SS92] Kelvin Sung, Peter Shirley, VI.1 - RAY TRACING WITH THE BSP TREE, Editor(s): DAVID KIRK, Graphics Gems III (IBM Version), Morgan Kaufmann, 1992, Pages 271-274, ISBN 9780124096738.

[SAND03] P. Sander, Z. Wood, S. Gotler, J. Snyder and H. Hoppe. Multi -Chart Geometry Images, in Eurographics Symposium on Geometry Proceedings, Eurographics. Switzerland, 2004.

[TK12] Tero Karras. November 26 2012. Thinking Parallel, Part II: Tree Traversal on the GPU. Nvidia Developer/ Developer Blog. URL: https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/

[TMB05] Tom McReynolds, David Blythe. Published 2005. Advanced Graphics Programming Using OpenGL. ISBN 978-1-55860-659-3

[VG17] Verhoeven Geert J. August 2017.Computer Graphics Meets Image Fusion: the Power of Texture Baking to Simultaneously Visualise 3d Surface Features and

Colour. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume IV-2/W2, 2017, pp.295-302

[VTLB15] Vo, Anh-Vu & Truong-Hong, Linh & Laefer, Debra & Bertolotto, Michela. (2015). Octree-based region growing for point cloud segmentation. ISPRS Journal of Photogrammetry and Remote Sensing. 104. 10.1016/j.isprsjprs.2015.01.011.

[WBMS05] Amy Williams, Steve Barus, R.Keith Morley, Peter Shirley. University of Utah. An Efficient and Robust Ray-Box Intersection Algorithm. January 2005 Journal of Graphics GPU and Game Tools 10(1):49-54 DOI:10.1145/1198555.1198748 Source DBLP

[WJ08] Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. Ph.D. dissertation, UC San Diego, September 2008.

[WIKa] Wikipedia. Rendering Equation- Wikipedia the free Encyclopedia. URL: https://en.wikipedia.org/wiki/Rendering_equation

[WIKb] Wikipedia. Cube Mapping- Wikipedia the free Encyclopedia. URL: https://en.wikipedia.org/wiki/Cube_mapping

[WIKc] Wikipedia. Spherical coordinate system- Wikipedia the free Encyclopedia. URL: https://en.wikipedia.org/wiki/Spherical_coordinate_system

[WIKd] Wikipedia. Texture Mapping- Wikipedia the free Encyclopedia. URL: https://en.wikipedia.org/wiki/Texture_mapping

[WW] Weisstein, Eric W. "Barycentric Coordinates." From *MathWorld*--A Wolfram Web Resource. https://mathworld.wolfram.com/BarycentricCoordinates.html

[TPPP07] Theoharis, T. & Papaioannou, Georgios & Platis, Nikos & Patrikalakis, Nicholas. (2007). Graphics and Visualization: Principles & Algorithms. pp. 480-505.

[LPRM02] Levy, Bruno & Petitjean, Sylvain & Ray, Nicolas & Maillot, Jérôme. (2002). Least Squares Conformal Maps for Automatic Texture Atlas Generation. ACM Trans. Graph.. 21. 362-371. 10.1145/566654.566590.