SUNG-EUI YOON, KAIST

RENDERING

FREELY AVAILABLE ON THE INTERNET

http://sglab.kaist.ac.kr/~sungeui/render

First printing, July 2018

Contents

Dedicated to TaeYoung and JaeHa.

Preface

Rendering is a way of visualizing various 3D models in 2D images or videos. It is one of fundamental tools in the field of computer graphics. Thanks to its ubiquitous demand, it is not only used for applications in computer graphics, but also widely used for many other fields.

There have been tremendous progress on rendering techniques. One of epitomes for rendering techniques is games, where we can see real-time, yet high-quality rendering images. These real-time techniques are commonly based on the concept of rasterization, which is the main theme of Part I of this book. Another successful application of computer graphics is movie. Unlike games, the movie production accommodates much longer computational time for higher rendering quality. Ray tracing based rendering techniques, therefore, are utilized more frequently for movies. These ray tracing techniques are mainly discussed in Part II.

These techniques have been developed for many decades. For example, the concept of ray tracing was introduced to the field of computer graphics at 1980. Since rendering techniques have been studied for a long period of time, it is very hard to catch up all the major concepts, unless properly guided. Also, new concepts and techniques have been constantly proposed.

Many graphics books are available, but only a few rendering books are available. Furthermore, most of them focuses either one of two main rendering techniques, rasterization and ray tracing, in an advanced manner. Given this situation, I decided to treat both of them, while covering most fundamental concepts of those techniques. I will also cover advanced topics as I have more time, built on top of those basic concepts.

In order to save time of writing this book and better explain concepts, I re-used many existing materials (e.g., images) of lecture slides and papers. For each of them, I mentioned its source, but here I'd like to point out that I borrowed many images from lecture slides used in a Computer Graphics course given at University of North Carolina at Chapel Hill for Part I and lecture slides of Prof. Kavita

Bala for Part II. Also, the latex template of this book is based on the Tufte's design style and is based on the Apache license.

Finally, many students of CS380, CS480, CS580 offered at KAIST posed many interesting questions that are the basis of many Q&A parts of this book. Also, many of them gave useful comments on different parts of this book.

Sung-eui, KAIST

July, 2018

1
Introduction

Rendering is one of the fundamental techniques in computer graphics, visualization, and many other related fields. Since the rendering technique has been widely used in many different applications, its perceived meaning can vary a lot depending on people using it.

For users and developers for games, rendering techniques should be interactive and can support many interesting visual effects (e.g., magic fire). In terms of the performance, the rendering part used in games should take less than 10 ms, since the whole frame takes 33 ms assuming 30 frames per second, and other parts (e.g., game logics and network) can take 10 ms to 20 ms. As a result, rendering methods adopted in such games should be extremely fast[1].

For viewers, artist, and developers for movies, rendering techniques should be photo-realistic and provide even artistic controls on effects that they want to express. In many movies (e.g., Jurassic Park), we see scenes captured from real cameras and mixed together with computer generated effects and virtual objects. Rendering methods for these movies should be indistinguishable between real and virtual scenes. As a result, these techniques are usually based on physics and simulations of light and material interactions. Furthermore, artists and directors making such movies are not satisfied with such realistic looking results[2]. They want to convey particular emotion and mood on computer generated effects. We thus need techniques accommodating such user inputs.

As you can see, there are such a wide variety of rendering applications with different characteristics. Therefore, a single rendering method satisfying all those characteristics and requirements is hard to be developed. As a result, many different rendering and visualization methods have been developed. Instead of covering them in detail in this book, we would like to cover main techniques and their variations.

[1] Games requires real-time rendering techniques spending only 10 ms for each frame

[2] Rendering used for movies needs to provide realistic results, while supporting various artistic directions

1.1 Rendering Techniques

At a high level, there are two main, but different rendering techniques: rasterization and ray tracing.

There are two main rendering techniques: rasterization and ray tracing

Rasterization is to traverse triangles of a model and project triangles to the frame buffer. Rasterzation is classified as an object driven rendering method and has been widely accelerated by various hardwares (e.g., GPUs) because of its simplicity. Thanks to the simplicity and the hardware acceleration, rasterization based rendering methods can show an interactive rendering performance even for massive models consisting of more than hundreds of millions of triangles [3]. Thanks to these features, rasterization techniques are available in OpenGL and DirectX, graphics APIs, and adopted in many games through game engines (e.g., Unity).

[3] Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha. Quick-VDR: Interactive View-dependent Rendering of Massive Models. In *IEEE Visualization*, pages 131–138, 2004

Ray tracing, however, generates rays per each pixel and finds triangles that intersect with these rays by traversing an acceleration hierarchy. Ray tracing is classified as a view-driven rendering method and requires random access on meshes and hierarchies. Therefore, it requires much complex control logics and caches and in turn has been showing much (e.g., two orders of magnitude) slower performance than that of rasterization based methods.

Although ray tracing shows much slower performance than rasterization, it can naturally support physically-correct rendering because its algorithm follows the physical intersections between lights and materials. Therefore, it has been widely used in offline applications (e.g., movies) that require high-quality rendering results. On the other hand, rasterization has been widely used for interactive applications such as games.

While there are such stereotypical usages of rasterization and ray tracing, these techniques are still under active, yet steady development, and are thus improved in many different directions. For example, many games want to interactively support realistic rendering effects that ray tracing has been able to support in the domain of rasterization. Furthermore, some of recent applications such as Poketmon Go, an AR (augmented reality) application, needs to seamlessly integrate camera-captured scenes and computer generated effects. To realize this, ray tracing and rasterization techniques are used together to achieve both the performance and quality[4].

[4] Ray tracing and rasterization are used together for achieving the fast performance and high quality.

Relationship with other fields. Computer graphics commonly assumes that virtual scenes are represented by various types of models such as triangles for the scene geometry and BRDF for material appearance (Fig. 1.1). The main output of various computer graphics methods is an image or a series of images known as video. Com-

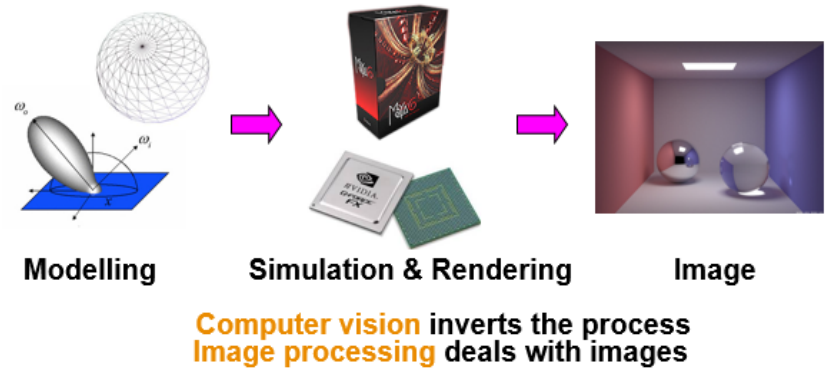


Figure 1.1: An overall structure of computer graphics. Images are adopted from Google image search.

mon methods for computer graphics include rendering, a type of simulation for light and material interactions, and many other types of simulations such as cloth, fire, character simulations. Computer vision commonly starts with images and attempts to extract models (e.g., geometry and BRDF), and image processing deals with images for denoising or many other image improvement. One of well known image processing tools is Photoshop from Adobe.

These different approaches have been developed and matured in its own fields (e.g., computer graphics and vision). Recently, these techniques developed from different fields are mixed together to create novel applications and approaches. As a result, their boundaries become rather blurred in these days.

Applications of computer graphics. Numerous applications of computer graphics exist. A lot of them are in the entertainment business for making games and movies (Fig. 1.2). Some movies are generated totally based on computer graphics, or some scenes of movies have various special effects. They also get renewed attentions with other related technology advances such as introduction of 3D TV to consumer markets and head mounted display (HMD) for virtual reality (VR) and augmented reality (AR).

In addition to various entertainment applications, various product designs and analysis such as computer-aided design (CAD) uses computer graphics. Also, medical and scientific visualization is a big part of computer graphics. Finally, information visualization that associates various geometric meaning to complex data (e.g., graphs) are getting bigger and bigger.

Organization of the book. Rendering has been studied in various aspects covering optics and novel applications. As a result, we focus on the following two parts in this book.

Figure 1.2: Applications of computer graphics. From the top left, image cuts of startcraft, toy story, a CT image of mouse skull, a weather visualization from LLNL, and a double eagle oil tanker for CAD.

1. **Rasterization.** Rasterization is an efficient rendering technique that mainly works in an image space that can be easily accelerated in GPUs. This approach is discussed in Part I.

2. **Ray tracing.** Ray tracing is a common approach of simulating the physical interaction between the light and materials. It is therefore widely used for providing physically-based rendering. This is discussed in Part II.

1.2 *Related Materials*

Rendering techniques have been studied for several decades, and excellent books are available. We list some of them here:

- Fundamentals of Computer Graphics, by Peter Shirley et al. [5]. This book covers various fundamental topics of computer graphics.

- Physically based rendering by Pharr et al. [6].. This book also covers a wide variety of topics of physical-based rendering. It also provides source codes for all the concepts discussed in the book. If you want to have hands-on experience on physics-based rendering, this book provides both theoretical concepts and practical programming tools.

- Advanced Global illumination, by Dutre et al. [7]. This book covers physics-based rendering techniques.

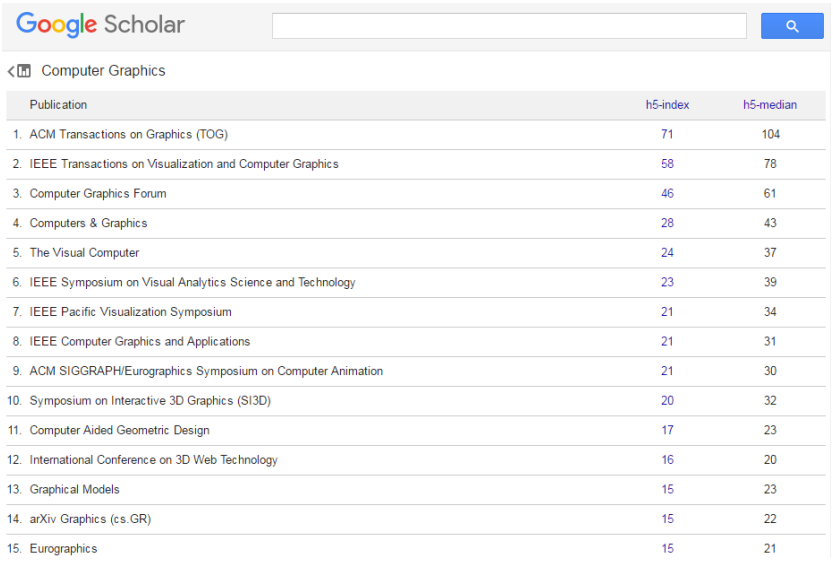
- Realistic ray tracing, by Shirley et al. [8]. While this book is rather old, it covers various concepts and detailed information of ray tracing, which is one of main ingredients of building physics-based rendering.

[5] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., 3rd edition, 2009

[6] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation 2nd*. Morgan Kaufmann Publishers Inc., 2010a

[7] Philip Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. AK Peters, 2006

[8] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. AK Peters, second edition, 2003



Google Scholar

Computer Graphics

	Publication	h5-index	h5-median
1.	ACM Transactions on Graphics (TOG)	71	104
2.	IEEE Transactions on Visualization and Computer Graphics	58	78
3.	Computer Graphics Forum	46	61
4.	Computers & Graphics	28	43
5.	The Visual Computer	24	37
6.	IEEE Symposium on Visual Analytics Science and Technology	23	39
7.	IEEE Pacific Visualization Symposium	21	34
8.	IEEE Computer Graphics and Applications	21	31
9.	ACM SIGGRAPH/Eurographics Symposium on Computer Animation	21	30
10.	Symposium on Interactive 3D Graphics (SI3D)	20	32
11.	Computer Aided Geometric Design	17	23
12.	International Conference on 3D Web Technology	16	20
13.	Graphical Models	15	23
14.	arXiv Graphics (cs.GR)	15	22
15.	Eurographics	15	21

Figure 1.3: This shows a list of graphics related conferences and journals according to Google Scholar at 2016. Note that many conferences papers in computer graphics are published at journals, and thus journals (e.g., ACM Trans. on Graphics) are ranked higher than well-known conferences (e.g., SIGGRAPH).

These books cover fundamental concepts of rendering, but lacks recent developments. If you want to follow those recent techniques, you can find recent papers through the following:

- Google scholar. You can find recent technical papers from various search engines. Especially, Google scholar is useful, since it also identifies papers that refer to a particular paper. By looking this information, you can find prior and future works given a particular paper.

- Graphics conferences and journals. Novel ideas are generated in every where. One can easily learn those novel ideas by looking at recent papers published at graphics conferences and journals. One of well-known of them is ACM SIGGRAPH, whose papers are published at a journal called ACM Trans. on Graphics (ToG). Google Scholar also provides a list of influential conferences and journals with their ranking (Fig. 1.3).

1.3 *Common Q & A*

Do we need an excellent artistic sense to study computer graphics or to become an technical expert in this field? Not really. Of course, it is always better to have a good artistic sense to work on visual data processing. However, if some jobs require such a high standard of artistic senses, those jobs may be for artistic designers, not for engineers. In my opinion, it is more important to have better

engineering backgrounds (e.g., mathematical backgrounds and algorithm developments) and problem-solving skills. For example, I don't have any sense of art, but I work on computer graphics!

I have found that something like tea pot and bunny models are widely used in many papers and technical videos. Why? You made a good observation. Some of models including the Utah teapot and Stanford bunny have been created earlier as research results or research benchmarks. Then, these models are distributed to other researchers for their follow-on research. That's why these models are widely used in many papers.

Part I

Rasterization

Rasterization is one of most popular rendering techniques developed for computer graphics. It simply projects triangles in a scene into a viewing space and color pixels overlapped with those triangles. This approach is very simple and thus can be implemented efficiently in specialized hardwares. Especially, many graphics hardware and GPUs support this rasterization scheme.

It, however, does not simulate the natural interaction between light and materials. Simply speaking, in reality, objects are not projected into our eyes! Due to this issue, rasterization schemes have fundamental drawbacks of simulating various rendering effects such as shadows, transparency, and so on. Nonetheless, thanks to its fast performance, many techniques and fixes have been proposed to improve its rendering quality.

In this part, we discuss the fundamental engine of rasterization, which is developed in many graphics library such as OpenGL and DiretX accelerated by GPUs. In other parts , we study global illumination that physically simulates interactions between lights and materials.

1.4 Related Materials

Many useful resources for rasterization techniques are available. Some of them are listed here:

- OpenGL Programming Guide. OpenGL is one of very popular computer graphics library that can be used in a wide variety of computing platform including Windows, Linux, and mobile OS. OpenGL provides various useful low-level graphics APIs, and they are well explained in this book and in its reference book. Early version of these books are available on free at internet. We also explain some of OpenGL APIs and their concepts, when we explain concepts of rasterization for delivering concrete examples.
- Real-time rendering [9] and its resource. This book covers a vast amount of topics that are related to rasterization and real-time rendering techniques. Its resource cite [10] has many useful web pages and links.
- OpenGL tutorials. Many OpenGL tutorials exist at Web. Some of them are based on the legacy OpenGL, but `http://www.opengl-tutorial.org/` discusses useful tutorials based on a recent OpenGL (ver. 3.3 and later).

[9] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., 2008

[10] `http://www.realtimerendering.com/`

2
Rendering Pipeline

Rendering triangles for scenes requires an excessive amount of computation time, since there could be many triangles representing scenes, and each triangle can map to hundreds of pixels in the screen space. As a result, carefully designed steps, known as rendering pipeline, has been proposed.

2.1 Classic Rendering Pipeline

Let us first discuss the classic rendering pipeline, before studying a modern, but complex one.

Fig. 7.1 shows an example of a classic rendering pipeline running on a GPU. An graphics application runs on a CPU in general and sends geometry of the scene and a camera setting that its user wants to see to a GPU by using a graphics library such as OpenGL. The rendering pipeline implemented in a GPU processes such requests and computes an output image displayed in a screen.

In general, the rendering pipeline consists of many steps for drawing an image from the user's camera position and orientation in an efficient manner. At a high level, they usually breaks into vertex processing and pixel processing units. The vertex processing step transforms input geometry into ones mapped in the screen space. Those ones are converted into pixels with appropriate colors by the pixel processing step, and this step is commonly known as the rasterization step.

Historically, these steps take a high computation time and thus are implemented in a chip in a hard-wired manner. These steps, therefore, are rather fixed functions and invoked through graphics APIs. As we have more processing power and developers request more flexibility on programming, the GPU implementing these steps become more general like CPU and can run various graphics programs such as OpenGL shaders.

While more accurate rendering techniques (e.g., global illumi-

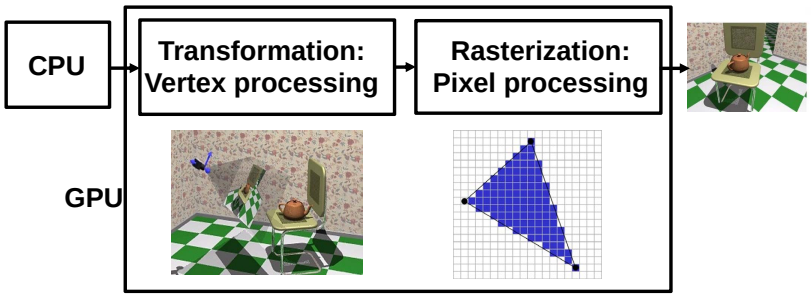


Figure 2.1: This shows a schematic diagram of classic rendering pipeline consisting only two steps: vertex and pixel processing steps.

nation) have been proposed with high performance, rasterization scheme is one of the most efficient rendering algorithms specializing on local illumination, which considers the light energy transfer between a surface and a light source. We therefore study this scheme in a detailed manner in Chapter 3 and 7.

2.2 *Modern Rendering Pipeline*

Fig. 7.2 shows a schematic view on a modern rendering pipeline adopted in OpenGL 3.0. While this differs a lot from the classical one, it shares both vertex and pixel (e.g., fragment) processing steps.

- **Vertex specification.** Vertices and triangles are defined and passed to the following step.
- **Vertex processing.** Each vertex is processed by a vertex shader, a program working on each vertex. It performs various modeling transformation, viewing, and projection transformations.
- **Vertex post-processing.** It performs various basic operations after the vertex processing step and serves as a setup stage for the following steps such as rasterization. It includes clipping (Sec. 6.4), homogeneous divide (Sec. 4.2.1), and viewport transformation (Sec. 3.1).
- **Primitive assembly.** Face culling is performed in this step.
- **Rasterization.** This step converts a triangle represented by vertices into a number of fragments.
- **Fragment shader.** It also processes each fragment generated by the prior rasterization step.

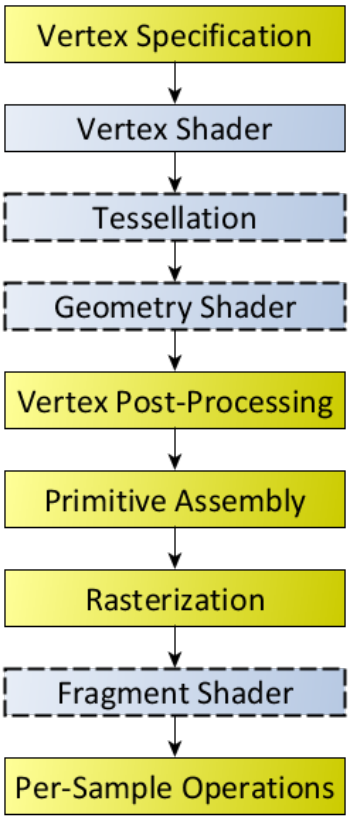


Figure 2.2: This shows a rendering pipeline adopted in OpenGL 3.0. This image is excerpted from the OpenGL homepage.

2.3 OpenGL and Other Tools

The rendering pipeline has been implemented and accelerated in GPUs. To enjoy such hardware acceleration, we use OpenGL and DirectX. OpenGL is more widely available in different operation systems and devices, since DirectX depends on Windows OS. Most concepts and techniques that are covered in this part are available at such APIs. Nonetheless, it is useful to know what other tools related to graphics are available and their goals. Fig. 2.3 shows other tools and languages that can utilize various features of GPU other than the rasterization.

Recently, Vulkan was introduced for achieving even higher performance on mobile phones [1] that have lower performance than PCs. For achieving its goal, Vulkan allows users to various low-level APIs with low overheads and multi-tasking. Nonetheless, it comes with certain costs such as higher programming burdens to users.

