UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento di Matematica e Informatica
Bachelor of Computer Science

Salvatore Daniele Calanna

Real-Time Physically Based Rendering oriented to GPGPU algorithms visualization

Supervisors:

Prof. Filippo Stanco

Prof. Giuseppe Bilotta

Dr. Dario Allegra

Academic Year 2017/2018

Acknowledgements

It has been a period of intense learning for me, but thankfully, professors, colleagues, friends and parents have been helping me in this course of study which has animated these last 3 years.

Firstly, I would like to express my profound gratitude to Prof. Filippo Stanco, for his support during the academic years, and to Prof. Giuseppe Bilotta, for his patient guidance and the quantity of knowledge transmitted to me. Last but not least, I would like to thank Dr. Dario Allegra, for his suggestions that helped me write this paper. Their teaching and enthusiasm for the topic have been crucial to achieving my goals and I will always carry positive memories of them.

I am forever thankful to all of the colleagues for their friendship and support, and for creating a cordial working environment.

I would also like to take this opportunity to express my sincere gratitude to Riccardo Torrisi, for his collaboration throughout the development of this thesis.

Finally, I must express my very profound gratitude to my parents Tania and Santo, my sister Debora and my girlfriend Miriam for providing me with continuous encouragements and love throughout these years. This accomplishment would not have been possible without them. Thank you.

Contents

Acknowledgements **i**

1 Introduction **1**

2 Basic concepts and definitions **3**

2.1 3D Cartesian coordinate system 3

2.2 Polygon mesh 4

2.3 Normal 5

2.4 Texture 8

2.5 OBJ file format 9

2.6 Smoothing 10

2.6.1 Gaussian Smoothing 10

2.6.2 Taubin Smoothing 11

3 GPU Parallel Computing **13**

3.1 Parallel Computing 13

3.1.1 Stream Processing 15

3.2 Hardware 15

3.2.1 Streaming Processor 16

3.2.2 Streaming Multiprocessor 17

3.2.3 Memory hierarchy 19

3.3 GPGPU 21

3.3.1 GPGPU architecture 22
3.4 Divergence 23

4 Rendering **25**
4.1 Techniques 25
4.1.1 Rasterization 26
4.1.2 Ray casting 27
4.1.3 Ray tracing 28
4.2 Photorealism 29
4.3 Rendering equation 30
4.4 Physically Based Rendering 32
4.4.1 Theory 33
4.4.2 Energy conservation 34
4.4.3 The microfacet model 35
4.4.4 BRDF 35
4.5 Metals and dielectrics 38
4.6 PBR Texture Maps 40

5 Implementation **42**
5.1 Engine structure 43
5.2 Camera 44
5.2.1 View matrix 44
5.2.2 Projection matrix 45
5.3 Mesh Renderer 46
5.4 Shaders 46
5.4.1 Vertex shader 48
5.4.2 Fragment shader 48
5.5 PBR Shader 49
5.5.1 Distribution GGX 49
5.5.2 Geometry Smith 50

5.5.3 Fresnel Schlick 52
5.5.4 BRDF 52
5.6 OpenCL/OpenGL Interoperability 53

6 Results **57**
6.1 Interoperability performance 57
6.2 PBR Comparisons 59

7 Conclusions **63**

Chapter 1

Introduction

In the last few years, computer-generated imagery has achieved photorealistic results that would have not been possible several years ago due to the lack of hardware computational power. The high-quality rendering that we can achieve today is obtained by simulating or approximating the laws of light of the real world. Many scientists have been trying to find formulas to well approximate the behaviour of real materials, thus marking the beginning of Physically Based Rendering which is the last rendering model adopted by many realistic video games and movies productions. This model simplifies the creation of realistic 3D models by specifying materials properties known in nature (roughness, metallic, etc.).

Another important area which is taking hold in recent years is GPGPU. The parallel computing capability of the modern computer graphics dedicated hardware (GPU) can be used to boost general purpose algorithms beyond rendering. This way, it is possible to make computer simulations (e.g. fluid, destructions, physics collisions) even in real-time, therefore GPGPU architecture has become really important in scientific research of many types.

This thesis is focused on explaining the concepts of Physically Based Rendering describing a Real-time PBR Engine oriented to visualize a GPGPU algorithm implementation of the Taubin mesh smoothing in order to see the results of the smoothing process in real-time. The software implementation discussed in this paper has been developed in collaboration with Riccardo Torrisi, who has described in details the GPGPU implementation of the Taubin mesh smoothing in his bachelor's thesis "3D Mesh Smoothing: Parallel implementation and real-time visualization"[1].

Through this paper, we will introduce some basic concepts of computer graphics. Then we will describe the modern GPU architecture that is the hardware core of 3D computer graphics. Finally, we will explain rendering techniques used in modern games and film productions and their software implementations.

Chapter 2

Basic concepts and definitions

2.1 3D Cartesian coordinate system

A *3D Cartesian coordinate system* is a coordinate system where each point is uniquely identified by an ordered triplet of numerical coordinates that represents the signed distances to the point from its perpendicular projections onto three mutually perpendicular lines that have the same unit of length.

Each reference line is called *coordinate axis* (or axis) of the system and the point where axes meet is known as *origin* of the coordinate system, identified by the triplet $(0, 0, 0)$. There are no standard names for the coordinates but they are often denoted by the letters X, Y, and Z and the lines are denoted X-axis, Y-axis, and Z-axis, respectively. The unit points on the three axes are $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$.

The Cartesian coordinate system chosen for the representation of 3D data in this thesis is shown in Figure 2.1, where X represents the right vector, Y represents the up vector and $-Z$ represents the forward vector.

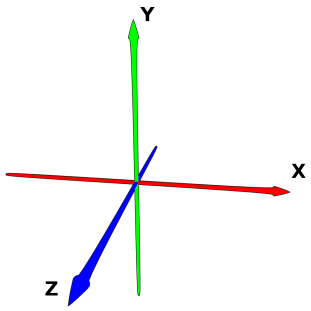


Figure 2.1: Cartesian Coordinate System

2.2 Polygon mesh

A *polygon mesh* is a set of *vertices*, *edges* and *faces* that defines the shape of a polyhedral object in 3D computer graphics.

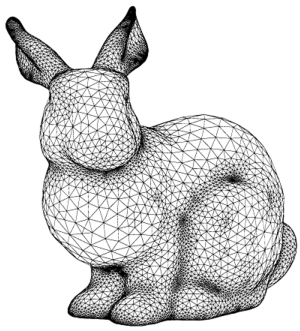


Figure 2.2: Polygon mesh

Let M a polygon mesh, n_V the number of vertices in M and n_F the number of faces in M. In mathematics we can define a polygon mesh as a pair of sets $M = \{V, F\}$, where $V = \{v_i : 1 \leq i \leq n_V\}$ is a list of vertices and $F = \{f_k : 1 \leq k \leq n_F\}$ a list of faces, with each face $f_k = (i_1^k, \ldots, i_n^k)$ such that $i_1^k \neq i_2^k \neq \cdots \neq i_{n-1} \neq i_n^k$, composed by at least three indices of vertices.

In the particular case in which all faces are made by exactly three vertices as

the one in Figure 2.3 (a), the polygon mesh is also called *triangulated mesh* or *triangle mesh*. Triangulated meshes are required in a lot of computer graphics applications such as the one described in this thesis.

(a) Triangle mesh (b) Quadrilateral mesh

Figure 2.3: Polygon mesh types

2.3 Normal

In the 3D space, a *normal vector* (or normal in short) to a surface at a point q is a vector perpendicular to the tangent plane to that surface at q.

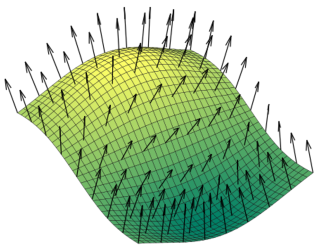


Figure 2.4: Normals vectors of a surface

Let $ax + by + cz + d = 0$ the equation of a plane P; the normal of P is the vector $v = (a, b, c)$ or $v = -(a, b, c)$ depending on the desired direction. In

the specific case of a triangle, the normal vector can also be computed as the cross product of two of its edges (keep in mind that the choice of the edges will influence the direction of the normal). The normal vector is commonly denoted with the capital letter N; sometimes it is denoted with the symbol $\hat{N}$ to explicitly indicate a *unit normal vector*. The unit normal vector (or unit normal) is obtained by normalizing the normal vector, namely dividing a non-zero vector by its norm.

Computing normals as we just told produces the same normal vector for each point lying on the entire face, for this reason, the resulting normals are also called *face normals*. A visualization of face normals is shown in Figure 2.5 where for each face of the triangle mesh, a blue line represent the normal direction of any point lying on the face itself.

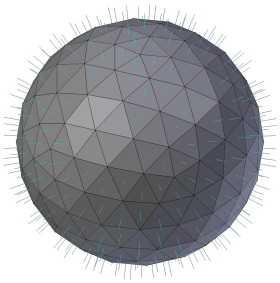


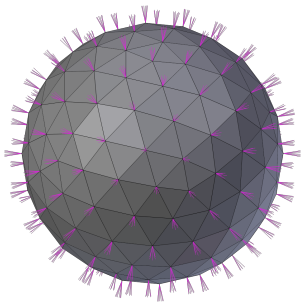
Figure 2.5: Face normals

Moreover, mesh normals can also be defined at the polygon vertices, in which case we call them vertex normals (See examples in Figure 2.6 (a) and Figure 2.6 (b)). This choice is really handy because rendering process deals with vertices so it is better to keep this information on them.

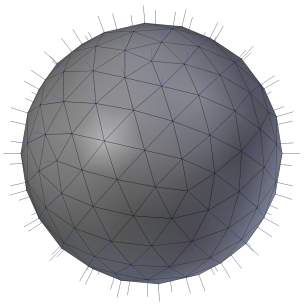
As we know, in computer graphics, objects are represented by polygons. The problem with polygon meshes is that they can't represent perfectly smooth surfaces and often curved objects have to be approximated. A good curves approximation can obviously be achieved by increasing the number of flat faces but there exist more efficient alternatives such as the smooth shading

techniques.

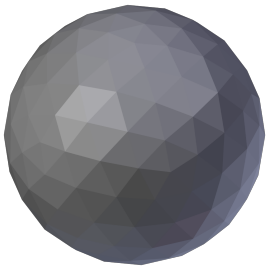
Vertex normals play an important role in how a polygon mesh looks and influence the mesh shading since the brightness of an object at a given point of its surface depends on the angle between the normal at that point and the light direction. Usually, each triangle in a polygon mesh has a set of three vertex normals (four in case of quads and so on), one per vertex.



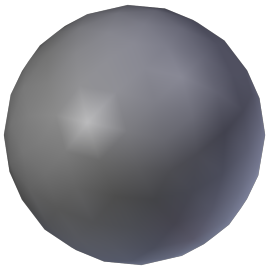
(a) Flat vertex normals



(b) Smooth vertex normals



(c) Flat shading



(d) Smooth shading

Figure 2.6: Shading examples

It is possible to simulate continuous shading across the polygons of a mesh, despite the fact that the surface is not continuous as it is built from a set of flat polygons or triangles, by linearly interpolating the vertex normals defined at each vertex.

