# Cameras

Steve Rotenberg
CSE168: Rendering Algorithms
UCSD, Spring 2014

# Radiance Equation

$$L_r(\boldsymbol{\omega}_r) = \int_\Omega f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_r) L_i(\boldsymbol{\omega}_i) \cos\theta_i \, d\boldsymbol{\omega}_i$$

$$L_r(\boldsymbol{\omega}_r) = \int_{\varphi=0}^{2\pi} \int_{\theta=0}^{\frac{\pi}{2}} f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_r) L_i(\boldsymbol{\omega}_i) \cos\theta_i \sin\theta_i \, d\theta_i \, d\varphi_i$$

$$L_r(\boldsymbol{\omega}_r) \approx \frac{2\pi}{N} \sum^{N} f_r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_r) L_i(\boldsymbol{\omega}_i) \cos\theta_i$$

# Camera Focus

# Camera Focus

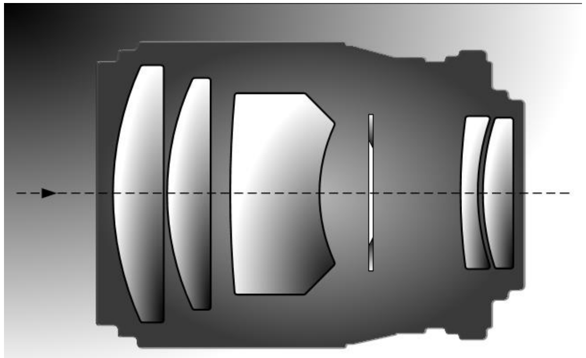
- So far, we have been simulating *pinhole* cameras with perfect focus
- Often times, we want to simulate more realistic camera lenses that blur objects that are not in focus
- We say that real lenses have a limited *depth of field*, where the depth of field refers to the zone that is in focus
- Sometimes, this blur is referred to as *defocus blur*

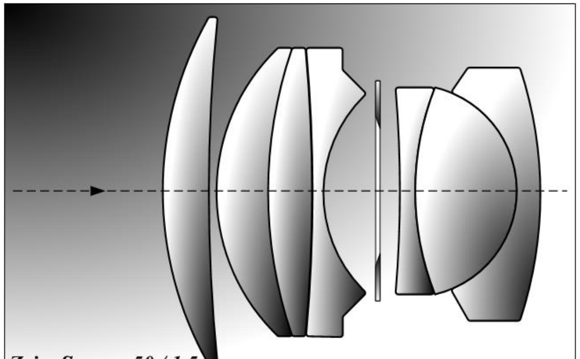
# Defocus Blur

- Defocus blur can be a bad thing if the subject of the image is out of focus
- However it can sometimes be a good thing
- If the subject is in focus and the background is blurred, this can have the effect of drawing the attention to the subject while removing distractions from the background
- It can have a nice artistic effect if handled properly

# Lenses

- In photography, the term *lens* refers to the whole optical system in front of the film
- This is generally made from several *lens elements*, which are the individual pieces of glass in the lens, plus the iris and any structural components
- Most modern lenses have at least 4 elements, and complex zoom lenses can have more than 10

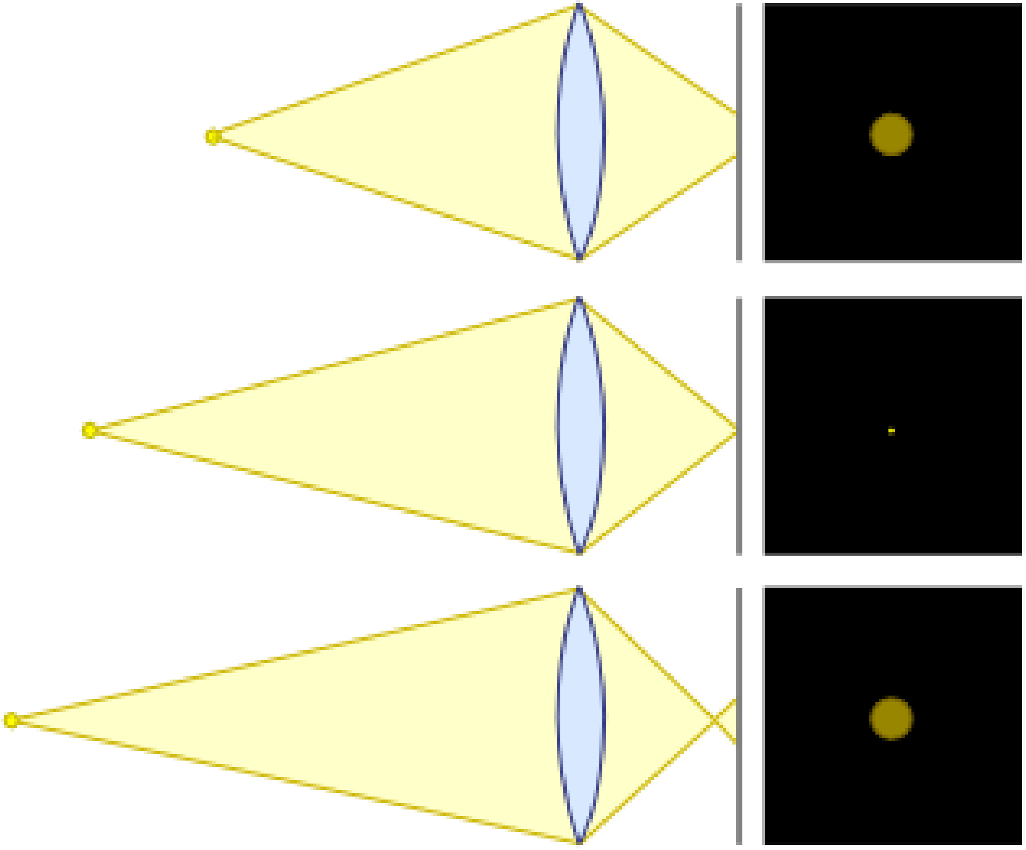

ZEISS T* 135mm / F2.8 - Sonnar telephoto lens


Zeiss Sonnar 50 / 1.5

# Focal Plane

- With a typical lens, there is a plane in front of the camera that is in perfect focus- this is called the *focal plane*
- Things get blurry as they get closer to the camera or further away from the focal plane