[1] G. Sellers and J.M. Kessenich. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison Wesley, 2016

While these APIs provide the full features of the rendering pipeline, they are rather low-level APIs. When we want to develop high-level applications such as a game, we need to utilize a more powerful set of tools and SWs. This is a gap that modern game and rendering engines such as Unity try to fill in. Additionally, in graphics applications (e.g., games and movies), content creation is one of main tasks, and many modeling and animation tools are available.

Initially, GPU is designed as a specialized hardware to accelerate the rendering process, which is captured in the rendering pipeline. However, as the performance of GPU is getting higher and various demands on programmability on the rendering pipeline arise. As a result, parts of vertex and fragment stages can be programmable through a dedicated language, i.e., GLSL and HLSL.

While these shading languages are designed to effectively utilize functions of GPUs for graphics applications, non-traditional needs on using GPUs for non-graphics applications keep increasing, thanks to its higher performance on streaming tasks than CPUs. To accommodate such demands, a general purpose language for utilizing GPUs has been proposed, and CUDA and OpenCL are two examples.

2.3.1 Common Questions

What if we have new input devices (e.g., joystick, or multiple input devices used in PlayStation or XBox)? How can we handle those devices in OpenGL programs? OpenGL does not have any functionality to support those various input devices. GLUT library supports some of basic input devices such as keyboard and mouses. For other devices, you need to use other external libraries that support those

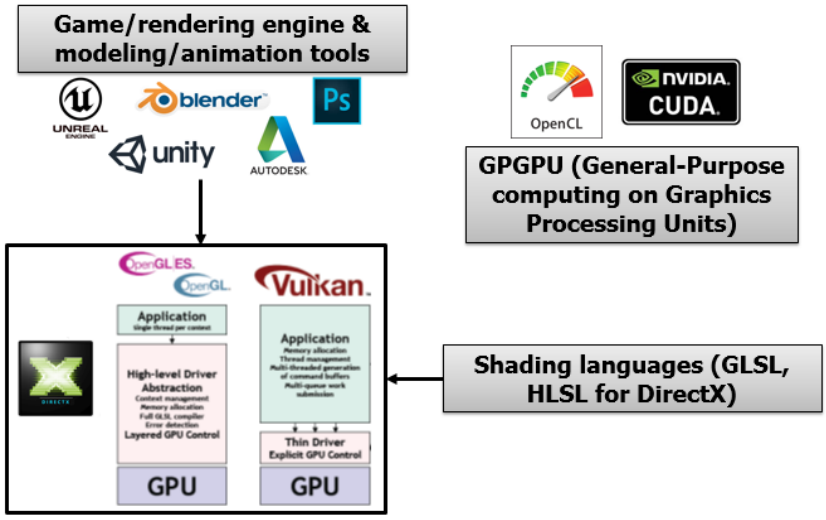


Figure 2.3: This figure shows other APIs, SWs, and languages that are related to OpenGL and computer graphics. In this book, we mainly discuss the core rendering pipeline that rasterizes input models. Nonetheless, many game and rendering engines (e.g., Unity) are commonly used as convenient, high-level tools. Also, shading languages are used in recent OpenGL verions, to add various details on rendering results. Additionally, general purpose computing languages for GPU (e.g., CUDA) are also used for implementing arbitrary programs on GPUs. Images are excerpted from the Vulkan overview and Google images.

devices.

In what cases, is OpenGL used rather than DirectX? OpenGL is cross-platform graphics API, while DirectX is proprietary library for Windows. Because of the openness of OpenGL, it, more specifically, OpenGL ES, is widely used for many embedded systems including mobile phones.

In what portions of my OpenGL program are executed in CPU and GPU? In a typical OpenGL program, rendering parts (e.g., portions started with glBegin and ended with glEnd) are performed in GPU, graphics hardware, if your computer is equipped with such GPU. All the control parts, e.g., calling OpenGL functions and handling events, are performed in CPU. In other words, various functionality inside OpenGL APIs are commonly performed in GPU, while all the other parts are performed in CPU.

3
Transformation

Many components of rasterization techniques rely upon different types of transformation. In this chapter, we discuss those transformation techniques.

3.1 Viewport Transformation

In this section, we explain the viewport transformation based on an example. Fig. 3.1 show different spaces that we are going to explain.

Suppose that you have an arbitrary function, $f(x, y)$, as a function of 2 D point (x, y); e.g., $f((x, y)) = x^2 + y^2$. Now suppose that you want to visualize the function in your computer screen with a particular color encoding method, e.g., heat map that assigns hot and cold colors depending on values of $f(x, y)$.

This function is defined in a continuous space, say x and y can be any real values. In computer graphics, we use a term of world to denote a model or scene that we would like to visualize or render. In this case, the function $f(x, y)$ is our world. Our goal is to visualize this function so that we can understand this function better. In many cases, the world is too large and thus we cannot visualize the whole world in a single image. As a result, we commonly introduce a camera to see a particular region of the world.

Unfortunately, our screen is not in the continuous space and has only a limited number of pixels, which is represented by a screen resolution. Our graphics application can use the whole screen space or some part of it. Let us call that area as a screen space. Fig. 3.2 show common conventions of the screen space. Finally, we visualize a part of the world seen through the camera into a part of our screen space, which is commonly known as a viewport; note that we can have multiple viewports in our screen.

Suppose a position, x_w, in the world that we are now seeing in the camera. In the end, we need to compute its corresponding position, x_s, in our screen space of the viewport. If we know x_s, we can draw

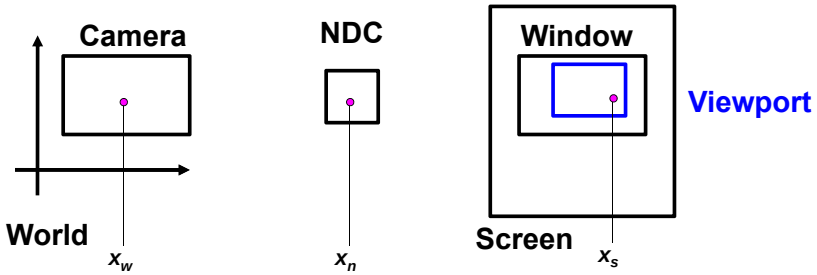


Figure 3.1: This shows a mapping from a viewable region in the world through a camera to the viewport in our screen space pass through the intermediate space, normalized device coordinate (NDC).

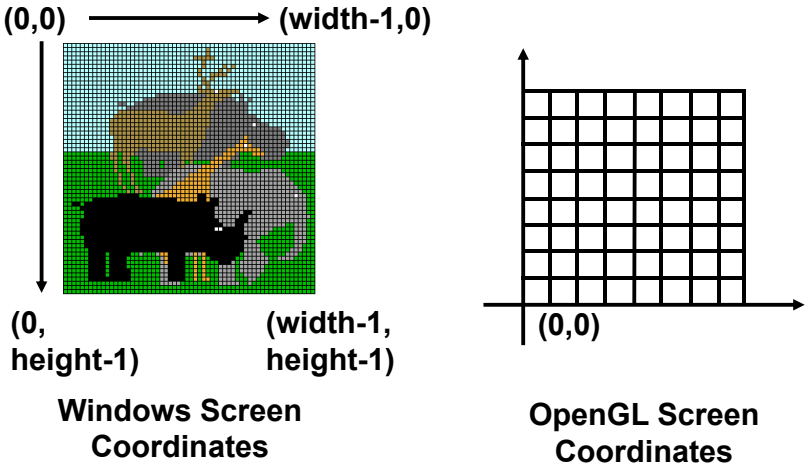


Figure 3.2: This shows two different conventions of screen coordinate spaces.

the color of the world position x_w at x_s. The question is how to compute x_s from x_w, i.e., the mapping from the world space to the viewport or screen space.

Normalized device coordinate (NDC). While world and screen spaces are two fundamental spaces, we also utilize NDC. NDC is a canonical space, whose both X and Y values are in a range of $[-1, 1]$. NDC serves as an intermediate space that is transformed to the screen space, which is hardware-dependent space. As a result, given the world potion x_w, we first need to compute a position in the NDC space, x_n, followed by mapping to x_s. We will also see various benefits of using NDC later, which include simplicity and thus efficiency of various rasterization operations.

Mapping from the world space to NDC. Suppose that the part of the world that we can see through a camera is represented by $[w.l, w.r] \times [w.b, w.t]$, where $w.l$ and $w.r$ are the visible range along X-axis and $w.b$ and $w.t$ define the visible range in Y-axis, while the

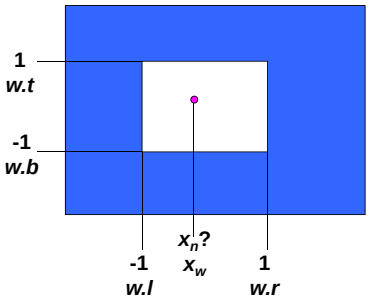


Figure 3.3: Mapping between the world space and NDC.

NDC space is represented by $[-1,1] \times [1,1]$.

Since the relative ratio of x_w and x_n is same in each space, we have the following relationship:

$$\frac{x_n - (-1)}{1 - (-1)} = \frac{x_w - (w.l)}{w.r - w.l}.$$
$$x_n = 2\frac{x_w - w.l}{w.r - w.l} - 1.$$
$$x_n = Ax_w + B,$$

where $A = \frac{2}{w.r - w.l}, B = -\frac{w.r + w.l}{w.r - w.l}$. This equation indicates that given the information, we can compute the NDC coordinate with one multiplication and one summation. Similarly, we can derive the mapping equation from x_n to x_s.

An issue of this approach is that there are too many pixels and thus evaluating such simple equations requires computational time. Since most graphics applications require interactive or real-time performance, we need to think about efficient way of handling these operations early in the history of computer graphics. Furthermore, it turns out that such mapping and similar transformations are very common operations in many graphics applications. The most common way of handling them in an efficient and elegant way is to adopt linear algebra and use matrix operations.

3.1.1 Common Questions

Can glBegin () with GL_POLYGON support concave polygons? According to its API description, GL_POLYGON works only with convex polygons. But, what may happen with concave polygons? Since it is not part of the specification of OpenGL, each vendor can have their own handling method for that kind of unspecified cases. If you are interested, you can try it out and let us know.

In the case of rendering circles, shown as an example in the lecture note, we render them by using lines. Is there a direct primitive that supports the circle? OpenGL has a limited functionality that supports continuous mathematical representations including circles, since a few model representations (e.g., triangles) have been widely used and it is hard to support all the possible representations. However, OpenGL keeps changing and it may support many continuous functions in a near future. At this point of time, we need to discretize continuous functions with triangles or other simple primitives and render them.

We use the NDC between the world space and the screen space. Isn't it inefficient? Also, don't we lose some precision during this

process? There is certainly some overhead by introducing the NDC. However, it is very minor compared to its benefits in terms of simplifying various algorithms employed throughout the rendering process. Yes. We can lose more precision during the conversion process due to float operations. However, it may be very small and may not cause significant problems for rendering purposes. Nonetheless, the transformation is based on analytic equations, not pixels, and thus can be easily recovered to the original information.

OpenGL is designed for cross-platform. But, I think that it means that we cannot use assembly programming for higher optimizations. Yes. You're right. We cannot use assembly languages for such optimizations. However, programmers for graphics drivers for each graphics vendor definitely use an assembly language and attempt to achieve the best performance. High-level programmers like us rely on such drivers and optimize programs with OpenGL API available to us.

Multi-threading with OpenGL: Since OpenGL has been designed very long time ago and has many different threads, it requires some cares to use multiple threads for OpenGL. There are many articles in internet about how to use multiple threads with OpenGL. I recommend you to go over them, if you are interested in this topic.

Why do we use a viewport? The viewport space doesn't need to be the whole window space. Given a window space, we can decompose it into multiple sub-spaces and use sub-spaces for different purposes. An example of using multiple viewports is shown in Fig. 3.4.

3.2 *2D Transformation*

In this section, we discuss how to represent various two dimensional transformation in the matrix form. We first discuss translation and rotation.

2D translation has the following forms:

$$x' = x + t_x, \tag{3.1}$$

$$y' = y + t_y, \tag{3.2}$$

where (x, y) is translated as an amount of (t_x, t_y) into (x', y'). They are also represented by a matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}. \tag{3.3}$$

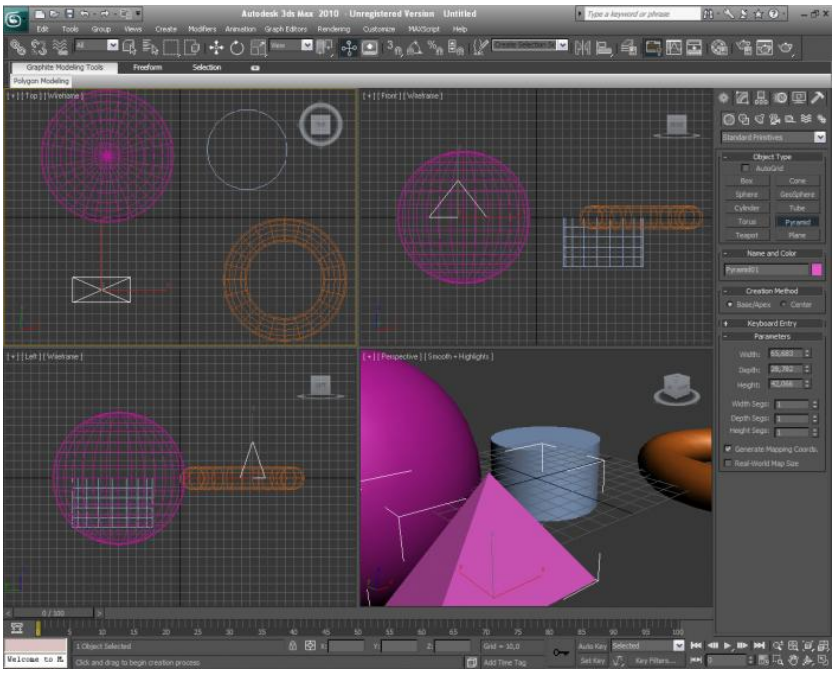


Figure 3.4: This figure shows multiple viewports, each of which shows an arbitrary 3D view in addition to top, front, and side views. The image is excerpt from screenshots.en.sftcdn.ne.

Given the 2D translation, its inverse function that undoes the translation is:

$$x = x' - t_x, \tag{3.4}$$
$$y = y' - t_y. \tag{3.5}$$

Also, its identity that does not change anything is:

$$x' = x + 0, \tag{3.6}$$
$$y' = y + 0. \tag{3.7}$$

Let us now consider 2D rotations. Rotating a point (x, y) as an amount of θ in the counter-clock wise is:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = R_\theta \begin{bmatrix} x \\ y \end{bmatrix}, \tag{3.8}$$

where R_θ is the rotation matrix. Its inverse and identity are defined as the following:

$$R^{-1} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}, \tag{3.9}$$
$$R_{\theta=0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{3.10}$$

Suppose that you want to rotate an object by 30 degrees, followed by rotating it again with 60 degrees. We intuitively know that rotating 90 degrees in a single time gives the same effect of rotating 30 degrees and 60 degrees again.V Formally, one can prove the following equation:

$$R_{\theta_2} R_{\theta_1} = R_{\theta_1 + \theta_2}. \tag{3.11}$$

3.2.1 *Euclidean Transformation*

In this subsection, we would like to discuss a particular class of transformation, Euclidean transformation. The Euclidean transformation preserves all the distances between any pairs of points. Its example includes translation, rotation, and reflection. Since the shape of objects under this transformation is preserved, the Euclidean transformation is also known as rigid transformation.

This rigid transformation is one of most common transformation that we use for various game and movie applications. For example, camera rotation and panning are implemented by the rigid transformation.

Mathematically, the Euclidean transformation is represented by:

$$T(x) = Rx + t, \tag{3.12}$$

where R and t are rotation matrix and 2D translation vector.

While this is a commonly used mathematical representation, this representation has a few drawback for graphics applications. Typically, we have to perform a series of rotation and translation transformation for performing the viewport transformation, camera operations, and other transformation applied to objects. As a result, it can take high memory and time overheads to apply them at runtime. Furthermore, there is cases that we need to compute a invert operation from a coordinate from the screen space to the corresponding one in the world space. Given the series of rotation and translation operations, the inverting operation can require multiple steps.

As an elegant and efficient approach to these issues, the homogeneous coordinate has been introduced and explained in the next section.

3.2.2 *Homogeneous Coordinate*

Homogeneous coordinates are originally introduced for projective geometry, but are widely adopted for computer graphics, to represent the Euclidean transformation in a single matrix.

Homogeneous coordinates provides various benefits for transformation and are thus commonly used in graphics.

Suppose a 2D point, (x, y) in the 2D Euclidean space. For the homogeneous coordinates, we introduce an additional coordinate,

and (x, y) in the 2D Euclidean space corresponds to $(x, y, 1)$ in the 3D homogeneous coordinates. In fact, $(zx, zy, z), z \neq 0$ also corresponds to (x, y) by dividing the third coordinate z to the first and second coordinates, to compute the corresponding 2D Euclidean coordinate.

Intuitively speaking, (zx, zy, z) represents a line in the 3D homogeneous coordinate space. Nonetheless, any points in the line maps to a single point (x, y) in the 2D Euclidean space. As a result, it can describe a projection of a ray passing through a pin hole to a point.

Let us now describe its practical benefits for our problem. Before we describe the Euclidean transformation (Eq. 3.12) and its problems. By using the 3D homogeneous coordinate, the Euclidean transformation is represented by:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{3.13}$$

Note that the translation amount t_x and t_y are multiplied with the homogeneous coordinate, which is one. As a result, the translation is incorporated within the transformation matrix that also encodes the rotation part simultaneously.

One of benefits of using the homogeneous coordinates is to support the translation and rotation in a single matrix. This property addresses problems of the Euclidean transformation (Sec. 3.2.1). Specifically, even though there are many transformations, we can represent each transformation in a single matrix and thus their multiplication is also represented in a single matrix. Furthermore, its inversion can be efficiently performed. Thanks to these properties resulting in a higher performance, the homogeneous coordinates have been widely adopted.

Revisit to mapping from the world space to NDC. We discussed viewport mapping, one of which operation transforms world space coordinates to those in NDC space (Sec. 3.1). Since this transformation uses multiplication, followed by the additions, it can be represented by homogeneous coordinates and thus in a single matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{w.r-w.l} & 0 & -\frac{w.r+w.l}{w.r-w.l} \\ 0 & \frac{2}{w.t-w.b} & -\frac{w.t+w.b}{w.t-w.b} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{3.14}$$

Nonetheless, the matrix is not exactly in the Euclidean transformation, since it involves scaling. This is covered in the affine transformation in the next section.

3.2.3 *Affine Transformation*

We discussed the Euclidean transformation that is a combination of rotation and translation in Sec. 3.2.1. We now study on an affine transformation, which covers wider transformation than the Euclidean transformation.

In the 2D case, the affine transformation has the following matrix representation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{3.15}$$

The affine transformation preserves parallel lines under the transformation, but does not necessarily preserve angles of lines. The affine transformation covers a combination of rotation, translation, shearing, reflection, scaling, etc. The transformation is also called projective transformation, since it also supports projection, which is discussed in Sec. 4.2.

OpenGL functions. Various transformation functions (e.g., $glTranslate(\cdot)$) available at early versions of OpenGL (e.g., version 2) are deprecated in recent versions. Nonetheless, it is informative to see its usage with corresponding matrix transformations, which are adopted in the recent OpenGL.

The following code snippet shows a display function of rendering a rectangle with a rotation matrix.

```
void display(void)
{
  // we assume the current transformation matrix to be the identify matrix.
   glClear(GL_COLOR_BUFFER_BIT); // initialize the color buffer.

   glPushMatrix(); // store the current matrix, the identify matrix, in the matrix stack
   glRotatef(spin, 0.0, 0.0, 1.0); // create a rotation matrix, M_r.
   glColor3f(1.0, 1.0, 1.0);
   glRectf(-25.0, -25.0, 25.0, 25.0); // create geometry, say, v.
   glPopMatrix(); // go back to the initial identify matrix.

   glFinish (); // send all OpenGL commands to GPU and finish once they are done.

   glutSwapBuffers();
}
```

The actual rasterization done in GPU occurs once $glFinish()$ is called. Before rasterizing the rectangle, we perform the specified transformation, which is to compute v', where $v' = M_r v$. We then rasterize the rectangles with transformed geometry, v'.

3.2.4 *Common Questions*

Is there any benefit of using column-major ordering for the matrix over row-major ordering? Not much. Some people prefers to use column-major, while others like to use row-major. Somehow, people who designed OpenGL may prefer column-major ordering.