So when we want to achieve a fake smooth surface, the vertex normals defined

on vertices that are shared by more than one triangle have to point in the same direction as shown in Figure 2.6 (b).

In Figure 2.6 (d) we can notice that the silhouette of the mesh still appears faceted, even though the surface is smooth. Unfortunately, the only solution to this problem is to subdivide the surface increasing the polygon count further.

2.4 Texture

A *texture* describes surface characteristics and appearance, as well as the set of chromatic and morphological variations possessed by any material as shown.

In computer graphics, a texture is an image that can be used to store many different kinds of data. During the rendering process, textures can be mapped to polygons to determine the appearance of them like defining colour, roughness, surface imperfections and other properties. This technique is called *texture mapping* and it allows to visualize one or more textures on the mesh surfaces.

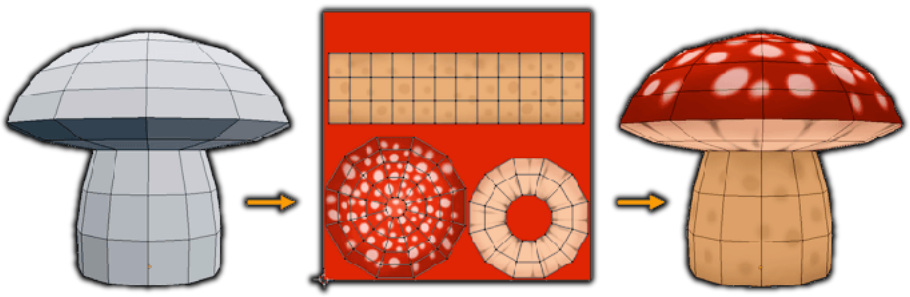


Figure 2.7: Texturing process

To map a texture over a mesh (texturing a mesh), as in Figure 2.7, each vertex needs to be associated with the 2D coordinates of the texture. These

are known as the *UV coordinates* of the vertex. Then, every mesh polygon will contain the piece of texture determined by the UVs of the vertices forming that polygon. This technique allows the simulation of photo-realistic scenes in real time without the need of complex polygonal mesh and lighting calculations.

2.5 OBJ file format

The *OBJ file format* (`.obj`) is an open-source data-format file, developed by Wavefront Technologies, to define 3D geometries. This file format represents only the essentials of a 3D geometry, defining vertex positions, UV texture coordinates, normals and the set of faces that the geometry is made up of. It is usually used to exchange data between different 3D software.

The object file structure is very simple, it does not require any external file or header at the begin to be read and usually, it starts with comment lines marked with the symbol `#`. All other lines of the file start with a keyword followed by some specific data. The keywords useful for our implementation are reported in Table 2.1.

Keyword	Syntax	Meaning
v	**v** x y z w	Vertex coordinates
vt	**vt** u v w	Texture coordinates
vn	**vn** i j k	Normals directions
f	**f** v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3	Faces

Table 2.1: Syntax and meaning of the .obj keywords

Vertex coordinates, texture coordinates and normals listed in the object file are enumerated for each keyword in order of appearance starting from one. Data

are floating point values with the exception of face values that are instead expressed as a set of integers that refer to the keyword positions: `v`, `vn` and `vt`.

2.6 Smoothing

Sometimes 3D polygon meshes appear faceted. To reduce curves and surfaces faceting, smoothing methods are used acting as a low-pass filter (LPF) that passes signals with a frequency lower than a certain cut-off frequency and attenuates signals with frequencies higher than the cut-off frequency which is assumed to be noise. The simplest smoothing algorithm with a linear complexity is the *Gaussian smoothing*.

2.6.1 Gaussian Smoothing

The basic idea of the Gaussian smoothing method is to compute the new position of each vertex adding to the old vertex position a weighted average of the distances between the vertex itself and its first order neighbours (vertices that share an edge or face with the current vertex). This simple operation must be applied iteratively a large number of times to produce significant smoothing.

Let $i*$ be the first order neighbourhood of a vertex v_i. A single Gaussian smoothing iteration is achieved at first computing for each vertex v_i a *vector average*

$$\triangle v_i = \sum_{j \in i*} w_{ij}(v_j - v_i) \tag{2.1}$$

that is computed as a weighted average of the vectors v_j-v_i where the weights w_{ij} are positive and add up to one. A common choice that produces good

results is to set w_{ij} equal to the inverse of the number of neighbours $i*$ of vertex v_i, for each element j of $i*$, otherwise a different weights can be chosen. After the vector average is computed, the new position of v_i is updated by adding to the current vertex position its corresponding *displacement vector*

$$v_i^{'} = v_i + \lambda \triangle v_i \qquad (0 < \lambda < 1) \tag{2.2}$$

computed as the product of the vector average $\triangle v_i$ and the scale factor λ.

A well-known disadvantage of the Gaussian smoothing is that when a large number of smoothing iterations are performed, the shape undergoes deformations and start to collapse to a point (Figure 2.8). This behaviour is also known as *mesh shrinkage* and a solution to this problem was proposed by Gabriel Taubin [2].

2.6.2 Taubin Smoothing

The smoothing algorithm proposed by Gabriel Taubin, which solve the problem of mesh shrinkage, consists of two consecutive Gaussian smoothing steps. At first a Gaussian smoothing step with a positive scale factor λ is applied to all the vertices, then a second Gaussian smoothing step is applied to all the vertices again, but with a negative scale factor μ, greater in magnitude than the λ scale factor $(0 < \lambda < -\mu)$.

$$v_i^{'} = v_i + \lambda \triangle v_i \tag{2.3}$$

$$v_i^{''} = v_i^{'} + \mu \triangle v_i^{'} \tag{2.4}$$

$$(0 < \lambda < -\mu < 1)$$

In order to achieve a good smoothing effect, the iterative smoothing process required in the standard Gaussian smoothing is required in Taubin smoothing

too, but this time alternating the positive and negative scale factors.

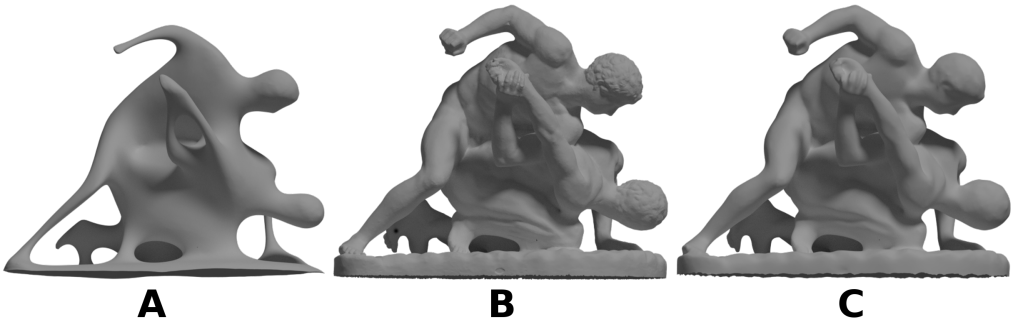


Figure 2.8: (B) The original surface. (A) The results of applying Gaussian smoothing to (B). (C) The results of applying Taubin smoothing to (B)

Chapter 3

GPU Parallel Computing

A *Graphics Processing Unit*, or simply GPU, is a particular co-processor that is specialized for fast rendering of 3D graphics images. Thanks to the development of computer graphics, photo-realistic rendering, video-game and film industry, GPUs have been improving.

What makes GPUs powerful for computer graphics is the data parallel processing capability (linear algebra matrix operations in particular). This computing capability has been exploited for purposes beyond computer graphics, thus marking the beginnings of *GPGPU: General-Purpose computing on Graphics Processing Units.* Before we dive into details, let's explain what parallel computing is.

3.1 Parallel Computing

Parallel computing consists of multiple computing resources working simultaneously to solve a computational problem. In the natural world, an infinite number of events are happening at the same time, so parallel computing is

suited for modelling and simulating natural phenomena. In addition, many problems are so large that approaching them sequentially can take too long whereas parallel computing can reduce computational time.

The benefit an algorithm can get, using Parallel Computing, can be enormous but it is not unlimited. Sooner or later there will be a point where the algorithm cannot be parallelized further. Amdahl's law [4] states that the maximum speed-up expected by parallelizing portions of an algorithm is

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{3.1}$$

where P is the fraction of the algorithm that can be parallelized and N is the number of processors over which the parallel portion of the code runs.

There are two kinds of parallelism:

- Task-based, where each unit carries out a different task;
  this is usually preferred when:
  - the problem can be divided into tasks;
  - number of tasks about equals to units;
  - units carrying out different tasks can communicate easily;
- Data-based, where all units do the same work on different subset of data;
  this is usually preferred when:
  - the work to be done is simple;
  - big data size;
  - it is easy to distribute data among units so as to minimize data

The task-based approach is widely used in all modern software running on CPUs while the Data-based is the kind of parallelism mainly used on GPUs.

3.1.1 Stream Processing

Stream processing is a programming paradigm using a simple parallel computing model. It is a form of data-based parallelism, where a series of operations is applied to each element in the data-stream, without management of the *Processing Elements* (PE). It also needs data locality property where processing an element requires only the element and, in some cases, the neighbours.

GPUs take inspiration from stream processing, indeed the taxonomy of GPUs is SIMT (Single Instruction, Multiple Threads), an execution model where SIMD (Single Instruction, Multiple Data) is combined with multi-threading. The hardware automatically manages multiple threads that all execute the same instruction (hardware-based multi-processing) and this is crucial for GPGPU.

3.2 Hardware

It is important to study the hardware implementation to better understand how GPUs achieve that parallel computational power and to get deep into the parallel programming paradigm.

GPUs have undergone large modifications during their evolution, especially due to the birth of GPGPU, but we will study some of the major features implemented nowadays to draw a clearer picture of their current state. It is worth noting that even if GPUs have evolved to support GPGPU, the main goal remains the computer graphics and that is why there are some hardware components dedicated for that purpose.

A GPU is quite a different chip compared to a conventional CPU. Designed for largely parallel problems, the hardware of a GPU has a considerable num-

ber of PEs, but there are many features implemented by CPUs that are not implemented by GPUs. Also, PEs run slower than CPU cores and with some constraints explained in the next sections.

Figure 3.1: GPU architecture of an Nvidia GeForce GTX 1080

3.2.1 Streaming Processor

A *Streaming Processor* (SP) is the PE of a GPU, contains an FPU (*Floating-point Unit*) and/or an ALU (*Arithmetic Logic Unit*), but it is different from a standard CPU core. For example, it does not contain computation registers or instruction units, therefore, it does not work alone. To better understand what SPs really are, we have to consider them as part of a streaming multiprocessor.

3.2.2 Streaming Multiprocessor

Threads are created, grouped in thread-blocks and scheduled to various Streaming Multiprocessor by a *Thread Block Scheduler* residing on the GPU.

A *Streaming Multiprocessor* (SM) is the unit responsible for managing thread-blocks execution. Each thread-block to be executed is then subdivided in thread-groups (*warps* in NVIDIA-speak, *wavefronts* in AMD-speak) with a fixed size intrinsic to the GPU.

An SM (Figure 3.2) contains groups of SPs, with some instruction units, thread-group schedulers and dispatch units. Once a thread-group is chosen by the scheduler, the dispatch units issue the thread-group to a subgroup of SPs that carry out the execution in lock-step.