# Focal Plane



# Aperture

- A camera *aperture* is an opening through which light travels
- A small aperture will lead to a sharper image and a large aperture will lead to a blurrier image
- Typically, in a real camera, the aperture size can be changed with an adjustable *iris*

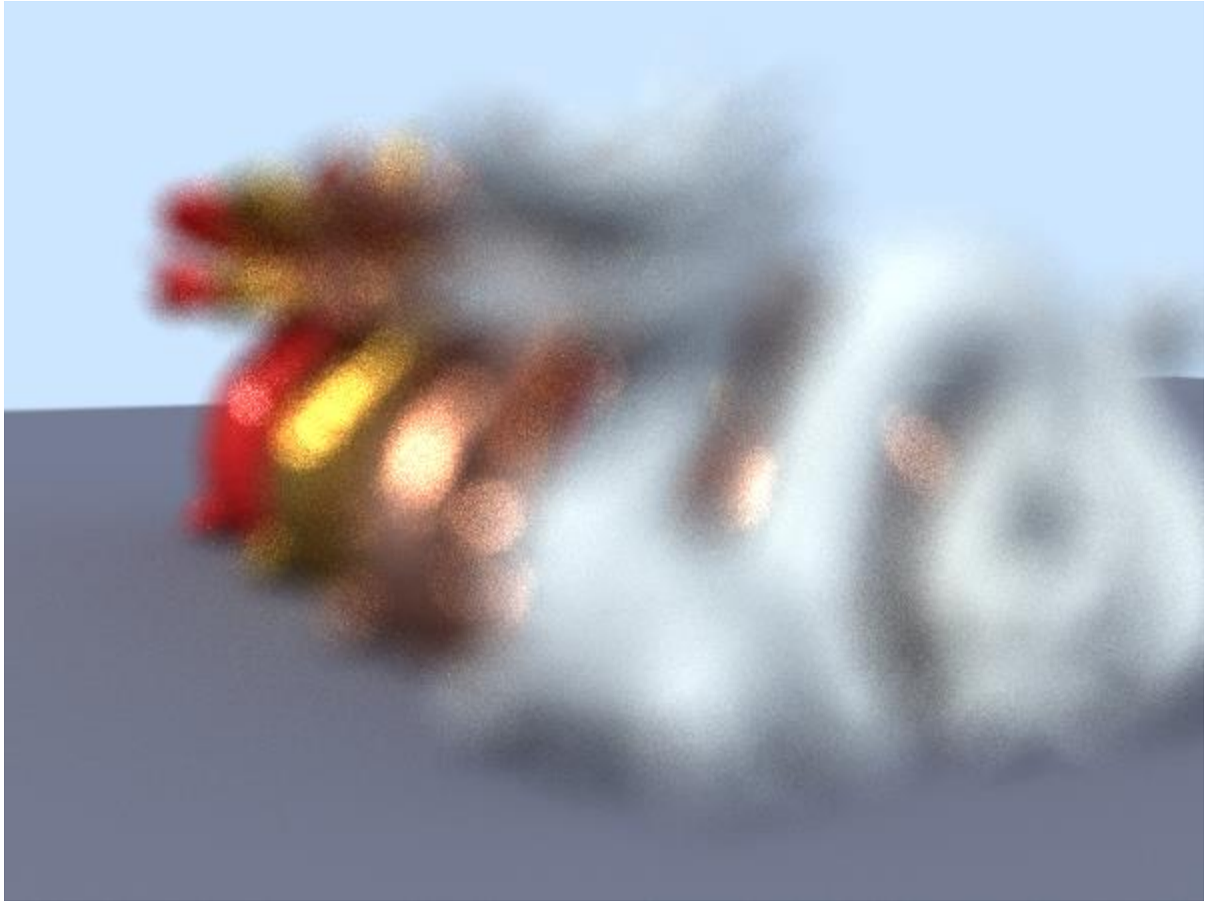
# Rendering Camera Focus

- To add camera focus blur to a ray tracer, we need to model the camera lens
- We could model the entire complex lens with multiple lens elements, lens coatings, and an iris. This is actually a fairly common approach in high end movies, where computer generated objects need to be integrated into live-action scenes
- However, we will examine a much simpler method
- This method requires adding two more parameters to the Camera class: Aperture and FocalPlane
- Aperture refers to the diameter of the lens and FocalPlane is the distance in front of the camera

