Real-Time Rendering Note

Ianaesthetic

October 9, 2017

1 Introduction

2 The Graphics Rendering Pipeline

Pipeline: to generate, or render, a two-dimensional images in a three dimensional environment.

2.1 Architecture

A pipeline consists of several *stages*. Rendering speed depends on the *bottleneck (the slowest pipeline stage).*

Conceptual Stages: Including *Application Stage*, *Geometry Stage*, *Rasterizer Stage.*

Functional Stages: Has certain tasks to perform without specifying the way that task is executed in the pipeline.

Pipeline Stages: The actual implementations of the functional stages. It may combine or divide functional stages in order to get better performance.

The rendering speed is expressed in *frames per seconds (fps)* or *Hertz (Hz).* Hertz is for hardware. The output fps is determined by both pipeline and display.

2.2 The Application Stage

Application Stage is executed on the CPU and can be fully controlled by developers. At the end of the application stage, *rendering primitive* including points, edges and other geometric elements are generated and passed onto geometry stage.

Application stage hasn't got any substage, and is charge of collision detection, input interaction and acceleration algorithm.

2.3 The Geometry stage

Geometry stage is responsible for per polygon and vertex operations, including several pipeline stages.

$$\textit{Model \& View Transform} \longrightarrow \textit{Vertex Shading} \longrightarrow \textit{Projection} \longrightarrow \textit{Clipping} \longrightarrow \textit{Screen Mapping}$$

2.3.1 Model and View Transform

every model has its own *model space*, and is associated with *model transforms* determining its orientation and positions. Thus, a single model can have different copies called *instances.*

$$\textit{Model Space} \xrightarrow{\textit{Model Transform}} \textit{World Space} \xrightarrow{\textit{View Transform}} \textit{Camera Space}$$

World Space is unique. In Camera Space, the camera locates at the origin and faces the negative z direction commonly.

2.3.2 Vertex Shading

The appearance of objects are related to materials and light. *Shading* determines the effect of light on specific material using *shading equation.* Those computations are performed in geometry stage per vertex or in rasterizer stage per pixel.

2.3.3 Projection

Projection transforms the view volume to a unit cube(*canonical view volume*). Projection methods involve *orthographic* and *perspective.*After Projection, the models are said to be in *normalized device coordinate.*

2.3.4 Clipping

Primitives that are partially inside the canonical view volume require *clipping* as only the part inside the view volume will be rendered. The previous stages are performed by programmable processing unit, while clipping is performed by fixed-operation hardware.

2.3.5 Screen Mapping

Screen Mapping maps normalized device coordinate to screen coordinate. DirectX 9 and its predecessors define the center of first pixel as $(0, 0)$. The successors of DirectX 10 and OpenGL define the center as $(0, 0)$, resulting a convenient conversions:

$$d = \boldsymbol{floor}(c)$$
$$c = d + 0.5$$

Different API puts the first pixel in different places.

2.4 The Rasterizer Stage

Given the transformed and projected vertices and associated shading data, the Rasterizer Stage will set the color of every pixels. This process is called *rasterization* or *scan conversion.*

Triangle Setup → *Triangle Traversal* → *Pixel Shading* → *Merging*

2.4.1 Triangle Setup

In this stage the differentials and other data for the triangle's surface are computed. This stage is performed by fixed-operation hardware dedicated to this task.

2.4.2 Triangle Traversal

Finding which samples or pixels are inside a triangle is often called *Triangle Traversal* or *Scan Conversion.*

2.4.3 Pixel Shading

Any per-pixel shading is here and resulting more color data passed to next stage. This stage is executed by programmable process unit. A lot of important technics such as *texturing* is employed here.

2.4.4 Merging

The information for every pixel is stored in *color buffer. Merging* is responsible for combine the fragment color with the color in the buffer and visibility. This stage is not programmable but highly configurable. Visibility is done by *Z-Buffer* or *Depth-Buffer* by update the smallest z value. However, when rendering transparent primitives, all opaque primitives must be rendered first and then rendered other in a **back-to-front** order.

The *alpha channel* is associated with color buffer and stores a related opacity value. The *stencil buffer* is an off-screen buffer used to record the locations of rendered primitives. It derives from primitives and can be used to control rendering into color buffer and z-buffer. All functions above are called *raster operation* or *blend operation.*

There are also *frame buffer*(consists of all the buffer), *accumulation buffer*(complement to frame buffer), *double buffer*(for display).

2.5 Through the Pipeline

3 The Graphics Processing Unit

Graphics processing unit(*GPU*) is different from previous rasterization-only chip, evolve from configuration implementations of complex fixed operation to highly programmable blank-slates. Programmable *shader* is the primary means.Vertex Shader and Pixel Shader allow operations per Vertex and pixel. Geometry shader allows create and destroy primitives on the fly.

3.1 GPU Pipeline Overview

Vertex Shader	*fully programmable*
Geometry Shader	*fully programmable*
Clipping	*configurable*
Screen Mapping	*completely fixed*
Triangle Setup	*completely fixed*
Triangle Traversal	*completely fixed*
Pixel Shader	*fully programmable*
Merger	*configurable*

The Geometry shader is optional.

3.2 The Programmable Shader Stage

Modern shader stages share a *common shader core*.The common shader core is the API and unified shaders is a GPU features. Shaders are programmed using C-like *shading language* which will be compiled to *intermediate language*(*IL*), and IL will be converted to machine language through drivers. IL can be seen as a virtual machine, with 4-way *single-instruction multiple-data*(SIMD) representing positions, vectors, textures. *Swizzling*, the replication of any vector component, and *masking*, the specific component of vector is used, are supported.

A *draw call* invokes the graphics API to draw a group of primitives, so causing the graphics pipeline to execute. Inputs of shaders involve *uniform* inputs that remain same in the draw call and *varying* inputs. The output of GPU is strictly constrained. Uniform inputs are accessed via *constant register* or *constant buffer*, much more than *varying input register* in which varying inputs lie. There are also *temporary registers* for scratch space.

Common operations are efficient in GPU and it has *intrinsic functions* for complex operations.

The term *flow control* refers to the use of **branching instructions**(loop is also included) to change the flow of code execution. *Static flow control* depends on uniform inputs while *dynamic flow control* depends on varying inputs.
Shader Programme can be compiled offline and have different output files sent to GPU via drivers as strings according to different situations.

3.3 The Evolution of Programmable Shading

3.4 The Vertex Shader

It's worth noticing that some data manipulations happen before this stage called *input assembler*, weaving streams of data to different sets of vertices and primitives. It also performs instancing, which allows an object to be drawn a number of times with different data.

The Vertex Shader first process vertices of triangle shader. Each vertex is processed independently thus they can be applied to parallel.

3.5 The Geometry Shader

The input to the geometry shader is a single object and its associated vertices. The output can be **zero** and more primitives. The type of primitives in the input and output can be different. The geometry shader guarantee the output has the same order as the input. This stage is more about programmatically modifying incoming data or making a limited number of copies, rather than massively replicating or amplifying it.

3.5.1 Stream Output

The idea of *stream output* is to gathered the output from vertex shader and geometry shader in the stream. The rasterization stage can be optionally turned off and data processed in this way can be sent back allowing interaction process.

3.6 The Pixel Shader

The rasterizer does not directly affect pixel's color, but generate data to describe how a triangle covers a pixel. Additional inputs are needed for the pixel shader. The pixel shader can only influence the fragment it handles for merging. One exception is dealing with gradient and derivative information, which is the special capability of pixel shader. GPU implements this by processing a group of 2×2 or more. However, as a group of pixels should follow do same operations, no flow control is available here.

Multiple Rendering Targets(MRT) is derived as the huge capability of pixel shader, saving resulted color data to different same-dimension even same-bit-depth buffer. For different intermediate images being computed from same data, MRT allows all the rendering being done in one pass. MRT are also related to the abilities to read from these resulting images as textures.

3.7 The Merging Stage

This stage is where stencil-buffer and Z-buffer operations occur. Other operations such as color blending are involved. Operations are highly configurable. Color blending can be set up to perform a large number of different operations.

3.8 Effect

As shader program can't be isolated, or some particular effects need rounds of processing, *effect file* is aimed to encapsulate all the relevant information with some arguments to produce certain effects written in *effect language*.

4 Transform

Transform is an operation that takes entities such as points, vectors or colors and converts them in some way. A *linear transform* is one that preserve vector addition and scalar multiplication, specifically:

$$\mathbf{f}(\mathbf{x}) + \mathbf{f}(\mathbf{y}) = \mathbf{f}(\mathbf{x} + \mathbf{y})$$

$$k\mathbf{f}(\mathbf{x}) = \mathbf{f}(k\mathbf{x})$$

Scaling and Rotation are both linear transform while translation is not. Combining linear transform and translation, we introduce *affine transform* with 4×4 matrices and *homogeneous notation.*

Vectors are presented as $\mathbf{v} = (v_x, v_y, v_z, 0)^T$ and points are $\mathbf{v} = (v_x, v_y, v_z, 1)^T$.

4.1 Basic Transform

4.1.1 Translation

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x\ t_y\ t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{T}^{-1}(\mathbf{t}) = \mathbf{T}(-\mathbf{t})$$

4.1.2 Rotation

Rotation, along with Translation, is *rigid-body transform*, which preserves the distances between points transformed, and preserves handedness.

Assume $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ are orthonormal, which means:

$$\mathbf{u} \cdot \mathbf{u} = \mathbf{v} \cdot \mathbf{v} = \mathbf{w} \cdot \mathbf{w} = 1$$

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{u} = 0$$

Place three vector Horizontally to form a matrix $\mathbf{R}_{uvw}$

$$\mathbf{R}_{uvw} = \begin{pmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{pmatrix}$$

we can find $\mathbf{R}_{uvw}\mathbf{u} = \mathbf{x}$, $\mathbf{R}_{uvw}\mathbf{v} = \mathbf{y}$, $\mathbf{R}_{uvw}\mathbf{w} = \mathbf{z}$. Take $\mathbf{u}$ as example:

$$\mathbf{R}_{uvw}\mathbf{u} = \begin{pmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{pmatrix} \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \mathbf{x}$$

We then consider the inverse of the $\mathbf{R}_{uvw}^T$

$$\mathbf{R}_{uvw}^T = \begin{pmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{pmatrix}$$

We can find $\mathbf{R}_{uvw}^T\mathbf{x} = \mathbf{u}$, $\mathbf{R}_{uvw}^T\mathbf{y} = \mathbf{v}$, $\mathbf{R}_{uvw}^T\mathbf{z} = \mathbf{w}$. Take $\mathbf{x}$ as example:

$$\mathbf{R}_{uvw}^T\mathbf{x} = \begin{pmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_y & z_v & z_w \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \mathbf{u}$$

As $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ are orthonormal, they can form a new group of base vectors. Matrix $\mathbf{R}_{uvw}$ and its inverse can be perceived as the transform of the *rotation* between different coordinate systems. Especially, $\mathbf{R}_{uvw}$ means converting from *current* base to *new* base while its inverse means converting from *new* base to *current* base.

Think like above, we can easily get the formula for rotation around x-axis, y-axis, z-axis:

$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_i^{-1}(\phi) = \mathbf{R}_i(-\phi) = \mathbf{R}_i^T(\phi)$$

Attention: every rotation matrix is orthogonal and has a determinant of 1. And the *trace* is constant independent of axis:

$$tr(\mathbf{R}) = 1 + 2\cos\phi$$

4.1.3 Scaling

$$\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x\ s_y\ s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S}^{-1}(\mathbf{s}) = \mathbf{S}(1/s_x\ 1/s_y\ 1/s_z)$$

The scaling operation is called *uniform* if $s_x = s_y = s_z$ and *nonuniform* otherwise.

For uniform scaling operation, there are two forms of matrix:

$$\mathbf{S} = \begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad \mathbf{S}^{'} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/s \end{pmatrix}$$

A negative on one or three of the components of **s** gives a *reflection matrix*, or *mirror matrix*. If only two scale factors are negative, then we will rotate π radians.If the matrix has a negative determinants, then it's reflective.

If you want scale the axis of the orthonormal, right-oriented vectors $\mathbf{f}_x$, $\mathbf{f}_y$ and $\mathbf{f}_z$. You can first rotate the current axis to the new one, do the scaling and then rotate it back. That is:

$$\mathbf{F} = \begin{pmatrix} \mathbf{f}_x & 0 \\ \mathbf{f}_y & 0 \\ \mathbf{f}_z & 0 \\ 0 & 1 \end{pmatrix}$$

$$\mathbf{S}^{'} = \mathbf{F}^T \mathbf{S}(\mathbf{s}) \mathbf{F}$$

4.1.4 Shearing

$\mathbf{H}_{ij}$ means i coordinate is changed by the shearing matrix and j coordinate does the shearing. For example:

$$\mathbf{H}_{xz}(s) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{H}_{ij}^{-1}(s) = \mathbf{H}_{ij}(-s)$$

Another form:

$$\mathbf{H}_{xy}^{'}(s,\ t) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{H}_{xy}^{'}(s,\ t) = \mathbf{H}_{xz}(s)\mathbf{H}_{yz}(t)$$

Since every shearing matrix has a determinant of 1, this is a volume preserving transformation.