3.3 *Affine Frame*

In this chapter, we started with viewport transformation, followed by 2D transformation. Overall, an underlying question along these discussions is this: suppose that we have two different frames and we know coordinates of a point in a frame. What is the coordinates of the point in the different frame? For example, the viewport transformation is an answer to this question with the world and viewport frames.

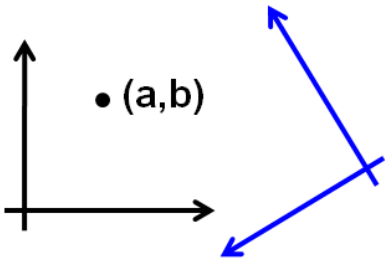


Figure 3.5: What is the coordinate of the point against the blue frame?

We use a set of linearly independent basis vectors to uniquely define a vector. Suppose that $\vec{V}_1, \vec{V}_2, \vec{V}_3$ are to be three such basis vectors represented in a column-wise vector. We can then define a vector, $\vec{X}$, with three different coordinates, c_1, c_2, c_3, as the following:

$$\vec{X} = \sum_{i=1}^{3} c_i \vec{V}_i = \begin{bmatrix} \vec{V}_1 & \vec{V}_2 & \vec{V}_3 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \mathbf{V}c, \tag{3.16}$$

where $\mathbf{V}$ is a 3 by 3 matrix, whose columns corresponds to the basis vectors.

Now let's consider how we can represent a point, $\dot{p}$, in the 3D space. Unfortunately, the point cannot be represented in the same manner as we used for defining a vector in above. To define a point in the space, we need an anchor, i.e., origin, of the coordinate system. This makes the main difference between points and vectors. Specifically, points are absolute locations, while vectors are relative quantity.

A point, $\dot{p}$, is defined with respect to the absolute origin, $\dot{o}$, as the following:

$$\dot{p} = \dot{o} + \sum_{i=1}^{3} c_i \vec{V}_i = \begin{bmatrix} \vec{V}_1 & \vec{V}_2 & \vec{V}_3 & \dot{o} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}. \tag{3.17}$$

Simply speaking, we can define a point by using 4 by 4 matrix $\begin{bmatrix} \vec{V}_1 & \vec{V}_2 & \vec{V}_3 & \dot{o} \end{bmatrix}$, whose each column includes three basis vectors and the origin. As a result, this matrix is also called a affine frame; in this chapter, we will just use a frame for simplicity.

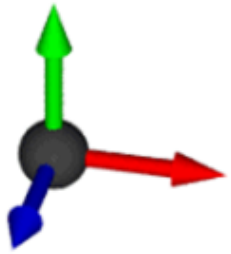


Figure 3.6: The affine frame consisting of three basis vectors and the origin.

We can also define a vector with the frame as the following:

$$\vec{x} = \sum_{i=1}^{3} c_i \vec{V}_i = \begin{bmatrix} \vec{V}_1 & \vec{V}_2 & \vec{V}_3 & o \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 0 \end{bmatrix}. \tag{3.18}$$

Interestingly, the fourth coordinate for any vector with the frame has 0, since the vector is not based on the origin.

Defining points and vectors with the frame has various benefits. Here are some of them:

1. **Consistent model.** Various operations between points and vectors reflects our intuition. For example, subtracting two points yields a vector and adding a vector to a point produces a point. These operations are consistent with respect to our representations with the frame:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ 1 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 0 \end{bmatrix}. \tag{3.19}$$

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}. \tag{3.20}$$

2. **Homogeneous coordinate.** We introduced the homogeneous coordinate to represent the rotation and translation in a single matrix (Sec. 3.2.2). Such homogeneous coordinates are actually defined in the affine frame, and the fourth coordinate indicates whether it represents points or vectors depending on its values.

3. **Affine combinations.** Adding one point to another point does not make sense. Nonetheless, there is a special case that makes sense. Suppose that we add two points with weights of α_1 and α_2, where the sum of those weights to be one, i.e., $\alpha_1 + \alpha_2$. We then have the following equation:

$$\alpha_1 \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} + \alpha_2 \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ 1 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix}. \tag{3.21}$$

Intuitively speaking, this affine combination results in a linear interpolation between those two points. This idea can be also extended to any number of points. One example with three points includes the barycentric coordinate (Sec. 10.2).

3.4 Local and Global Frames

We would like to conclude this section by discussing local and global frames, followed by revisiting the viewport transformation in these frames.

Suppose that you have a point, $\dot{p}$, defined in the affine frame, $\mathbf{W}$, with a coordinate of c; i.e., $\dot{p} = \mathbf{W}c$. We now want to translate the point with $\mathbf{T}$ and then rotate it with $\mathbf{R}$. The transformed point, $\dot{p}'$, is defined as the following and can be interpreted in two different directions:

$$\begin{aligned}\dot{p}' =& \mathbf{WRT}c \\ =& \mathbf{W}(RTc) = \mathbf{W}c' \text{ // use the global frame} \quad (3.22)\\ =& (\mathbf{WRC})c = \mathbf{W}'c \text{ // use a local frame.} \quad (3.23)\end{aligned}$$

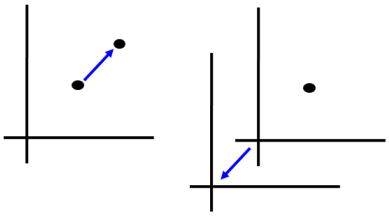


Figure 3.7: Global (left) and local (right) frames of the same transformation.

The second equation is derived by changing the coordinate given the global frame $\mathbf{W}$. The third equation is derived by modifying the frame itself into a new local frame, say $\mathbf{W}'$, while maintaining the coordinate. These two different interpretation can be useful for understanding different transformations.

Let us remind you that we started with this chapter by discussing the viewport transformation. Let's apply local and global frames to the viewport transformation. During the viewport transformation, the point does not move. Instead, we want to compute a coordinate in the viewport space, $\mathbf{V}$, from that in the world space, $\mathbf{W}$. In other words, we can represent them as the following:

$$\dot{p} = \mathbf{W}c = \mathbf{V}c', \quad (3.24)$$

where the relationship between the world and viewport spaces is represented by $\mathbf{V} = \mathbf{WS}$.

In this case, the coordinate c' in the viewport space is computed as the following:

$$\dot{p} = \mathbf{W}c = \mathbf{VS}^{-1}c = \mathbf{V}(\mathbf{S}^{-1}c) = \mathbf{V}c'. \quad (3.25)$$

This approach, considering coordinates with different frames, can be very useful for considering complex transformation. We will use this approach for explaining 3D rotation transformations in the next section.

3.5 3D Modeling Transformation

To create a scene consisting of multiple objects, i.e., models, we need to place those models in a particular place in the world. This

operation is modeling transformation that commonly consists of translation and rotation.

3D translation is extended straightforwardly from the 2D translation:

$$c' = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_y \\ 0 & 0 & 0 & 1 \end{bmatrix} c. \tag{3.26}$$

The rotation in the 3D space along the canonical axis is easily extended from the 2D case. For example, the rotation along the X axis is computed as the following:

$$\mathbf{R}_{\grave{\mathbf{X}}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.27}$$

The 3D rotation against an arbitrary vector requires additional treatments. Nonetheless, the affine frame we studied in Sec. 3.2.3 simplifies this process and we thus discuss this approach here in this section.

Suppose that we would like to rotate a vector, $\vec{x}$, given a rotation axis vector, $\vec{a}$. When the rotation axis aligns with one of canonical X, Y, or Z axis, we can easily extend the 2D rotation matrix to 3D rotation matrix. Unfortunately, the rotation axis may not be aligned with those canonical axes, complicating the derivation of the rotation matrix. We now approach this problem in the perspective of the affine frame. The vector $\vec{x}$ can be considered to be defined in the frame of three basis vectors consisting of $\vec{a}$, the red one, and two other orthogonal vectors, the black and green vectors in the figure.

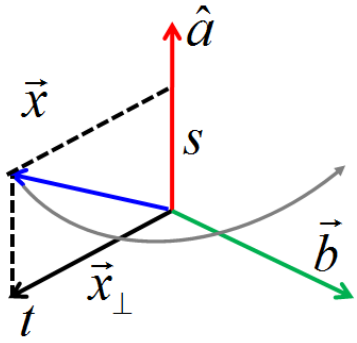


Figure 3.8: Geomtry for 3D rotation.

Let's first compute the black vector $\vec{x}_\perp$, which is orthogonal to $\vec{a}$, and the plane spanned by these two vectors contains the rotation vector $\vec{x}$. We can decompose two coordinates, s and t, of $\vec{x}$ in the plane defined by $\vec{a}$ and $\vec{x}_\perp$, respectively. To compute such coordinates, we can apply the dot product. s and t, and $\vec{x}_\perp$ are then computed by canceling the coordinate of $\vec{a}$, as follows:

$$\begin{aligned} s &= \vec{x} \cdot \vec{a}, \\ \vec{x}_\perp &= \vec{x} - s\vec{a}, \\ t &= \vec{x} \cdot \vec{x}_\perp. \end{aligned}$$

The green vector $\vec{b}$ that is orthogonal to both $\vec{a}$ and $\vec{x}_\perp$ is computed by the cross product between $\vec{a}$ and $\vec{x}_\perp$; i.e., $\vec{b} = \vec{a} \times \vec{x}_\perp$.

So far, we have computed three basis vectors of a local affine frame that can define the vector $\vec{x}$. Specifically, the vector $\vec{x}$ is defined as the

following:

$$\begin{bmatrix} \vec{a} & \vec{x}_\perp & \vec{b} & \dot{o} \end{bmatrix} \begin{bmatrix} s \\ t \\ 0 \\ 0 \end{bmatrix}, \tag{3.28}$$

where $\dot{o}$ is a virtual origin of our local affine frame. The rotation in the amount of θ along the rotation axis $\vec{a}$ is transformed to the rotation along the X axis in the local affine frame. As a result, coordinates of the rotated vector are computed as the following:

$$\begin{bmatrix} \vec{a} & \vec{x}_\perp & \vec{b} & \dot{o} \end{bmatrix} \mathbf{R}_{\grave{\mathbf{X}}} \begin{bmatrix} s \\ t \\ 0 \\ 0 \end{bmatrix}. \tag{3.29}$$

Quaternion is an popular alternative for the 3D rotation, and many tutorials are available for the topic.

4
Camera Setting

In this chapter, we discuss two important aspects of a camera setting: 1) how to setup camera parameters, and 2) how to project objects into a 2D viewing space.

For the simplicity, we discuss these issues with a pinhole camera, one of simple camera setting. Modern cameras employ many different types of lenses and thus are much more complex than the pinhole camera. We also discuss how to extend such realistic cameras in other chapters .

4.1 Viewing Transformation

To see a particular portion of the world scene, it is natural to specify the camera. The camera is specified with its origin, and X, Y, and Z axis in the world space (Fig. 4.1), which define the affine frame of the camera space. The viewable image is then mapped to the X-Y space in the camera space. As a result, the goal of the viewing transformation is to convert the coordinates defined in the world space into those in the camera or viewing space.

Unfortunately, defining those parameters, e.g., X-axis of the camera, in the world space is neither an intuitive nor easy task. Instead, we would like to design an intuitive and easy way of defining those parameters. Following quantities are commonly adopted ones for defining the viewing space:

1. **Eye point,** e**.** This is simply the position of the camera.

2. **Look-at point,** p**.** We typically have a specific target that we want to look at. As a result, requiring such a look-at point is not a big burden to users.

3. **Up-vector,** $\vec{u}_a$**.** While we have the look-at point, the orientation of the camera is not specified. For example, we can look at the target point, while we maintain our head upward or downward.

Figure 4.1: To generate an image, we specify a camera in the world space, which consists of the origin and X, Y, and Z axis of the camera.

As a result, we require to specify an up-vector, $\vec{u}_a$, that define the orientation of the camera.

While we prepared an intuitive way of defining the camera, we still need to define the affine frame of the viewing space. The next goal is to define the affine frame from these parameters, as the following:

1. **Look-at vector, $\vec{l}$.** The Z-direction of the camera can be computed by computing the look-at vector, $\vec{l}$, which is computed by $p - e$ with a proper normalization, $\hat{l} = \frac{l}{|l|}$. Note that we use the hat notation, $\hat{}$, to denote a normalized vector, whose magnitude is one.

2. **Right vector, $\vec{r}$.** The X-axis of the camera is computed by the cross product between the look-at vector $\hat{l}$ and the given up-vector $vecu_a$:

r

$$\begin{aligned} \vec{r} &= \vec{l} \times \vec{u}_a, \\ \hat{r} &= \frac{\vec{r}}{|r|}. \end{aligned} \tag{4.1}$$

3. **Adjusted up-vector, $\hat{u}$.**

   The given up-vector may not be perpendicular to the look-at and right vectors. As a result, we recompute a new up-vector, $\hat{u}$, that is perpendicular to both of them: $\hat{u} = \hat{r} \times \hat{l}$. Since it is difficult and cumbersome for users to specify the initial up-vector in this way, we adjust the up-vector in this way. Usually, this process is performed within a graphics library such as OpenGL.

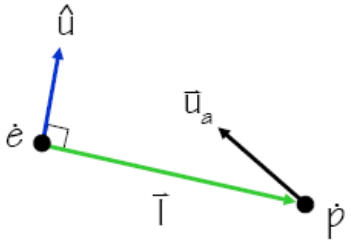


Figure 4.2: Adjusting the initial up-vector.

Let's consider how to transform coordinates in the world space to the viewing space defined in the camera space. This problem is exactly same one that we discussed for local and global frames of Sec. 3.4. As a result, we apply the concept of changing frames to this problem.

Suppose that the coordinate in the world space is c. What we want is to translate the camera origin such that the camera origin becomes the origin in the viewing space. This is represented by $\mathbf{T}_{-\mathbf{e}}$. We then rotate the coordinate with a rotation matrix, $\mathbf{R_v}$, into the camera space. As a result, we have the following equation:

$$\mathbf{W}c = \mathbf{E}\mathbf{R_v}\mathbf{T}_{-\mathbf{e}}c, \tag{4.2}$$

where $\mathbf{W}$ and $\mathbf{E}$ are frames of the world space and camera space. Therefore, the viewing matrix is defined as $\mathbf{R_v}\mathbf{T}_{-\mathbf{e}}$ that convert the world space coordinate c into one in the camera space.

For the world space, we use canonical basis vectors and thus $\mathbf{W} = \mathbf{I}$. Also, the viewing space $\mathbf{E}$ is represented by the three basis vectors. As a result, we have the following relationship:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix} \mathbf{R_v} \tag{4.3}$$

$$\begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix}^{-1} = \mathbf{R_v} \tag{4.4}$$

The matrix of $\begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix} = M$ is an orthonormal matrix, whose columns are orthogonal to each other and unit normal vectors. In this case, $M^T M = I$ is satisfied and thus M^{-1} can be easily computed by M^T. As a result, the rotation matrix $\mathbf{R_v}$ is computed as the following:

$$\mathbf{R_v} = \begin{bmatrix} \hat{r}^T \\ \hat{u}^T \\ -\hat{l}^T \end{bmatrix} \tag{4.5}$$

Given the rotation matrix and translation matrix, the viewing matrix $\mathbf{V}$ is computed as the following:

$$\mathbf{V} = \mathbf{R_v}\mathbf{T}_{-\mathbf{e}} = \begin{bmatrix} \hat{r}_x & \hat{r}_y & \hat{r}_z & 0 \\ \hat{u}_x & \hat{u}_y & \hat{u}_z & 0 \\ -\hat{l}_x & -\hat{l}_y & -\hat{l}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.6}$$

Connections to OpenGL. In an old version of OpenGL, the viewing transformation is setup by calling "gluLookAt $(\cdot)$". This function simply constructs the viewing matrix (Eq. 4.6) and composes it with the current matrix that OpenGL maintains. In a recent OpenGL version, e.g., 3.0, gluLookAt is no longer available, and one needs to maintain their own viewing transformation in a vertex shader . Fortunately, there are many available codes to implement equivalent functions in recent versions of OpenGL.

4.2 *Projection*

Figure 4.3: Orthographic projection.

Projection occurs right after viewing transformation. Projection maps 3D points defined in the camera or eye space into 2D points in the image space. There are two common projection methods: orthographic and perspective projection.

The orthographic projection simply flattens 3D objects into the 2D image space. It preserves parallel lines before and after the projection. It is used for top and side views in various modeling tools (e.g., 3ds Max). It can, however, appear unnatural due to the lack of perspective foreshortening.

In a simplest form, the orthographic projection is defined as the following:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{4.7}$$

As an additional details to the viewing and projection transformation, we also define a view volume for each camera. Fig. 4.4 shows an example of the view volume for the orthographic projection with related parameters defining the view volume. After the orthographic projection, we map those 3D coordinates into ones in the NDC space (Sec. 3.1).

In this context, the orthographic projection mapping to the NDC space is computed as the following:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{-(r+l)}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{-(t+b)}{t-b} \\ 0 & 0 & \frac{2}{f-n} & \frac{-(f+n)}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \tag{4.8}$$

where r, l, t, b, f, n indicates right, left, top, bottom, far, and near, respectively. As a sanity check, when we have a coordinate of $(l, 0, 0, 1)$, it should give us -1 after the orthographic projection. This is verified as the following:

$$x'(l) = \frac{2l}{r-l} - \frac{r+l}{r-l} = -\frac{r-l}{r-l} = -1. \tag{4.9}$$

Note that we do not cancel the Z-coordinate even after the orthographic projection. We actually use the Z-coordinate for an important rendering task, visibility check using the depth buffer (Ch. 7.4).

4.2.1 *Perspective Projection*

Perspective projection is very common in modern computer animation. It, however, takes a long history to be fully understood and

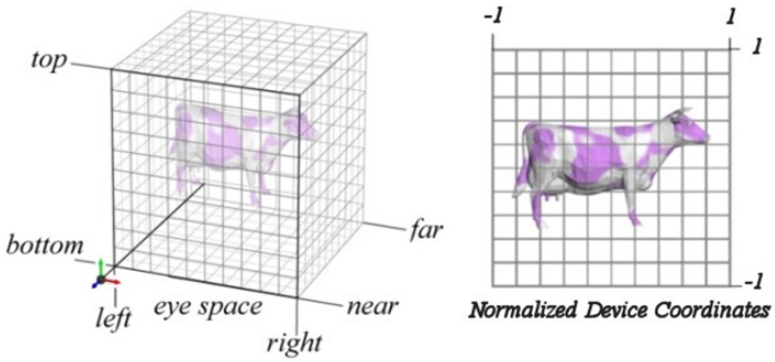


Figure 4.4: The left figure shows a view volume for the orthographic projection. After the orthographic projection, we map 3D coordinates into ones in the NDC space.

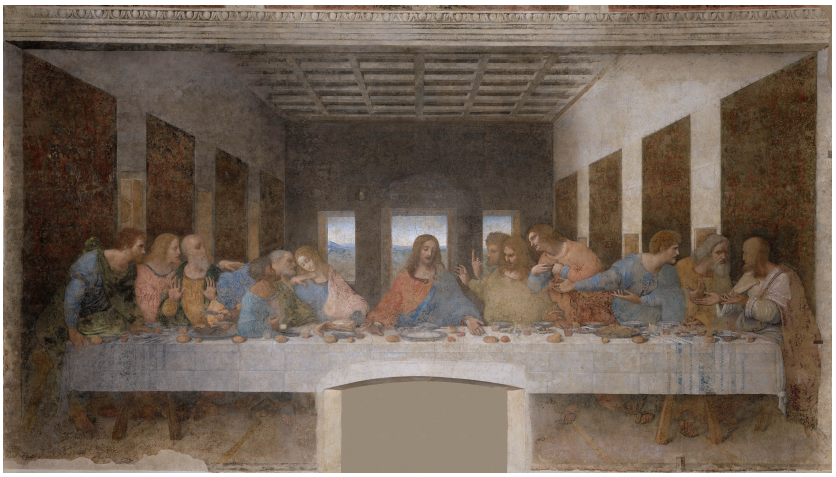


Figure 4.5: This shows the last supper drawn by Leonardo da Vince. This painting shows that objects are drawn under the perspective projection. Furthermore, the vanishing point is located at the Jesus to emphasize the theme of the paining. In other words, perspective projection may be intentionally used for artistic expression.

used in arts. Fig. 4.5 shows an early example of a painting adopting the perspective projection and its intentional use for artistic expression.

