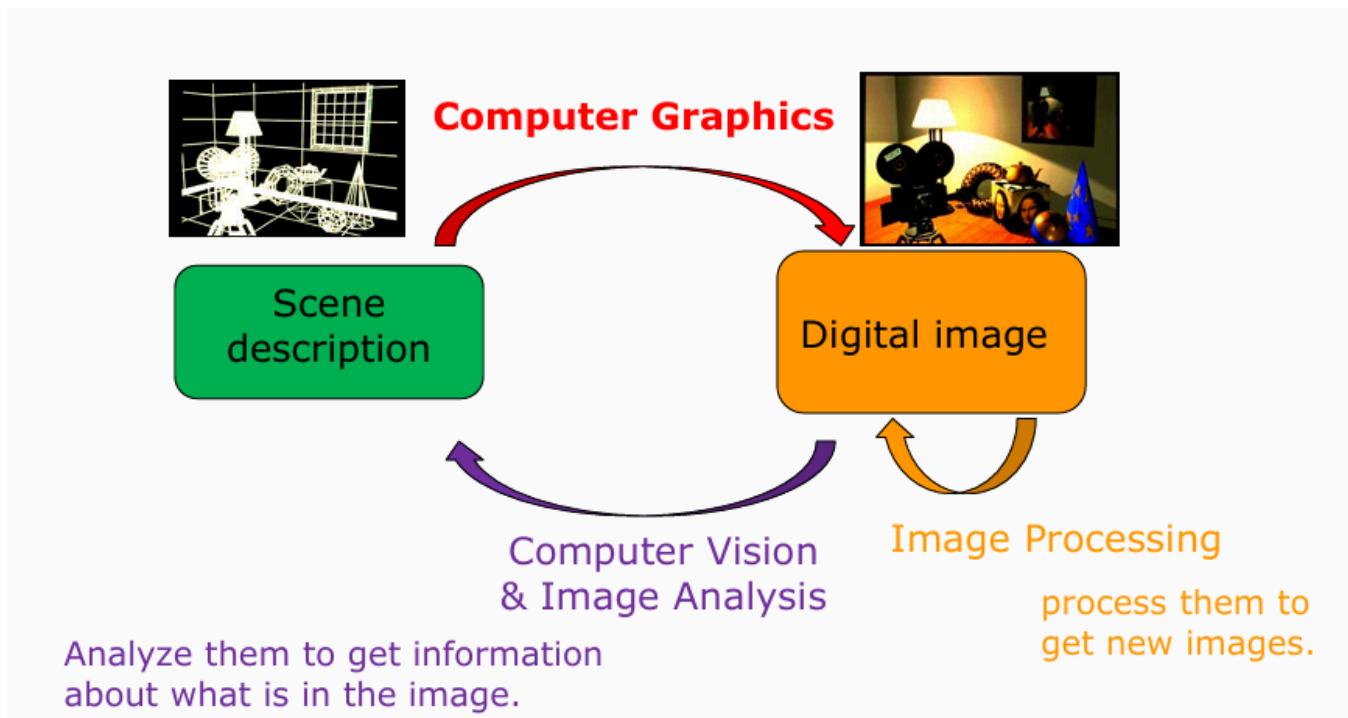


CG Review

1. Introduction

What is Computer Graphics?



Computer Graphics: Scene description to Digital image, generates images from 2D/3D models

计算机图形学：从场景描述到数字图像，从二维/三维模型生成图像

Image Processing: image to image, filtering, enhancement and other operations on images to generate new images

图像处理：图像到图像、过滤、增强以及对图像进行其他操作以生成新图像

Computer Vision & Image Analysis: image to scene description, extract information from images

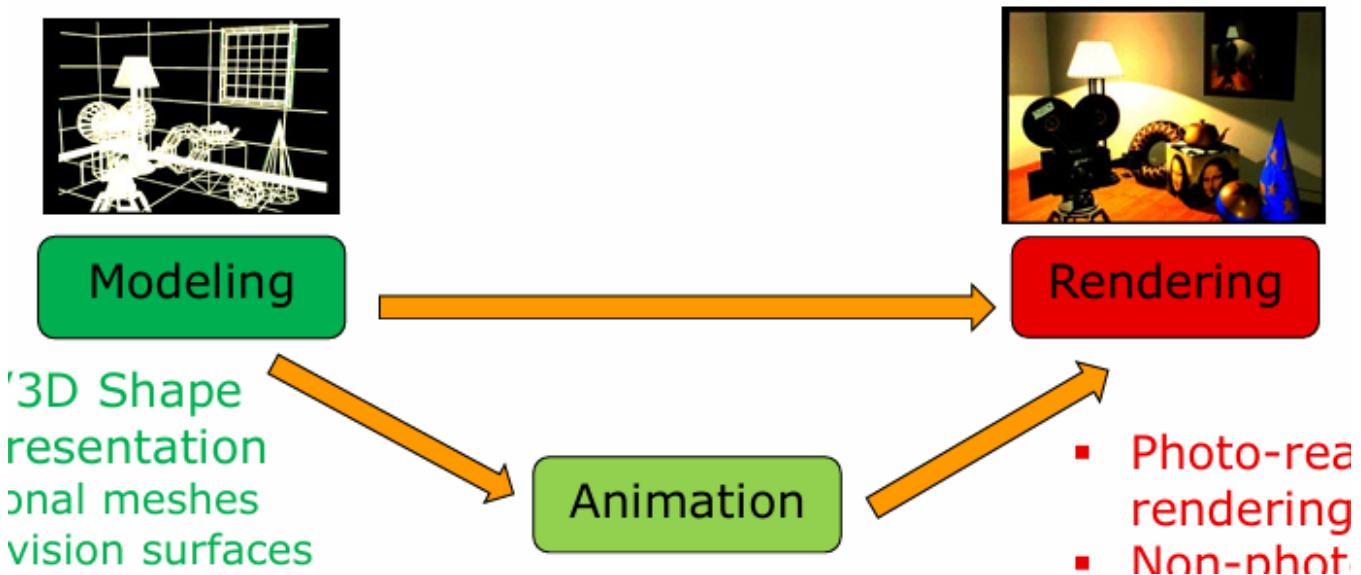
计算机视觉与图像分析：从图像到场景描述，从图像中提取信息

2D/3D Shape Representations

- 3D point clouds 3D点云
- Polygonal meshes 多边形网格
- Subdivision surfaces 细分曲面
- Implicit Surfaces 隐式曲面
- Parametric surfaces 参数曲面
- Voxel Representation 体素表示法
- Constructive Solid Geometry (CSG) 建构实体几何

- Fractals 分形

How Computer Graphics Works?



Modeling: data structures for representing 2D/3D objects and scenes.

建模：用于表示二维/三维物体和场景的数据结构。

Rendering: convert models with geometry/materials/lighting to digital images

渲染：将带有几何图形/材料/照明的模型转换为数字图像

Animation: manipulate the 2D/3D objects according to the laws of physics

动画：根据物理定律操纵 2D/3D 物体

2. Object Modeling

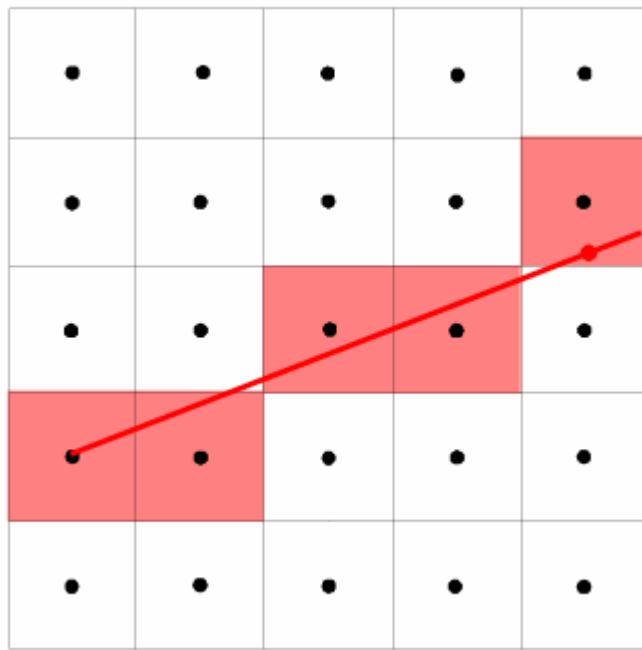
2D Drawing

Drawing an object involves converting the object into pixel pattern and updating the corresponding pixels in the frame buffer.

绘制对象包括将对象转换为像素图案，并更新帧缓冲区中的相应像素。

Drawing a Thin Line

given a line $y = mx + c$

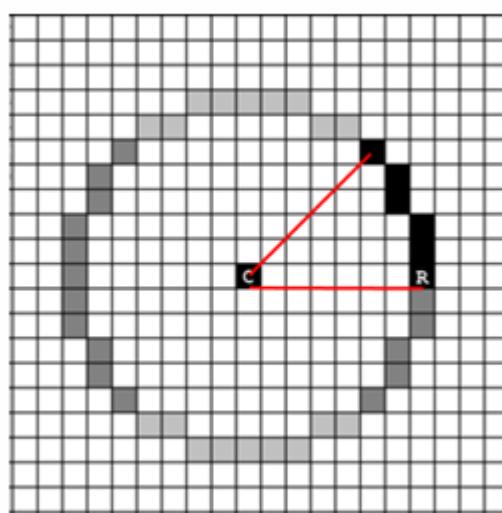


steps:

1. find a pixel (x_1, y_1) , set x to x_1 , y to y_1 查找像素 (x_1, y_1) , 设置 x 为 x_1 , y 为 y_1
2. Increase x by 1, correspondingly get $y = y + m$ 将 x 增加 1, 相应得到 $y = y + m$
3. Compute D1, which is the distance of (x, y) from the center of the upper pixel; 计算 D1, 即 (x, y) 与上像素中心的距离;
4. Compute D2, which is the distance of (x, y) from the center of the lower pixel; 计算 D2, 即 (x, y) 到下像素中心的距离;
5. If $D1 < D2$ (i.e., the line is closer to the upper pixel), shade the upper pixel; otherwise, shade the lower pixel. 如果 $D1 < D2$ (即线条更靠近上像素), 则对上像素进行阴影处理; 否则, 对下像素进行阴影处理。
6. keep the loop of 2-6 until reaching the endpoint (x_2, y_2) 保持 2-6 的循环, 直到到达终点 (x_2, y_2)

Drawing a Circle

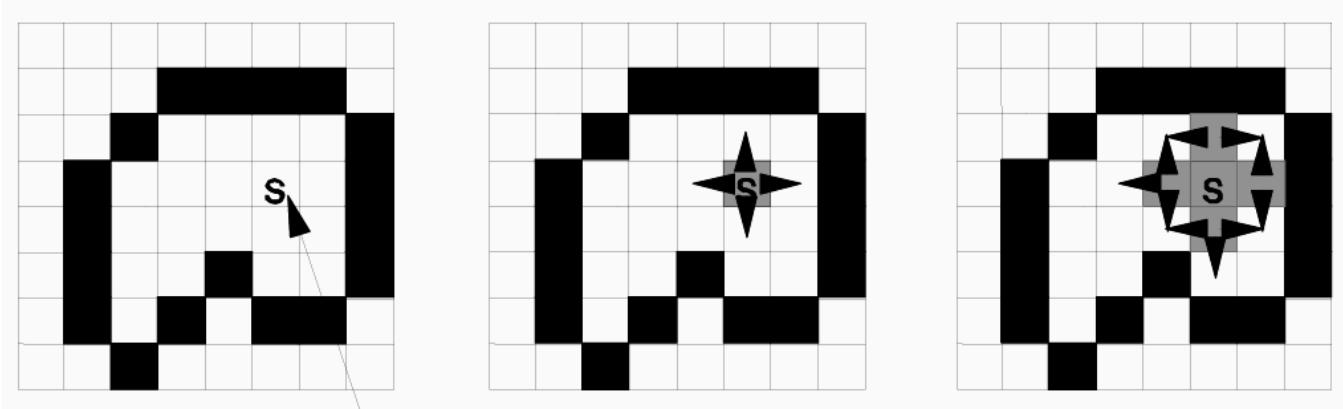
given a circle $(x - x_c)^2 + (y - y_c)^2 = r^2$



- At each step, the path is extended by choosing the adjacent pixel which minimizes $|(x - x_c)^2 + (y - y_c)^2 - r^2|$

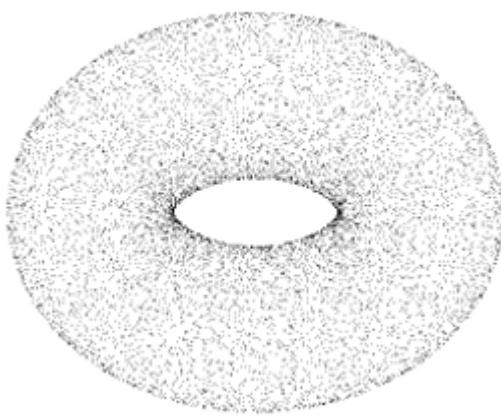
- 在每一步中，通过选择相邻像素来扩展路径，从而使 $|(x - x_c)^2 + (y - y_c)^2 - r^2|$ 最小化。
- By making use of the symmetrical property of a circle, we only need to calculate pixels of 1/8 circle.
- 利用圆的对称特性，我们只需计算 1/8 圆的像素。

Region Filling



1. Set border pixels to border color 将边框像素设置为边框颜色
2. Choose a pixel inside the region as start point 在区域内选择一个像素点作为起点
3. If the current pixel has a value the same as the border color or the fill color, we stop here. 如果当前像素的值与边框颜色或填充颜色相同，我们就到此为止。
4. Otherwise, we set the current pixel to the fill color, Usually this is the same as the border color. 否则，我们会将当前像素设置为填充色，通常这与边框颜色相同。
5. We recursively set the left, right, top and bottom pixels as the current pixel and repeat the checking process. 我们递归设置左、右、上、下像素为当前像素，并重复检查过程。

3D Point Clouds



Unstructured set of 3D point: Each point consists of geometry information (x, y, z) and other attributes, e.g. color (r, g, b)

非结构化三维点集合： 每个点都包含几何信息 (x, y, z) 和其他属性，例如颜色 (r, g, b)

Widely used in autonomous driving.

广泛应用于自动驾驶。

Advantage:

- Real-time acquisition 实时采集
- Connectivity information-free 无连接信息
- Suitable for real-time applications 适合实时应用

Disadvantage:

- Difficult to perform geometry computation 难以进行几何计算

Polygon Meshes

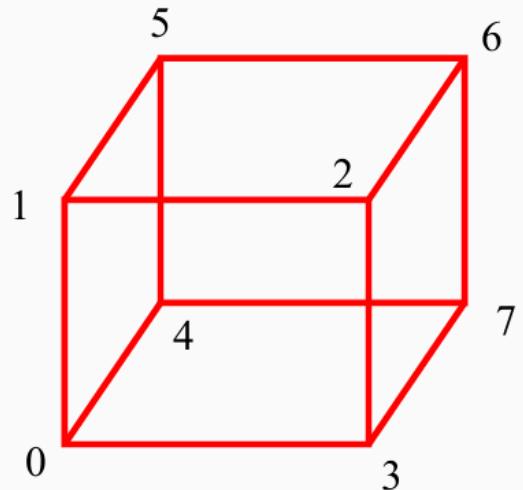
A polygon mesh M is a 3-tuple (V, E, F) , where:

- V is the set of vertices
- E is the set of edges (An edge is a boundary edge if it belongs to only one face)
- F is the set of faces