# Rendering Camera Focus

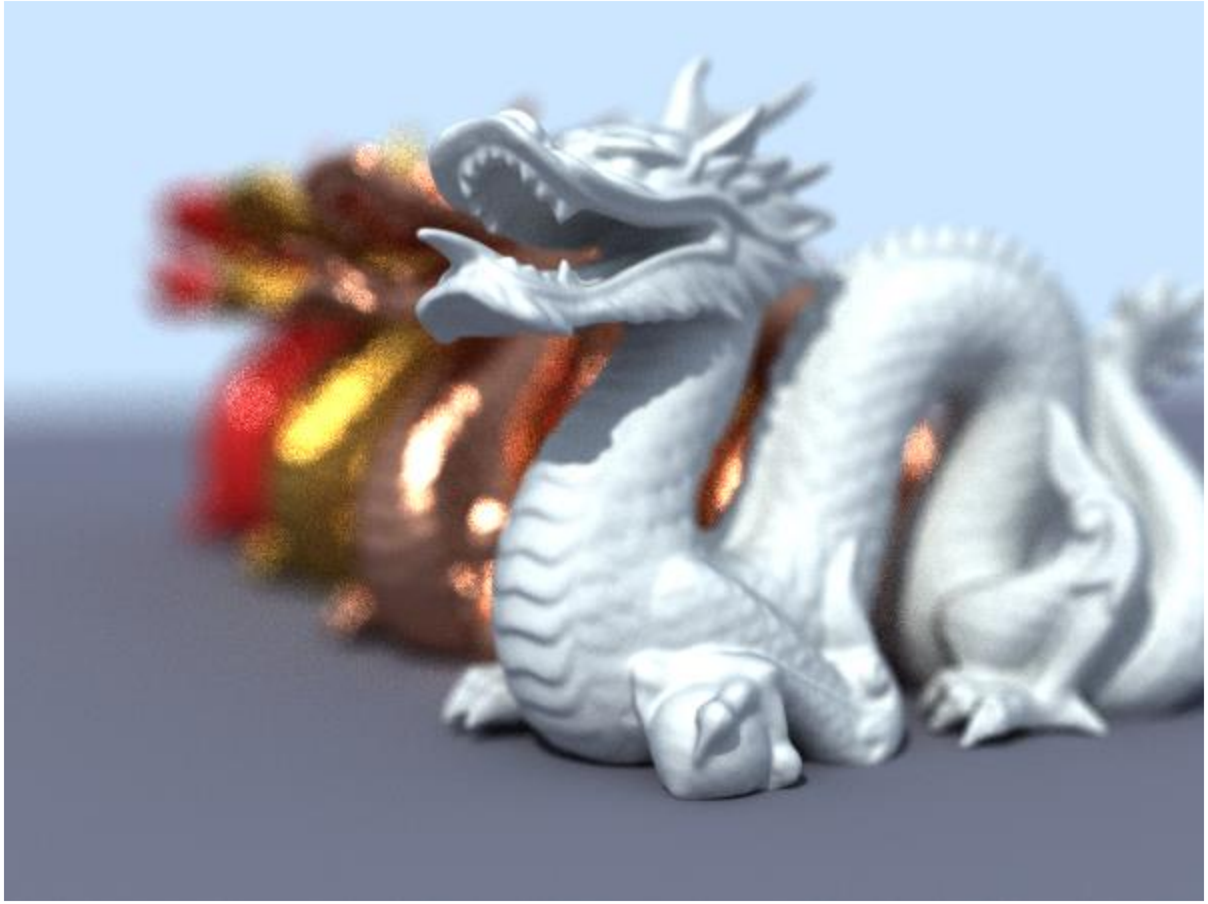
- Our existing approach to generating camera rays uses a virtual image plane, which is 1.0 unit in front of the camera. We first generate a point on the image plane and then generate a ray from the camera origin through the point
- To modify this, all we need to do is scale the virtual image plane distance to the focal plane, and then generate a ray origin by choosing a random point on a circular disk the size of the camera aperture