It is worth noting that even if the program needs to launch a single thread, an entire thread-group will run because of the hardware architecture. In addition, SMs run independently from each other.

Each SM also has some load/store units that are responsible to execute all load and store instructions for threads, generate source and destination addresses, load and store data between registers and cache or DRAM.

Another important feature inherited from the old 3D rendering GPUs is the integration of several SFUs (*Special Function Units*) inside the MPs. These units are useful in computing graphics interpolation instructions and fast approximations of transcendental functions such as sin, cosine, reciprocal, and square root.

Figure 3.2: A Streaming Multiprocessor with 4 subgroup of 32 SPs

3.2.3 Memory hierarchy

The memory architecture of a GPU is hierarchical as in a common computer architecture but it is optimized for parallel computing.

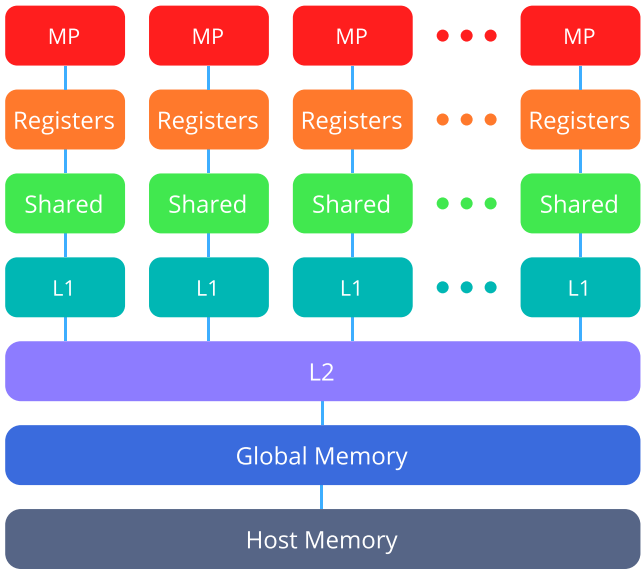


Figure 3.3: GPU Memory Hierarchy

Global memory

The *global memory* is the widest memory in a GPU, used to store data persistent during program execution. This memory is important for dedicated GPUs because, being mounted on PCIe slots, using the host memory would mean transferring data through the PCIe bus which at best is 10 times slower than the global memory, representing a bottleneck.

Technically, the global memory is a RAM, based on the GDDR (Graphics Double Data Rate) technology, with a high bandwidth and high latency that can serve data at high speed with the right access patterns. For example, if a thread-group accesses consecutive memory addresses the request is performed

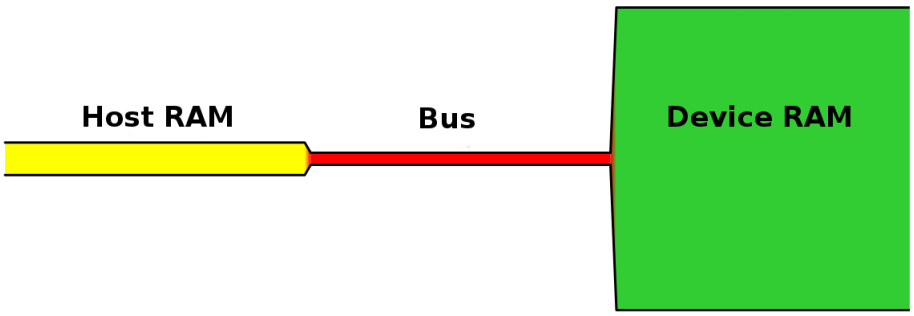


Figure 3.4: PCIe bottleneck

in one transaction. This is a best practice of GPGPU programming and it is called coalescing memory access patterns.

The GPU can access it directly but, even if this memory is very fast compared to a DDR RAM, lots of read/write operations on it can decrease performance. To overcome this problem GPUs may have an L2 cache, common to all MPs, that is used transparently to the programmer. There may be also an L1 cache on each SM used to improve memory access speed and for register spills to reduce performance drops in case of excessive register usage.

Local Memory

The *local memory*, also called *shared memory*, is a programmable on-chip memory contained in each SM that is much faster than global memory. In fact, local memory latency can be roughly 100x lower than global memory latency. It is allocated per thread-block, so all the contained threads have access to the same local memory, therefore they can access data loaded from global to local memory by other threads within the same thread-block. This capability is very important to create high-performance cooperative parallel algorithms. This memory is organized in different banks, which are served by a bus allowing very fast access if each bank serves one thread. Otherwise, the accesses

are serialized, and the access time is multiplied by the number of threads that access the bank. This behaviour is known as *bank conflict* and it should be avoided.

Registers

The *registers* of a GPU reside on each MP rather than on each SP so they are not dedicated to a PE in particular, but can be assigned to a thread when its execution starts. GPUs reach their performance by exploiting thread level parallelism, so every thread needs to store its own set of registers.

Therefore the register file needs to be relatively large, in order to allow many threads to be active at the same time. There is an architecture-dependent upper limit to the number of registers that can be allocated by a single thread, so if threads use more private variables that can be held in the given limit, spilling will occur, firstly to the L1 cache if available, and if needed, to the global memory, causing big drops in performance.

3.3 GPGPU

Originally, video cards had a fixed rendering pipeline and they could be used by rendering APIs (OpenGL, Direct3D) with the goal of hiding low-level hardware details. This made the APIs simple but not very flexible. To overcome this problem, with the growth of video games, GPUs had started to appear and a programmable pipeline was introduced in the existing APIs. This new feature allows writing programs (*shaders*) that execute some rendering stages. With the increasing computing power of GPUs, users had started to write shaders for purposes beyond rendering, encoding general purpose algorithms in 3D scenes to render. However, shaders were designed for rendering and were therefore

not very flexible for general computing. Between 2007 and 2008, new GPUs and APIs (CUDA, OpenCL) started to appear, allowing programs to perform general purpose calculations on GPU, even outside of the rendering pipeline. This marks the beginning of true GPGPU.

3.3.1 GPGPU architecture

GPGPU is based on the stream processing paradigm and the fundamental concepts are independent of the specific hardware (and thus common to CUDA and OpenCL), aside from semantic differences.

A GPU is seen as an independent entity called *device* that is controlled by the CPU considered *host*. Each device may have its own memory and its own compute units.

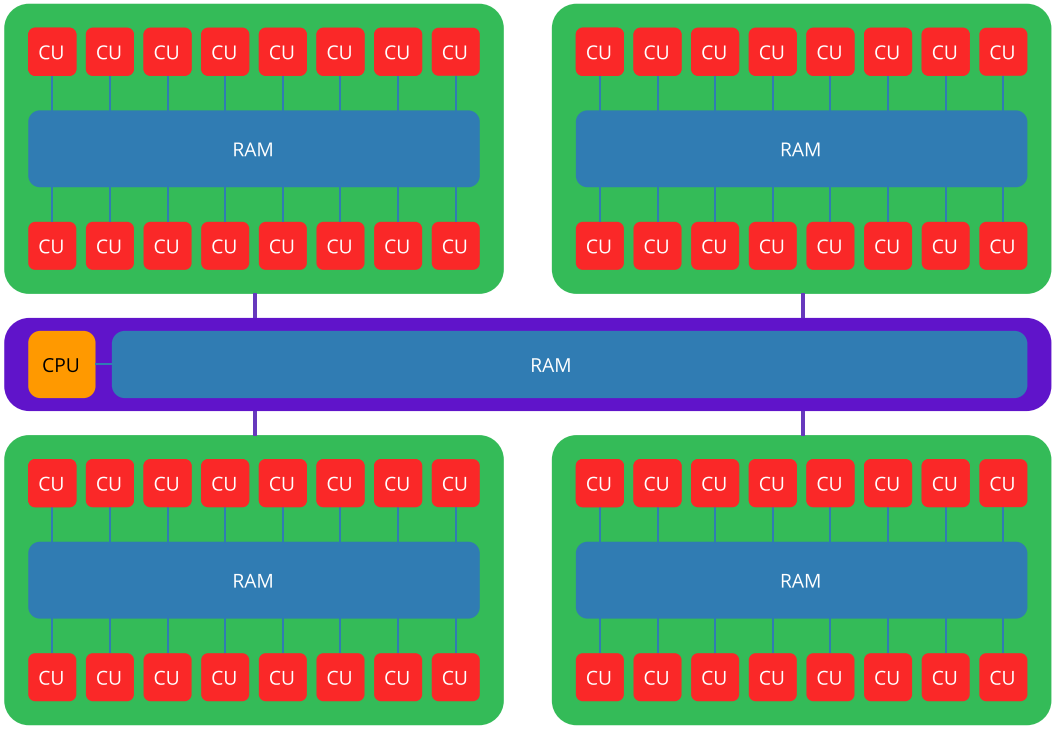


Figure 3.5: Host and devices

The source code of a software that uses a GPGPU API consists of a host and a device part. The host part, running on the CPU as a common program, initializes the framework, loads and prepares data that needs to be processed and controls the device asynchronously. The device part consists of functions, called kernels, executed by a device in a particular way.

When a kernel is invoked by the host, specifying the launching grid size and all the parameters for the function, the device creates as many instances of the kernel as required, concurrently executing them on the PEs over a subset of the data-stream.

The launch grid is the collection of all the kernel instances issued in one invocation of a kernel. These instances are called work-items and they correspond to hardware threads. Each work-item, as we said earlier, contains the execution state, the memory context and the registers on which they work.

The host can also specify the size of the work-groups, groups of work-items (not to be confused with the threads group) corresponding to the hardware thread-blocks, guaranteed to always execute in the same MP so that they can cooperate synchronizing and exchanging information using local memory.

3.4 Divergence

As we said earlier, when an MP executes thread-group, all the threads are issued to the SPs to run the same instructions. Branch divergence occurs when work-items belonging to the same subgroup need to take different execution paths at a given conditional statement.

Since the subgroup proceeds in locksteps for all intents and purposes, in such a situation the hardware must mask the work-items not satisfying either branch,

execute one side of the branch, then invert the mask and execute the other side of the branch: the total runtime cost is then the sum of the runtimes of each branch. At high doses, these divergences slow down the execution of the code and should be avoided.

If the work-items taking different execution paths belong to separate subgroups, this cost is not incurred, because separate subgroups can execute concurrently on different code paths, leading to an overall runtime cost equal to that of the longer branch.

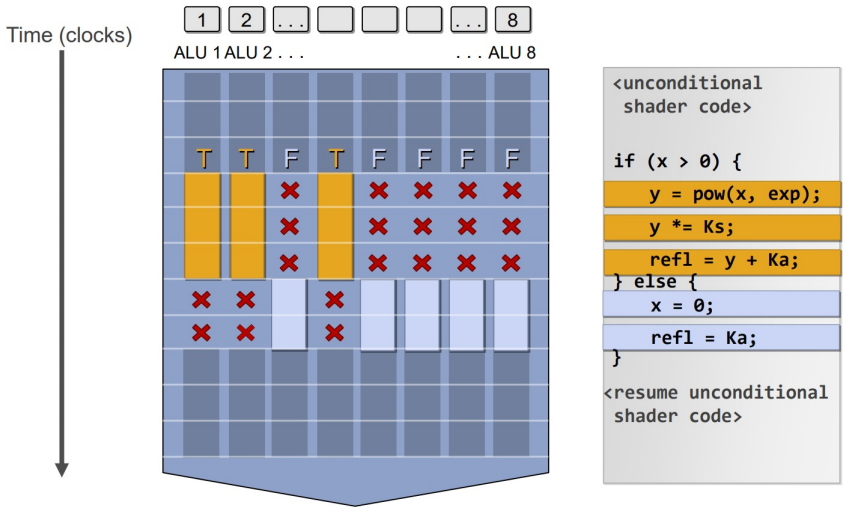


Figure 3.6: Conditional statement on a GPU

Chapter 4

Rendering

Rendering is one of the major sub-topics of 3D computer graphics which consists of generating images from 3D scenes using computer programs. This process is heavily used in films, video-games, architecture and scientific visualizations.