A key characteristic of the perspective projection is foreshortening of far-away objects compared to close objects. Another characteristic of perspective projection is that parallel lines in perspective projection always intersect at a point, i.e., vanishing point.

In this section, we discuss such a perspective projection under a simplistic camera model, pinhole camera. Fig. 4.7 shows a 2D schematic illustration of a point into a view plane under a pinhole camera. The point, p, has (y, z) coordinates in the Y-Z world space. Under the pinhole, we can see the point by observing on the ray that is reflected from the point and passes through the pinhole. We draw the ray in the blue color.

In a camera we commonly have some kind of sensors (e.g., camera sensors or film) to capture the light energy that the ray carries at the

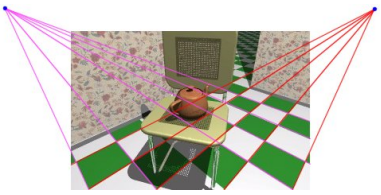


Figure 4.6: Vanishing points.

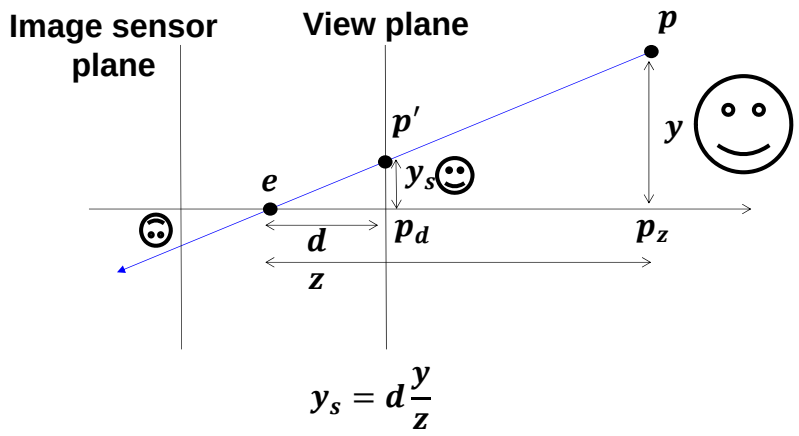


Figure 4.7: This figure illustrates how a point maps in the world space maps to one in the view plane space.

end of the optical systems behind of the eye point, i.e., focal point. In computer graphics, we, however, have such image recording plane in front of the eye position, i.e., the camera center.

Given this configuration of the view plane, our goal is to compute coordinates of the point, p', in the view plane that is projected from the 3D point p. Since the projected point p' is in the view plane, its Z-coordinate is d, which is the distance from the camera origin e to the view plane. The unknown of p' is its Y-coordinate.

To derive this, we apply properties of similar triangles between $\triangle p'ep_d$ and $\triangle pep_z$, and we then have the following relationship based on the same proportion of same sides:

$$\frac{y_s}{d} = \frac{y}{z} \Rightarrow y_s = d\frac{y}{z}, \tag{4.10}$$

where d and z are Z-coordinates of points p_d and p_z, respectively.

The next question is how to represent this equation in a matrix form. The bottom line is how to represent $\frac{1}{z}$ in a matrix form. We address this problem by utilizing the homogeneous coordinate with the following simple matrix form:

$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{4.11}$$

The trick is on the homogeneous coordinate. In this case, the homogeneous coordinate after applying the perspective matrix is set to the depth of the point, i.e., $w = z$. We then have the following the homogeneous divide and accomplish the perspective projection:

$$w = z, x' = \frac{x}{w} = \frac{x}{z}, y' = \frac{y}{w} = \frac{y}{z}, z' = 0. \tag{4.12}$$

The final homogeneous coordinate after the homogeneous divide is $1 = \frac{w}{w}$.

We also define a view volume for the perspective projection and convert it to the unit view volume, followed by mapping to the NDC space. Based on this, we can also setup a perspective projection matrix for the NDC space. In an old OpenGL version, this function is supported by call $glFrustum(\cdot)$ or $gluPerspective(\cdot)$.

4.2.2 Common Questions

Can we support other projections than orthographic and perspective projections? For example, a projection simulating the image observed from bug's eyes? What if this projection is not represented as a simple matrix? Yes, we can support many other projections that are represented as some mathematical equations. Also, current GPU can support arbitrary projections although the projection is not represented as a simple matrix.

I felt that there are something missed in the image generated by using perspective projection. Then, I realized that those images do not have effects like out-of-focusing and in-focusing. How can we support these effects? To correctly simulate these kinds of effects, we need to simulate a lens that we are using in camera. This can be supported by using ray tracing, but may take long computation time. Instead, we can mimic similar effects by considering depth values of rasterized objects. For example, the depth values of the rasterized objects are far away from the user-defined focal depth, we blur the image of the object. This is not a correct solution, but a hacky solution that can run very fast in the rasterzation rendering mode.

5
Interaction

In this chapter, we discuss basic ways of interacting with 3D objects. We first discuss a file format of 3D objects (Sec. 5.1), and how to select and manipulate those objects (Sec. 5.2). We then discuss a simple way of supporting 3D rotation based on a concept of the virtual trackball (Sec. 5.3), followed by handling hierarchically defined models (Sec. 5.4).

5.1 Loading Objects

One can create a 3D object using various modeling tools such as Blender, a free and open-source software, and Autodesk 3ds Max, a commercial tool. Also, many 3D models have been created and available commercially and freely at various websites. As a result, it is also common to load those models and compose a 3D scene with them.

As a step to compose and render such a scene, it is necessary to read and load 3D objects. Many file formats are proposed to enable such operations easily. In this section, we discuss an obj format, one of simplest and widely available formats. A simple example of an obj file format is shown in Frame 5.1.

```
# A simple cube in an obj file format          // strings starting
with # are comments
   v 1 1 1                                     // vertex specification
   v 1 1 -1
   v 1 -1 1
   v 1 -1 -1
   v -1 1 1
   v -1 1 -1
   v -1 -1 1
   v -1 -1 -1
```

```
f 1 3 4                                  // face specification
f 5 6 8
f 1 2 6
f 3 7 8
f 1 5 7
f 2 4 8
```

Basic obj file tokens are explained in below:

- **# comments.** The rest of the line starting with # is comment.
- **v float float float.** It specifies X, Y, and Z coordinates of a vertex.
- **vn float float float.** It defines a normal.
- **vt float float.** It specifies U and V texture coordinates.
- **f int int int ..** It defines a triangle (or other polygon) with vertices with specified indices. These arguments are 1-based indices. When we do not have normal information associated with the triangle, we compute the normal out of the plane passing the triangle. The direction, i.e., inward or outward, of the normal vector depends on the ordering of those vertices (Ch. 6.3). As a result, an extra attention is required on the ordering of vertices.

We can also read and store these files in an ASCII mode or binary mode. It is usually more intuitive for human to use the ASCII mode, since we can effective understand what the file describes. On the other hand, the binary mode has benefits in terms of compact storage and thus fast I/O operations.

Layouts. One can have an arbitrary ordering, i.e., layout, of vertices and triangles. Nonetheless, the layout has been identified to play an important role in terms of performance. Since modern computer architectures adopt a block-based cache, the cache fetches a block containing consecutively located data, when one of those data is accessed. As a result, data that are likely to be accessed together are recommended to be stored closely. This idea leads to cache-coherent and cache-oblivious layouts [1]. .

[1] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-Oblivious Mesh Layouts. *ACM Transactions on Graphics (SIGGRAPH)*, 24 (3):886–893, 2005

5.2 *Selection*

To interact with objects in graphics applications, we first need a way of selecting a particular object in the 3D scene. Suppose that we would like to select an object that the mouse pointer is locating at.

Many possible approaches are possible, and two of them are listed here:

1. **Object-space approach.** Given the point of the mouse cursor, we can imagine a virtual ray passing from the camera origin to the point. We can then identify objects that are intersecting objects and choose the object that has the closest intersection point to the viewer. Overall, this approach is ray casting, which is the basis for ray tracing, a critical component of physically-based rendering (Ch. 10).

2. **Image-space approach.** Since we have a rendered image in the color buffer, we can directly access the pixel in the buffer, where the mouse cursor is located at. Unfortunately, the pixel has only the color of a rendered triangle, not the ID of the triangle. We explain a concept of an item buffer that encodes the ID of each triangle based on a color. This approach unlike the object space method based on ray tracing works on the image buffer and thus is an image-space approach.

It is worthwhile to mention that many graphics problems can be approached in either the object-space, image-space, or even a hybrid approach combining both of them, as described for the selection problem.

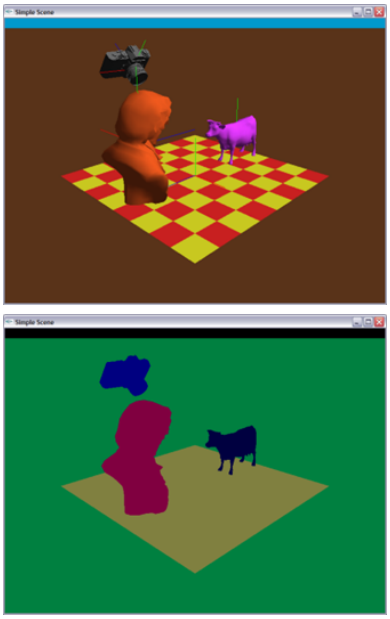


Figure 5.1: The top image shows the color buffer of a scene, while the bottom image shows its item buffer rendered in the back buffer.

5.2.1 *Selection with Item Buffer*

For the selection problem mentioned in the prior section, we want to encode a triangle ID on each pixel on a buffer.

A simple way given the rendering pipeline is to use the concept of the item buffer. The item buffer is simply a different name to the color buffer, with the difference of encoding IDs of triangles, not the original colors of them. To encode an ID for each triangle, we use a unique RGB color value, ID color, for each triangle or each object that serves a smallest selection granularity.

We render all the objects with those ID colors, but we should not show this result to a viewer, since this is not the final result. Therefore, we render it to a back buffer, but do not swap it to the front buffer that is accessed by a display device and thus visible to the viewer. We then read the back buffer by calling an appropriate access function, e.g., $glReadPixels(\cdot)$. Once we fetch the color ID given the chosen pixel, we can identify its associate triangle or object. We then provide a feedback based on the selection operation and render the scene with its original colors.

Note that this selection method works by reading the buffer and thus is categorized by an image space approach. As a result, this

method shows common features of many image space approaches, and some of them are them are:

Accuracy of image-space methods are commonly controlled by the chosen resolution of images.

1. Accuracy depending on the image resolution. A main characteristic of the image-space method is that its accuracy depends on the chosen image resolution, since we identify the triangle ID by accessing an pixel in the item buffer. When we have multiple triangles in a pixel, the pixel can encode only a single triangle. As a result, as we have higher resolutions, we have a higher accuracy in terms of identifying a chosen triangle. Note that when we choose a triangle based on a ray in the object-space method, we do not have such a characteristic.
2. Different performance characteristics to the object space approach. While the image-space method has its accuracy issue, it is commonly used in many different problems including the selection problem, since it is relatively easy to implement and to show high performance, mainly thanks to the support from GPUs. For example, we render triangles and read the buffer through GPU, and thus they can be done quite quickly. Nonetheless, it is less obvious whether this approach has a better time complexity. Specifically, the image-space method using the item buffer explained in this section has a linear time complexity, while the ray tracing based approach using an acceleration structure such as bounding volume hierarchy has sub-linear complexity (Ch. 10.3). As a result, when we have many objects and triangles, the object-space approach can be faster.

In this section, we studied about an image-space selection method using the item buffer. More importantly, we discussed its different characteristics with those of an object-space method using ray tracing.

Figure 5.2: A trackball. The image is excerpted from the homepage of its vendor, Kensington.

5.3 *Virtual Trackball*

In the prior section, we discussed how to pick an object. Once we select an object, we would like to re-position or re-orient the object. For such operations, we can do that through many input devices such as keyboard, mouse, touch screen, etc. For example, many modeling tools (e.g., Autodesk 3ds Max) provide various object and camera manipulations through mouse, which is an inexpensive and widely used input device.

Discussing various interaction operations with available input devices is beyond the scope of this section. Instead, we focus on how to rotate an object in a 3D space. Fig. 5.2 shows a trackball, where a rolling ball is attached. We can use the trackball to intuitively rotate

an object, which is mapped to the ball on the track ball. Unfortunately, the trackball is not widely available compared to keyboard and mouse. We now see how we can support such convenient rotation mechanisms with a mouse.

Suppose that we enclose a sphere on an object that we would like to rotate. Fig. 5.3-top image shows such a configuration. The 2D grid represents our viewing plane. The interaction scenario for rotating the object with the mouse works as the following: 1) the user locates the mouse cursor and clicks a button at a point, $\vec{a}$ [2], and then move and specify the cursor into a different position, $\vec{b}$. Basically, based on this interaction scheme, we want to roll the ball from $\vec{a}$ to $\vec{b}$. The next question is how to compute rotation information, the rotation axis, $\vec{r}$, and its rotation amount, θ, realizing the rolling operation?

Suppose that you grasp the ball from $\vec{a}$ to $\vec{b}$ in your right (or left) hand. The thumb in this case indicates the rotation axis $\vec{r}$. The rotation axis is a vector orthogonal to both $\vec{a}$ and $\vec{b}$, and this can be computed by the cross product between them:

$$\hat{r} = \hat{a} \times \hat{b}, \tag{5.1}$$

where $\hat{r}$ represents a normalized vector whose magnitude is one, i.e., $\hat{r} = \frac{\vec{r}}{|\vec{r}|}$. The rotation angle θ is computed by the inverse of the dot product:

$$\theta = \cos^{-1}(\hat{a} \cdot \hat{b}). \tag{5.2}$$

If necessary, we can also compute a rotation matrix based on computed axis and angle (Sec. 3.5).

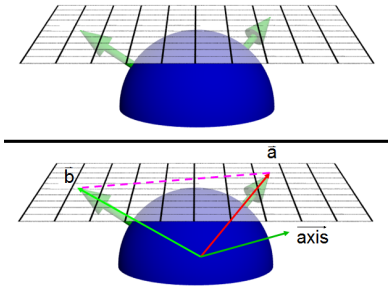


Figure 5.3: Top: we place a ball on an object under the rotation, while the viewing space is touching the ball. Bottom: suppose that we push a button of a mouse at the location of $\vec{a}$ and release it at the another location, $\vec{b}$. In this user input, we want to rotate the ball and its enclosed object from $\vec{a}$ to $\vec{b}$.

[2] We represent this as a vector starting from the ball origin to the point.

5.4 *Transformation Hierarchy*

Some objects have many joints (Fig. 5.4), and we can move each joint independently. In this section, we would like to compute transformations for those parts of an object.

As an example for the study, let's consider an object consisting of two parts with a joint (the rightmost object in Fig. 5.4). Each part is usually defined in its own modeling coordinate; its center is commonly located at the origin, say $(0, 0, 0)$, in its modeling coordinate. We then apply appropriate transformations to those parts to locate it at the world space. Since these parts are defined hierarchically, these transformations are also defined in the same, hierarchical way.

Suppose that $\mathbf{M_b}$ and $\mathbf{M_p}$ are two transformation matrices that converts from the base to the world, and from the part to the base, respectively. In this context, to compute the base, say, its coordinate b in its modeling space, in the world, we compute such transformed locations based on $\mathbf{M_b}b$. For the part, p, we need to apply $\mathbf{M_p}$ to

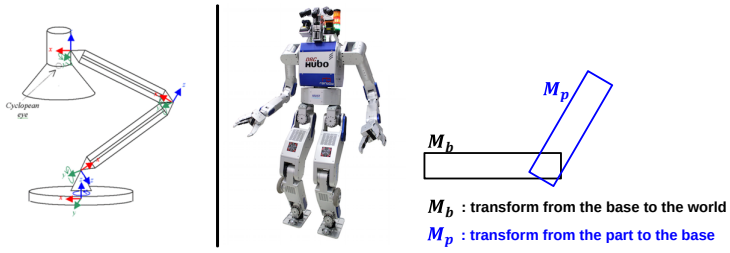


Figure 5.4: The left and middle show two examples of objects with many joints. The left is a lamp with many joints, and the middle shows a humanoid robot, DRC hubo from KAIST, who won DRC (DARPA Robotics Challenge) held at 2015. The right shows an example object consisting of two parts.

locate the part in the base space, followed by $\mathbf{M_b}$ to the world space. As a result, we apply $\mathbf{M_b M_p} p$.

6
Clipping and Culling

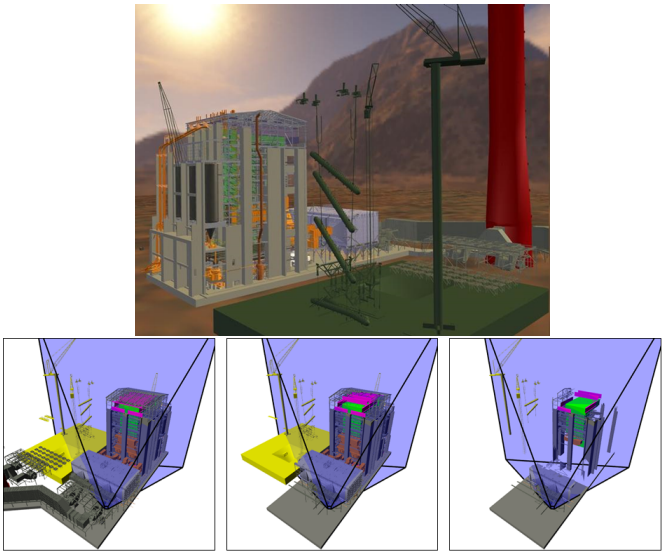


Figure 6.1: The top model shows a coal-fired power plant model consisting of 12 millions of triangles. The model has many pipes within the green, furnace room. It has drastically irregular distributions of triangles across the model ranging from a small bolt to large walls in the furnace. This model is courtesy of an anonymous donor. Bottom images show effects of performing various culling operations. The middle image is the result after performing view-frustum culling to the original power plant model shown in the left. We show these models in a 3rd person view, while the light blue shown in black lines represents the 1st person's view where we perform various culling. The right image shows the result after performing occlusion culling. Since the model has a depth complexity, occlusion culling shows a high culling ratio in this case.

The performance of rasterization linearly increases as we have more triangles. While GPU accelerates the performance of rasterization, it improves only a constant factor, not the time complexity, i.e., growth rate, of the rasterization method. Especially, when we have so many triangles in a scene, it may be prohibitively slow for such scenes. An example includes a power plant scene consisting of 12 millions of triangles (Fig. 6.1).

In this chapter, we discuss two acceleration techniques, clipping and culling, to improve the performance of rasterization. At a high level, their main concepts are:

1. **Culling.** Culling throws away entire primitives (e.g., triangles) and objects that cannot possibly be visible to the user. This is one of important rendering acceleration methods.

2. **Clipping.** Clipping clips off the visible portion of a triangle and throws away the invisible part. This simplifies various parts of the rasterization process.

6.1 *Culling*

Culling conservatively identifies a set of triangles and objects that are invisible to the viewer, and does not pass them to the rendering pipeline. Since the culling process itself can have its own overhead, it is important to design an efficient culling method, while identifying a large portion of invisible triangles among their maximum set.

Fig. 6.1 shows two culling methods, view-frustum culling and occlusion culling, applied to the power plant model. Since this model has a high depth complexity, i.e., many triangles map to a pixel in the screen image, and widely distributed triangles across its scene, such culling methods can be very effective, while they have their own computational overheads. Some of culling methods work as the following:

1. **Back-face culling.** We cannot see triangles heading away from us, unless such triangles are transparent. In opaque models, back-face triangles are blocked by front-face triangles. Back-face culling can be done quite easily and integrated in the rendering pipeline (Sec. 6.5).

2. **View-frustum culling.** The view-frustum (Fig. 6.1) shows an example of the view-frustum and its culling result. Typically, the view-frustum is defined as a canonical view volume within the rendering pipeline and performed by checking whether a triangle or an object is inside the volume or not.

3. **Occlusion culling.** In the case of opaque models, we cannot see triangles located behind the closest triangle to the viewer. As we have more complex models, such models tend to have more numbers of triangles and thus more numbers of triangles map to a single pixel, resulting in a higher depth complexity. In this case, occlusion culling identifies such occluded triangles or objects. Typically, occlusion culling has been more difficult to be adopted, since knowing whether a triangle is occluded or not may require rasterizing the triangle, which we wanted to avoid initially through occlusion culling.

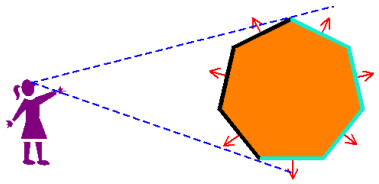


Figure 6.2: Back-face triangles of closed objects are invisible, and back-face culling aims to cull such triangles.