多边形网格 M 是一个三元组 (V, E, F) ，其中

- V 是顶点集合
- E 是边的集合（如果一条边只属于一个面，那么它就是边界边）
- F 是面的集合

```
void colordcube( )
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```



Use the right-hand rule to indicate outward facing polygons

用右手定则表示朝外的多边形

For a **convex polyhedron** (a closed manifold mesh without holes/handles), there is:

对于**凸多面体**（无孔/无柄的封闭流形网格），有：

$$|V| - |E| + |F| = 2$$

For a closed orientable manifold mesh of genus g (num of holes):

对于属数为 g 的封闭可定向流形网格 (孔洞数) :

$$|V| - |E| + |F| = 2 - 2g$$

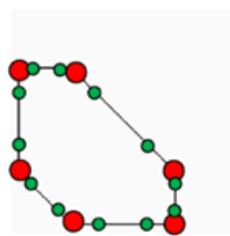
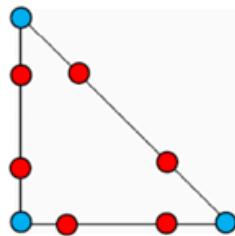
For a genus-g orientable manifold mesh with b boundaries:

对于具有 b 个边界的 g 属可定向流形网格来说

$$|V| - |E| + |F| = 2 - 2g - b$$

Subdivision Surfaces

细分曲面



Corner cutting

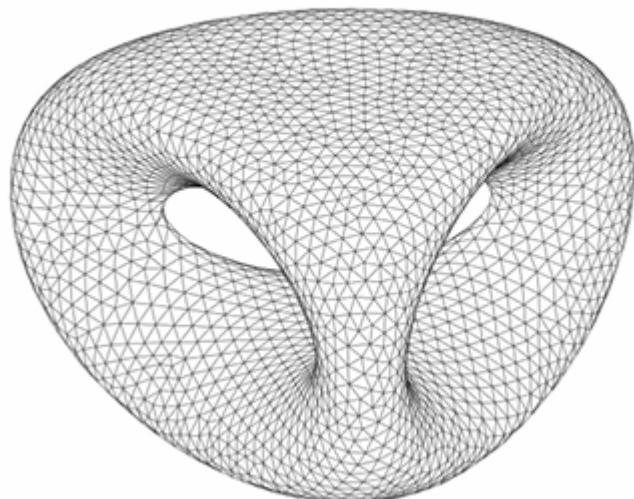
Increase the number of vertices by Corner Cutting to make the curve smoother.

通过切角增加顶点数量，使曲线更平滑。

Implicit Surfaces

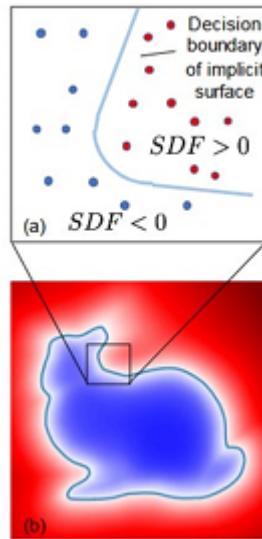
隐式曲面

$$2y(y^2 - 3x^2)(1 - z^2) + (x^2 + y^2)^2 - (9z^2 - 1)(1 - z^2) = 0$$



Define the set of points by an implicit function: 用隐式函数定义点集：

$$f(x, y, z) = 0$$



SDF(Signed Distance Field): given $d(x)$ represents x 's closest distance to the surface.

$$f(x) = \begin{cases} +d(x) & \text{if } x \text{ is inside the shape} \\ -d(x) & \text{else} \end{cases}$$

SDF is more like a field, inside the shape is positive direction while outside is negative direction.

Parametric Surfaces

参数曲面

Use 2 variables to represent surfaces

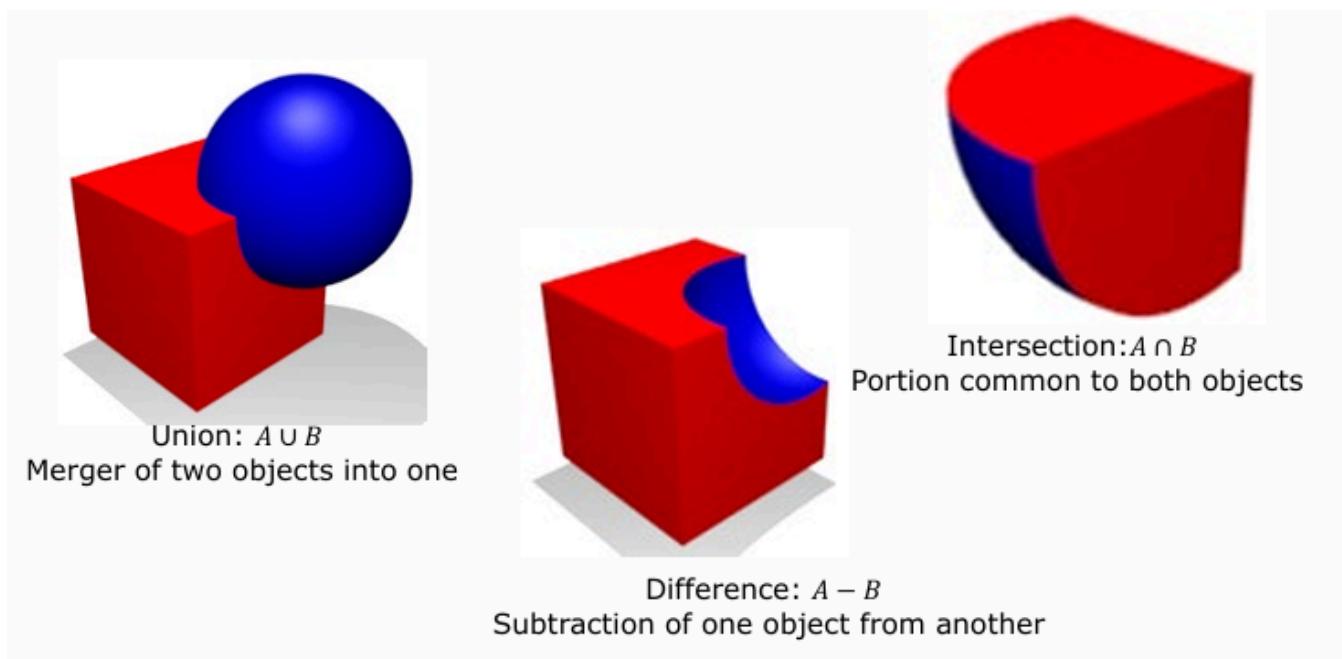
$$\begin{cases} x = f_x(s, t) \\ y = f_y(s, t) \\ z = f_z(s, t) \end{cases} \quad \tilde{P}(s, t) = [x(s, t), y(s, t), z(s, t)]^T$$

Constructive Solid Geometry (CSG)

建构实体几何 (CSG)

By using Boolean operations to combine simpler objects, create a complex solid object.

使用布尔运算将较简单的对象组合起来，创建复杂的实体对象。

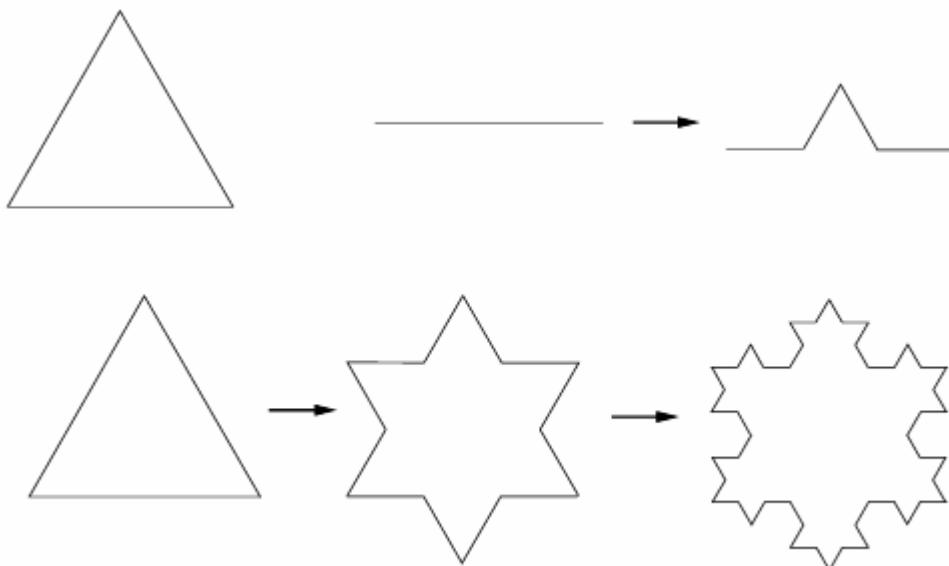


Fractals

分形

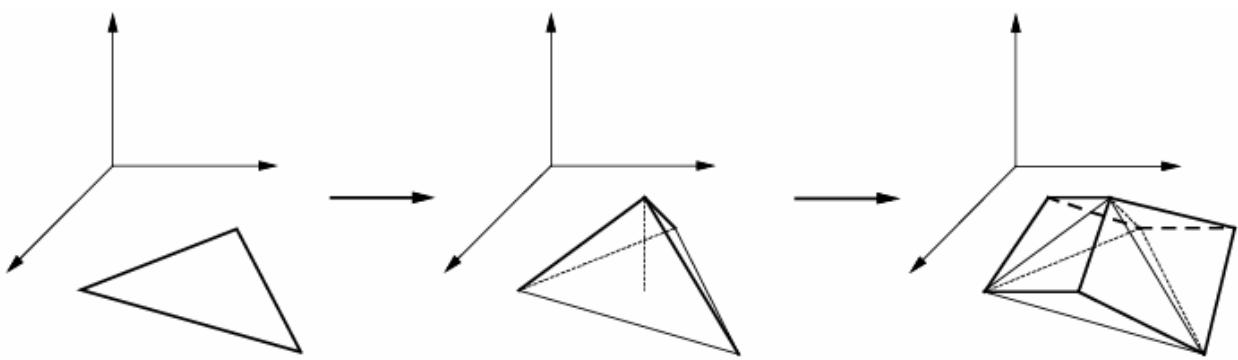
Objects that are self-similar, generated by applying the same transformation function.

自相似对象，通过应用相同的变换函数生成。



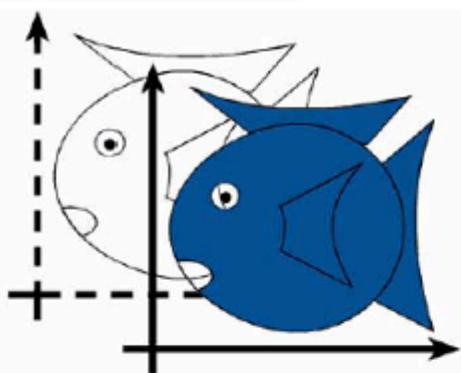
3D fractal objects could be created by adding a new dimension and applying a recursive function.

通过增加一个新维度并应用递归函数，可以创建三维分形物体。

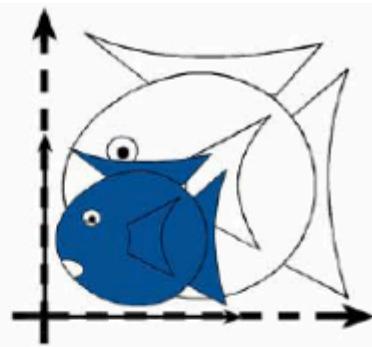


3. Geometric Transformation

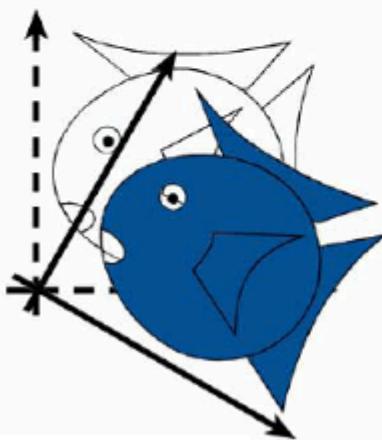
2D Transformation



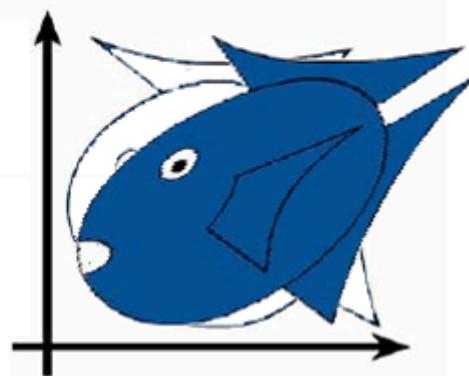
Translation



Uniform Scaling

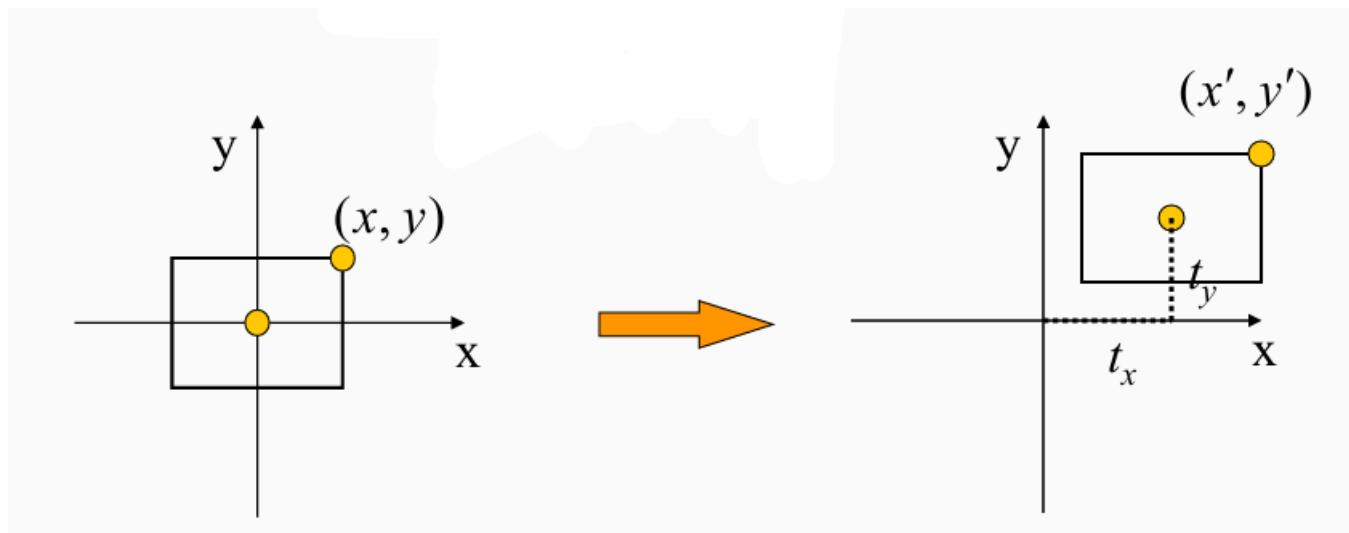


Rotation



Shearing

2D Translation



Move the object along the 2D plane 沿二维平面移动对象

$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

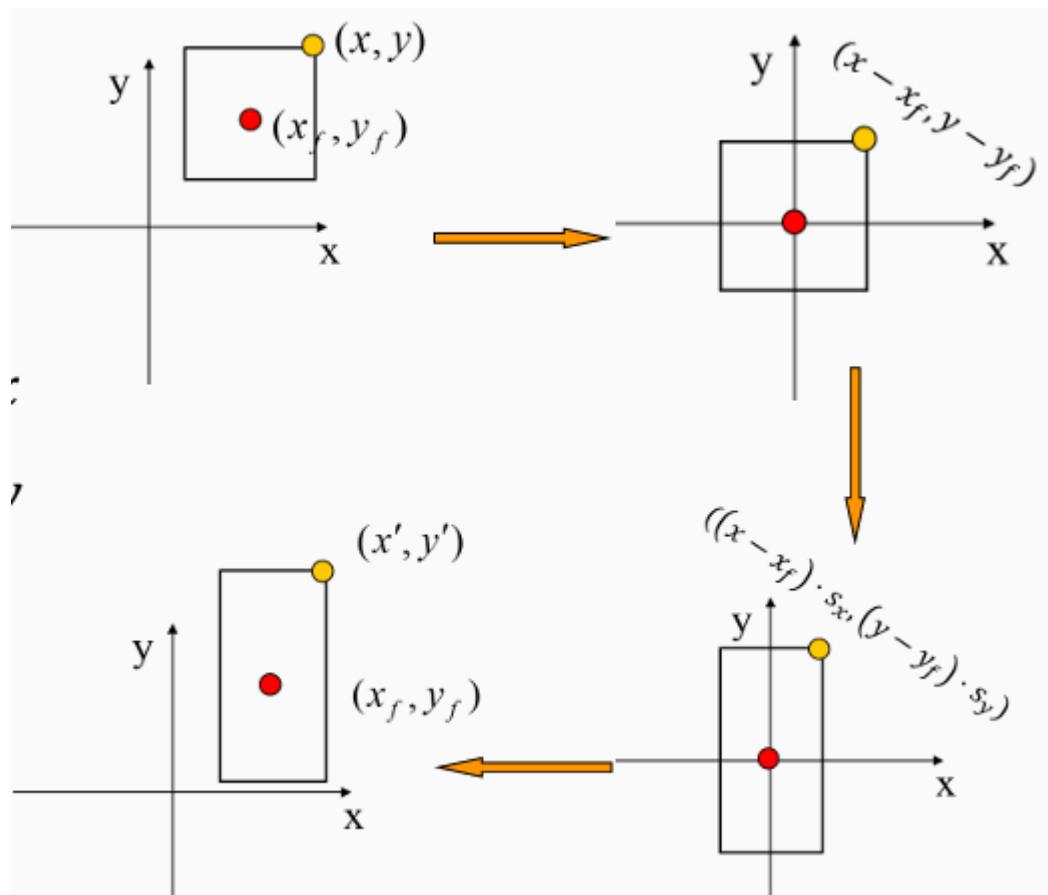
2D Scaling

Change the size of the object

更改对象的大小

Find an arbitrary reference point (x_f, y_f) , then

找到一个任意参考点 (x_f, y_f) , 然后



- Original vertex: (x, y)
- Scaling factors: (s_x, s_y)
- New vertex: (x', y')

$$\begin{cases} x' = x_f + (x - x_f) \cdot s_x \\ y' = y_f + (y - y_f) \cdot s_y \end{cases}$$

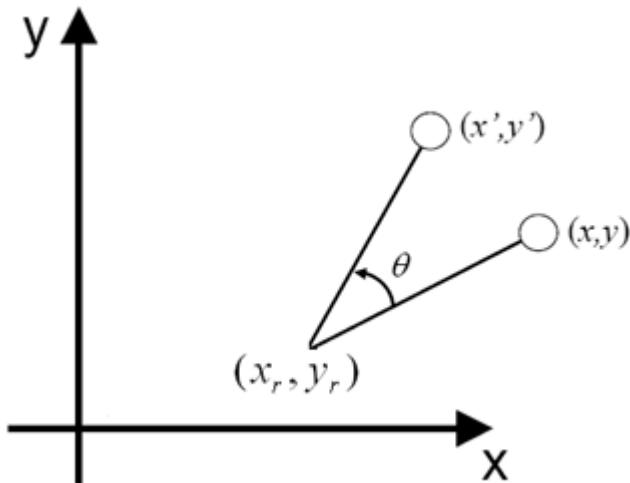
Could be seen as translate to origin, then do original scaling.

可以看成是平移到原点，然后进行原始缩放。

2D Rotation

Rotate the object about a point on the 2D plane

围绕二维平面上的一点旋转对象



- Rotation (pivot) point: (x_r, y_r)
- Rotation angle (counter clockwise): θ ($2\pi - \theta$ for clockwise)
- Original vertex: (x, y)
- New vertex: (x', y')
- 旋转 (支点) 点: (x_r, y_r)
- 旋转角度 (逆时针): θ ($2\pi - \theta$ 顺时针)
- 原始顶点: (x, y)
- 新顶点: (x', y')

1. Translate the object such that the pivot point coincides with the origin;

2. rotate the object around the origin (the pivot point)

3. undo the translation

4. 平移对象，使支点与原点重合；

5. 围绕原点 (支点) 旋转对象

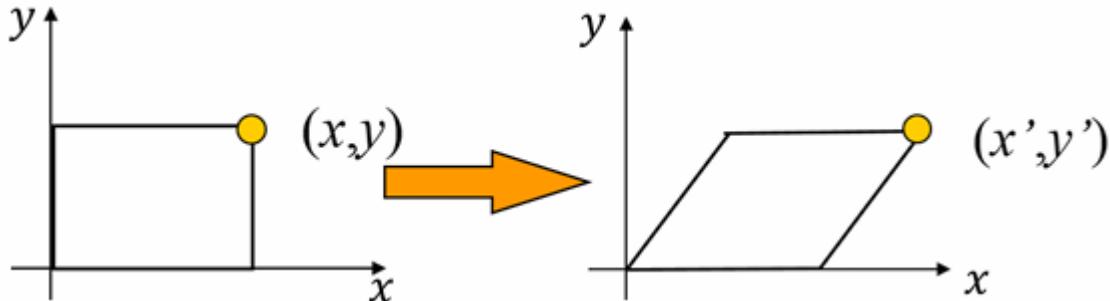
6. 撤销平移

$$\begin{cases} x' = x_r + (x - x_r)\cos(\theta) - (y - y_r)\sin(\theta) \\ y' = y_r + (x - x_r)\sin(\theta) + (y - y_r)\cos(\theta) \end{cases}$$

2D Shearing

Keep all points on one axis in place, and move the other points parallel to the axis, with the distance to the axis depending on how far away they are from it.

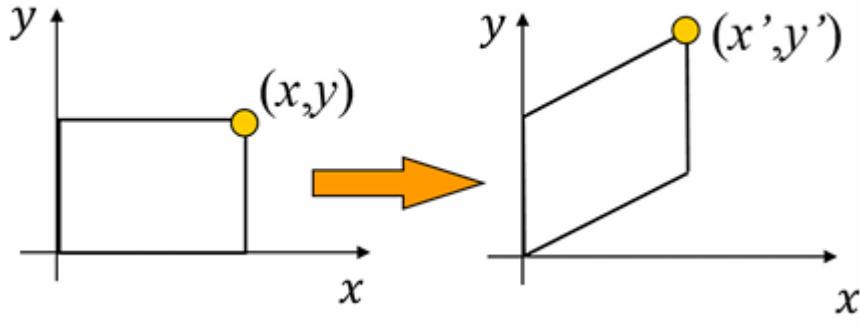
将一个坐标轴上的所有点保持在原位，然后平行于坐标轴移动其他点，与坐标轴的距离取决于这些点与坐标轴的距离。



horizontal shearing (x-shearing):

$$\begin{cases} x' = x + \lambda_x y \\ y' = y \end{cases}$$

vertical shearing (y-shearing)



$$\begin{cases} x' = x \\ y' = y + \lambda_y x \end{cases}$$

Homogeneous Coordinates

A 2D coordinate (x, y) can be written as (x_h, y_h, h) , so:

$$\begin{cases} x = \frac{x_h}{h} \\ y = \frac{y_h}{h} \end{cases}$$

(x_h, y_h, h) is called a homogeneous coordinate.

Usually $h = 1$, (x, y) could be represented as $(x, y, 1)$

(x_h, y_h, h) 称为齐次坐标。

通常 $h = 1$, (x, y) 可表示为 $(x, y, 1)$

By using homogeneous coordinate, translation, scaling, rotation and shearing could all be expressed by the product of the transformation matrices.

通过使用齐次坐标，平移、缩放、旋转和剪切都可以用变换矩阵的乘积来表示。

2D Translation Matrix

$$\begin{aligned}x' &= x + t_x \\y' &= y + t_y\end{aligned}\quad \text{→} \quad \begin{bmatrix}x' \\ y' \\ 1\end{bmatrix} = \begin{bmatrix}1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}x \\ y \\ 1\end{bmatrix}$$

$$P = \begin{bmatrix}x \\ y \\ 1\end{bmatrix}, T(t_x, t_y) = \begin{bmatrix}1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1\end{bmatrix}, P' = \begin{bmatrix}x' \\ y' \\ 1\end{bmatrix}$$

$$P' = T(t_x, t_y)P$$

2D Scaling Matrix

$$\begin{aligned}\begin{cases}x' = x_f + (x - x_f) \cdot s_x \\ y' = y_f + (y - y_f) \cdot s_y\end{cases} \\ P = \begin{bmatrix}x \\ y \\ 1\end{bmatrix}, T(t_x, t_y) = \begin{bmatrix}1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1\end{bmatrix}, S(s_x, s_y) = \begin{bmatrix}s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1\end{bmatrix}, P' = \begin{bmatrix}x' \\ y' \\ 1\end{bmatrix} \\ P' = T(x_f, y_f)S(s_x, s_y)T(-x_f, -y_f)P\end{aligned}$$

2D Rotation Matrix

