Real-Time Rendering Note

Ianaesthetic

September 25, 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.

Model & View Transform $\longrightarrow$ *Vertex Shading* $\longrightarrow$ *Projection* $\longrightarrow$ *Clipping* $\longrightarrow$ *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 offscreen 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 blankslates. 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** 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 rasterizeraion 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 $\mathbf{s}$ 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:

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

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:

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

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**:

$$\mathbf{A} = \mathbf{Q}\mathbf{D}\mathbf{Q}^T$$

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**:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

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:

$$\mathbf{M} = \mathbf{A}\mathbf{A}^T = (\mathbf{U}\mathbf{S}\mathbf{V}^T)(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T = \mathbf{U}\mathbf{S}\mathbf{V}^T\mathbf{V}\mathbf{S}^T\mathbf{U}^T = \mathbf{U}\mathbf{S}^2\mathbf{U}^T$$

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 $\times$3 part).

$$\mathbf{A} = \mathbf{R}\mathbf{S}\mathbf{R}^T$$

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 $(\lambda_1\ \lambda_2\ \lambda_3)$
3. Rotate the coordinate to the original one.

It's similar for singular decomposition which form is:

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

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 $\times$ 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:

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

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 $\cos\phi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$ 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:

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

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; **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 **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 **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:

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

$$= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

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

$$= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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 irradiace measured perpendicular to **l** times the cosine of the angle θ_i between **l** and normal **n**.Irraidance is proportional to the *density* of light and inversely proportional to the *distance* between rays. We use E to stand for irriaidance:

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

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 chage 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 **M**. Light-matter interaction is linear. The ratio of existance and irradiance is between 0 to 1 in opaque objects and differentiates in colors. There are represented as RGB vector called the *surface color* **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):

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

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

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 qute 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 subexpressions, $\mathbf{K}_d$ 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 $\mathbf{p}_v$:

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

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

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

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 *sumpersampling* (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. 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 on the fragment.The z-depth is also computed, even slope is retained for precise z-depth value. 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. 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. This can be avoided by using converge mask or by blurring the edges outwards. Ins 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. This allows images to be given an outline shape by using just RGB values with out α. The drawback is that the actual *alpha* is 0.0 or 1.0.

Thought **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 eys, 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. LCDs have 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:

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

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. It's responsible of the render to convert physical radiance value into nonlinear frame buffer values by applying the appropriate transfer function. This *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 GPUs 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. 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.

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