# Distant Focal Plane

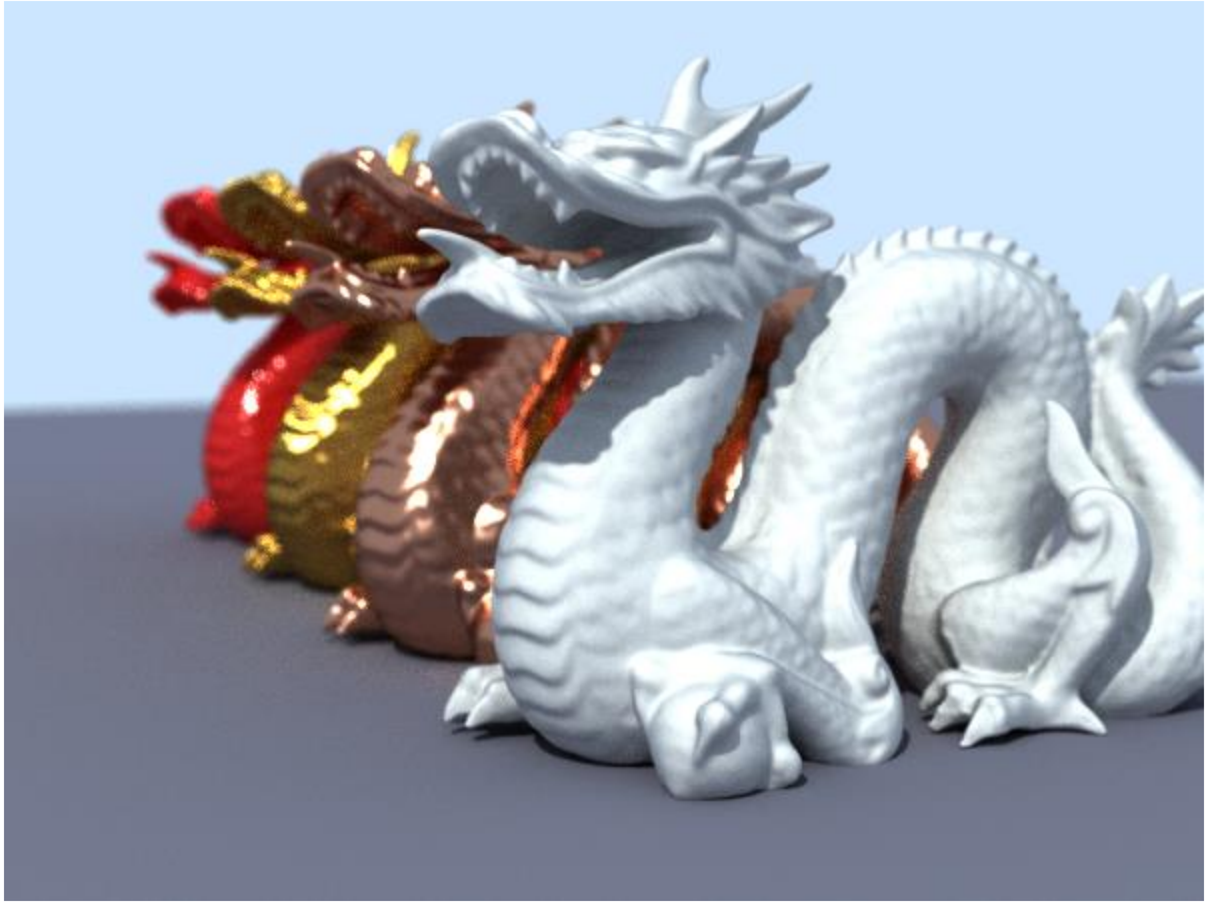


# Medium Focal Plane

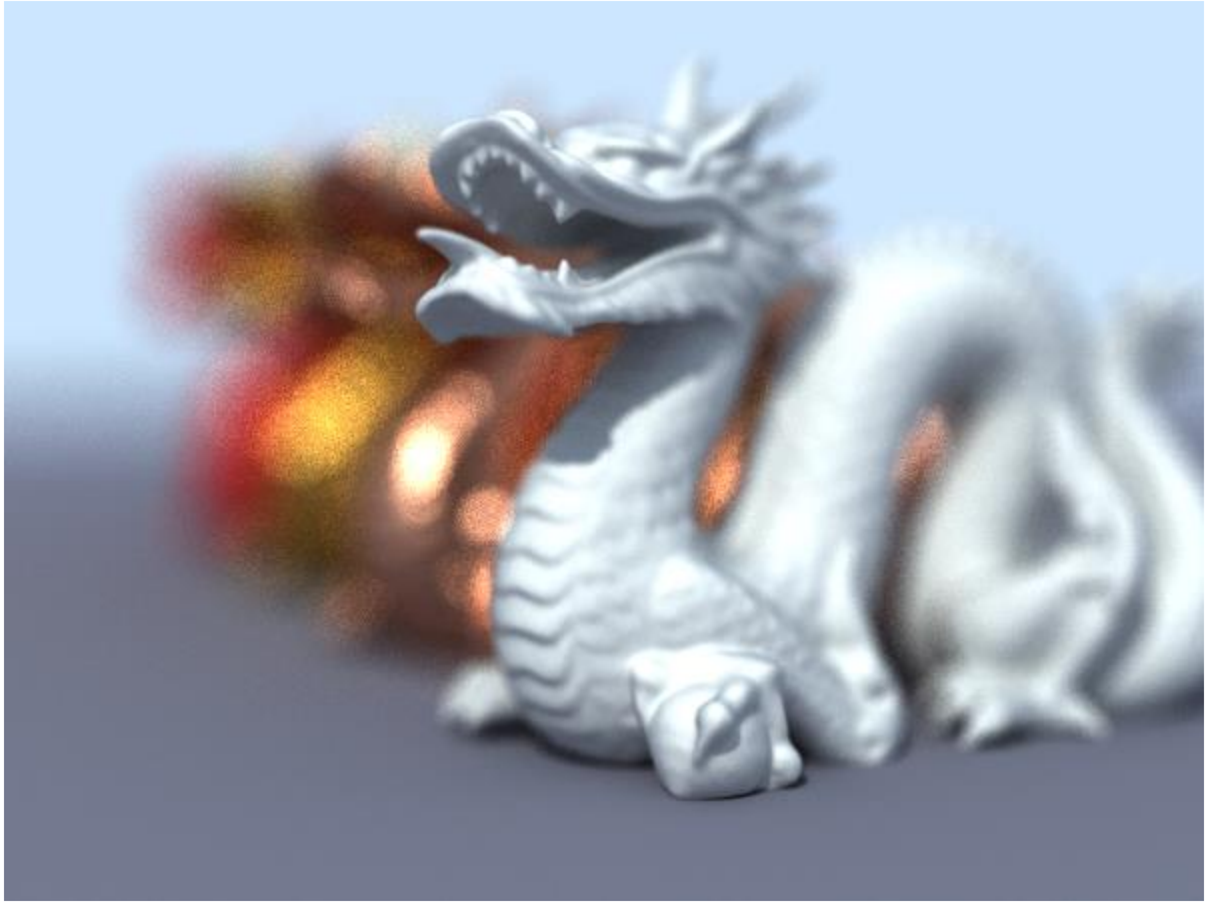
# Close Focal Plane



# Small Aperture



# Large Aperture

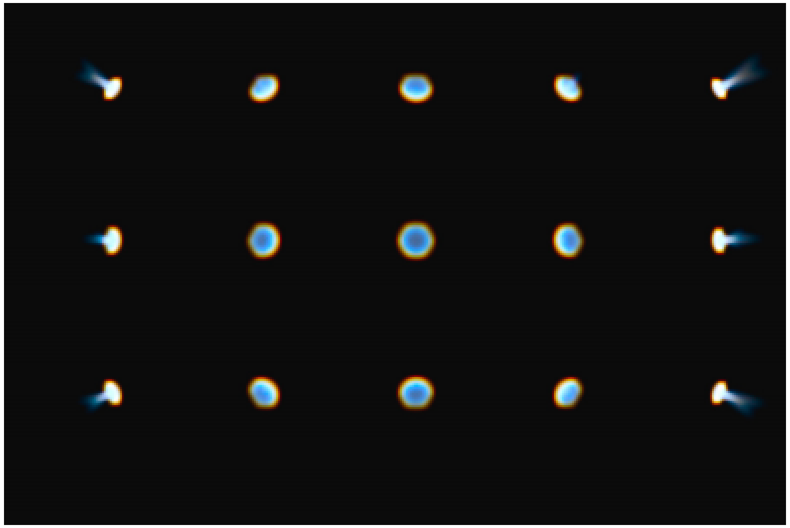


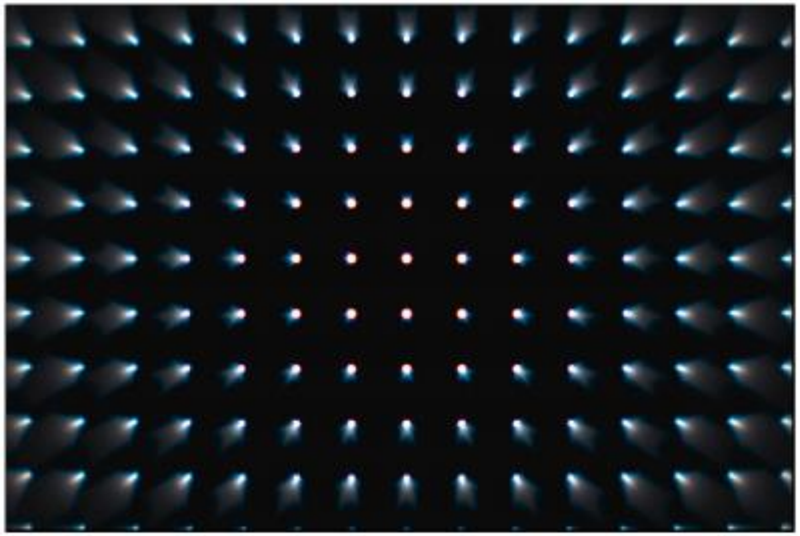
# Bokeh

- In recent years, the Japanese word *bokeh* has been adopted by English speaking photographers and computer graphics practitioners to refer to the artistic effect of defocus blur
- More specifically, the term refers to how the lens renders out-of-focus points of light

# Point Spread Function

- The *point spread function* (PSF) of an optical system describes how an individual point of light in the scene will may to the image





# Motion Blur

# Motion Blur

- *Motion blur* refers to the blurring we see on fast moving objects
- Motion blur is generally a good thing, and can improve the perceived realism in animations
- Motion blur is sometimes called *temporal antialiasing* (at least that's what it's called in the computer graphics world), and it reduces the aliasing phenomenon known as *strobing*

# Shutter Speeds

- Motion blur occurs because the camera shutter is open for a finite length of time
- Camera shutter speeds vary based on light levels, exposure settings, film speed, etc.
- Typical shutter speeds range from $1/30^{th}$ of a second down to $1/4000^{th}$ of a second
- Some still images use very long shutter speeds (maybe a few seconds)
- Motion picture cameras (video or film) require the shutter speed to be faster than the frame time, so for 60Hz video, a typical shutter speed would be $1/100^{th}$ of a second or less
- Older film cameras typically run at 24 fps, and the shutter is typically open for up to half of the frame time, so $1/48^{th}$ of a second or less

# Rendering Motion Blur

- To add motion blur to a ray tracer, we will need to distribute rays *in time*
- We add a ‘Time’ field to the Ray class