4.1.5 Concatenation of Transforms

It's associative.

$$\mathbf{TRSp} = (\mathbf{T}(\mathbf{R}(\mathbf{Sp}))) = (\mathbf{TR})(\mathbf{Sp})$$

4.1.6 The Rigid-Body Transformation

Rigid-body transformation refers to that consists of only rotations and translations.

$$\mathbf{X} = \mathbf{TR} = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix}$$

$$\mathbf{X}^{-1} = \begin{pmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix}$$

4.1.7 Normal Transformation

The original Transform can't directly applied to normals, but it can be applied to tangent vector.The correct matrix for normals are actually the *transpose of the original matrix's adjoint.*Attention, the normal needs normalization after the transform.

Prove when original matrix's inverse exists:

$$\begin{aligned} \mathbf{n}^T\mathbf{t} &= 0 \\ \mathbf{n}^T(\mathbf{M}^{-1}\mathbf{M})\mathbf{t} &= 0 \\ (\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}\mathbf{t}) &= 0 \\ (\mathbf{n}^T\mathbf{M}^{-1})\mathbf{t}_M &= 0 \end{aligned}$$

As normal is always perpendicular to tangent:

$$\mathbf{n}_M^T = \mathbf{n}^T\mathbf{M}^{-1}$$

So the real transform matrix is:

$$\mathbf{M}^{'} = (\mathbf{M}^{-1})^T = (\frac{\mathbf{M}^*}{|\mathbf{M}|})^T \rightarrow \mathbf{M}^{'} = (\mathbf{M}^*)^T$$

There some optimization. As the normal is a vector, translation will not affect it. In affine transformation, they will not change the w component. So we only need to calculate the adjoint of the upper-left 3×3 component. Often even this adjoint is unnecessary: Consider the concatenation of translations, *uniform* scaling and rotations. Rotations will remain same as the inverse of rotation matrix is it's transpose, which will cancel out, and other two have no effect on normals.

Normalization is not always needed as long as the $|\mathbf{M}| = 1$. It means that the concatenation involves a scaling that $|\mathbf{S}| \neq 1$

4.1.8 Computations of Inverses

Something about eigenvalues and singular value decomposition.
Eigenvalue:

$$\mathbf{Aa} = \lambda \mathbf{a}$$

Factor λ is called eigenvalue, and **a** is called eigenvector. To calculate eigenvalue:

$$(\mathbf{A} - \lambda \mathbf{I})\mathbf{a} = 0$$

$$|\mathbf{A} - \lambda \mathbf{I}| = 0$$

When **A** is *symmetric matrices* ($\mathbf{A} = \mathbf{A}^T$), the decomposition is quite simple for matrix **A**:

$$\mathbf{A} = \mathbf{QDQ}^T$$

Where **Q** is an orthogonal matrix and **D** is a diagonal matrix. The columns of **Q** are the eigenvector of **A** and the diagonal elements of **D** are the eigenvalues of **A**.

There is another generalization of the symmetric eigenvalue decomposition for non-symmetric matrices:*singular value decomposition*(SVD).For matrix **A**:

$$\mathbf{A} = \mathbf{USV}^T$$

U,**V** are potentially different, orthogonal matrices whose column are known as *singular vectors*. **S** is a diagonal matrix whose entries are *singular value*. To calculate the singular value and **U**, **V**, take **U** as example:

$$\mathbf{M} = \mathbf{AA}^T = (\mathbf{USV}^T)(\mathbf{USV}^T)^T = \mathbf{USV}^T\mathbf{VS}^T\mathbf{U}^T = \mathbf{US}^2\mathbf{U}^T$$

This is the form of eigenvalue decomposition for matrix **M**(which is a symmetric matrix) and we can get **U**, **S**. Set $\mathbf{M} = \mathbf{A}^T\mathbf{A}$can work **V** out.

This two kinds of decomposition stands for two kinds of transform decomposition, which can lead us to the inverse of the transformation(only consider the left upper 3 ×3 part).

$$\mathbf{A} = \mathbf{RSR}^T$$

Where $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$ are the eigenvectors and λ_1, λ_2, λ_3 are the eigenvalues.

1. Rotate the coordinate to make $\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ become the coordinate axis.
2. Scale in x and y by (λ_1 λ_2 λ_3)
3. Rotate the coordinate to the original one.

It's similar for singular decomposition which form is:

$$\mathbf{A} = \mathbf{R}_2\mathbf{SR}_1^T$$

There are also some basic ways to compute inverse, and another thing is when dealing with vector, we only needs to calculate the left upper 3 × 3 matrix's inverse.

4.2 Special Matrix Transform Operations

4.2.1 The Euler Transform

The idea to *Euler transform* is pretty easy, as to decompose a rotation to the rotation around three different axis:

$$\mathbf{E}(h,\ p,\ r) = \mathbf{R}_i(r)\mathbf{R}_j(p)\mathbf{R}_k(h)$$

However, when $p = \frac{\pi}{2}$, the *gimbal lock* will happen as one degree of freedom is lost. Directly, it will lead to one of axis in the model space to be rounded twice. Mathematically, when p is set to $\frac{\pi}{2}$, take one order as example:

$$\mathbf{E}(h,\ \frac{\pi}{2},\ r) = \begin{pmatrix} \cos(r+h) & 0 & \sin(r+h) \\ \sin(r+h) & 0 & -\cos(r+h) \\ 0 & 1 & 0 \end{pmatrix}$$

Since the matrix is only depend on $(r+h)$, we can see on degree of freedom is lost. Meanwhile, the *interpolation* of Euler transform is not interpolating the angle, which is its main weakness.

4.2.2 Extracting Parameters from the Euler transform

$$\mathbf{F} = \begin{pmatrix} f_{00} & f_{01} & f_{02} \\ f_{10} & f_{11} & f_{12} \\ f_{20} & f_{21} & f_{22} \end{pmatrix} = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_z(h) = \mathbf{E}(h,\ p,\ r)$$

$$\mathbf{F} = \begin{pmatrix} \cos r\cos h - \sin r\sin p\sin h & -\sin r\cos p & \cos r\sin h + \sin r\sin p\cos h \\ \sin r\cos h + \cos r\sin p\sin h & \cos r\cos p & \sin r\sin h - \cos r\sin p\cos h \\ -\cos p\sin h & \sin p & \cos p\cos h \end{pmatrix}$$

$$\frac{f_{01}}{f_{11}} = -\tan r \qquad \frac{\mathrm{f}_{20}}{\mathrm{f}_{22}} = -\tan h$$

$$h = \mathbf{atan2}(-f_{20}, -f_{22})$$

$$p = arcsin f_{21}$$

$$r = \mathbf{atan2}(-f_{10}, -f_{11})$$

In a special case, when $f_{10} = 0$ implying $f_{21} = \pm 1$:

$$\mathbf{F} = \begin{pmatrix} \cos(r+h) & 0 & \sin(r+h) \\ \sin(r+h) & 0 & -\cos(r+h) \\ 0 & \pm 1 & 0 \end{pmatrix}$$

$$h = 0 \qquad r = \frac{f_{10}}{f_{00}}$$

4.2.3 Matrix Decomposition

4.2.4 Rotation about an Arbitrary Axis

Use the given vector r to form orthonormal axis of base **r**, **s**, **t**. r needs normalization first. Use the base to form matrix **M** and do the transform.

$$\bar{\mathbf{s}} = \begin{cases} (0, -r_z, r_y), \ if & |r_x| < |r_y| \ and \ |r_x| < |r_z| \\ (-r_z, 0, r_x), \ if & |r_y| < |r_x| \ and \ |r_y| < |r_z| \\ (-r_y, 0, r_x), \ if & |r_z| < |r_x| \ and \ |r_z| < |r_y| \end{cases}$$

$$\mathbf{s} = \frac{\bar{\mathbf{s}}}{\parallel \bar{\mathbf{s}} \parallel}$$

$$\mathbf{t} = \mathbf{r} \times \mathbf{s}$$

$$\mathbf{M} = \begin{pmatrix} \mathbf{r}^T \\ \mathbf{s}^T \\ \mathbf{t}^T \end{pmatrix}$$

$$\mathbf{X} = \mathbf{M}^T \mathbf{R}_x(\phi) \mathbf{M}$$

Another method for a normalized vector **r** by ϕ radians:

$$\mathbf{X} = \begin{pmatrix} \cos\phi + (1-\cos\phi)r_x^2 & (1-\cos\phi)r_x r_y - r_z \sin\phi & (1-\cos\phi r_x r_z + r_y \sin\phi \\ (1-\cos\phi)r_x r_y + r_z \sin\phi & \cos\phi + (1-\cos\phi)r_y^2 & (1-\cos\phi r_y r_z - r_x \sin\phi \\ (1-\cos\phi)r_x r_z - r_y \sin\phi & (1-\cos\phi)r_y r_z + r_x \sin\phi & (\cos\phi + (1-\cos\phi)r_z^2 \end{pmatrix}$$

4.3 Quaternions

4.3.1 Mathematical background

Quaternions are much more straight forward than matrix and Euler transform in rotation and orientation.
Definition:

$$\hat{\mathbf{q}} = (\mathbf{q}_v,\ \mathbf{q}_w) = iq_x + jq_y + kq_z + q_w = \mathbf{q}_v + q_w,$$
$$\mathbf{q}_v = iq_x + jq_y + kq_z + q_w = (q_x,\ q_y,\ q_z)$$
$$i^2 = j^2 = k^2 = -1, jk = -kj = i, ki = -ik = j, ij = -ji = k$$

All operations of $\mathbf{q}_v$ are the same as the vectors'.
Multiplication:

$$\begin{aligned}\hat{\mathbf{q}}\hat{\mathbf{r}} &= (iq_x + jq_y + kq_z + q_w)(ir_x + jr_y + kr_z + r_w)\\ &= (\mathbf{q}_v \times \mathbf{r}_v + r_w\mathbf{q}_v + q_w\mathbf{r}_v,\ q_wr_w - \mathbf{q}_v \cdot \mathbf{r}_v)\end{aligned}$$

Addition:
$$\hat{\mathbf{q}} + \hat{\mathbf{r}} = (\mathbf{q}_v + \mathbf{r}_v,\ q_w + r_w)$$

Conjugate:
$$(\hat{\mathbf{q}})^* = (-\mathbf{q}_v,\ q_w)$$

Norm:
$$n(\hat{\mathbf{q}}) = \sqrt{\hat{\mathbf{q}}\hat{\mathbf{q}}^*} = \sqrt{\mathbf{q}_v \cdot \mathbf{q}_v + q_w^2} = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$$

Identity:
$$\hat{\mathbf{i}} = (\mathbf{0}, 1)$$

Inverse:
$$\hat{\mathbf{q}}^{-1} = \frac{\hat{\mathbf{q}}^*}{n(\hat{\mathbf{q}})^2}$$

Commutative:
$$\hat{\mathbf{q}}s = s\hat{\mathbf{q}} = (\mathbf{0}, s)(\mathbf{q}_v, q_w) = (s\mathbf{q}_v, sq_w)$$

Conjugate rules:
$$(\hat{\mathbf{q}}^*)^* = \hat{\mathbf{q}}$$
$$(\hat{\mathbf{q}} + \hat{\mathbf{r}})^* = \hat{\mathbf{q}}^* + \hat{\mathbf{r}}^*$$
$$(\hat{\mathbf{q}}\hat{\mathbf{r}})^* = \hat{\mathbf{r}}^*\hat{\mathbf{q}}^*$$

Norm rules:
$$n(\hat{\mathbf{q}}^*) = n(\hat{\mathbf{q}})$$
$$n(\hat{\mathbf{q}}\hat{\mathbf{r}}) = n(\hat{\mathbf{q}})n(\hat{\mathbf{r}})$$

Linearity:
$$\hat{\mathbf{p}}(s\hat{\mathbf{q}} + t\hat{\mathbf{r}}) = s\hat{\mathbf{p}}\hat{\mathbf{q}} + t\hat{\mathbf{p}}\hat{\mathbf{r}}$$
$$(s\hat{\mathbf{p}} + t\hat{\mathbf{q}})\hat{\mathbf{r}} = s\hat{\mathbf{p}}\hat{\mathbf{r}} + t\hat{\mathbf{q}}\hat{\mathbf{r}}$$

Associativity:

$$\hat{\mathbf{p}}(\hat{\mathbf{q}}\hat{\mathbf{r}}) = (\hat{\mathbf{p}}\hat{\mathbf{q}})\hat{\mathbf{r}}$$

Unit quaternion which is $n(\hat{\mathbf{q}}) = 1$ with $\mathbf{u}_q \cdot \mathbf{u}_q = 1$:

$$\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q, \cos\phi) = \sin\phi\mathbf{u}_q + \cos\phi = e^{\phi\mathbf{u}_q}$$

$$\log(\hat{\mathbf{q}}) = \log(e^{\phi\mathbf{u}_q}) = \phi\mathbf{u}_q$$

$$(\hat{\mathbf{q}})^t = e^{t\phi\mathbf{u}_q} = \sin(\phi t)\mathbf{u}_q + \cos(\phi t)$$

4.3.2 Quaternion Transform

The *unit quaternions* can represent any three-dimensional rotation. Put the four coordinates of a point or a vector $(p_x,\ p_y,\ p_z,\ p_w)^T$ into a quaternion $\hat{\mathbf{p}}$. With the unit quaternion $\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q, \cos\phi)$:

$$\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1}$$

the vector or point will be rotated around axis $\mathbf{u}_q$ by $\mathbf{2}\phi$. Not that here $\hat{\mathbf{q}}^{-1} = \hat{\mathbf{q}}^*$. Any nonzero real multiple of $\hat{\mathbf{q}}$ also represents the same transform.Extracting a quaternion from a matrix can return either $\hat{\mathbf{q}}$ or $-\hat{\mathbf{q}}$.