Rendering may be off-line or real-time. The off-line option is widely used in films industry due to the fact that there isn't any time constraint leading to high-quality imagery using computationally intensive processes. Real-time rendering, instead, is often used for 3D video games and interactive simulations which rely on faster algorithms and graphics cards with 3D hardware accelerators.

4.1 Techniques

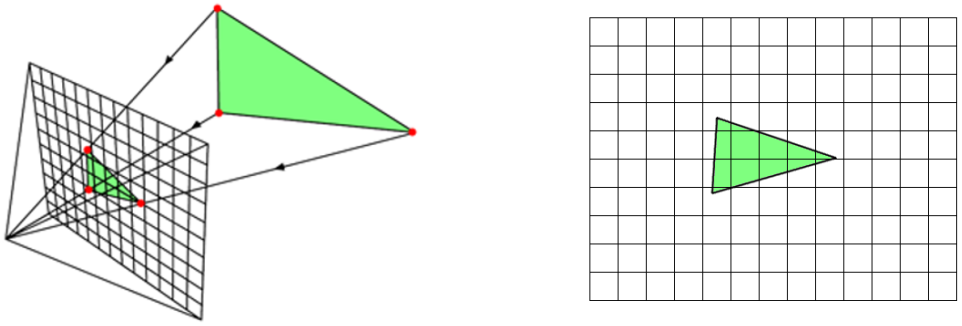
To render a scene we have to convert a 3D scene to a 2D image. To do that we first need a camera placed in the space, which defines a *point of view*. As the images we want to obtain are rectangular, the camera also needs to define an

image plane in the space which will be the projection plane for the 3D scene. In addition, an image is a grid of pixels so the rendering algorithm has to know the resolution of the resulting image. Once all of these parameters are defined, there are different rendering techniques as described below.

4.1.1 Rasterization

Rasterization is one of the most common rendering techniques due to the speed of the algorithm that allows rendering scenes in real-time. GPUs are optimized to perform rasterization.

The algorithm consists of iterating through all the triangles of the scene and projecting them to the view plane by transforming vertices into corresponding 2-dimensional points (Figure 4.1a) and filling in the portion of the view plane that the new triangles are covering (Figure 4.1b).

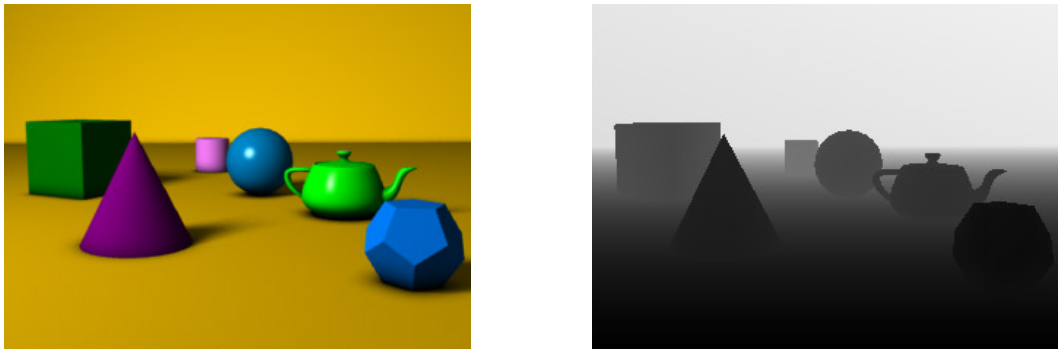


(a) Vertices projection (b) Pixels coverage

Figure 4.1: Rasterization steps

With only one triangle, rasterization is straightforward but a scene is composed by an enormous amount of triangles and this algorithm suffers from

the *visibility problem*, which is the problem of deciding which triangles of a rendered scene are visible and which are occluded by others. The problem is solved by using the *Z-buffer* (Figure 4.2b) which is a grayscale image with the same resolution of the image to render. When the algorithm loops over the pixels covered by a triangle, for each of these it calculates the distance from the point of view to the polygon and compares it with the one contained in the Z-buffer. If the distance is lower it means that the part of the polygon covering that pixel is in front of the one that has written the Z-buffer before and it can override the Z-buffer as well as the output image buffer. Vice versa, if the distance is greater than the one currently contains in the Z-buffer, that the part of the polygon is simply discarded because it is occluded.



(a) Image buffer (b) Z-buffer

Figure 4.2: An image buffer with the corresponding Z-buffer

4.1.2 Ray casting

Ray casting is a simple rendering algorithm that uses the concept of rays. Since simulating light rays could be too expensive and, in some cases, useless due to rays not ending up into the camera, this algorithm traces the rays back out from the camera to the scene. More precisely, it consists of casting one ray through every pixel in our image grid. Then, it checks each ray against every single object and looks for the first intersection. The simplicity

of ray casting comes from the fact that once the ray hit a surface, it computes the colour without recursively tracing additional rays, but this means that it cannot achieve accurately reflections, refractions, and shadows. Instead, this process automatically solves the visibility problem that plays in the rasterization technique.

Although the mathematics for finding intersections is relatively simple, the big problem of ray casting is that the algorithm may potentially have to check every ray against every polygon of the scene. There are solutions to decrease the number of checks using particular data structures, for example, octree, but this still requires a preprocessing pass.

4.1.3 Ray tracing

Ray tracing is based on the idea of tracing rays from the camera to the scene to calculates the path followed by the incoming light ray. In contrast with ray casting, ray tracing recursively traces additional rays to take in account the bounces of light happening in real world, thus automatically solving the shadows, reflections and refractions problem as well as the visibility problem like in ray casting. Thanks to this, the technique is capable of rendering photorealistic images, but at a greater computational cost. This makes ray tracing best suited for off-line rendering where time is not a constraint.

The algorithm consists of casting a ray through every pixel in our image grid called primary ray. Then, when a ray hits a surface, it traces three other new types of rays (Figure 4.3): reflection, refraction, and shadow. A reflection ray is traced using the reflection law to simulate specular light of the material. If the surface is transparent, refraction ray simulates ray that goes through the material using the index of refraction(IOR) determining the new direction. A shadow ray is traced to each light of the scene to simulate shadows. If an

opaque object is hit by the ray before reaching the light it means that the light source is occluded and the surface is in shadow. When the reflection ray and the refraction ray hits another surface, we start again recursively.

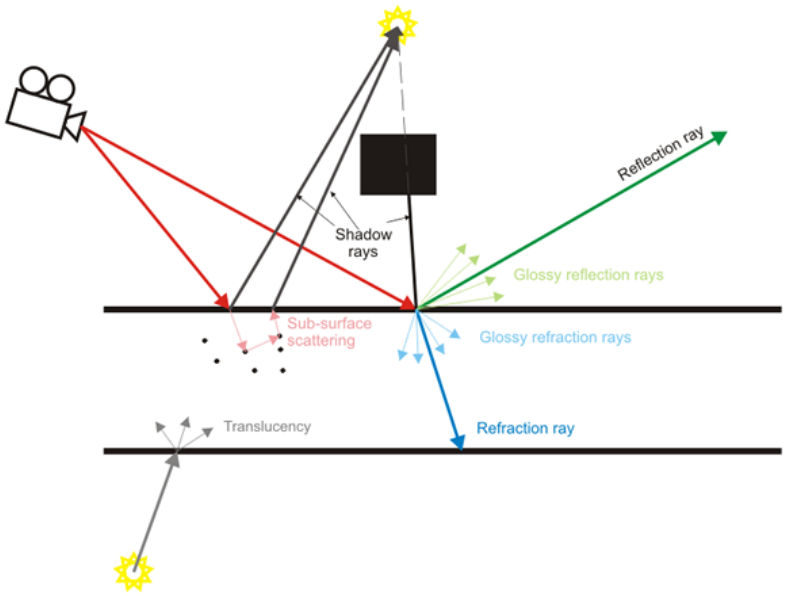


Figure 4.3: Ray Tracing

4.2 Photorealism

We now have a good picture of how a computer can generate an image starting from a 3D scene, but how do we get it to generate a photorealistic rendering? How to simulate good shadows, reflections and refractions? Using mathematics and physics we can understand the laws of light, but simulating them is a heavy task. Indeed, light is a complex phenomenon described as waves by electromagnetism, it has also particles properties in quantum mechanics. As a result, different models have been created to describe the behaviour of light. In physics, radiometry is the science which deals with measurement, detection and analysis of the optical portion of the electromagnetic spectrum, including

visible light. It is important to study light's interaction at the size of wavelength, but regarding rendering, we can do without it because it would be too expensive. In addition, we don't really care about all the light rays present in our scene. Instead, we only need to simulate rays that will end up in the camera.

4.3 Rendering equation

In computer graphics, the rendering equation is an integral equation, presented by James Kajiya [9], that gives the formula to calculate the radiance leaving a point. In the real world, light bounces off everything and this is an important factor to consider in a rendering process. This phenomenon is called global illumination (or indirect illumination) and makes the rendering equation massive to calculate, but achieving realism. The formula is described below:

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_0, \lambda, t) + \int_\Omega f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t)(\omega_i \cdot n) \mathrm{d}\omega_i \tag{4.1}$$

where

- $L_o(x, \omega_o, \lambda, t)$ is the total spectral radiance of wavelength λ directed outward along direction ω_o at time t, from a particular position x;
- x is the location in space we want to calculate the total spectral radiance;
- ω_o is the direction of the outgoing light;
- λ is a particular wavelength of light;
- t is time;
- $L_e(x, \omega_0, \lambda, t)$ is emitted spectral radiance;
- $\int_\Omega ... \mathrm{d}\omega_i$ is an integral over Ω;

- Ω is the unit hemisphere centred around n containing all possible values for ω_i;
- $f_r(x, \omega_i, \omega_o, \lambda, t)$ is the bidirectional reflectance distribution function, the proportion of light reflected from ω_i to ω_o at position x, time t, and at wavelength λ;
- ω_i is the negative direction of the incoming light;
- $L_i(x, \omega_i, \lambda, t)$ is the spectral radiance of wavelength λ, coming inward toward x from direction ω_i at time t;
- n is the surface normal at x;
- $\omega_i \cdot n$ is the weakening factor due to the incident angle, as the light flux depends on it.

The various realistic rendering techniques in computer graphics attempt to solve or approximate this equation. Kajiya also introduced path tracing (Figure 4.4), an approach that solves the integral part by using Monte Carlo integration, and calculates the light bounces by using a finite recursion. This is usually used in off-line rendering but it is too expensive for real-time.

