Contents

Table of contents **1**

1 Modeling **2**
1.1 Splines . . . 2
1.1.1 Cubic Hermite Interpolation . . . 2
1.1.2 More than 1D . . . 3
1.1.3 Cubic blossom . . . 3
1.1.4 Bernstein polynomials . . . 5
1.1.5 General spline formulation . . . 5
1.1.6 Orders of continuity . . . 6
1.1.7 Cubic B-splines . . . 6
1.1.8 Bezier vs B-spline . . . 6
1.2 2D Surfaces . . . 7
1.2.1 Bicubic Bezier surfaces . . . 7
1.2.2 Tangents and Normals for patches . . . 8
1.2.3 Matrix notation for patches . . . 8
1.2.4 Implicit surfaces . . . 8
1.3 Hierarchical modeling . . . 8
1.3.1 Forward kinematics . . . 8
1.3.2 Inverse kinematics . . . 9
1.3.3 Jacobian . . . 9
1.4 Skinning . . . 9
1.4.1 SSD . . . 9
1.4.2 Vertex weights . . . 9
1.4.3 Linear blend skinning . . . 9

2 Raytracing **10**
2.1 Geometry representation . . . 10
2.1.1 Sphere . . . 10
2.1.2 Infinite plane . . . 11
2.1.3 Triangle . . . 11
2.2 Fundamentals . . . 12
2.2.1 Raycasting . . . 12
2.2.2 Shadows . . . 12
2.2.3 Reflection . . . 12
2.2.4 Refraction . . . 13
2.2.5 Lighting . . . 13
2.2.6 BRDF . . . 13
2.2.7 Phong lighting model . . . 15
2.2.8 Normal interpolation . . . 16
2.2.9 Textures . . . 16
2.3 Optimization . . . 17
2.3.1 Bounding volume: Axis-aligned bounding box . . . 17
2.3.2 Bounding volume hierarchy (BVH) . . . 17
2.3.3 Kd-trees . . . 17
2.4 Advanced techniques . . . 18
2.4.1 Global illumination . . . 18

1 Modeling

1.1 Splines

1.1.1 Cubic Hermite Interpolation

Each point is defined by its position h_n and slope h_{m+n}, m being the number of control points. To simplify calculations, it is assumed that $t_0 = 0$ and $t_1 = 1$.

The goal is to convert from a monomial basis

$$\begin{aligned}\phi_0(t) &= 1\\ \phi_1(t) &= t\\ \phi_2(t) &= t^2\\ \phi_3(t) &= t^3\end{aligned}$$

to a hermite basis

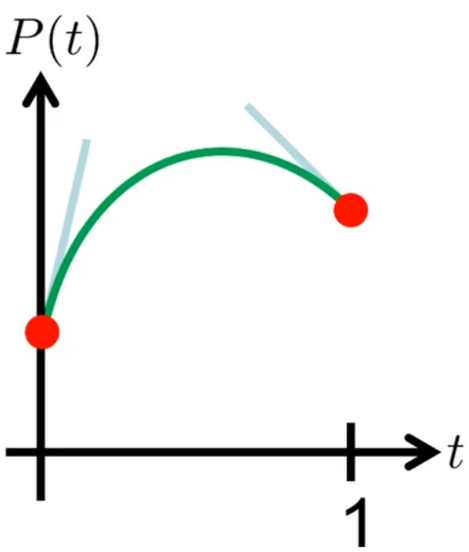
$$\begin{aligned}H_0(t) &= 2t^3 - 3t^2 + 1\\ H_1(t) &= -2t^3 + 3t^2\\ H_2(t) &= t^3 - 2t^2 + t\\ H_3(t) &= t^3 - t^2\end{aligned}$$

so that instead of having to manipulate polynomial coefficients

$$f(t) = a\phi_3(t) + b\phi_2(t) + c\phi_1(t) + d\phi_0(t)$$

an easier point slope method can be used:

$$f(t) = h_0H_0(t) + h_1H_1(t) + h_2H_2(t) + h_3H_3(t)$$



$$\begin{aligned}P(t) &= at^3 + bt^2 + ct + d\\ P'(t) &= 3at^2 + 2bt + c\end{aligned}$$

$$\begin{aligned}h_0 &= P(0) = d\\ h_1 &= P(1) = a + b + c + d\\ h_2 &= P'(0) = c\\ h_3 &= P'(1) = 3a + 2b + c\end{aligned}$$

Unknowns in this equation are a, b, c, and d, so a matrix can be used to solve the systems of equations:

$$\begin{pmatrix}0 & 0 & 0 & 1\\ 1 & 1 & 1 & 1\\ 0 & 0 & 1 & 0\\ 3 & 2 & 1 & 0\end{pmatrix}\begin{pmatrix}a\\ b\\ c\\ d\end{pmatrix} = \begin{pmatrix}h_0\\ h_1\\ h_2\\ h_3\end{pmatrix}$$

a, b, c, and d can be obtained from h values by inverting the matrix:

$$\begin{pmatrix}0 & 0 & 0 & 1\\ 1 & 1 & 1 & 1\\ 0 & 0 & 1 & 0\\ 3 & 2 & 1 & 0\end{pmatrix}^{-1}\begin{pmatrix}h_0\\ h_1\\ h_2\\ h_3\end{pmatrix} = \begin{pmatrix}a\\ b\\ c\\ d\end{pmatrix}$$

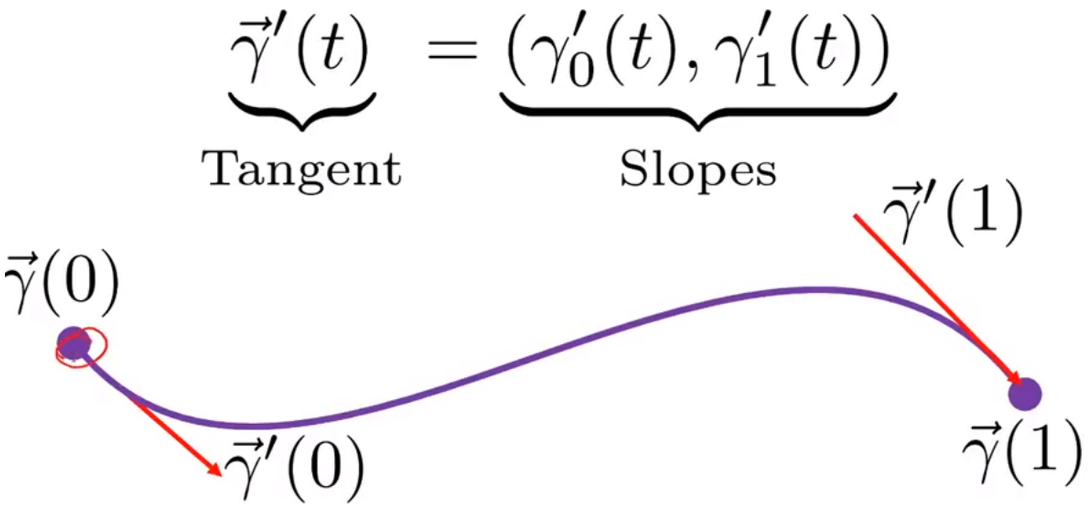
From these solved h values, $P(t)$ can now be converted to a form that is easier for a user to manipulate, in terms of h values:

$$\begin{aligned} P(t) &= at^3 + bt^2 + ct + d \\ &= (2h_0 - 2h_1 + h_2 + h_3)t^3 \\ &\quad + (-3h_0 + 3h_1 - 2h_2 - h_3)t^2 \\ &\quad + h_2 t + h_0 \\ &= h_0(2t^3 - 3t^2 + 1) + h_1(-2t^3 + 3t^2) + \\ &\quad h_2(t^3 - 2t^2 + t) + h_3(t^3 - t^2) \end{aligned}$$

Each equation in $P(t) = h_0(2t^3-3t^2+1)+h_1(-2t^3+3t^2)+h_2(t^3-2t^2+t)+h_3(t^3-t^2)$ that is multiplied by an h value is called a cubic hermite.