The concatenation is similar to matrix:

$$\hat{\mathbf{r}}(\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^*)\hat{\mathbf{r}}^* = (\hat{\mathbf{r}}\hat{\mathbf{q}})\hat{\mathbf{p}}(\hat{\mathbf{r}}\hat{\mathbf{q}})^*$$

The conversion from quaternion to matrix as $s = \frac{2}{n(\hat{\mathbf{q}})}$:

$$\mathbf{M}^q = \begin{pmatrix} 1 - s(q_y^2 + q_z^2) & s(q_xq_y - q_wq_z) & s(q_xq_z + q_wq_y) & 0 \\ s(q_xq_y + q_wq_z) & 1 - s(q_x^2 + q_z^2) & s(q_yq_z - q_xq_w) & 0 \\ s(q_xq_z - q_yq_w) & s(q_yq_z + q_wq_x) & 1 - s(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For unit quaternion especially:

$$\mathbf{M}^q = \begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_xq_y - q_wq_z) & 2(q_xq_z + q_wq_y) & 0 \\ 2(q_xq_y + q_wq_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_yq_z - q_xq_w) & 0 \\ 2(q_xq_z - q_yq_w) & 2(q_yq_z + q_wq_x) & 1 - 2(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To extract quaternion from matrix:

$$\mathbf{M}^q_{21} - \mathbf{M}^q_{12} = 4q_wq_x$$

$$\mathbf{M}^q_{02} - \mathbf{M}^q_{20} = 4q_wq_y$$

$$\mathbf{M}^q_{10} - \mathbf{M}^q_{01} = 4q_wq_z$$

$$tr(\mathbf{M}^q) = 4q_w^2$$

It's better to first find out the biggest entry for numerical precision.

Spherical linear interpolation is used to interpolated between two quaternions as quaternion can be seen as a point on a four-dimensional sphere. The original form for any interpolation is:

$$\mathbf{s}(\mathbf{q}, \mathbf{r}, t) = \frac{\sin{(1-t)\phi}}{\sin\phi}\mathbf{q} + \frac{\sin t\phi}{\sin\phi}\mathbf{r}$$

$\mathbf{q}$, $\mathbf{r}$ are points on the sphere and ϕ is the angle between two radiuses towards points. For quaternion it can also be considered as:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}},\ t) = (\hat{\mathbf{r}}\hat{\mathbf{q}}^{-1})^t\hat{\mathbf{q}}$$

in quaternions, ϕ can be computed through $\cos\phi = q_xr_x + q_yr_y + q_zr_z + q_wr_w$ The interpolate path is arc on the sphere.

The transform from one vector $\mathbf{s}$ to $\mathbf{t}$: First $\mathbf{s}$ and $\mathbf{t}$ should be normalized.

$$\mathbf{u} = \mathbf{s} \times \mathbf{t}$$

$$e = \mathbf{s} \cdot \mathbf{t} = \cos 2\phi$$

$$\parallel \mathbf{u} \parallel = \parallel \mathbf{s} \times \mathbf{t} \parallel = \sin 2\phi$$

$$\hat{\mathbf{q}} = (\frac{\sin\phi}{\sin 2\phi}\mathbf{u}, \cos\phi) = (\frac{1}{\sqrt{2(1+e)}}(\mathbf{s} \times \mathbf{t}), \frac{\sqrt{2(1+e)}}{2})$$

For matrix form(this form can avoid numerical issue):

$$\mathbf{R}(\mathbf{s},\ \mathbf{t}) = \begin{pmatrix} e + hu_x^2 & hu_xu_y - u_z & hu_xu_z + u_y & 0 \\ hu_xu_y + u_z & e + hu_y^2 & hu_yu_z - u_x & 0 \\ hu_xu_z - u_y & hu_yu_z + u_x & e + hu_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$h = \frac{1 - \cos 2\phi}{\sin^2 2\phi} = \frac{1 - e}{\mathbf{u} \cdot \mathbf{u}} = \frac{1}{1 + e}$$

For $2\phi \approx 0$, transform matrix is identity; for $2\phi \approx \pi$, we can choose one of the vector as the axis to rotate by.

4.4 Vertex Blending

Consider the transformation of the arms made up of forearm and upper arm. When they are both *rigid-body*, the joint will be presented as the overlap, which is not flexible. *stitching* is the technique that the join is an elastic skin on which vertex may have different transforms of the forearm or upper arm. One implementation is that keep the transforms the same in a triangle. *Vertex blending*, which is also called *skinning*, *enveloping*, and *skeleton-subspace deformation*, the object has a skeleton of *bones* and *skin* referring to whole mesh. Every vertex will be affected by more than one bones thus having many different transforms. To blend all these transforms, in mathematical background:

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t)\mathbf{M}_i^{-1}\mathbf{p}, \quad \text{where} \quad \sum_{\mathrm{i=0}}^{\mathrm{n-1}} \mathrm{w_i} = 1, \quad \mathrm{w_i} > 0$$

$\mathbf{u}$ is the final output that change with time t; $\mathbf{M}_i$ is the transform from the bone's coordinate space to the world space; $\mathbf{B}_i$ is the transform in the world space; $\mathbf{p}$ stands for the vertex.

In implementations, $\sum i = 0^{n-1} w_i$ can be out of range [0, 1] for specific blending algorithm like *morph targets*. *Dual quaternions* are employed to resolve unwanted folding, twisting and self-intersection.

4.5 Morphing

Morphing uses linear interpolation to represent frames between key frames. After we find the one-to-one *vertex correspondence* which may be very difficult, the interpolation is obvious enough:

$$\mathbf{m} = (1 - s)\mathbf{p}_0 + s\mathbf{p}_1, \quad s = \frac{t - t_0}{t_1 - t_0}$$

Morph targets, or *blend shapes* is used to interpolate between shapes. Consider we have a initial shape $\mathbf{N}$ and other shape models $\mathbf{P}_i$. $\mathbf{D}_i = \mathbf{P}_i - \mathbf{N}$ stands for data describing the differentials between models and initial. An morphed model $\mathbf{M}$ can be obtained by:

$$\mathbf{M} = \mathbf{N} + \sum_{i=0}^{n-1} w_i \mathbf{D}_i$$

w_i can be negative symbolizing a inverse change of the model.

4.6 Projection

Setting: we are looking at negative z-axis as DirectX.

4.6.1 Orthographic Projection

The simplest one with intervals on z-axis from n(near plane) to f(far plane) $(n > f)$

$$\mathbf{P}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Often the orthographic projection is expressed in terms of a AABB(*Axis-Aligned Bounding Box*) with it's minimum corner(l, b, n) and maximum (r, t, f). The transform involves first translate it to the origin and then scale it to make it in the canonical view volume. For OpenGL, the minimum corner of canonical view volume is$(-1, -1, -1)$ and maximum corner is $(1, 1, 1)$ while the DirectX's bounds are $(-1, -1, 0)$ and $(-1, -1, 1)$.

Theoretically:

$$\begin{aligned} \mathbf{P}_o = \mathbf{S}(\mathbf{s})\mathbf{T}(\mathbf{t}) &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

For DirectX, we should apply an additional transform for different z intervals:

$$\begin{aligned} \mathbf{P}_{o[0,1]} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

As a left-handed projection is used after projection, a reflect transform is needed:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4.6.2 Prospective Projection

Prospective projection project point $\mathbf{p}$ to point $\mathbf{q}$ on plane $-d, d > 0$:

$$\frac{q_x}{p_x} = \frac{-d}{p_z} \iff q_x = \frac{-dp_x}{p_z}$$

in matrix form:

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{pmatrix}$$

$$\mathbf{q} = \mathbf{P}_p\mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ -\frac{p_z}{d} \end{pmatrix} \iff \begin{pmatrix} -\frac{dp_x}{p_z} \\ -\frac{dp_y}{p_z} \\ -d \\ 1 \end{pmatrix}$$

Like orthogonal projection, rather than projecting onto a plane, we transform the *view frustum* to canonical view volume with (l, r, b, t, n, f)(minimum corner: (l, b, n) and maximum corner (r, t, n) on near plane). We can concatenate the transform from view frustum to rectangular and orthogonal transform. The following transform $\mathbf{P}_t$ guarantees that all points on the near plane remain the same.

$$\begin{aligned} \mathbf{P}_p &= \mathbf{P}_o\mathbf{P}_t \\ &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -2fn \\ 0 & 0 & 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{aligned}$$

For OpenGL, as $n^{'} = -n$, $f^{'} = -f$:

$$\mathbf{P}_{OpenGL} = \mathbf{P}_o\mathbf{S}(1,\ 1,\ -1) = \begin{pmatrix} \frac{2n^{'}}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n^{'}}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f^{'}+n^{'}}{f^{'}-n^{'}} & -\frac{2f^{'}n^{'}}{f^{'}-n^{'}} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

For DirectX, as it implements left-handed coordinates and z-interval in [0, 1]:

$$\mathbf{P}_{p[0,1]} = \mathbf{P}_{o[0,1]}\mathbf{S}(1,\ 1,\ -1) = \begin{pmatrix} \frac{2n^{'}}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n^{'}}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f^{'}}{f^{'}-n^{'}} & -\frac{f^{'}n^{'}}{f^{'}-n^{'}} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

5 Visual Appearance

5.1 Visual Phenomena

Light is emitted by light sources, scattered by object and finally absorbed by a sensor.

5.2 Light Source

Light discussed here are *directional lights*, presented by *light vector* **l**. **l** has a length of 1 and always point to the *opposite* of the direction that light travels. The science measuring of light is *radiance*. The emission quantity of a direction light source is *irradiance*, which stands for the power through a unit area perpendicular to **l**, describing the brightness of an area(like surface). We represent irradiance as an RBG vector as light has color. Environmental light is called *ambient light*.

The surface irradiance is equal to irradiance measured perpendicular to **l** times the cosine of the angle θ_i between **l** and normal **n**.Irradiance is proportional to the *density* of light and inversely proportional to the *distance* between rays. We use E to stand for irradiance:

$$E = E_L \bar{\cos}\,\theta_i = E_L \max(\mathbf{l} \cdot \mathbf{n},\ 0)$$

In reality the direction of light is arbitrary, requiring a *light meter* to measure. For multiple light sources:

$$E = \sum_{k=1}^{n} E_{L_k} \bar{\cos}\,\theta_{i_k}$$

5.3 Material

Fundamentally, all light-matter interaction are the result of two phenomena:*scattering* and *absorption*. Scattering happens when light encounters any kind of optical discontinuity, usually interface between surface and air, involving *reflection* and *refraction* (*transmission*), which change the direction without changing the amount. Absorption happens inside matter and causes some of the light to be converted into another kind of energy and disappear, which reduces the amount but don't affect its direction.

In opaque objects, surface shading equation is divided to two parts: *specular term* representing the light that was reflected at the surface, and *diffuse term* representing light which has undergone transmission, absorption and scattering. Scattering is different with reflection here as reflection obeys the law of reflection. To characterize by a shading equation, we need to represent the amount and direction of *out going light* based on the amount and direction of the *incoming light*.

Incoming illumination is measured as surface irradiance. The outgoing light is measured as *exitance* with symbol $\mathbf{M}$. Light-matter interaction is linear. The ratio of exitance and irradiance is between 0 to 1 in opaque objects and differentiates in colors. There are represented as RGB vector called the *surface color* $\mathbf{c}$.It can be divided into two terms *specular color* $\mathbf{c}_{\text{spec}}$ and *diffuse color* $\mathbf{c}_{\text{diff}}$. They are dependent on the its composition.

In this chapter we assume diffuse term has no directionality. The directional distribution of the specular term depends on the surface smoothness, which is also involved in shading. Smoothness can either be a parameter in shading equation, or be made into model or texture, depending on the situation.

5.4 Sensor

Sensors to form images should include a light-proof enclosure with a single small *aperture*(opening) that restricts the directions of light. The combination of enclosure, lens and aperture causes the sensor to be *directionally specific*. That is, the sensor measure average *radiance*($\mathbf{L}$), the density of light flow per unit area per incoming direction, rather than average irradiance, the density of light flow per unit area from all incoming direction. Radiance can used to describe the brightness and color of a ray of light.

In the model, each sensor only measure a single radiance sample which is along a ray goes through the sensor and the center of perspective projection. The detection of the sensor is replaced by the shader equation evaluation to evaluate radiance. This ray in the equation is represented as *view vector* $\mathbf{v}$ whose length is set to be 1. After the evaluation, the transform between radiance and signal is required. The physical sensors measure the *average* value of the radiance over their area, over incoming directions focused by the lens, and over a time interval.

5.5 Shading

Shading is the process of using an equation to compute the outgoing radiance $\mathbf{L}_o$ along the view ray, $\mathbf{v}$, based on the material properties and light sources. In this chapter we will introduce a simple shading model.

$$M_{\text{diff}} = \mathbf{c}_{\text{diff}} \otimes E_L \bar{\cos}\,\theta_i$$

$$L_{\text{diff}} = \frac{M_{\text{diff}}}{\pi} = \frac{\mathbf{c}_{\text{diff}}}{\pi} \otimes E_L \bar{\cos}\,\theta_i$$

This type of shading term is also called *Lambertian*.In real time shading we will fold $\frac{1}{\pi}$ into E_L.