- When the camera generates a ray, we assign a random time
- We can either base it on actual time in seconds, or we can normalize it to a [0…1] range
- When we trace the ray, we need to intersect it with objects moved to their correct position, based on the ray time

# Moving Objects

- Not all objects in the scene need to move, so we can treat moving objects as a special case
- Just like we used the InstanceObject to position an object with a matrix, we can create a MotionObject which handles moving objects
- The MotionObject is a lot like an InstanceObject except it allows the matrix to change over time
- A simple way to do this is give it an initial and final matrix
- In the MotionObject::Intersect() function, we first use the input ray time to interpolate between the initial and final matrix. Then we have to compute the inverse on the fly, and from there, it behaves like a normal InstanceObject
- A more complex implementation could do an animation channel lookup and compute a matrix based on that

# Matrix Interpolation

- Assuming we go with the simpler option, we still have to address the issue of how we interpolate between the initial and final matrix
- The simplest way is to just do a linear interpolation (lerp) for each component of the matrix

$$\text{Lerp}(t,a,b) = (1-t)a + tb = a + t(b-a)$$

- This will work reasonably well, assuming the object doesn't rotate too much in the time interval, which is usually the case

# Matrix Interpolation

- However, for fast rotating objects (like a propeller), this may not be good enough
- When the matrix is linearly interpolated, every part of the object will move in a straight line from the initial to final position
- This is OK if the object only rotates a few degrees, but will start to break down if there is more than say 20 or so degrees of movement
- To improve on this, we can use quaternion interpolation or twist extraction