In the next section, we discuss inside/outside tests that are basis for many culling and clipping methods.

6.2 *Inside/Outside Tests*

Many culling and clipping methods check whether a point (or other primitives) is inside or outside against a line in 2D or a plane in 3D. We thus start with a definition of a line for the sake of simplicity; the discussion with the line naturally extends to 3D or other dimensions.

Among many alternatives on definitions on lines, we use the following implicit line representation:

$$\begin{aligned}(n_x, n_y)\cdot(x,y) - d = 0 \rightarrow \\ n_x x + n_y y - d = 0 \rightarrow \\ \begin{bmatrix} n_x & n_y & -d \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0 \rightarrow \\ \bar{l}\dot{p} = 0,\end{aligned} \tag{6.1}$$

where $(n_x, n_y) \equiv \vec{n}$ is a unit normal vector of the line equation and $\dot{p}$ is a point in the homogeneous coordinate. We use $\bar{l}$ to denote coefficients of the line.

Given the line equation, we also define the positive half space, $\dot{p}^+$, where $\bar{l}(\dot{p}^+) \equiv \bar{l}\dot{p}^+ > 0$; we also define the negative half space in a similar way. We use the following lemma for explaining culling techniques.

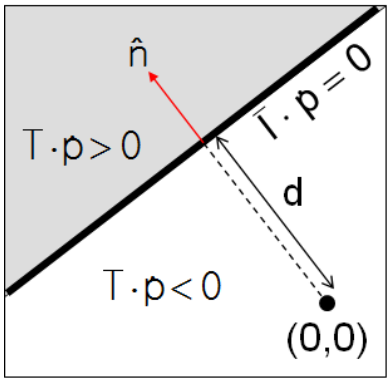


Figure 6.3: Notations of the implicit line equation.

Lemma 6.2.1. *When the normal of the line equation, Eq. 6.1, is a unit normal vector, d gives the L2 distance from the origin of the coordinate system to the line.*

Proof. Let us define (x, y) to be the point in the line realizing the minimum $L2$ distance from the origin to the line, and we then have the following equation:

$$\begin{aligned}(n_x, n_y) = s(x,y), \\ n_x^2 + n_y^2 = 1, \\ s^2(x^2+y^2) = 1.\end{aligned} \tag{6.2}$$

where s is a non-zero constant. Since the point (x, y) is in the line, we have the following equation:

$$\begin{aligned}n_x x + n_y y = d, \\ d = \frac{1}{s}\left(n_x^2 + n_y^2\right) = \frac{1}{s}, \\ = \sqrt{x^2+y^2} \because \text{Eq. 6.2.}\end{aligned} \tag{6.3}$$

□

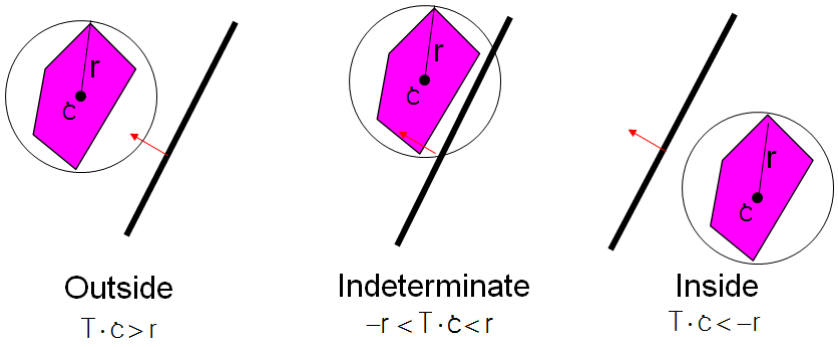


Figure 6.4: This shows three different cases of culling the polygon given its enclosing spherical bounding volume.

In a similar way of proving the lemma, we can see that given a point (x, y) that is or is not on a line whose normal is (n_x, n_y), $n_x x + n_y y$ gives a distance from the point to the line. We also utilize this property for designing culling techniques.

6.3 *View-Frustum and Back-Face Culling*

Let us discuss a simple culling scenario against a line before moving to view-frustum and back-face culling. Suppose that we have a polygon, and we can cull it when the polygon is located totally outside a culling line, as shown in Fig. 6.4. Since it takes a high culling overhead against each vertex of the polygon with many vertices against the line, we use a bounding volume that tightly encloses the polygon.

There are many different types of bounding volumes (BVs) including spheres, boxes, oriented boxes, etc. Commonly, spheres and axis-aligned bounding boxes (AABBs) are frequently chosen bounding volumes, since they are easy to compute with a low computational overhead and a reasonably high culling ratio. Detailed discussions are available in the chapter of bounding volumes and bounding volume hierarchy for ray tracing (Sec. 10.3). In this section, we simply use the sphere for the sake of clear explanation.

Suppose that we use a sphere enclosing the polygon. As a simple culling method in this case, we use its center, c, and radius, r, irrespective of how many vertices the polygon has. Specifically, we test the center against a culling line, $l(\dot{p})$, by plugging its center position to its implicit line equation. There are three different cases (Fig. 6.4), depending on the value of $l(c)$. Since we assume to cull the polygon when it is located outside the line, we focus on this case only in this chapter.

The value of $l(c)$ indicates the $L2$ distance from the line to the center c. When $l(c) > r$ indicates that the sphere is conservatively

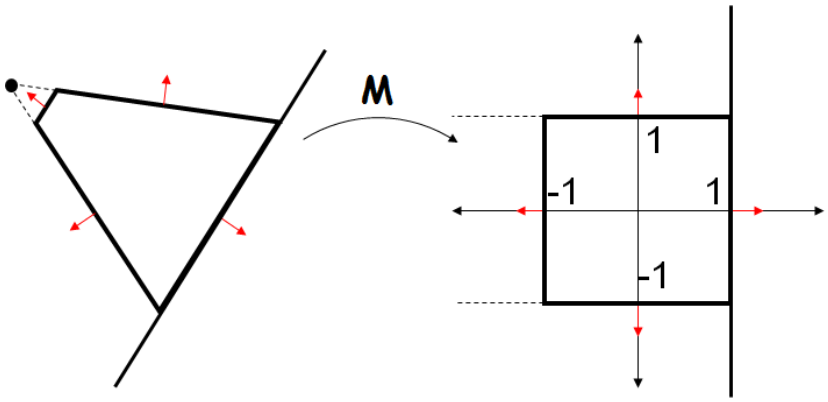


Figure 6.5: The left image shows a view-frustum in 2D, while the right image shows its canonical view volumes. These lines in 2D and planes in 3D of the canonical view are compactly represented and thus can result in fast runtime performance.

outside the line, we can cull it from the further rendering process. With the provided information, it is unclear whether this simple culling operations results in a higher rendering performance than naively rendering all those objects. Nonetheless, we have discussed a basic concept of culling against a line. The performance of this basic approach can be significantly improved by using hierarchically computed bounding volumes, known as bounding volume hierarchy (Sec. 10.3).

Let's see how we perform the view-frustum culling. In rasterization, we assume that we see objects only located within the view-frustum, while this is not the case in reality [1]. Based on this assumption, we can safely cull triangles located outside the view-frustum.

The view-frustum is defined as the left image of Fig. 6.5. We can define such planes with the implicit plane equations, but the view-frustum defined by the given camera setting is transformed to the canonical view volume, which are defined as $x = \pm 1, y = \pm 1, z = \pm 1$, as mentioned in Sec 4.2. The right image of Fig. 6.5 shows the canonical view-volume in 2D.

When a triangle is located outside either one of these six planes, we cull the triangle. This operation applies to each triangle, and is adopted in the rendering pipeline. For large-scale scenes where the view-frustum contains only a portion of them, we can apply the culling method in a hierarchical manner by using a hierarchical acceleration data structure such as bounding volume hierarchy. This approach is more involved and thus a rendering engine supports it.

Back-face culling can be done in a different way. In this section, we discuss a method utilizing the inside/outside tests. One can observe that we cannot see a triangle, when it faces backward (Fig. 6.2). More specifically, suppose that we compute a plane passing the triangle. Then, the triangle is classified as the back-face, when the eye is

[1] We can see other objects that are reflected by objects (e.g., mirror) located within the view-frustum.

located in the negative half side of the plane.

To compute such a plane, we need a normal, the orthogonal vector heading outward to the plane. Given a vertex ordering from v_0, v_1, v_2 in the counter-clock wise, the normal of the triangle, $\vec{n}$, and the distance, d, of the plane is computed as the following:

$$\begin{aligned} \vec{n} &= (v_1 - v_0) \times (v_2 - v_0), \\ d &= \vec{n} \cdot v_0, \end{aligned} \tag{6.4}$$

where the dot product computes the projected distance of the vertex v_0 to the normal direction.

Later, in Ch. 7.3, we discuss a faster back-face culling method, which is more appropriate to be adopted in the rendering pipeline.

Back-face culling in OpenGL. To cull back facing triangles in OpenGL, we use $glCullFace(\cdot)$ after enabling the feature (e.g., GL_CULL_FACE). OpenGL identifies back-face or front-face based on its normal computed from its vertex ordering (Ch. 7.3). OpenGL also provides a way of defining back-face and front-face based on a winding order of vertices between clockwise or counter-clockwise. The counter-clockwise ordering indicates that when we wrap those vertices starting from v_0, passing v_1 to v_2 with the hand, the thumb direction is the front-face. By culling away such back facing triangles, we can avoid to generate fragments from those triangles, resulting in a higher performance.

6.4 *Clipping*

In this section, we discuss clipping that identifies only a visible portion of a primitive, i.e., triangle, and pass it to the following stage (e.g., rasterization stage) in the rendering pipeline.

Let's first discuss a simple case, clipping a line segment consisting of two points, $\dot{p}_0, \dot{p}_1$, against another line, whose coefficient is represented by $\vec{l}$. Our goal here is to identify the clipping point, $\dot{p}$, that intersects with another line $\vec{l}$. To compute the point, we present $\dot{p}$ with a line parameter, t, as the following:

$$\dot{p} = \dot{p}_0 + t(\dot{p}_1 - \dot{p}_0). \tag{6.5}$$

The point should be in another line and thus $\bar{l}\dot{p} = 0$. We then have the following equation:

$$\begin{aligned} &\bar{l} \cdot (\dot{p}_0 + t(\dot{p}_1 - \dot{p}_0)), \\ &t = \frac{-(\bar{l} \cdot \dot{p}_0)}{\bar{l} \cdot (\dot{p}_1 - \dot{p}_0)}. \end{aligned} \tag{6.6}$$

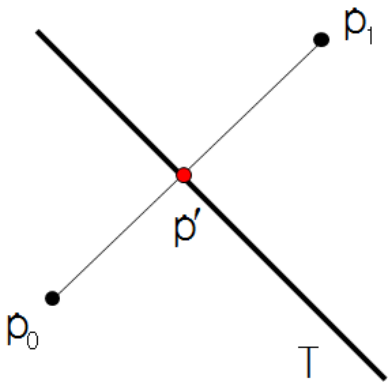


Figure 6.6: A configuration of culling an edge against a line.

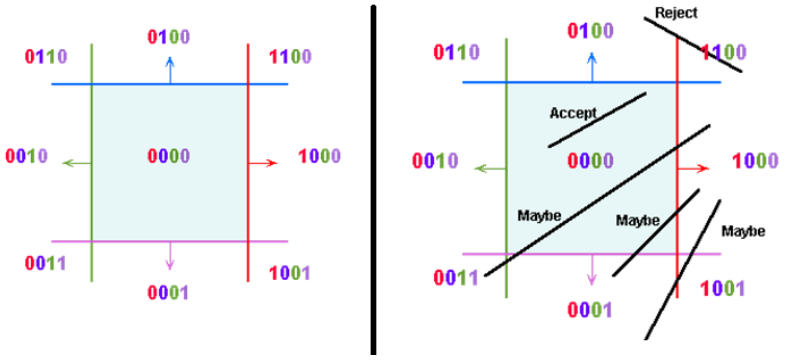


Figure 6.8: The left shows outcodes for each region defined by four lines of the view region. The right shows results of culling edges based on the Cohen-Sutherland method.

Each vertex is also associated with other attributes like colors and texture coordinates. We can also compute those attributes for the clipping point based on the same interpolation method.

Based on this simple line-by-line clipping method, we explain a clipping method, Sutherland-Hodgman algorithm for a polygon including a triangle against a line (e.g., a line of the viewport rectangle) of a convex viewport.

In this method, we traverse each edge of the polygon and check whether the edge is totally inside against the line or not. When it is totally inside or outside, we keep it or throw away it, respectively. Otherwise, we compute two clipping points as shown in Fig. 6.7 and connect them with a new edge. We also apply this process repeatedly against each line of the viewport region.

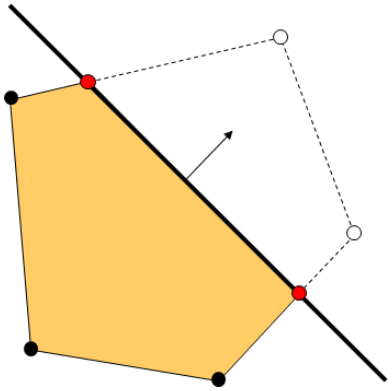


Figure 6.7: The Sutherland-Hodgman method computes a clipped polygon against a line.

6.4.1 *Cohen-Sutherland Clipping Method*

The Cohen-Sutherland method is used to quickly check whether an edge is totally inside or outside given the view region, by using the concept of outcodes. An outcode is assigned to each vertex of primitives, whose each bit encodes whether the vertex is inside or outside against its corresponding line (Fig. 6.8). For example, the first bit in the figure corresponds inside (1) or outside (0) regions against the red line.

When we consider two binary codes, c_1 and c_2, assigned to two vertices of an edge, we have the following conditions and actions:

- If $(c_1 \vee c_2) = 0$, the edge is inside.
- If $(c_1 \wedge c_2) \neq 0$, the edge is totally outside.
- If $(c_1 \wedge c_2) = 0$, the edge potentially crosses the clip region at planes indicated by true bits in $(c_1 \oplus c_2)$. Nonetheless, this could be false positive, meaning that they are identified to be potentially crossing the clip region, but are not actually.

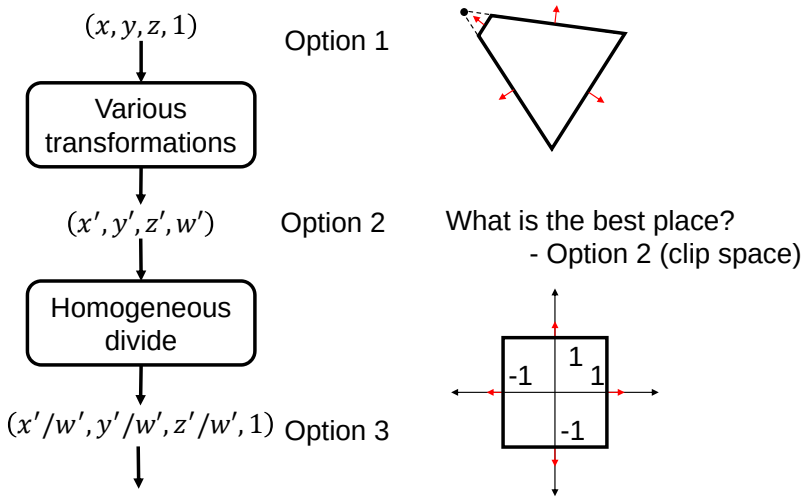


Figure 6.9: This shows different stages of the rendering pipeline with vertex coordinates and view-frustum in each space.

This also applies to a triangle case by utilizing three outcodes computed from three vertices of the triangle.

6.5 *Clipping in the Pipeline*

We discussed how to clip an edge against a plane of the view-frustum before. We would like to now discuss in which stage of the rendering pipeline we perform the clipping operation.

Fig. 6.9 shows how vertex coordinates change as we perform different steps in the rendering pipeline. Overall, there are three different places where we can perform the clipping operation. The first option is the world space where the view-frustum is defined. The second and third options are before and after performing the homogeneous divide.

Each option has its pros and cons. The most intuitive option would be the first one. Also, the third option seems to be good, since the plane equations of the view-frustum in that space are canonical like $x = 1$, and the clipping operation can be done quite quickly. Nonetheless, if we do not clip an edge that spans outside the view-frustum before this stage, the edge flips around due to the projection carried by the homogeneous divide, and generates an unexpected behavior. As a result, the third option is not possible.

Interestingly, the second option has been identified empirically to show the best place to perform the operation, since it does not have the problem of the option three and their plane equations are also defined quite easily. The space of the option two is known as the clip space. Let us discuss how the view-frustum is defined in this

clip space. Specifically, $x'/w' = 1$ in the third space corresponds to $x'/w'w' = w' \rightarrow x' = w'$ in the clip space, which does not depend on the camera setting, and thus can be done efficiently.

[2] As you may realize through this discussion, the rendering pipeline has been heavily tested and optimized to deliver the highest rendering performance. Nonetheless, these choices can change depending on different workloads (e.g., some games use geometry or texture heavily) and hardware performance (e.g., faster memory read or computation).

[2] Structures of the rendering pipeline are not fixed and can be changed for better performance and usability.

6.6 Common Questions

Even though some objects are outside the view frustum, they can be seen though transparent objects or reflected from mirrors. Exactly. The rasterization algorithm is a drastically simplified rendering algorithm over the real interactions between lights and materials. The direct illumination, seen thought primary rays, are well captured by rasterization, while other indirect illuminations are not captured well in the rasterization. To address this problem, many techniques have been proposed in the field of rasterization. However, the most natural way of handling them is to use ray tracing based rendering algorithms.

7
Rasterization

The main idea of rasterization is to project a triangle into the view space and rasterize it into fragments in the color and depth buffers. In this chapter, we assume that vertices of the triangle are projected into the view space, after they undergo various transformations, followed by clipping and NDC transformation.

7.1 Primitive Rasterization

Rasterization is optimized for processing triangles thanks to their simplicity.

For the rasterization process, we commonly use triangles as input primitives, mainly because it is the simplest polygon and simplifies the rasterization process. Nonetheless, these other representations are also decomposed into a set of triangles and fed into the rasterization process.

Rasterization process has two main goals: 1) pixel coverage determination (Fig. 7.1) and 2) parameter interpolation (Fig. 7.5). Given a pixel of the color buffer (or other buffers), we determine whether the pixel belongs to a given triangle or not. Once the pixel is covered by the triangle, we also need to compute its color or other parameters such as its depth value for the depth buffer.

For the coverage problems, many directions are possible. One is to check whether the center of a pixel is inside of a triangle. Another is to measure an area coverage ratio of a pixel against the triangle. The first one is based on a point sample, while the latter one is based on area computation. While the area based computation is more correct, the sample based approach is more efficient, and thus is commonly adopted for rasterization process. They share common pros and cons between point sample based and area based approaches, as we discussed for image-space and object-space methods (Ch. 5.2).

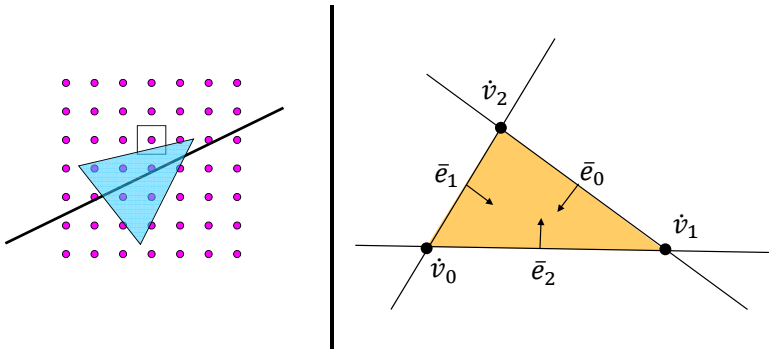


Figure 7.1: The left shows one, pixel coverage determination, of two main goals of the rasterization process. The right shows configurations of vertices and edges of a triangle used for our discussion.

Scanline based triangle rasterization. Some of early techniques for rasterization are based on a concept of scanline, a row of pixels that span a triangle (Fig. 7.2). At those days, the memory was very expensive, and thus having the full resolution of color and depth buffers is not preferred. Instead, these scaneline based approaches maintain a scanline and incrementally update the scanline to raster the whole triangle. Specially, we rasterize an input triangle from top to bottom. Once we meet a vertex of the triangle, we setup the scanline information (e.g., starting and end coordinates shown as red pixels in the figure). For the next scanline, we incrementally update those starting and end coordinates by utilizing slope information of two edges starting from the vertex.