Similarly, for specular term:

$$M_{\text{spec}} = \mathbf{c}_{\text{spec}} \otimes E_L \bar{\cos}\,\theta_i$$

As the specular term is directional, we introduce *half vector* $\mathbf{h}$(some of the explanation is in chapter 7):

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\| \mathbf{l} + \mathbf{v} \|}$$

$$\begin{aligned} L_{\text{spec}}(\mathbf{v}) &= \frac{m+8}{8\pi} \bar{\cos}^m \theta_h M_{\text{spec}} \\ &= \frac{m+8}{8\pi} \bar{\cos}^m \theta_h \mathbf{c}_{\text{spec}} \otimes E_L \bar{\cos}\,\theta_i \end{aligned}$$

where θ_h is the angle between $\mathbf{h}$ and $\mathbf{n}$.

The total outgoing radiance $\mathbf{L}_o$:

$$\mathbf{L}_o(\mathbf{v}) = (\frac{\mathbf{c}_{\text{diff}}}{\pi} + \frac{m+8}{8\pi}\overline{\cos}^m\,\theta_h \mathbf{c}_{\text{spec}}) \otimes E_L \overline{\cos}\,\theta_i$$

which is quite similar to Bling-Phong equation.

$$\mathbf{L}_o(\mathbf{v}) = (\overline{\cos}\,\theta_i \mathbf{c}_{\text{diff}} + \overline{\cos}^m\,\theta_h \mathbf{c}_{\text{spec}}) \otimes B_L$$

5.5.1 Implementing the Shading Equation

For multiple light sources:

$$\mathbf{L}_o(\mathbf{v}) = \sum_{k=1}^{n}((\frac{\mathbf{c}_{\text{diff}}}{\pi} + \frac{m+8}{8\pi}\overline{\cos}^m\,\theta_{h_k} \mathbf{c}_{\text{spec}}) \otimes E_L \overline{\cos}\theta_{i_k})$$

When implemented into shader, the computations need to be divided according to their *frequency of evaluation*. The lowest frequency of evaluation is *per-model*: Expressions that are constant over the entire model can be evaluated once, and the result passed to the graphics API. *Per-primitive* can be performed by geometry shader if the result of computation is constant over each of the primitives comprising the model. In *per-vertex* evaluation, where the evaluation is performed in vertex shader and passed to pixel shader. The highest frequency of evaluation is *per-pixel*, which the evaluation is performed in pixel shader. We assume all material properties are same across the entire mesh.

Separating out the *p*er-model sub-expressions, $\mathbf{K}_d$ and $\mathbf{K}_s$ will be evaluated in the application:

$$\mathbf{L}_o(\mathbf{v}) = \sum_{k=1}^{n}((\mathbf{K}_d + \mathbf{K}_s \overline{\cos}^m\,\theta_{h_k}) \otimes E_L \overline{\cos}\theta_{i_k})$$

The view vector $\mathbf{v}$ can be computed from the surface position $\mathbf{p}$ to the view position $\mathbf{p}_v$:

$$\mathbf{v} = \frac{\mathbf{p}_v - \mathbf{p}}{\| \mathbf{p}_v - \mathbf{p} \|}$$

The normal $\mathbf{n}$ derives from the vertex normal as triangle mesh is used to represent underlying curved structure with each vertex has normals per triangle.

Now we can create **Shade()** function. Per-primitives evaluation (also called *flat shading* is not desirable as vertex normals involved. Per-vertex evaluation, known as *Gouraud Shading*, passes normals and positions to the function and get an interpolated result passed to pixel shader and the result is directly written to the output. It's reasonable for matte surface but have noticeable artifacts on specular surface. Per-pixel, known as *Phong shading*, uses vertex shader to interpolate normals and positions, then passed by the pixel to **Shader()**. It has no interpolation artifacts while costly. Solution to it is to adopt hybrid approach where some evaluations are per-vertex and some are per-pixel.

Implementing a shading equation is a matter of deciding what parts can be simplified, how frequently to compute various expressions, and how the user will be able to modify and control the appearance.

5.6 Aliasing and Anti-aliasing

5.6.1 Sampling and Filtering Theory

$$\text{Continuous image} \xrightarrow{\textit{sampling}} \text{sampled signal} \xrightarrow{\textit{reconstruction}} \text{reconstructed signal}$$

When sampling is done, aliasing can occur when samples are taken in a series of time steps, which is called *temporal aliasing.* It's because signal is sampled at a too low frequency.According to *sample theory*, in order to sample properly, the sampling frequency has to be more than twice the maximum frequency, and the sampling frequency is called the *nyquist rate* or *Nyquist limit.* This also implies that the signal has to be *bandlimited.* However, an three-dimensional scene is normally never bandlimited when rendered with point samples. But there is signal that is bandlimited like textures.

Reconstruction from sampling requires for *filtering.* Note that the area of filter should always be 1, or reconstructed signal can appear to grow or shrink. *Box filter* is used as the simplest filter. *Tent filter* is another one implementing interpolating between nearby samples.

In order to get better continuity, *low-pass filter* is introduced. The *frequency component* of a signal is a sine wave: $\sin(2\pi f)$, where f is the frequency of that component. A low-pass filter removes all frequency components with frequencies higher than a certain frequency defined by the filter, removing sharp features of the signal. A ideal low-pass filter is sinc filter:

$$\mathrm{sinc(x)} = \frac{\sin \pi \mathrm{x}}{\pi \mathrm{x}}$$

In fact, the sinc filter eliminates all sine waves with frequencies higher than $\frac{1}{2}$ sampling rate, perfect for sampling rate at 1.0. More generally, assume the sampling frequency is f_s, the perfect filter is $\mathrm{sinc(f_s x)}$, which eliminates all frequencies higher than $\frac{f_s}{2}$. However, as the filter width is infinite and is also negative at times, it's rarely useful in practice. In practice, we often use filters similar to sinc filter with limited pixels they influence. When negative filter value are undesirable or impractical, filters with no negative lobes.

After reconstruction, we need *resampling* to display signal. Resampling is used to magnify or minify a sampled signal. Originally all sampled points are on integer coordinates. For new sampled points with interval a uniformly, if $a > 1$ *minification* (*downsampling*) take place, and for $a < 1$, *magnification* (*upsampling*) occurs. For magnification, it's pretty straight forward to sample from a perfectly reconstructed, continuous signal. For minification, it's better to use $\mathrm{sinc}(\frac{\mathrm{x}}{\mathrm{a}})$ to get reconstructed signal for resampling in order to blur the continuous signal.

5.6.2 Screen-Based Algorithm

Some anti-aliasing schemes are focused on particular primitives. There are two special cases: texture aliasing and line aliasing. For line aliasing, one method is to treat the line as quadrilateral one pixel wide that is blended with it's background; another is to consider it an infinitely thin, transparent object with a halo; third is to render the line as an anti-aliased texture. Hardware solution is dedicated to rapid, high-quality rendering.

One problem of aliasing is low sampling rate, which can be better when introducing more samples in the cell. The general strategy of screen-based anti-aliasing schemes is to use a sample pattern for the screen and then weight and sum the samples to produce a pixel color, **p**:

$$\mathbf{P}(x,y) = \sum_{i=1}^{n} w_i \mathbf{c}(i,x,y), \quad \sum_{i=1}^{n} w_i = 1$$

Where the sample is taken on the screen grid is different for each sample, and optionally the sampling pattern can vary from pixel to pixel. As sampling point is point in real time rendering system, **c** function can be thought of first retrieve position of the point and then the color. For w_i in most design is set to be constant, i.e., $w_i = \frac{1}{n}$. The simplest anti-aliasing is a single sample at the center of pixel.

Anti-aliasing algorithms that compute more than one full sample per pixel are called *supersampling* (or *oversampling*) methods. *Full-scene anti-aliasing* (FSAA) renders the scene at a higher resolution and then average neighboring samples to create an image. It's costly but simple. Other, low quality FSAA is to sample at twice the rate on only one screen axis.

A related method is the *accumulation buffer*. This method instead uses a buffer that has the same resolution with more color bits. To obtain an 2×2 sampling, four images are generated with the moved half a pixel in the screen *x*- or *y*- axis. They can be used to create effects such as *motion blurring* and *depth of field* (where objects not at camera focus appear to be blurry).

An advantage that the accumulation buffer has over the FSAA(and A-buffer) is that sampling does not have to be uniform orthogonal pattern within a pixel's grid cell. *Rotate grid supersampling*(RGSS), this pattern gives more levels of anti-aliasing for nearly vertical and horizontal edges.

However, techniques like supersampling generating samples that are fully specified has relatively high cost. *Multisampling* strategies lessen the high computational costs of these algorithms by sampling various types of data at different frequencies. It may have some samples per fragment on the edge of the object, while may have one sample per fragment for shadow. Within GPU hardware, these techniques are called *multisample anti-aliasing* and more recently, *converge sampling anti-aliasing*, which saves time by computing fewer shading samples per fragment. If all MSAA positional samples are covered by the fragment, the shading sample is in the center. If the fragment covers fewer samples, the center may shift to avoid shade sampling off the edge of a texture. This is called *centroid sampling* or *centroid interpolation*.

MSAA is faster than a pure supersampling scheme because the fragment is shaded only once. It stores separate color and z-depth for each samples, which is not necessary. in CSAA, pixels are subdivided to subpixels which stores a index to the fragment it associated with. And a table with limited entries stores color and z-depth associated with fragment, which can be indexed by each subpixel . It may occur artifacts with too many kinds of fragments within a pixel, however it's not so common in practice.

This idea is similar to *A-buffer*, which is commonly used as software at non-interaction speed. For each polygon rendered, a *converge mask* is created for coverage. The shade for the polygon associated with this coverage mask is typically computed once at the centroid location and shared with all samples on the fragment.The z-depth is also computed and stored in some way, even slope is retained for precise z-depth value. All the information will form a A-buffer fragment.

A critical way A-buffer is different from Z-buffer is that a screen grid cell can hold any number of fragments at one time, rather than maintaining one particular z-depth value. As they collect, fragments can be discarded if they are hidden judged from z-depth and coverage mask. Coverage mask can also be merged by **or** to form a larger area of coverage. Such merging can happen when a fragment buffer becomes filled, or as a final step before shading and display. After all the polygons are sent to A-buffer, colors of pixels can be computed by multiplying the percentage of coverage with fragments color.

All these anti-aliasing technique result in better approximation of how each polygon covers a grid cell, however, they have limitations. One limitation is that a scene made of arbitrarily small objects. This can be solved by *stochastic sampling*, which distribute samples randomly in the pixel with different sampling pattern at each pixel. The most common kind of stochastic sampling is *jittering*, a form of *stratified sampling* which works by place sample points in a random location of a subpixel divided equally. *N-rooks sampling* is another form, for n samples being placed in a n×n grid.

Another technique called *interleaved sampling* with different sample patterns can be intermingled in a repeating pattern and be done with a pure jittering scheme. It's seen as the generalization of accumulation buffer.

One real-time anti-aliasing scheme that lets samples affect more than one pixel is Quincunx method, also called *high resolution anti-aliasing.* There four samples in the corner and the fifth in the center. In average every pixel has two samples, with the center has a weight of $\frac{1}{2}$ and the corner has a weight of $\frac{1}{8}$. This patter approximate a two-dimensional tent filter. Though this technique appears to die out, its method is reused, such as *custom filter anti-aliasing* and FLIPQUAD which mix the idea of Quincunx and RGSS.

Sampling rate can be varied, at low rate when scene is changing, and at high rate for static scene.

5.7 Transparency, Alpha, and Compositing

Transparency effect can be divided view based effect and light based effect. Z-buffer as the dominating algorithm, has a problem of not able to deal with a number of transparent objects overlapping on one pixel. One method for giving the illusion of transparency is called *screen-door transparency.* This idea is to render the transparent polygon with a checkerboard fill pattern. That is, every other pixel of the polygon is rendered, thereby leaving the object behind it partially visible. However, a transparent object looks best when 50% transparent; Only one transparent object can be convincingly rendered on one area of the screen. It's simple and the same idea is used in *alpha to converge* at a subpixel level.

The concept of *alpha blending* is used for blend transparent object's color with the color of the object behind it. Alpha value α is introduced describing the degree of opacity of an object fragment for a given pixel. To make an object transparent, it is rendered on the top of the existing scene with an alpha class less than 1.0. Every pixel will finally receive RGBA value to blend with its own color using **over** operator.

$$\mathbf{c}_o = \alpha_s \mathbf{c}_s + (1 - \alpha_s)\mathbf{c}_d$$

$\mathbf{c}_s$ and α_s refer to the incoming transparent object's color and alpha value(*source*).$\mathbf{c}_d$ is the pixel color(*destination*). Especially, when incoming object is opaque, which means $\alpha_s = 1$, it's the form of the original Z-buffer. It requires specific order, as the opaque objects been rendered first and the transparent object are blended on top of them in back-to-front order. The equation can be modeled for front-to-back-order, which is another blending mode called *under operator*. However, sorting is not available some time. At this time, it's often best to use Z-buffer testing with all transparent objects at least appearing rather than Z-buffer replacement. Other technique also helps.

A-buffer can achieve hardware sorting rather than software sorting, and multisample fragment's alpha represents purely the transparency as it stores a separate converge mask.

Transparency can also be computed using two or more depth buffers and multiple passes: First, a rendering pass is made so that the opaque surfaces' z-depth are in the first Z-buffer. In the next pass, find the transparent surfaces that are closer than the first stored Z-buffer and the farthest among them to find the backmost transparent layer, then put their depths into second Z-buffer , and so on. The pixel shader can be used to compare z-depths in this fashion and so perform *depth peeling*, where each visible layer is found in turn.