1.1.2 More than 1D

A parametric curve described by $\vec{\gamma}(t) = (\gamma_0(t), \gamma_1(t))$ can be converted into hermite basis like this:



where cubic hermite interpolation can be done for both dimensions.

1.1.3 Cubic blossom

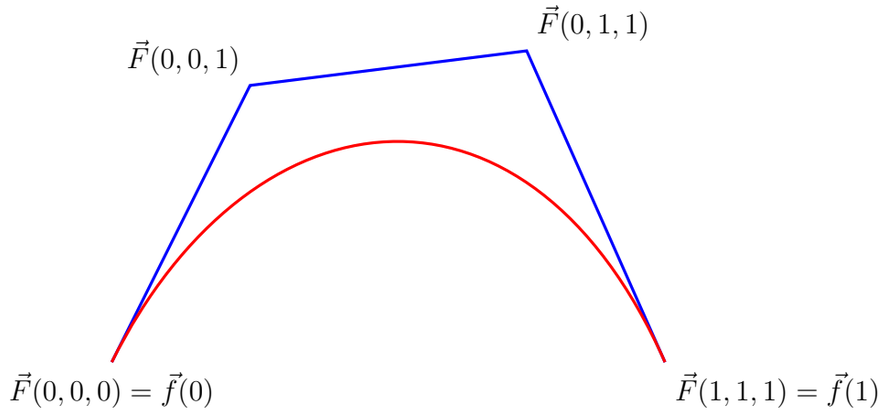
The cubic blossom of a function $\vec{f}(t)$ is $\vec{F}(t_1, t_2, t_3)$.
Cubic blossoms have three properties:

1. Symmetric
   - $\vec{F}(t_1, t_2, t_3) = \vec{F}(t_1, t_3, t_2) = \vec{F}(t_3, t_1, t_2) \cdots$
2. Affine
   - $\vec{F}(\alpha u + (1-\alpha)v, t_2, t_3) = \alpha\vec{F}(u, t_2, t_3) + (1-\alpha)\vec{F}(v, t_2, t_3)$
   - If only one of $\vec{F}$'s arguments $t_c = \alpha u + (1-\alpha)v$ is changing between different points on $\vec{F}$, then any value $\vec{F}(t_c, t_2, t_3)$ in between $\vec{F}(u, t_2, t_3)$ and $\vec{F}(v, t_2, t_3)$ is scaled equivalently with α like t_c is scaled between u and v.
3. Diagonal
   - $\vec{f}(t) = \vec{F}(t, t, t)$

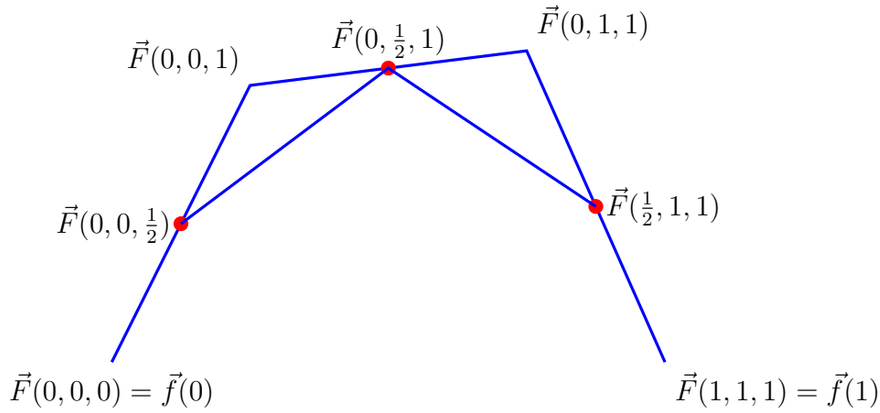
Blossoming examples

- $\vec{f}(t) = t^3 \mapsto \vec{F}(t_1, t_2, t_3) = t_1 t_2 t_3$
- $\vec{f}(t) = t^2 \mapsto \vec{F}(t_1, t_2, t_3) = (t_1 t_2 + t_1 t_3 + t_2 t_3)/3$
- $\vec{f}(t) = t \mapsto \vec{F}(t_1, t_2, t_3) = (t_1 + t_2 + t_3)/3$
- $\vec{f}(t) = 1 \mapsto \vec{F}(t_1, t_2, t_3) = 1$
- $\vec{f}(t) = 3t^3 - t + 1 = 3(t_1 t_2 t_3) - (t_1 + t_2 + t_3)/3 + 1$

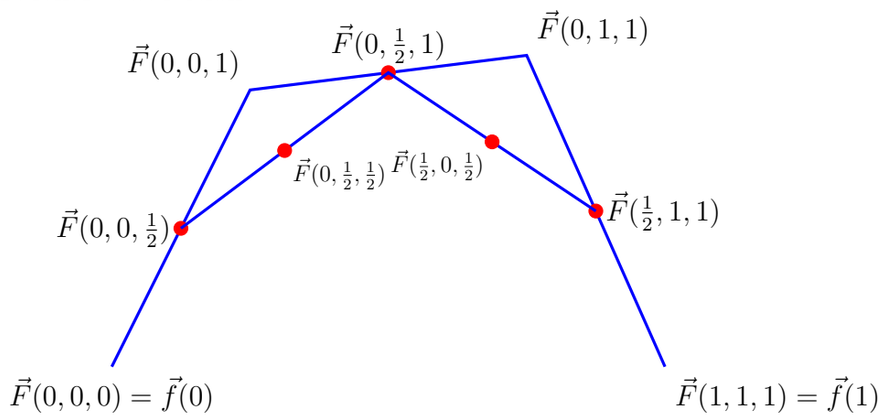
Cubic curves can be blossomed by blossoming each coordinate function separately, which will give a function that maps 3 t variables to two dimensions x and y: $\vec{F}(t_1, t_2, t_3) : \mathbb{R}^3 \mapsto \mathbb{R}^2$. A cubic curve can be obtained from a blossom by specifying four points $\vec{F}(0,0,0), \vec{F}(0,0,1), \vec{F}(0,1,1), \vec{F}(1,1,1)$ (which form a cubic control polygon) and subdividing the surface given by the selected points (known as the De Castelijau's Algorithm). Only these four points are required because of the symmetry property of a blossom.