$$\begin{aligned}\begin{cases}x' = x_r + (x - x_r)\cos(\theta) - (y - y_r)\sin(\theta) \\ y' = y_r + (x - x_r)\sin(\theta) + (y - y_r)\cos(\theta)\end{cases} \\ P = \begin{bmatrix}x \\ y \\ 1\end{bmatrix}, T(t_x, t_y) = \begin{bmatrix}1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1\end{bmatrix}, R(\theta) = \begin{bmatrix}\cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1\end{bmatrix}, P' = \begin{bmatrix}x' \\ y' \\ 1\end{bmatrix} \\ P' = T(x_f, y_f)R(\theta)T(-x_f, -y_f)P\end{aligned}$$

2D Shearing Matrix

Horizontal shearing (x-shearing):

$$\begin{aligned}\begin{cases}x' = x + \lambda_x y \\ y' = y\end{cases} \\ \begin{bmatrix}x' \\ y' \\ 1\end{bmatrix} = \begin{bmatrix}1 & \lambda_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}x \\ y \\ 1\end{bmatrix}\end{aligned}$$

Vertical shearing (y-shearing):

$$\begin{cases} x' = x \\ y' = y + \lambda_y x \end{cases}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \lambda_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Order of 2D Transformations

The order of transformations **does** matter 变换的顺序十分重要

Matrix multiplication is not commutative:

矩阵乘法不是交换式的：

$$AB \neq BA$$

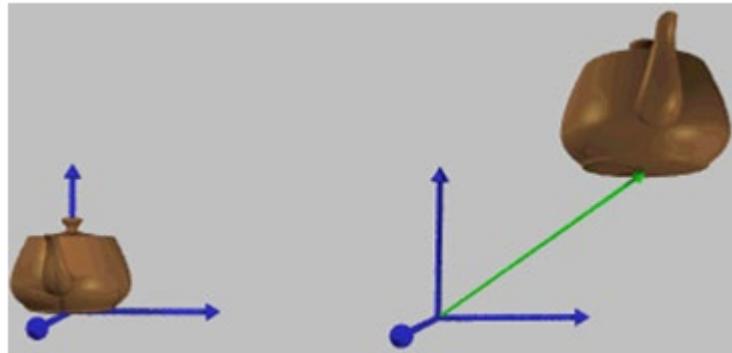
e.g. Rotate then translate is not equal to translate then rotate

例如，先旋转后平移不等于先平移后旋转

3D Transformation

The coordinate of a point in 3D space is denoted as (x, y, z) , and its homogeneous coordinate is represented in $(x, y, z, 1)$

3D Translation



$$\begin{cases} x' = x + t_x \\ y' = y + t_y \\ z' = z + t_z \end{cases}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, T(t_x, t_y) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, P' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

$$P' = T(t_x, t_y, t_z) \cdot P$$

3D Scaling

Also need to first translate to origin.

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} T(t_x, t_y) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, P' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

$$P' = T(x_f, y_f, z_f) S(s_x, s_y, s_z) T(-x_f, -y_f, -z_f) \cdot P$$

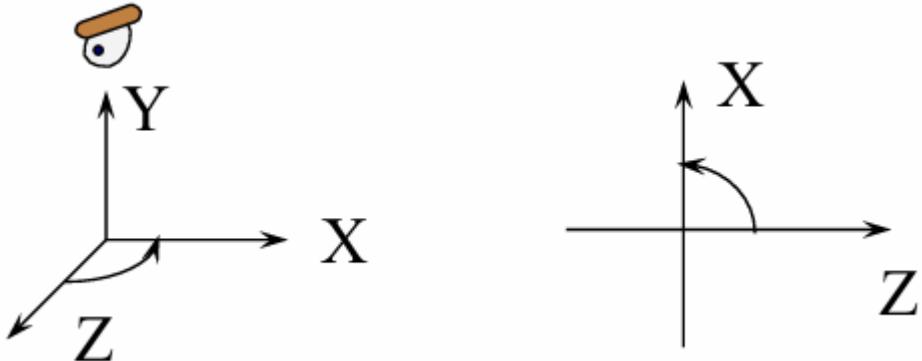
3D Rotation

Rotation around x- axis:



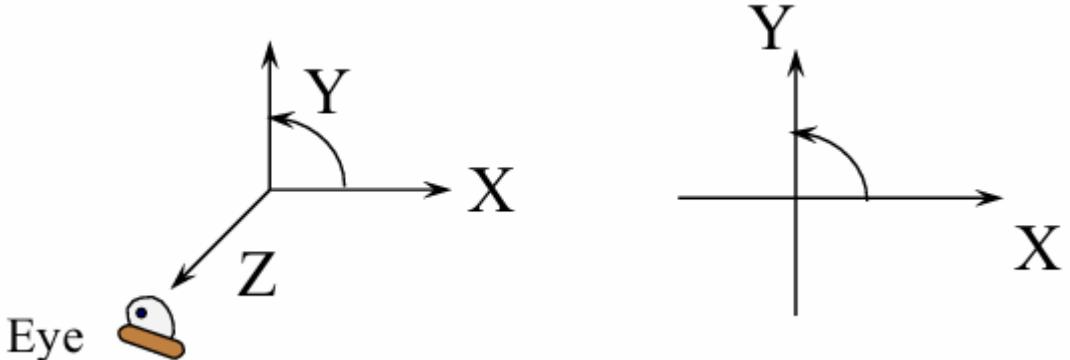
$$R(\theta, 1, 0, 0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation around y- axis(Special!!!):



$$R(\theta, 0, 1, 0) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation around z-axis



$$R(\theta, 0, 0, 1) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

An arbitrary rotation about the origin can be decomposed into three successive rotations about the three axes.

绕原点的任意旋转可以分解为绕三个轴的连续旋转。

But the order **matters!**

但顺序很重要！

$$\mathbf{R}_x(\alpha)\mathbf{R}_y(\beta)\mathbf{R}_z(\gamma)\mathbf{P} \neq \mathbf{R}_x(\alpha)\mathbf{R}_z(\gamma)\mathbf{R}_y(\beta)\mathbf{P}$$

↓ ↓ ↓ ↓ ↓ ↓ ↓
 3rd 2nd 1st 3rd 2nd 1st

Exercise Questions

1.

Q: A 3D object is rotated by 90 degrees about an axis passing from $(0, 1, 1)$ to $(2, 1, 1)$. Write out the transformation matrix.

问：一个3D物体绕着从 $(0, 1, 1)$ 到 $(2, 1, 1)$ 的轴旋转90度。写出变换矩阵。

A: The axis passing from $(0, 1, 1)$ to $(2, 1, 1)$, so it's parallel to x-axis. We first need to translate the start point of axis to origin, so a translation matrix first:

A: 轴从 $(0, 1, 1)$ 到 $(2, 1, 1)$ ，所以它平行于x轴。我们首先需要将轴的起始点平移到原点，所以首先是平移矩阵：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then, considering it's x-axis rotation, we could write the rotation matrix:

然后，考虑它是x轴旋转，我们可以写出旋转矩阵：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate 90 degrees, write the rotation matrix as:

旋转90度，将旋转矩阵写成：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, undo the translation:

最后，撤销平移：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So we have the final answer:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.

Q: Describe what transformation the matrix M performs when applied to a 3D object:

Q：描述矩阵M应用于3D对象时执行的变换：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A:

First, look at the last column, we could see there's no translation.

首先，看最后一列，我们可以看到没有平移。

Then look at the left 3×3 matrix, its characteristic is quite similar to x-axis rotation:

再看左边的 3×3 矩阵，它的特征与x轴旋转非常相似：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By comparing we could know $\cos\theta = 0, \theta = 90^\circ$, so the rotation matrix is :

通过比较，我们知道 $\cos\theta = 0, \theta = 90^\circ$ ，因此旋转矩阵为：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It's obvious that the value is not correspond to the transformation matrix, guessing there's scaling:

很明显，这个值与变换矩阵并不对应，可能是缩放了：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So the final answer is :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It rotates 90 degrees around x-axis, then scales with factors: $s_x = 1, s_y = 2, s_z = 3$

4. Hidden Surface Removal

Hidden Surface Removal

An algorithm used to determine which surface of the object is visible from a given view.

一种算法，用于确定从给定视角可以看到物体的哪个表面。

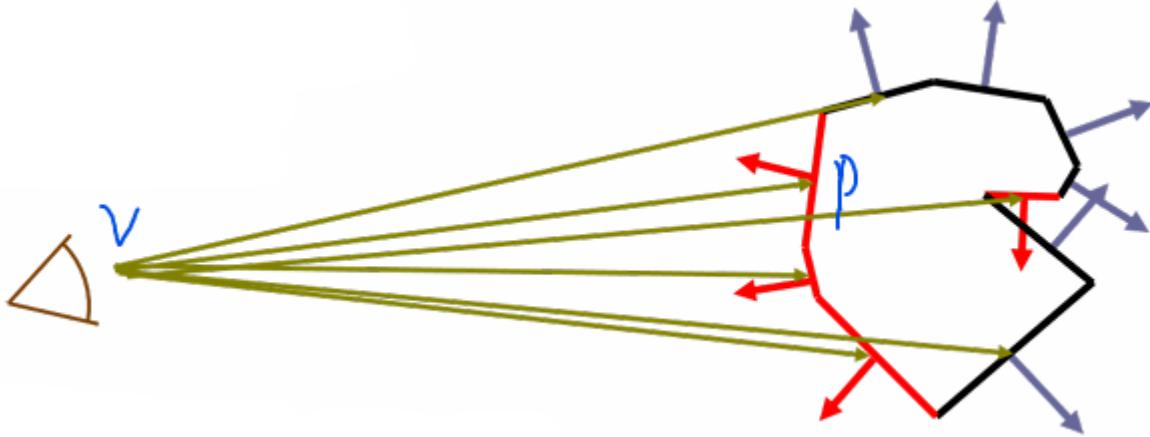
2 types of algorithms:

- Object space algorithms: Deal with objects visibility relationship before converting into pixels, in 3D form.
e.g. Back-face culling algorithm
- 对象空间算法 在转换为三维形式的像素之前，先处理物体的可见度关系，例如：背面剔除算法
- Image space algorithms: Deal with objects visibility relationship after converting into pixels, in 2D form.
e.g. Depth-sorting (or Painter's) algorithm, Z (or Depth)-buffering algorithm.
- 图像空间算法 如深度排序算法（或画家算法）、Z（或深度）缓冲算法。

Back-face culling algorithm

Deal with polygons, hide the ones which are facing away from viewpoint.

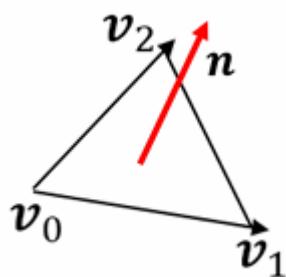
处理多边形，隐藏远离视角的多边形。



For each face f , compute the vector from viewpoint v to any point p on f :

对于每个面f计算从视点 v 到 f 上任意点 p 的矢量。

- Visible if $n \cdot (p - v) < 0$. i.e. The angle between the sight line vector and the normal to the surface is greater than 90° .
- 如果 $n \cdot (p - v) < 0$ 即可见，视线矢量与表面法线之间的夹角大于 90° 。
- Invisible if $n \cdot (p - v) > 0$. i.e. The angle between the sight line vector and the normal to the surface is less than 90° .
- 如果 $n \cdot (p - v) > 0$ 即不可见，视线矢量与表面法线之间的夹角小于 90° 。



Calculate the normal of a plane: 右手螺旋定则

$$n = (v_1 - v_0) \times (v_2 - v_0)$$

Advantage:

- easy to implement and highly efficient
- 易于实现，效率高
- The complexity is only $O(n)$
- 复杂度仅为 $O(n)$