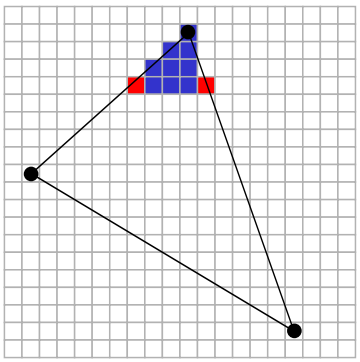


Figure 7.2: This shows a scanline based rasterization. The scanline can be incrementally computed between two neighboring rows.

While this technique was adopted early on, it was identified to show poor scalability to handle scenes with many triangles, since this technique relies on expensive sorting operations and is not friendly for parallelization. Instead, ray tracing and Z-buffer techniques as visible surface determination, i.e., visibility techniques, are prevail techniques in these days (Sec. 10.4).

In the next section, we discuss another rasterization technique combined with the Z-buffer technique.

7.2 *Rasterization with Edge Equations*

In this section, we discuss a rasterization technique for triangles based on edge equations, as shown in the right side of Fig. 7.1. We will see that this approach is simply and friendly for parallelization, to achieve a high performance and thus handle a scene with many triangles.

Let us first compute an edge equation given two vertices, v_0 and v_1, of a triangle (Fig. 7.3). Our goal is to construct an edge equation,

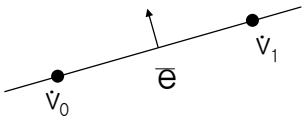


Figure 7.3: An edge representation from two vertices of a triangle.

$\bar{e}$, whose normal vector heads towards the inside of the triangle. Overall, coefficients of the edge equation is given by the cross product between those two vertices:

$$\begin{aligned}\bar{e} &= \dot{v}_0 \times \dot{v}_1 \\ &= \begin{bmatrix} x_0 & y_0 & 1 \end{bmatrix}^t \times \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix}^t \\ &= \begin{bmatrix} (y_0 - y_1) & (x_1 - x_0) & (x_0 y_1 - x_1 y_0) \end{bmatrix} \\ &= \begin{bmatrix} A & B & C \end{bmatrix}. \end{aligned} \tag{7.1}$$

It is not intuitive to compute the edge equation in this way. Here is the rationale. Think of a line passing $\dot{v}_0$ in the homogeneous space, i.e, $(x_0 w, y_0 w, w)$ with an arbitrary value w. We also think another line passing $\dot{v}_1$. The edge in the 2D space maps to a plane in the 3D homogeneous space. Since these two lines and the plane passes the origin, $(0, 0, 0)$, of the 3D homogeneous space, the normal of the plane, i.e., the edge equation, is computed by the cross product between $\dot{v}_0 - (0, 0, 0)$ and $\dot{v}_1 - (0, 0, 0)$.

Once we set the edge equation $\bar{e}$ in this way, points, $\dot{p}$, inside the triangle have $\bar{e}\dot{p} > 0$. We then see that pixels of the triangle reside in the positive half-spaces against three edge equations from the triangle (Fig. 7.1).

While the aforementioned approach is simple enough to identify which pixels are inside a triangle, there are a few special cases requiring certain treatments. They are two cases of sharing edges and vertices.

Sharing an edge. The left image of Fig. 7.4 shows that a shared edge of two triangles passes the center of a pixel. This case arises rarely, but can happen, since there are many pixels, say 1 M pixels when we use a 1 K by 1 K image resolution. When we assign the pixel to both of those two triangles, the pixel color varies depending on an order of rendering those two triangles, which is not a desirable effect. We thus need a tie-breaker assigning only a single triangle to the pixel.

A simple method is to consider the normal of each edge of a triangle and to assign the pixel to either one of them. For example, we can use the following simple tie-breaker:

$$\text{bool} \quad t = \begin{cases} A > 0 & \text{if } A \neq 0, \\ B > 0 & \text{otherwise,} \end{cases}$$

where (A, B) are the normal vector of an edge computed by Eq 7.1. We then assign a triangle to the pixel, when $(\bar{e}(\dot{p}) > 0) \vee (\bar{e}(\dot{p}) = 0 \wedge t)$.

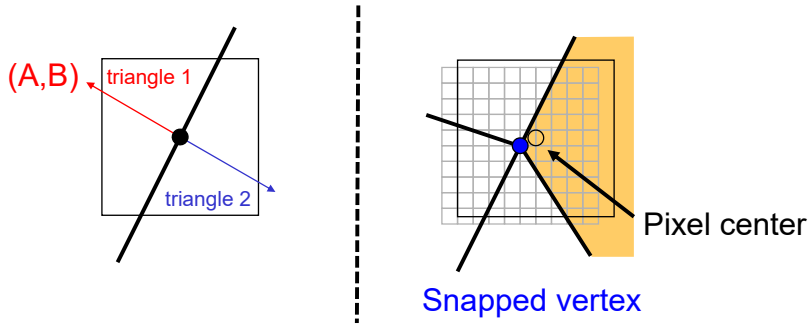


Figure 7.4: This figure shows two cases requiring special treatments for the pixel coverage.

Sharing a vertex. The right image of Fig. 7.4 shows another degenerated case, where a shared vertex of triangles is located at the center of the pixel. For handling this case, one can use a similar tie-breaker that we designed for the shared edge case. Another approach is to snap or quantize vertices of triangles in a way that those snapped or quantized vertices are not aligned with center coordinates of pixels.

7.3 *Interpolation Parameters*

In the last section, we discussed which pixels are covered by a triangle based its edge equations. In this section, we study how to compute colors and other parameters for the pixel, given associated information of the triangle.

Suppose that each vertex has associated information such as color, normal, etc. For the sake of simplicity, we explain various concepts based on the color, especially, red channel information, $r(x, y)$, given a pixel (x, y). Given three red values associated with three vertices of a triangle, we need a way of interpolating these values for a pixel within the triangle. The simplest method is to pick a red value among those three values. While this is simple, it does not produce reasonably high-quality rendering results.

Among many options, we use the linear interpolation from those available values associated with three vertices (Fig. 7.5). The linear red plane is then defined as the following:

$$r(x, y) = A_r x + B_r + C_r, \tag{7.2}$$

where A_r, B_r, C_r are three coefficients of the 2D plane. There are three unknowns and we thus need three equations to compute the plane. Fortunately, these three equations are defined by available

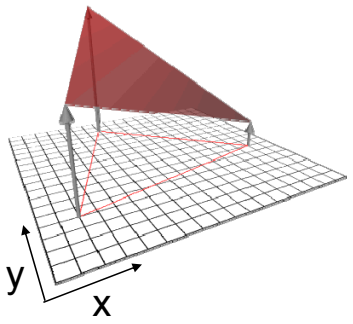


Figure 7.5: This shows the linear interpolation of color values associated with three vertices.

information of three vertices as the following equation:

$$\begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} = \begin{bmatrix} A_r & B_r & C_r \end{bmatrix} \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} A_r & B_r & C_r \end{bmatrix} = \begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} \frac{\begin{bmatrix} (y_1 - y_2) & (x_2 - x_1) & (x_1 y_2 - x_2 y_1) \\ (y_2 - y_0) & (x_0 - x_2) & (x_2 y_0 - x_0 y_2) \\ (y_0 - y_1) & (x_1 - x_0) & (x_0 y_1 - x_1 y_0) \end{bmatrix}}{\det \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix}}, \tag{7.3}$$

where r_0, r_1, r_2 are three red values associated to corresponding three vertices.

An interesting fact is that the area of the triangle, $A_{\dot{v}_0\dot{v}_1\dot{v}_2}$, is computed as the following by utilizing the determinant of a matrix:

$$\begin{aligned} A_{\dot{v}_0\dot{v}_1\dot{v}_2} =& \frac{1}{2} \det \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix} && (7.4) \\ =& \frac{1}{2} \left((x_1 y_2 - x_2 y_1) + (x_2 y_0 - x_0 y_2) + (x_0 y_1 - x_1 y_0) \right) \\ =& \frac{1}{2} (C_{20} + C_{12} + C_{01}), && (7.5) \end{aligned}$$

where C_{20}, C_{12}, C_{01} are coefficients of three edge equations, $\bar{e}_{20}, \bar{e}_{12}, \bar{e}_{01}$, respectively; $\bar{e}_{20}$ indicates the edge equation constructed from $\dot{v}_2$ to $\dot{v}_0$.

Note that when the area is zero, the triangle is invisible. Furthermore, when the area is negative, the triangle is back-facing. If the back-face culling is enabled (Ch. 6.3), we cull the triangle for later rasterization. Otherwise, we flip normals of edge equations and perform later rasterization.

Let's consider the interpolation equation (Eq. 7.3). Actually, other components of the top matrix are coefficients of three edge equations! We then have the following interpolation equation:

$$\begin{bmatrix} A_r & B_r & C_r \end{bmatrix} = \frac{1}{2A_{\dot{v}_0\dot{v}_1\dot{v}_2}} \begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} \begin{bmatrix} \bar{e}_{20} \\ \bar{e}_{12} \\ \bar{e}_{01} \end{bmatrix}, \tag{7.6}$$

where $\bar{e}_{20}$ represents an 1 by 3 vector containing its three coefficients, A_{20}, B_{20}, C_{20}.

Once we compute coefficients of the red plane (Eq. 7.2), we can compute a color on any pixel within the triangle.

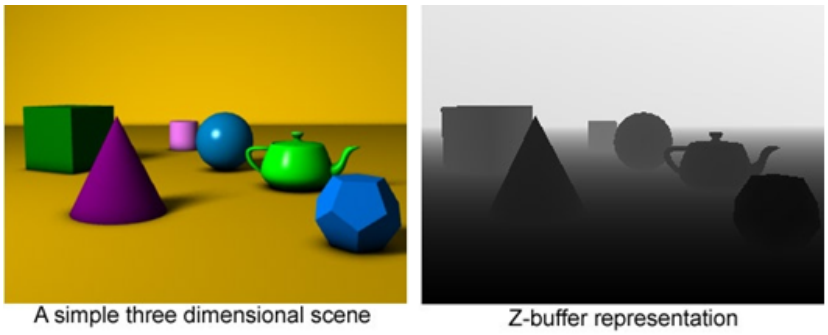


Figure 7.6: The left shows an input scene, while the right image shows its Z-buffer. The white color represents the farthest depth value, 1, while the black one indicates the closest value, 0.

7.4 *Z-Buffering*

The Z-buffer technique is a visibility determination technique, which encodes the depth value of a visible triangle per each pixel. Overall, it is an image-space technique to determine the visible triangle by using a 2D buffer, i.e., depth-buffer.

Z-buffer is one of the most important concepts for rasterization. Simply speaking, we address a complex problem of visibility determination using a 2D map.

Fig. 7.6 visualizes the depth buffer, i.e., Z-buffer, given a scene. The depth buffer simply contains depth values of visible triangles. Note that each vertex of a triangle has its position information (x, y, z). Once we project it to the image space, we also have its depth value in the canonical view volume (Ch. 4.2). The depth value in the canonical view volume spans in the range of $[0, 1]$, where 0 indicates the closest one, while 1 indicates the farthest one.

Given the depth value of each pixel, more correctly, fragment, rasterized from a triangle, we can easily know that whether the fragment has a depth value smaller than the one stored in the depth buffer and thus visible. Once the fragment has a smaller depth value, we update the depth buffer with that depth value at the pixel. We continue this process until we process all the fragments generated from the rasterization process.

As you can see, this Z-buffer is very simple, and thus can be well adopted to a hardware implementation. While there have been many advanced techniques, this Z-buffer technique is the most common technique adopted in rasterization. Nonetheless, recent ray tracing techniques are getting wider attentions thanks to its conceptual simplicity and better functionality supporting realistic rendering effects (Ch. 10).

Processing order. Note that the rasterization method based on the edge equation can be parallelized among different pixels. For example, a rasterization result of a pixel does not depends on anything of another pixel. This opens up various approaches to parallelize the

process for achieving higher performance.

Fig. 7.7 shows two examples of the processing ordering of pixels for rasterizing the triangle. In practice, we identify a bounding box covering the triangle and process the region based on tiles. A tile is a sub-region, say 4 by 4 pixels, of the image space. A GPU core is assigned to process each tile. Different GPU cores process those tiles in a parallel manner, to achieve a high performance. A GPU core assigned to a tile needs to setup three edge equations for a pixel, (x, y), in the tile. For the neighboring pixel, say $(x, y+1)$, we incrementally compute those edge equations, as the following:

$$\begin{aligned} E(x,y) &= Ax + By + C, \\ E(x+1,y) &= A(x+1) + By + c \\ &= E(x,y) + A. \end{aligned} \tag{7.7}$$

So far, we have discussed the rasterization process converting a triangle into a set of fragments. This is one of main concepts of rasterization, setting apart it from ray tracing.

While the rasterization process adopted back-face culling, it can be very slow, especially, when the given scene has so many triangles. There have been many scalable techniques (e.g., mesh simplification) to handle such cases.

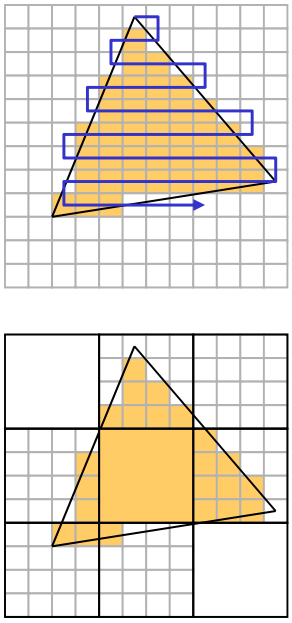


Figure 7.7: Rasterization process can be parallelized, and any ordering of processing pixels or tiles can be possible.

8
Illumination and Shading

In this chapter, we look into basic concepts of illumination and shading. These topics can have different meanings depending on the context. In this chapter, we focus on computing effects of lights for illumination per vertex, followed by applying those illumination results to fill triangles. Before we talk about these concepts, let us first think about how we see things.

8.1 How can we see objects?

Each one of us might have thought about how we can see objects at one point in the past, since seeing things is a part of our daily activity. To fully explain the whole process is beyond the scope of this book. Instead, we would like to point out main components of this process.

At a high level, seeing objects means that we receive the light energy in our eye, which is in the end transferred to our brain. Let us first talk about the light. The light is electromagnetic waves, and our human eyes see only a portion of the spectrum of those electromagnetic waves, commonly called visible light (Fig. 8.1). Visible light refers to wavelengths in a range of 400 and 700 nm.

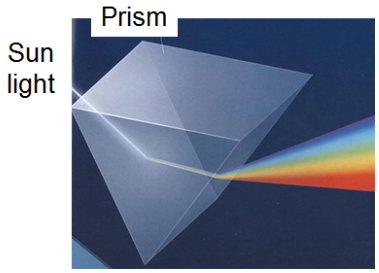


Figure 8.1: This illustrates that the sun light is composed of different wavelengths, which are perceived in different colors. The image is excerpted from the Newton magazine.

Our eye has multiple layers and one of them is retina, which contains photoreceptor cells sensing the visible light. There are mainly two types of such cells: rod and cones. The rod cell is extremely sensitive to photons and can be even triggered by even a single photon. The rod cells give information mainly about intensity of the light, while cone cells are about the color information. There are also three types of cones, each of which responds to different wavelengths, which we call red, green, and blue colors (Fig. 8.2). In reality, color does not exist, but based on response levels from these three cone types, our brain reconstructs the color.

Color does not exist in reality. Instead, our brain reconstructs based on response levels of red, green, blue cones.

Now let's consider how the light interacts with materials. This process can be explained in different levels including quantum physics, but in this chapter, we give only a high level idea on the process.

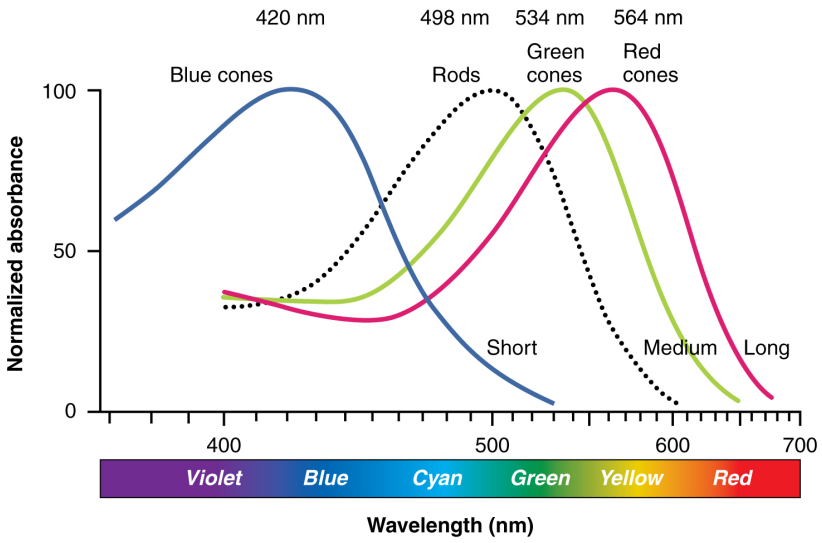


Figure 8.2: This figure shows the response level of rod and cone cells as a function of wavelengths. This figure is available from Anatomy and Physiology under the Creative Commons Attribution 3.0 license.

Once a material or an atom receives a photon, the atom enters into its excited sate. It then returns back to its normal state, while emitting its energy into the space. The energy can be interpreted into another photon or wave, thanks to the duality of the light. The key factor that we need to know is directions of the emitted photon and their wavelengths that determine the perceived color.

As an concrete example, please consider a leaf shown in Fig. 8.3. The incoming sun light is a mix of various electromagnetic waves with a set of different wavelengths, and thus can be perceived as a white light. Once the sun light hits with the leaf, the leaf receives its energy, which is used for its photosynthesis and dissipated as heat. Nonetheless, some of its received energy is emitted into waves (and particles) with different wavelengths. In this leaf case, the emitted energy has the wavelengths corresponding to the green color. As a result, we see the green color to the leaf. Furthermore, the emitted energy is distributed into all the possible directions, and thus we can see the leaf in any directions towards it.

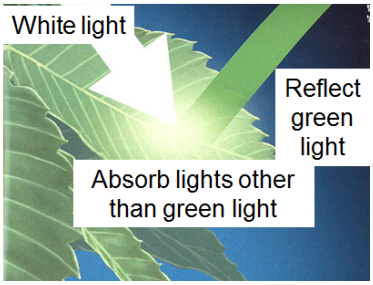


Figure 8.3: This illustrates how the leaf interacts with the coming light. This image is excerpted from the Newton magazine.

8.2 *Bi-Directional Reflectance Distribution Function*

Depending on materials, they have different reflectance behaviors. For example, the chalk is diffuse, meaning that it reflects the light in all possible directions. We thus see the chalk in any view directions. On the other hand, when we look at the surface of an apple, there can be a highlight, a bright spot, when we have a particular view direction. We call such materials to be glossy.

BRDF is used to explain the reflection behavior of a material.

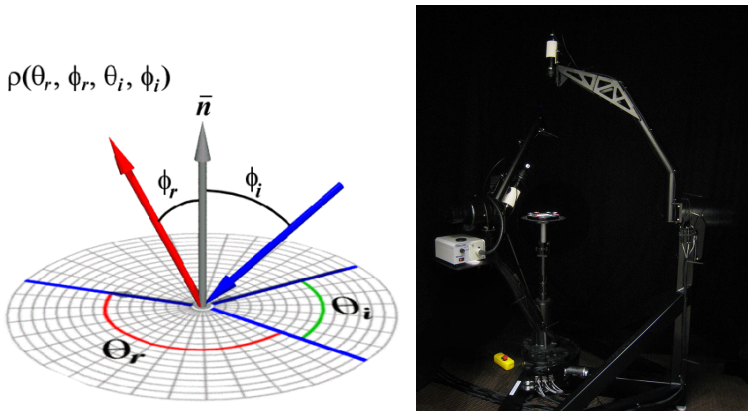


Figure 8.4: The left image shows incoming and outgoing directions, which are four parameters of a BRDF. The right image shows a gonioreflectormeter measuring the BRDF; it is from Univ. of Virginia.

Since materials have different reflectance distributions, we need a function to encode such behaviors. Bi-directional reflectance distribution function (BRDF) is introduced to meet the requirement. BRDF, $f(\cdot)$, is defined over incoming light direction and outgoing light directions. Each direction in the 3D space is encoded with two parameters, θ and ϕ. As a result, BRDF is a four dimensional function (Fig. 8.4). Detailed explanation on BRDF is available at the chapter on radiometric quantities (Ch. 12).