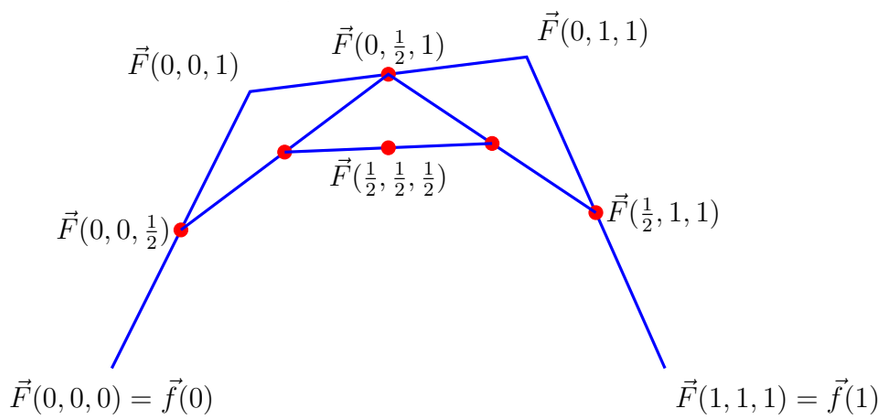
Subdividing the polygon:



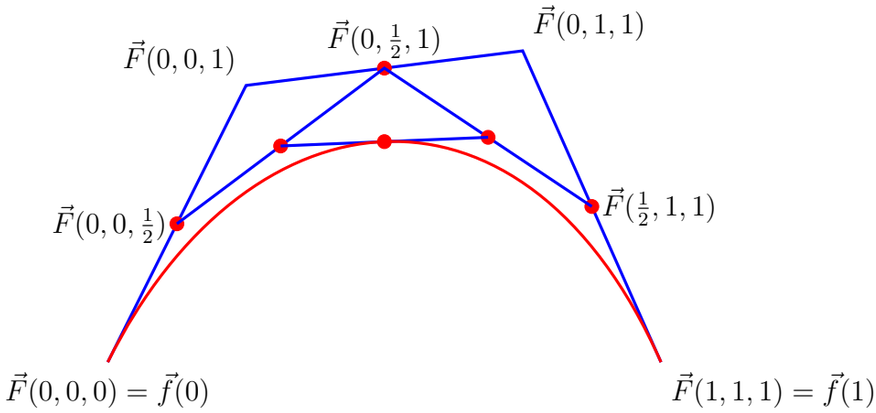
$\vec{F}$'s arguments at the subdivision points can be determined due to the affine property of a blossom. Further subdivision



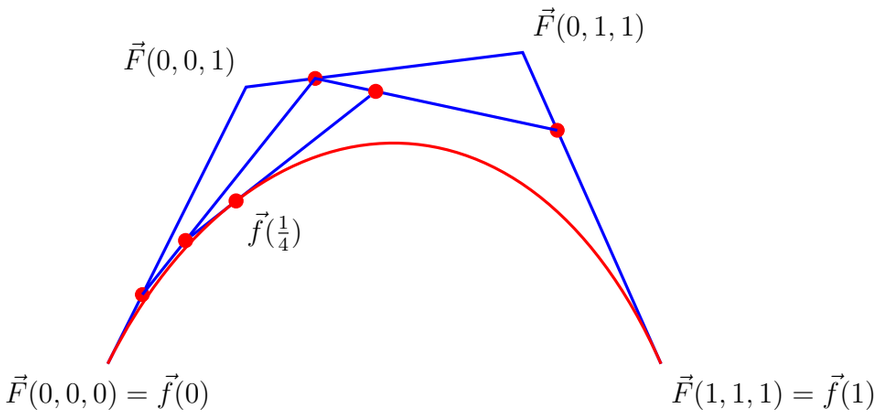
One more subdivision (most recent two subdivision point values omitted because there's no room left)



The final subdivision point can be shown to equal $\vec{f}(\frac{1}{2})$ because of the diagonal property of blossoms, meaning the $\vec{f}$ curve will intersect this third subdivision point:

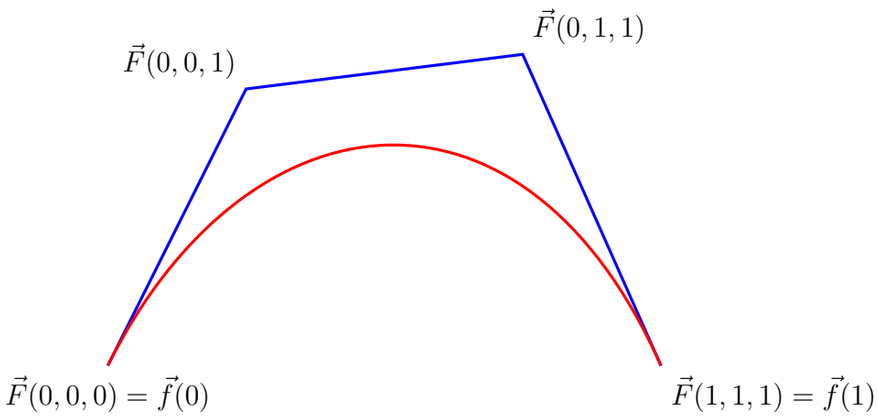


Other values of $\vec{f}$ can be found with subdividing the cubic control polygon on different values. For example, to find $\vec{f}(\frac{1}{4})$, each edge would be divided $\frac{1}{4}$ of the way along it.



1.1.4 Bernstein polynomials

Another basis that can represent cubic curves. Bernstein basis is canonical for Bezier.



The red curve above can be represented as $\vec{f}(t) = \vec{F}(0,0,0)B_0(t) + \vec{F}(0,0,1)B_1(t) + \vec{F}(0,1,1)B_2(t) + F(1,1,1)B_3(t)$, knowing

$$
\begin{aligned}
B_0(t) &= (1-t)^3 = 1 - 3t + 3t^2 - t^3 \\
B_1(t) &= 3t(1-t)^2 = 3t - 6t^2 + 3t^3 \\
B_2(t) &= 3t^2(1-t) = 3t^2 - 3t^3 \\
B_3(t) &= t^3
\end{aligned}
$$

Changing basis from monomial to Bernstein:

$$
\begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}
=
\begin{pmatrix} B_0(t) \\ B_1(t) \\ B_2(t) \\ B_3(t) \end{pmatrix}
$$

And to change from Bernstein to monomial, the inverse of the matrix can be used.