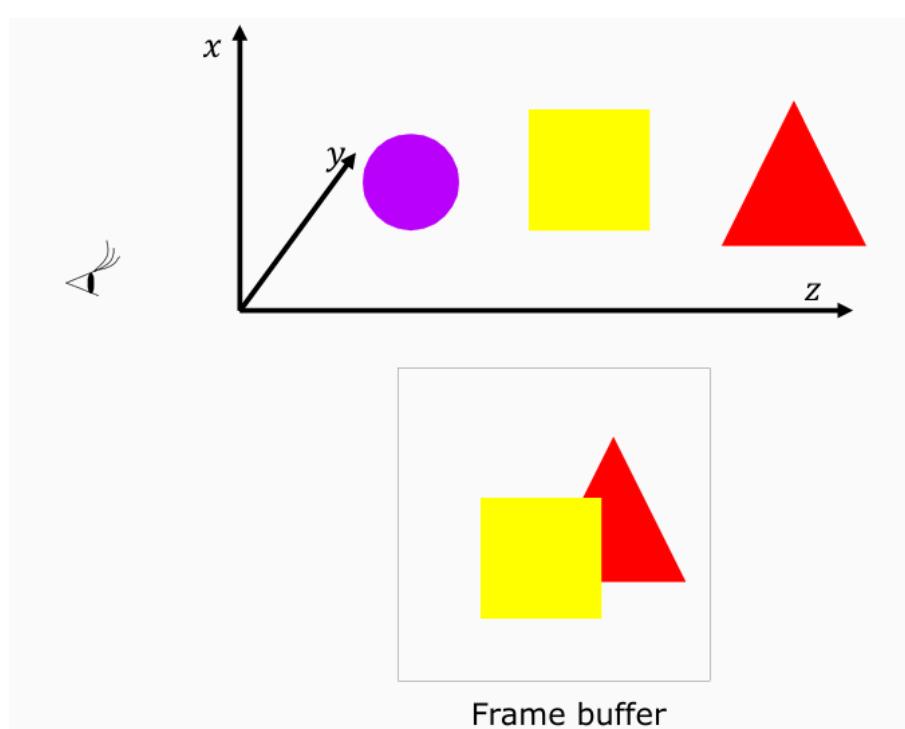
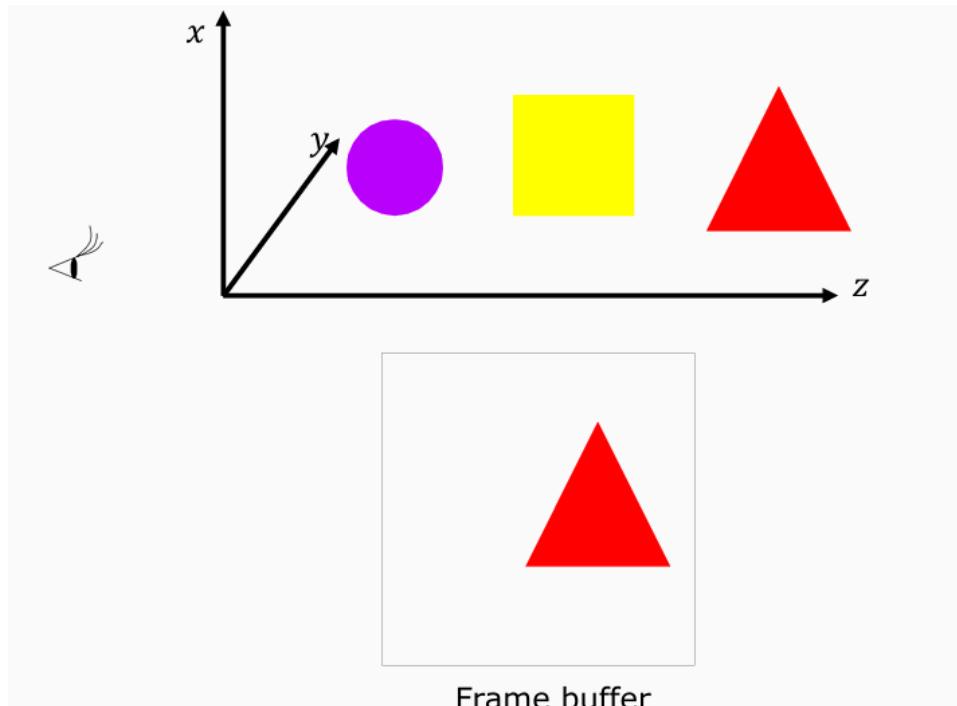
Disadvantage:

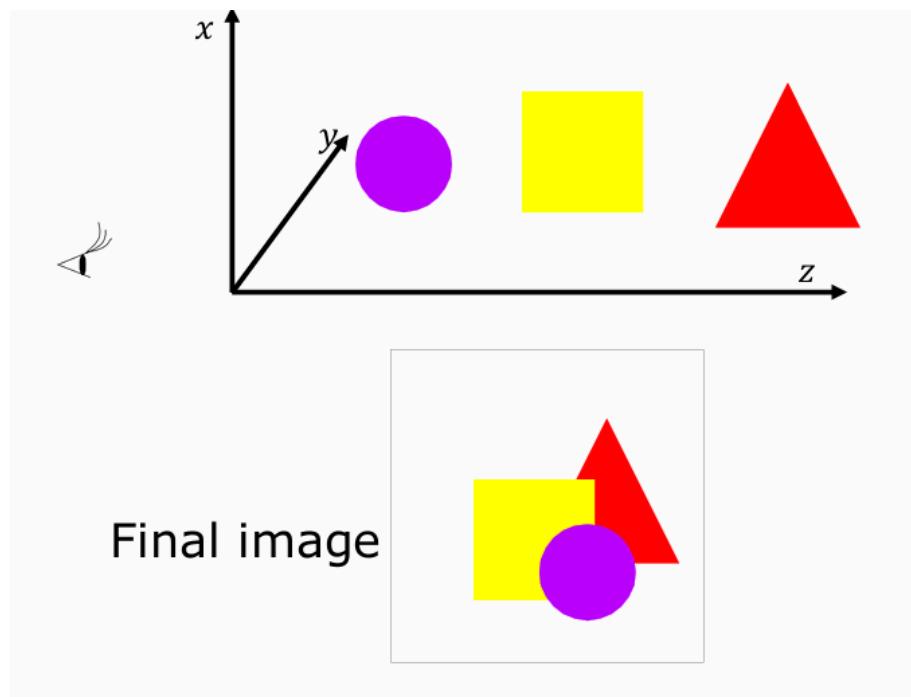
- only feasible for convex polygon, fail for concave polygon
- 只适用于凸多边形，对凹多边形无效

Depth-sorting (or Painter's) algorithm

Renders the surfaces of objects in order from far to close, making sure that near objects cover far objects.

按从远到近的顺序渲染物体的表面，确保近的物体覆盖远的物体。



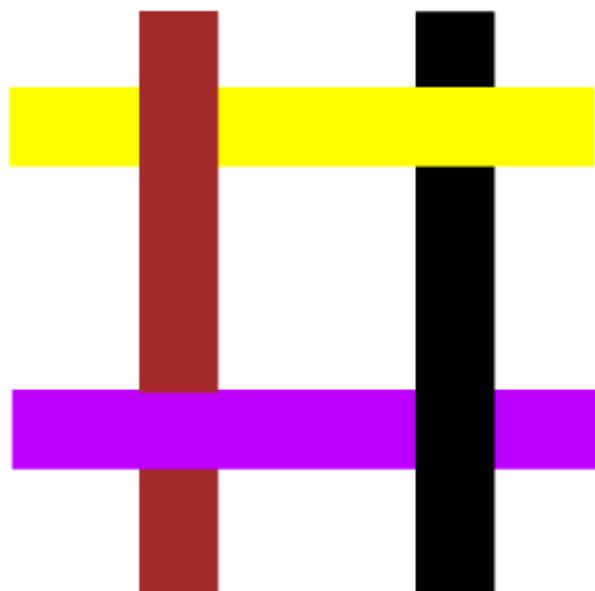


Steps:

1. Sort all surfaces according to their distances from the view point. 根据表面与视点的距离对所有表面进行排序。
2. Render the surfaces to the frame buffer one at a time starting from the farthest surface. 从最远的曲面开始，逐个将曲面渲染到帧缓冲区。
3. A surface drawn later will replace overlapping surfaces drawn earlier. 之后绘制的曲面将取代之前绘制的重叠曲面。
4. After all the surfaces have been processed, the frame buffer stores the final image 处理完所有曲面后，帧缓冲区将存储最终图像

This method is simple but as the number of objects increases, it becomes very complex and time consuming.

这个方法很简单，但随着对象数量的增加，它变得非常复杂和耗时。



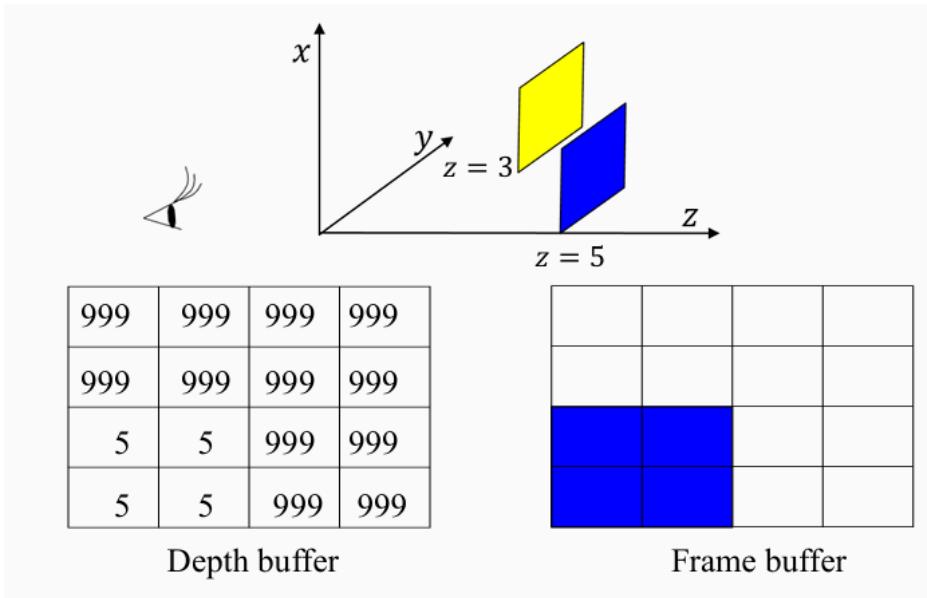
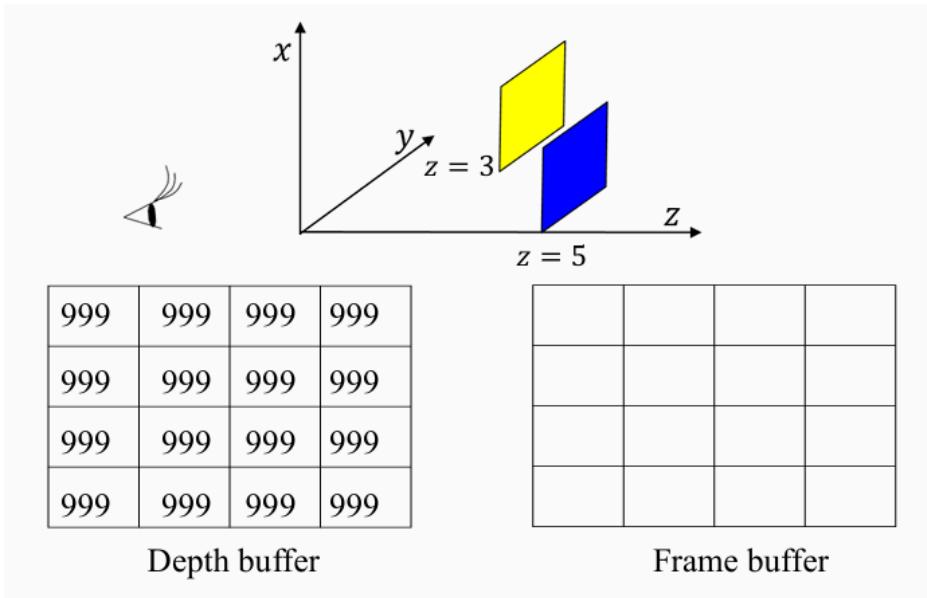
And it can't process intersecting polygons or cyclic overlap.

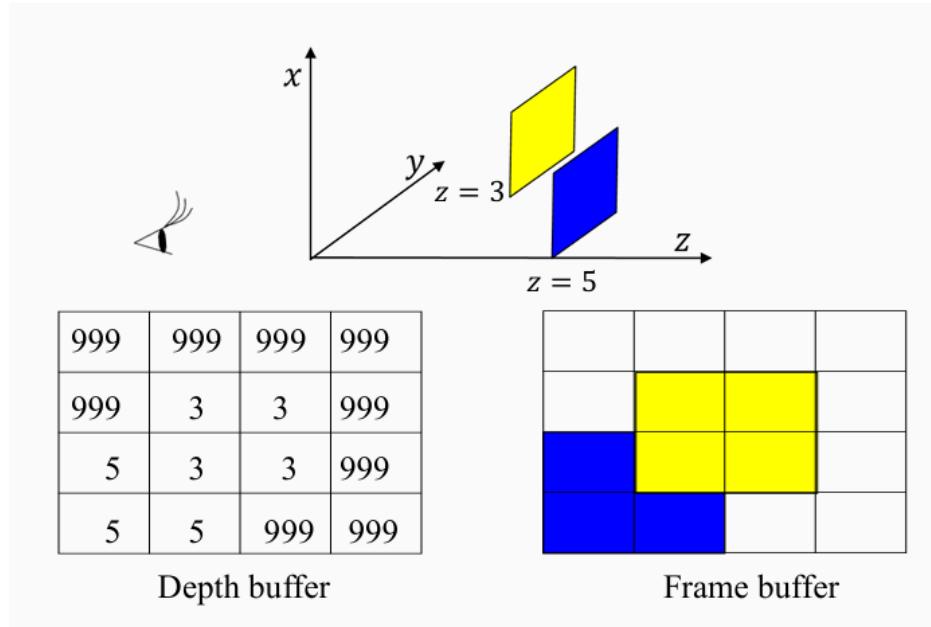
它不能处理相交多边形或循环重叠。

Z (or Depth)-buffering algorithm

Use 2 buffers to determine what color to render: depth buffer(store depth information) and frame buffer(store color value).

使用2个缓冲区来确定渲染什么颜色：深度缓冲区（存储深度信息）和帧缓冲区（存储颜色值）。





Steps:

1. Initialize 2 buffers. Each pixel of depth buffer is set to maximum value(999), each pixel of frame buffer is set to background color. 初始话2个缓冲区。将深度缓冲区的每个像素设置为最大值（999），将帧缓冲区的每个像素设置为背景颜色。
2. Randomly render one surface (ordering doesn't matter) 随机渲染一个表面（顺序不重要）
3. For current surface, calculate each of its pixel's depth value from viewing point. 对于当前表面，从视点计算其每个像素的深度值。
4. If current pixel depth value < the corresponding value in depth buffer (i.e. closer), update depth buffer and frame buffer with current pixel(depth value and color value.) 如果当前像素深度值<深度缓存中的对应值（即更接近），则用当前像素（深度值和颜色值）更新深度缓存和帧缓存。

When there's only a few objects, this method has lower efficiency than depth-sorting algorithm.

当对象数量较少时，该方法的效率低于深度排序算法。

However as the number of objects increases, this method becomes more attractive.

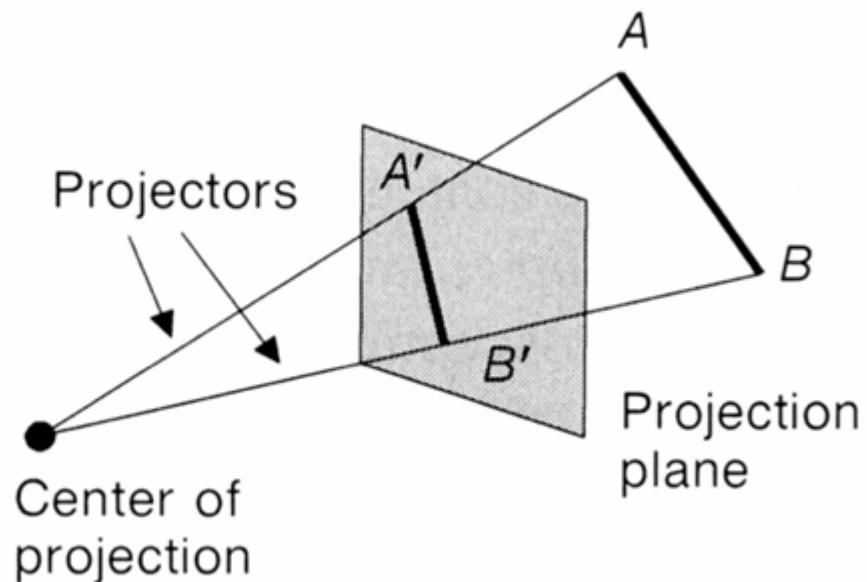
然而，随着对象数量的增加，这种方法变得更有吸引力。

Majority of graphics accelerators available in the market are based on the z-buffer method.

市面上的大多数图形加速器都是基于z-buffer方法。

5. Projection and Clipping

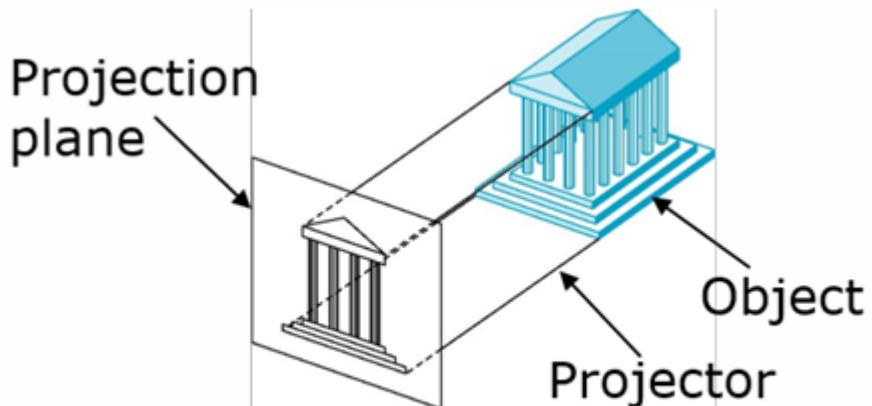
Projection



Definition: 3D objects are projected from the center of projection (COP) onto a 2D plane by means of straight projectors to form a projected image.

定义: 三维物体通过直线投影仪从投影中心 (COP) 投射到二维平面上, 形成投影图像。

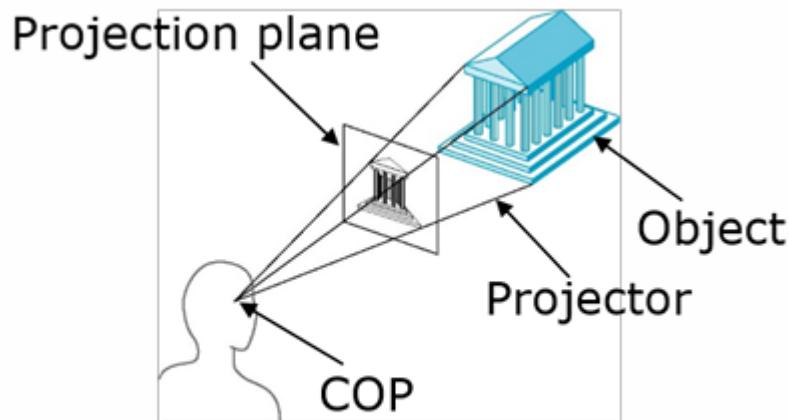
Parallel vs. Perspective Projection



Parallel projection

Parallel projection:

- The COP is at infinity COP 位于无限远处
- All projectors are parallel. 所有投影线都是平行的。
- Parallelism is preserved. 平行性得以保留。

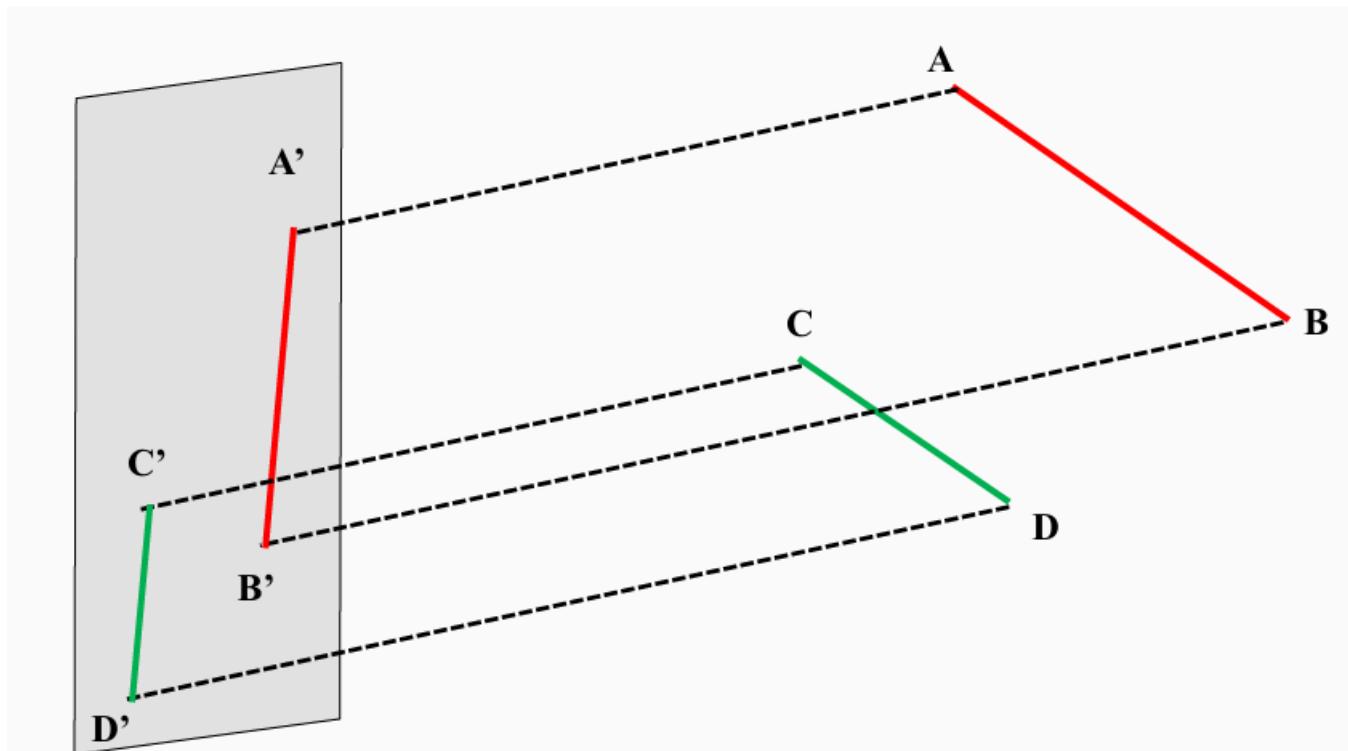


Perspective projection

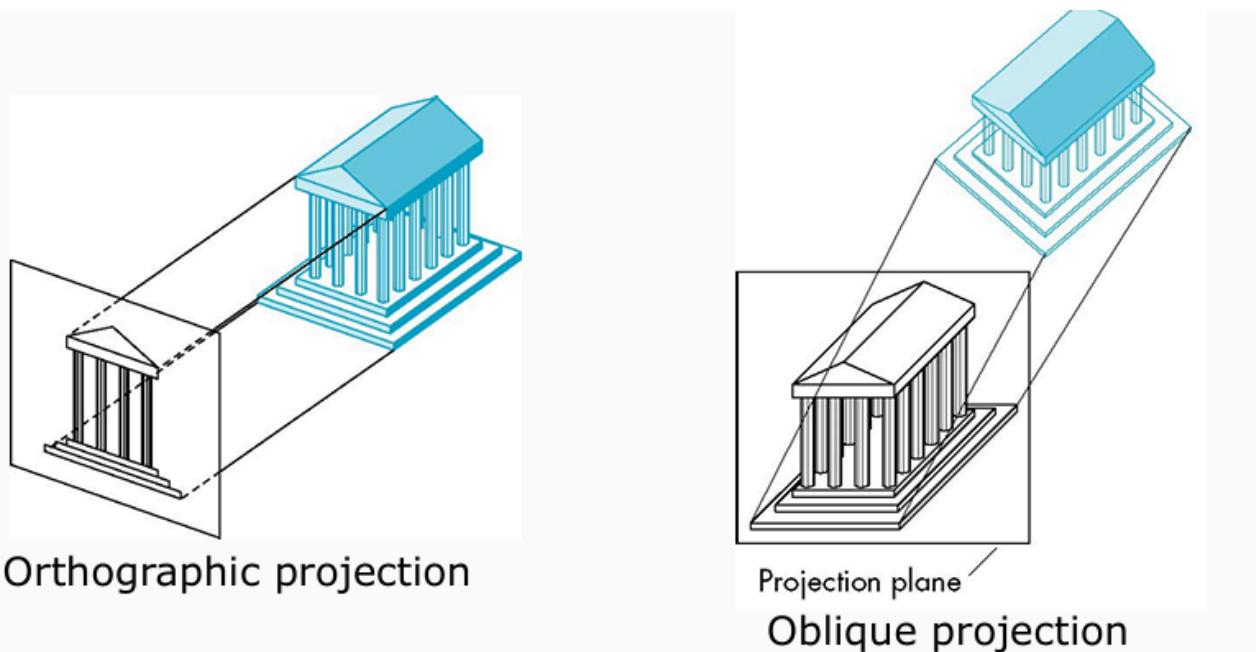
Perspective projection:

- The COP is at a finite distance from the projection plane. COP 与投影面的距离是有限的。
- All projectors meet at COP 所有投影汇聚到COP上
- Parallelism is not preserved in general. 平行性一般不会保留。

Parallel Projection



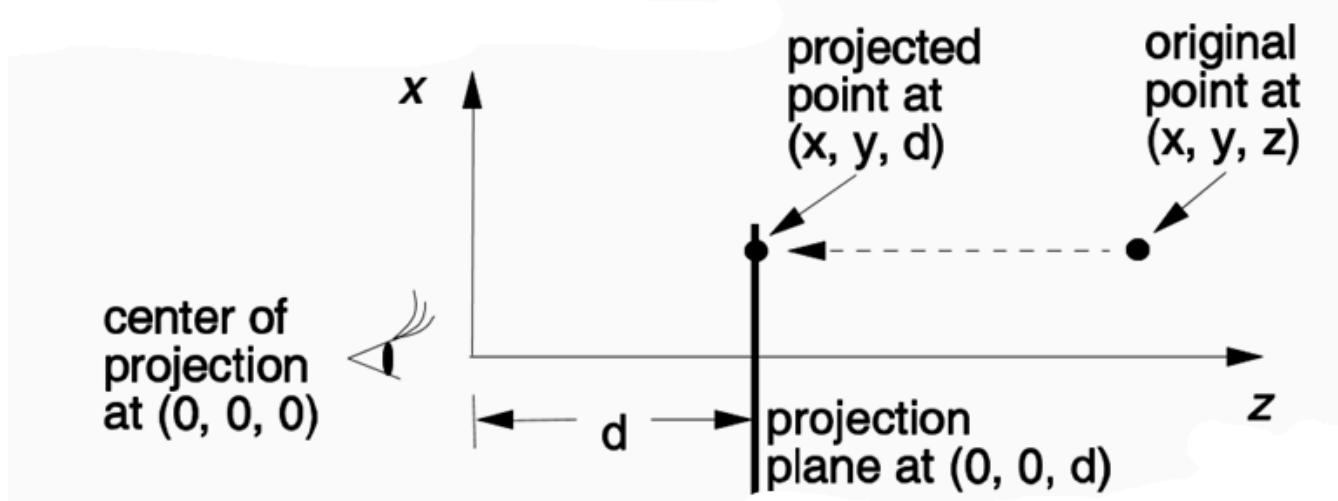
Parallel projection preserves the parallelism, i.e., parallel lines remain parallel under parallel projection
平行投影保持平行性，即平行投影下的平行线保持平行



2 types of parallel projection:

- Orthographic projection: projectors are perpendicular to projection plane. 正投影：投影线垂直于投影面。
- Oblique projection: projectors are not perpendicular, but still parallel. 斜投影：投影线不垂直，但仍平行。

Parallel Projection matrix:



Assume the viewer is at origin (0,0,0), the projection plane is at (0,0,d) and parallel to XY-plane. All objects' z-ordinate will be fixed to d.

假设观察者在原点 (0,0,0) 处，投影平面在 (0,0,d) 处平行于xy平面。所有对象的z轴坐标将固定为d。

$$\begin{cases} x' = x \\ y' = y \\ z' = d \end{cases}, \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Perspective Projection

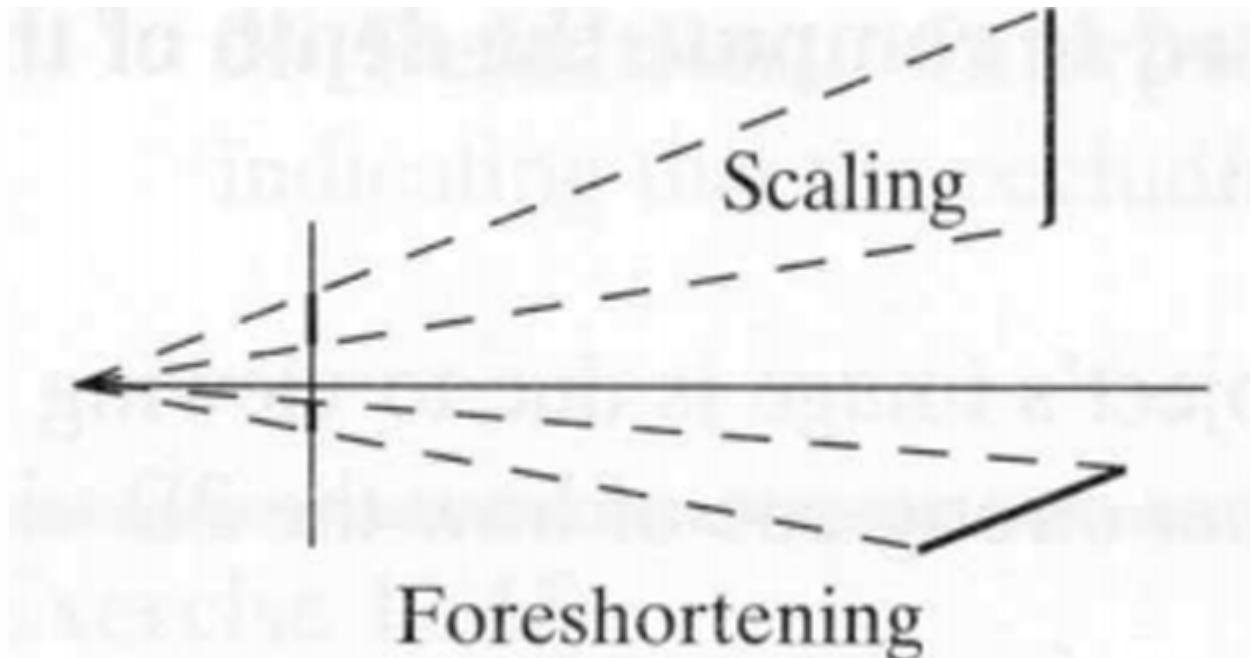


The parallel projection doesn't give us depth information while perspective projection does. In perspective projection, the closer object is from the projection plane, the smaller.

平行投影并不能提供深度信息，而透视投影却能。在透视投影中，物体距离投影面越近，就越小。

Parallelism is not preserved in general! Parallel lines could meet on the projection plane.

平行性一般不会保留！平行线可能在投影面上相遇。



If a line is parallel to the projection plane, it's like scaling. Otherwise it's foreshortening.

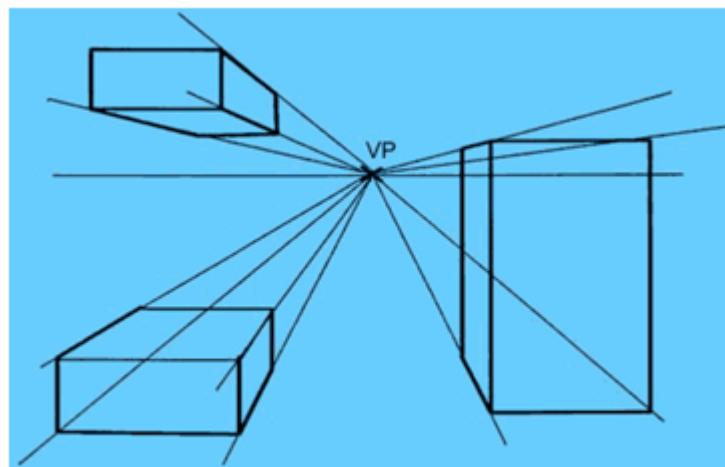
如果一条线与投影面平行，就像缩放一样。否则就是透视收缩。

Vanishing Points: Any set of parallel lines that are not parallel to the projection plane will converge to a vanishing point.

消失点：任何一组与投影面不平行的平行线都会汇聚到一个消失点。

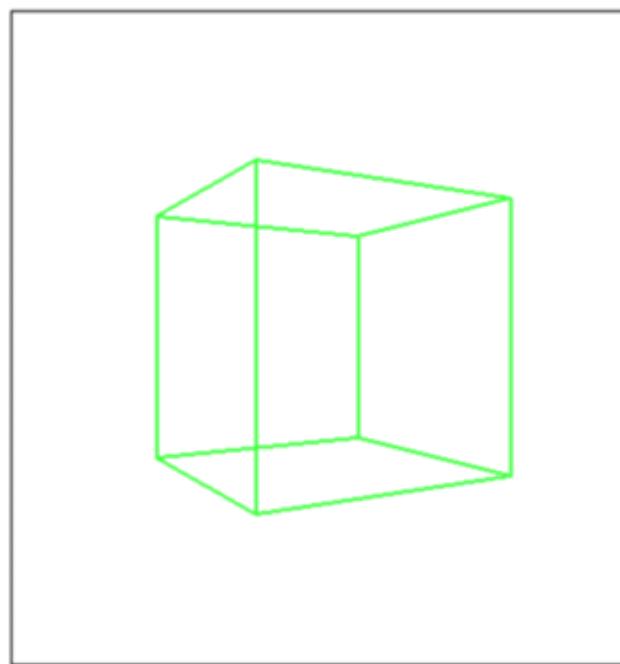
One-Point Perspective Projection: The projection plane is parallel to two principal axes (a plane), 1 vanishing point.

一点透视投影：投影平面平行于两个主轴（一个平面），1个消失点。



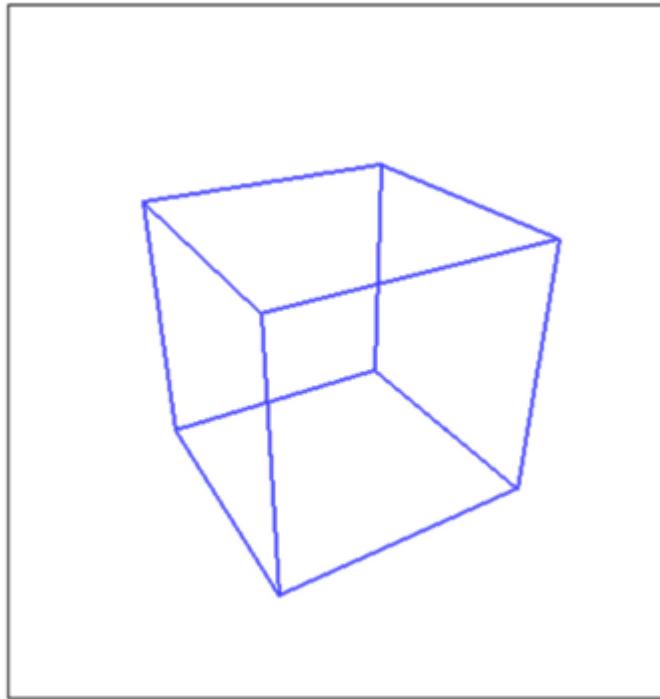
Two-Point Perspective Projection: The projection plane is parallel to one principal axes, 2 vanishing points.

两点透视投影：投影平面平行于一个主轴，两个消失点。



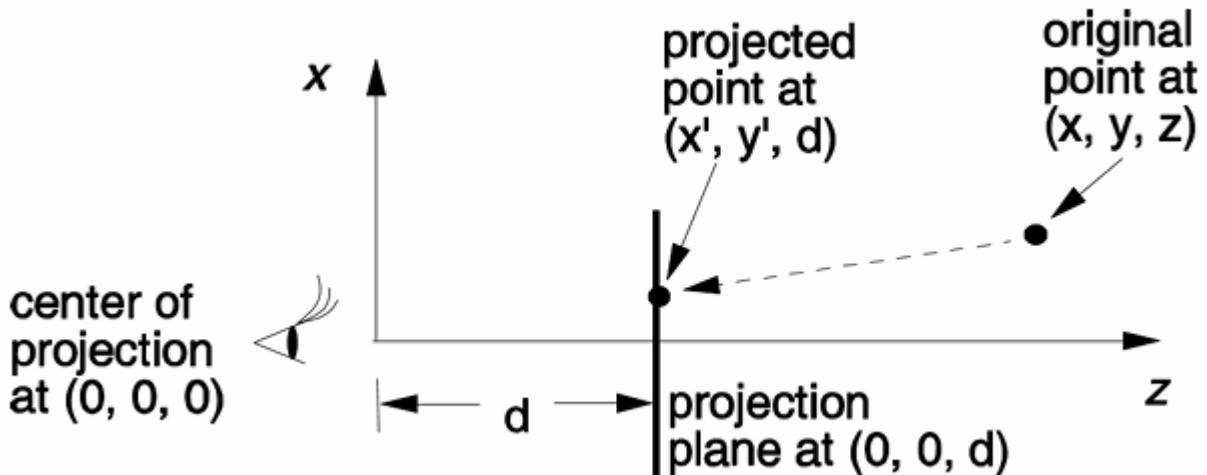
Three-Point Perspective Projection: The projection plane is NOT parallel to any of the principal axes. 3 vanishing points correspond to one of three axis.

三点透视投影: 投影平面不平行于任何主轴。三个消失点对应于三个轴中的一个。



Perspective projection matrix: Assume the viewer is at origin (0,0,0), the projection plane is at (0,0,d) and parallel to XY-plane. All objects' z-ordinate will be fixed to d, considering Similar Triangle Relationship:

透视投影矩阵: 假设观察者在原点 (0,0,0)，投影平面在 (0,0,d) 并平行于xy平面。考虑到相似的三角形关系，所有对象的z坐标都固定为d：



$$\begin{cases} x' = \frac{d}{z} x \\ y' = \frac{d}{z} y \\ z' = d \end{cases}$$

Since perspective projection isn't linear transformation, we could import **Homogeneous Coordinates**:

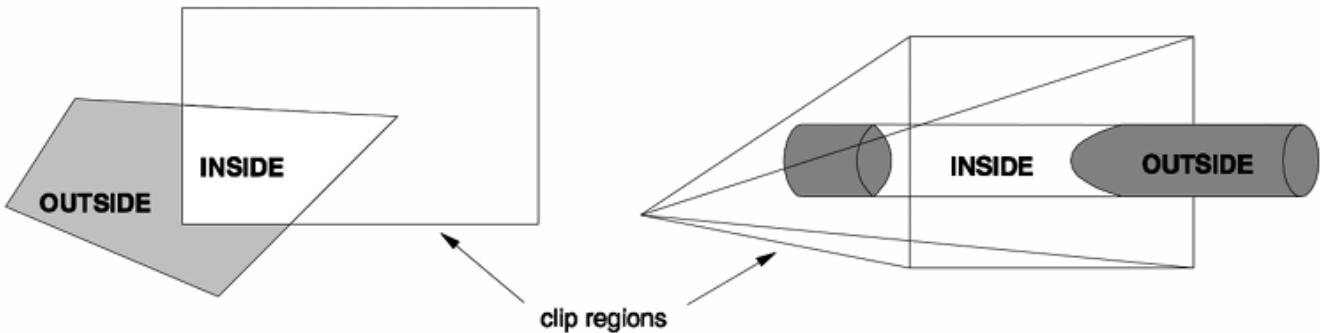
由于透视投影不是线性变换，我们可以导入齐次坐标：