The **over** operator can also be used for anti-aliasing edges. Instead of storing a converge mask, an alpha can be generated to approximate the edge cover. This alpha value is then used to blend the object's edge with the scene, using **over** operator. It's unwise to generate alpha value for every polygon's edge. For example, when two adjacent polygons fully cover a pixel each by 50%, when using the alpha it will lead to 75% coverage over the pixel. This can be avoided by using converge mask or by blurring the edges outwards. In summary, alpha value can represent transparency and coverage.

The **over** operator turns out to be useful for blending together photographs or synthetic rendering of the objects. This process is called *compositing*, leading to RGBα values. The α channel is sometimes called *matte* and shows the silhouette shape of the object.

Additive blending is another way:

$$\mathbf{c}_o = \alpha \mathbf{c}_s + \mathbf{c}_d$$

This blending mode doesn't require sorting between triangles. It works well for glowing effects that do not attenuate the pixel behind them but only brighten them. It's not suitable for transparency but work well for semitransparent surfaces.

The most common way to store synthetic RGBα images is with *pre-multiplied alphas* (*associated alpha*). That is, the RGB values are multiplied by α before stored(RGB values are smaller than α), which make **over** operator more easily:

$$\mathbf{c}_o = \mathbf{c}_s^{'} + (1 - \alpha)\mathbf{c}_d$$

Another way images are stored is with *un-multiplied alphas* (*un-associated alpha*), means that the RGB value is not multiplied by α. It's rarely used in synthetic image but has the advantage of represent the actual color. It's also useful to mask a photograph without affecting the underlying image's origin data.

A concept related to the alpha channel is *chroma-keying.*It derives from the term *green-screen* or *blue-screen matting.* The idea here is that a particular color is designated to be transparent, where α needs to be stored. This allows images to be given an outline shape by using just RGB values with out α. The drawback is that the actual *alpha* of image is 0.0 or 1.0.

Though **over** operator can represent transparency, it's better to be the approximation of pixel coverage. In reality, the transparent object is represented of filtering and reflection. To filter, the scene behind the transparent object should be multiplied by objects spectral opacity. Reflection is to multiply the frame buffer by one RGB color and add another RGB color, which can be done in two process or *dual-color blending.*

5.8 Gama Correction

The content above discuss about the pixel values. For display, *cathode-ray tube* (CRT) monitors exhibit a power law relationship between input voltage and display radiance, which turns out to match the inverse of light sensitivity of human eyes, leading to that an encoding proportional to CRT input voltage is roughly *perceptually uniform.* This near-optimal distribution of values minimizes *banding* artifacts. The desire for compatibility is another important factor. LCD has different tone response curve with hardware providing compatibility.

The *transfer functions* that define the relationship between radiance and encoded pixel values are slightly modified power curves by the exponent γ:

$$\mathbf{f}_{\text{xfer}}(x) \approx x^{\gamma}$$

Two γ values are needed to fully characterize an imaging system. The *encoding gamma* describes the *encoding transfer function*, which is the relationship between scene radiance values captured by a imaging device and encoded pixel values. The *display gamma* characterizes the *display transfer function*, which is the relationship between encoded pixel values and displayed radiance. The product of the two gamma values is the overall or *end-to-end gamma* of the *end-to-end transform function.* It seems to be ideal to have end-to-end gamma being 1, there two differences: Absolute display radiance is much less than scene radiance; the *surrounded effect*, refers to the fact that the the original scene radiance values fill the entire filed of view of the observer, while the display radiance values are limited to a screen surrounded by ambient room illumination. To counteract that, a non-unit end-to-end *gamma* is used from 1.5 for dark environment to 1.125 for bright environment.

The relevant gamma for rendering purposes is encoding gamma. For television is 0.5, and for personal computer with a standard called *sRGB* is 0.45. The sRGB standard assume the display gamma is 2.5 as the CRT usually has and set to 0.45 to ensure an appropriate end-to-end gamma.

Most image data read to rendering system has been applied to encoded transfer function, which turns from linear-space to a non-linear space. *Gamma correction* guarantee the correctness of linear interpolating between radiance. Ignoring gamma correction also affects the quality of anti-aliased edges like *roping*.Fortunately, modern GPU can be set to automatically apply the encoding transfer function when values are written to the color buffer. But this feature can't be misused. It's important to apply conversion at the final stage(Where the values are written to the display buffer for the last time). This doesn't mean intermediate buffers can't contain nonlinear encodings, but it must be converted carefully before post-processing.

It's also necessary to convert any nonlinear input values to a linear space as well, such as texture. GPU is now available to convert it automatically. For authoring texture, care must be taken to use the correct color space. Various other inputs need similar conversion as well.

6 Texture

6.1 The Texturing Pipeline

Texturing, at its simplest, is a technique for efficiently modeling the surface's properties. The pixels in the image texture are called *texels.* The gloss texture modifies the gloss value, and *bumping texture* changes the direction of the normal, which will all influence lighting equation.

$$\begin{aligned} &\textit{object space location} \xrightarrow{projector\ function} \textit{parameter space coordinate} \\ &\xrightarrow{corresponder\ function} \textit{texture space coordinate} \\ &\xrightarrow{obtain\ value} \textit{texture value} \\ &\xrightarrow{valuetransformfunction} \textit{transformed texture value} \end{aligned}$$

This projector function is called *mapping*, which needs to *texture mapping.*

6.1.1 The Projector Function

Projector functions typically work by converting a three-dimensional point in space into texture coordinates, including spherical, cylindrical, planar projections. Problems may occur at the seams where faces meet. *Polycube map* maps a model to a set of cube with different volumes of space mapping to different cubes. Other form of projector functions are not projections but are implicit part of surface formation. The goal of the projector function is to generate texture coordinate.

Various projector functions can be applied to a single model. most projector function are applied in modeling stage and results are stored in vertices. Some functions require vertex or pixel shader like animation and *environmental mapping.*

Spherical projector function, according to spherical coordinates:

$$\phi(x,\ y,\ z) = (\frac{\pi + \mathbf{atan2}(y,\ x)}{2\pi}, \frac{\pi - \mathbf{acos}(\frac{z}{\|x\|})}{\pi})$$

Cylindrical:

$$\phi(x,\ y,\ z) = (\frac{\pi + \mathbf{atan2}(y,\ x)}{2\pi}, \frac{1}{2}(1+z))$$

Planar projection simply uses orthogonal projection to apply texture maps to characters, as the texture glued onto a paper doll.

$$\phi(x,\ y,\ z) = (\frac{\tilde{u}}{w}, \frac{\tilde{v}}{w}), \quad \text{where} \begin{pmatrix} \tilde{u} \\ \tilde{v} \\ * \\ w \end{pmatrix} = \mathbf{P}_t \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Artists often must manually decompose the model into near-planar pieces to avoid distortion by unwrapping the mesh. The goal is to have each polygon be given a fairer share of a texture's area, while also maintaining mesh connectivity (without too many seams).

The parameter space is not always a two-dimensional plane; sometimes is a three-dimension volume. For three-element vector (u, v, w), w represents the depth along the projection direction. For four-coordinate (s, t, r, q), q is used as fourth value in a homogeneous coordinate for spotlighting effect. Another important type of parameter space is directional, where each point in the parameter space represents a direction. The most common one is *cube texture*. The one-dimensional projector function also have its own use(coloration according to altitude). Line can also be textured.(Rain is textured with a semi-transparent image)

6.1.2 The Corresponder Function

Corresponder function convert parameter-space coordinate to texture-space location in order to provide flexibility. One type is to use the API to select a portion of existing texture for display. Another type is a matrix transformation, which can be applied in vertex or pixel shader. The order of the transformation must be reserved as it's the location of image will be changed rather the location of object.

Another class of corresponder functions controls the way an image is applied when parameters are out of range $[0, 1)$.

Warp(DirectX), **Repeat**(OpenGL), or**title**: The image repeats itself across the surface; algorithmically, the integer part will drop.

mirror: The image repeats itself across the surface, but is mirrored on every other repetition. This provide some continuity along the

Clamp(DirectX), **Clamp to edge**(OpenGL): Values out of range of$[0, 1)$ will clamp to the edge. This function is useful for avoiding accidentally taking samples from the opposite edge of a texture when bilinear interpolation happens near a texture's edge.

Border(DirectX), **Clamp to Border**: Values out of range of $[0, 1)$ will be set to a border color. This function is good for rendering decal onto surfaces.

The *periodicity* problem like repeating tiles of a texture can be solved by combining the texture values with another, non-tiled texture. Another option to avoid periodicity is to use shader programs to implement specialized corresponder functions that randomly recombine texture pattern or tiles. *Wang tiles* are one example which choose randomly from small square tiles with matching edges.

For real-time work, the last corresponder function is implicit, and is derived from the image's size. It allows the parameter being in range $[0, 1)$ with different resolution of texture can be applied.

6.1.3 Texture Value

The texture value is retrieved after corresponder function. For image textures, this is done by using the texture to retrieve texel information from the image. Procedural functions are sometimes used.

The most common texture values include RGB triplet, gray scale and RBGα value. The values returned from the texture are optionally transformed before use. One common example is the remapping of data from unsigned range to a signed range. Another one is to compare the value with a reference value and return a flag.

6.2 Image Texturing

The texture used in GPU has usually $2^m \times 2^n$ texels. Modern GPU allows for an arbitrary size. When finally projecting texture onto the screen, the projected square may contain *magnification* and *minification.* The filtering can take place in the input(values read from textures) or in the output(final pixel colors). When input and output is linear, there will be no difference.

6.2.1 Magnification

The most common filtering techniques for magnification is *nearest neighbor*(application of box filter) and *bilinear interpolation.* There is also *cubic convolution*, enabling high quality. It's not commonly available in the native hardware but can be realized in shader program.

One characteristic of nearest neighbor is that the individual texels may become apparent, which is called *pixelation* for every pixel only fetch one nearest texel to each pixel center.

In bilinear interpolation, for each pixel, this kind of filtering finds the four neighboring texels and linearly interpolates in two dimension to form a blended value for pixel. The cubic convolution is more expensive. Specifically, for a image coordinate (p_u, p_v). According to the coordinate, four centers of texels can be calculated.

$$\begin{aligned} x_l &= \lfloor p_u \rfloor, \quad y_b = \lfloor p_v \rfloor \\ x_r &= \lceil p_u \rceil, \quad y_t = \lceil p_v \rceil \end{aligned}$$

When interpolating:

$$u^{'} = p_u - \lfloor p_u \rfloor, \quad v^{'} = p_v - \lfloor p_v \rfloor$$

$$\begin{aligned} \mathbf{c}(p_v,\ p_u) = &(1-u^{'})(1-v^{'})\mathbf{t}(x_l,\ y_b) \\ &+ u^{'}(1-v^{'})\mathbf{t}(x_r,\ y_b) \\ &+ (1-u^{'})v^{'}\mathbf{t}(x_l,\ y_t) \\ &+ u^{'}v^{'}\mathbf{t}(x_r,\ y_t) \end{aligned}$$

This kind of interpolation will have poor performance when the image has a nonlinear change such as the edge of a object.

A common solution to the blurriness that accomplish magnification is to use *detail textures.* These are textures that represent fine surface details. The high-frequency of repetitive pattern of the detail texture, combined with low-frequency magnified texture will have a similar visual effect with high resolution texture.

Sometime, linear interpolation is not required. In order to magnify checkerboard, remapping will have better visual effect. Such remapping is useful in cases where well-defined edges are important, such as text. Alpha value can be used to be threshold the results. This method is called *cutout texture.* *Vector* *textures* can handle high quality edges generation. It stores information at texel describing how the edges cross the grid. These information can contain more subjects leading to accuracy and resolution-independence. However, it's too costly to apply to read-time rendering application.

There is simpler technique using *sampled distance filed* data structure. A distance filed is a scalar signed field over a space or surface in which an object is embedded(that means, the filed can be curved). At any point, the distance field has a value with a magnitude equal to the distance to the object's nearest boundary point. The value is negative within the object and positive outside the object. This will interpolated with the real distance rather than space coordinate so it's linear and fit bilinear interpolation to get a sampled distance filed for further rendering. The evaluation of sampled distance field is to use alpha map and built-in GPU interpolation and alpha testing rather than pixel shader modification.

6.2.2 Minification

Directly, we consider how to aggregate the influence of texels, which is hard to accomplish.

Nearest neighbor can be used to use one texel at the very center of the pixel to represent, which will cause big alias and even noticeable when surface is moving and result *temporal aliasing*. Similarly, bilinear interpolation can be used but don't perform much better than nearest neighbor.

According to Nyquist, we need to increase the pixel sample rate or the the texture frequency has to decrease. The previous chapter introduces increasing of sampling rate, but has a limited increase that don't fit minification. The design of anti-aliasing algorithm is to pre-process texture and create data structure to help with effect approximation.

The most popular method is called *mipmapping*. Mipmapping generate a set of images, called *mipmap chain*, with a level start at 0 and each level downsample to a quarter of the original area as the next level. level 0 is the original one and the downsampled ones are called subtexture.

Two important elements in forming high-quality mipmaps are good filtering and gamma correction. The common way to form a mipmap level is to take each 2×2 sets of texels and average them to get the mip level texel. This is actually a 2-dimensional box filter, which is better to use a Gaussian filter instead.

For textures encoded in a non-linear space (such as color texture), gamma correction is obviously needed to apply to the texture before mipmapping to a linear space. After filtering, converting texture back to non-linear space is also needed.