Another important aspect of the equation that has to be taken into account is the wavelength of light. The human eye can detect light in the range 380-780nm, a portion of the electromagnetic spectrum known as visible light, which we perceive as colour. To calculate the equation for each wavelength in this range involves other integrals and it is slightly impractical.

In computer graphics, we usually represent the radiant flux using a colour model, a mathematical model to encode colours in a numeric format, using tuples of numbers. The most common colour model is the RGB colour model and it is the one used in this thesis. This encoding suffers from information loss, but this is a sensible compromise for visual aspects.

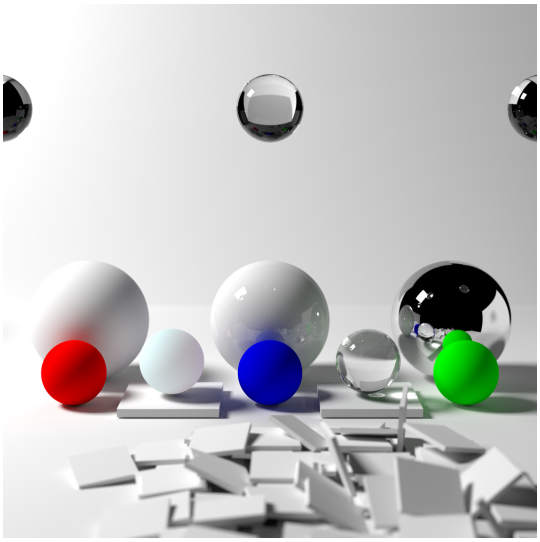


Figure 4.4: Path Tracing

The rendering equation gives us a useful instrument to achieve realism, but we still have to define a *Bidirectional Reflectance Distribution Function* (BRDF), which is a function that describes the reflectance properties of an opaque surface. This function also needs to well approximate the reflection phenomena happening in the real world. This is done by *Physically Based Rendering*, a collection of physically based functions and techniques used in modern rendering to achieve photorealism.

4.4 Physically Based Rendering

In the 1970s, computer graphics was focused on solving fundamental problems like geometric representations and visibility algorithms. At that time, computing capability was limited, thus making physical simulations for rendering infeasible.

Since 1980, as computers have started to become more powerful, thanks to hardware acceleration, graphics researchers started to consider physically based

approaches. An important advancement was Cook and Torrance's reflection model (1981, 1982), which introduced the microfacet reflection model to graphics. This model is still used nowadays and is also adopted for this thesis. Among other contributions, this model was the first to render metal surfaces accurately.

Physically Based Rendering (PBR) is an approximation of how the reality works, based on physics principles. This gives the ability to create photorealistic imagery. Moreover, for artists, PBR is an enormous improvement to work-flow and quality because they don't have to tweak parameters based on different light conditions to make materials looking photo-realistic. In addition, a very small set of material parameters can be sophisticated enough for both off-line and real-time feature film rendering.

4.4.1 Theory

PBR is based on the ray optics that treat the light as multiple rays. When a light ray hits a surface, two phenomena can happen: the light ray can be refracted or reflected.

Refraction is the phenomenon that happens when the light ray travels through mediums with a different index of refraction (IOR). For example, a light ray, travelling in the air (with a constant IOR), hitting a glass will bend due to the change of the IOR transmission medium.

Reflection is the phenomenon that happens when the light ray bounces off of the surface. Two types of reflection exist as we can see in figure 4.5: the diffuse reflection due to refracted rays going out the surface and the specular reflection due to rays perfectly bouncing on the surface, following the law of reflection which states that the angle of reflection is equal to the angle of incidence. We

will only consider the reflection phenomena in this thesis.

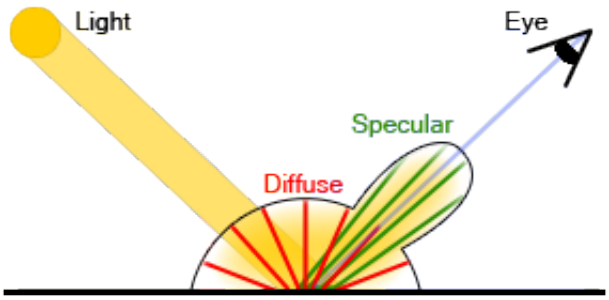


Figure 4.5: Diffuse and specular reflection

Once we know this, we can start modelling the BRDF as follow:

$$f_r = k_d f_{\text{diffuse}} + k_s f_{\text{specular}} \tag{4.2}$$

where k_d is the percentage of diffuse reflected light while k_s is the percentage of specular reflected light.

A physically based lighting model has to satisfy three conditions:

- Be energy conserving;
- Be based on the microfacet surface model;
- Use a physically based BRDF.

We will talk in more details about these three key features in the next sections.

4.4.2 Energy conservation

In the real word, outgoing light energy from a microfacet cannot exceed the incoming one unless the surface is emissive. The law of conservation of energy states that the total energy of an isolated system remains constant thus the

energy can neither be created nor destroyed. We have to consider this law for PBR. Indeed if a ray is specular reflected by a surface, it will no longer be refracted, so the two types of energy are mutually exclusive. To accomplish that, looking at the equation 4.2, $k_d + k_s \leq 1$.

4.4.3 The microfacet model

As PBR technique tries to simulate physics we have to consider how surfaces physically are. In the real word, flat surfaces may not be completely smooth on a microscopic level so it is important to consider this when rendering. The microfacet model describes that any surface at a microscopic scale is made up of tiny little polygons called microfacets. The alignment of these microfacets describe the roughness of the surface: chaotically alignment means high roughness and vice versa. Speaking about rendering, surfaces are subdivided into fragments (portions of a surface covering a pixel). Unfortunately, microfacets are smaller than fragments, therefore we have to statistically approximate how the light bounces off all the microfacets belonging to that fragment.

Figure 4.6: Roughness of a surface

4.4.4 BRDF

A BRDF (Bidirectional reflectance distribution function) is a function that describes the reflectance properties of an opaque surface. As we mentioned above in the rendering equation, this function takes an incoming light direction,

an outgoing direction, a point on the surface, and returns the reflected radiance exiting along the outgoing direction. In the equation 4.2 we have seen that this function can be split into two parts. In this thesis we will use the Lambert BRDF for the diffuse part and the Cook-Torrance BRDF for the specular part as follow:

$$f_r = k_d f_{lambert} + f_{cook-torrance} \tag{4.3}$$

It is worth noting that the k_s factor is already contained in $f_{cook-torrance}$.

Lambertian BRDF

Lambertian BRDF reflects radiance equally in all directions thus it does not depend on the location of the observer.

$$f_{lambert} = \frac{\text{diffuse}}{\pi} \tag{4.4}$$

where diffuse is the albedo of the material. There are no Lambertian surfaces in nature that means that k_d will never be 1.

Specular BRDF

The Cook-Torrance microfacet specular BRDF is:

$$f_{cook-torrance} = \frac{DFG}{4(n \cdot l)(n \cdot v)} \tag{4.5}$$

where l is the light direction, v is the view direction, n is the normal, F is the Fresnel term, G is the Geometry function, and D is the normal distribution function (NDF).

Normal distribution function

Specular reflections come from those microfacets that are oriented towards a vector that sits halfway between the light vector l and the view vector v.

This particular vector is called halfway vector, defined as $h = \frac{l+v}{|l+v|}$. The normal or facets distribution function calculates the density of microfacets oriented towards the halfway vector. The one that we have chosen for the implementation is discussed in "Microfacet models for refraction through rough surfaces" [15] called GGX distribution:

$$D_{GGX} = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \tag{4.6}$$

with α remapped by Disney [13] as $\alpha = \text{roughness}^2$.

Geometry function

The geometry function describes the attenuation of the light due to the roughness of the surface. This is important to take into account in the BRDF because microfacets could occlude or shadow each other. A possible solution is to evaluate how many facets get directly hit by a light ray and how many of them are visible by the observer. This is the Smith's method described by Smith [18] and it consists of splitting the function into two components: light and view.

To evaluate both light and view component we can use a function that calculates how many facets get directly hit by a certain vector s. We will adopt the GGX and Schlick-Beckmann approximation known as Schlick-GGX.

$$G_{SchlickGGX}(s, k) = \frac{n \cdot s}{(n \cdot s)(1 - k) + k} \tag{4.7}$$

where k is a remapping of the roughness. Then we can calculate the geometry factor as follows:

$$G_{SmithGGX} = G_{SchlickGGX}(l, k)G_{SchlickGGX}(v, k) \tag{4.8}$$

Fresnel equation

The Fresnel phenomenon is related to the physics of the electromagnetic waves and how they are either reflected, transmitted or absorbed. For the sake of the BRDF, the Fresnel equation, given an angle of incident θ_i and a factor $F_0 \in [0, 1]$ that is the base reflectivity of the surface, calculates the ratio of light rays that get reflected over the light rays that get refracted. The typical choice is the Schlick's approximation [16] because it is very close to the reality.

$$F_{Schlick} = F_0 + (1 - F_0)(1 - cos\theta_i)^5 \tag{4.9}$$

This function not only is contained in the specular part of the BRDF, but it estimates the k_s and k_d factors.

The Fresnel equation also shows that when a light ray hits a surface perpendicularly, thus with an angle of incidence $\theta_i = 0$, the base reflectivity F_0 is used as reflectivity factor, but when the angle of incidence tends to 90 degrees, the surface tends to be completely reflective. It is also important to say that since specular reflection comes from microfacets with the normal oriented towards the halfway vector, θ_i should be interpreted as the angle between the light vector and the halfway vector, and not the angle of incidence of the light vector with the surface normal.

4.5 Metals and dielectrics

Since light is composed by electromagnetic waves, the optical properties of a material are actually related with the electric properties. We can group materials in:

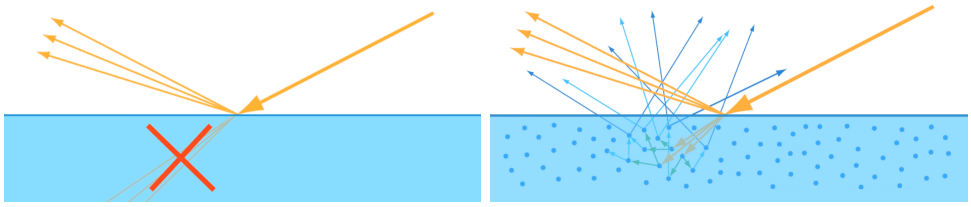
- Metals;

- Dielectrics;
- Semiconductors.

We can ignore the latter category for the lack of its presence in CGI.

Metals immediately absorb all refracted light (Figure 4.7a) and thanks to the high base reflectivity they reflect a big part of the incoming light. This means that metals don't have diffuse reflection due to the absorptivity of refracted light.

Dielectrics, instead, can either absorb, transmit or scatter refracted light (Figure 4.7b). Moreover, common dielectric materials (plastic, glass, etc.) have a relatively low absorptivity, therefore a big part of the light is scattered back out of the surface creating diffuse reflection.



(a) Metals (b) Dielectrics

Figure 4.7: Light scattering

Another important difference between metal and dielectric materials can be observed in the specular reflection. While dielectrics reflect all wavelengths roughly equally and thus have white reflections, conductors reflect wavelengths differently, resulting in coloured reflections. In our case, this is traduced in achromatic F_0 for dielectrics and chromatic for metals.