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \frac{d}{z} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The 4th row's $\frac{1}{d}$ is essential for perspective projection.

第四行 $\frac{1}{d}$ 对于透视投影至关重要。

Clipping



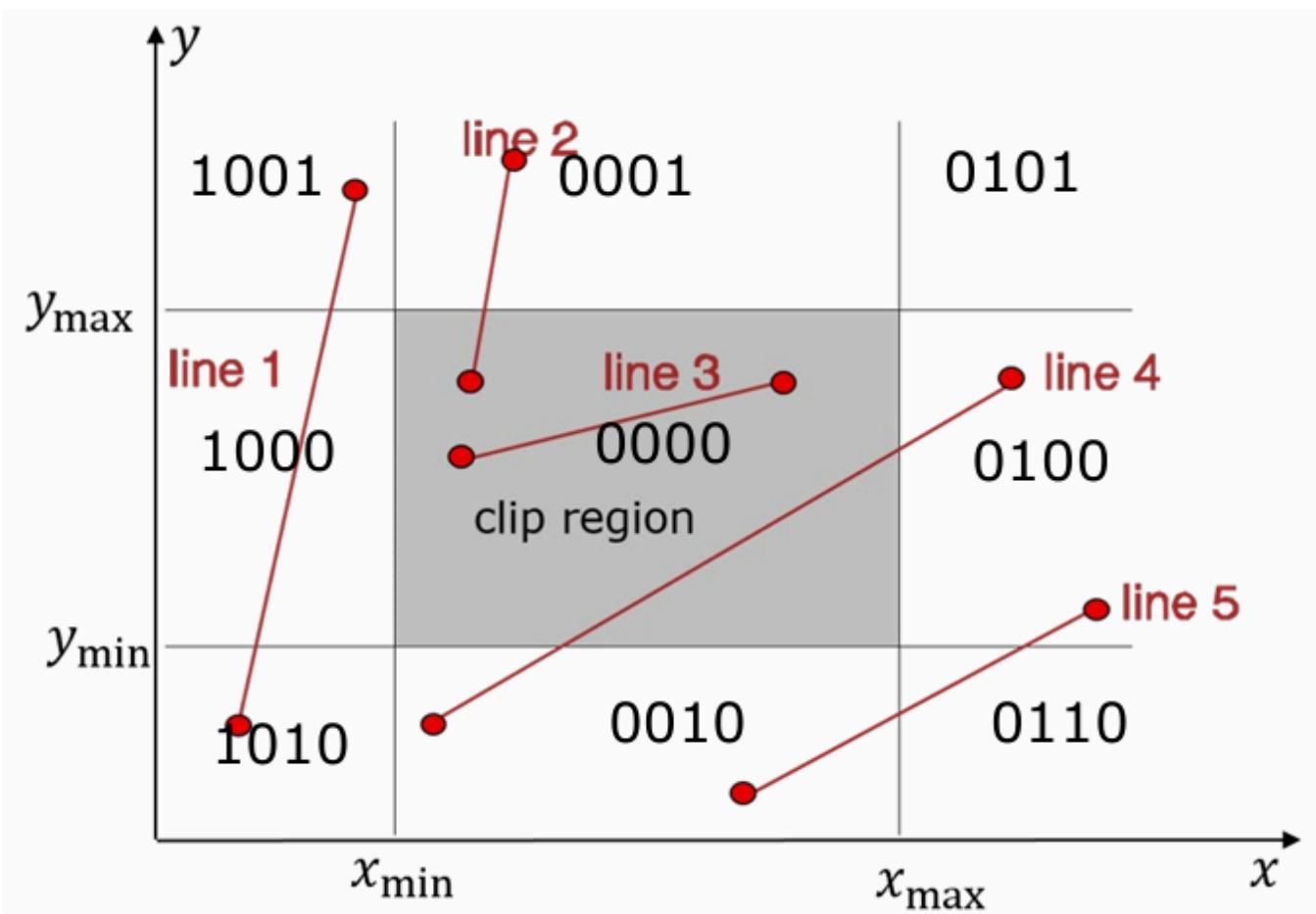
Clipping is a process to determine what portion of an object is inside or outside. We only want inside part to save computation time.

剪切是一个确定物体内部或外部的过程。我们只需要内部部分，以节省计算时间。

The Cohen-Sutherland Line-Clipping Algorithm

This algorithm aims at quickly identifying whether a line needed to be clipped or not.

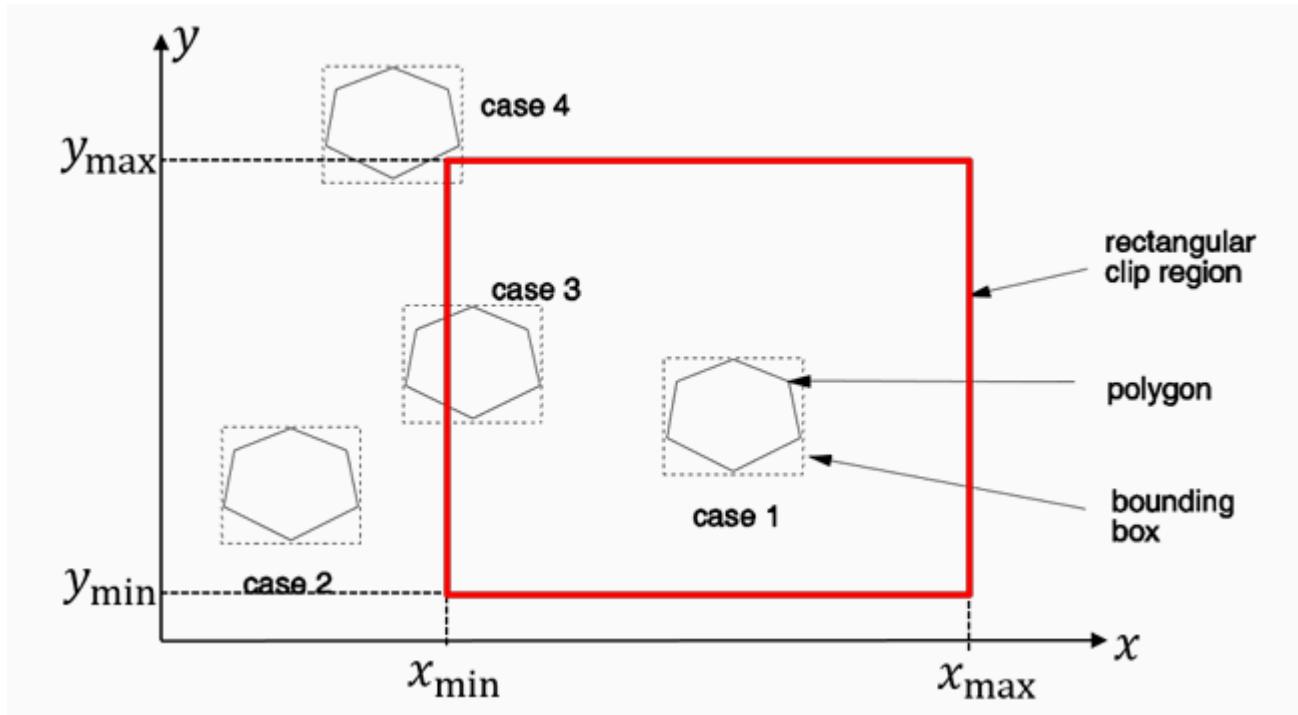
这种算法旨在快速识别是否需要剪切一条线。



- Assign a value to a point based on whether that point is inside the clip region or not.
- 根据点是否在剪辑区域内为该点赋值。
 - 1st bit set to 1 if $x < x_{min}$
 - 2nd bit set to 1 if $x > x_{max}$
 - 3rd bit set to 1 if $y < y_{min}$
 - 4th bit set to 1 if $y > y_{max}$
- Accept: both start point and end point have the same code "0000", both points are in the region. The logical **OR** is zero(line 3)
- 接受：起点和终点都有相同的代码 "0000"，两点都在区域内。逻辑 OR 为零（第3条线）
- Reject: both start point and end point have the same bit set to "1", then the line is complete out of the region. The logical AND is not zero(line 1).
- 拒绝：起点和终点都有相同的位被设置为 "1"，则该行完全退出该区域。逻辑 AND 不为零（第 1 行）。
- Otherwise the line need to be clipped:
 - Find the boundary of the region that line crosses by the "1" of code
 - Computer the intersection point between the boundary and the line.
 - Replace the outside point of line and re-compute the code.
 - Repeat until the line is accepted or rejected.
- 否则，需要对线条进行剪切：
 - 通过代码 "1" 找到直线穿过区域的边界
 - 计算边界与直线的交点。

- 替换直线的外点并重新计算代码。
- 重复上述步骤，直到直线被接受或拒绝。

The Sutherland-Hodgman Polygon-Clipping Algorithm



The algorithm is used to clip 2D polygon.

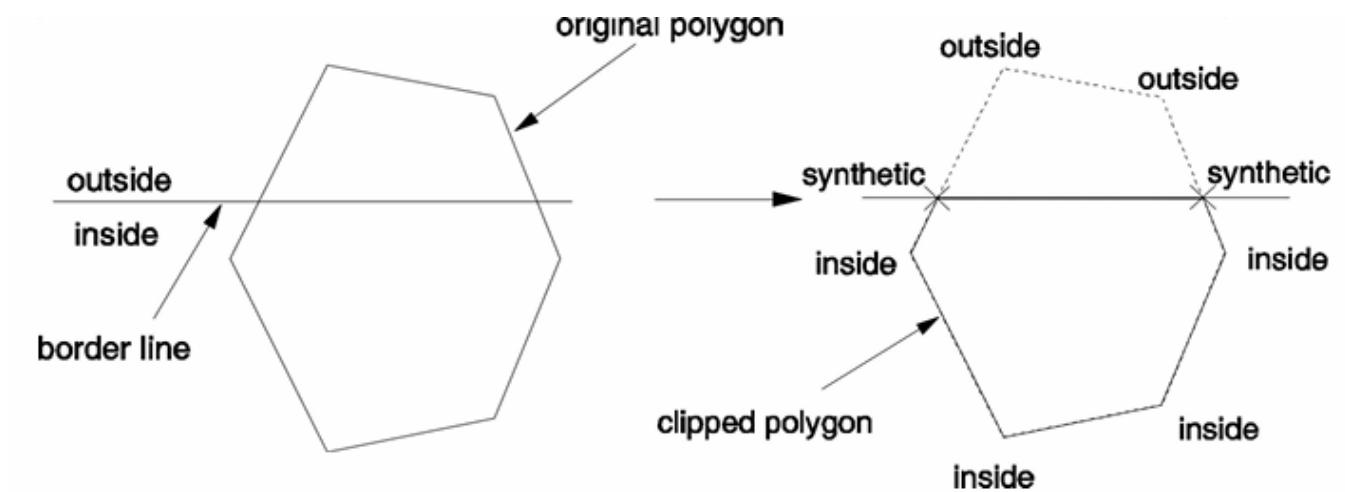
该算法用于剪切二维多边形。

If the polygon is completely outside(line 2) or inside(line 1), no clipping is needed.

如果多边形完全位于外侧（第 2 行）或内侧（第 1 行），则无需剪切。

Otherwise, the bounding box of polygon overlaps with the clip region(line 4, line 3), we need to clip it:

否则，多边形的边界框与剪辑区域（第 4 行，第 3 行）重叠，我们就需要剪辑它：

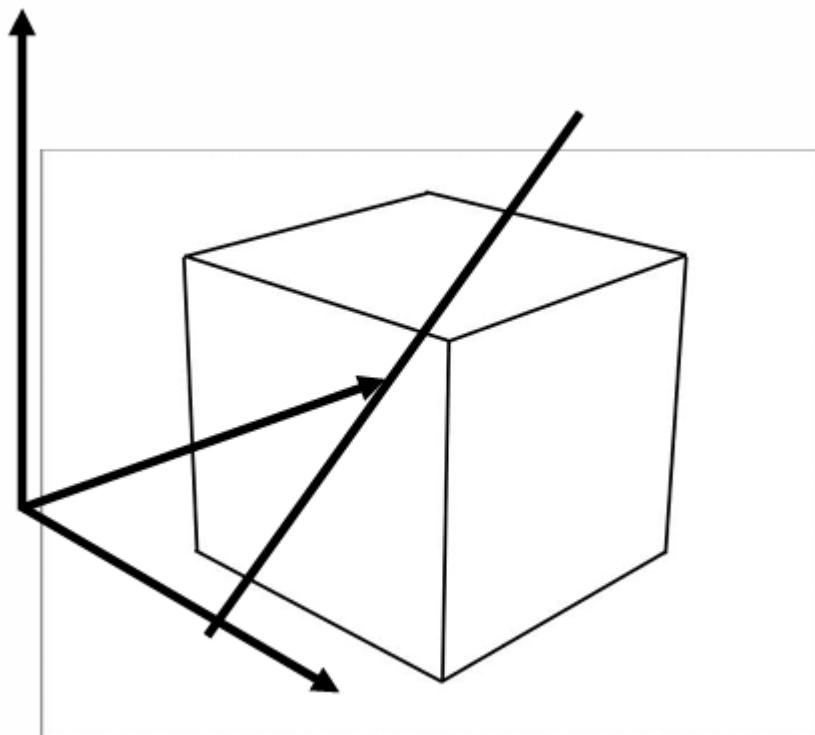


- Choose a border and use a vertex, mark it as "inside" or "outside".
- 选择边界并使用顶点，将其标记为 "内" 或 "外"。
- Check next vertex and mark it as "inside" or "outside".

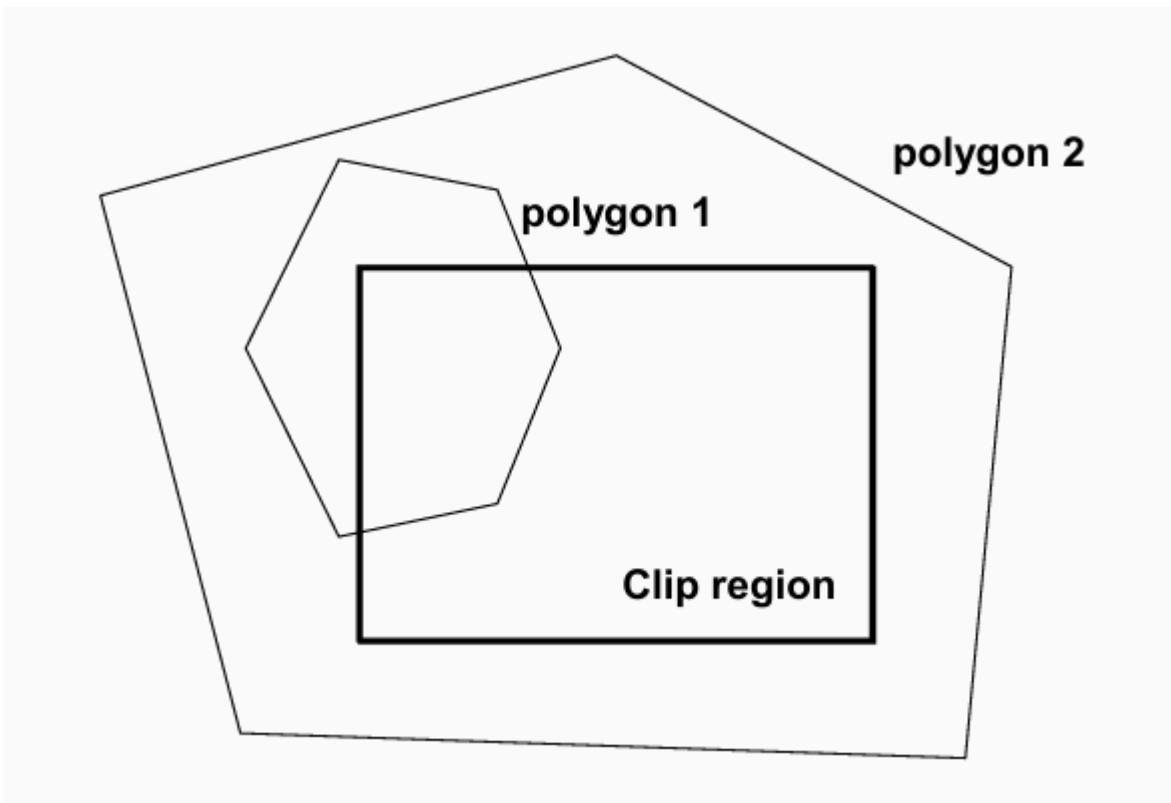
- 检查下一个顶点，标记为 "内" 或 "外"。
- Compare these 2 vertexes, if one is inside and the other is outside, compute the intersection point of the edge joining the two vertexes and border line, then take it as a new vertex and mark it as synthetic.
- 比较这两个顶点，如果一个在内侧，另一个在外侧，计算连接这两个顶点的边与边界线的交点，然后将其作为一个新顶点并标记为合成顶点。
- Repeat the process until all edges have been processed.
- 重复该过程，直到处理完所有边缘。
- After the whole polygon has been clipped by the border, throw away all outside vertexes and create a new polygon using inside and synthetic vertexes.
- 在整个多边形被边框剪切后，丢弃所有外顶点，然后使用内顶点和合成顶点创建一个新的多边形。
- Repeat the clipping process against next border line of the clip region.
- 针对剪辑区域的下一条边界线重复剪辑过程。

Exercise Question

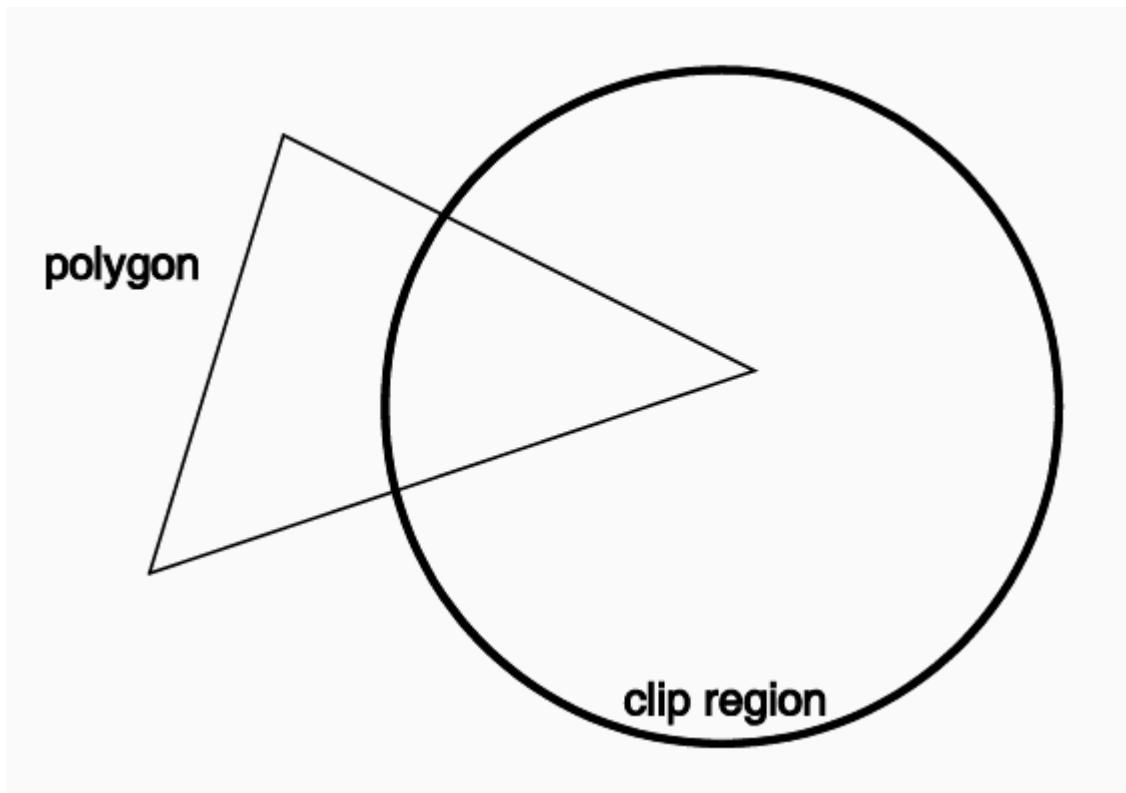
Q: Please show the main idea for extending the line clipping algorithm in 3D space, i.e., clipping a line against a 3D cubic.



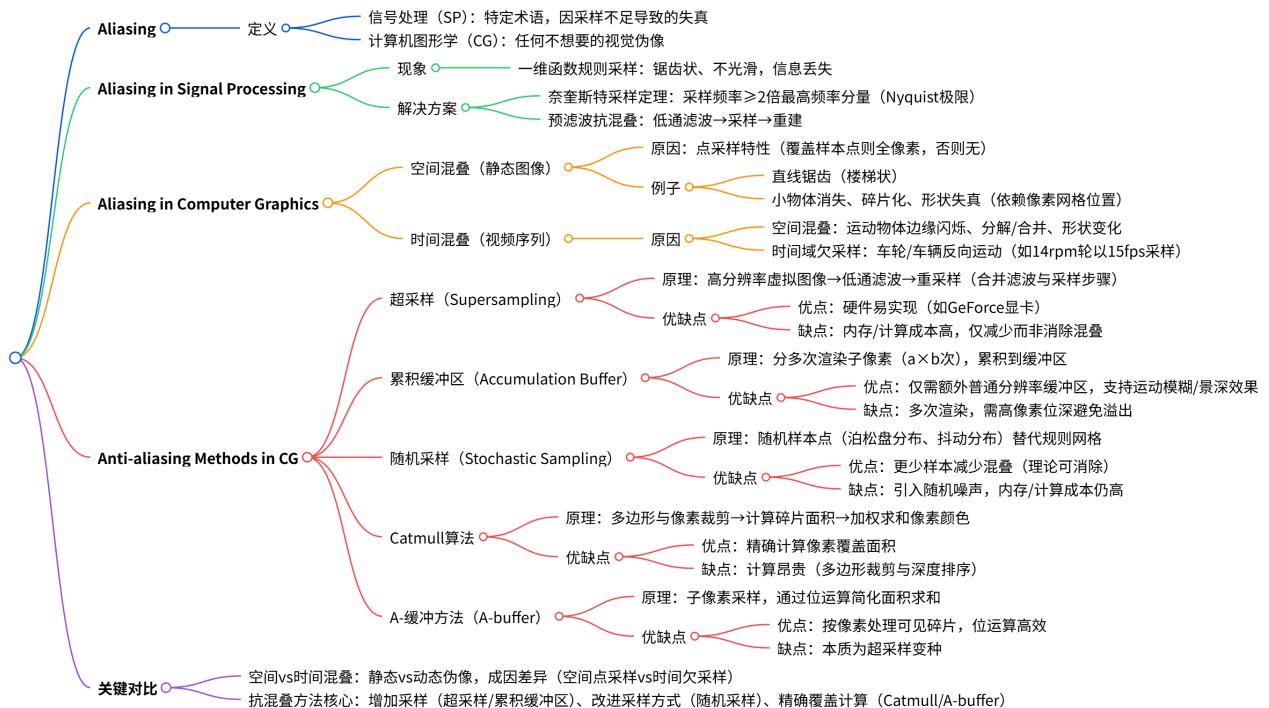
Q: Show how you would clip each of the two polygons against the clip region using the Sutherland Hodgman Polygon-Clipping Algorithm.



Q:Suggest a method to clip the following polygon against the circular clip region.



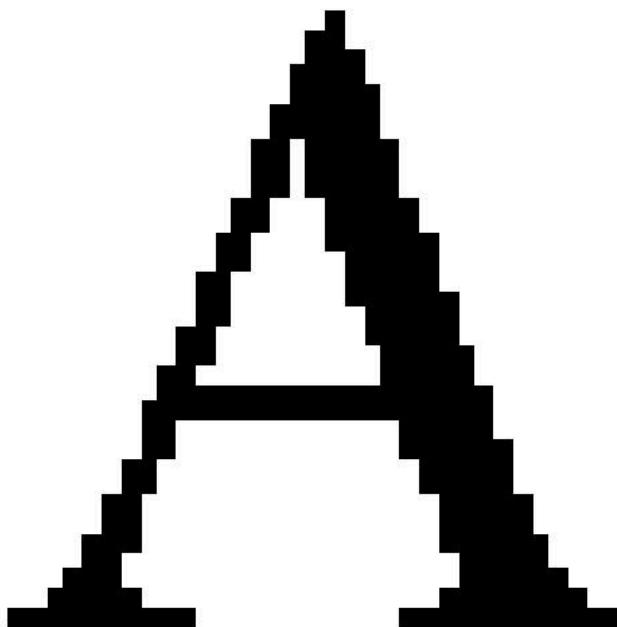
6.Aliasing and Anti-aliasing



豆包

你的AI助手, 助力每日工作学习

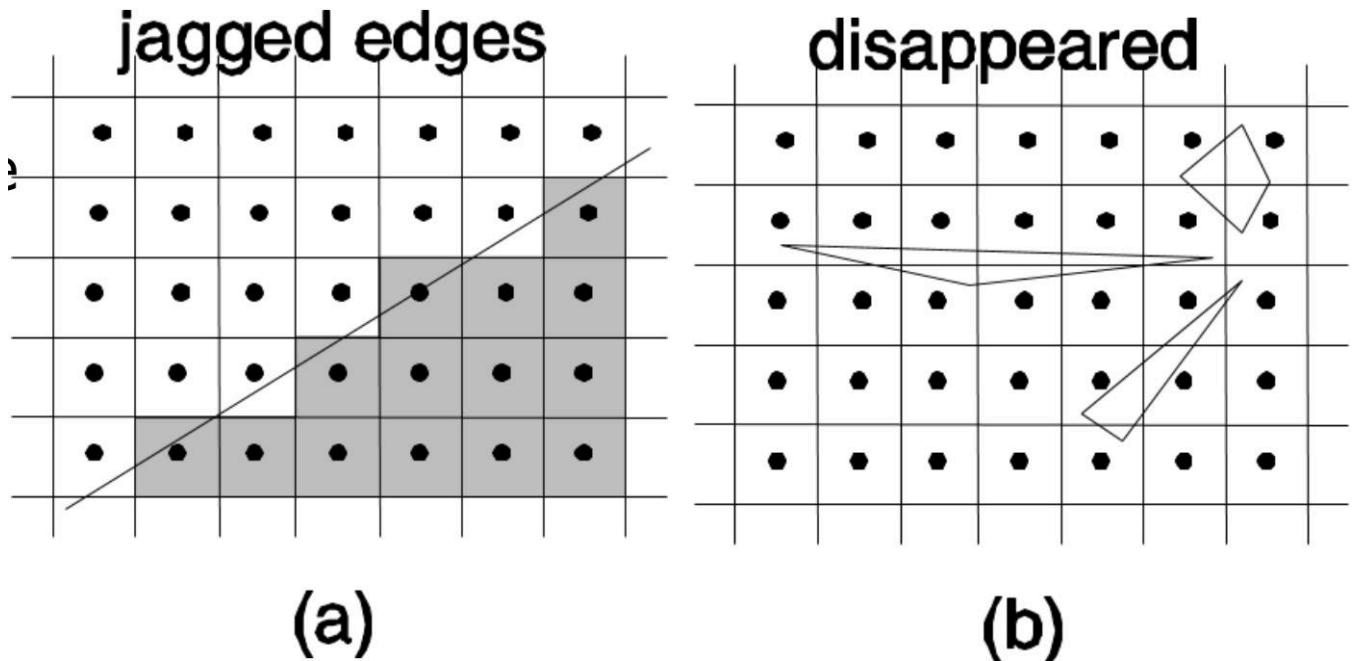
在 2D 绘图中, 直线可能呈现楼梯状, 字符可能显得离散化, 这种现象称为 **混叠 (aliasing)**



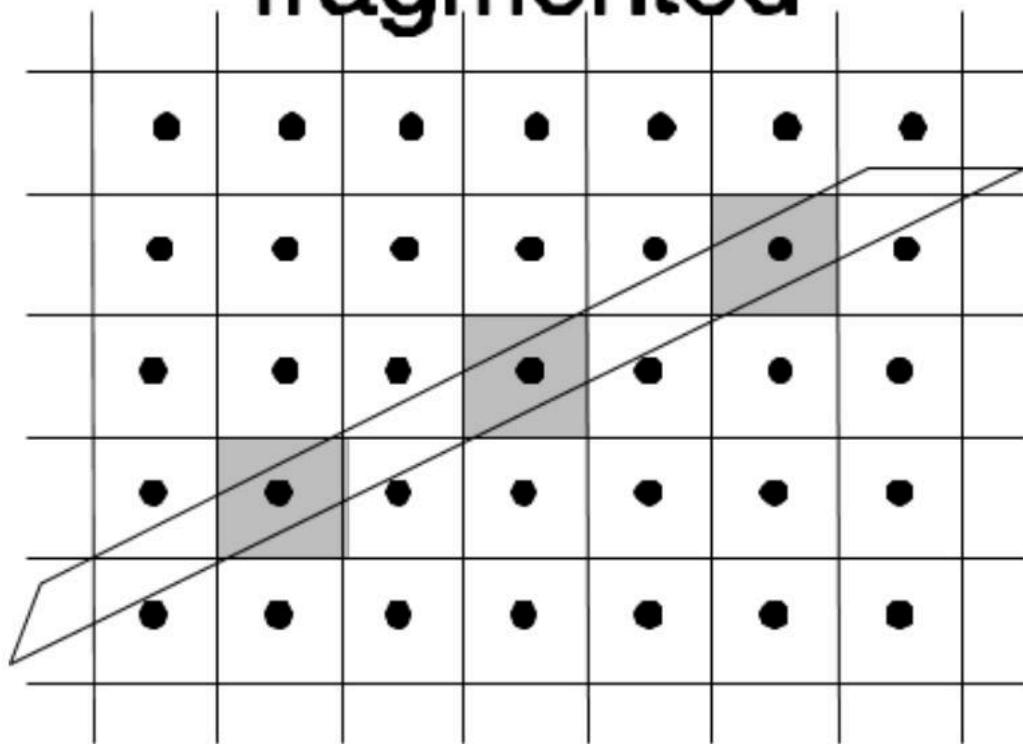
Spatial aliasing 空间混叠

静态图像中出现的混叠

空间混叠主要源于大多数绘图 / 渲染算法的 **点采样特性 Point Sampling Nature**。在 2D 绘图中，若一条线覆盖采样点，就假定它覆盖整个像素；若未覆盖采样点，就假定它完全不覆盖该像素。这种判断方式是一种近似 approximation，易引发混叠问题

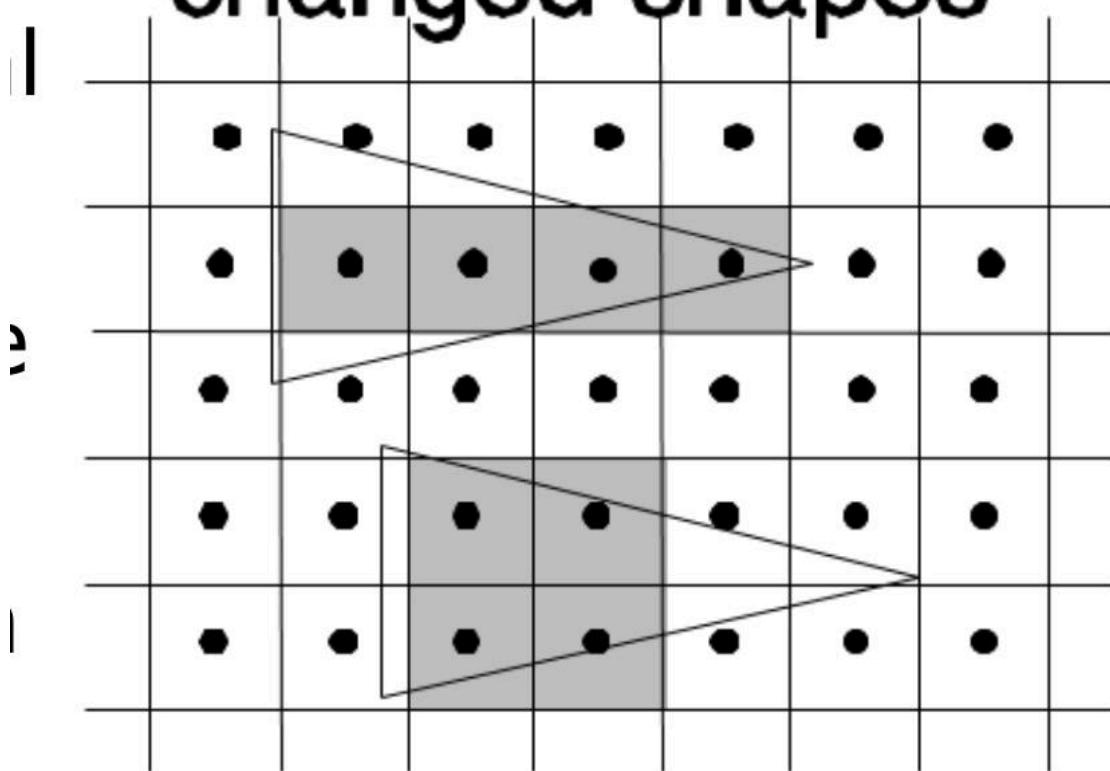


fragmented



(c)

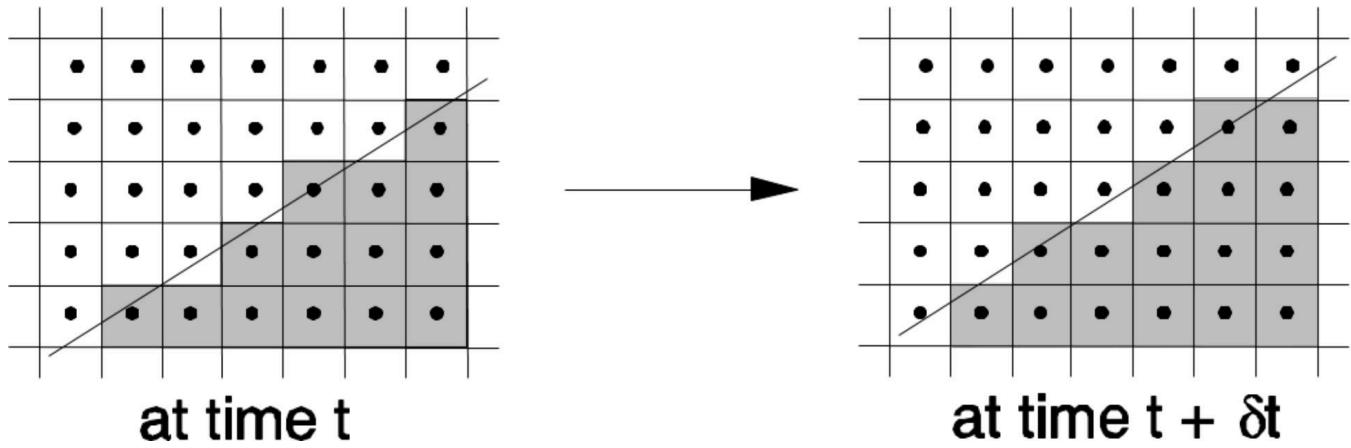
changed shapes



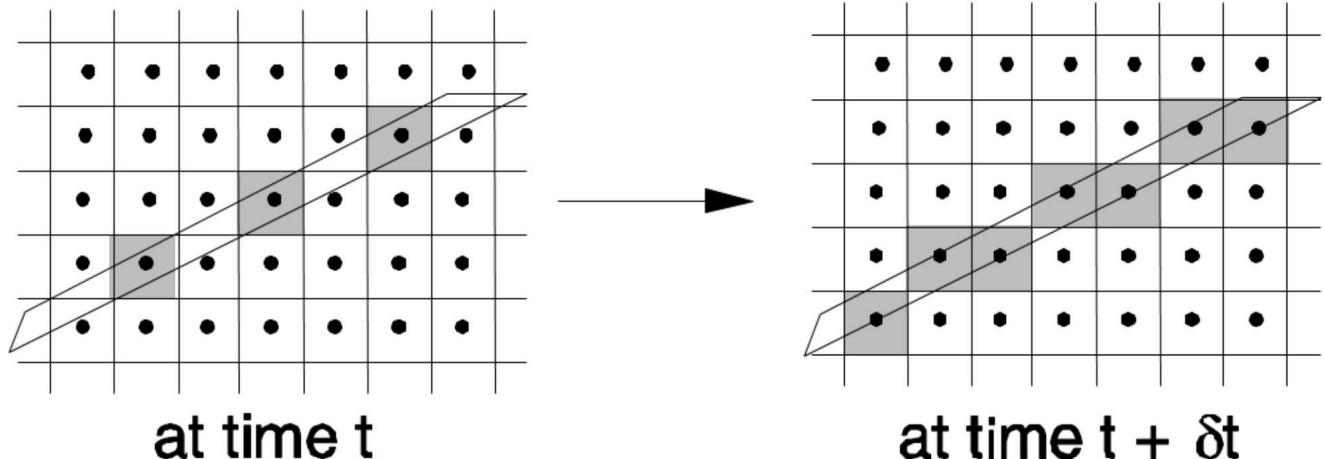
(d)

Temporal Aliasing 时间混叠

若多边形代表的物体在动画序列中从时间 t 到 $t + \delta t$ 轻微向上移动，其边缘的锯齿状（楼梯状）可能会显得跳动
常出现在动画中

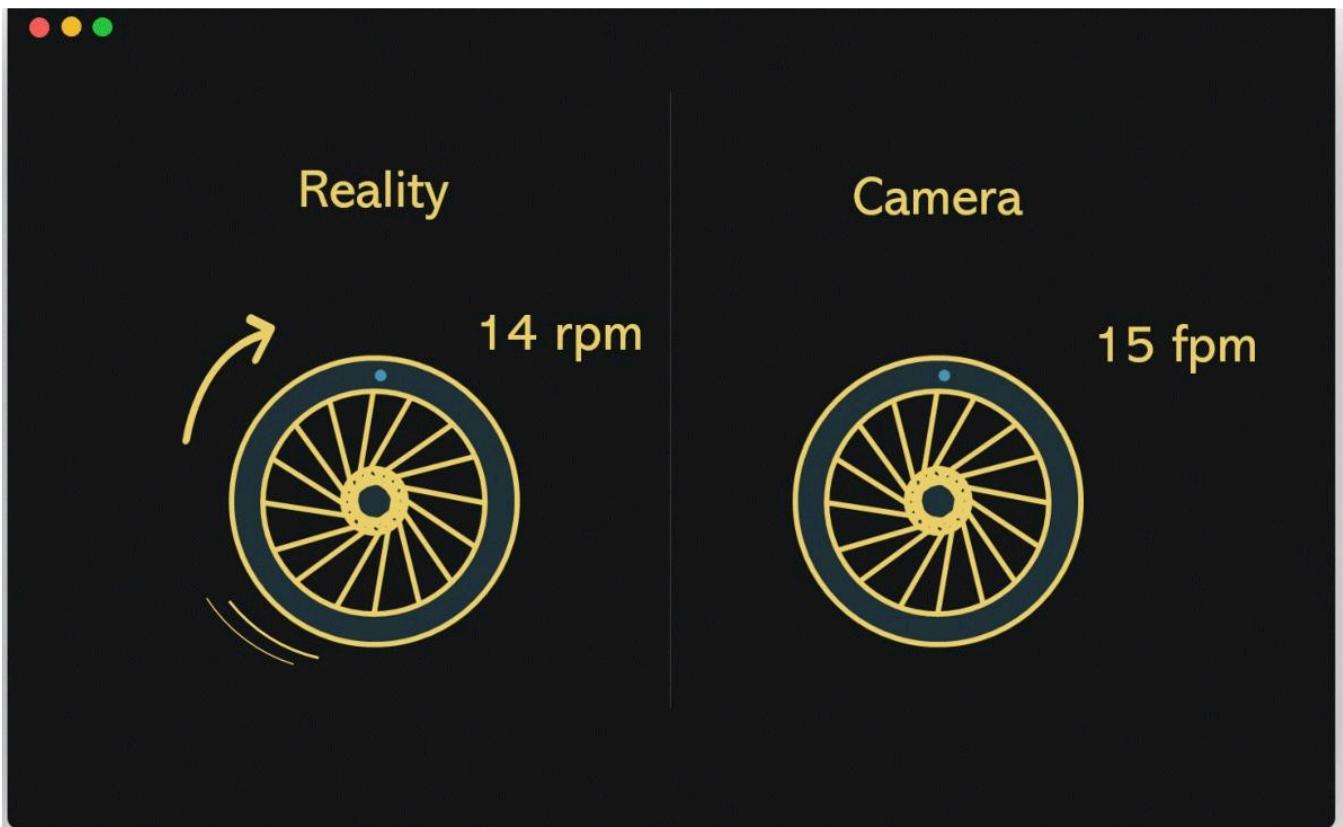


本质上还是由空间堆叠引起



此外也存在欠采样（undersampling）导致的时间混叠

例如由于采样不足导致轮子转动方向错误



Anti-aliasing 抗混叠

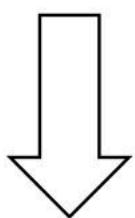
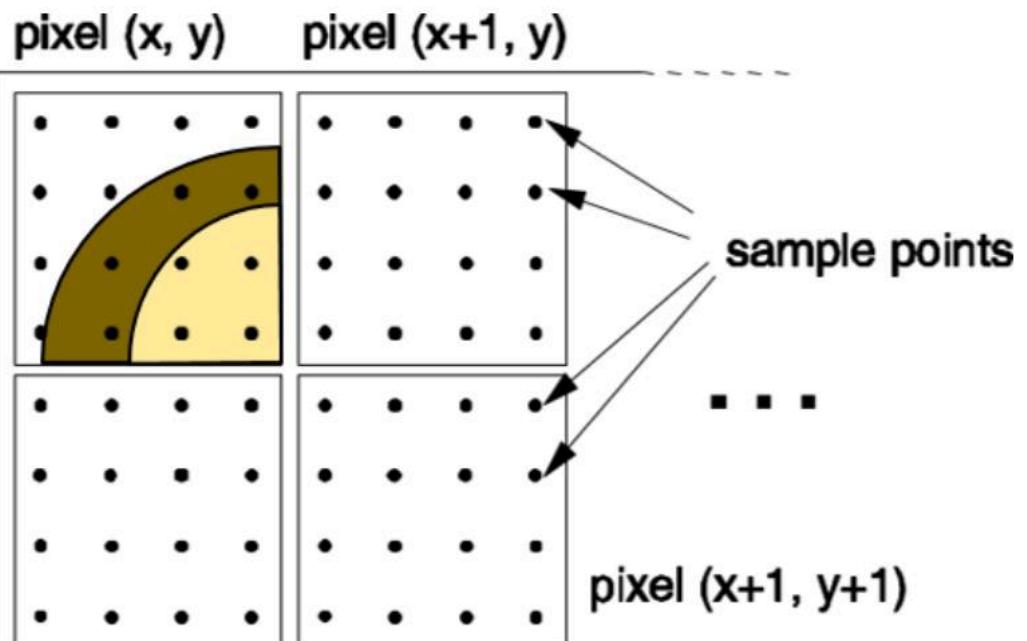
超采样 (Supersampling): 通过增加样本数量来减少混叠效果

过程:

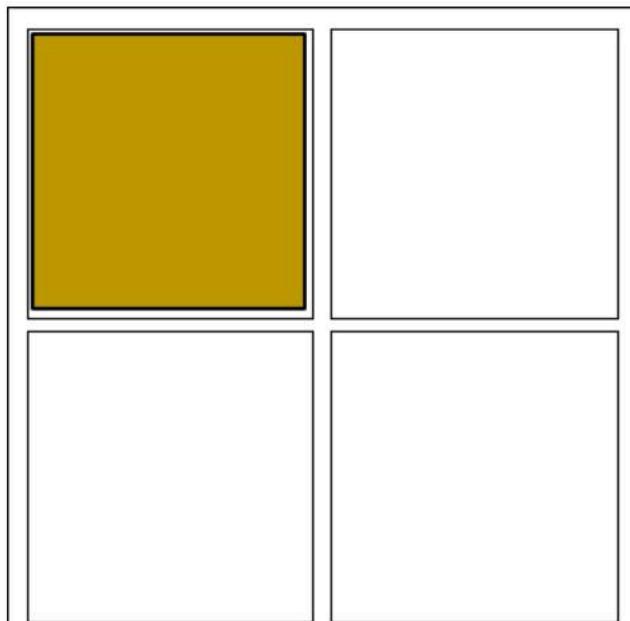
1. 创建一个比最终图像分辨率更高的虚拟图像
2. 应用 low-pass filter，用于平滑图像并减少高频噪声
3. 对滤波后的图像进行重新采样

实际应用中，步骤 2 和 3 会合并，即在新的采样点直接对图像进行卷积滤波

低通滤波器用于平滑图像，去除高频噪声或细节



$$\frac{\text{yellow square} + \text{brown square} + \dots + \text{white square}}{16} = \text{brown square}$$



⋮
⋮
⋮

优点：在硬件上易于实现

缺点：

- 需要大量内存和处理时间
- 不能消除混叠，只能减少混叠。只要还有采样点存在就会有堆叠现象

Exercise Question

Describe how to extend the ray-tracing method to supersampling to address spatial aliasing problem.

分出子像素后对每个子像素都应用光线追踪，都发送一条光线。最后对所有子像素颜色值取平均

Accumulation Buffer 累积缓冲区

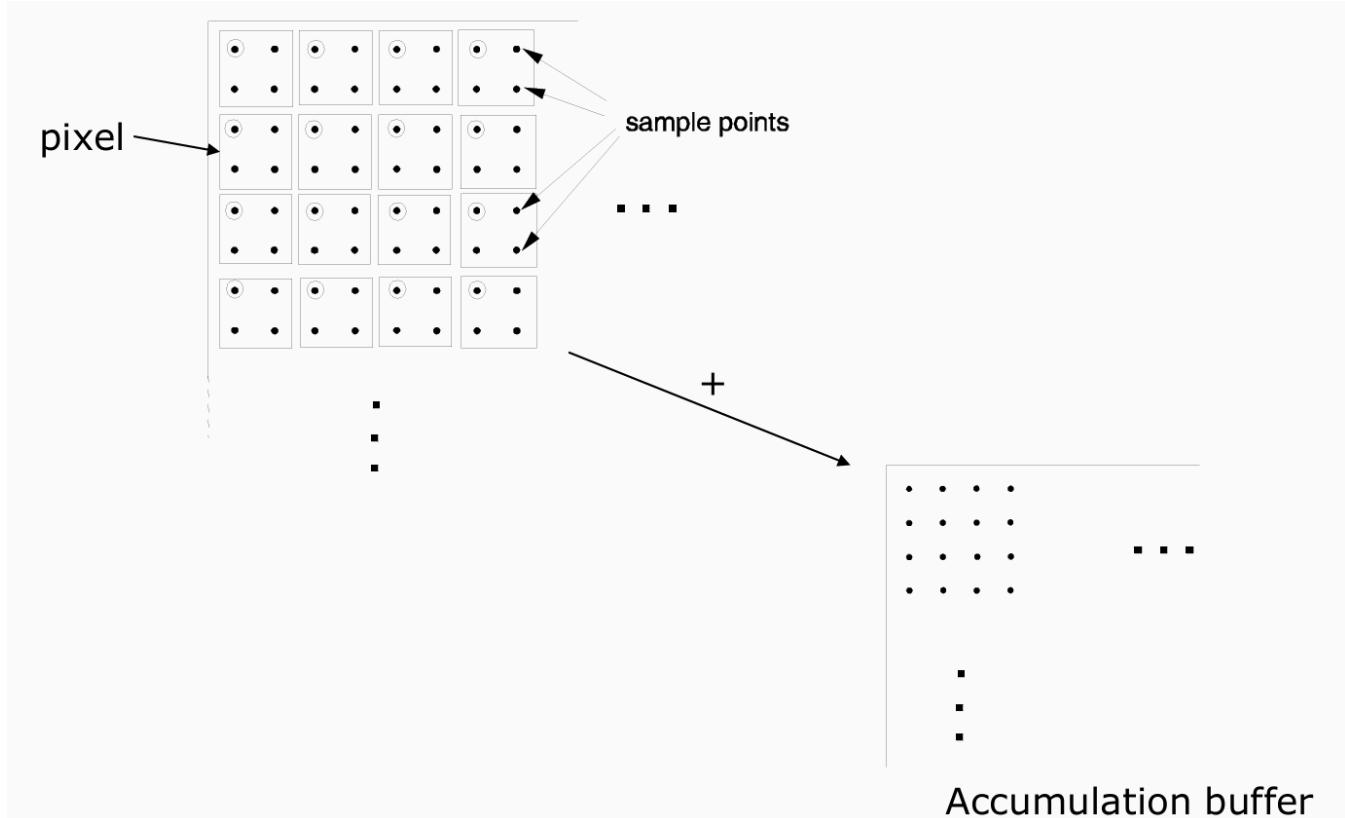
解决了超采样（Supersampling）内存成本高的问题，且具备额外优势

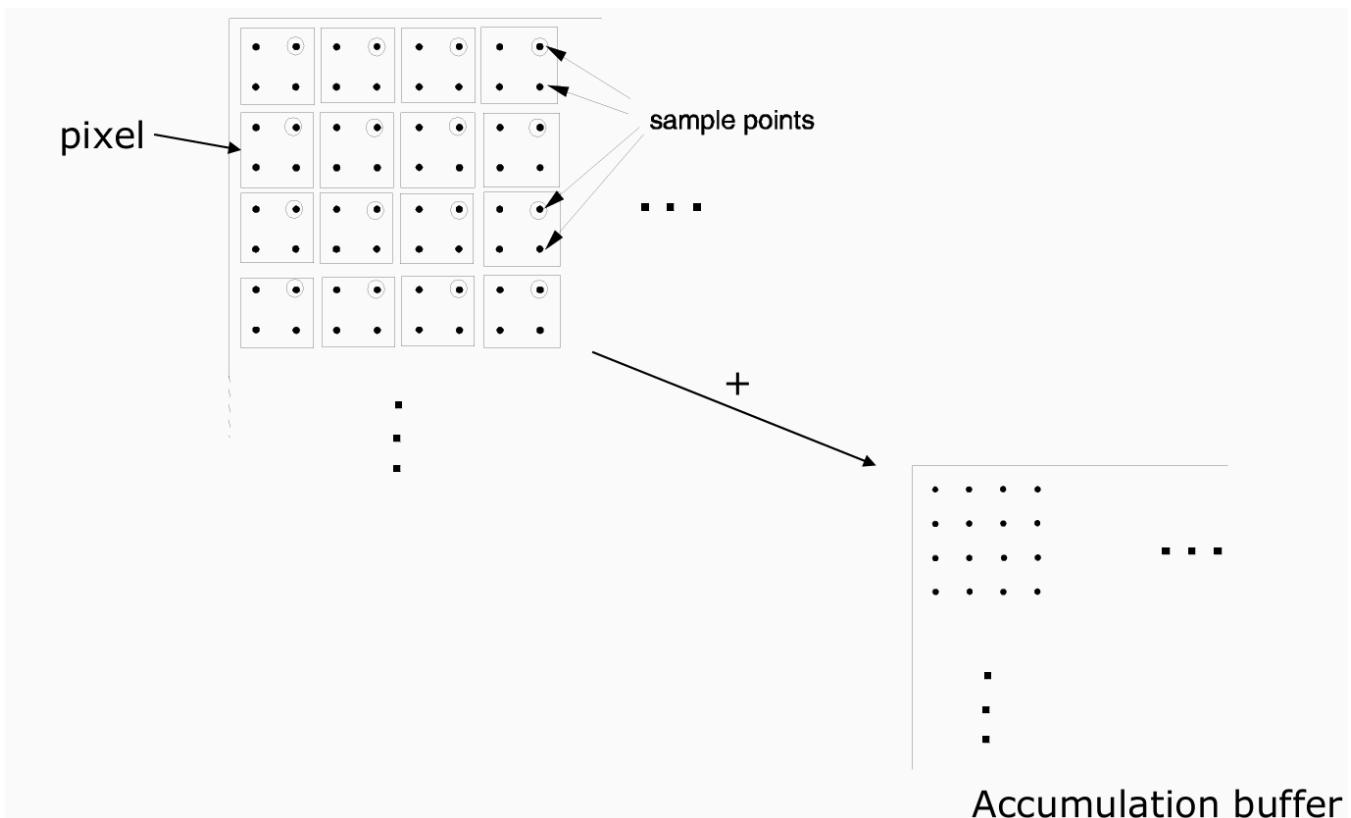
并非一次性渲染所有子像素，而是 **每次渲染一个子像素**

假设子像素分辨率为 $(a \times b)$ ，最终图像分辨率为 $(m \times n)$ 。此方法需对图像渲染 $(a \times b)$ 次，每次生成 $(m \times n)$ 分辨率的图像，每个位置存储一个像素的子像素颜色值

输出图像在 **累积缓冲区** 中逐步累积，直至完成所有 $(a \times b)$ 次图像渲染

最终将累计缓冲区内的颜色值平均化处理





优点

- **内存需求低**: 仅需多一个普通分辨率的缓冲区，相比超采样大幅降低内存成本。
- **硬件兼容性好**: 易于集成到现有图形硬件中，方便实际应用。
- **支持运动模糊**: 通过沿时间轴累积多个图像，可模拟运动模糊效果，增强画面真实感。
- **实现多样效果**: 每次渲染时改变相机位置，还能实现景深 (depth of field) 等其他视觉效果，功能丰富。

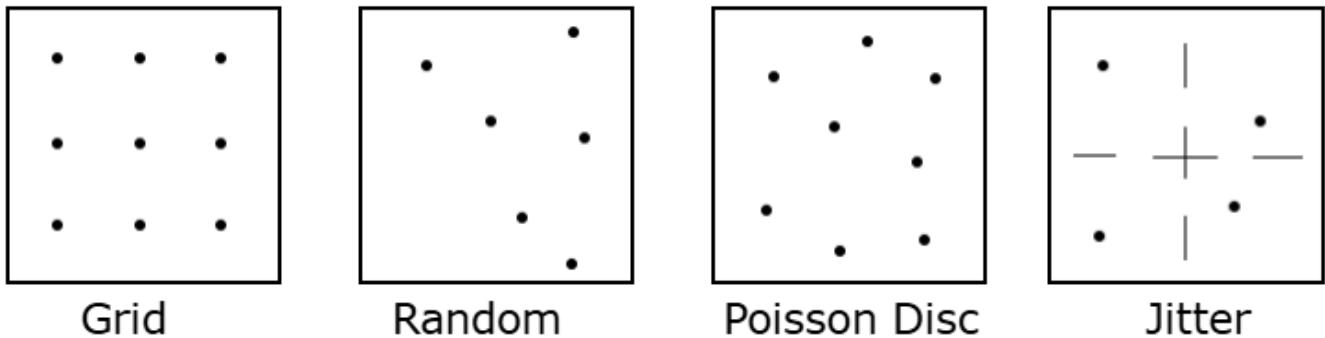
缺点

- **渲染效率低**: 生成一张图像需多次渲染 (rendering passes)，耗时较长。
- **像素位深要求高**: 累积缓冲区每个像素需更多位 (bits)，以避免累积过程中出现溢出 (overflow)，对硬件存储有更高要求。

Stochastic sampling 随机采样

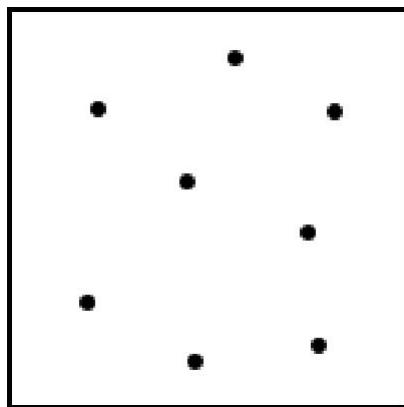
随机采样使图像每个区域都有被采样的概率。因此，均匀采样点间的小特征更易被非均匀的随机采样点检测到，弥补规则采样的不足即空隙问题。

该方法在每个像素内使用随机采样点（如随机分布、泊松盘分布、抖动等），替代超采样中的规则网格。通过这种方式，将混叠现象转化为噪声，从视觉上减轻混叠的规律性瑕疵。



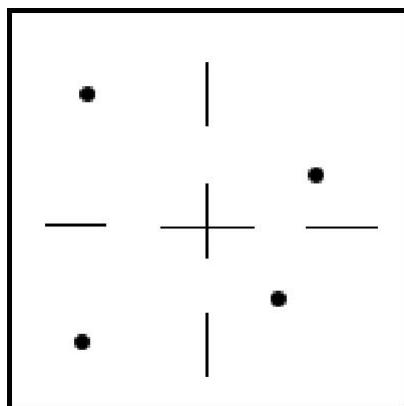
泊松盘分布 (Poisson - disc distribution)

- **特性：**基于泊松分布，样本点之间存在最小距离约束。
- **生成方式：**随机添加点，若新点与之前任何点过近，则移除该点，确保点与点之间保持一定距离。右侧图示展示了这种分布的点，彼此间隔均匀，避免聚集。