1.1.5 General spline formulation

$\vec{\gamma}(t)$ = Geometry · Spline basis · Monomial basis

- Geometry contains control point coordinates
- Spline basis defines the type of spline (Hermite, Bernstein, etc)
- Monomial basis is a column vector $(1, t, t^2, \cdots, t^n)$

Example of a spline represented in the Bernstein basis:

$$P(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{pmatrix} \begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

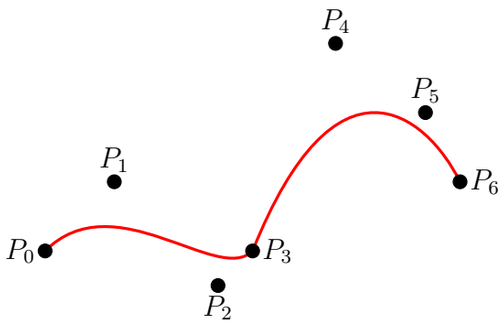
1.1.6 Orders of continuity

C^0 = continuous (seam can be sharp)
G^1 = geometric continuity (Tangent vectors align at the seam)
C^1 = parametric continuity (Same velocity at the seam)
C^2 = curvature continuity (Tangents and tangent derivatives are the same)

1.1.7 Cubic B-splines

≥ 4 control points

Chain together splines in fours, popping the back control point off and pushing the next control point on



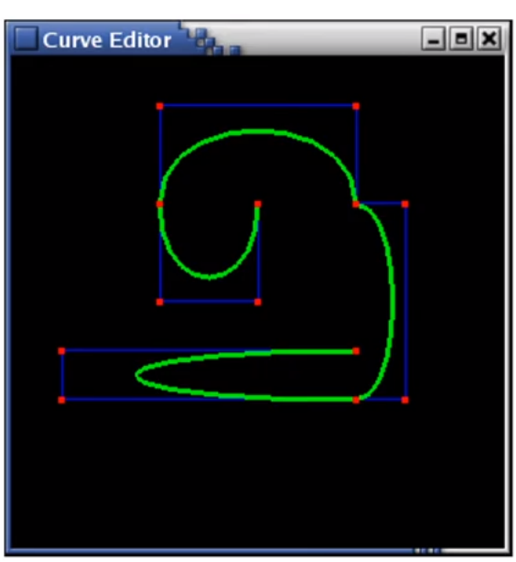
The full curve is composed of many smaller splines, in which the nth spline is formed by points $P_{n..n+3}$. The benefit of this method is that it guarantees C^1 continuity since every spline shares three control points with the one that comes before and after it.

The end points won't connect with this method, but repeating endpoints will fix it.

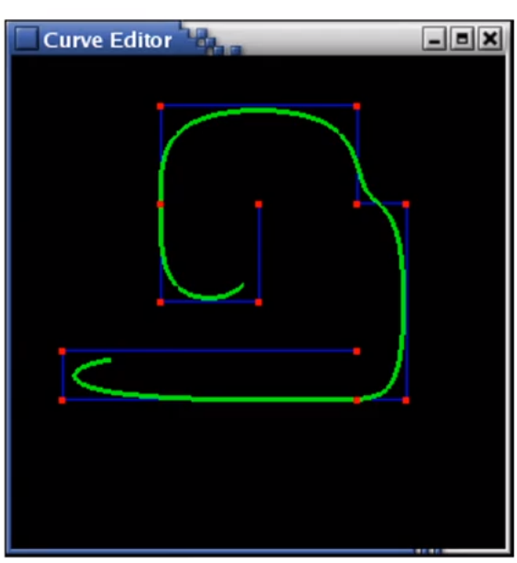
The basis functions as well as the basis conversion matrix for cubic b-spline are listed here:

$$B_1(t) = \frac{1}{6}(1-t)^3$$
$$B_2(t) = \frac{1}{6}(3t^3 - 6t^2 + 4)$$
$$B_3(t) = \frac{1}{6}(-3t^3 + 3t^2 + 3t + 1)$$
$$B_4(t) = \frac{1}{6}t^3$$
$$B_{B-spline} = \frac{1}{6}\begin{pmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1.1.8 Bezier vs B-spline



Bézier



B-Spline

Bezier is derived from Bernstein polynomials, while B-spline uses a different set of basis functions. Additionally, Bezier will try to intersect with control points and only guarantees C^0 continuity, while B-spline does not and guarantees C^1 continuity.

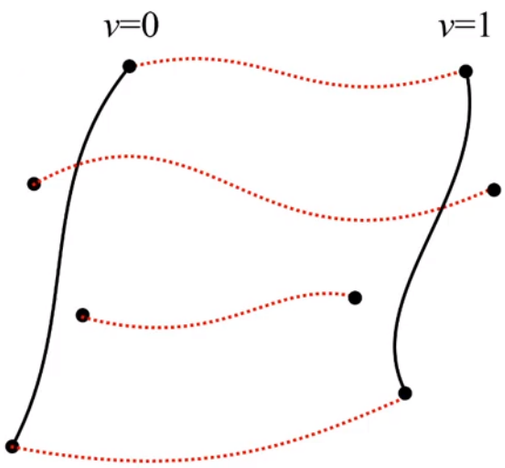
Converting between bezier and b-spline, given G = geometry, B_0 = current basis matrix, $T(t)$ = monomial basis, and B_1 = the basis matrix to convert to:

$$\begin{aligned}\vec{\gamma}(t) &= G \cdot B_0 \cdot T(t) \\ &= G \cdot B_0 \cdot B_1^{-1} \cdot B_1 \cdot T(t) \\ &= (G \cdot B_0 \cdot B_1^{-1}) \cdot B_1 \cdot T(t)\end{aligned}$$

The new geometry matrix is then represented as $G \cdot B \cdot B_1^{-1}$, which shows that to convert between bezier and b-spline, a different set of data is required for the same curve.

1.2 2D Surfaces

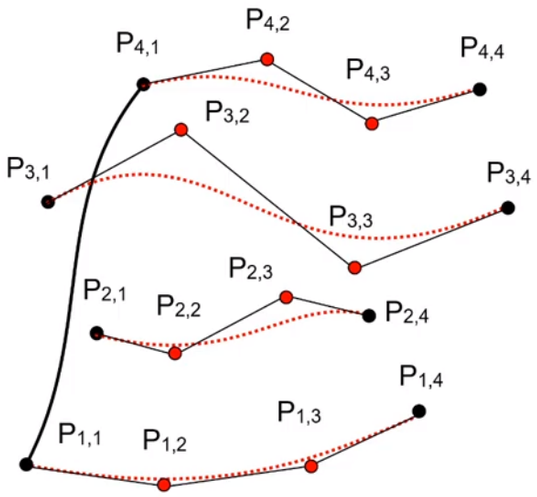
1.2.1 Bicubic Bezier surfaces



Given u changes vertically and v changes horizontally,

$$\begin{aligned}P(u,v) = &(1-u)^3 P_1(v) \\ &+ 3u(1-u)^2 P_2(v) \\ &+ 3u^2(1-u) P_3(v) \\ &+ u^3 P_4(v)\end{aligned}$$

For any constant u for $0 \leq u \leq 1$ there is a bezier curve as a function of v for $0 \leq v \leq 1$ at that u, and vice versa.



$$P(u,v) = \sum_{i=1}^{4} B_i(u) P_i(v)$$

$$P_i(v) = \sum_{j=1}^{4} B_j(v) P_{i,j}$$

Which can be compactly expressed as:

$$P(u,v) = \sum_{i=1}^{4} B_i(u) \left[\sum_{j=1}^{4} P_{i,j} B_j(v) \right]$$
$$= \sum_{i=1}^{4} \sum_{j=1}^{4} P_{i,j} B_{i,j}(u,v)$$

where $B_{i,j}(u,v) = B_i(u)B_j(v)$.