For the texture is fundamentally in a non-linear space, specialized mipmapping method is required.

In order to measure the coverage of a single pixel, d or λ is introduced. There two approaches to compute it: First we use the longest edge of quadrilateral formed by the pixel's cell to approximate; Or we can select the largest absolute value of $\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial y}$. Each differentials is a measure of the amount of change in the texture coordinate with respect to a screen axis. These gradient value is not accessible by dynamic flow control. And as the vertex shader can't access derivative data so it needs to be calculated first and then sent into GPU.

d is used to determine where to sample along the mipmap's pyramid axis. The goal of texel ratio is at least 1:1 according to Nyquist. The important principle is that the more texels a pixel covers and *d* increases, a smaller and blurrier version of texture will be accessed by a triplet (u, v, d). Instead of a integer representing mipmap level, *d* is a fractional value between two level. Using u, v coordinates we can get two bilinear texture values by bilinear interpolation and linear interpolated them using the decimal part of *d* to get the final value. This process is called trilinear interpolation and is performed in each pixel.

Level of detail bias(LOD bias) is used to control by being added to *d* in different circumstances. It can be specified for the texture as whole, or per-pixel in the pixel shader.

Instead of summing all the texels that affect a pixel, pre-combined set of texels are accessed and interpolated. It may lead to *overblurring*, which happens when camera looks along nearly edge-on. It may have a rectangular area. In order to avoid aliasing, we may use the largest derivative as *d* and result in relatively blurring. One extension to solve this is *ripmap*, which will also create rectangular texture. However, it's too costly to use in real-time rendering.

Another method to avoid overblurring is *summed-area table*(SAT). Every texel has more bits of precision and record its individual data(color for example) and the sum of all the corresponding texture's texels in the rectangle formed by this location and texel (0, 0). When a pixel grid is back-projected (which means project the pixel grid to the texture space; normally, we perceive as the texture projected on the pixel space on the screen) onto the texture, the projected gird is bounded by a rectangular. The average data in the rectangular will be returned as the texture value.

$$\mathbf{c} = \frac{\mathbf{s}(x_{ur}, y_{ur}) - \mathbf{s}(x_{ll}, y_{ur}) - \mathbf{s}(x_{ur}, y_{ll}) + \mathbf{s}(x_{ll}, y_{ll})}{(x_{ur} - x_{ll})(y_{ur} - y_{ll})}$$

However, both ripmap and summed-area table will have blurring image when a texture is viewed along the diagonal as the average of the whole texture will be averaged rather than the average the thin real projected rectangle.

Ripmap and summed-area table are examples of *anisotropic filtering* algorithm. They have good effect in horizontal and vertical directions but is memory intensive. Summed-area table are more suitable in modern GPU with more algorithms to optimize.

Other from anisotropic filtering, *unconstrained anisotropic filtering* is widely used. Similar to summed-area table, pixel grid is back-projected to a quad and sampled a number of time with associated squarish area. Instead of using a single mipmap sample to approximate the coverage, the algorithm uses a number of squares to cover the quad. The shorter side of the quad is used to determine *d* with a relatively small averaged are for each sample. The longer side creates a *line of anisotropy* parallel to the longer side and through the middle of the quad. Samples will be taken on the line and its amount is decided by the value of anisotropy.

This scheme allows the line of anisotropic to run in any direction (Correctly sample the diagonal view of texture as the line can be diagonal rather than horizontal and vertical in common anisotropic filtering). Meanwhile it has same cost as the mipmap does.

6.2.3 Volume Textures

A direct extension of image textures is three-dimensional image data that is accessed by (u, v, w) value. Being inside A single mipmap level requires a trilinear interpolation, while filtering between mipmaps require *quadrilinear interpolation.* This requires for high precision texture. Volume textures have the advantage that the good projection function which avoids distortion and seam.

6.2.4 Cube Maps

Another type of texture is the *cube texture* and *cube map.* The mapping point is specified by a three-dimensional vector. It points from the center of the cube and choose the texture face by the face with largest magnitude. (choose $-Z$ for $(1, 2, -3)$. The other entries will be divided by magnitude to be in range in [-1, 1] and can simply be mapped to range [0, 1]. It's useful for environmental mapping.

Cube maps support bilinear interpolation and mipmaps, but may have problems near the seam as the sample can't be across the boundaries of faces and faces can't affect each other. Another problem that the angular size of a texel varies over a cube face. That is, a texel at the center of a cube represents a greater visual solid angle than a texel at the corner.

6.2.5 Texture Caching

Memories for textures are always not enough. *Texture Caching* aims to a balance between the overhead of uploading textures to memory and the amount of of memory taken up by textures at one time.

Some general advice is to keep the texture no larger than necessary and to try to keep polygons grouped by their use of texture.

A *least recently used*(LRU) strategy is one commonly used in texture caching scheme. Every texture has a time stamp representing when it's last accessed. When new texture is uploaded, the texture with oldest time stamp is unloaded first. Priority is optionally given to prevent unnecessary texture swapping.

It's suggested to check the texture being swapped out for problems will happen when this texture is used in current frame. When it occurs, LRU has terrible performance. It better to first transfer to *most recently used* and after no textures are swapped out to switch back to LRU.

Prefetching where future needs are anticipated making the texture loading over a few frames as there may be sudden big amount of texture loading and it takes a lot of time.

When data set is huge, such as flight simulations program. The traditional approach is to break these images into smaller tiles that hardware can handle. A improved structure is *clipmap.* The entire data set is treated as a mipmap, but only a small part of the lower levels of the mipmap is required for certain view and memory.

6.2.6 Texture Compression

Texture Compression can directly attack memory and bandwidth problem. *S3 texture compression* becomes the standard for DirectX and called *DXTC* or *BC.* It has independently encoded pieces on 4×4 texel blocks, which is also called *tile*. Encoding is based on interpolation which is simple and fast. There are five variants:

DXT1 / BC1: It has two 16-bit reference RGB565 values as the limitation. Each texel has a 2-bit interpolation factor to refer to 2 references or 2 intermediate value, while only 1 intermediate value is included and another stands for transparent when alpha is introduced. The compression ratio is 6 : 1 compressing 24-bit RGB texture.

DXT3 / BC2: The encoding pattern is same as DXT1. Meanwhile, every texel has a 4-bit alpha value stored separately. The compression ratio is 4 : 1.

DXT5 / BC3: The encoding pattern is same as DXT1. In addition, alpha data is encoded using two 8-bit reference values. Each texel has 3 interpolation factor which can refer to one of the reference alpha values or one of the six intermediate alpha values.

ATI1/ BC4: It supports single color channel and encoded as the alpha channel in DXT5.

ATI2/ BC5: It supports dual color channels and encoded as the aloha channel in DXT5.

The disadvantage of these compressions is *lossy*. Once a tile has many distinct value it will lead to some loss. Fortunately, These compression scheme generally give acceptable image fidelity. Another problem is that all the colors are on a straight line in RGB space. *color distribution* can attack it as it uses colors from the neighbors blocks in order to achieve more colors.

For OpenGL ES, *Ericsson texture compression(ETC)* is chosen. It shares the same features as S3TC. It encodes 4×4 texels with 4 bits per texel. Each 2×4 or 4×2 block stores a basic color, and each texel in a block can select to add one of the values in the selected lookup table.

Compression of normal maps requires some care. We assume the normal is unit vector and the z-component is positive. So we can derive z by:

$$n_z = \sqrt{1 - n_x^2 - n_y^2}$$

So the three-dimensional texture can be stored as two-dimensional texture, a modest compression. Further compression is usually achieved in BC5 manner: A block of texel has 16 possible values, which can be bounded by a bounding box. So x, y values can be perceived as two channels of color and interpolation factor allows for value to choose from for each axis.

A fallback when GPU does not support BC5 can use DXT5 to substitute.

For normal maps, the aspect ratio decides the layout of the normal inside a block. For example, as the normal has a width nearly twice as height, the current 8×8 grids can be converted to 4×16 grids with different bits allocation.

6.3 Procedural Texturing

Procedural texture is to evaluate a function to look up texture value rather than generate texture space coordinate. It's often used in offline rendering system and not so common in real-time rendering. Procedural texture is used when image texture requires for costly memory access. Volume texture is a typical example.

Cellular texture, calculated the closest distance to a set a special points for every location. The color or shading normal will change by this distance.

Another type of application is on physical simulation like water ripples.

As parameterization will be more difficult for procedural texture, it's better to synthesis texture onto the surface directly. Anti-aliasing procedural texture becomes both easier and more difficult. On one hand, pre-computations methods are not available; On other hand, procedural texture have more inside information.

6.4 Texture Animation

The image applied and coordinate can both be dynamic. For example, dynamic coordinate for water downfall to make the water move. The texture can be applied to matrix transformation and blending techniques.

6.5 Material Mapping

A common use of a texture is to modify a material property affecting the shading equation. Shader can read values from the texture directly. For equation:

$$\mathbf{L}_o(\mathbf{v}) = (\frac{\mathbf{c}_{\text{diff}}}{\pi} + \frac{m+8}{8\pi}\overline{\cos}^m\,\theta_h \mathbf{c}_{\text{spec}}) \otimes E_L \overline{\cos}\,\theta_i$$

$\mathbf{c}_{\text{diff}}$, $\mathbf{c}_{\text{spec}}$ and m is the material information can be read from textures, particularly *diffuse color map, specular color map*(grayscale value rather than RGB for the reality) and *gloss map*(describing the smoothness). As $\mathbf{c}_{\text{diff}}$, $\mathbf{c}_{\text{spec}}$ is linear to the output, the input can be filtered without anti-aliasing, when m needs some care.

6.6 Alpha Mapping

The alpha map can be used for many interesting effect like decaling with clamp corresponder function and transparent edge to present the part you want. A similar application is in making cutouts that have 3D visual effect like cross tree. In order to reserve the visual effect from other angle, cross tree is set by rotating the cutouts. As illusion will break down when viewer see from the above, more cutouts can be added. Combining alpha texture and texture animations can make many visual effects.

Alpha map has many options like alpha blending. As discussed before, alpha blending needs a relative order. *Alpha test* is another option that conditionally discarding pixels with alpha values below a given threshold in the merge unit or pixel shader, which enable polygons to be rendered in any order.

Alpha to coverage, and the similar feature *transparency adaptive anti-aliasing*, take the transparency value of the fragment and convert this into how many samples inside a pixel are covered. This is similar to screen-door transparency but at a sub-pixel level. The alpha value determines the proportion of samples being covered within a pixel.

6.7 Bump Mapping

Bump mapping is used to represent small-scale detail which is implemented in per-pixel shading. The scale of detail on object can be classified into: *macro-features*, *meso-feature* and *micro-features*

Macro features cover many pixel. Macro-geometry is represented by geometric primitives, while micro-geometry is encapsulated in the shading model and used in pixel shader using texture maps as parameters for equation. Meso-geometry describe everything between these two scales.

Bumping mapping techniques are commonly used for mesoscale shading, adjusting the shading parameters(like normals) at pixel level in such a way that the viewer perceives small perturbations away from the base geometry. For example, when introducing a change to color component, instead of changing the base texture directly, we access another texture which is used to modify the surface normal. This will preserve the original geometry with different effect.

For bump mapping, the normal must change direction with respect to some frame of reference. *Tangent space basis* is stored per vertex used to transform the lights to a surface location's space to compute the effect of perturbing the normal and is expressed by *tangent and bitangent vectors* and corresponding basis matrix.

$$\begin{pmatrix} t_x & t_y & t_z & 0 \\ b_x & b_y & b_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Tangent and Bitangent don't have to truly be perpendicular to each other since the normal map itself maybe distorted to fit the surface. The normal can be calculated by the product of tangent and bitangent vectors, which can save memory with attention to handedness(e.g, in symmetric model). The handedness can be marked with a bit. The idea of tangent space is important for other algorithms like the orientation of the material.

6.7.1 Blinn's Methods

The original bump mapping method stores two offset values b_u and b_v for interpolated normals along **u** and **v** image axes. This type of of bump map texture is called an *offset vector bump map* or *offset map.*

Another way to represent bumps is to use *heightfield* to represent bumps so the direction of normals.

6.7.2 Normal Mappings

Normal map directly stores perturbed normals, represent the same effect with original bump mapping with different storage format. (x, y, z) values are mapped to [-1, 1], e.g., for an 8-bit texture the x-axis value 0 represents -1.0 abd 255 represents 1.0. It's similar to RGB values and light blue [128, 128, 255] represents flat surface.

Originally, the normal map is in world-space. However, this approach can't deal with any deformation, like rotation, which will lead to change in orientation. So it's rarely used.

Normal map can also be defined in object space. It now can deal with rigid transformation. Lights should be transformed into object space which can be done in application stage.

Tangent space is usually used. It allows for deformation of the surface and maximal usage of normal maps. It can be compressed as introduced before while requiring or more transformations.

There are two methods to use normal maps. If there is seldom lights, it's preferred to transform the light to tangent space as it move slowly per pixel and can be interpolated in triangle. However, it's preferred to transform perturbed normals to world space when there are more lights as it involves less transformation and avoid tangent space distortion.

Filtering normal maps is a difficult problem as the relationship between normal and shaded color is not linear. However, Lambertian surfaces are a special case where the normal map has an almost linear effect on shading. In Lambertian shading, it satisfies:

$$\mathbf{l} \cdot (\frac{\sum_{j=1}^{n} \mathbf{n}_j}{n}) = \frac{\sum_{j=1}^{n} (\mathbf{l} \cdot \mathbf{n}_j)}{n}$$

So Lambertian shading *almost* produce the right result while it's a *clamped* dot product. But in practice it's not objectionable.