抖动分布 (Jittered distribution)

- **特性：**从规则网格样本出发，对每个样本在随机方向上进行轻微扰动。
- **视觉效果：**外观上更具“块状”或颗粒感。



优点：

- 每个像素使用更少的样本就能更好地减少混叠。
- 理论上，当样本数量足够大时，至少在理论层面可以消除混叠。

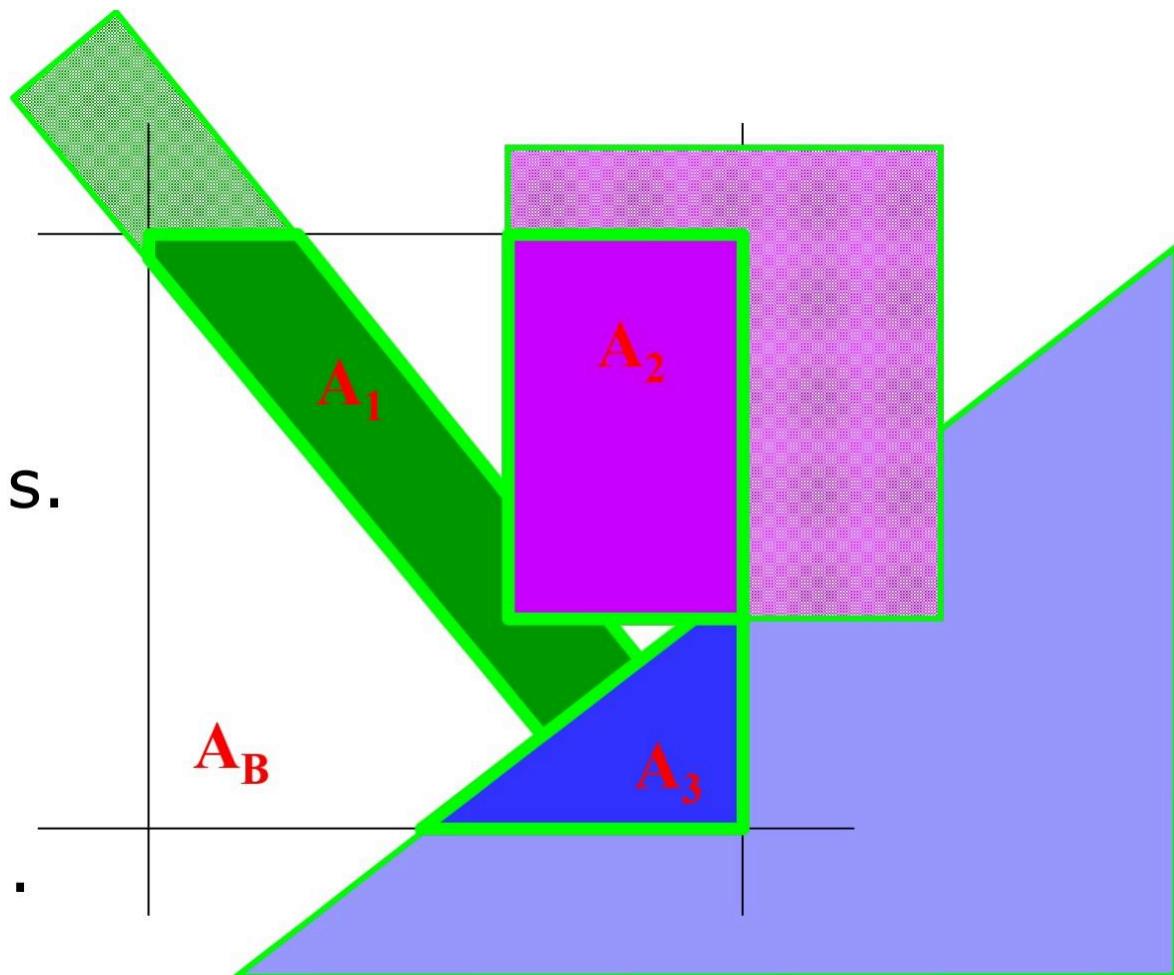
缺点：

- 会给图像增加随机噪声。
- 仍然需要大量的内存和处理时间。

Catmull 算法

算法步骤

1. **多边形裁剪**: 将每个多边形与每个像素进行裁剪, 形成多边形碎片 (“Clip each polygon against each pixel to form polygon fragments”)。
2. **确定可见碎片**: 判断哪些碎片是可见的 (“Determine visible fragments”)。
3. **计算碎片面积**: 求出每个可见碎片的面积 (“Find fragment areas”)。
4. **颜色加权**: 将每个碎片的面积与其颜色值相乘 (“Multiply by fragment colors”)。
5. **求和得到最终颜色**: 将所有碎片的加权结果相加, 得到最终像素颜色 (“Sum for final pixel color”)。

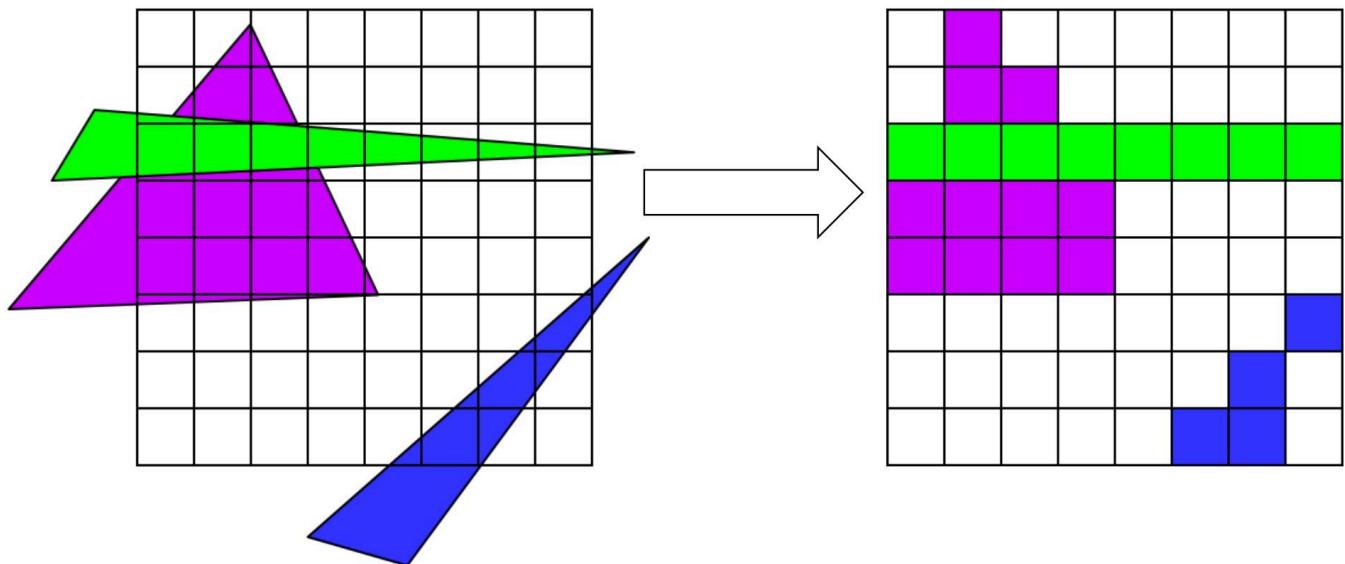


- **优点**: 精确计算多边形对像素的覆盖程度, 即其对像素颜色的贡献, 能精准处理抗混叠, 提升图像质量。
- **缺点**: 因涉及多边形裁剪以及按深度对多边形碎片排序, 计算成本极高。多边形裁剪需细致处理每个多边形与像素的关系, 深度排序又增加了复杂度, 导致算法在计算资源和时间上消耗巨大。

The A-buffer method

和z-buffer类似，只考虑可见片段

使用 **子像素采样 Subpixel sampling** 来简化求和区域



优点

- 处理效率与可见片段相关：**每个像素的处理仅取决于可见片段的数量，避免了对大量无关采样点的无效计算，提升了处理针对性。
- 位操作实现高效：**通过对子像素掩码进行按位逻辑操作来高效实现，利用底层位操作特性，能有效提升算法执行速度。

缺点

- 本质仍属超采样：**A - buffer 方法从根本上仍是一种超采样算法，这意味着它依然可能面临超采样算法共有的问题，如较高的内存需求和计算成本，尽管在实现上有优化，但未脱离超采样的核心框架。

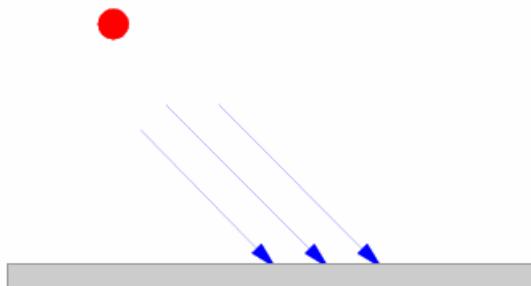
7. Illumination and Shading

Illumination

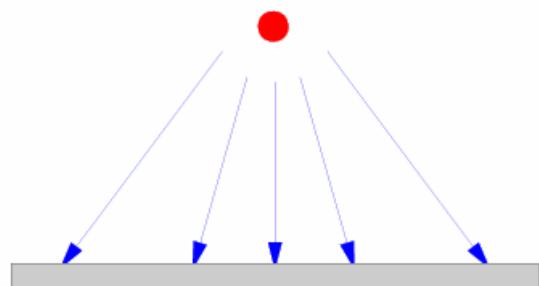
Light sources:

- Ambient light: no fixed direction, causing objects to be uniformly illuminated
- 环境光：没有固定的方向，导致物体被均匀照明
- point light source: an ideal light source. It emits directional light rays
- 点光源：理想的光源。它发出定向光线

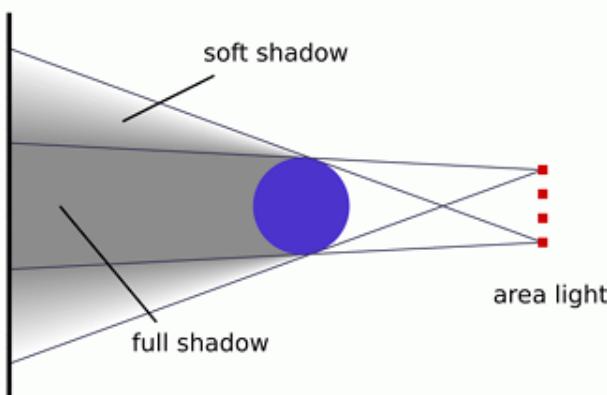
distant light source



close light source



- Spotlight: emit approximately parallel light in a particular direction.
- 聚光：在特定方向发出近似平行的光。
- Area light source: occupies a finite, one- or two-dimensional area of space, can cause soft shadow
- 面光源：占据有限的、一维或二维的区域空间，可造成软影



Illumination Model

Determines the light intensity of a surface point by simulating some light attributes.

通过模拟一些光照属性来确定表面点的光照强度。

Ambient reflection: 环境反射

Ambient light intensity I_c is equal in all directions.

环境光强度 I_c 在所有方向上相等。

The final light intensity effect I_a :

最终的光强效果 I_a :

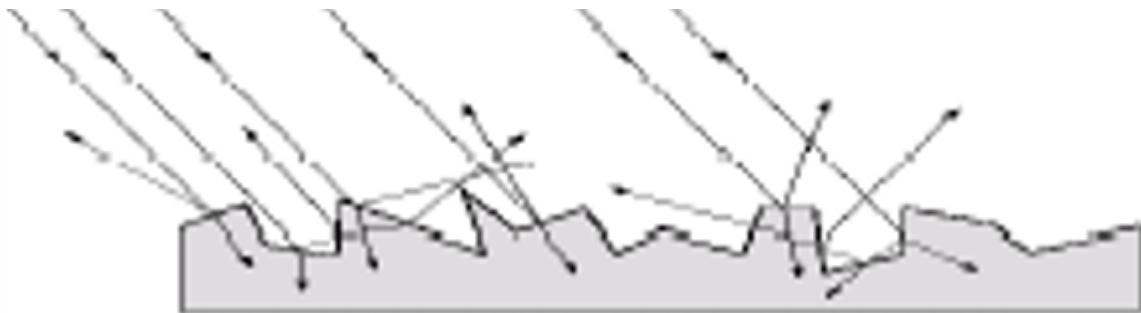
$$I_a = k_a * I_c$$

k_a means the ambient reflection coefficient, and $0 < k_a < 1$

k_a 表示环境反射系数, $0 < k_a < 1$

Diffuse reflection: 漫反射

When light hits a rough surface, the light is equally reflected in any direction over the hemisphere. 当光线照射到粗糙的表面上时，光线在半球上的任何方向都会受到同样的反射。

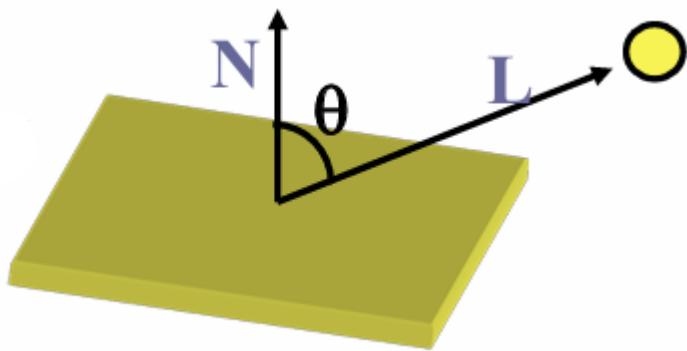


The intensity is proportional to $\cos\theta$:

强度与 $\cos\theta$ 成正比:

$$I_d = k_d \cdot I_L \cdot \cos\theta = k_d \cdot I_L \cdot (N \cdot L)$$

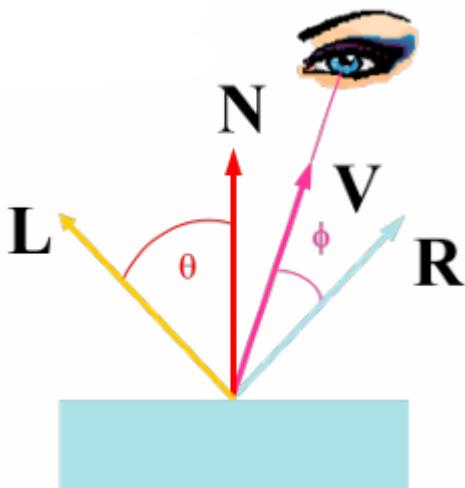
- k_d -surface diffuse coefficient.
- k_d -表面扩散系数。
- θ - the angle between the surface normal and the incoming light ray, called the angle of incidence.
- θ -表面法线与入射光线之间的角度，称为入射角。
- N and L are normalized vectors.
- N和L是归一化向量。



Specular reflection 镜面反射

Caused by glossy, shiny surfaces. When viewing from different angle, the light intensity is different.

由光滑、有光泽的表面引起。从不同角度观察，光强不同。



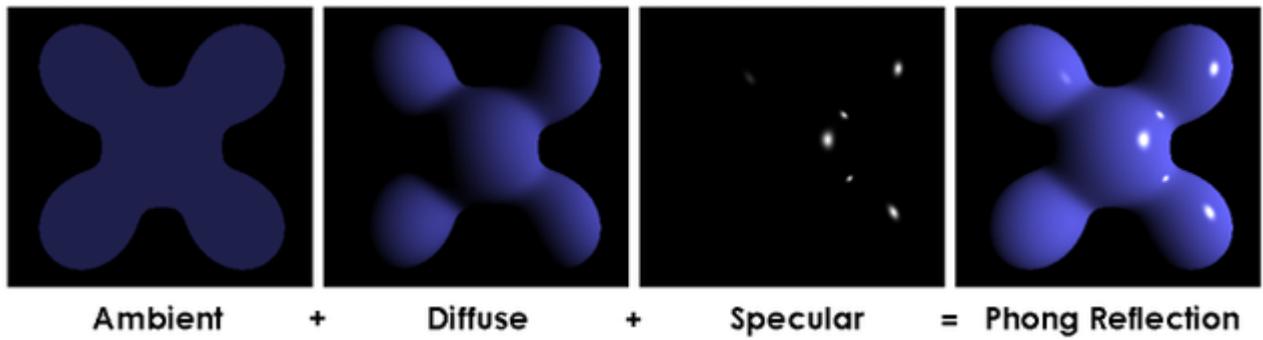
The intensity mainly depends on viewing angle and specular exponent:

强度主要取决于视角和高光指数：

$$I_s = k_s * I_L * \cos^n \varphi = k_s * I_L * (V \cdot R)^n$$

- k_s -surface specular coefficient.
- k_s -表面高光系数。
- n-specular-reflection parameter or specular exponent (for mirror $n = \infty$).
- n-镜面反射参数或镜面指数（镜面 $n=\infty$ ）。
- φ -angle between R and V. R and V are normalized vectors.
- φ -R和V之间的角。R和V是归一化向量。

Total intensity



Finally, in Phong Illumination Model, the total light intensity is:

最后，在Phong光照模型中，总光照强度为：

$$I = I_a + \sum_{i=1}^{lights} I_{d,i} + I_{s,i}$$

Shading

Use the color calculation from an illumination model to determine the pixel colors

使用光照模型中的颜色计算来确定像素的颜色

Flat shading



Only compute the light intensity value once, then shade the whole polygon with the same value.

只计算一次光照强度值，然后用相同的值对整个多边形进行着色。

It's fast and simple but the result is very coarse, and the intensity discontinuity is its main problem.

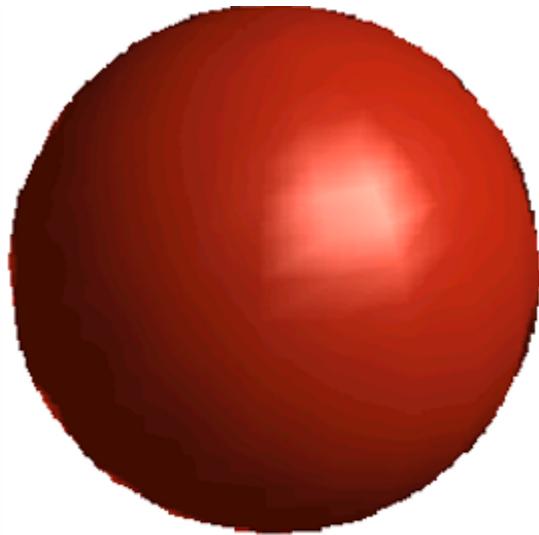
它快速、简单，但结果过于粗糙，且灰度不连续是其主要问题。

Smooth shading

Need to calculate per-vertex normal, make shading result more smooth.

需要计算每个顶点的法线，使着色结果更加平滑。

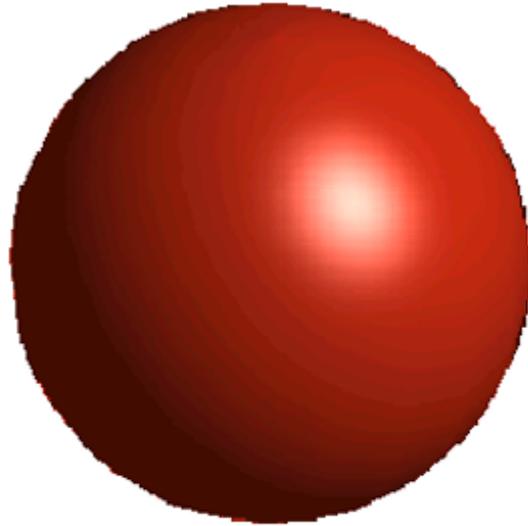
Smooth shading——Gouraud Shading



Gouraud

- Interpolate color across triangles
- 在三角形中插值颜色
- Fast, supported by most of the graphics accelerator cards but difficult to handle specular reflection
- 快速，支持大多数图形加速器卡，但难以处理镜面反射

Smooth shading——Phong shading

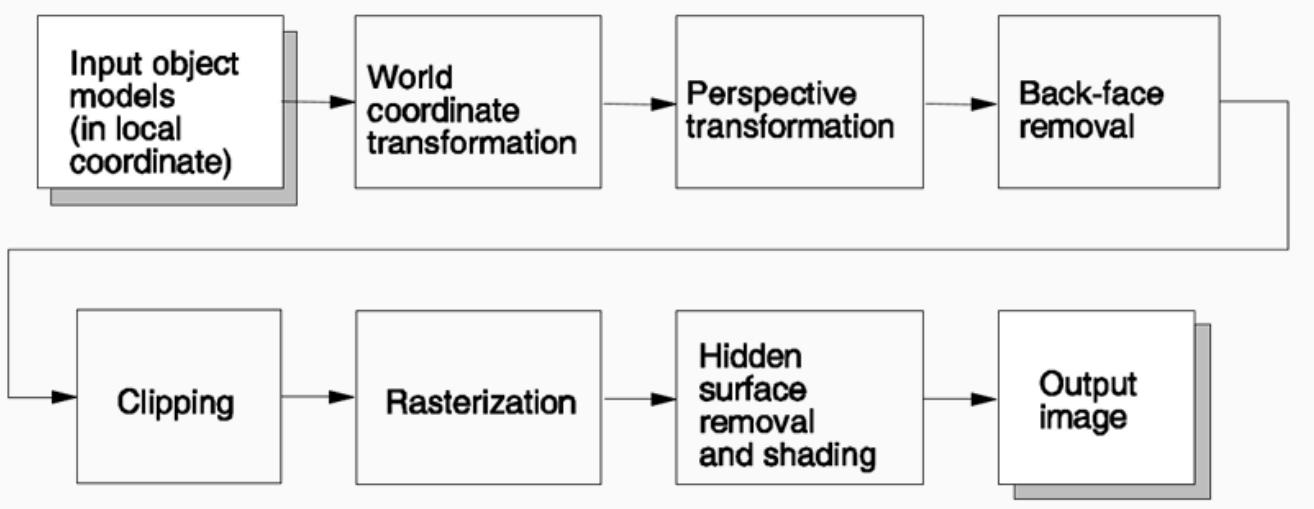


Phong

- Interpolate normals across triangles
- 在三角形上插值法线
- More accurate but slower.

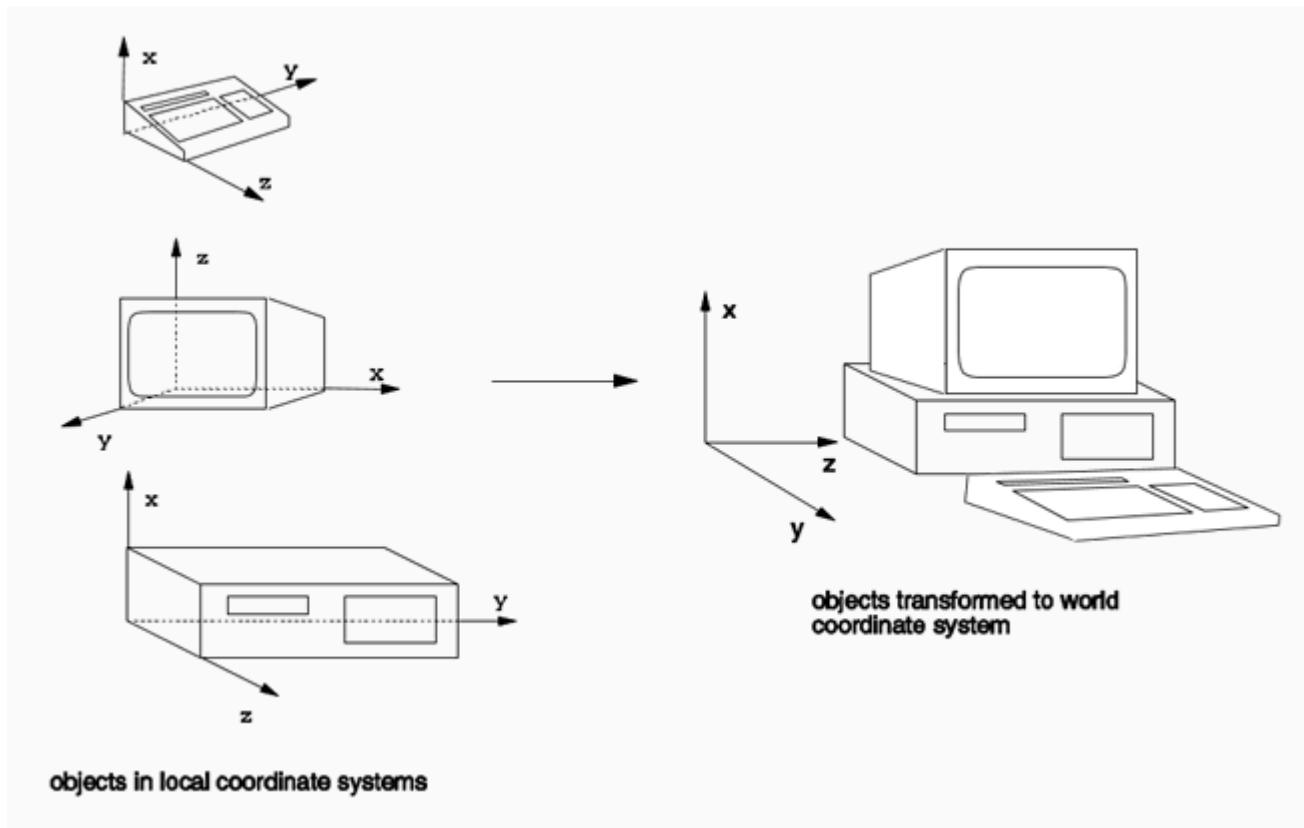
- 更精确但更慢

8.The Rendering Pipeline (Rasterization-based)



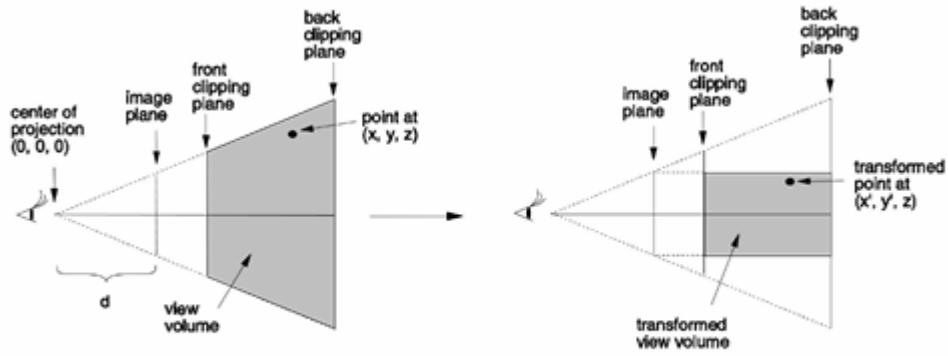
Input: Object Models(local coordinate)

输入：对象模型（局部坐标）

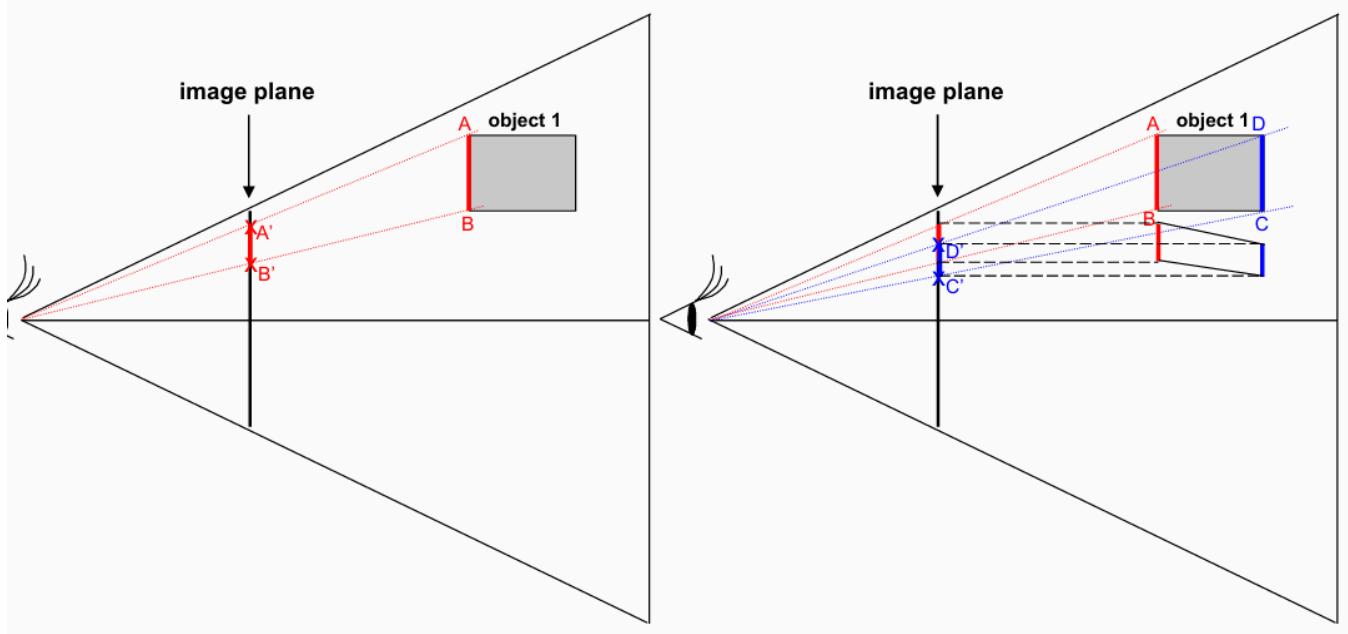


- Word coordinate transformation: transform each objects from its local coordinate to a common world coordinate. (2.Modeling)

Word坐标转换 (Word coordinate transformation)：将每个对象从其局部坐标转换为公共世界坐标。



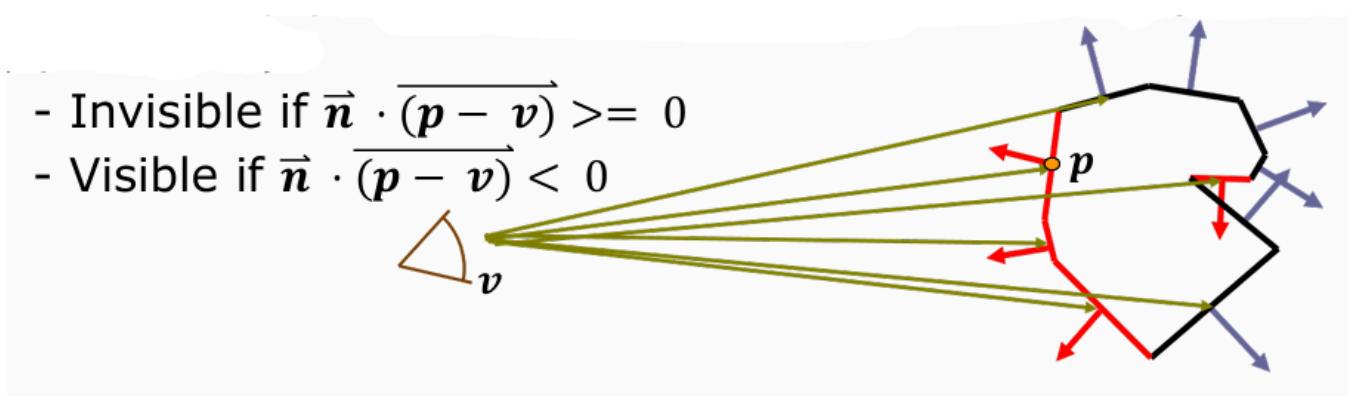
Perspective transformation



2. Perspective transformation: This step perspectively transforms each object such that distant object will appear smaller. Different from perspective projection because it retains depth value z .

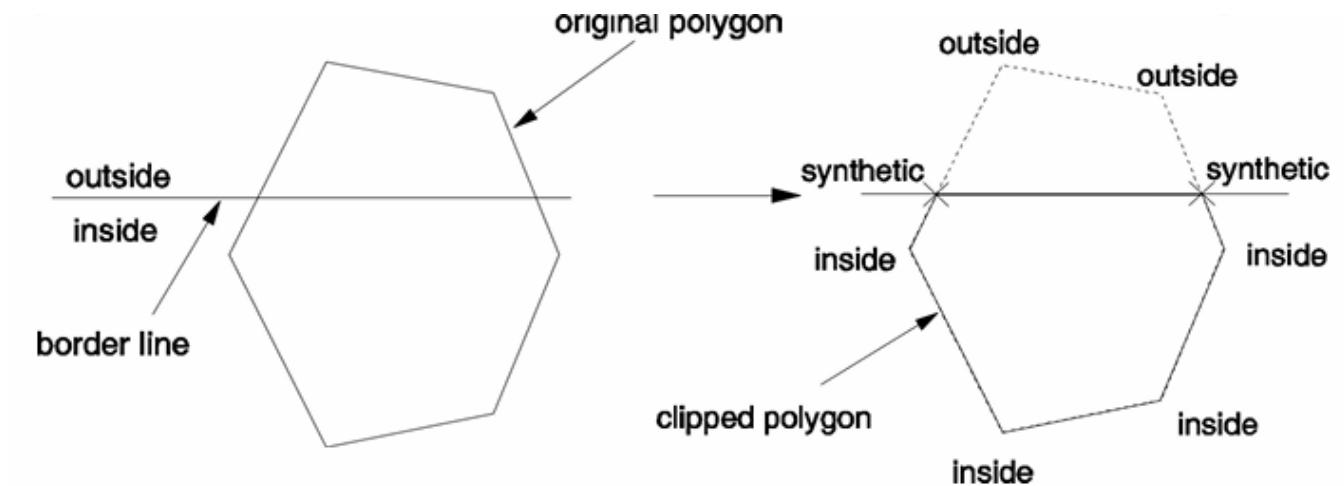
透视变换：这一步透视变换每个物体，使远处的物体看起来更小。不同于透视投影，因为它保留了深度值 z 。

$$\begin{cases} x' = \frac{d}{z} x \\ y' = \frac{d}{z} y \\ z' = z \end{cases}$$



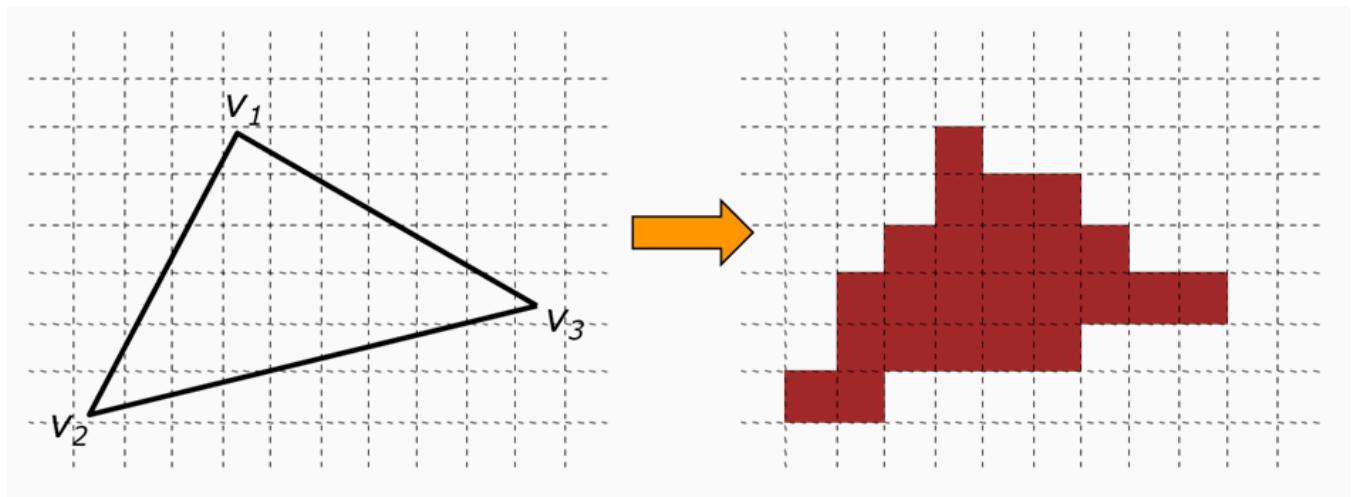
3. Back-face removal: Remove those surfaces that are unable to see to save time.

背面移除：移除那些看不见的表面以节省时间。



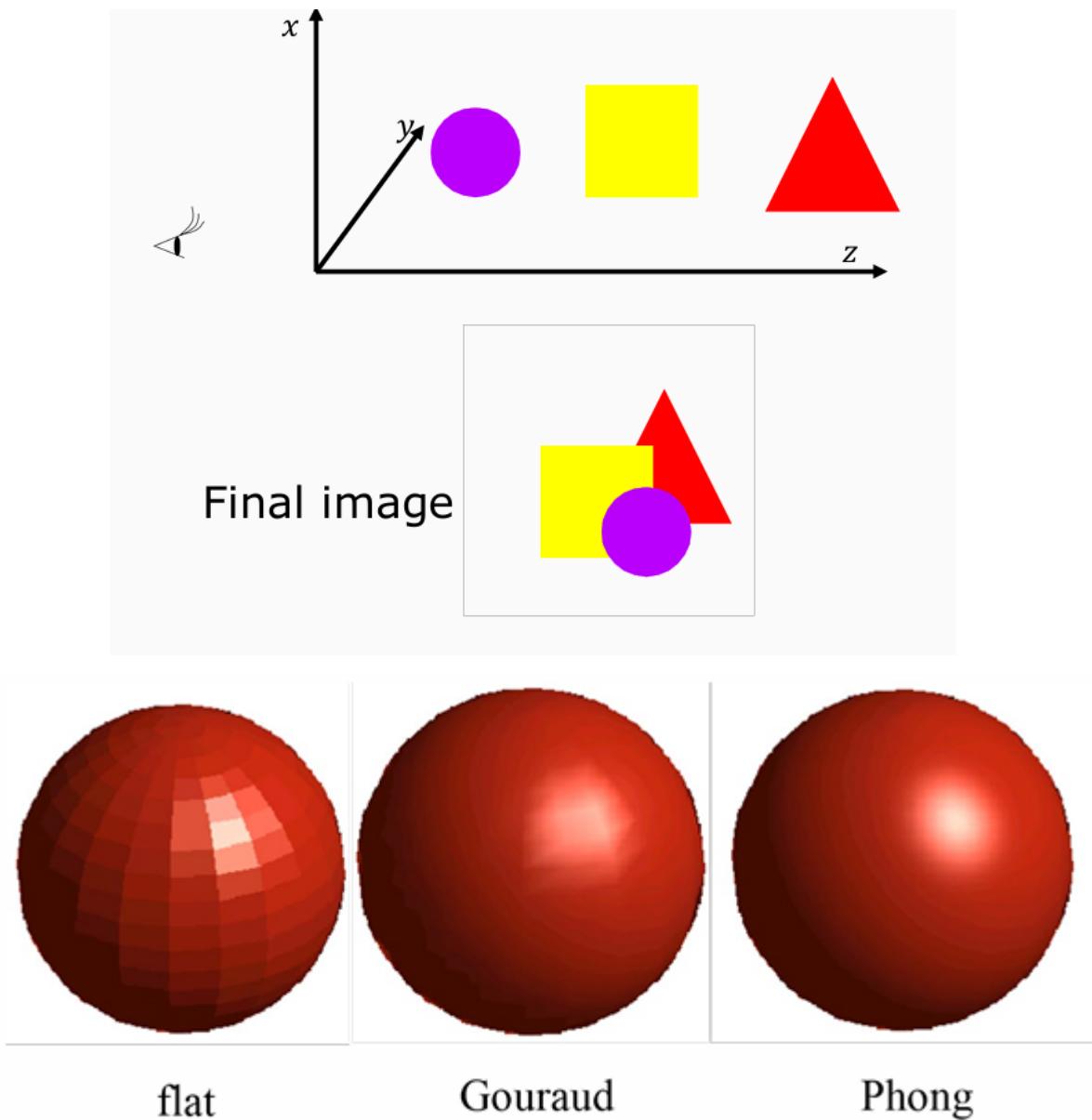
4. Clipping: Remove all parts of objects which are outside the view.

裁剪：移除视图之外对象的所有部分。



5. Rasterization: Take a primitive and figuring out which pixels it covers.

光栅化 (Rasterization): 取一个基元，找出它覆盖了哪些像素。

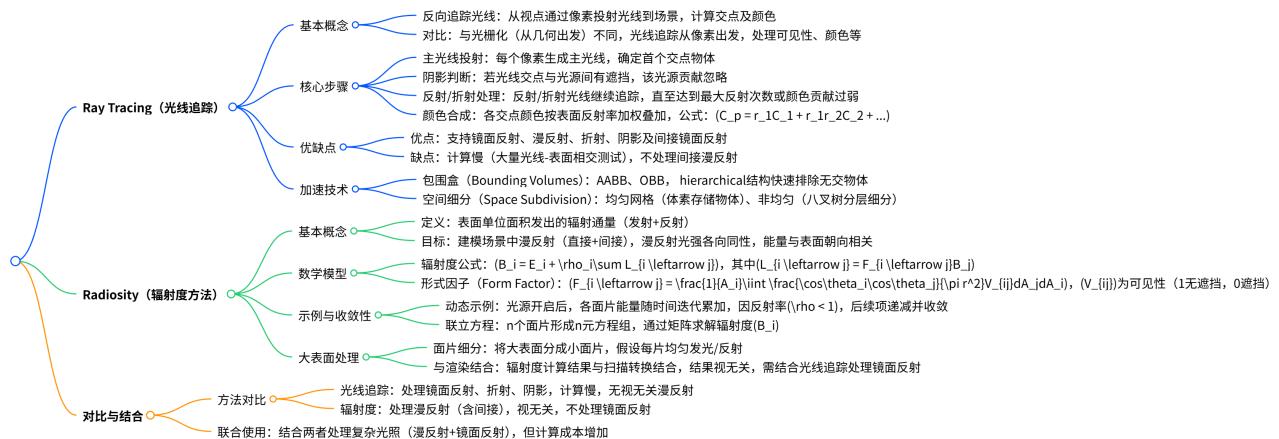