1.2.2 Tangents and Normals for patches

The partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ are tangent vectors.
Normal vector is equal to the cross product of the tangent vectors:

$$\vec{N} = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}$$

$\vec{N}$ is typically not unit, so it needs to be normalized

1.2.3 Matrix notation for patches

$$P(u,v) = \begin{pmatrix} B_1(u) & \cdots & B_4(u) \end{pmatrix} \begin{pmatrix} P_{1,1} & \cdots & P_{1,4} \\ \vdots & \ddots & \vdots \\ P_{4,1} & \cdots & P_{4,4} \end{pmatrix} \begin{pmatrix} B_1(v) \\ \vdots \\ B_4(v) \end{pmatrix}$$

1.2.4 Implicit surfaces

$f(x,y,z) = 0$ is on the surface
$f(x,y,z) < 0$ is inside the surface
$f(x,y,z) > 0$ is outside the surface

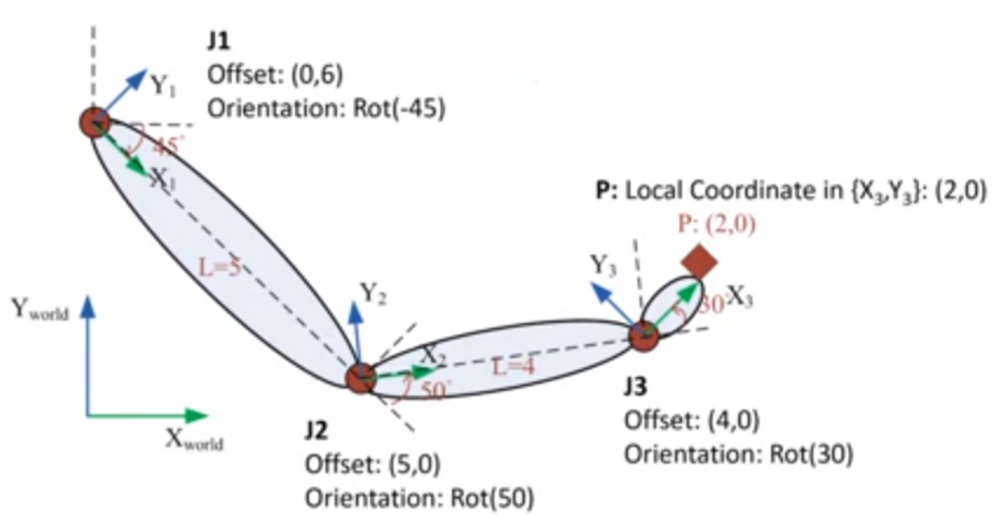
1.3 Hierarchical modeling

1.3.1 Forward kinematics

Specifies base position and joint angles. To compute the final position and orientation of a joint in world coordinates, the motion of all its ancestor joints must be computed as well.

Given values for joint dof (degree of freedom) $\vec{\theta} = [\theta_1, \theta_2, \cdots, \theta_n]^T$, compute the end effector in world space $\vec{\boldsymbol{e}} = [\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_m]^T$.

$\vec{\boldsymbol{e}} = F(\boldsymbol{\theta})$



In the diagram above, P in world coordinates can be calculated with $P_{world} = M_{01}M_{12}M_{23}P_{local}$, or $T(0,6)R(-45)T(5,0)R(50)T(4,0)R(30)P_{local}P_{local}$, T being a transformation matrix and R being a rotation matrix.

1.3.2 Inverse kinematics

Forward kinematics has some issues such as determining what orientation the arm should be in to interact with other objects, which is what inverse kinematics aims to solve.

In forward kinematics, $(\boldsymbol{e}_1, \boldsymbol{e}_2) = F(\theta_1, \theta_2, \theta_3)$; however, in inverse kinematics, $(\theta_1, \theta_2, \theta_3) = G(\boldsymbol{e}_1, \boldsymbol{e}_2)$.

Inverse kinematics is difficult because sometimes there is no solution / multiple solutions to a problem. Typically it can't be analytically solved and requires numerical methods.

- Jacobian iterative method
- Optimization based methods

Given position of a point in local coordinates $\boldsymbol{v}_s$ and desired position in world coordinates $\boldsymbol{v}_w$, find skeleton parameters $\boldsymbol{p}$.

$$\boldsymbol{v}_w = S(\boldsymbol{p})\boldsymbol{v}_s = S(x_h, y_h, z_h, \theta_h, \phi_h, \sigma_h, \theta_t, \phi_t, \sigma_t, \theta_c, \theta_f, \phi_f)\boldsymbol{v}_s$$

1.3.3 Jacobian

$\boldsymbol{e} = F(\boldsymbol{\theta})$ where $\boldsymbol{e}$ is known and $\boldsymbol{\theta}$ is to be solved for.

The Jacobian is the matrix of partial derivatives of $\boldsymbol{F}$: $J = \mathrm{d}F/\,\mathrm{d}\boldsymbol{\theta}$

For $\boldsymbol{F} = [\boldsymbol{F}_1, \boldsymbol{F}_2, \cdots, \boldsymbol{F}_m]^T$ and $\boldsymbol{\theta} = [\theta_1, \theta_2, \cdots, \theta_n]^T$:

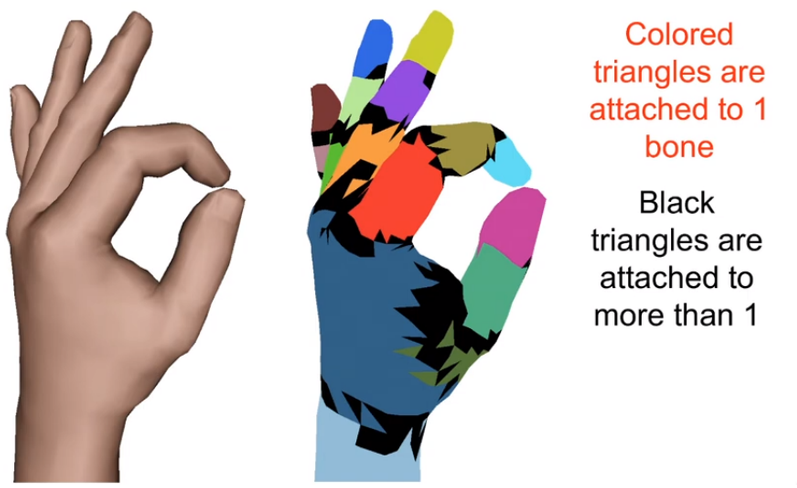
$$J = \begin{bmatrix} \frac{\partial \boldsymbol{F}_1}{\partial \theta_1} & \cdots & \frac{\partial F_1}{\partial \theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial \theta_1} & \cdots & \frac{\partial F_m}{\partial \theta_n} \end{bmatrix}$$

1.4 Skinning

1.4.1 SSD

Skin is made up of vertices, some of which are attached to a single bone, while others are attached to multiple.

Example



1.4.2 Vertex weights

Weight w_{ij} is assigned for each vertex $\boldsymbol{p}_i$ for each bone $\boldsymbol{b}_j$. ("How much will vertex i move with bone j?")

$w_{ij} = 1$ means $\boldsymbol{p}_i$ is rigidly attached to bone $\boldsymbol{b}_j$. Weights should not be negative, and the sum of weights across all bones for each vertex should equal 1.