4.6 PBR Texture Maps

There are two workflows to define PBR materials, the Metal/Roughness and the Specular/Glossines workflow. The most used is the first one and we will use this for our implementation. In the next sections, we will describe all the texture maps (Figure 4.8) of the chosen workflow in details.

(a) Albedo (b) Metallic (c) Roughness (d) AO (e) Normal

Figure 4.8: Texture maps of Dwarven Drakefire Pistol by Teliri

Albedo

Albedo map is an RGB image that encodes two types of information: for dielectrics surfaces, it contains the diffuse reflectance colour, while for metallic surfaces it indicates the base reflectivity F_0 of each wavelength of the RGB channels. Unlike the diffuse map used in the past, no shadows or highlights are supposed to be present in the albedo map thus all lighting information is extracted from this texture map.

Metallic

The metallic map defines which areas of the surface are metal or dielectric. It is a one channel map and should be only black and white (dielectric or metal respectively) with some exception about using grey value for dielectric (e.g.

dirt, rust) occluding a metal underneath. As we said before, this is strictly used in combination with the albedo map to define metallic surfaces.

Roughness

The roughness map controls the surface's roughness. This map is really important in the PBR workflow because of the presence of microfacets. In contrast to smooth surfaces, rough surfaces reflect light chaotically in all directions, thus the roughness controls the appearance of reflection and how blurry or sharp it is. This map is usually a one channel map stored in linear space.

Ambient Occlusion

The ambient occlusion (AO) map describes how much each point on the surface can be reached by ambient light. In the case of real-time rendering, AO is usually baked from the scene or from the 3D model itself because it requires lots of sampling to calculate and we cannot afford it in real-time. In the past, AO maps used to be encoded onto the diffuse map, but in PBR, since we threat environment light separately, we need this information on its own so that we only occlude ambient light excluding direct lights.

Normal

Normal maps are special textures used to encode surface detail like scratches or grooves. Thanks to this type of texture map it is possible to reduce the number of polygons of a 3D model without sacrificing details. Each pixel in a normal map is a vector that represents a deviation of the surface direction thus it is still important to have face normals to use normal maps. Then, this deviation is taken in account by shaders to calculate lights reflections.

Chapter 5

Implementation

In this chapter, we will discuss the details of the presented software, and how it implements the theory introduced above. The software consists of a Real-time PBR Engine that offers tools, interfaces and classes to compose and render a scene. This engine is also oriented to visualize GPGPU algorithms by sharing data between rendering and computing parts, therefore we will explain how the GPGPU implementation of Taubin smoothing, discussed in details by Riccardo Torrisi, can be visualized in Real-time.

The Real-time PBR Engine is written in the C++ programming language using the OpenGL API (v 4.6.0); the GPGPU Taubin smoothing algorithm is based on OpenCL (v 1.2). In OpenGL, shaders are written in a high-level language known as GLSL (OpenGL Shading Language), which is based on the C programming language, while in OpenCL, the programming language that is used to write compute kernels is called OpenCL C and is based on C as well.

5.1 Engine structure

The rendering engine structure is implemented following the object-oriented programming (OOP) and its two main abstractions are Actor and Component.

- **Actor**
  any object which takes part of the scene and has a transform is an actor, even the scene itself. Actors can be nested, creating the hierarchy of the scene. In addition, they can contain a collection of Components that define their behaviour; for example, how actors move, how they are rendered and in particular how it can be affected by a GPGPU algorithm.

- **Component**
  it is the base class from which all the actor components derive from. Each component defines a reusable behaviour which can be added to different types of actors. These are generally used to render a mesh, response to inputs or control how actors move through the world. Every component implements two functions, `Start` and `Update`. The first one is called by the engine once the component has been activated and the scene is running, just before the `Update` methods is called. The second one is called every frame and it is used to implement the behaviour logic.

There are other core classes in the engine such as:

- **Texture**
  each instance of this class is associated with a texture image that is used to compose one or more materials. It has the responsibility to load the texture into RAM and then to create the OpenGL texture object by calling `glGenTextures` and uploading the texture data to the device RAM with `glTexImage2D`;

- **Material**
  it defines a set of parameters describing the surface properties of an object. This is used as an input for the mesh renderer to bind the material textures for the render pass;

- **Shader**
  it is responsible to load and compile the shader source. Each instance creates and manages the status of a shader program and it can be used to render meshes by activating it.

- **Mesh**
  it loads the mesh geometry from the file, creates and holds the relative OpenGL vertex buffers and their attributes buffers used by the mesh renderer component. This class is also important for sharing these buffers with other algorithms beyond rendering; for example, this is used by the mesh smoothing component to perform the smoothing algorithm.

5.2 Camera

The camera is a component that provides essentials information to render the scene, such as view and projection matrices, using the location and rotation of the owning actor. These matrices define a point of view and a projection method to convert the 3D scene to a 2D image.

5.2.1 View matrix

If you want to view the scene from a different point with a different angle, you can think about moving the camera but, while practical in real life, this is not what happens in computer graphics. The camera in OpenGL cannot

move and is located at (0,0,0) facing the negative Z direction, so to simulate a camera transformation, we transform the world with the inverse of that transformation. This is done by creating a view matrix, which is the inverse of the camera transformation matrix, and multiplying it for each vertex of the scene. This may seem very expensive at first glance but the matrix is calculated once every frame by the CPU and it is then multiplied by vertices being processed on the GPU. This way, rasterization can perform projections without worrying about camera transformations.

5.2.2 Projection matrix

When rendering a scene we need to transform the entire 3D scene in a 2D image, thus we need a method for mapping three-dimensional points to a two-dimensional plane. This process is called 3D projection and it is shown in Figure 5.1.

Orthographic projection

The orthographic projection is a type of 3D projection where an object is drawn on the screen using parallel lines to project its outline on to the view plane. All the lines are orthogonal to the projection plane. This type of projection is not used for realistic rendering but it is useful for 3D visualization as objects always appear with their real dimensions, regardless of the distance from the camera. This means that parallel lines are still parallel in the screen.

Perspective projection

In the real world, a camera, as human eyes, views things in a *perspective* way. Perspective refers to the concept that objects in the distance appear smaller

than objects closer to the camera. It also means that if you are sitting in the middle of a straight road, you actually see the borders of the road as two converging lines. With perspective, the output image looks more real as it seems that a real camera is filming the scene.

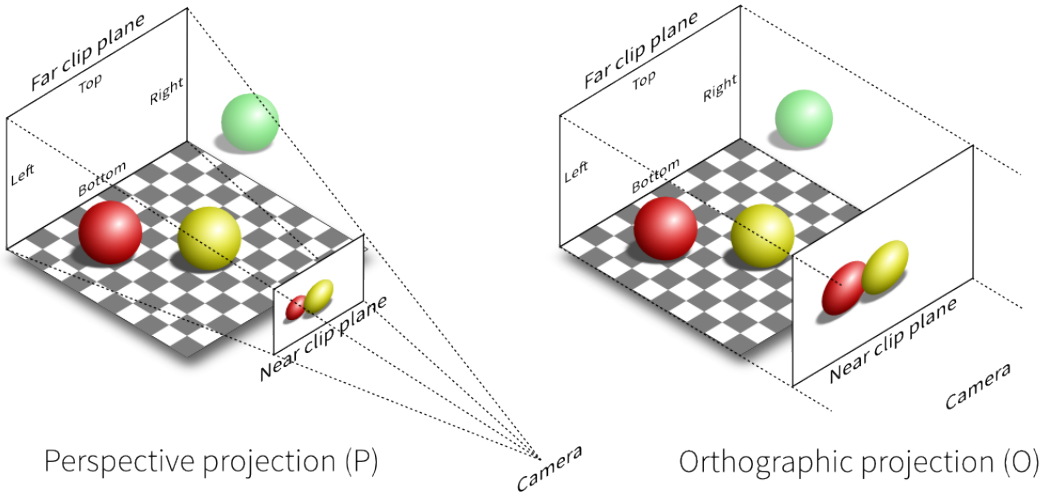


Figure 5.1: A scene example projected in orthographic and perspective mode

5.3 Mesh Renderer

The Mesh Renderer is a component that renders a mesh using the transform of the owning actor. In the `Update` function, it selects the vertex buffer of the owning mesh and the material textures to use, loads the transformation, camera and view matrix, and renders the geometry to the screen.

5.4 Shaders

A Shader is a user-defined program designed to customize the execution of some stages of the rendering pipeline. The rendering pipeline is composed by certain

sections to be programmable as shown in Figure 5.2. Each of these sections, or stages, represents a particular type of programmable processing. Each stage has a set of inputs and outputs, which are passed from prior stages and on to subsequent stages (whether programmable or not). In OpenGL, shaders are written in GLSL and compiled at runtime by the shader component of the engine. For the rendering engine, we have programmed vertex and fragment shaders.

Another important aspect of shaders is that their source is separated from the host source and it is compiled at run-time to allow platform portability.

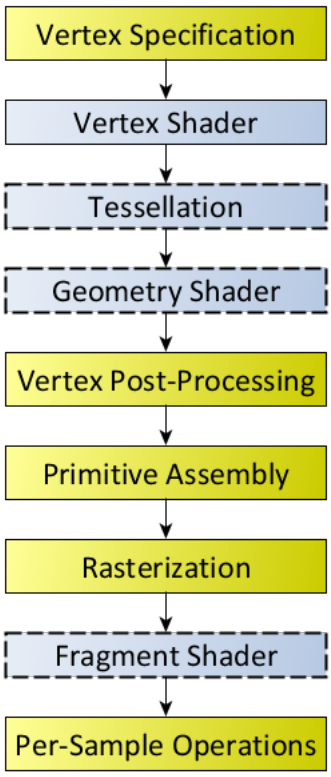


Figure 5.2: A common rendering pipeline where the blue stages are programmable while the yellow ones are not

5.4.1 Vertex shader

The vertex of a mesh is made up of parameters like position, normal, UV that are passed to the vertex shader. When loading the mesh into the GPU, it generates a vertex stream, an ordered sequence of vertices to be consumed.

The vertex shader is the program responsible for handling the processing of individual vertices. For each vertex in the stream, the GPU creates an instance of the vertex shader executed concurrently. The instance receives a single vertex composed by the set of parameters from the vertex stream, processes it and generates the output for the next stages.

This stage, together with the others, can also receive uniform parameters specified by the user of the shader program that cannot change during the rendering call (e.g transformation, projection and View matrices). Vertex shaders are usually used to perform transformations to the geometry and provide parameters for the fragment shader.

5.4.2 Fragment shader

The rendering pipeline contains a fixed stage, called rasterization (where the name of the rendering technique comes). In this stage, for each sample of the pixels covered by a polygon, a fragment is generated.

A Fragment Shader is the shader stage dedicated to process a generated fragment by the rasterization and produce a set of colours and a single depth value for the specific pixel. The inputs to the fragment shader are interpolated across the surface of the primitive. The output is a depth value, and different colour values to be potentially written to the current frame buffer.

Fragment shaders also have the possibility to execute the discard command

which causes the fragment's output values to be discarded. This stage is where lights, shadows, reflections calculation using the material textures take place so where the physically based computation is carried out.

