Spectral colors are light at a single frequency (also called **monochromatic**). **Most** colors, however, are seen as a mix of several different fequencies.
The human eye has **rods** and **cones** that sense light.
Rods: no color (sort of), spread over the retina, more sensitive
Cones: three types of cones, each sensitive to different frequency distribution. Respond to different range of frequencies: long, medium, also called red, green, blue. Concentrated in fovea (center of retina). Less sensitive
Eyes records color by 3 measurements, so we can fool it by using a combination of 3 signals (this is how TVs work with RGB). Cone responses are linear, so we can use different linear combinations to produce the same result.
Response to stimulus Φ_1 is (L_1, M_1, S_1), Φ_2 is (L_2, M_2, S_2).
So, response to $\Phi_1 + \Phi_2 = (L_1 + L_2, M_1 + M_2, S_1 + S_2)$, response to $n\Phi_1 = (nL_1, nM_1, nS_1)$. The cone response is an integral: $L(\lambda) = \int \Phi(\lambda)L(\lambda)d\lambda$ (similar for M, S)
Metamers are different light inputs Φ_1, Φ_2 that produce the same LMS cone response.
Additive Mixing: given three primaries p_1, p_2, p_3, we can match an input light with $\Phi = \alpha p_1 + \beta p_2 + \gamma p_3$, so color can now be described by α, β, γ. (We can't have negative components because we can't have negative light, but we can overcome this by adding that primary to the input light in order to match them).

Also, Subtractive Mixing: Make a color $\Phi = W - (\alpha p_1 + \beta p_2 + \gamma p_3)$. Max is limited by W.
Don't forget! You can't say "the additive primaries are red, green and blue", because **any** set of three non-colinear primaries yields a gamut!
Color Matching Functions For a monochromatic light of wavelength λ_i, we know the amount of each primary necessary to match it is: $\bar{r}(\lambda_i), \bar{g}(\lambda_i), \bar{b}(\lambda_i)$. Given a new light input signal: $\Phi = \begin{bmatrix} \phi(\lambda_1) \\ \vdots \\ \phi(\lambda_N) \end{bmatrix}$. Given color matching functions in matrix form: $C = \begin{bmatrix} \bar{r}(\lambda_1) & \cdots & \bar{r}(\lambda_N) \\ \bar{g}(\lambda_1) & \cdots & \bar{g}(\lambda_N) \\ \bar{b}(\lambda_1) & \cdots & \bar{b}(\lambda_N) \end{bmatrix}$, the amount of each primary necessary to match is given by $C\Phi$

Gamut: A gamut is the chromaticities generated by a set of primaries. Because everything we've done is linear, interpolation between chromaticities on a chromaticity plot is also linear.

Color Phenomena – Some Definitions

Reflection: light strikes an object, some frequencies reflect and some absorb. The reflected spectrum is light times surface.
Transmission: light strikes an object, some frequencies pass, some are absorbed (or reflected)
Scattering: interactions w/ small particles in some medium. Long wavelengths pass through fine, but short ones scatter.
Interference: wave behavior of light. Reinforce if two light waves are "in phase" (the oscillations mach up), but they cancel if they're out of face. This is wavelength dependent (obviously)
Iridescence: this is the interaction of light with small structures and thin transparent surfaces. A light wave may be partially reflected and partially transmitted at the surface of a thing layer of transparent material - the two parts of the wave interfere with each other when the transmitted wave is reflected from a lower layer and reemerges at the surface.

BRDF: Bi-directional Reflectance Distribution Function

Given surface material, incoming light direction, direction of viewer, orientation of surface → returns fraction of light that reaches the viewer. Frequency dependent: $\rho = \rho(\theta_V, \theta_L, \lambda_{in}, \lambda_{out})$ OR $\rho = \rho(v, l, n)$ (viewer, incident, normal). The BRDF model doesn't capture everything (such as subsurface scattering or inter-frequency interactions).

We can approximate the BRDF as the sum of a **diffuse** component, a **specular** component and an **ambient** term.

Diffuse component
Lambert's Law: intensity of reflected light is proportional to the cosine of the angle between surface and the incoming light direction (independent of viewing angle): $\max(k_d I(\hat{I} \cdot \hat{n}), 0)$

Specular Component
This is a mirror-like reflection that **does** depend on the view direction: $k_s I \max(\hat{r} \cdot \hat{v}, 0)^p$. The reflected direction $\hat{r} = -\hat{I} + 2(\hat{I} \cdot \hat{n})\hat{n}$. We can do a "half angle" approximation, but this was from computers couldn't do floating point math that well.

Ambient Term
This is a **cheap hack**. It's just a uniform color added to everything.