The number of bones that can influence a single vertex is typically limited to $N = 4$ bones/vertex.

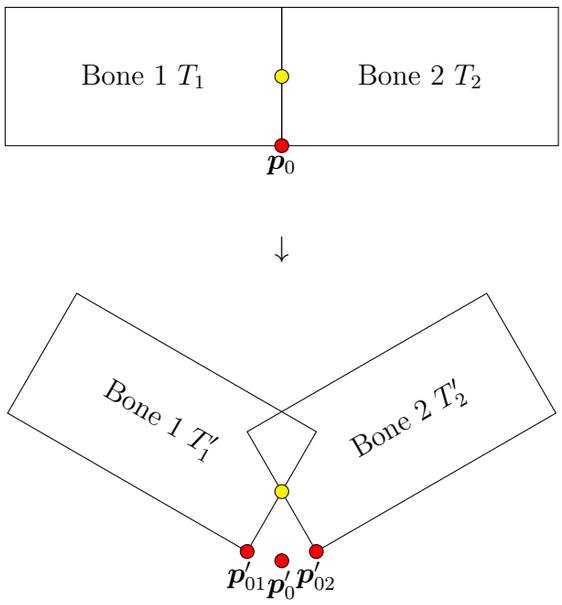
1.4.3 Linear blend skinning

First transform each vertex $\boldsymbol{p}_i$ with each bone as if it were tied to it rigidly, and then blend the results using weights.

Given $\boldsymbol{p}'_{ij}$ is vertex i transformed by bone j, T_j is the current transformation of bone j, and $\boldsymbol{p}'_i$ is the new position of vertex i:

$$\boldsymbol{p}'_{ij} = T_j \boldsymbol{p}_i$$
$$\boldsymbol{p}'_i = \sum_j w_{ij} \boldsymbol{p}'_{ij}$$

Example



Vertex $\boldsymbol{p}_0$ has weights $w_{01} = 0.5$ and $w_{02} = 0.5$. Points $\boldsymbol{p}'_{01}$ and $\boldsymbol{p}'_{02}$ are generated by transforming point $\boldsymbol{p}_0$ using T'_1 and T'_2. $\boldsymbol{p}'_0$ can be determined using $\boldsymbol{p}'_0 = w_{01}\boldsymbol{p}'_{01} + w_{02}\boldsymbol{p}'_{02} = 0.5\boldsymbol{p}'_{01} + 0.5\boldsymbol{p}'_{02}$.

2 Raytracing

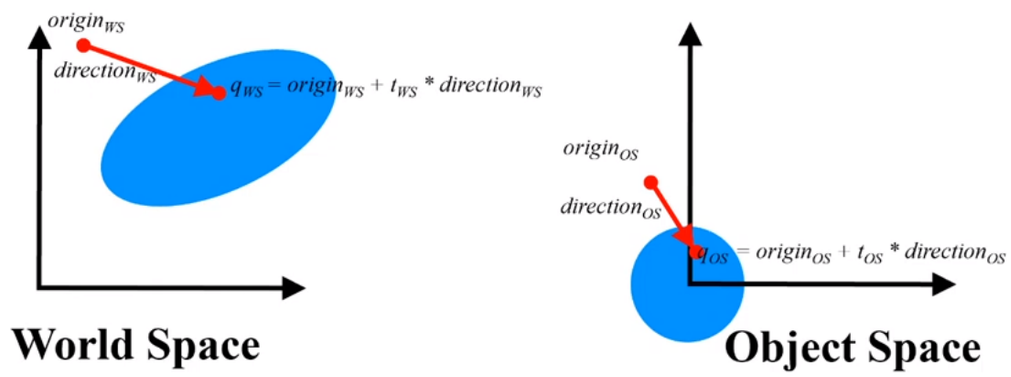
2.1 Geometry representation

A ray is represented as $\boldsymbol{p}(t) = \boldsymbol{R}_o + \boldsymbol{R}_d t$, $\boldsymbol{R}_o$ being the ray origin and $\boldsymbol{R}_d$ being the ray direction. All solutions for t under section 2.1 refer to the t parameter for $\boldsymbol{p}(t)$.

Implicit equations $H(\boldsymbol{P})$ are boolean functions. For example, given the implicit equation $H(\boldsymbol{P}) = \boldsymbol{a} \cdot \boldsymbol{P} = 0$, $H(\boldsymbol{P})$ returns $\boldsymbol{a} \cdot \boldsymbol{P} \leq 0$.

For the computation of intersections with any object, the object should be assumed to be at the origin, with the ray being transformed by the inverse of the object transformation matrix, as well as the ray direction. For point transformation, use $w = 1$, and direction transformation $w = 0$ because direction transformation doesn't include translation.

If ray direction is normalized, $t_{ws} \neq t_{os}$. and must be rescaled after the intersection. If ray direction is not normalized, $t_{ws} = t_{os}$, which is why it's preferred to not normalize transformed ray direction.



To transform object normals properly, use $\boldsymbol{n}_{ws} = (M^{-1})^T \boldsymbol{n}_{os}$.

2.1.1 Sphere

Implicit equation: $H(\boldsymbol{P}) = \boldsymbol{P} \cdot \boldsymbol{P} - r^2 = 0$
Intersection detection (r is sphere radius)

$$a = \boldsymbol{R}_d \cdot \boldsymbol{R}_d$$
$$b = 2(-\boldsymbol{R}_o \cdot \boldsymbol{R}_d)$$
$$c = \boldsymbol{R}_o \cdot \boldsymbol{R}_o - r^2$$
$$d = b^2 - 4ac$$
$$t = \frac{-b \pm \sqrt{d}}{2a}$$

If $d < 0$, there is no intersection. If both solutions for t are negative, there is no intersection. If only one solution for t is positive, that is where the intersection occurs. If both solutions for t are positive, the intersection occurs at the closest t.

2.1.2 Infinite plane

Implicit equation: $H(\boldsymbol{P}) = \boldsymbol{n} \cdot \boldsymbol{P} + D = 0$, with $\boldsymbol{n}$ = plane normal, $D = -(\boldsymbol{n} \cdot \boldsymbol{P}_0)$, $\boldsymbol{P}_0$ = some point on the plane.

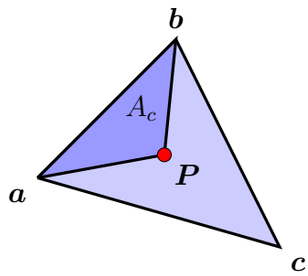
If there is an intersection, $\boldsymbol{n} \cdot \boldsymbol{P} + D$ gives the distance from $\boldsymbol{P}$ to $\boldsymbol{P}_0$.

2.1.3 Triangle

Any point $\boldsymbol{P}$ on a triangle $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ can be represented as $\boldsymbol{P}(\alpha, \beta, \gamma) = \alpha\boldsymbol{a} + \beta\boldsymbol{b} + \gamma\boldsymbol{c}$, with $\alpha + \beta + \gamma = 1$.

If α, β, $\gamma \geq 0$ then the barycentric coordinates are inside the triangle.