For other surfaces, more details are included in following chapters.

6.7.3 Parallax Mapping

A problem with normal mapping is that the bumps never block each other. The idea of *parallax mapping* is an approximation to shift the pixel location to simulate this effect.

The amount to shift is based on the height retrieved and the angle of the eye to the surface.The heightfield values are scaled and biased before being used. The scale determines how height the heightfield meant to extend and the bias gives a sea-level height at which no shift take place. The equation follows:

$$\mathbf{p}_{\text{adj}} = \mathbf{p} + \frac{h \cdot \mathbf{v}_{xy}}{v_z}$$

Note that unlike most shading equations, here the view vector needs to be in tangent space. The new location also uses the same height as it's about the same. However, this method falls apart at shallow viewing angles. In order to solve this, limited the amount of shifting is needed:

$$\mathbf{p}_{\text{adj}}^{'} = \mathbf{p} + h \cdot \mathbf{v}_{xy}$$

It has some drawbacks, like lessen bumpiness at shallow angles, etc. But it is still considered the practical standard for bump mapping.

6.7.4 Relief Mapping

Parallax mapping is just an approximation. The actual effect we want is what is visible at the pixel, or where the view vector first intersects the heightfield.

Methods are similar to traditional ray tracing with different implementations. Relief mapping is actually a class of three independent research outcomes based on same approaches.

The key idea is to test a fixed number of texture samples along the projected vector in the cut of the surface along the view direction. The algorithm finds the first intersection of the eye ray with the line segments approximating the curved height field. Once the location is determined, the attached map will be used. It can also be used to have the bumpy surface casts shadows onto it self.

The problem of determining the actual intersection point is a root-finding problem. There is a critical importance in sampling the heightfield frequently enough. This can be done either by using better filtering methods or higher resolution of heightfield texture.

Another approach to increasing both performance and sampling accuracy is to not initially sample the heightfield at a regular interval, but instead to try to skip intervening empty space.

One problem with relief mapping methods is that the illusion breaks down along the silhouette edges of objects. The key idea is that the triangles rendered define which pixels should be evaluate by the pixel shader program, not where the surface actually is located. *Shell map* is introduced to extrude each polygon in the mesh outwards and form a prism. Rendering this prism forces evaluation of all pixels in which the heightfield could possibly appear.

6.7.5 Heightfield Texturing

Displacement mapping method has a flat meshed polygon access a heightfield texture and the height retrieved from the texture is used by the vertex shading program to modify the vertex's location. So the heightfield is called *displacement texture*. It's a similar concept to relief mapping. They both have a drawbacks like making collision detection more challenging.

7 Advanced Shading

7.1 Radiometry

Radiometry deals with the measurement of electromagnetic radiation consisting of a flow of *photons* which behave as either particles or waves. One wave-related property of photons which can't be disregarded is the fact that each has an associated frequency or wavelength, and the energy of each photon and interaction with rods and cones are to do with its frequency. Different frequencies of photons are perceived as light of different colors or not perceived at all. The relation between wavelength λ, frequency ν and energy Q:

$$\nu = \frac{c}{\lambda}$$

$$\lambda = \frac{c}{\nu}$$

$$Q = h\nu$$

where c is the speed of light and h is Planck's constant.

Electromagnetic radiation exists from ELF(extremely low frequency) radio waves to gamma rays. *visible spectrum* that is from roughly 380 to 780 nanometers. Colors are perceived for *monochromatic* light. The various radiometric units are presented below.

Wavelength:	m
Frequency:	Hertz
Radiant Energy:	J
Radiant flux:	W
Irradiance:	W/m^2
Radiant intensity:	W/sr
Radiance:	W/(m^{2}sr)

The *radiant flux* or *radiant power*, Φ or P, of a light source is equal to the number of joules per second emitted.

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

Irradiance is the density of radiant flux with respect to area.

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

irradiance E is used to measure light flowing into a surface and exitance M (also called *radiosity* or *radiant exitance* is used to measure light flowing out of a surface, and both of them can be referred by *radiant flux density.* The irradiance is constant for a single and further directional light. Assume a small light bulb in relatively large space so the light can be considered to be emitted from a point. When we want to examine the irradiance in a particular direction using a narrow cone, we can find:

$$E_L(r) \propto \frac{1}{r^2}$$

r is the distance from the light source to the surface.

$$E_L(r) = \frac{I}{r^2} \quad \Rightarrow \quad I = E_L(r)r^2$$

This quantity, I, is called *intensity* or *radiant intensity.* We observe when $r = 1$, intensity is flux density with respect to an area on an enclosing unit sphere, similar to the definition of radians. A *solid angle* is a three-dimensional extension of the concept, measured with *steradians*(sr). Using the definition of solid angle:

$$I = \frac{d\Phi}{d\omega}$$

When light sources can be treated as a point, it's called *point lights.* Intensity is useful for measuring the illumination of a point light source. If the distance from the light source is five times or more that of the light's width, the light source can be recognized as point light when considering the relationship between intensity and irradiance.

As mentioned before, *radiance* L measures density of light flow per unit area for a given ray direction, which is important for rendering.

$$L = \frac{d^2\Phi}{dA_{\text{proj}}d\omega}$$

where dA_{proj} refers to dA projected to the plane perpendicular to the ray. We can also perceive the projection as there is a mixed directional light perpendicular to a small area. The relationship between them is :

$$dA_{\text{proj}} = dA\overline{\cos}\theta$$

Where θ is the angle between two surface normals.

It's helpful to perceive:

$$L = \frac{dI}{dA_{\text{proj}}} = \frac{dE_{\text{proj}}}{d\omega}$$

The radiance in an environment can be thought of as a function of five variables (or six, including wavelength), called the *radiance distribution.* Three of the variables specify a location, the other two a direction. This function can describes all light traveling anywhere in space.

An important property of radiance is that it is not affected by distance. To perceive this, what is happening physically is that the solid angle covered by the light's emitting surface gets smaller as distance increases. For the light bulb example:

$$L^{'} = \frac{dE_{\text{proj}} \cdot \frac{k}{r_1^2}}{d\omega \cdot \frac{k}{r_1^2}} = L$$

7.2 Photometry

Photometry is like radiometry, except that ti weights everything by the sensitivity of the human eye. The results of radiometric computations are converted to photometric units by multiplying by the *CIE photometric curve*, a bell- shaped curve centered around 555nm that represents the eye's response to various lengths of light. The another difference is the unit.

Photometry does not deal with the perception of color itself, but rather with the perception of brightness from light of various wavelengths. The units for photometry:

Luminous Energy:	talbot
Luminous Flux:	lumen(lm)
Illuminance:	lux(lx)
Luminous intensity:	candela(cd)
Luminance:	$\mathrm{cd/m^2} = \mathrm{nit}$

Luminance is often used to describe the brightness of flat surfaces.

7.3 Colorimetry

Light from a given direction consists of a set of photons in some distribution of wavelengths. This distribution is called the light's *spectrum.* Three numbers can be used to precisely represent any spectrum seen as only three different signal can be receives.

By color matching experiments, as color can be divided two three base color in [-1, 1] knob range or energy value, three values are derived for each wavelength and for every wavelength it forms a color matching curve. The relative amount of three values decide the visual color. Given an arbitrary spectrum, multiply it with three color matching curve and the integral of the resulting curves give the relative amount of three values. So different spectra can resolve to the same three weights and representing same color perceived. Matching spectra presenting same color are called metamers.

r, g, b value can not represent all color directly for negative weights. $\overline{x}(\lambda)$, $\overline{y}(\lambda)$ and $\overline{z}(\lambda)$ is used instead as no negative weight appears. $\overline{y}(\lambda)$ color is the same one as the photometric curve. The surface reflectance and light source will define a color function $C(\lambda)$. Like r, g, b weights:

$$X = \int_{380}^{780} C(\lambda)\overline{x}(\lambda)d\lambda$$

$$Y = \int_{380}^{780} C(\lambda)\overline{y}(\lambda)d\lambda$$

$$Z = \int_{380}^{780} C(\lambda)\overline{z}(\lambda)d\lambda$$

These X, Y, and Z *tristimulus values* are weights that define a color in CIE XYZ space. A better three-dimensional space is use the plane $X + Y + Z = 1$ as the relative weights count.

$$x = \frac{X}{X+Y+Z}$$

$$y = \frac{Y}{X+Y+Z}$$

$$z = \frac{Z}{X+Y+Z}$$

As z value is omitted, the plot of the *chromaticity coordinates* x and y values is known as the *CIE chromaticity diagram.* The curved line in the diagram shows where the spectrum lie, and the straight line connecting the ends of the spectrum is called the *purple line.* For $x = y = z = \frac{1}{3}$, it's used to define white. For a computer monitor, the *white point* is the combination of the three color phosphors at full intensity.

Given a color point (x, y), draw a line from the white point through this point to the spectral line. The relative distance of the color point compared to the distance to the edge of the region is the *saturation* of the color. The point on the region edge defines the *hue* of the color.

The third dimension to fully describe a color is the Y (the curve is the same as photometric curve) value, luminance, defining a xyY coordinate system. The display system is limited in its ability to present different colors by the spectra of its three channels. An obvious limitation is in luminance and saturation.

The triangle in the chromaticity diagram represents the *gamut* of a typical computer monitor. The three corners of the triangle are the most saturated red, green and blue colors. An important property of the chromaticity diagram is that these limiting colors can be joined by straight lines to show the limits of the monitor as a whole.

Conversion from XYZ to RGB space is linear:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 3.24049 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

This conversion matrix is for a monitor with a D65 white point. Some XYZ values can transform to RGB values that are negative or greater than one. Theses are colors out of gamut. A common conversion is to transform a RGB color to a grayscale luminance value is use the inverse of the matrix:

$$Y = 0.212671R + 0.715160G + 0.072169B$$

While any given spectrum can be precisely represented by an RGB triplet, there is some contrast. For example, multiplying two RGB colors is not the sam as multiplying two spectra. However, in practice ,multiplying RGB values together works well.

7.4 Light Source Types

The combination of light source and housing is called a *luminaire.* In rendering it's common to model the luminaire as a light source with a directional distribution that implicitly models the effect of the housing.

Directional lights are the simplest model. They are fully described by $\mathbf{l}$ and E_l. Recall E_L is expressed as an RGB vector for rendering purposes.

Point lights have a softer assumption compared to directional light that can be applied to more circumstances.

For all variants of directional and point lights introduce in this section, they follow the same assumption: at a give surface location, each light source illuminates the surface from one direction only.

7.4.1 Omni Lights

Point lights with a constant value for I_L are known as *omni lights.* I_L is also expressed as an RGB vector. For computations:

$$r = \|\mathbf{p}_L - \mathbf{p}_S\|$$

$$\mathbf{l} = \frac{\mathbf{p}_L - \mathbf{p}_S}{r}$$

$$E_L = \frac{I_L}{r^2}$$

It's often preferable to use *distance falloff functions* instead to describe the relationship between E_L and distance.

$$E_L = I_L f_{\text{dist}}(r)$$

Distance falloff functions can proved ore control for lighting. One historically important falloff function:

$$f_{\text{dist}}(r) = \frac{1}{s_c + s_l r + s_q r^2}$$

where s_c, s_l and s_q are properties of light source.

A much simpler one used in games and modeling:

$$f_{\text{dist}}(r) = \begin{cases} 1 & \text{where } r \leqslant r_{\text{start}} \\ \frac{r_{\text{end}} - r}{r_{\text{end}} - r_{\text{start}}} & \text{where } r_{\text{start}} < r < r_{\text{end}} \\ 0 & \text{where } r \geqslant r_{\text{end}} \end{cases}$$

7.4.2 Spotlights

In reality, different visual effects can be produced using different functions to describe how I_L varies with direction. One important type of effect is *spotlight.* The angle θ_s is defined as the angle between the spotlight direction $\mathbf{s}$ and $-\mathbf{l}$. In OpenGL:

$$I_L(\mathbf{l}) = \begin{cases} I_{L_{\max}}(\cos\theta_s)^{s_{\exp}} & \text{where } \theta_s \leqslant \theta_u \\ 0 & \text{where } \theta_s > \theta_u \end{cases}$$

s_{exp} controls the tightness and θ_u controls the edge as the *umbra angle.*

In DirectX:

$$I_L\mathbf{l} = \begin{cases} I_{L_{\max}} & \text{where } \cos\theta_s \geqslant \cos\theta_p \\ I_{L_{\max}}(\frac{\cos\theta_s - \cos\theta_u}{\cos\theta_p - \cos\theta_u})^{s_{\exp}} & \text{where } \cos\theta_u < \cos\theta_s < \cos\theta_p \\ 0 & \text{where } \cos\theta_s \leqslant \cos\theta_u \end{cases}$$

The angle θ_p defines the *penumbra angle* of the spotlight, or the angle at which its intensity starts to decrease.

7.4.3 Textured Lights

Textures can be used to add visual richness to light sources and allow for complex intensity distribution or spotlight functions. Projected textures can be sued for projector effects limited in a frustum. These lights are often called *gobo* or *cookie* lights.

For lights that are not limited to a frustum but illuminate in all directions, a cube map can be used. Textures can be added to any light type to enable additional visual effects. Textured lights allow for easy control of the illumination by artists.