TYPES OF SHADING

Flat Shading: use a constant normal for each triangle/polygon. The polygon objects don't look very smooth, and the faceted appearance is obvious (esp w/ specular).
Smooth Shading: compute "average" normal at vertices and interpolate across polygons. Use threshold for "sharp" edges.
Gouraud Shading: Compute shading at each vertex. Interpolate colors from vertices. It's fast and easy and looks smooth, but it's not that great at specular reflections.
Phong Shading: Compute shading at each pixel. Interpolate **normals** from vertices. It looks smooth and has better speculars, but is expensive and takes longer.
A mapping function $f(p) = p'$ maps a point in the original image to a point in the transformed image.

We do everything linearly
Polygons are defined by points, and edges are defined by interpolation between two points. Interior is defined by the interpolation between all points. Composing two linear functions is still linear. We transform the polygon by transforming vertices.
$f(x) = a + bx \quad g(f) = c + df \quad \rightarrow g(x) = c + df(x) = c + ad + bdx$. The basic transformations are rotate, scale and translate (shear is a composite transformation).

Rotations
Rotations are positive counter-clockwise, and are consistent with the right-hand rule. Rotation matrices are orthonormal. In 2D, they commute, but in 3D they don't!
So, for everything but translations, we have $x' = A \cdot x$.
Rotations and scales are always about the origin, but how do we rotate/scale about another point? We could translate the point to the origin, rotate it how we want, then put it back where it was: $p' = (-T)RTp = Ap$.
To scale about an arbitrary axis, we can translate axis to origin, rotate axis to align with one of the coordinate axes, then scale as desired, then undo steps 2 and 1 in that order.
In 3D, we can make arbitrary rotations from axis-aligned matrices: $R = R_z \cdot R_y \cdot R_x$. These are called Euler Angles. These allow tumbling, and euler angles are non-unique. We have the problem of gimbal-lock!

Exponential Maps: These are a direct representation of arbitrary rotation (axis-angle and angular displacement vector). We rotate θ degrees about some axis, and we encode θ by the length of the vector $\theta = |r|$. No gimbal-lock! Alows tumbling. Orientations are space within π-radius ball, and they're nice for interpolation

Quaternions: more popular than exponential maps, and another alternative for doing rotations. They're a natural extension of $e^{i\Theta} = \cos(\theta) + i\sin(\theta)$. We get uber-complex numbers: $q = (z_1, z_2, z_3, s) = (\mathbf{z}, s) = iz_1 + jz_2 + kz_3 + s$. Features: no tumbling, no gimbal-lock. Nice for interpolation.

Singular Value Decomposition (SVD)
For any matrix A, we can write an SVD: $A = QSR^T$, where Q and R are orthonormal, and S is diagonal.
We can also write a Polar Decomposition: $A = PRSR^T$, where P is also orthonormal: $P = QR^T$.
We can force P and R to have Determinant=1 so that they are rotations. Any matrix is now Rotation:Rotation:Scale:Rotation
Matrix multiplication composites matrices: $p' = BAp \leftarrow$ "Apply A to p and then apply B to that result": $p' = B(Ap) = (BA)p = Cp$. However, with this, translations are left out of compositing transformations, so we need **homogeneous coordinates**.
Normals don't transform normally (heh). To get the transformed normal, we use $n' = (M^{-1})^T \cdot n$

SCENE GRAPHS

Draw scene with pre and post-order traversal (Apply node, draw children, undo node if applicable). Nodes do pretty much everything: geometry, transformations, groups, color, switch, etc. Instances make it a DAG, and we use these for bounding boxes later.

Refracted Rays
Transparent materials bend light: snell's Law: $\frac{n_i}{n_t} = \frac{\sin(\theta_t)}{\sin(\theta_i)}$. If $\sin(\theta_t) > 1$, then we have a total internal reflection. For the refracted rays, the coefficient depends on θ_i. The attenuation is wavelength/color dependent.

Anti-Aliasing
The on/off for pixels causes problems at the borders. What we can do to solve it is to send multiple rays through each pixel and average the results together. This is sort of like "distributed" raytracing, where we use multiple rays for reflection and refraction (at each bounce, send out more rays in quasi-random directions).
Use something similar for soft-shadows: send out multiple shadow rays.
Spatial Aliasing happens because we undersample - we don't look at enough pixels in order to correctly represent an image.

BSP - BINARY SPACE PARTITION TREES

We split space along planes, which allows fast queries of some spatial relations.
Simple construction: select a plane as a sub-tree root. Everything one one side belongs to one child, and everything on the other side belongs to the other child. Use a random polygon for splitting the plane.
Visibility Traversal: variation of in-order traversal $\rightarrow$ child one, sub-tree root, child two. Select "child one" based on location of the viewpoint - choose child one to be on the same side of subtree root as the viewpoint.