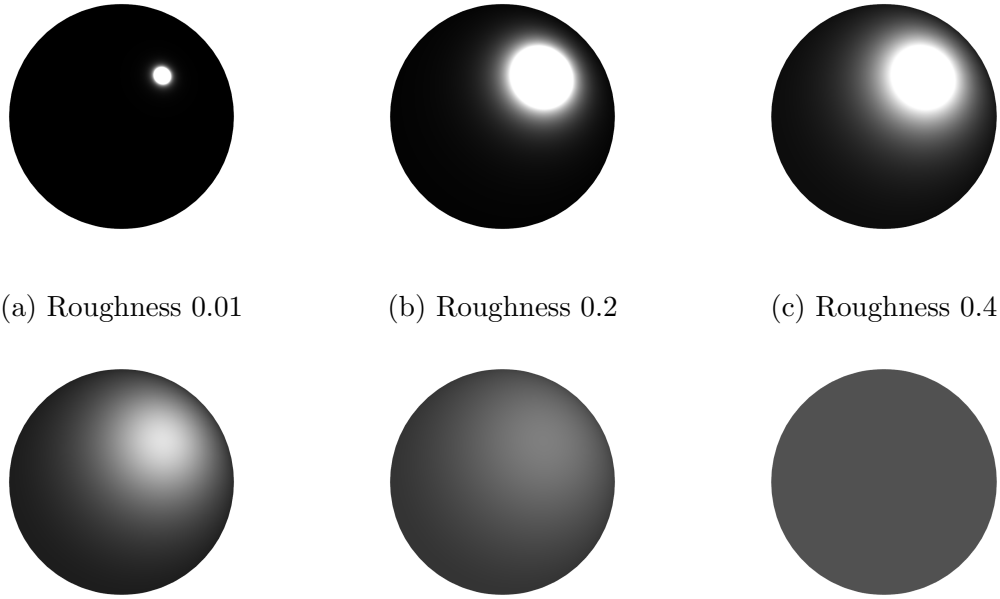
5.5 PBR Shader

In this section we will see the implementation and the visualization of the PBR functions described above and we will discuss the visual results.

5.5.1 Distribution GGX

In figure 5.3 we can see that when the sphere has a low roughness value (thus the surface is smooth), there is a high amount of microfacets aligned to halfway vectors. Vice versa, a high roughness value means that the microfacets are aligned in much more random directions, thus a large portion of the surface is likely to have micro-facets aligned to halfway vector but less concentrated.

```
float DistributionGGX(vec3 N, vec3 H, float Roughness)
{
    float alpha = Roughness * Roughness;
    float alpha2 = alpha * alpha;
    float NoH = max(dot(N, H), 0.0);
    float NoH2 = NoH * NoH;
    float Dpart = NoH2 * (alpha2 - 1.0) + 1.0;
    return alpha2 / PI * Dpart * Dpart;
}
```



(a) Roughness 0.01 (b) Roughness 0.2 (c) Roughness 0.4

(d) Roughness 0.6 (e) Roughness 0.8 (f) Roughness 1.0

Figure 5.3: Distribution GGX visual results of a sphere

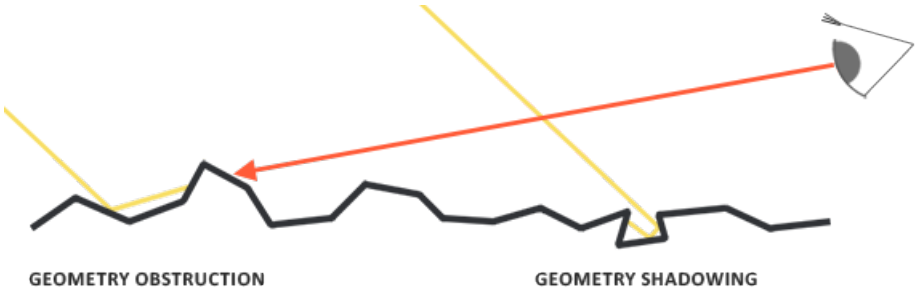


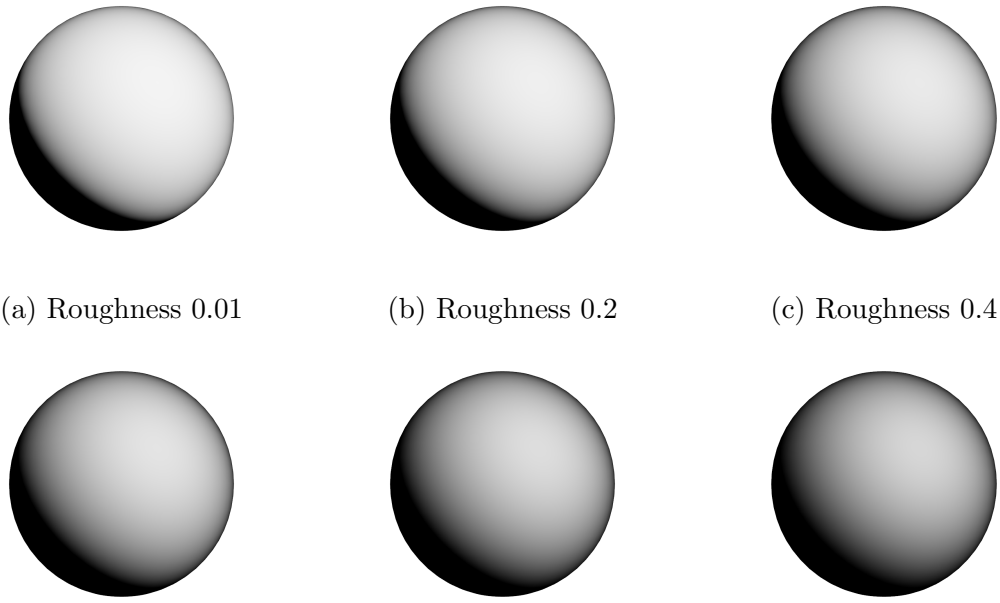
Figure 5.4: Geometry obstruction and shadowing

5.5.2 Geometry Smith

As we can see in figure 5.4, when light rays hit a surface, the roughness, at the microscopic level, can occlude part of them causing the eyes of the observer to never see those rays. This is taken into consideration by using the Geometry Schlick GGX function. In figure 5.5 we can see that a lower roughness value

corresponds to more rays reaching the camera and vice versa.

```
float GeometrySchlickGGX(float NoS, float K)
{
    return NoS / (NoS * (1.0 - K) + K));
}
float GeometrySmith(float NoV, float NoL, float Roughness)
{
    float K = Roughness * Roughness;
    float GGX1 = GeometrySchlickGGX(NoV, K);
    float GGX2 = GeometrySchlickGGX(NoL, K);
    return GGX1 * GGX2;
}
```



(a) Roughness 0.01 (b) Roughness 0.2 (c) Roughness 0.4

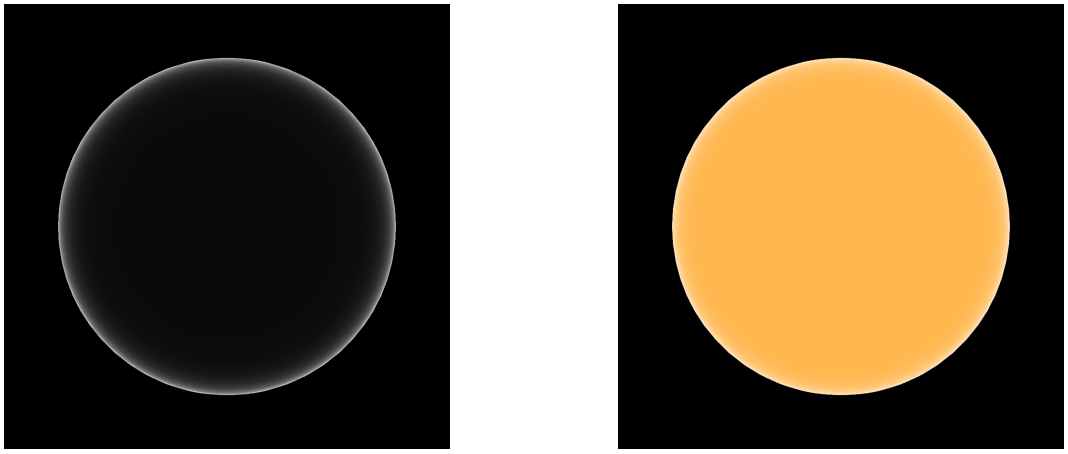
(d) Roughness 0.6 (e) Roughness 0.8 (f) Roughness 1.0

Figure 5.5: Geometry Smith visual results of a sphere

5.5.3 Fresnel Schlick

The Schlick approximation of the Fresnel equation is straightforward and give us the reflectivity of the surface. In Figure 5.6, we can notice that for fragments where $cos\theta$ is nearly equal to 1, the corresponding pixel colour is close to F_0, but when $cos\theta$ tends to 0, pixels start to be white, which means that those fragments are completely specular reflective.

```
vec3 FresnelSchlick(float CosTheta, vec3 F0)
{
    return F0 + (1.0 - F0) * pow(1.0 - CosTheta, 5.0);
}
```



(a) Fresnel of a dielectric surface. $F_0 = 0.04$

(b) Fresnel of a gold surface. $F_0 = (1.00, 0.71, 0.29)$

Figure 5.6: Schlick approximation with $cos\theta = n \cdot v$

5.5.4 BRDF

It is now simple to create the BRDF code assembling all the concepts discussed above.

```
vec3 BRDF(vec3 Albedo, float Metallic, float Roughness,
          float NoV, float NoL, float NoH, float VoH)
{
    float D = DistributionGGX(NoH, Roughness);
    float G = GeometrySmith(NoV, NoL, Roughness);
    vec3 F0 = mix(vec3(0.04), Albedo, Metallic);
    vec3 F = FresnelSchlick(VoH, F0);
    vec3 KD = (vec3(1.0) - F)*(1.0 - Metallic);
    vec3 Diffuse = KD * DiffuseLambert(Albedo);
    vec3 Specular = D * G * F / (4.0 * NoV * NoL);
    return Diffuse + Specular;
}
```

5.6 OpenCL/OpenGL Interoperability

OpenCL and OpenGL are two APIs that support interoperability. One of the main features given by interoperability is the capability of sharing data on the same device using the same buffers to store data, with the advantage of executing "zero-copy" operations. This advantage permits us to perform OpenCL computation while sharing result in real-time with OpenGL.

For the implementation discussed in this thesis, once the smoothing algorithm, running over a GPU with OpenCL, has completed its execution writing results in a specific buffer, we request OpenGL to execute the rendering using the same buffer without copying data in another memory area. The creation of a shared buffer requires a shared context between OpenCL and OpenGL which operate in the same device. This context is created by GLFW, an API for the management of windows, contexts and input, and required by OpenGL to perform rendering. A context can be created by calling `clCreateContext` of OpenCL API but due to the fact that we want the same OpenGL context, we

have to specify a parameter obtained from the function `wglGetCurrentContext` that contains the existing context created by GLFW.

```
#include <CL/cl_gl_ext.h>
cl_platform_id platform = select_platform();
cl_device_id deviceID;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &deviceID,
    NULL);
cl_context_properties props[] = {
        CL_GL_CONTEXT_KHR, (cl_context_properties)
            wglGetCurrentContext(),
        CL_WGL_HDC_KHR, (cl_context_properties)
            wglGetCurrentDC(),
        CL_CONTEXT_PLATFORM, (cl_context_properties)
            platform,
        0
};
cl_context context = clCreateContext(props, 1, &deviceID,
    ...);
```

Then, it is possible to create command queues, OpenCL programs and shared buffer within that context. It is worth noting that an OpenCL memory object is created from an OpenGL object and not vice versa. In our case, an OpenGL vertex buffer will be shared as an OpenCL buffer by calling `clCreateFromGLBuffer`.