7.5 BRDF Theory

7.5.1 The BRDF

In radiometry, the function that is used to describe how a surface reflects light is called the *bidirectional reflectance distribution function*(BRDF). The precise definition of the BRDF is the ratio between differential outgoing radiance and differential irradiance.

$$f(\mathrm{l}, \mathrm{v}) = \frac{dL_o(\mathbf{v})}{dE(\mathbf{l})}$$

The value of the BRDF depends on wavelength, so for rendering purposes it is represented as an RGB vector.

For non-area light sources, the BRDF definition can be expressed in a non-differential form.

$$f(\mathrm{l}, \mathrm{v}) = \frac{L_o(\mathbf{v})}{E_L(\mathbf{l})\overline{\cos}\theta_i}$$

So for evaluating radiance in a particular viewing direction, it's straightforward with n non-area light sources:

$$L_o(\mathrm{v}) = \sum_{k=1}^{n} f(\mathbf{l}_k, \mathbf{v}) \otimes E_L\overline{\cos}\theta_{i_k}$$

In order to describe an direction($\mathbf{l}$ or $\mathbf{v}$), the parameterization is to use two angles: elevation θ relative to the surface normal and rotation ϕ about the normal. This gives total four scalar variables in the general case of BRDF. *Isotropic* BRDFs remain the same when the incoming and outgoing direction are rotated around the surface normal with same relative angles. They just have three scalar variables.

The BRDF is defined as radiance dived by irradiance, so its units are sr^{-1}. Intuitively„ the BRDF value is the relative amount of energy reflected in the outgoing direction, given the incoming direction.

Lights interaction with optical discontinuity will have various phenomena occur as discussed before. The phenomena called *subsurface scattering* describe light that transmitted into the object may return and exit the surface with absorption and scattering. There are two circumstances when rendering: When the area covered by a pixel is relatively small compared to the distance between the entry and exit locations of subsurface scattering, BRDF can't be used. Instead *bidirectional surface scattering reflectance distribution function* (BSSRDF)is used to exhibit large-scale subsurface scattering.

When a pixel covers a relatively large area, reflection, refraction and subsurface scattering can be approximated as happening at a single point, allowing for the usage of BRDF. This shows that whether a BRDF can be used depends both on the surface material and observation scale.

The derivation of BRDF assumes BRDF as a property of uniform surfaces, which is unreal. A function that captures BRDF variation based on spatial location is called a *spatially varying BRDF*(SVBRDF) or *SBRDF*. To handle transmission of light, two BRDFs and two BTDFs are defined for the surface, one for each side, and so make up the BSDF. In most situation, BRDF or SVBRDF is sufficient.

The laws of physics put two constraints on BRDF: the first one is *Helmholtz reciprocity*

$$f(\mathbf{l}, \mathbf{v}) = f(\mathbf{v}, \mathbf{l})$$

In practice, BRDF will violate it without noticeable artifacts.

The second constraint is conversion of energy: the outgoing energy con not be greater that the income energy. In the real time rendering, strict energy conversion is not necessary, but approximate energy conversion is desired.

The *directional-hemispherical reflectance* $R(\mathbf{l})$ is a function related to the BRDF. It measures measures the amount of light coming from a given direction that is reflected at all, regardless of outgoing direction. Essentially, it measures the energy loss for a given incoming direction.

$$R(\mathbf{l}) = \frac{dM}{dE(\mathbf{l})}$$

For non-area light, non-differential form can be used:

$$R(\mathbf{l}) = \frac{M}{E_L \overline{\cos}\theta_i}$$

Like BRDF, $R(\mathbf{l})$ is expressed in RGB form. Besides, it is in range [0,1] so it can be considered as RGB color. Note that the restriction of energy conversion does not apply to BRDF with arbitrarily high values in certain directions. The relationship between directional-hemispherical reflectance and BRDF:

$$R(\mathbf{l}) = \int_{\Omega} f(\mathbf{l}, \mathbf{v}) \cos\theta_o d\omega_o$$

where θ_o is the angle between $\mathbf{n}$ and $\mathbf{l}$.

The most simple BRDF model is Lambertian BRDF. It has a constant value and usually represents subsurface scattering. The directional-hemispherical is also a constant value:

$$R(\mathbf{l}) = \pi f(\mathbf{l}, \mathbf{v})$$

The constant reflectance value of a Lambertian BRDF is commonly referred to as the *diffuse color* $\mathbf{c}_{\text{diff}}$:

$$f(\mathbf{l}, \mathbf{v}) = \frac{\mathbf{c}_{\text{diff}}}{\pi}$$

So for lambertian shading equation:

$$L_o(\mathbf{v}) = \frac{\mathbf{c}_{\text{diff}}}{\pi} \otimes \sum_{k=1}^{n} E_{L_k} \overline{\cos}\theta_{i_k}$$

The $\frac{1}{\pi}$ is usually integrated into E_{L_k}.

For visualized BRDFs with fixed incoming direction, the spherical terms terms reflect diffuse term as it goes to every direction. The *reflectance lobe* represents the specular and the thickness of it represents the fussiness of the reflection.

7.5.2 Surface and Body Reflectance

When using BRDF representations, surface phenomena are simulated as *surface reflectance*, and the interior phenomena are models as *body reflectance*(or *volume reflectance*). The surface is a optical discontinuity and as such scatters light with reflections and transmissions but does not absorb any; The object's interior contains matter, which may absorb some of the light and this process is under the surface. It may also contain additional optical discontinuities that will further scatter the light. Surface reflectance is modeled by specular BRDF terms while body reflectance is modeled with diffuse term.

7.5.3 Fresnel Reflectance

The interaction of light with a planar interface between two substances follows the *fresnel reflectance.* Perfect flat surface is expected, while in practice a surface can be considered as perfectly flat with irregularities much smaller than the smallest light wavelength.

An optically planar interface between two substances will scatter light into ideal reflection direction and ideal refraction direction.

Note that the ideal reflection direction forms the same angle with the surface normal $\mathbf{n}$ as the incoming direction $\mathbf{l}$. The amount of light reflected is described by the *Fresnel reflectance* R_F. So the proportion of transmitted flux is $1 - R_F$ due to the energy conversion. However, radiance proportion of transmitted light is different due to the projected area and solid angle:

$$L_t = (1 - R_F(\theta_i)) \frac{\sin^2 \theta_i}{\sin^2 \theta_t} L_i$$

The refraction vector r_i can be computed:

$$r_i = -\mathbf{l} + 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n}$$

The relationship between the θ_t and θ_i is described by Shell's Law with *refraction index* or *index of refraction*:

$$n_i \sin \theta_i = n_t \sin \theta_t$$

So for the transmitted radiance:

$$L_t = (1 - R_F(\theta_i)) \frac{n_t^2}{n_i^2} L_i$$

The importance of Fresnel equation is description of the dependence of R_F, θ, n_1 and n_2.

External Reflection is the case where light reflects from an object's external surface. In other word, the light is traveling from air to the object.

For rendering purpose, $R_F(\theta_i)$ is treated as RGB vector. $R_F(0°)$, sometimes called *normal incidence*, can be thought of as the characteristic specular color of the substance. As θ_i increases, $R_F(\theta_i)$ intends to increase. When $\theta_i = 90°$, the $R_F(\theta_i)$ reaches 1 for all frequencies. In rendering, the increase in reflectance at glancing angles is called *Fresnel effect*.

Besides the complexity of $R_F(\theta_i)$, they require refractive index values sampled over the visible spectrum, which make it difficultly to directly use in rendering. A simple approximation for most substances of Fresnel equation:

$$R_F(\theta_i) = R_F(0°) + (1 - R_F(0°))(1 - \overline{\cos}\theta_i)^5$$

The approximation performs not so well when $R_F(\theta_i)$ is not a monotonic. For precise value, it's better to pre-compute these value into a one-dimensional lookup texture. $R_F(0°)$ is the only control parameter here, which is convenient. It can be derived from object's refractive index:

$$R_F(0°) = (\frac{n-1}{n+1})^2$$

If refractive indices vary from frequencies, it's better to converted to RGB vector using methods described at 7.3

For typical objects, according to their $R_F(0°)$, can be divided to insulators(or dielectrics), metals and semiconductors. Semiconductors too rare to put into discussion.

Most encountered materials are insulators with $R_F(0°)$ lower than 0.05. This makes fresnel effect more obvious. Besides, the optical properties of insulators rarely vary much over the visible spectrum, leading to colorless reflectance value. The light may go over further scattering and absorption; It may have internal reflection going through transparent objects.

Metals have high value of $R_F(0°)$, almost always 0.5 or above. The optical properties of insulators vary over the visible spectrum, leading to colorful reflectance value. The light will be immediately absorbed.

Internal reflection and External reflection have different refractive indices determine different Fresnel reflectance.

Internal Reflection occurs when light is travelling in the interior of a transparent object and encounters the object's surface.

The difference between external reflection and inter reflection is the *critical angle* for *total internal refraction*. The $R_F(\theta_i)$ curve of internal reflection is a compressed version of that of external reflection with same $R_F(0°)$. The compute to critical angle is straightforward:

$$\sin\theta_c = \frac{n_2}{n_1} = \frac{1 + \sqrt{R_F(0°)}}{1 - \sqrt{R_F(0°)}}$$

The approximation for external reflectance can be used in internal reflectance when θ_t substitutes θ_i.

7.5.4 Local Subsurface Scattering

For insulators, body reflectance needs to be taken into consideration. If the insulators are *homogeneous*, with few internal discontinuities to scatter light, like glass; they partially absorb bur don't change their direction.

Most insulators are *heterogeneous*, containing numerous discontinuities. These will cause light to scatter inside the substance. The light will be partially absorbed and those is not absorbed will re-emitted. We assume that the light is re-emitted from the same point at which it entered. This assumption is called *local subsurface scattering.*

The *scattering albedo of* ρ of a heterogeneous insulators is the ratio between the energy of the light that escapes a surface compared to the energy of the light entering into the interior of the material. ρ is in range [0,1] and is modeled into RGB vectors. The bigger ρ is, the more light scattered and the brighter the objects are visualized.

Since insulators transmit most incoming light rather than reflecting it at the surface, ρ is more visually important than the $R_F(\theta_i)$. Since it results from a completely different physical process than the specular color, ρ may have a completely different spectra distribution.

As introduced before, the diffuse term of BRDF in Lambertian is usually used to represent local subsurface scattering. Recall:

$$f(\mathbf{l},\mathbf{v})_{\text{diff}} = \frac{\mathbf{c}_{\text{diff}}}{\pi}$$

To set $\mathbf{c}_{\text{diff}}$, we should first subtract specular term and get the diffuse term. If we have constant specular directional-hemispherical reflectance, it can be computed as :

$$\mathbf{c}_{\text{diff}} = (1 - \mathbf{c}_{\text{spec}})\rho$$

Only using ρ to present $\mathbf{c}_{\text{diff}}$, it just count specular term into subsurface scattering.

As fresnel reflectance represents the proportion of reflection, the diffuse term can be computed as:

$$f_{\text{diff}}(\mathbf{l},\mathbf{v}) = (1 - R_F(\theta_i))\frac{\rho}{\pi}$$

It seems reasonable for the diffuse term is not affected by $\mathbf{v}$. However, as it's implied by reciprocity and fresnel reflectance direction preference, the outgoing direction should be taken into consideration.

The assumption support both energy conservation and Helmholtz reciprocity:

$$f_{\text{diff}}(\mathbf{l},\mathbf{v}) = k_{\text{norm}}(1 - R_{\text{spec}}(\mathbf{l}))(1 - R_{\text{spec}}(\mathbf{v}))\rho$$

where k_{norm} is a constant computed to ensure energy conversion. Given a specular BRDF term, using this equation to derive a matching diffuse term is not trivial since in general R_{spec} does not have a closed form.

7.5.5 Microgeometry

Surface detail modeled by a BRDF is *microscale* – smaller than the visible scale, or in other words, smaller than a single pixel. Since such *microgeometry* is too small to be seen directly, its effect is expressed statistically in the way light scatters from the surface.

The most important visual effect of the microgeometry is due to the fact that many surface normals are present at each visible surface point. As normal can determine the outgoing direction, and the directions are somewhat random, it make sense to model them statistically as a distribution. For most surfaces, the distribution of microgeometry surface normals is a continuous distribution with a strong peak at the macroscopic surface normal. The tightness of this distribution determined by surface normal.

The visible effect of increasing microscale roughness is greater blurring of reflected environmental detail. It also can be seen that statistical patterns in the many small highlights eventually become details in the shape of the resulting aggregate highlight.

For most surfaces, the distribution of the microscale surface normals is isotropic. While some surfaces have microscale structure that is *anisotropic*, resulting in directional blurring of reflections and highlights. Some surfaces have highly structured microgeometry, resulting in interesting microscale normal distributions and surface appearance, e.g., fabric.

Other effects can also be important.*Shadowing* refers to occlusion of the light source by microscale surface detail. *Masking* refers to the visibility occlusion of microscale surface detail. If there is a correlation between this and surface normal, the distribution will change. As glancing angle θ_i increases, the irregularities will decrease. When $\theta_i \approx 90$, the surface will become optically smooth. This effect will be reinforced by Fresnel effect.

Light may experience inter reflections. Under the effect of Fresnel reflection, it tends to be attenuated and not noticeable in insulators. In metals, this is the source of diffuse reflection. Note that the reflection will make the light more colored.

In certain cases, microscale surface detail can affect body reflectance. If the scale of the microgeometry is large relative to the scale of subsurface scattering, then shadowing and masking can cause a *retro-reflection* effect, where light is preferentially reflected back toward the incoming direction .