Gonioreflectormeter is used to measure BRDF, by rotating a light source and sensor location (Fig. 8.4). This approach takes very long time, and thus it is one of active research areas to efficiently measure the BRDF.

Measured BRDF itself can be very large in terms of memory footprint. It is thus common to encode and use them in a compact representation. In the following section, we discuss one of most simple illumination models.

8.3 *Phong Illumination Model*

The Phong model is an empirical model, but is used a lot for its simplicity.

The Phong illumination model is a simple and classical illumination model that is adopted in early versions of OpenGL. This model is just empirical, not based on physics, and does not even preserve basic physical assumption such preserving energy. Nonetheless, it has been commonly used thanks to its simplicity.

The Phong illumination model has the following three components:

- **Ambient term.** The ambient term represents a kind of background illumination, and works as a constant value (Fig. 8.5). Specifically, for computing the reflected ambient illumination, $I_{r,a}$, it multiplies an ambient reflectance coefficient, k_a, to an incoming ambient illumination, $I_{i,a}$, i.e., $I_{r,a} = k_a I_{i,a}$. Intuitively, this is a drastic

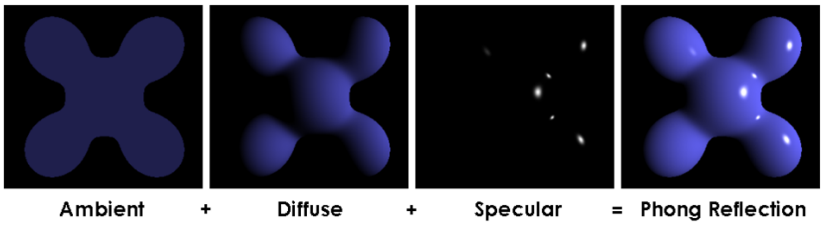


Figure 8.5: This shows different terms of the Phong model. This figure is made by Brad Smith.

simplification of complex inter-reflection between lights and materials. Simulating this term well is a critical component of global illumination, while it is drastically ignored in the Phong model.

- **Diffuse term.** Most objects can be seen in any view directions, and this indicates that they are diffuse. The diffuse term aims to support this visual phenomenon.

- **Specular term.** Certain objects such as metals show strong highlights in a particular viewing direction. The specular term simulates this feature.

Before we discuss diffuse and specular terms in a detailed manner, let's first discuss light sources, which are also mentioned when we explain the ambient term. For the ambient term, we use an ambient light that virtually emits light energy to every location of triangles. We discuss point and direction light sources, followed by briefly mentioning other types of light sources.

Point and directions light sources. The light direction plays an important role on computing illumination. A point light source emits light energy from a single point, p_l. The point light may seem too crude approximation compared to light sources that we encounter in real life. Nonetheless, we can approximate them by using a set of point light sources.

The light direction, $\vec{L}$, on a point, p, on a surface is then computed as the following:

$$\vec{L} = \frac{p_l - p}{|p_l - p|}. \tag{8.1}$$

Note that the light direction varies depending on the location of the surface p.

Unlike the point light, the directional light is located far away from the observer, and thus the light direction is considered as a constant, irrespective of observing locations. The directional light can be thought as a point light source whose location is set at infinity.

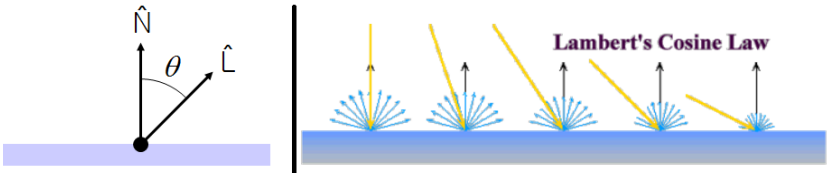


Figure 8.7: The left shows the configuration of the Lambert's cosine law, and its effects are shown in the right.

For example, sun is located far away from us, and thus we use the directional light to represent the sun light source.

Area light sources are a common type of light sources. The area light has a certain light shape with an area and thus can generate soft shadows (Fig. 8.6). Directly considering area lights is more complex than working with point lights. A simple approximation to an area light is to generate a set of point lights on the area light. The number of generated point lights defines illumination levels of soft shadow.

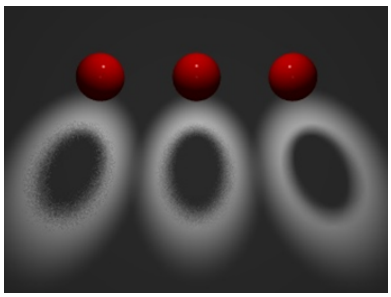


Figure 8.6: Area lights.

Diffuse term. Many objects have the diffuse property, and that is why we can see them! Here we assume the ideal diffuse material; the chalk is close to such a material. The ideal diffuse material reflects an incoming energy into all the possible directions with the same amount of energy. As a result, the reflection becomes view-independent. This diffuse property is caused by a rough surface in a microscopic level, and is perceived as the uniform distribution in the space in the macroscopic level.

The diffuse term is explained by the Lambert's cosine law.

The Lambert's cosine law explains how an incoming energy is reflected depending on the configuration between the surface and the light direction. Suppose that we want to compute the reflected energy I_r on a surface having a normal $\vec{N}$ given the light direction $\vec{L}$. The reflected energy I_r is then computed as the following:

$$\begin{aligned} I_r &= I_i \cos\theta \\ &= I_i(\vec{N} \cdot \vec{L}), \end{aligned} \tag{8.2}$$

where θ is the angle between two vectors of $\vec{L}$ and $\vec{N}$. Fig. 8.7 shows the configuration of these vectors.

Note that as the reflected energy becomes the highest, when the light direction is aligned with the surface normal. Fig. 8.7 also shows how the reflected energy behaves as we have different light directions. When we have a material-dependent, diffuse coefficient, k_d, the reflected energy of the diffuse term is $I_{r,d} = k_d I_i(\vec{N} \cdot \vec{L})$.

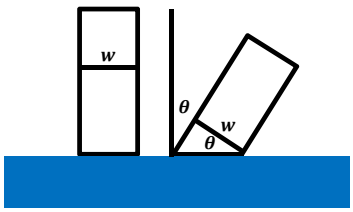


Figure 8.8: The configuration for the Lambert's cosine law.

Proving the cosine law. Let us see how to prove the Lambert's cosine law. Suppose that we have a beam of light with a width of w and energy of I (Fig. 8.8). In this case, the light density per unit area

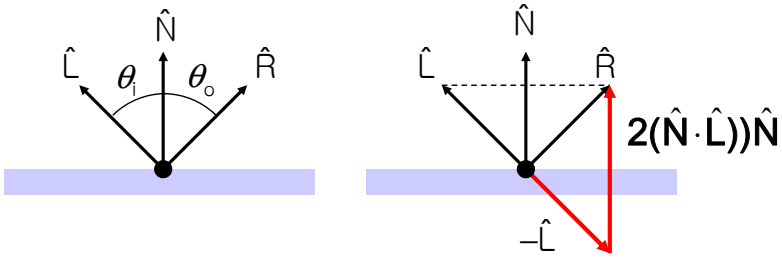


Figure 8.9: This shows how the reflected light direction is computed for the perfect specular object.

is $\frac{I}{w}$. We then lean the beam as an amount of θ. The area receiving the light energy become larger, and takes $\frac{w}{\cos\theta}$. Its light density is then $\frac{I\cos\theta}{w}$. As a result, we can see that the light density reduces as an amount of $\cos\theta$.

Specular term. Let us first consider a perfect mirror-like object. In this case, the reflected light angle is same to the incoming light angle (Fig. 8.9). This is explained by the Snell's law, which is described in a detailed manner in Ch. 10.1. Under the ideal specular material, the reflected light direction, $\vec{R}$ is computed as $2(\vec{N}\cdot\vec{L})\vec{N}-\vec{L}$.

The ideal specular material rarely exists in practice. More common objects are glossy materials, which have highlight along a particular direction and spreads its energy out from the direction. Specifically, when the viewing direction, $\vec{V}$, is on the ideal reflected direction $\vec{R}$, the viewer sees the highest illumination. We then reduce the energy as the viewing direction is away from $\vec{R}$. To capture this observation, the Phong illumination uses the following specular term:

$$\begin{aligned} I_{r,s} =& k_s I_s(\cos\phi)^{n_s} \\ =& k_s I_s(\vec{V}\cdot\vec{R})^{n_s}, \end{aligned} \tag{8.3}$$

where k_s, I_s, n_s are material-dependent specular coefficient, intensity for the specular component of a light, and specular exponent, respectively. Fig. 8.10 shows example results of the specular term.

The Phong model describes materials by treating them to have ambient, diffuse, and specular properties together.

The final Phong illumination is computed by summing these different terms, ambient, diffuse, and specular terms, of different lights (Fig. 8.5). Note that most common objects are described by combining these terms, not a single term.

Local and global illumination. While the Phong illumination is not a physically-based model, it has been widely used for its simplicity and efficiency. Nonetheless, it has a fundamental issue, a local illumination model. The Phong illumination achieves its

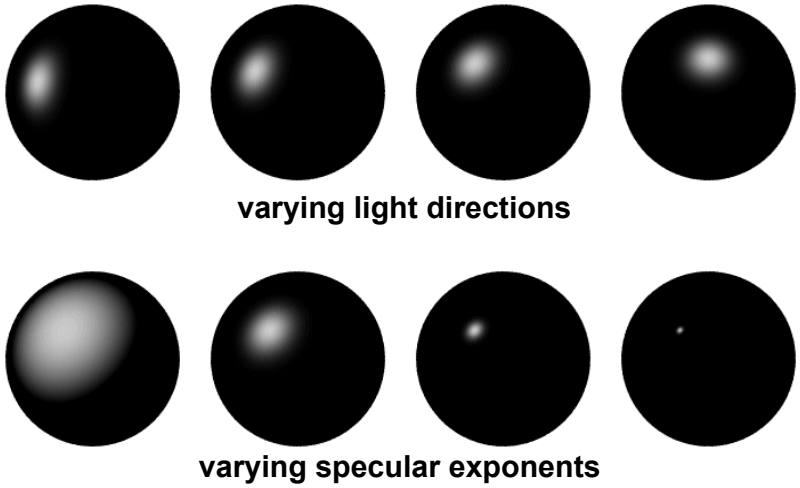


Figure 8.10: This shows how the illumination changes as a function of a viewing direction and specular exponents of the specular term of the Phong illumination model.

efficiency by considering only the local information such as the surface normal, viewing information, and the light information.

The Phong model and rasterization considers the rendering process locally for efficiency.

When there is a blocker occluding the light energy, it, however, generates shadows, which is not considered at all at the Phong illumination. To support such effects, we need to consider the reflected energy or blocked energy from other geometry. This requires us to access global information, which slows down the overall processing. Rasterization is designed in a way to reduce such global access for achieving a high performance and thus the Phong illumination has been well suited for rasterization. We discuss how to generate shadow within rasterization in Ch. 9.4.1. A more physically based approach is to use ray tracing techniques, which are explained in Part II.

8.4 Shading

Shading is a process of adjusting a color of a primitive based on various information such as the normal of the primitive and its angle to the light or view direction. Shading can refer to the illumination process and cover broader approaches including various effects (e.g., lens flare), which are implemented by shader programs. While shading is a broad topic, we discuss how to compute colors within the primitive (e.g., triangle) in this section.

Common shading (or interpolation) methods are as the following:

- **Flat shading.** For flat shading, we use only a single color for the primitive. As a result, each triangle in this approach looks to be constant, i.e., flat, in the image space. To perform flat shading, we use a normal of a triangle and perform the Phong illumination

model or other illumination models to compute the single color. This is the simplest and fastest approach.

- **Gouraud shading.** This approach provides a smooth rendering result by computing different colors for vertices of a primitive and interpolating them within the primitive. While this approach shows smooth rendering results, it comes with performing three independent illumination for vertices and interpolation.
- **Phong shading.** This approach interpolates normals of vertices and computes colors with them within the primitive by evaluating an illumination model with interpolated normals. Since we interpolate normals of vertices, we can generate highlight within the triangle, even though we do not have such highlight in each vertex. When we use the Gouraud shading in this case, we cannot generate the highlight within the primitive, since we interpolate colors of vertices. Fig. 8.11 shows difference between Gouraud and Phong shading methods.

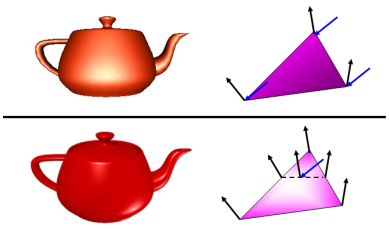


Figure 8.11: This shows Gouraud (top) and Phong (bottom) shading methods. Blue arrows indicate the locations of evaluating an illumination model.

8.5 Common Questions

We have learned that we compute an illumination value for each vertex. For smooth objects (like the teapot model shown in the slide), it should be okay. But, when we draw a box, then the box may look smooth, not showing different and discontinuous colors between neighboring faces of the box. If we use a normal for each vertex of the box, we may get such smooth rendering result, which is not correct one. Instead, we use multiple vertices for each point of the box. For example, we use different vertex data for each point in different faces of the box. Note that these vertices should have the same positional value, but they can have different normals, which can generate discontinuous colors between different faces. Refer to the slide of "Decoupling Vertex and Face Attributes via Indirection" in the lecture slide of "Interacting with a 3D world".

Is there any techniques that can show better quality than the Phong Illumination and can be used in interactive games? I want to know techniques that can show near physically-based illumination that can be used in games? Ambient occlusion has been proposed as an approximation for physically-based global illumination. It can be pre-computed and used quite quickly at runtime, leading to be suitable for interactive games. Moreover, in some CG movies, this technique has been used.

9
Texture

Achieving higher realism has been one of main goals of computer graphics. For this goal, we have developed many modeling techniques by using more triangles, lights, and materials. Unfortunately, using additional resources (e.g., triangles and lights) come with sides effects such as additional running time and memory overheads.

Texture mapping adds additional details, without much overheads.

Since achieving the interactive performance has been another main goal of computer graphics, various approximation rather than directly relying upon additional geometry and lights has been studied. Among various techniques, texture mapping has been one of main approximation techniques (Fig. 9.1).

One thing that we need to understand is that while textures are originally designed for representing complex shapes of geometry, they can be utilized for various other purposes. At a high level, a texture is simply a 2D array, which is one of the most simplest data representations in computer architectures, and can be readily precomputed and used at runtime. Note that the Z-buffer used for visibility determination can be considered as a type of a texture. Thanks to these nice properties, textures have been widely used.

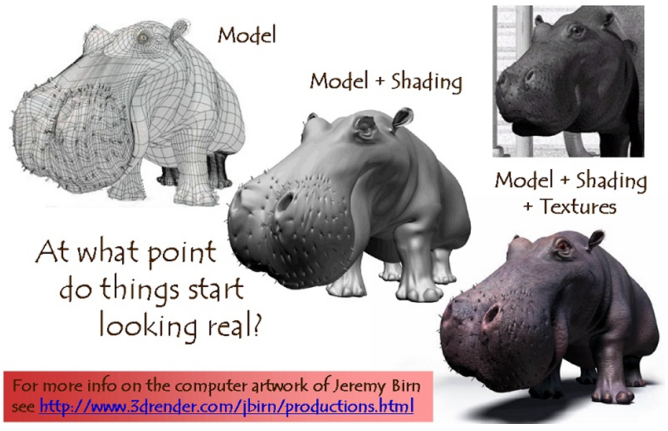


Figure 9.1: Texture mapping adds a lot of details to the geometry illuminated by lights, enabling higher realism without adding much overheads.

We first explain the main purpose of using texture mapping, followed by its various applications.

9.1 *Texture Mapping*

A texture is a 2 D (or 3 D and even a higher dimensional) buffer, whose each element represents a pixel color or some other values. Commonly a texture refers to a 2D image. Texture mapping indicates a mapping from a part of the texture to a part of a model. Fig. 9.3 shows an example of a 2D texture mapping.

We use 2 D texture coordinates, commonly (u, v), to locate a particular location of a 2D texture. We link the texture location to a particular location or a vertex of a mesh by using 2D or higher geometry coordinates (e.g., 3D coordinate (x, y, z) of a vertex). Fig. 9.2 shows that we map (u, v) coordinates of multiple texture locations to a 2D mesh, i.e., quad in the 2D space, represented by (x, y) coordinates. You may also recall that we use vt token to specify (u, v) texture coordinates to a vertex for an obj file format (Ch. 5.1).

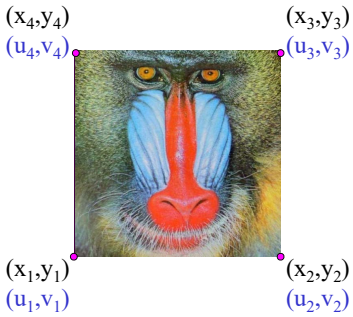


Figure 9.2: Texture mapping.

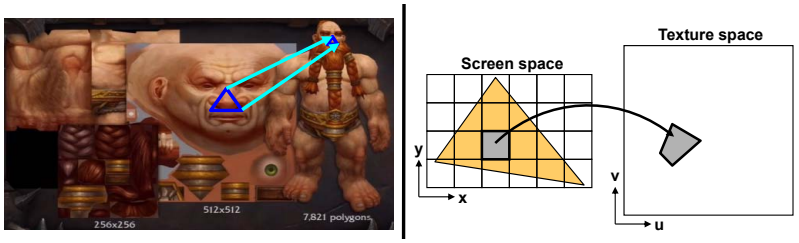


Figure 9.3: The left figure shows a mapping from a texture to a triangle representing a part of a character. We use a few texture maps with different resolutions. The image is excerpted from a MMO-champion site. To perform texture mapping, we compute a representative color of a pixel within the triangle from the texture space, as shown in the right figure. Note that a pixel of the triangle maps to an arbitrarily shaped quadrilateral in the texture space.

To apply the texture mapping, we compute a texture coordinate of a fragment of a triangle, while rasterizing the triangle. We compute the texture coordinate by interpolating texture coordinates associated with vertices of a triangle, as we did for other attributes (e.g., color) (Ch. 7.3).

Once we compute the texture coordinate, we compute the 2D indices of the corresponding texture pixel, known as texel, and use the color of the texel for illumination or other purposes.

Perspective-correct interpolation. Note that a naive interpolation of various vertex attributed in the image space does not provide the expected results that are supposed to be computed by the object-space interpolation. To achieve the correct result even in the image-space interpolation, the perspective-correct interpolation is developed.

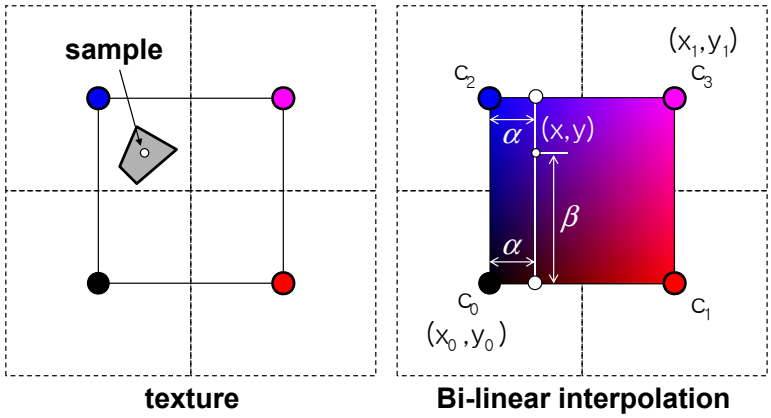


Figure 9.4: The left image shows the case of oversampling. The sampling in the texture space is too small compared to the texture resolution. The right image shows the bi-linear interpolation to address the oversampling problem.

9.2 *Oversampling of Textures*

Let's take a look at the right image of Fig. 9.3. A box-shaped fragment generated for rasterizing a triangle maps to a quadrilateral, i.e., a polygon with four sides, instead of the uniform box shape. This phenomenon occurs due to various transformations (e.g., modeling and projective transformations) and the angle of the triangle against the view direction.

Oversampling occurs when we zoom in the triangles.

Since the pixel in the image space does not match with that in the texture space, we have two cases: oversampling and undersampling cases. Oversampling refers to the case where the sampling resolution in the texture space is smaller than the available resolution of the texture. Fig. 9.4 shows this oversampling case. The quadrilateral in the texture space is even contain in a texel of the texture. The oversampling issue occurs when we magnify the geometry.

We aim to reconstruct the original signal out of available texture samples and compute the signal value at the sampled texture location.