6. Hidden surface removal and shading: Remove part of objects that are obscured by other objects, and then determine what color it is.

隐藏表面去除和阴影：去除被其他物体遮挡的部分物体，然后确定它是什么颜色。

Output: an image.

9. Ray Tracing and Radiosity



豆包
你的 AI 助手，助力每日工作学习

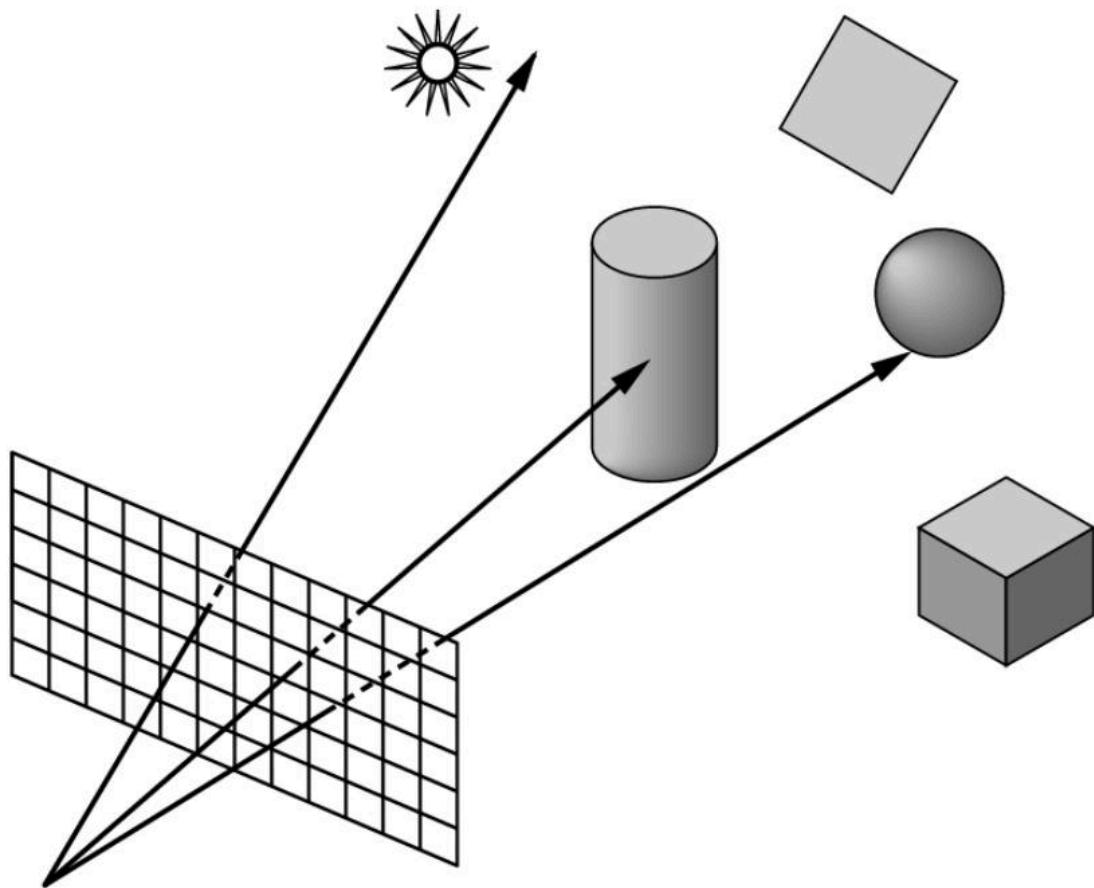
两种方法：

1. **Start from geometry**: 针对每个多边形进行判断，速度快但难以精确计算
2. **Start from pixels**: 针对图像的每个像素进行判断，能更准确地处理

Ray Tracing (Start from pixels)

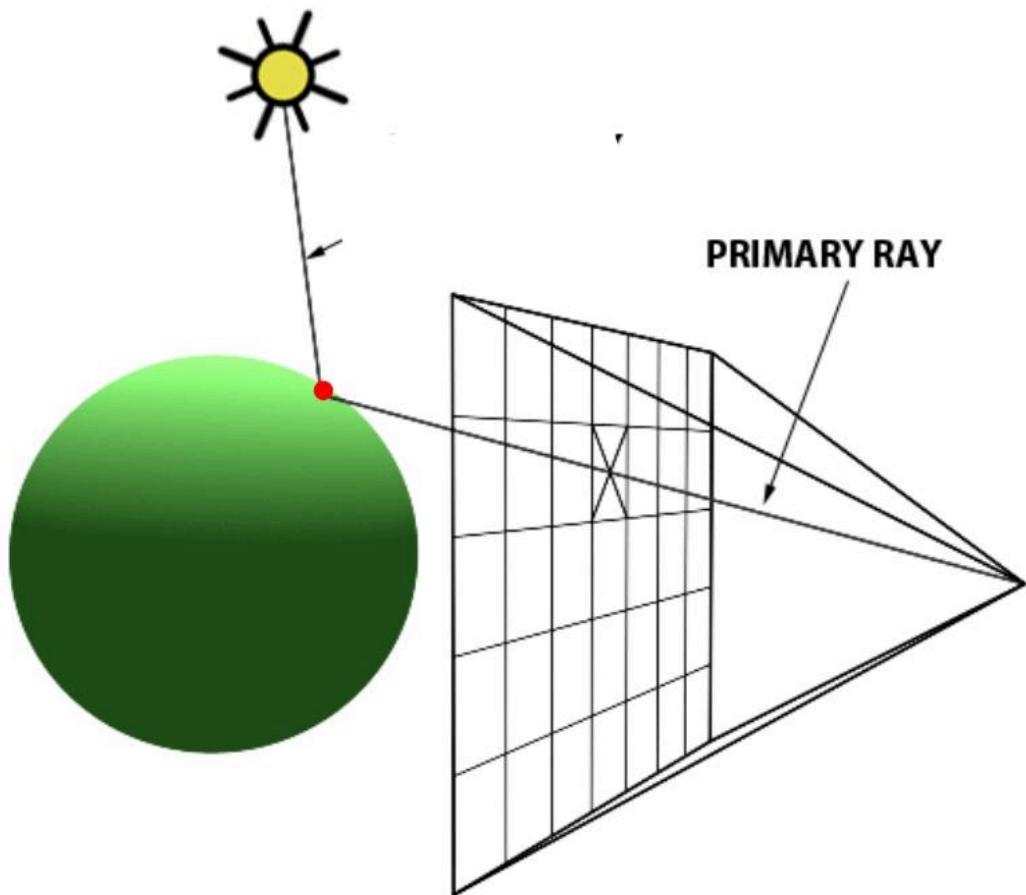
光线追踪通过追踪场景中光线的流动生成图像，典型方式是 **反向 (backward)** 追踪，即从视点（如眼睛或相机）出发，让光线穿过场景中的物体表面，最终指向光源

反向：从视点出发指向光源，而不是正常的从光源出发指向视点，因为这样可以确定计算的每个光线都是有效的，避免从光源出发追踪到了无法传入视点的无效光线

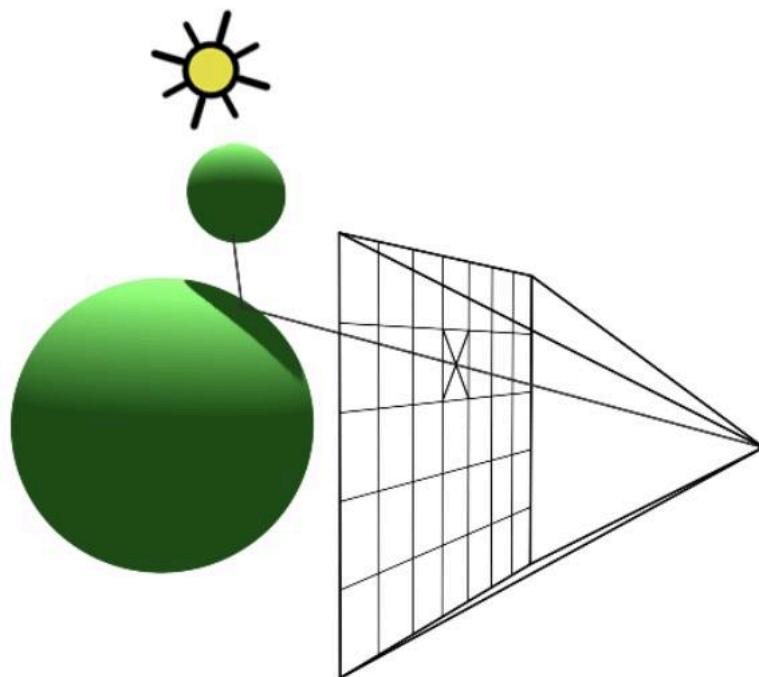


Process

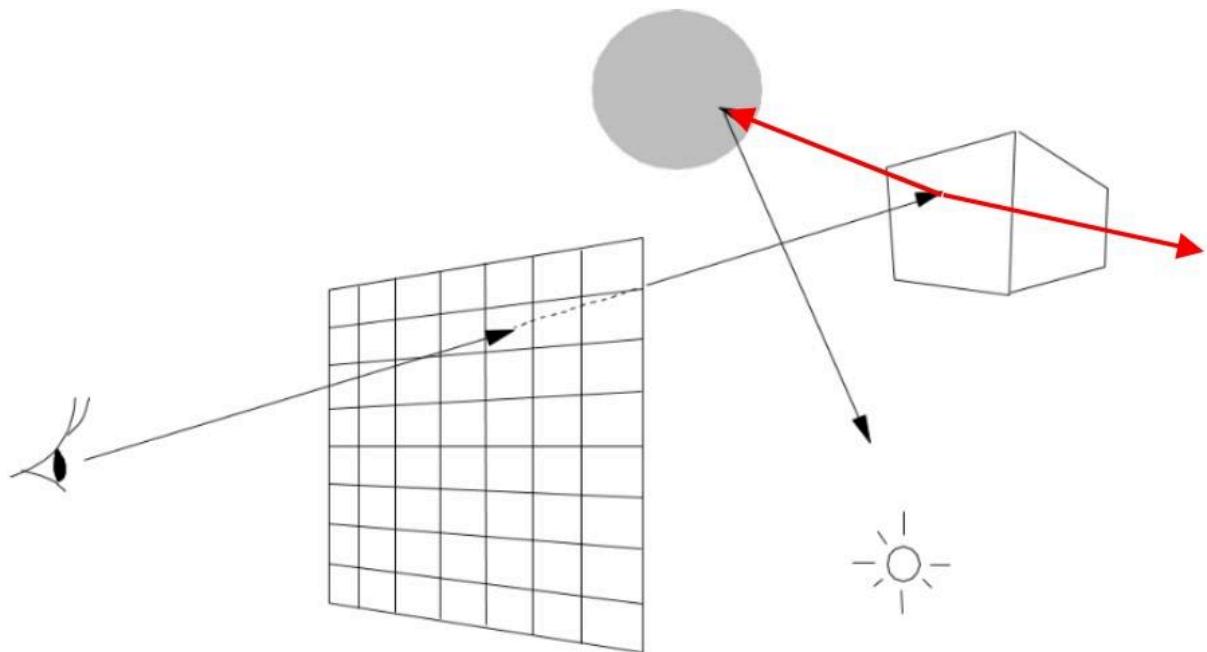
1. **光线投射**: 对于图像平面上的每个像素，从投影中心发射一条光线，穿过该像素进入场景，确定光线在场景中相交的第一个物体
2. **交点与颜色计算**: 确定光线击中物体的交点后，根据物体表面的法线方向和光源信息，计算该交点的颜色值



若该过程中，像素和光源之间存在物体，那么此光源对该像素的影响将不被考虑



光线与物体相交后，继续追踪反射reflected后的光线，若是透明表面则额外追踪折射refracted后的光线

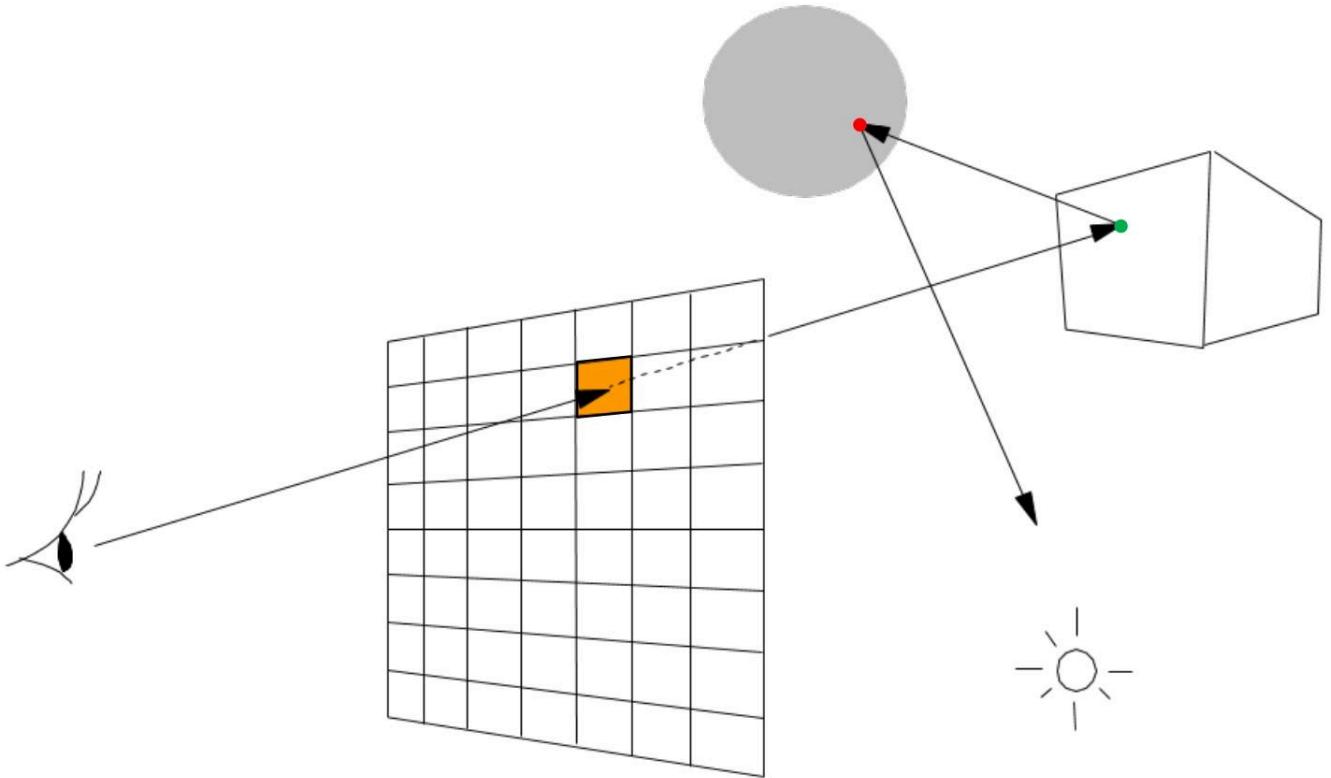


以上过程是迭代的recursive，直到达到预设的最大反射次数 the preset maximum reflection number

Color Formula

从一条光线在场景中所有相交物体上计算出的颜色值，会根据**衰减因子 attenuation factor**（由物体表面反射率决定）进行加权处理。随后，这些经过加权的颜色值被累加，最终生成一个单一的颜色值，作为该像素的最终颜色

$$C_p = r_1 C_1 + r_1 r_2 C_2 + \dots$$



Advantages and Disadvantages

优点 (Advantages)

- **光照效果全面：**光线追踪通过追踪场景中光线的传播，能够考虑**直接镜面反射 Direct specular reflection**（如镜子般规则的反射）、**直接漫反射 Direct diffuse reflection**（表面散射的直接光照）以及**间接镜面反射 Indirect specular reflection**（经一次或多次反射后的镜面效果）。
- **支持复杂光学效果：**能够模拟光的**折射**（如光线穿过玻璃的弯曲）和**阴影**（判断光线是否被遮挡），增强场景真实感。

缺点 (Disadvantages)

- **计算效率低：**由于需要进行大量光线 - 表面相交测试及颜色计算，光线追踪的运算速度较慢，难以快速生成图像。
- **忽略间接漫反射：**无法处理物体表面之间的漫反射（如墙面间的漫反射光照传递），这一局限将在辐射度 (Radiosity) 方法中进一步探讨。

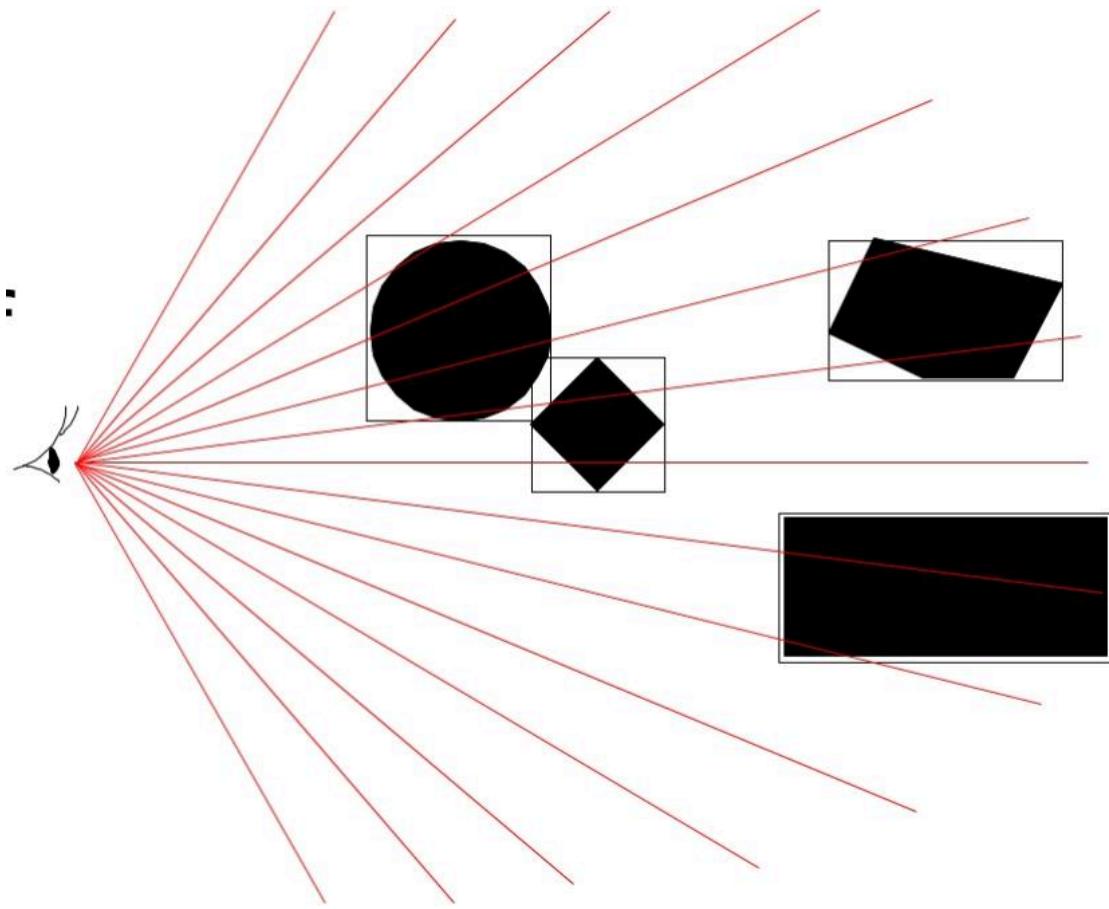
Acceleration

两种加速方法：

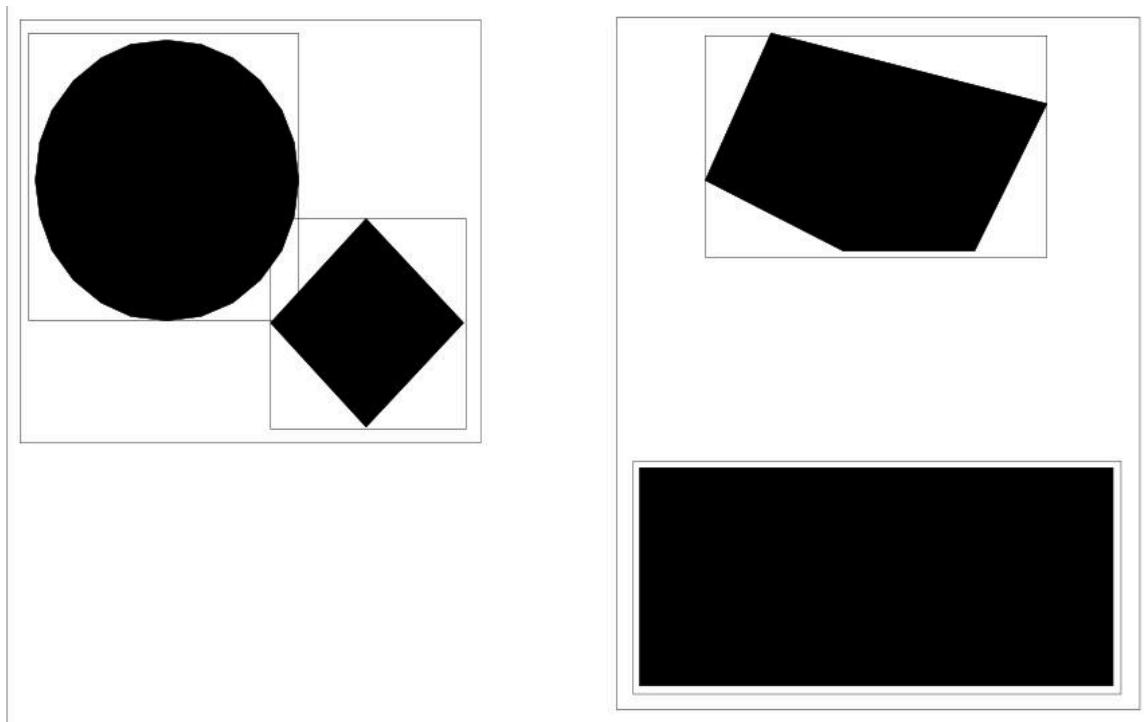
Bounding volumes 包围盒

为场景中每个物体构造包围盒，常见类型有**轴对齐包围盒 (AABBs)** 和 **定向包围盒 (OBBs)**

该方法无需计算光线与每个物体的直接相交，而是先检查光线是否与物体的包围盒相交 intersect。可快速排除不相交的物体，减少计算量

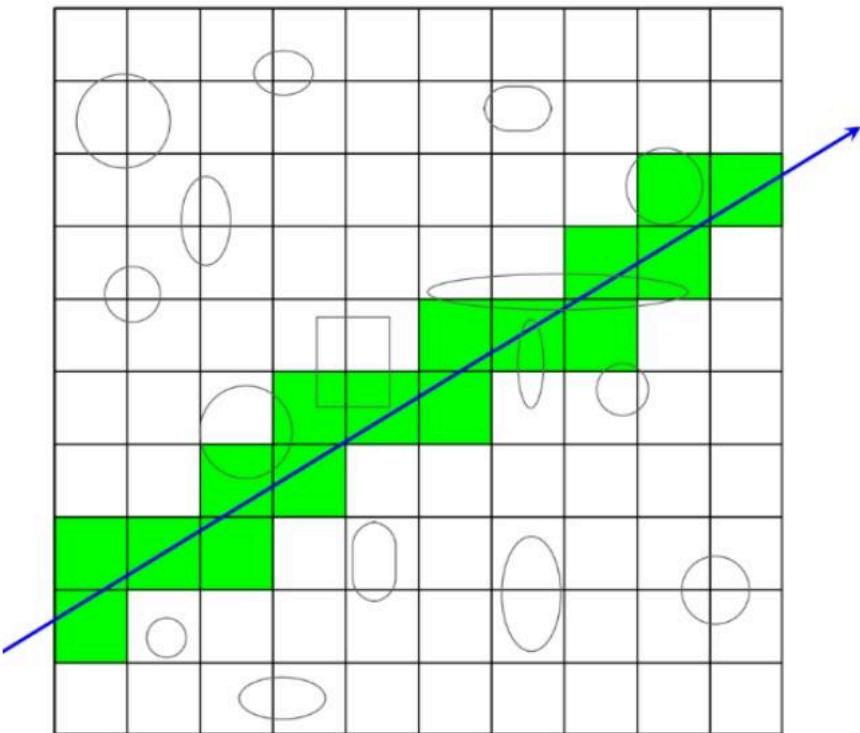


进一步可构建**层次化包围盒 hierarchical bounding volume**，若光线在某一层级level不与层次化包围盒相交，那么它必然不会与该层次化包围盒所包含的子体积内的任何物体相交

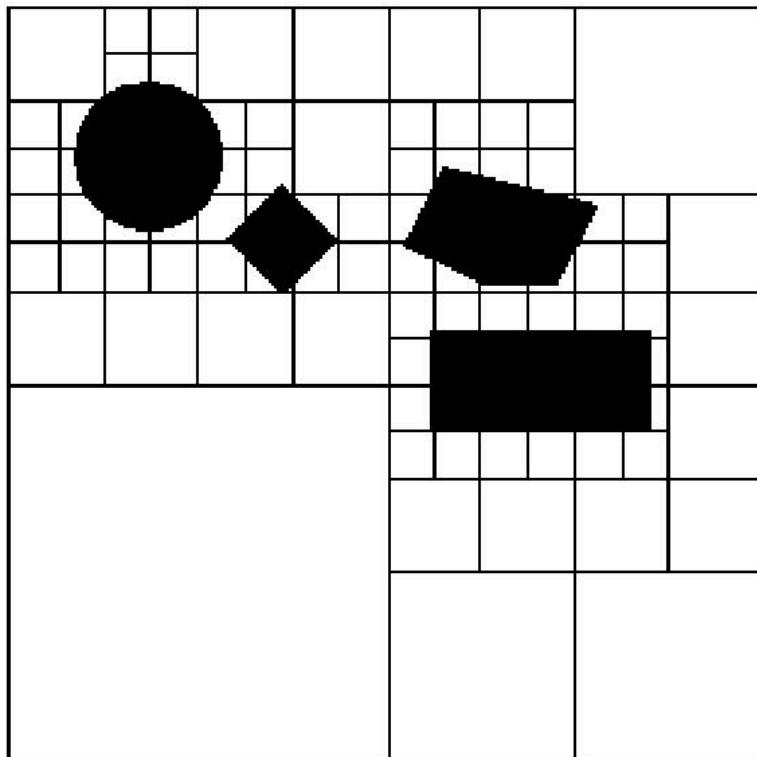


空间细分 (Space Subdivision)

均匀空间细分 (Uniform space subdivision)：在每个体素 (voxel) 中，存储与该体素相交的物体列表。当计算光线追踪时，仅对存储的物体进行相交测试

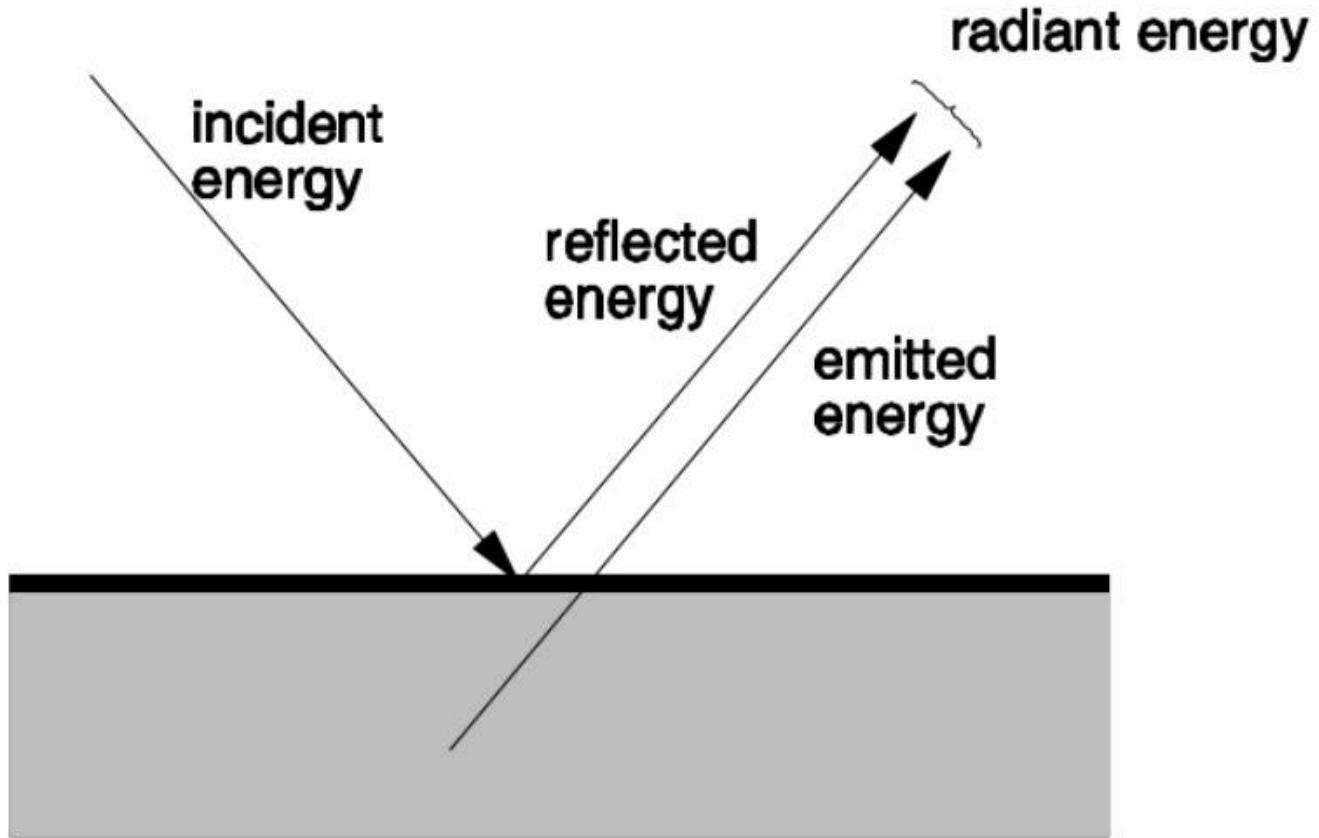


非均匀空间细分 (Non - uniform space subdivision): 通过层次化细分场景 Subdivides the scene hierarchically (如八叉树) 以减少体素总量



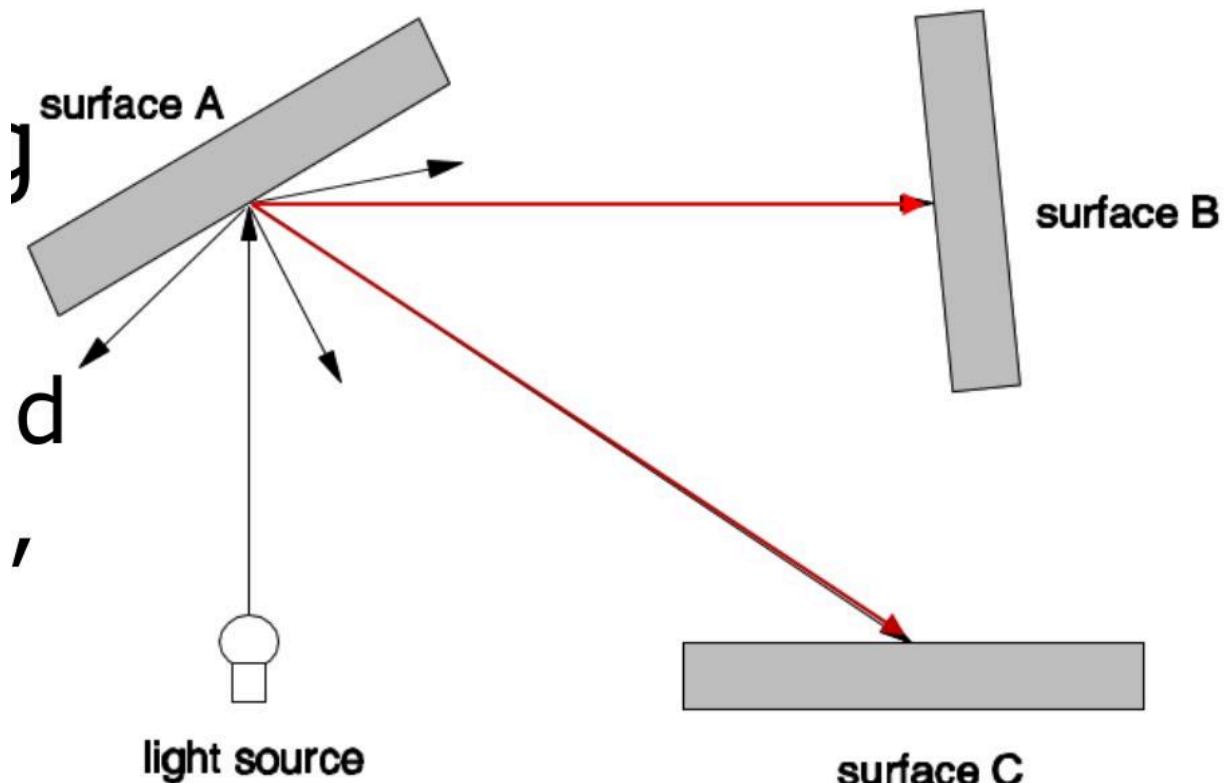
Radiosity (Start from geometry)

辐射度是指离开单位面积表面的辐射通量，包括表面自身发射的能量和反射的能量



方法旨在模拟场景中表面之间的漫反射 **Diffuse Reflection**（包括直接漫反射和间接漫反射）。在漫反射中，光向所有方向以相等强度反射，但一个表面接收的光能量取决于其相对于光源的朝向

例如，图中表面 A 向表面 B 和 C 反射光的强度相等，但由于 B 的入射角更小，B 接收的光能量比 C 更高



面片*i*的辐射度等于其自身的光能量加上反射率乘上其他各面片传递来的光能量

$$B_i = E_i + \rho_i \sum_{j=all\ patches} L_{i \leftarrow j}$$

对于大表面，由于入射角的变化，其辐射度值可能从一端到另一端发生显著变化。为提升计算准确性，场景中的表面被划分为若干小面片。

辐射度方法计算出的辐射度值具有**与视角无关**的特性。这使得它无法处理**镜面反射** (specular reflection)，因为镜面反射效果依赖于观察视角，需特定方向的光线交互才能准确呈现

为同时处理场景中的漫反射（辐射度方法擅长）和镜面反射（需视角相关计算），可将辐射度方法与光线追踪方法结合。但这种结合会进一步增加计算时间

10. Real Time Rendering



豆包
你的 AI 助手, 助力每日工作学习

在实时 3D 应用（如虚拟环境、电脑游戏）中，交互性是关键因素。用户执行操作时，期望系统能像现实生活中的互动一样，立即做出响应。因此渲染时间非常重要。

为实现实时反馈，需在给定时间（即帧时间）内完成渲染。

帧时间指两连续图像帧之间的间隔。例如，每秒生成 10 帧图像时，帧时间为 0.1 秒（作为对比，电影通常每秒 24 帧）

因此在时间关键型渲染**Time - Critical Rendering**中，需要用图像精度换取速度。即在给定的一帧时间内，尽可能渲染出准确的图像，平衡速度与精度的关系

为达成上述目标，需满足以下三点：

- **渐进式渲染技术 (A progressive rendering technique)**: 通过逐步增加图像细节的方式，在有限时间内先呈现低精度版本，再逐步细化
- **评估对象视觉质量的方法 (A way to estimate the visual quality of objects)**: 用于判断场景中各对象所需的细节程度，确定哪些部分需精细渲染，哪些可简化
- **评估对象渲染成本的方法 (A way to estimate the rendering costs of objects)**: 用于衡量渲染各对象所需的时间、计算资源等，以便合理分配时间，确保在帧时间内完成渲染

Progressive Rendering

一般而言，渲染时间大致与要渲染的图元 Primitives（通常为三角形）数量成正比 proportional。因此，为减少渲染时间，可以通过减少表示对象的三角形数量来实现

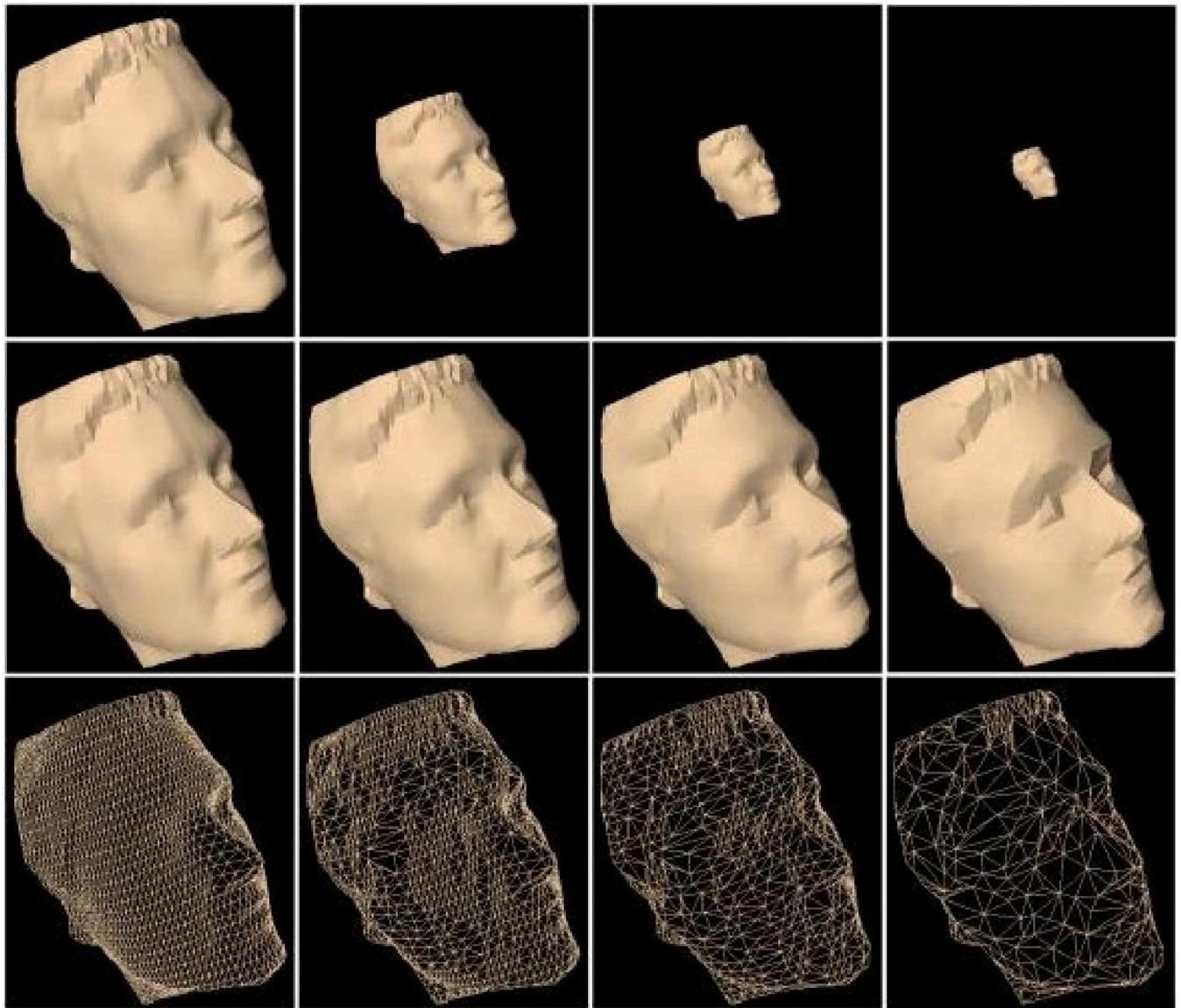
此外，如果对象离观察者较远，其细节对观察者而言可能并不清晰，可以简化

考虑引入细节层次技术 Levels of Details (LoDs)

Discrete LoDs

离散细节层次 (Discrete LoDs)，为每个对象预算计算 pre-compute 不同分辨率 resolutions (即不同细节层次，LoDs) 的模型

当必要时直接切换



优势：高效，渲染时只需简单选择最适合的模型进行渲染。

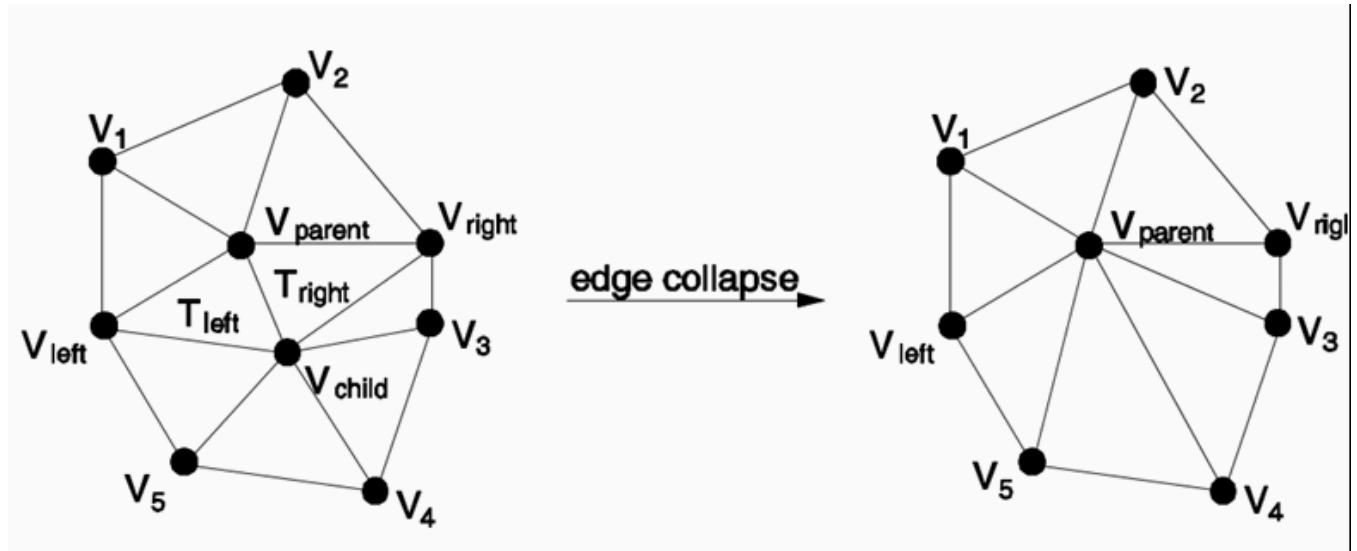
劣势：

- 难以确定所需的 LoD 数量；
- 切换模型时可能出现视觉伪像 Visual artifacts

Progressive meshes

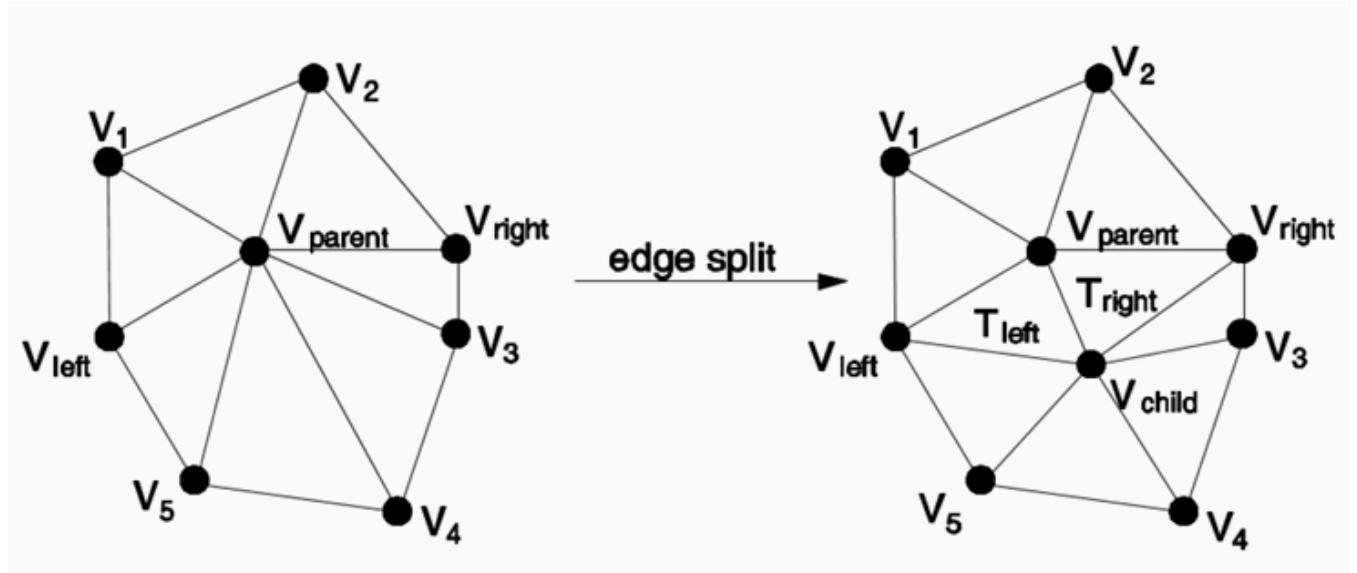
渐进网格 (Progressive meshes)，通过“边折叠 edge collapse” 的操作来降低对象模型的分辨率

每次边折叠操作会从模型中移除两个三角形，逐步小幅度降低模型的分辨率，持续降低分辨率直至达到最低分辨率，这种最低分辨率的模型被称为 “基础网格 base mesh”

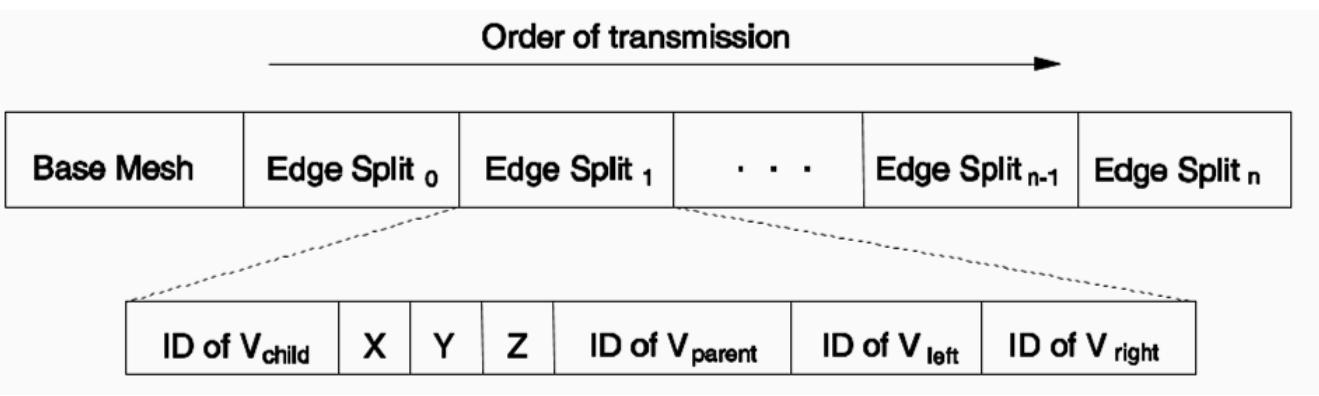


此外，通过称为“顶点分裂 vertex split” 的反向操作逐渐增加分辨率，原理是增加两个三角形

从基础网格base mesh开始不断执行顶点分裂操作即可恢复原始模型



模型可由基础网格以及一系列分裂操作组成，这种渐进网格结构支持高效存储和渐进传输，可根据需求逐步恢复模型细节



然而，其对大型模型（如复杂场景或高精度模型）的处理效果可能不佳

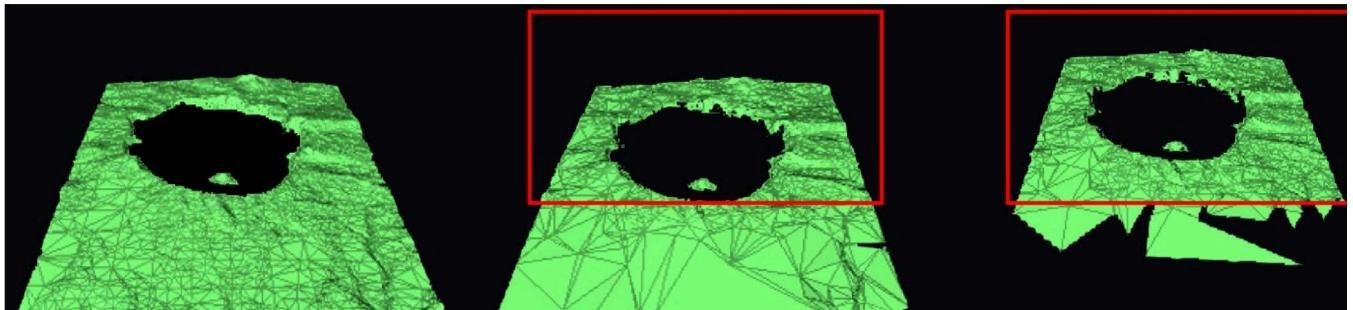
Selective refinement

对于大型模型（如景观模型），用户通常只会详细观察其一小部分，而非整个模型

而渐进网格以与视图无关的方式构建，即模型整体同时处于高分辨率或低分辨率状态，无法实现“仅用户关注的局部区域高分辨率，其余区域低分辨率”的灵活渲染

若能选择性地细化感兴趣的局部区域模型分辨率，同时保持其余部分低分辨率，便可优化三角形数量，提升渲染性能。

例如，专注于观察模型的某一局部时，仅对该局部细化，其余部分保持低分辨率，减少整体渲染的三角形数量，从而提高效率



但是执行边折叠（edge collapses）或顶点分裂（vertex splits）操作时，相邻三角形区域间存在强依赖性 strong dependency。一个区域的调整会影响到相邻区域，难以孤立地对局部进行独立操作，增加了实现的复杂性

Visual Quality Estimation

视觉质量的衡量依据

- **模型细节层次 detail level**（或图元 primitives 数量）：渲染时使用的模型细节越多（图元数量越多），通常视觉质量越高。
- **对象与观察者的距离**：对象离观察者越近，需更高细节以保证视觉质量；距离远时，细节可适当简化。
- **对象在屏幕上的投影大小 projected size**：投影大的对象需更精细的细节呈现，投影小的对象细节要求相对较低。

另外移动速度 moving speed 和角距离 angular distance 也有影响

Rendering Cost Estimation

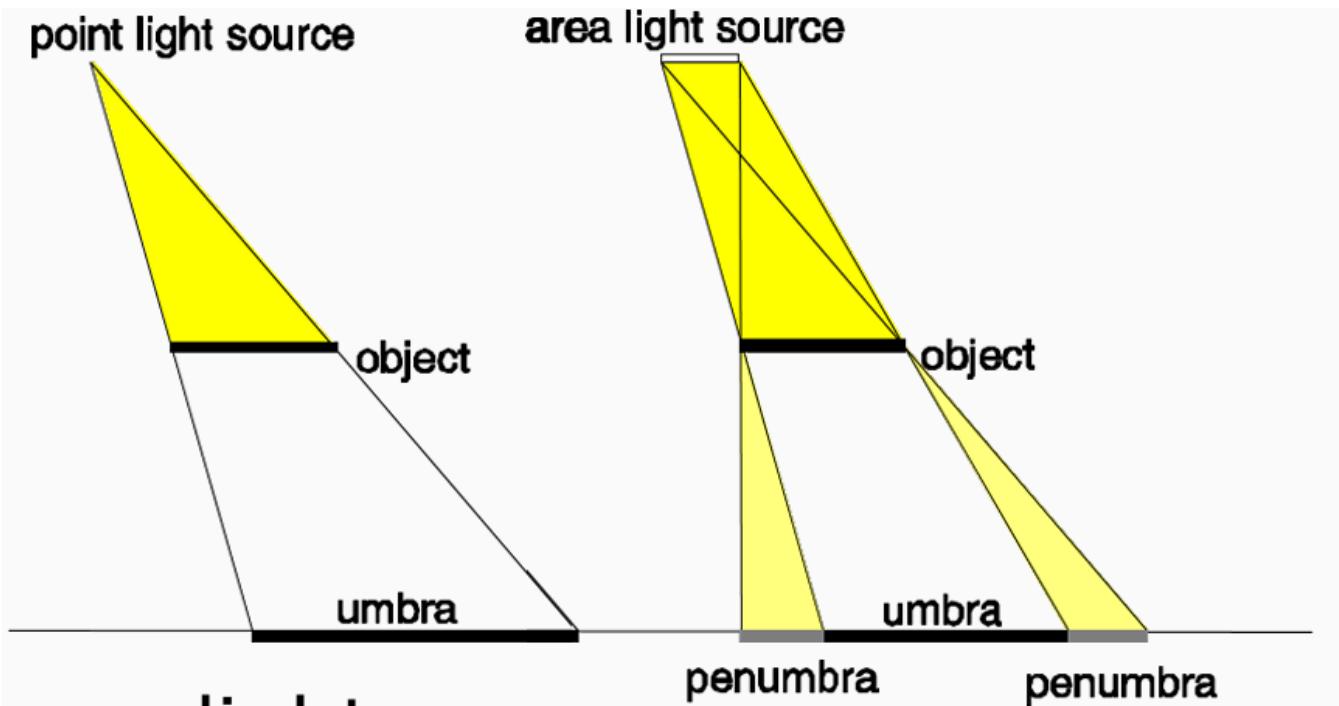
我们需要一种高效的评估方法，使得能够在短时间内确定渲染的质量并完成渲染过程

Shadows

阴影能够提供对象与光源之间几何关系的信息，增强场景的立体感与真实感

硬阴影 (Hard shadows): 仅包含本影 (umbra) 区域。由点光源 (point light source) 产生，阴影边界清晰锐利

软阴影 (Soft shadows): 包含本影 (umbra) 和半影 (penumbra) 区域。由面光源 (area light source) 产生，阴影边界柔和

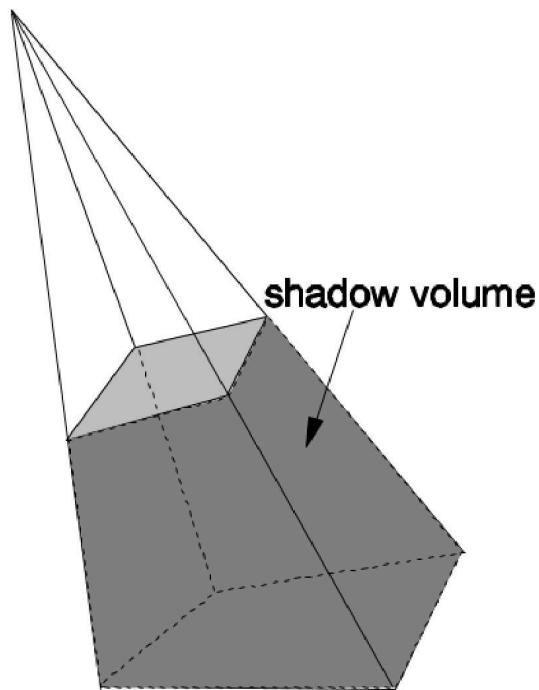


Hard Shadow

阴影体积法 Shadow Volume method

确定处于阴影中的区域（体积），并用多边形对这些区域进行界定。随后检查某一点是否位于“阴影体积”内，来确定该点是否处于阴影中

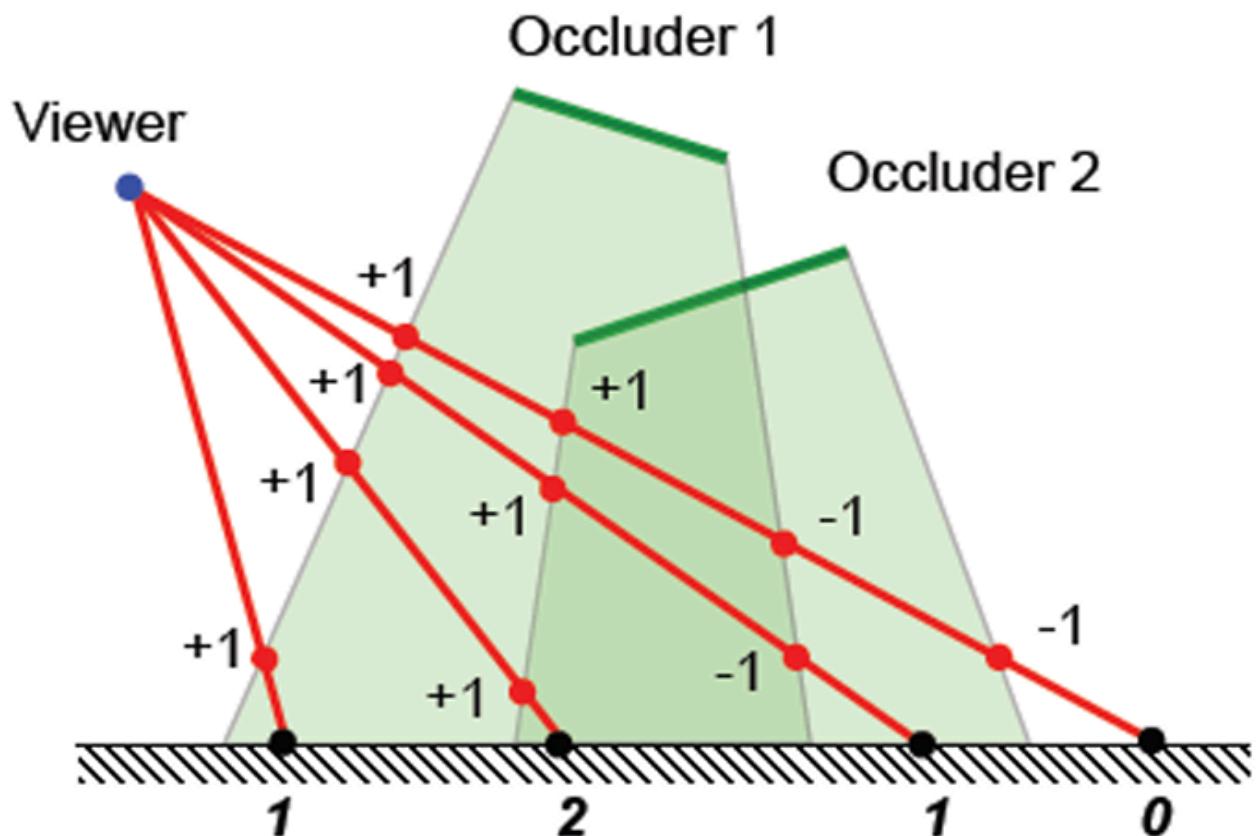
总体而言，阴影体积法更为复杂，因为它需要处理额外的几何信息（如阴影体积的多边形界定），增加了计算与处理的难度



具体方法：

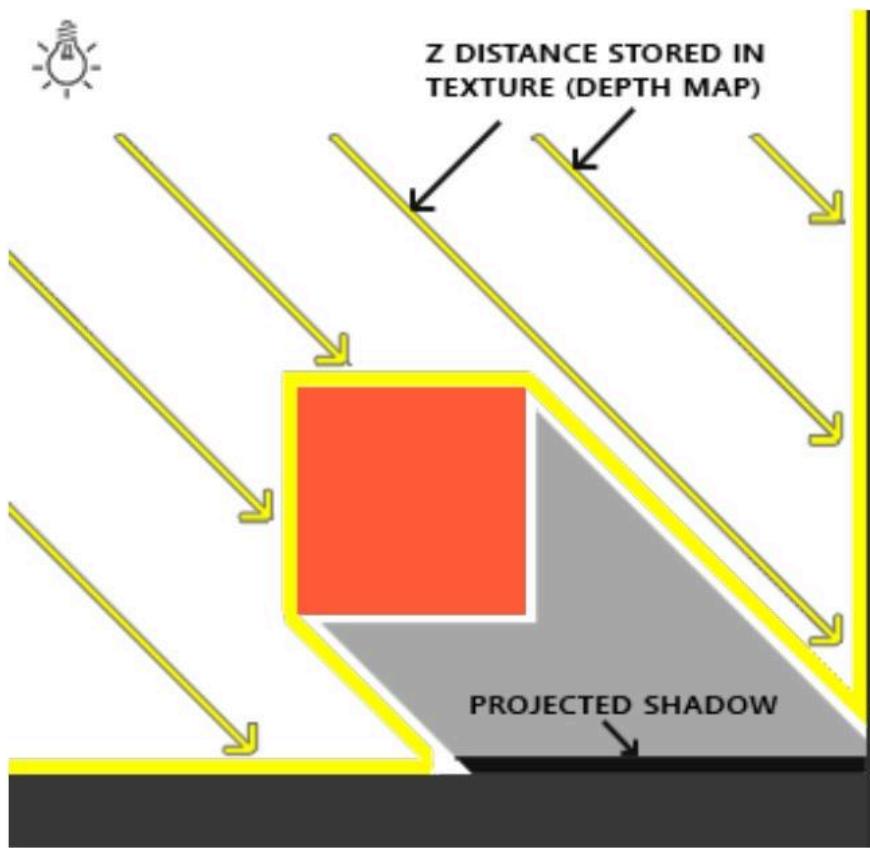
1. **连线**：从观察者视点（Viewer）向每个感兴趣的点连接一条线
2. **计数初始化**：设置一个计数器，初始值为 0。
3. 穿越多边形计数
 - 当连线穿过阴影体积的 **正面多边形** (**front - facing polygon**) 时，计数器加 1
 - 当连线穿过阴影体积的 **背面多边形** (**back - facing polygon**) 时，计数器减 1
4. **判断阴影**：若最终计数器值为正数，则该点处于阴影中