# Matrix Twist Extraction

- This function extracts a rotation axis and angle from an orthonormal matrix

```
float Matrix34::ExtractTwist(Vector3 &out) const {
        float theta=0.5f*sqrtf(1.0f + a.x*a.x + b.y*b.y + c.z*c.z);
        if(theta<0.0000005f || theta>0.9999995f) {
                if(theta<0.5f) printf("ERROR: Can't extract twist vector\n");
                out.Set(1.0f,0.0f,0.0f);
                return 0.0f;
        }
        float tmp=0.25f/theta;
        out.Set(tmp*(b.z-c.y),tmp*(c.x-a.z),tmp*(a.y-b.x));
        out.Normalize();
        return 2.0f*acosf(theta);
}
```

# Matrix Interpolation

- To set up: (only need to do this once)

```
Matrix34 delta=InitialMatrix;
delta.Inverse();
delta.Dot(delta,FinalMatrix);
Angle=delta.ExtractTwist(Axis);
```

- To interpolate:

```
Matrix34 mtx;
mtx.MakeRotateUnitAxis(Axis,ray.Time*Angle);
mtx.Dot(InitialMatrix,mtx);
mtx.d.Lerp(ray.Time,InitialMatrix.d,FinalMatrix.d);
```

- Note: this process assumes orthonormal matrices

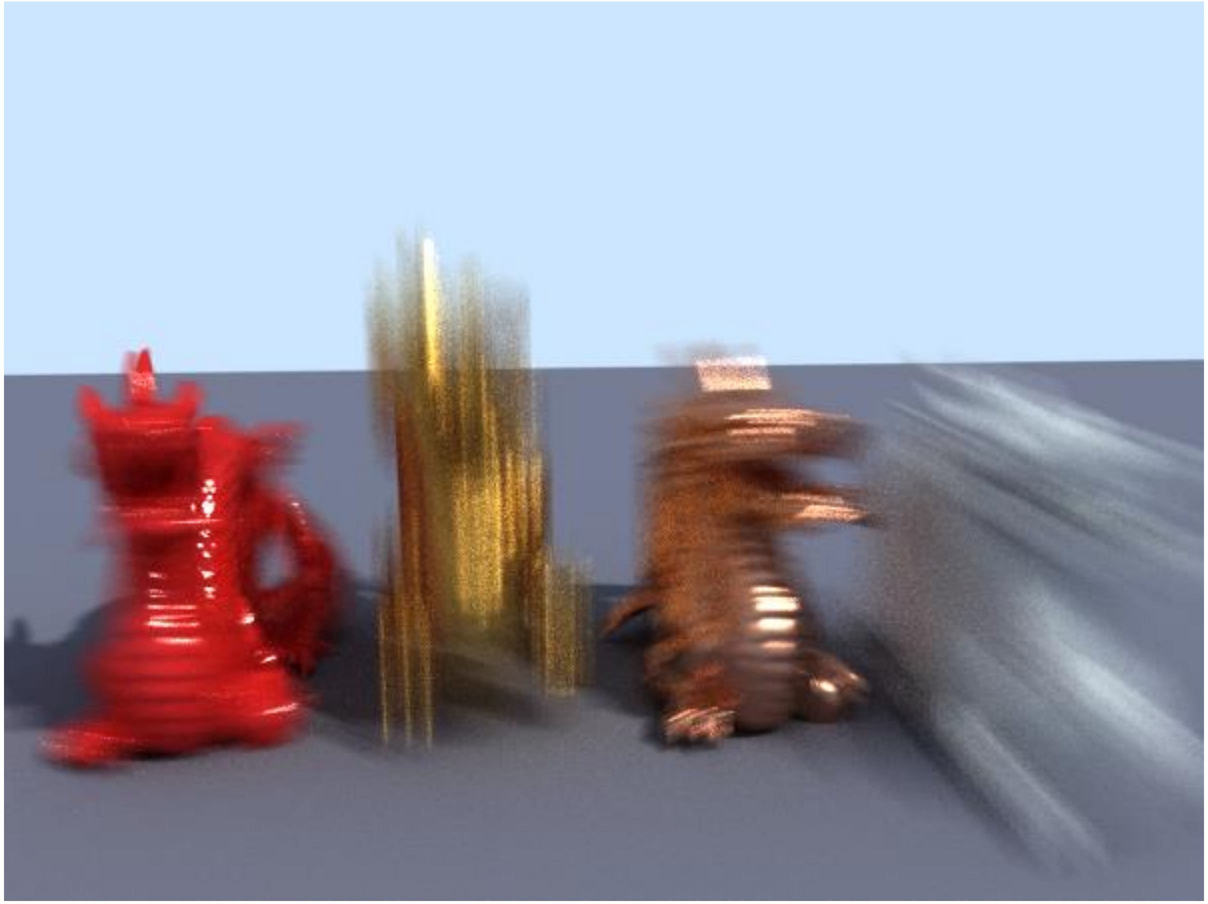
# Non-Orthonormal Matrices

- For non-orthonormal matrices, it might be best to just lerp the components
- Alternatively, one can extract twist, shear, and scale, and then interpolate all of those and reconstruct a matrix, but this might be overkill

# MotionObject

- Handling moving objects using this scheme is quite simple, and doesn’t have a large performance penalty, apart from the necessity to shoot enough rays to reduce the noise in the blurred areas
- One catch to remember though, is that if you allow Objects to be placed in spatial data structures, you will need to compute a bounding box for the MotionObject that encompasses the object for the entire time interval