Please take a moment to think about how to compute the representation color for the quadrilateral. Surprisingly, this kind of issues is quite common in computer graphics, image processing, etc. A simple method is to identify the nearest neighbor texel center and use its color for the quadrilateral. In the case of the left image of Fig. 9.4, the blue pixel is the closest to the quadrilateral, more exactly, the sampling point location. Note that during the rasteriation, we compute colors or other attributes based on center positions of pixels.

While this nearest neighbor approach is quite fast, its visual quality is poor, especially along the boundary of texels. In other words, when two sampling locations are very close, but are in different texels, they get different colors, resulting in visual gaps in the image space (Fig. 9.5).

Another approach is to use linear interpolation. Given the sam-

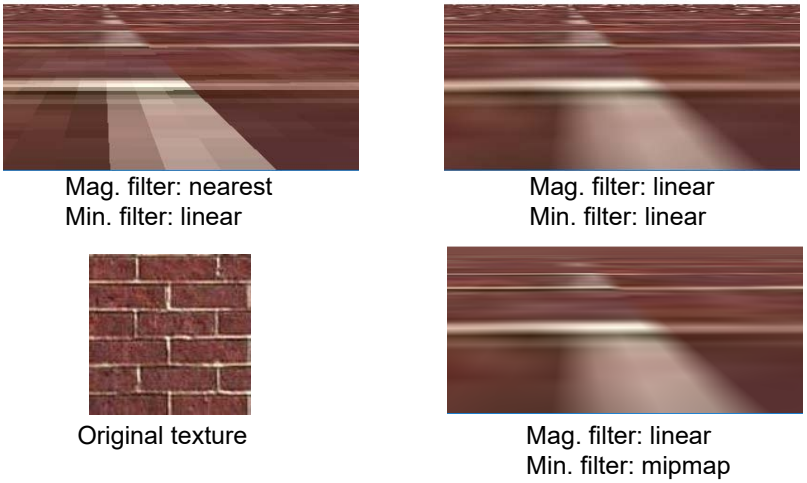


Figure 9.5: Different rendering results with different mag. and min. filters for texture mapping.

pling location, we identify four nearby texels, e.g., c_0 to c_3 in the right image of Fig. 9.4. We then perform the linear interpolation along the U and V texture directions, thus named as bi-linear interpolation. Let us define α and β to be a blending factor along X and Y directions, respectively. They are then defined as the following:

$$\alpha = \frac{x - x_0}{x_1 - x_0}, \beta = \frac{y - y_0}{y_1 - y_0}. \tag{9.1}$$

The color, c, at the sampling location under the bi-linear interpolation is computed as follows:

$$c = (1 - \beta)\left((1 - \alpha)c_0 + \alpha c_1\right) + \beta\left((1 - \alpha)c_2 + \alpha c_3\right). \tag{9.2}$$

The effect of using the bi-linear interpolation over the nearest neighbor one is shown in Fig. 9.5. We can see that boundary shapes of texels were smoothened. Nonetheless, we can also see that the edge information inherent in the original texture was filtered out too. We can thus see that there are trade-off in terms of filtering unnecessary edges and preserving original edges. This boils down to the classical reconstruction and sampling problem.

9.3 *Under-sampling of Textures*

Under-sampling arises when we zoom out the geometry, and thus a fragment of a triangle maps to a large area in the texture space.

Let us know discuss the other sampling issue, undersampling. Undersampling rises when we zoom out from the geometry and thus each triangle become small in the image space. Therefore, a pixel of a triangle maps to a large quadrilateral area in the texture space. The problem is thus to compute a representative color out of many texels covered by the quadrilateral.

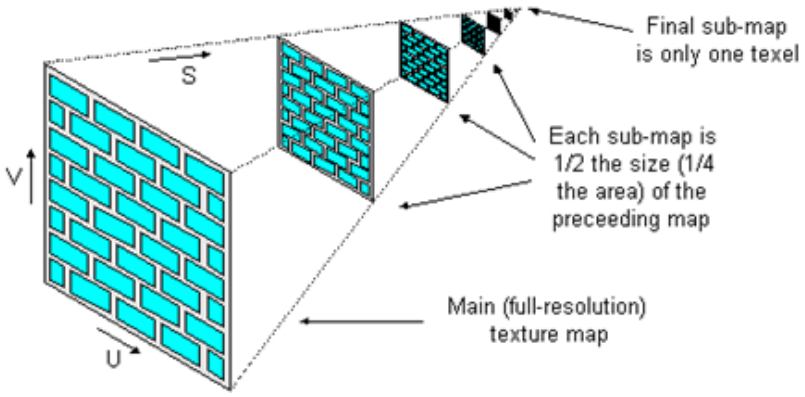


Figure 9.6: This shows a mipmap or image pyramid of an image.

A naive approach to the undersampling is to compute all those texels under the quadrilateral and compute a representative color value, e.g., the average value of them. This approach, unfortunately, slows, since it requires us to access many texels and computations. Instead of this on-demand approach, pre-filtering that pre-filters the original texture in a way to efficiently handle undersampling has been more widely used and studied. In this section, we discuss two approaches, mip mapping and summed area table, for the undersampling problem.

Mipmap or mipmapping is a multi-scale representation for a texture (or any other types of images) to efficiently handle the undersampling issue. Given an input image, a mipmap is composed of a sequence of images whose U and V resolutions are reduced half over its higher resolution (Fig. 9.6). As a result, mipmap is also called an image pyramid. Each low-resolution image is a pre-filtered version of its higher one.

At runtime when we use the mipmap, we pick a particular image level among the available image resolutions of the mipmap given the required texture resolution. If necessary, we can also perform interpolation between two image resolutions, resulting in tri-linear interpolation for computing a color for the sampling location. In whatever cases, we access only a few samples on the mipmap and get pre-filtered texture values, resulting in faster and better visual quality.

The memory requirement of using a mipmap is $\frac{1}{3}$, about 33%, since the total size of using the mipmap is computed as the following:

$$\sum_{0}^{\infty} \left(\frac{1}{4}\right)^{i} = \frac{1}{1-\frac{1}{4}} = \frac{4}{3}. \tag{9.3}$$

Fig. 9.5 shows different rendering results w/ linear filtering or

mipmap. By using the mipmap, we get smoother image results over linear filtering for far-away regions where we minify the geometry. Again, the mipmap is a fast way of handling the undersampling problem, but can remove the original edge information.

A reason why the mipmap produces a over-smoothed result is that the mipmap computes its image pyramid only based on an isotropic filtering shape, i.e., square shapes. As a result, when we have have an very elongated quadrilateral shape in the texture space, we cannot find filtering resolutions along both U and V directions. A solution to this case of anisotropic filtering is the summed area table.

Summed-area table. A summed-area table is proposed to support anisotropic filtering, specifically, a rectangular shape, not the squared shape, on the texture space. Given a texture, $T(u, v)$, its summed-area table, $S(u, v)$, is computed by summing all the elements whose elements are smaller than u or v:

$$S(u, v) = \sum_{i \leq u \wedge j \leq v} T(u, v). \tag{9.4}$$

We then compute the average color value, c_a, on a rectangular regions, e.g., the blue region given by $[u_0, u_1] \times [v_0, v_1]$ as shown in Fig. 9.7, as the following:

$$c_a = \frac{T(u_1, v_1) - T(u_1, v_0) - T(u_0, v_1) + T(u_0, v_0)}{(u_1 - u_0)(v_1 - v_0)}. \tag{9.5}$$

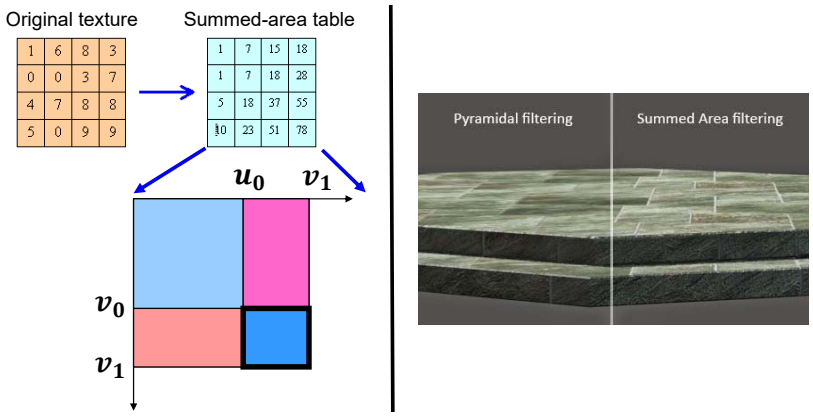


Figure 9.7: The left images show a configuration of the summed area table, while the right image shows rendering results of the summed area table and mipmapping. The right image is created by Denny.

Fig. 9.7 also compares the rendering results computed by the mipmap and summed-area table. The summed-area table shows better quality, since it provides anisotropic filtering. Nonetheless, it has additional runtime and memory overheads over the mipmap.

9.4 *Approximating Lights*

It is easy to paint on images and capture images than constructing geometry, and thus textures have been widely used for various applications. In this section, we discuss two techniques, shadow mapping and environment mapping, of using textures for approximating complex lights.

Before we move on to them, let us first discuss light maps. Light maps are images that contain light intensity. We then use these light maps as textures for adjusting colors of triangles. A simple method of computing colors with a light map is to multiply the intensity contained in the light map with the color computed by illumination or other functions. Fig. 9.8 shows an example of using textures and light maps. Creating complex lighting effects requires high computation, and thus pre-computing, also called baking, them in light maps are still commonly used in many interactive applications.

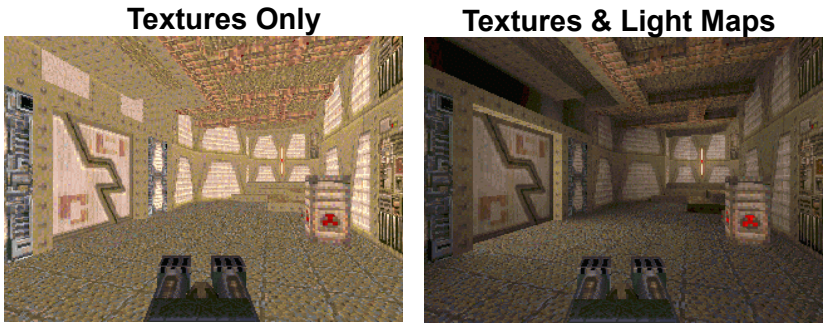


Figure 9.8: This shows results only with textures and both with textures and light maps used for a game called Quake.

9.4.1 *Shadow Mapping*

Shadow is one of fundamental lighting effects that we can see in daily life, and provide various 3D depth cues. While providing shadows is important, it is not that easy to efficiently and correctly generate shadows in rasterization. This problem has been studied for many decades and shadow mapping as a type of texture mapping is proposed for creating realistic rendering results within the rasterization framework.

Please recall our discussion on the Phong illumination model (Ch. 8.3). The model has three components of ambient, diffuse, and specular terms. Unfortunately, diffuse and specular terms do not consider any other objects that block lights from light sources, while the ambient term is a drastic simplification by using a constant for considering inter-reflection. Essentially, the Phong illumination model does not consider the case of having shadows, i.e., the existence of

other objects that block the light. This is mainly attributed since the Phong illumination model, more importantly, rasterization itself, is a local model that mainly aims for high efficiency.

Our challenge is to generate shadows within the rasterization framework. While considering shadows itself requires us to access other objects, resulting in global access on various data, we approach this problem as a two-pass algorithm using shadow mapping. Its main concept is shown in Fig. 9.9.

Shadow mapping is a two pass rendering method to generate shadows without global and random access on other objects.

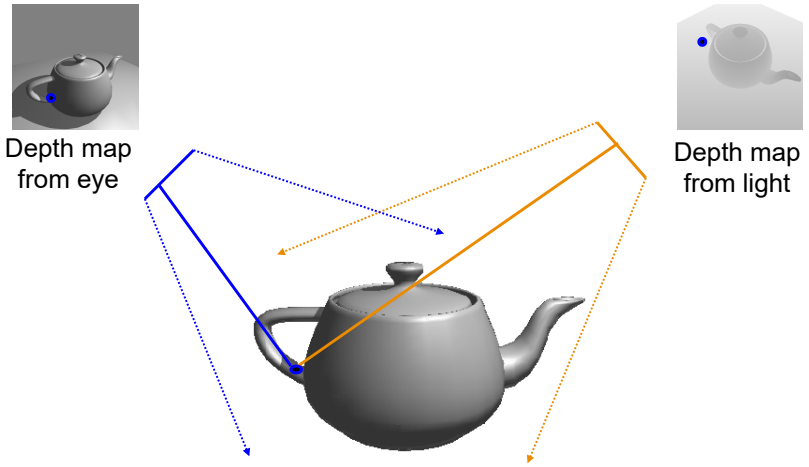


Figure 9.9: This visualizes the process of using shadow mapping to generate shadows on the rendering result seen by the eye.

The problem of the rasterization process is that when we perform an illumination on a fragment of a triangle, we cannot know whether the fragment can receive the light energy from a light source. When we do not receive the light energy due to a blocking object, we add only the ambient term, since the diffuse and specular terms become zero. To know whether a fragment can receive the light energy from a light source, we rasterize the whole scene at the position of the light source and treat its depth map as a shadow map for the light. This is the first-pass of generating shadows.

The depth map generated from the light position contains depth values of visible geometry from the light. We then raster the whole scene at the viewer's position, similar to the regular rasterization process. This is the second pass of our method. A difference in this second pass compared to the regular rasterization is that we check whether we can receive the light energy on a fragment that we generate at the second pass.

To know whether the fragment receives the light energy or not, we compute its depth from the light position, d_l. When d_l is bigger than the stored depth value, d, of the shadow map, we determine that the fragment cannot receive the light energy and we thus give only the

ambient term to the fragment, not the diffuse nor specular terms.

While we explain shadow mapping in a concise manner above, there are a lot of technical issues. Most of them are related to the oversampling and undersampling that we discussed for texture mapping; note that the shadow map is another type of textures and thus inherits issues of texture mapping. Nonetheless, it is very important to understand how we address a kind of global illumination, shadow generation, through a texture, the shadow map.

Shadows maps are just a type of textures and thus inherits pros. and cons. of texture mapping.

9.4.2 *Environment Mapping*

In the prior section, we discussed how to generate shadows using shadow mapping. Another common rendering effect is to support reflection on mirrors or other metal-like objects. For those models, we see other objects reflected on such reflecting objects. In other words, supporting this effect belongs to a type of global illumination requiring the access to other objects.

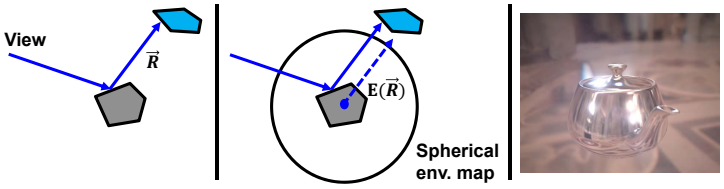


Figure 9.10: Two images in the left visualize how we use the spherical environment mapping, while the right image shows an example of using the environment map to simulate the reflection effect.

The rasterization framework also relies upon using another type of texture mapping, environment mapping, for this reflection effect. Suppose that we have a view direction on a reflecting object shown in the gray color in Fig. 9.10. When the object is the specular object, the reflected ray, $\vec{R}$, is computed by the Snell's law (Ch. 8.3). We then need to access an object along the reflected ray, $\vec{R}$. Unfortunately, this is a ray tracing process, and is not efficiently adopted for rasterization.

To enable the reflection efficiently, we introduce environment mapping, which captures colors of the surrounding environment in a texture. For environment mapping, we can use different types of geometry capturing the environment. Examples include sphere, cube maps, etc. In this chapter, we explain environment mapping based on a sphere for the sake of the simplicity.

An environment map captures surrounding geometry or lights, and can be used as a texture to approximate them at runtime.

As shown in the middle image of Fig. 9.10, we place a sphere at the center of the reflection object. We map the sphere into a 2D texture space; since we can represent the sphere with two angles, θ and ϕ, the 2D texture space can be constructed by these two angles. We then generate a ray starting from the center of the sphere to each texel of the sphere and encode the color of the ray at that texel; we

use a projection instead of ray tracing for efficiently building the map.

At runtime, when we raster a triangle of the reflecting model, we know the viewing direction, and thus identify a texel ID that the reflection ray from the center of the sphere, $E(\vec{R})$, will access. Unfortunately, since the environment map is generated at the center of the object, not each location that we have reflection, there are visual gaps between the computed one and the ground truth. Nonetheless, we can support an approximate reflection by using an additional texture.

The environment map is also used to encode complex types of lights and used for providing realistic lighting for rasterization.

9.5 *Approximating Geometry*

Textures are also used to approximate complicated geometry. Especially, when we have many geometry, it requires long running computation time with high memory requirement. A single or multiple textures are effective ways of approximating them with reduced running and memory overheads.

Figure 9.11: We use the bump map (shown in the middle) to adjust normals of the geometry during the rasterization, to enrich the appearance of the model (shown in the right.) Since we do not change the actual geometry, we can see that the geometry is unchanged at its silhouette.

Bump and normal mapping. Bump mapping modifies normals of geometry, not the actual geometry. The texture used for bump mapping encodes an amount of changes to normals of the geometry (Fig. 9.11). This is an approximate, yet effective way of enriching the geometry. Nonetheless, we can observe that the actual geometry is not aligned with the adjusted normals, especially when we look at the silhouette of the object. Normal mapping is similar to bump mapping, but the normal map directly gives the normal that we use on top of a simple geometry (Fig. 9.12).

Displacement mapping. Unlike bump and normal mapping, displacement mapping adjusts the actual geometry based on a provided displacement map. A common usage of displacement mapping is to encode a height change on the displacement map and adjust the geometry along its normal direction according to the height. Adjusting the geometry requires tessellation, subdividing the geometry into

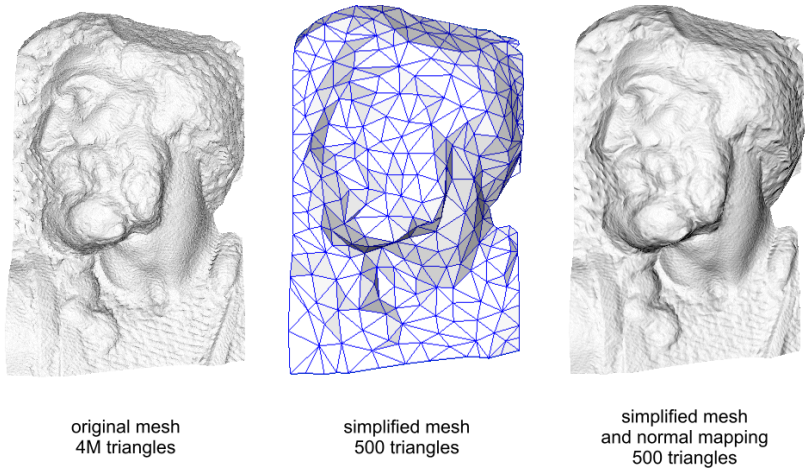


Figure 9.12: We can provide detailed look on a simple geometry by using normal mapping. The image is created by Paolo Cignoni.

smaller patches and adjusting them to accommodate the given height (Fig. 9.13).

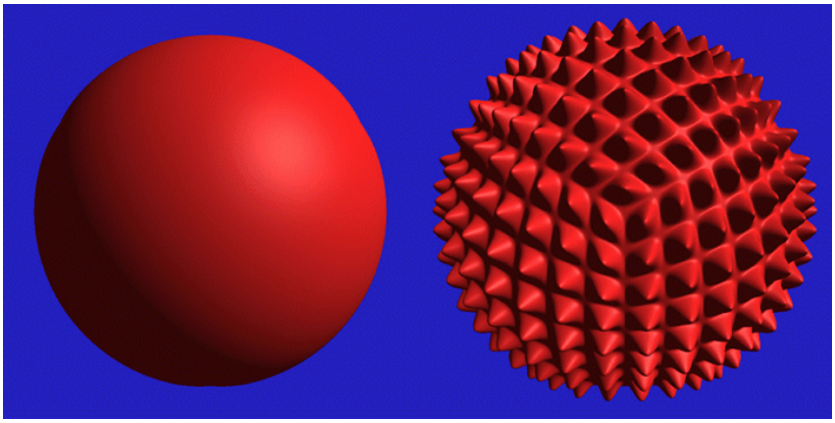


Figure 9.13: Displacement mapping changes the actual geometry according to its map unlike bump mapping. To enable displacement mapping, we tessellate the initial geometry into smaller ones.

We covered only a few examples of approximating geometry. Other notable examples include 3D or solid textures representing 3D shapes and billboards, which are a set of 2D textures representing complex geometry (e.g., trees).