Due to the fact that a lot of OpenGL and OpenCL API calls are not immediately executed but they are inserted in a command queues, concurrency management coordination is required to accesses shared resources. Therefore, before using them, shared buffers need to be acquired by an OpenCL command enqueued on a OpenCL command-queue. `clEnqueueAcquireGLObjects` call carries out this acquisition operation, returning a `cl_event` that will be

added to the computing kernel waiting-list.

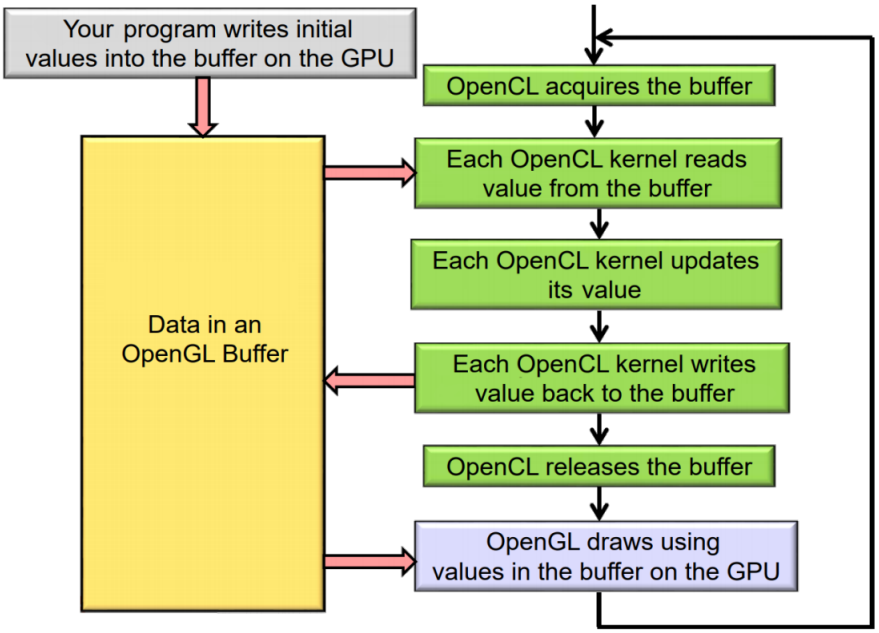


Figure 5.7: OpenCL and OpenGL interoperability work-flow

Once the computing kernel has finished its computation, it must release the used shared buffer calling `clEnqueueReleaseGLObjects`, in order to make it available again to OpenGL for the rendering process. This behaviour is encapsulated in a engine component called Mesh Smoothing that perform the process described every frame to accomplish the smoothing of a mesh.

```
cl_event lock, unlock;
cl_mem buffersToAquire[] = { cl_vertex_buffer,
    cl_normals_buffer };
clEnqueueAcquireGLObjects(queue, 2, &cl_vertex_buffer, 0,
    NULL, &lock);
cl_event smooth_evt = smooth(queue, smooth_kernel, &lock,
    cl_vertex_buffer, ...);
cl_event normals_per_face_evt = normals_per_face(queue,
    normals_per_face_k, &smooth_evt, cl_vertex_buffer, ...);
```

```
cl_event normals_per_vertex_evt = normals_per_vertex(queue,
    normals_per_vertex_k, &normals_per_face_evt,
    cl_normals_buffer, ...);
clEnqueueReleaseGLObjects(queue, 2, buffersToAquire, 1, &
    normals_per_vertex_evt, &unlock);
clWaitForEvents(1, &unlock);
```

Chapter 6

Results

6.1 Interoperability performance

The interoperability is a useful instrument, but executing GPGPU algorithms while rendering in the same device can also slow down performance. We will do some performance measurements to analyse this issue. The tests discussed below are performed using a Quad-Core i7-4790k 4.0 GHz and a Nvidia GeForce GTX 970 (specifications are summarized in Table 6.1) running Windows 10. The mesh test will be the Asian Dragon from XYZ RGB Inc. This mesh counts 3,609,455 vertices and 7,218,906 triangles and the mean vertex adjacents count is approximately 6.

The PBR engine frame-rate is stable, running at 60 frames per second while rendering the dragon in Full HD (1920x1080) resolution using PBR shaders and 4K textures. The frame-rate drops to 30 frames per second when enabling the Taubin smoothing that requires 2 smoothing iterations per frame due to the algorithm structure. The real-time smoothing also requires normals to be recalculated every frame because of the vertices changing their positions.

GPU	GM204 (Maxwell)
Multiprocessors	13
CUDA Cores	1,664
Base Clock	1,050 MHz
Boost Clock	1,178 MHz
Texture Units	104
L2 Cache	1.75 MB
Memory Type	GDDR5
Memory Clock	7.0 Gbps
Memory Bus	256-bit
Bandwidth	196 GB/s (3.5 GB), 28 GB/s (512 MB)
Memory Capacity	4 GB
Local Memory Banks	32

Table 6.1: Nvidia GeForce GTX 970 Specifications.

Fortunately, this is also performed on the GPU, exploiting the interoperability.

The smoothing kernel execution takes 2.15 milliseconds, 35% slower than the execution without rendering that takes 1.6 milliseconds. In addition, normals recalculation takes 5.5 milliseconds on the GPU. Moreover, as we said earlier, the smoothing kernel needs to be executed twice resulting in an overall time of 9.8 milliseconds of GPU computing every frame.

(a) Gaussian smoothing $\lambda = 0.5$

(b) Original mesh

(c) Taubin smoothing $\lambda = 0.5 \quad \mu = -0.51$

Figure 6.1: Asian Dragon after 200 smoothing iterations rendered by the PBR engine.

6.2 PBR Comparisons

We will present some comparisons against other major rendering engines to see what is the potentiality of real-time PBR.

Blender Cycles

Cycles is an unbiased render engine integrated within Blender. This engine is based on the path tracing algorithms thus implementing the global illumination. Cycles is also unbiased, which means that the rendering technique does not introduce any systematic error. This leads to a much more realistic rendering, but longer processing times due to sampling. Cycles has the ability to use the GPU through OpenCL or CUDA in order to speed up the calculation, however, it took 1 minute 37 seconds to render the image in Figure 6.2a with 512 samples and a Full HD resolution.

Unreal Engine 4

Unreal Engine 4 (UE4) is a cross-platform game engine developed by Epic Games. The software is continuously in development, adapting to the power of the available hardware and implementing the new features researched in the game industry. As well as our engine, UE4 is based on rasterization and it uses PBR shaders. As we can see in Figure 6.2c the rendering result is really close to Cycles and this is amazing since UE4 is real-time.

PBR Engine

The PBR Engine result (Figure 6.2b) is photorealistic, but, it does not look similar to the other two engines. This is due to the fact that this implementation is missing features such as exposure, colour balancing, and bloom effect.

(a) Blender Cycles

(b) PBR Engine

(c) Unreal Engine 4

Figure 6.2: Dwarven Drakefire Pistol from Warhammer Fantasy by Teliri

(a) Blender Cycles

(b) PBR Engine

(c) Unreal Engine 4

Figure 6.3: Asian Dragon by XYZ RGB Inc.

Chapter 7

Conclusions

"As technology advances, rendering time remains constant."

James F. Blinn

Blinn's Law is due to the fact that as computers become faster, graphic artists exploit the extra computational power to render more complex and sophisticated scenes adding more details and using expensive algorithms instead of keeping the same quality to lower rendering times. This means that rendering tends to consume all the computational power that modern hardware can offer.

Physically based rendering systems, started to be adopted by movie productions, are now widely used in video games, thanks to the computing capability of common GPUs, and the game development industry has been innovated. It is clear that in the near future, other innovations such as soft shadows, area lights, subsurface scattering, volumetric lights, and global illumination will come to life in real-time.

Together with a vast range of physics simulations taking place in GPUs, we are ready to see amazing graphics features on our screens.

Bibliography

[1] Riccardo Torrisi, *3D Mesh Smoothing: Parallel implementation and real-time visualization.*

[2] Gabriel Taubin, *Curve and surface smoothing without shrinkage*, https://graphics.stanford.edu/courses/cs468-01-fall/Papers/taubin-smoothing.pdf

[3] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, G. Ranzuglia, *MeshLab: an Open-Source Mesh Processing Tool*, http://vcg.isti.cnr.it/Publications/2008/CCCDGR08/MeshLabEGIT.final.pdf

[4] Mark D. Hill and Michael R. Marty, *Amdahl's Law in the Multicore Era*, http://research.cs.wisc.edu/multifacet/papers/tr1593_amdahl_multicore.pdf

[5] Ogier Maitre, *Understanding NVIDIA GPGPU Hardware*, https://www.springer.com/cda/content/document/cda_downloaddocument/9783642379581-c1.pdf?SGWID=0-0-45-1430823-p175116730

[6] Christos Kyrkou, *Stream Processors and GPUs: Architectures for High Performance Computing*, http://sokryk.tripod.com/Stream_Processors_and_GPUs_-_Architectures_for_High_Performance_Computing.pdf

[7] Whitepaper, NVIDIA GF100, *World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism*, http://www.hardwarebg.com/b4k/files/nvidia_gf100_whitepaper.pdf

[8] Whitepaper, *NVIDIA GeForce GTX 1080*, https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

[9] James T. Kajiya, California Institute of Technology, Pasadena, Ca. 91125, *THE RENDERING EQUATION*, http://www.cse.chalmers.se/edu/year/2011/course/TDA361/2007/rend_eq.pdf

[10] Robert L. Cook and Kenneth E. Torrance, *A Reflectance Model For Computer Graphics*, http://inst.cs.berkeley.edu/~cs294-13/fa09/lectures/cookpaper.pdf

[11] Sébastien Lagarde and Charles de Rousiers, *Moving Frostbite to Physically Based Rendering 3.0*, file:///C:/Users/Daniele/Documents/Thesis/External/course_notes_moving_frostbite_to_pbr_v32.pdf

[12] Brian Karis, Epic Games, *Real Shading in Unreal Engine 4*, http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf

[13] Brent Burley, Walt Disney Animation Studios, *Physically-Based Shading at Disney*, https://disney-animation.s3.amazonaws.com/library/s2012_pbs_disney_brdf_notes_v2.pdf

[14] Naty Hoffman, *Background: Physics and Math of Shading*, http://blog.selfshadow.com/publications/s2013-shading-course/hoffman/s2013_pbs_physics_math_notes.pdf

[15] Bruce Walter, Stephen R. Marschner, Hongsong Li, Kenneth E. Torrance, *Microfacet models for refraction through rough surfaces* https://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.pdf

[16] Christophe Schlick, *An Inexpensive BRDF Model for Physically-based Rendering*, http://igorsklyar.com/system/documents/papers/28/Schlick94.pdf

[17] Eric Heitz, *Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs*, http://jcgt.org/published/0003/02/03/paper.pdf

[18] Bruce G. Smith, *Geometrical shadowing of a random rough surface.*