AABB - AXIS-ALIGNED BOUNDING BOXES

A bounding shape completely encloses the associated object (i.e. Rays cannot hit the object w/o intersecting the bounding shape. Two objects cannot collide if shapes don't overlap). For the **bounding boxes**, we use the min and max x, y, z of the object. If we apply a transformation to the object, then the bounding box is not necessarily axis-aligned, so we have to compute a new bounding box (in linear time) from the new min/max of the points.
We construct an AABB tree such that the children of a node are boxes or groups of objects that are contained within the box defined by the root (sorta...you should probably just look at the picture).

PROJECTION

Ray Generation vs Projection:
In viewing in ray tracing, we start w/ an image point and compute the ray that projects to that point by using geometry. In viewing by projection, we start w/ a 3D point, compute the image point that it projects to, and we do this with transforms.

Linear projection is projection onto a **planar surface**. These either converge to a point (perspective), or are parallel (converge at infinity) (orthographic).
With orthographic projections, there is no foreshortening (so perceiving distance is a bit weird), and parallel lines stay parallel. 1. translate **center** to origin 2. rotate **view** to -Z and **up** to +Y 3. center view volume 4. scale to canonical size $M = (S \cdot T_2) \cdot (R \cdot T_1) = M_0 \cdot M_v$

With perspective projections, there **is** foreshortening (further objects appear smaller). Some lines stay parallel, but most don't. Lines still look like lines, and the **z ordering is preserved**. We also have **vanishing points** with perspective projections. These depend on the scene (not intrinsic to the camera). Can have 1, 2, 3,etc .
1. Translate **center** to origin 2. rotate **view** to -Z, **up** to +Y 3. shear center-line to -Z axis 4. perspective a. Note: i is the distance from the center to the image plane. f is the distance from the farthest point on the object to the center. b. points at $z = -i$ stay at $z = -i$ c. points at $z = -f$ stay at $z = -f$ d. points at $z = 0$ go to $z = \pm\infty$ e. points at $z = -\infty$ go to $z = -(i + f)$ f. x and y value are divided by $\frac{-z}{i}$ g. straight lines stay straight h. Depth ordering preserved in $[-i, -f]$ i. movement alone lines is distorted 5. center view volume 6. scale to canonical size
M_v is steps 1,2. M_p is steps 3,4. M_o is steps 5,6. So we get $M = M_o \cdot M_p \cdot M_v$.
vanishing points: Consider a ray: $r(t) = p + t \cdot d$. Ignore the Z part of the matrix. X and Y will give you location in image plane. Assume the image plane at $z = -1$, then all points with direction x, y (remember, z doesn't matter) will all vanish at the same point.
All lines in direction **d** converge to the same point in the image plane — the vanishing point. Every point in plane is a vanishing point for some set of lines. Lines parallel to image plane $d_z = 0$ vanish at infinity. A horizon is a line of vanishing points.
clipping: The implicit equation for a plane through point **q** with normal **n** is $f(p) = n \cdot (p - q) = 0 = n \cdot p + D$. The parametric equation for a line going through points **a** and **b** is $p = a + t(b - a)$, so to find the intersection of this line with the plane, we plug it into the $f(p) = 0$ plane: $n \cdot (a + t(b - a)) + D = 0$, so $t = \frac{n \cdot a + D}{n \cdot (a - b)}$.

TEXTURES

Displacement Maps: Move the geometry based on a texture map (expensive and difficult to implement in many rendering systems).
Environment Maps: Treat the object as infinitesimal as compared to the rest of the scene (the reflection is based only on the surface normal). Sphere based parameterization: have the image on a ball, and compare normals on the object for which you're making the reflection. Pixels that have the same normal have the same color/shading.
Shadow Maps: Pre render the scene from perspective of the light source (only render the Z-buffer from this perspective and call it the shadow buffer), then render the scene from the camera perspective and compare the Z-buffer here with the shadow buffer. If nearer, light. If farther, shadow.

HERMITE

Specifying a Curve: Given the desired values, how do we determine coefficients for some sort of basis? Say for a cubic line, we're given endpoints $x(0)$ and $x(1)$ and the slopes at those points: $x'(0)$ and $x'(1)$. We need to determine the coefficients! We know it's cubic (should be **given**), so we can say:

$$x(0) = c_0 = x_0 \quad x(1) = \sum c_i = x_1 \quad x'(0) = c_1 = x'_0 \quad x'(1) = \sum i c_i = x'_1$$
$$\begin{bmatrix} x_0 \\ x_1 \\ x'_0 \\ x'_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

We take the inverse of that big matrix there (B) to get β_H. The equation for the line is given by $x(u) = P^3 \beta_H p$, where p is the initial points given above. $P^3 \beta_H$ is a vector with rows given by the dot product of the columns of β_H with a simple cubic $1 + u + u^2 + u^3$. Can do this construction for any odd degree.