# Moving Objects



# Moving Cameras

- Objects are not the only thing that moves
- Often, the camera moves as well
- This may result in blurring everything during fast camera moves or turns
- The camera can be handled much like an object
- It can have an initial and final matrix and that can get interpolated as well
- The camera ray time is chosen first, then the camera matrix is interpolated, and then the ray origin and direction are built from the interpolated camera matrix

# Moving Camera

# Fast Panning Shot

# Reflections & Shadows

- Motion blur comes from averaging across a finite interval of time
- Each camera ray is meant to be a single instant of that time interval
- Therefore, any reflected rays or shadow rays spawned from the initial camera ray **must use the same time as the camera ray**

# Animation Blur

- Technically, anything that changes over time can be blurred
- This isn't limited just to matrices
- For example, camera FOV angles can change- a fast zoom-in will be blurred
- One could even blur changes in lighting properties (position, color, brightness…) or any other dynamic property

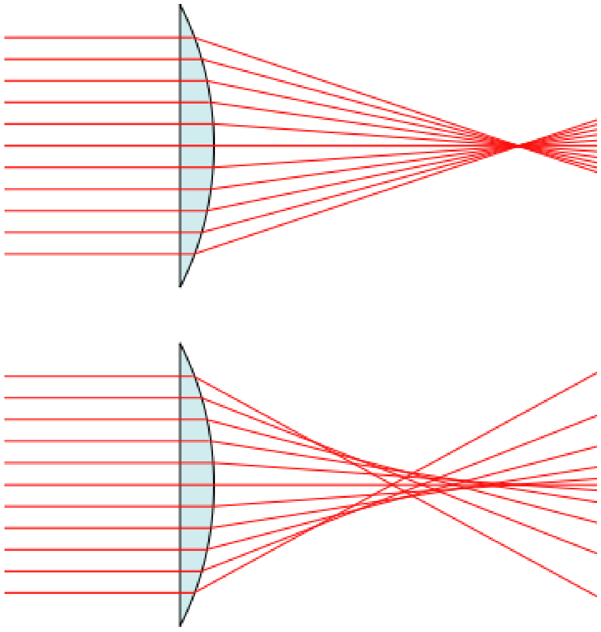
# Lens Imperfections

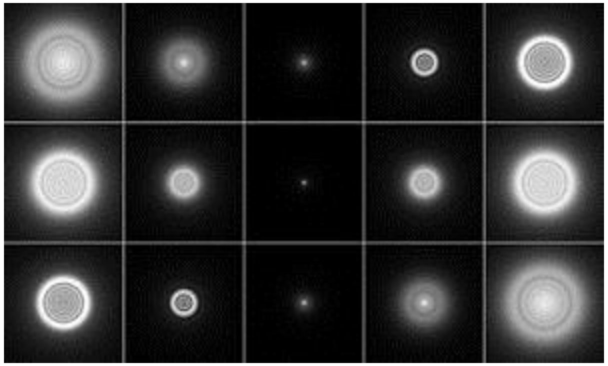
# Camera Imperfections

- Real camera lenses aren’t perfect and can suffer from imperfections or *aberrations*
- Like defocus and motion blur, these can sometimes be desirable and sometimes undesirable
- However, if our goal is to model a real lens or integrate synthetic objects into a real scene, we may want to include some of these

# Spherical Aberration

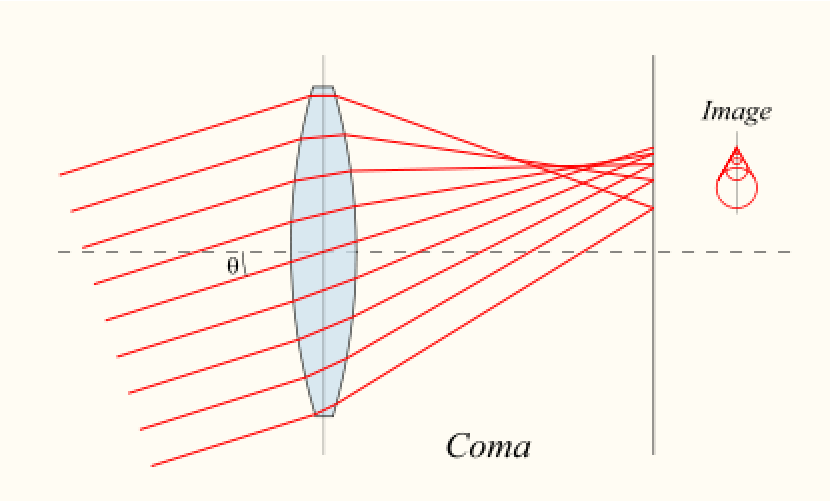
- Spherical aberration occurs when the individual rays coming from a point in the scene do not converge to a point on the film plane





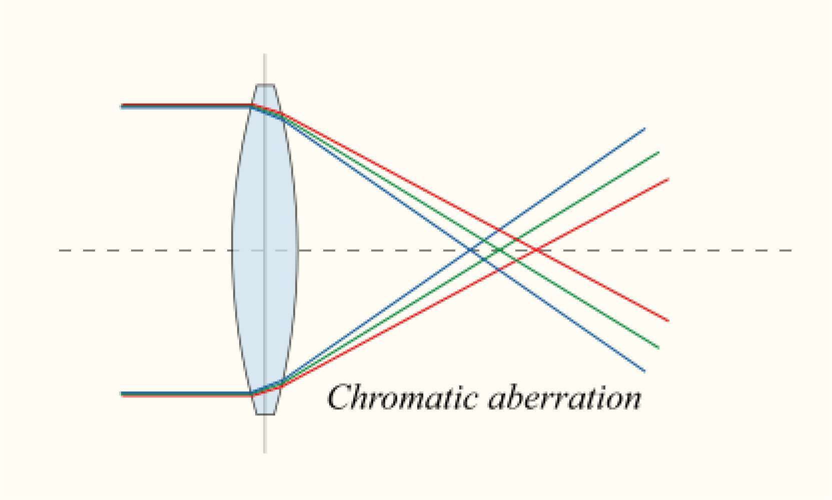
# Coma Aberration

- *Coma aberration* refers to the lens distortion that can cause off-axis point sources to appear to have a tail like a comet