Computation of α, β, γ - geometric approach (A = area of entire triangle, A_x = area of subsection of triangle opposite to triangle vertex $\boldsymbol{x}$, $\boldsymbol{P}$ = random point on triangle)



$$\alpha = \frac{A_a}{A}$$
$$\beta = \frac{A_b}{A}$$
$$\gamma = \frac{A_c}{A}$$

Computation of α, β, γ - algebraic approach

Since α, β, and γ sum to 1, $\alpha = 1 - \beta - \gamma$, allowing for $\boldsymbol{P}(\alpha, \beta, \gamma)$ to be written as $\boldsymbol{P}(\beta, \gamma) = (1 - \beta - \gamma)\boldsymbol{a} + \beta\boldsymbol{b} + \gamma\boldsymbol{c}$, or $\boldsymbol{P}(\beta, \gamma) = \boldsymbol{a} + \beta(\boldsymbol{b} - \boldsymbol{a}) + \gamma(\boldsymbol{c} - \boldsymbol{a})$. To simplify, $\boldsymbol{P}(\beta, \gamma)$ can also be written as $\boldsymbol{a} + \beta\vec{e}_1 + \gamma\vec{e}_2$, where $\vec{e}$ is an edge vector.

$$\boldsymbol{P} = \boldsymbol{a} + \beta\vec{e}_1 + \gamma\vec{e}_2$$
$$\vec{e}_1 \cdot (\boldsymbol{P} - \boldsymbol{a}) = \vec{e}_1 \cdot (\beta\vec{e}_1 + \gamma\vec{e}_2)$$
$$\vec{e}_2 \cdot (\boldsymbol{P} - \boldsymbol{a}) = \vec{e}_2 \cdot (\beta\vec{e}_1 + \gamma\vec{e}_2)$$
$$\begin{pmatrix} \vec{e}_1 \cdot (\boldsymbol{P} - \boldsymbol{a}) \\ \vec{e}_2 \cdot (\boldsymbol{P} - \boldsymbol{a}) \end{pmatrix} = \begin{pmatrix} \vec{e}_1 \cdot \vec{e}_1 & \vec{e}_1 \cdot \vec{e}_2 \\ \vec{e}_2 \cdot \vec{e}_1 & \vec{e}_2 \cdot \vec{e}_2 \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \end{pmatrix}$$

Intersection with barycentric triangle - set ray equal to barycentric equation

$$\boldsymbol{P}(t) = \boldsymbol{P}(\beta, \gamma)$$
$$\boldsymbol{R}_0 + t\boldsymbol{R}_d = \boldsymbol{a} + \beta(\boldsymbol{b} - \boldsymbol{a}) + \gamma(\boldsymbol{c} - \boldsymbol{a})$$

Intersection if $\beta + \gamma \leq 1$, $\beta \geq 0$, $\gamma \geq 0$

Separate equation into x, y, z components and turn it into matrix form:

$$\begin{pmatrix} \boldsymbol{a}_x - \boldsymbol{R}_{ox} \\ \boldsymbol{a}_y - \boldsymbol{R}_{oy} \\ \boldsymbol{a}_z - \boldsymbol{R}_{oz} \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}_x - \boldsymbol{b}_x & \boldsymbol{a}_x - \boldsymbol{c}_x & \boldsymbol{R}_{dx} \\ \boldsymbol{a}_y - \boldsymbol{b}_y & \boldsymbol{a}_y - \boldsymbol{c}_y & \boldsymbol{R}_{dy} \\ \boldsymbol{a}_z - \boldsymbol{b}_z & \boldsymbol{a}_z - \boldsymbol{c}_z & \boldsymbol{R}_{dz} \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \\ t \end{pmatrix}$$

Cramer's rule can be used to solve for one variable at a time:

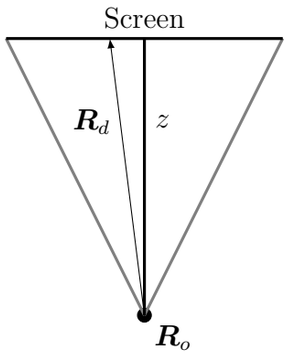
$$A = \begin{pmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{pmatrix}$$
$$\beta = \frac{\begin{vmatrix} a_x - R_{ox} & a_x - c_x & R_{dx} \\ a_y - R_{oy} & a_y - c_y & R_{dy} \\ a_z - R_{oz} & a_z - c_z & R_{dz} \end{vmatrix}}{|A|}$$
$$\gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - R_{ox} & R_{dx} \\ a_y - b_y & a_y - R_{oy} & R_{dy} \\ a_z - b_z & a_z - R_{oz} & R_{dz} \end{vmatrix}}{|A|}$$
$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

2.2 Fundamentals

2.2.1 Raycasting

Raycasting is the process of casting a ray for every pixel on the screen, which means $\boldsymbol{R}_d$ needs to be determined for each pixel.

With z = distance from ray origin to screen, f = fov, p_x and p_y = pixel x and y, w and h = screen width and height:



With θ = x angle, ϕ = y angle:

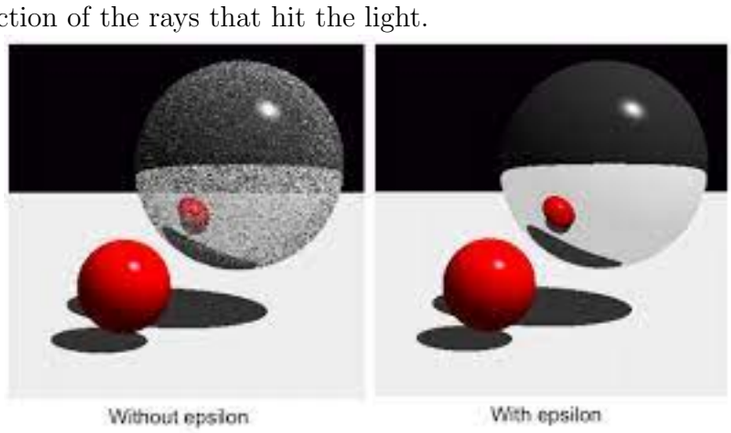
$$\theta = \frac{x}{w}f - \frac{f}{2}$$
$$\phi = \frac{y}{h}f - \frac{f}{2}$$
$$p_x = \sin(\theta)$$
$$p_y = \sin(\phi)$$
$$\boldsymbol{R}_d = [p_x, p_y, 1, 0]^T$$

2.2.2 Shadows

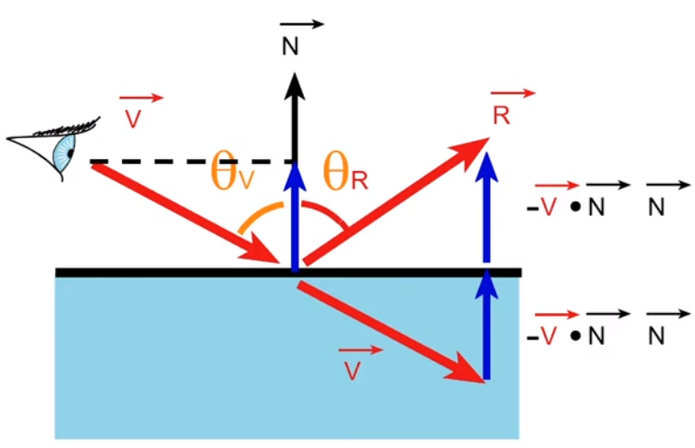
Shadows can be implemented by casting a ray from the original ray intersection point towards each light, and determining if they contribute to the light at that point.