● Light source



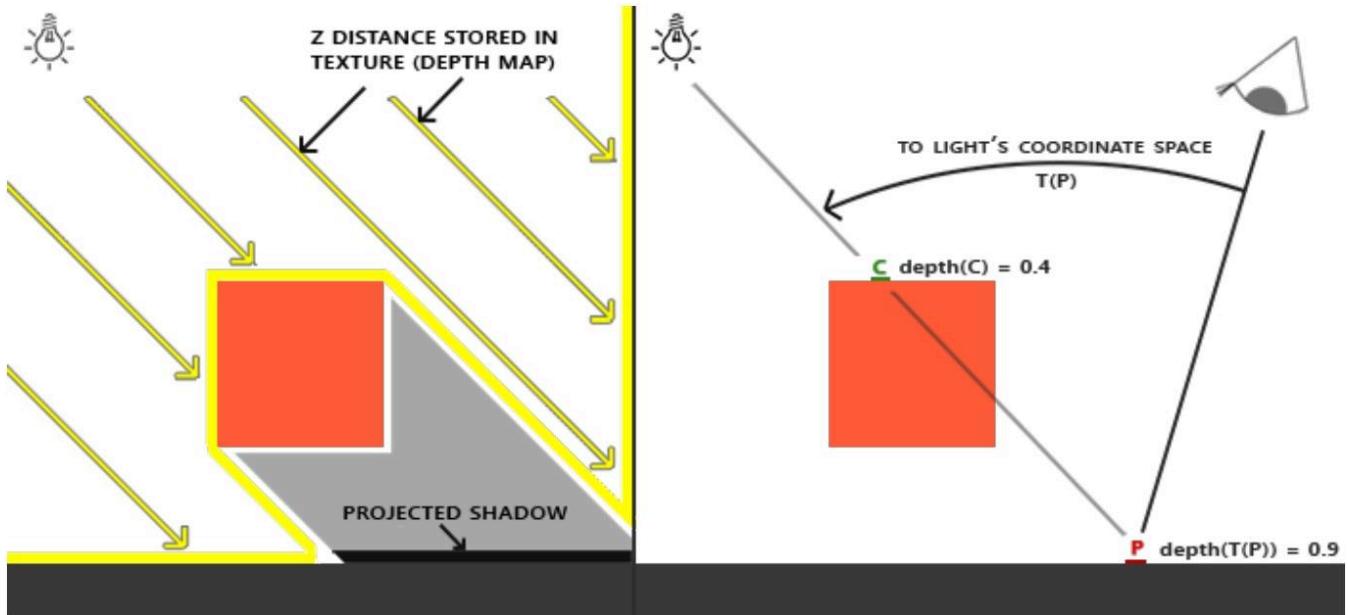
阴影贴图法 (Shadow Map methods)

- 生成阴影贴图：从光源视角，使用 z - 缓冲方法渲染深度图（即阴影贴图）。该图记录了光源在每个像素位置能看到的物体深度（离光源的距离），标识出光源可见的物体
 - 从光源看，哪些地方有物体挡着



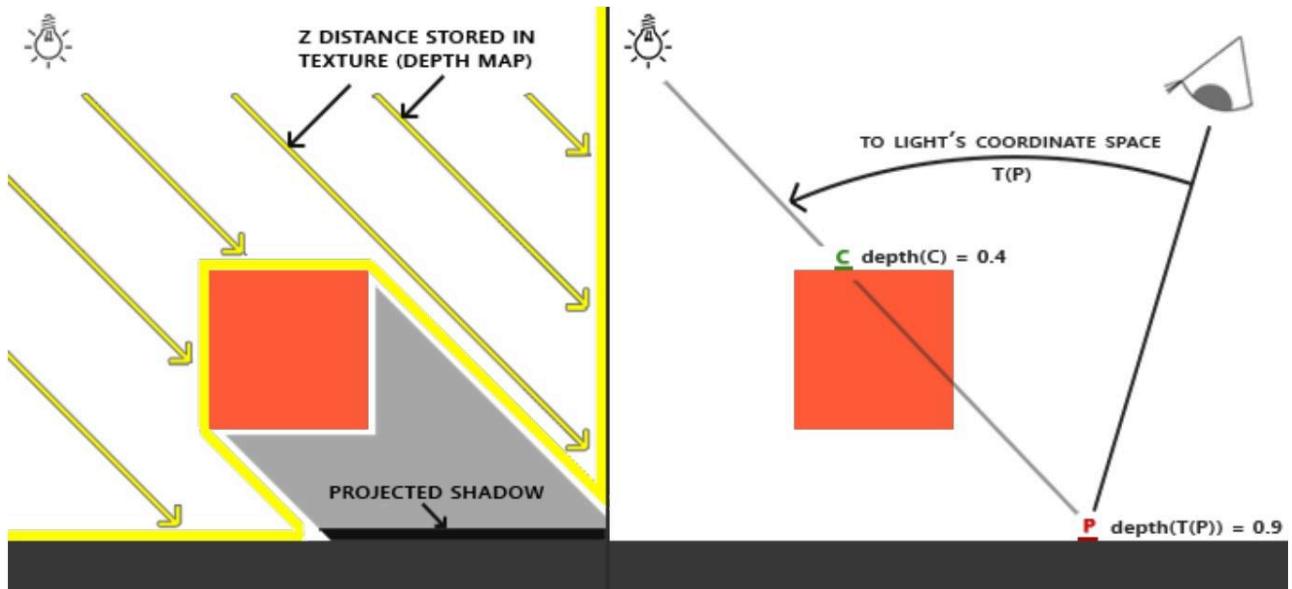
2. 转换深度值：从观察者视角生成输出图像时，对每个计算的像素，将其在观察者空间的深度值转换到光源空间

- 从光源角度看观察者看到的点，也即把观察者眼中的点，“翻译”成光源眼中的距离

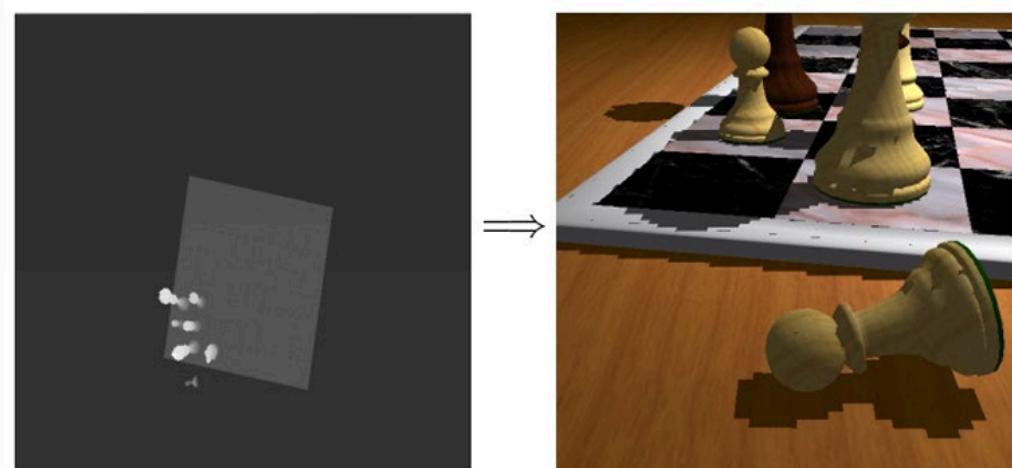


3. 判断阴影：将转换后的深度值与阴影贴图中对应位置的深度值比较。若转换后的深度值大于阴影贴图中的深度值，说明该点被其他物体遮挡，处于阴影中

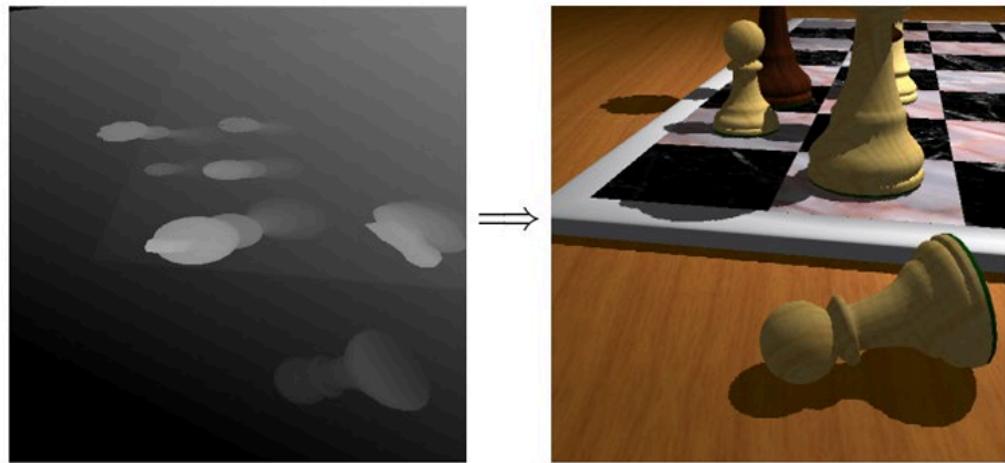
- 转换后的距离和阴影贴图里对应的距离比一比，如果转换后的距离更大，说明从光源看，这个点被更近的物体挡住了，光照不到，这个点就处于阴影中



标准阴影贴图法的局限：存在“混叠（aliasing）”问题，即图像可能出现锯齿、模糊等不清晰现象



为改进，先从观察者视角对物体进行透视变换可改善混叠问题

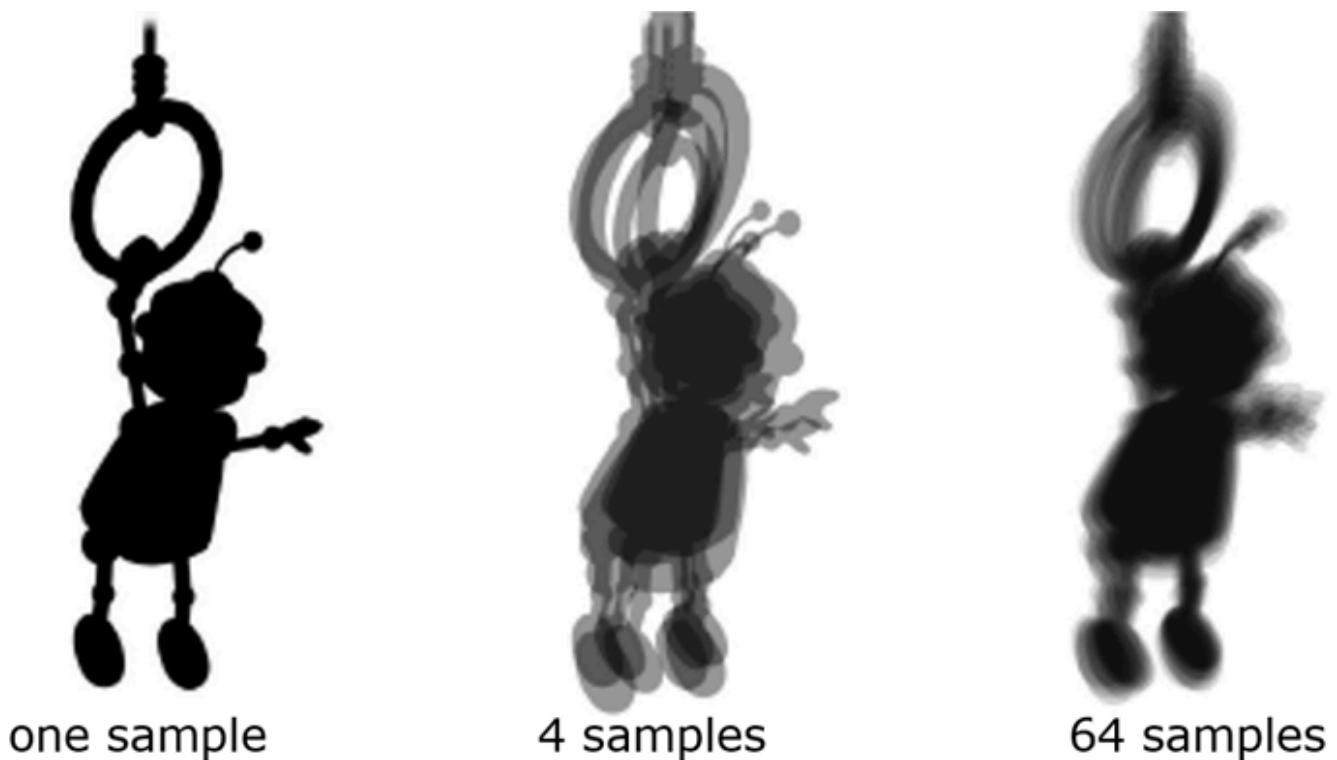


**perspective shadow map
and resulting image**

Soft Shadow

组合点光源法 Combining point light sources

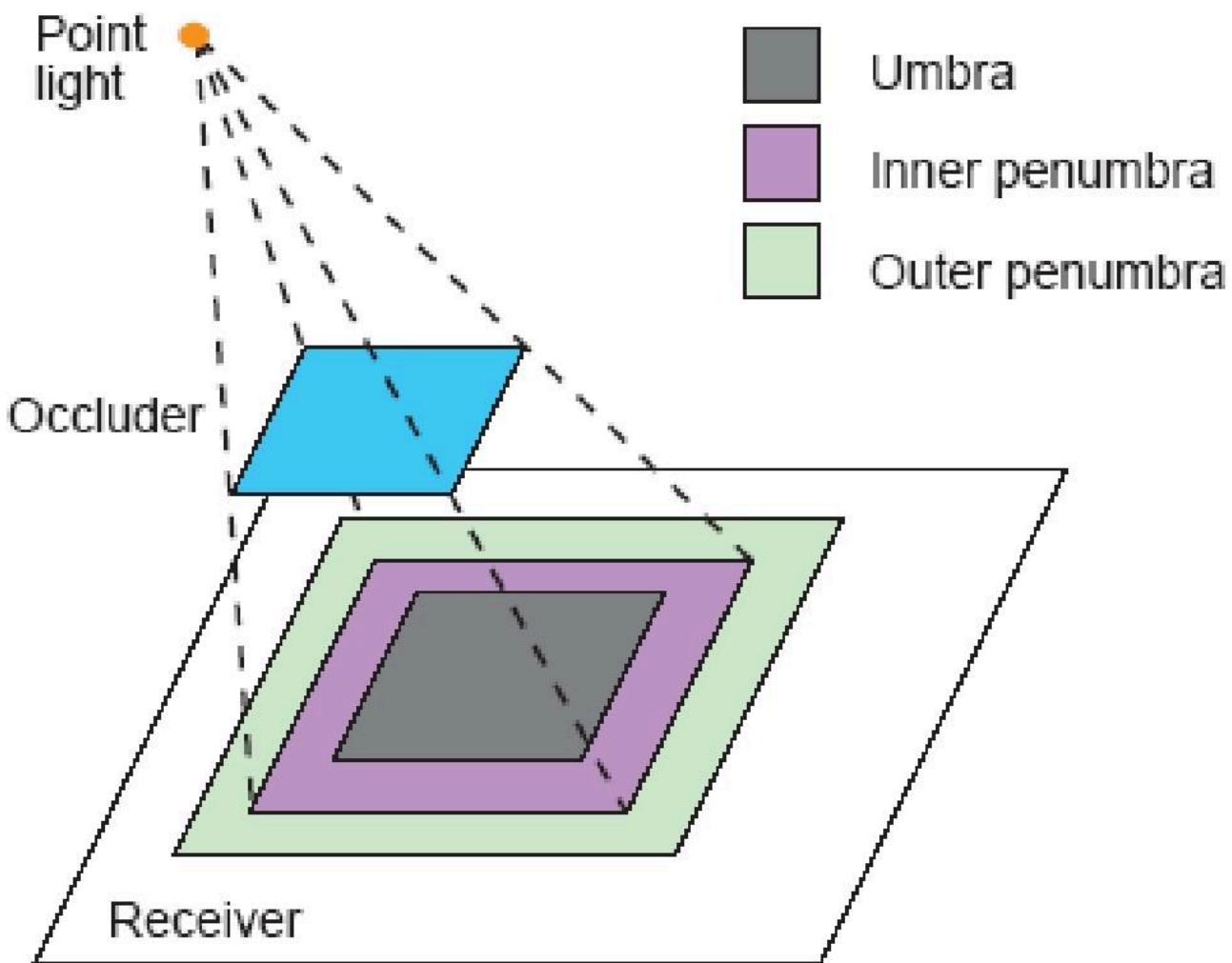
1. 多位置采样光源：对光源在不同位置进行多次采样，为每个光源采样点生成一个阴影贴图。将面光源视为多个点光源的集合，每个点光源独立生成阴影贴图
2. 合成衰减图：把这些阴影贴图组合起来，形成一张衰减图 attenuation map
3. 调整最终光照：在最终图像渲染阶段，利用这张衰减图来调整每个阴影点的光照强度 illumination



单样本软阴影 Single sample soft shadow

1. 生成标准阴影贴图：首先从光源视角生成一张标准的阴影贴图 shadow map，确定基本的阴影范围

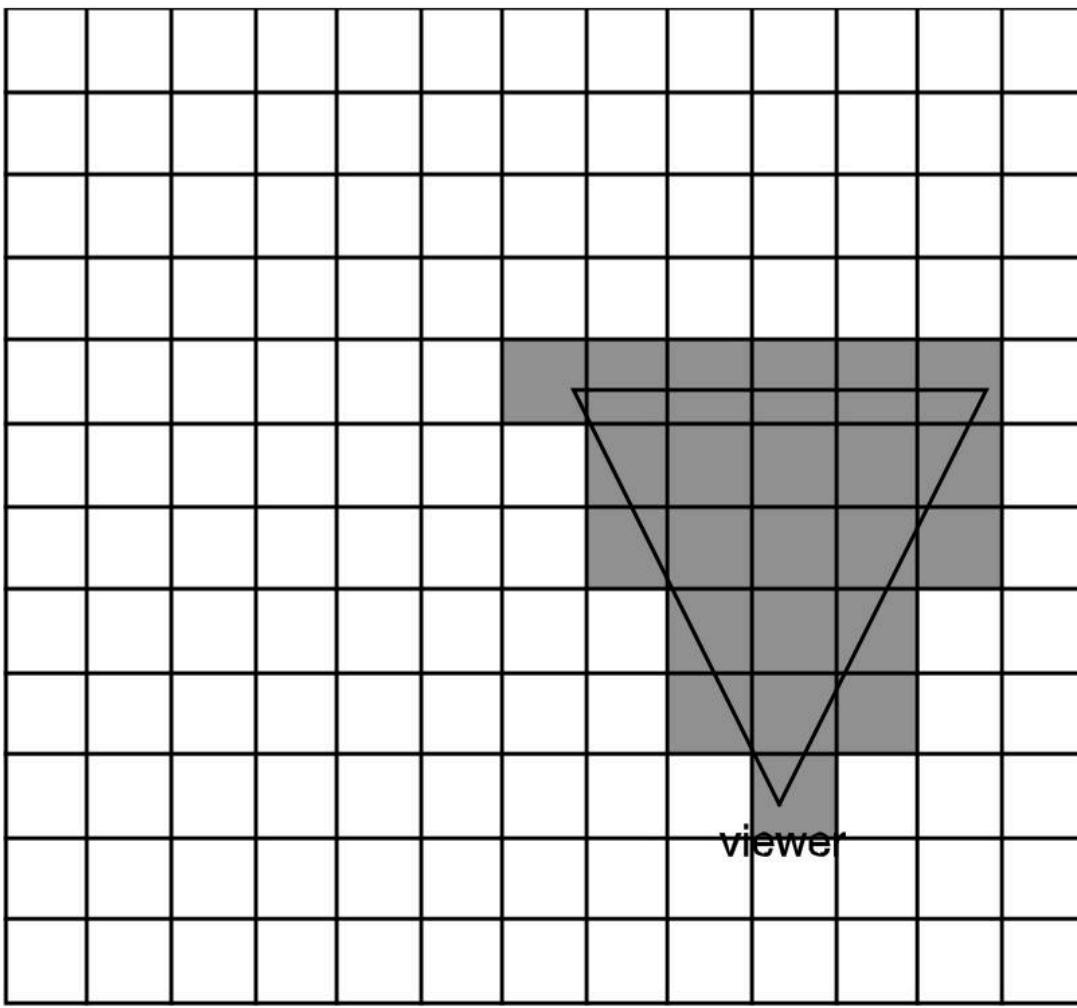
2. 创建半影区域：基于遮挡物（Occluder）的深度，在阴影边界之外创建区域，形成半影区
3. 生成衰减因子 **attenuation factor**：从内半影区的内边缘（值为 1，表示完全遮挡开始减弱）到外半影区的外边缘（值为 0，表示无遮挡），生成一个衰减因子。这个因子用于在渲染时调整光照强度，使阴影边界呈现出柔和过渡的效果，模拟软阴影的半影区域



Visible Object Determination

在包含大量对象的大型场景中，若为渲染单张图像而处理所有对象，效率会极低。因此，需快速识别潜在可见的对象。

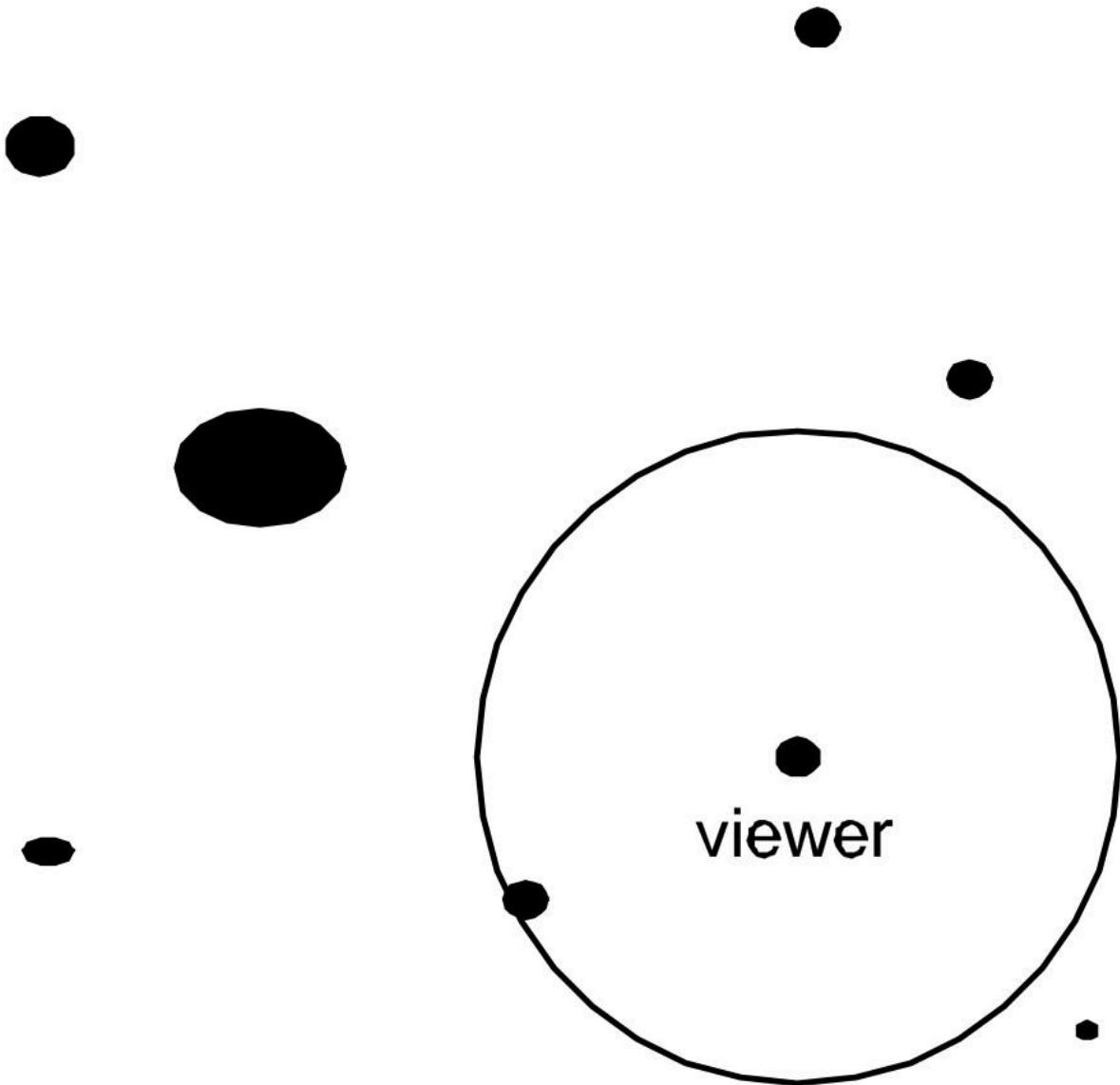
一种方法是将场景划分为小单元格，确定所有与视图区域重叠的单元格，仅考虑与这些重叠单元格相交的对象进行渲染。这样可减少处理对象的数量，提升渲染效率



另一种为感兴趣区域（Area of Interest, AOI）

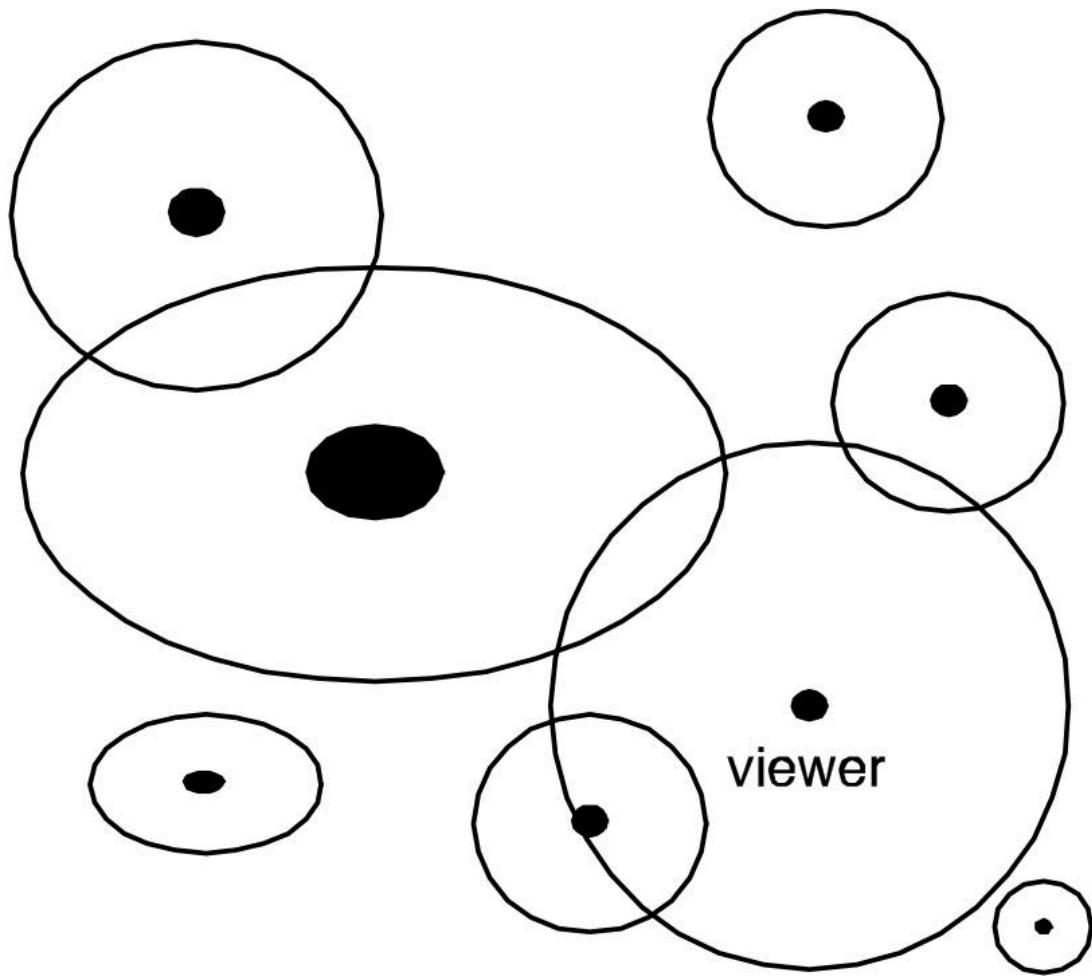
划分出观察者的感兴趣区域，只有位于观察者 AOI 内的对象才被视为可见对象

通过比较观察者与对象中心的欧几里得距离 Euclidean Distance 和 AOI 的半径来确定可见性。若距离小于 AOI 半径，对象在 AOI 内，为可见；反之则不可见

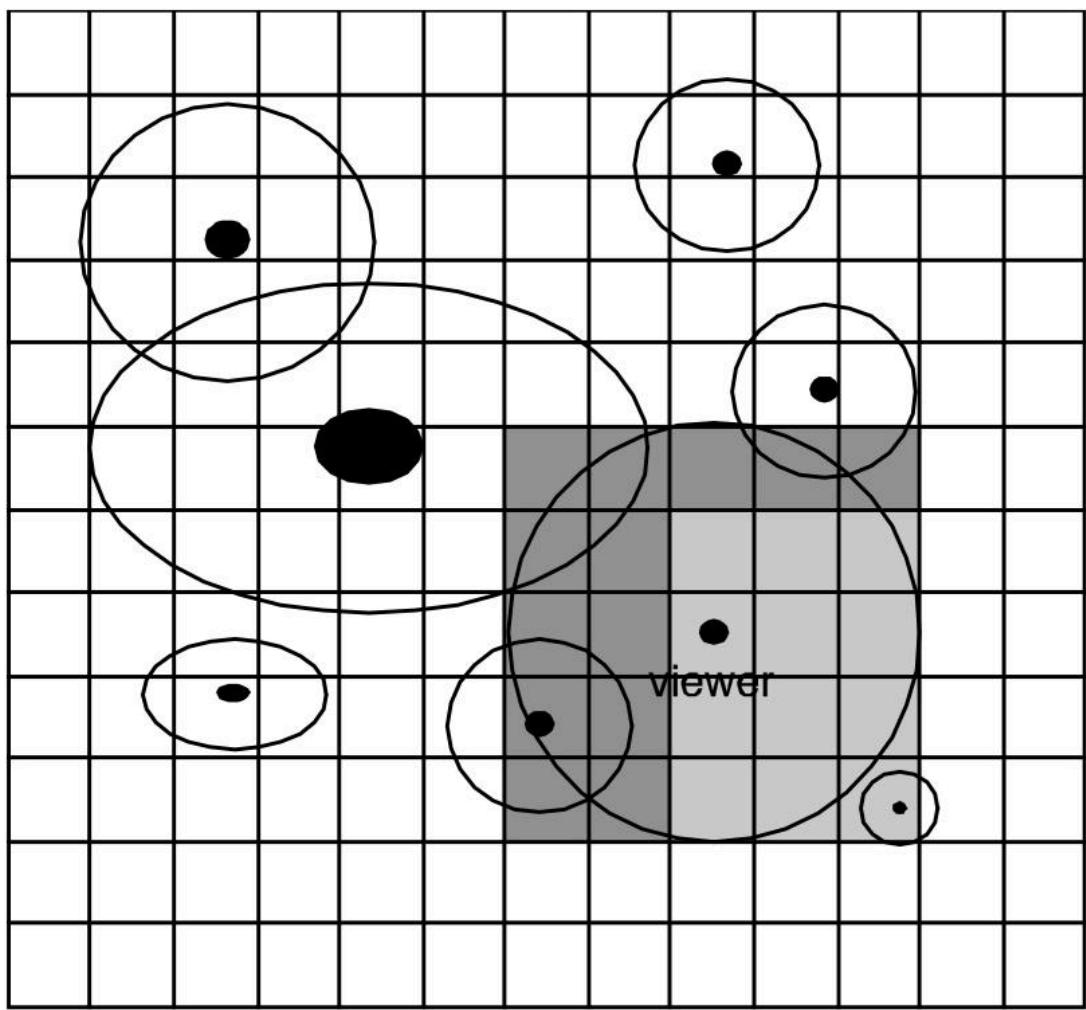


可进一步扩展，为对象也生成AOI，仅对象自身 AOI 与观察者 AOI 重叠的对象，才被视为对观察者可见

通过比较观察者与对象中心的欧几里得距离和两者半径之和来确定可见性。若距离小于两半径之和，说明两者 AOI 重叠，对象可见；反之则不可见



更进一步地，将场景细分为小单元格cells，每个单元格维护一个列表，记录所有 AOI 与该单元格重叠的对象
通过检查观察者 AOI 所覆盖的每个单元格的列表，可高效确定可见对象



11. GPU and Computer Animation



豆包

你的 AI 助手, 助力每日工作学习

GPU: 位于显卡上的快速数学计算处理器

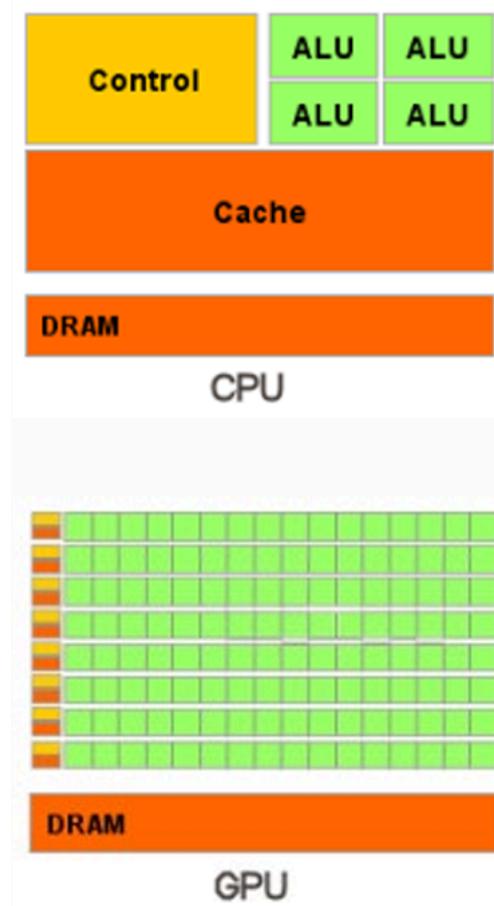
CPU vs. GPU

CPU 特性:

- 擅长控制密集型 Control-intensive tasks 任务, 但不擅长数据密集型 Data-intensive tasks 任务
- 算术单元 (ALU) 少, 空间有限
- 针对低延迟优化, 非高带宽设计

GPU 特性：

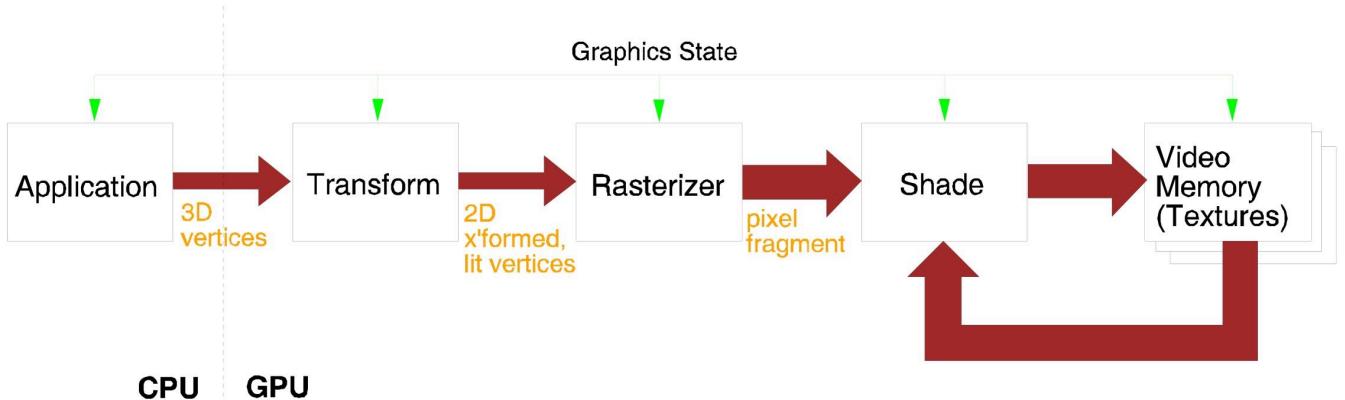
- 支持大量计算与并行处理。
- 控制简单，具备多阶段处理能力。
- 具有延迟容忍性。



高度可编程（highly programmable），高精度计算（high precision computations），可广泛应用于多种计算任务，而非局限于图形处理

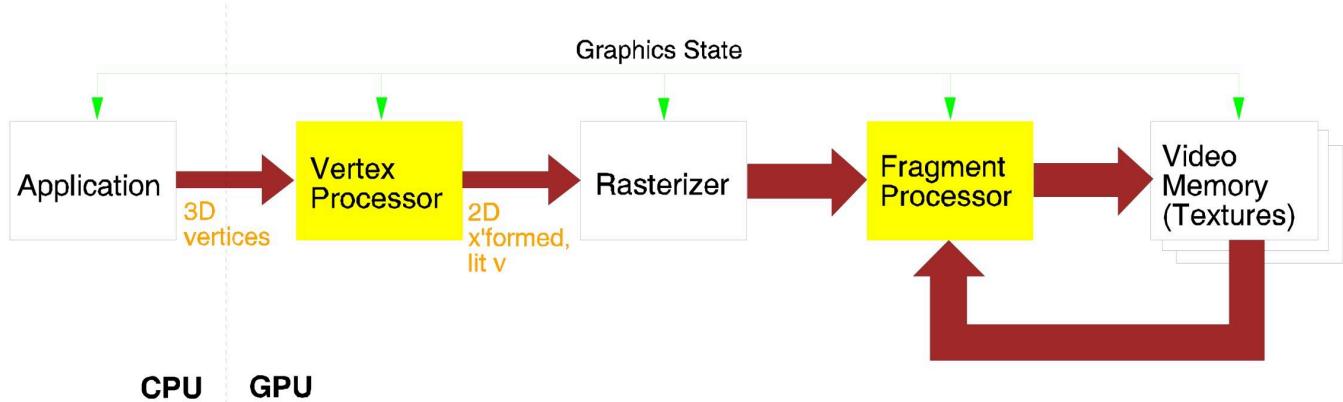
GPU Architecture

The traditional hardware graphics pipeline



1. **Application (CPU 处理)**: 应用程序生成 3D 顶点 (3D vertices), 由 CPU 负责处理
2. **Transform (GPU 处理)**: GPU 将 3D 顶点转换为 2D 变换且光照处理后的顶点 (2D transformed, lit vertices)
3. **Rasterizer (GPU 处理)**: 将几何图形转换为像素片段 (pixel fragment)
4. **Shade (GPU 处理)**: 对像素片段进行着色处理
5. **Video Memory (Textures)**: 着色后的结果存储到视频内存 (纹理) 中

The advanced hardware graphics pipeline



Vertex processors (顶点处理器): 就像给 3D 模型 “摆姿势”

- **Transformation (变换)**: 对顶点进行坐标变换。
- **Back - face removal (背面剔除)**: 剔除不可见的背面顶点, 提升渲染效率。
- **Per - vertex lighting computation (逐顶点光照计算)**: 计算每个顶点的光照效果。

Rasterizer (光栅化器) 负责把几何图形 “拆解” 成像素

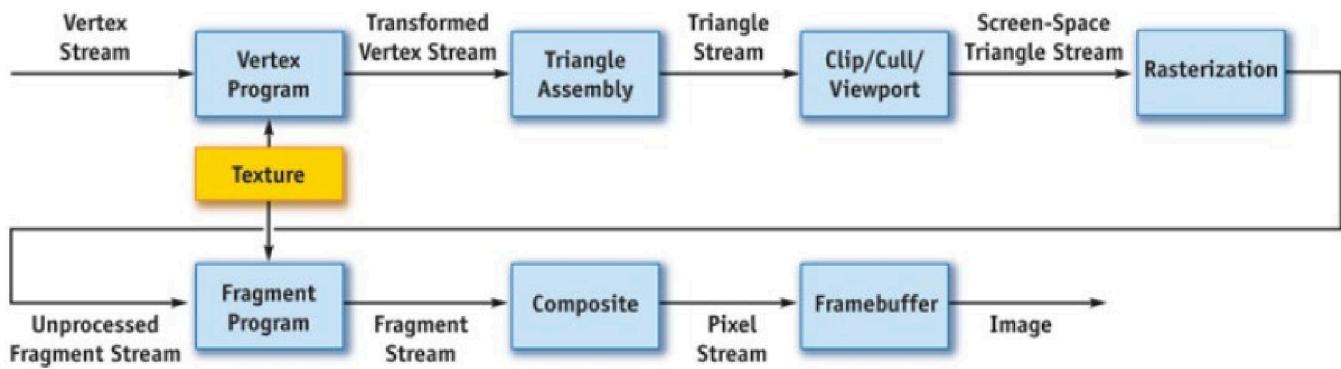
- **Clipping (裁剪)**: 裁剪超出视口的图形
- **Convert geometric representation (vertices) to image representation (fragments)**: 将几何顶点转换为图像片段 (像素数据, 如颜色、深度等)
- **Interpolate per - vertex quantities across pixels**: 在像素间插值逐顶点的属性 (如颜色、光照等)

Fragment processors (片段处理器) 专注于给每个像素 “上色”

- **Depth comparison (深度比较)**: 确定像素的前后位置, 处理遮挡关系
- **Compute a color for each pixel (计算每个像素的颜色)**: 根据光照、纹理等因素计算像素最终颜色
- **Optionally read colors from textures (images) (可选: 从纹理读取颜色)**: 可从纹理中获取颜色信息, 丰富像素色彩

Stream Programming Model

图形流水线与流模型高度适配，通过数据流连接各操作步骤



Efficient Computation

数据级并行 (Data - level parallelism)

对相同数据的不同子集执行相同操作。

例如，在图形渲染时，每个三角形都能独立于其他三角形进行处理，无需相互等待，从而提升计算效率

想象你有一堆苹果要洗。如果有很多人一起洗，每个人都做“洗苹果”这个同样的动作，但每个人洗不同的几个苹果（即处理同一批数据的不同子集）

任务级并行 (Task - level parallelism)

对相同或不同数据执行不同操作

常见形式是“流水线”，即让一组数据依次经过一系列独立任务，每个任务都能独立执行。

例如，将渲染过程拆分为几个独立阶段，每个阶段处理完数据后交给下一阶段，无需重复或回溯。这样各阶段可同时处理不同部分，像流水线一样连续运作，提高整体计算效率。

还是以苹果为例，现在不是大家都洗苹果，而是有人专门负责切苹果，有人专门负责给苹果去核，有人专门负责包装苹果。每个环节（任务）做不同的操作，但处理的是同一批苹果（或不同数据）

Efficient Communication

GPU 并行化的性能受限于 CPU - GPU 通信的复杂性

以下三种方法提升通信效率：

- 传输整个流而非单个元素：**在片外通信off-chip communication时，优先传输数据的“整个流”，而非逐个元素传输。这样每个元素的平均传输成本更低，就像一次运一批货物比多次运单个货物更高效
- 构建流水线式应用：**将应用设计成由多个内核组成的流水线结构。这样中间结果无需先存储到片外再读取，可直接流向下一处理阶段，避免了片外存储中间结果的开销
- 采用深度流水线：**尽可能增加流水线的阶段数量（深度流水线）。这样数据访问时间会被其他流水线阶段的操作“隐藏”，例如一个阶段在等待数据时，其他阶段可继续工作，从而提升整体效率

Computer Animation

艺术家导向型 (Artist - directed): 以关键帧动画 (keyframing) 为典型

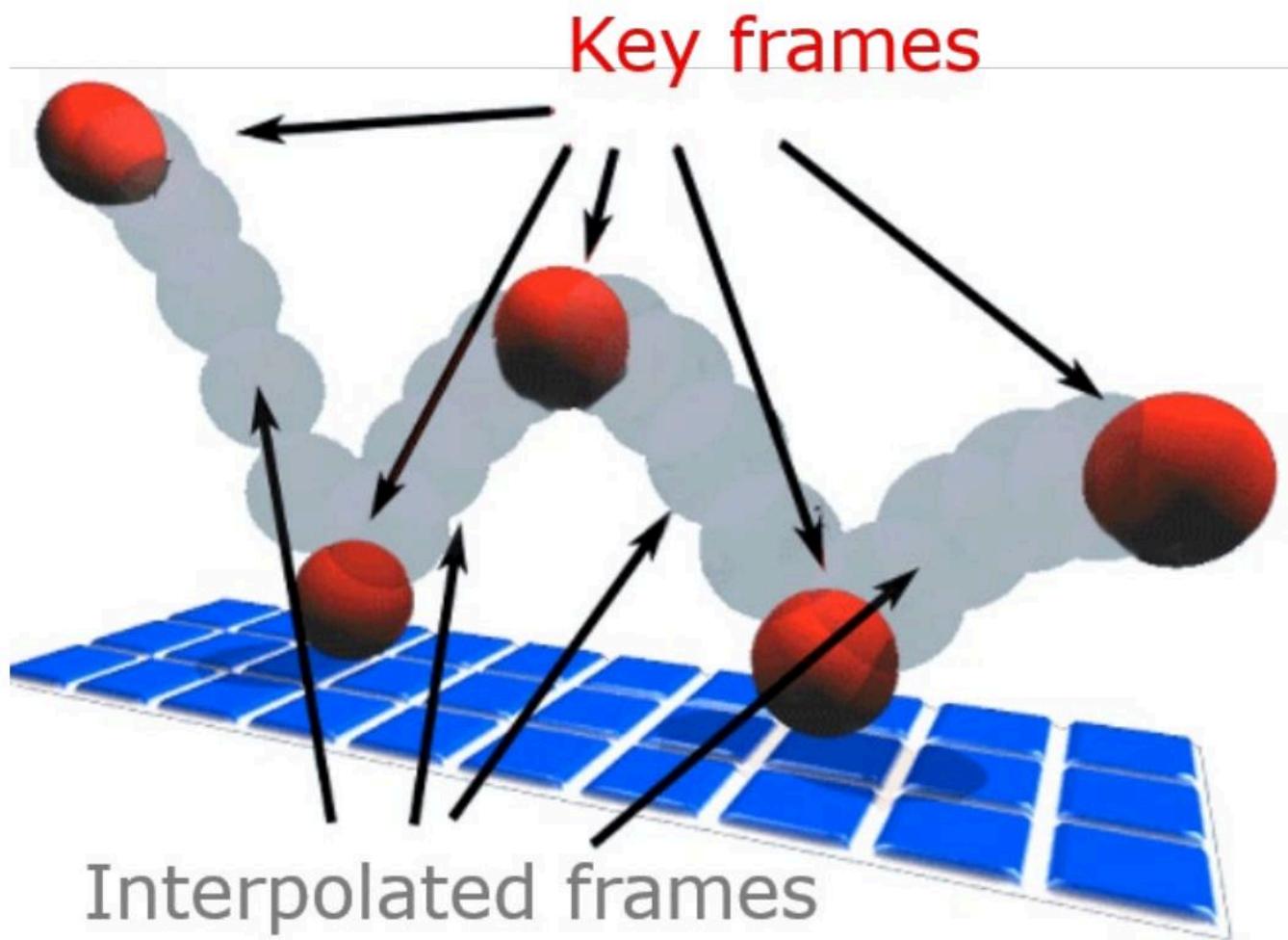
过程式 (Procedural): 例如物理模拟

数据驱动型 (Data - driven): 以动作捕捉 (motion capture) 为代表

Keyframing animation

只需指定重要的“关键事件”，计算机通过插值或近似 interpolation/approximation 的方式自动填充其余部分，生成流畅的动画过渡。例如制作一个物体移动的动画，只需设定物体在起始点和终点的位置（关键事件），中间的移动过程由计算机计算生成

“事件”不仅限于物体的位置，还可以是颜色、光照强度、相机变焦等多种属性。比如制作一盏灯从暗到亮的动画，只需设定灯在初始时刻的暗度和结束时刻的亮度（关键事件），中间的亮度变化由计算机自动处理



Procedural animation

动画师通过定义一个过程或一组操作来制作动画。

每个操作可生成或改变流经的数据，且能有条件或无条件地执行

以粒子系统、刚体动力学和柔体动力学等为例，用户只需指定一套规则、初始条件（如物体的初始位置、速度）和参数值，然后运行模拟，计算机便会依据这些设定自动生成动画



Motion capture (MoCap) animation

MoCap（动作捕捉）是对人类、动物以及无生命物体的运动进行采样，并将其记录为三维数据的技术



电磁动作捕捉 (Electromagnetic mocap)

通过相对磁通量 relative magnetic flux 计算每个关节joint的三维位置和方向

优势

- 可测量三维位置和方向。
- 不存在遮挡问题（如物体或身体部位相互遮挡不影响捕捉）。
- 无需特殊光照条件。
- 能同时捕捉多个对象的动作

劣势

- 易受磁干扰（如金属物体影响）。
- 无法捕捉变形动作（如面部表情变化）。
- 难以精确捕捉小骨骼的运动（如手指动作）。
- 精度不如光学动作捕捉系统



光学动作捕捉（Optical mocap）

使用多个经过校准的相机，将表演的不同视角进行数字化处理

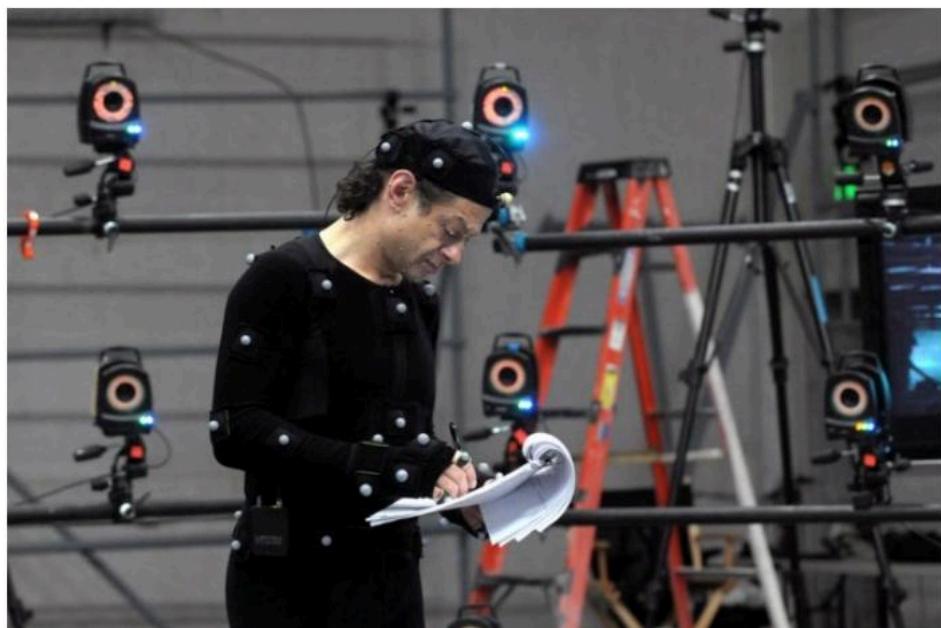
捕捉对象需穿戴反光标记点，系统通过相机准确测量这些标记点的三维位置，从而获取动作数据

优点

- 可测量三维位置和方向。
- 是最精确的动作捕捉方法。
- 帧率非常高，能捕捉细腻动作（如身体姿态、手指动作、面部表情变化等）。

缺点

- 存在遮挡问题（标记点被遮挡时影响捕捉）。
- 难以精准捕捉多个演员之间的互动。
- 设备和技术成本高，较为昂贵。



无标记动作捕捉（Markerless mocap）

基于视频的动作捕捉（Video - based mocap），是指通过单个摄像头的视频流来捕捉三维动作表现的技术。它无需依赖反光标记点

基于深度传感器（Depth sensor-based），是指利用单个深度相机采集场景的深度信息，进而实现三维动作捕捉的技术

Optical MoCap Pipeline

规划（Planning）：角色 / 道具设置，标记点设置

校准（Calibration）：相机和对象校准

处理标记点（Processing markers）：基于标记点做记录，标注，提取，计算

数据处理（Data processing）：将收集的数据做补充过滤完善等过程