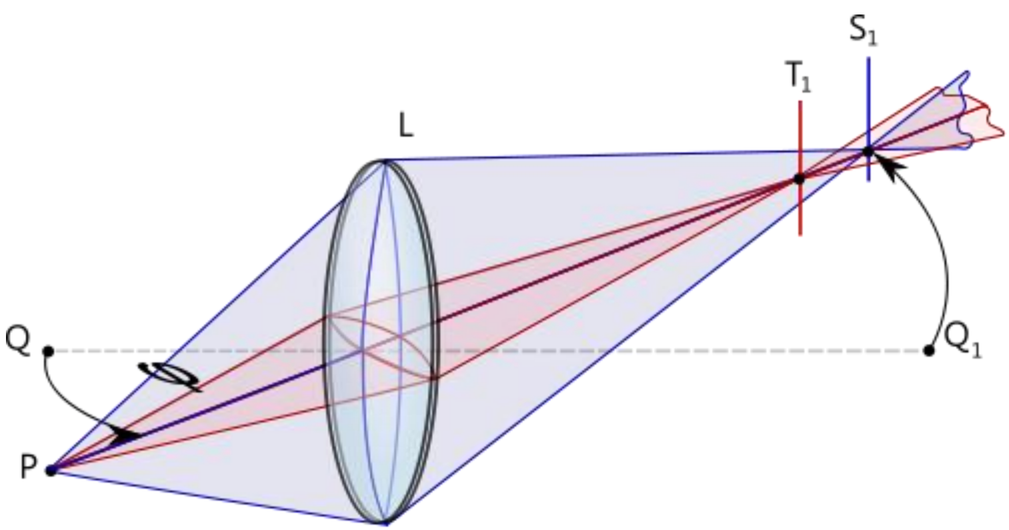
# Chromatic Aberration

- Chromatic Aberration is caused by the fact that the index of refraction for a lens varies with the wavelength of the light
- It can cause *color fringing* especially towards the edges of the image



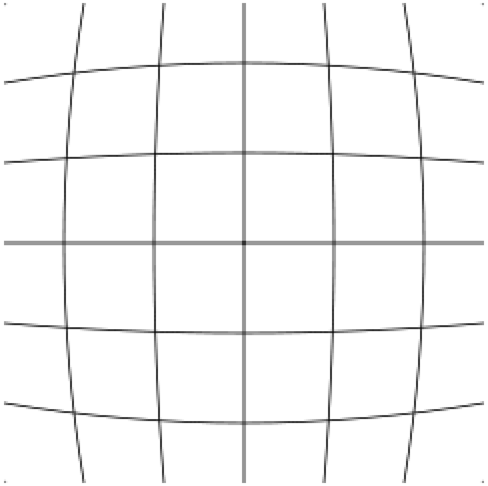

# Astigmatism

- Astigmatism is caused by lens elements that are not radially symmetric
- Rays in different planes focus to different points, leading to asymmetric distortions in the final image

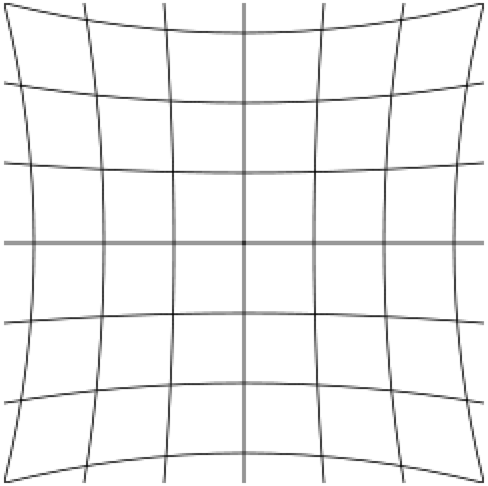



# Radial Distortion

- Lenses can cause various types of geometric distortion of the image
- Fish-eye lenses take advantage of this effect



Barrel distortion



Pincushion distortion

Fisheye lens (barrel)

# Vignette

- *Vignetting* is the reduction in brightness or color saturation towards the edge of the image

# Lens Flares

- Lens flares are caused by interreflection and scattering between the different elements and other components in a lens
- Very bright light sources tend to cause flares even if they are outside the image frame

# Bloom, Halos, & Stars

- Bloom, halos, and stars are forms of lens flares

# Modeling Lens Imperfections

- Most of the lens imperfections shown can be faked as a 2D post process
- We will discuss this in some more detail when we talk about HDR (high dynamic range) imaging in a later lecture
- There are some modern techniques however, that attempt to fully model the lens and capture all of these effects purely from the shape and arrangement of the lens elements

# Camera Research Papers

- “A realistic camera model for computer graphics”, Kolb, Mitchell, Hanrahan, 1995
- “Polynomial optics: a construction kit for efficient ray-tracing of lens systems”, Hullin, Hanika, Heidrich, 2012
- “Efficient Monte Carlo rendering with realistic lenses”, Hanika, Dachsbacher, 2014