Rays will often self-intersect when checking for shadows on the surface they start off of, which is why the new $\boldsymbol{R}_o$ should be equal to $\boldsymbol{R}_{hit} + \epsilon\boldsymbol{n}$, with $\boldsymbol{R}_{hit}$ = ray object intersection point, $\boldsymbol{n}$ = object surface normal, ϵ = some small value (ex. 0.001), depending on how large objects typically are.

For light sources that have area, cast multiple rays to different spots on the light source and take the fraction of the rays that hit the light.



2.2.3 Reflection



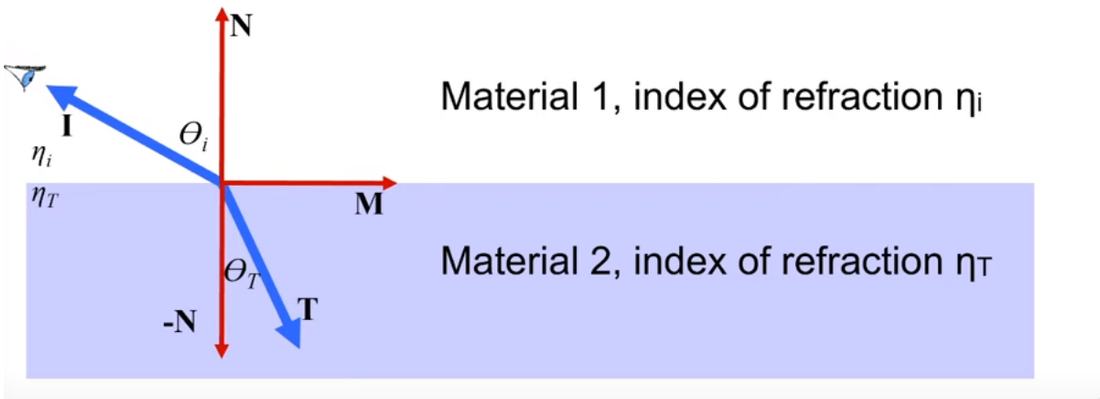
For reflection, the result ray $\boldsymbol{R}$ is expressed as $\boldsymbol{R} = \boldsymbol{V} - 2(\boldsymbol{V} \cdot \boldsymbol{N})\boldsymbol{N}$.

Amount of reflection in traditional raytracing is determined by some constant k_s which gave the final pixel color as $c_p = k_s c_m + (1 - k_s)c_r$, with c_p = final pixel color, c_m = material color, c_r = color the reflected ray returns.

Fresnel discovered that reflections at a grazing angle reflected more, and Schlick approximated $R(\theta) = R_0 + (1 - R_0)(1 - \cos(\theta))^5$, with $R(\theta)$ = reflectiveness, R_0 = base reflectiveness.

Ideally a raytracer should cast multiple rays with slight variation in the same general direction and take the average of their resulting colors when reflecting to render a more realistic surface.

2.2.4 Refraction



Snell-Descartes law states $n_i \sin(\theta_i) = n_T \sin(\theta_T)$

$$\frac{\sin(\theta_T)}{\sin(\theta_i)} = \frac{n_i}{n_T} = n_r$$

n_r is called the relative index of refraction

The vector $\boldsymbol{T}$ can be calculated using $\boldsymbol{T} = \left(n_r(\boldsymbol{N} \cdot \boldsymbol{I}) - \sqrt{1 - n_r^2(1 - (\boldsymbol{N} \cdot \boldsymbol{I})^2)}\right)\boldsymbol{N} - n_r\boldsymbol{I}$

Similar to reflection, refraction should use the same technique of average of multiple rays to render more realistic surfaces.

2.2.5 Lighting

Typically light intensity as a function of distance is expressed as $I(x) = 1/(ax^2 + bx + c)$, with I = intensity, x = distance, a, b, c = positive nonzero constants.

The amount of light energy received by a surface is determined by the angle between the light vector and surface normal: $\hat{\boldsymbol{l}} \cdot \hat{\boldsymbol{n}}$, with $\hat{\boldsymbol{l}}$ = unit light vector pointing towards the light from the surface, $\hat{\boldsymbol{n}}$ = unit surface normal. Combined with falloff, $I_{in} = I_{light}\frac{\cos\theta}{r^2}$, with θ measured between light direction $\boldsymbol{l}$ and surface normal $\boldsymbol{n}$.

Directional lights are lights that are "infinitely far", such as the sun. In that case, it's easier to model light intensity at any point on a surface as $I_{in} = I_{light}\cos\theta$.

Spotlights typically have some angle for which there is no attenuation, and then begin to fall off at some larger angle, combined with distance falloff $1/r^2$.

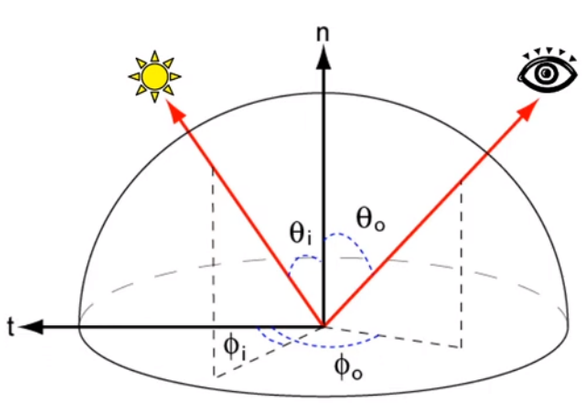
2.2.6 BRDF

BRDF, or Bidirectional Reflectance Distribution Function calculates the ratio of light coming from one direction that gets reflected in another direction

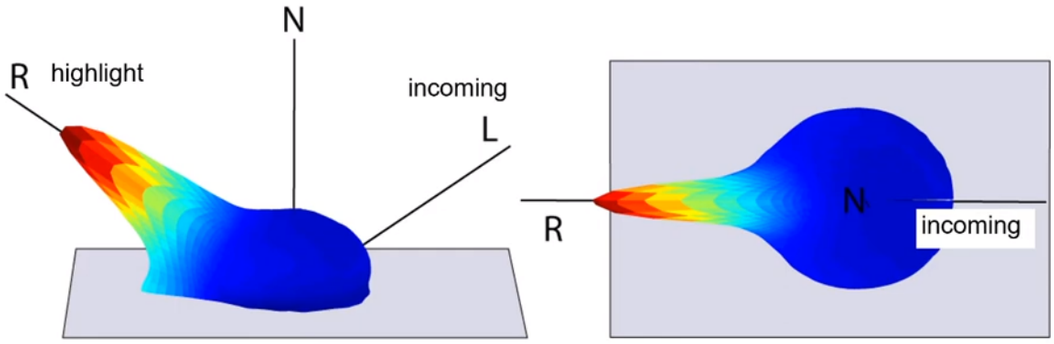
BRDF = $f_r(\theta_i, \phi_i; \theta_0, \phi_0)$ or $f_r(\boldsymbol{l}, \boldsymbol{v})$, with $\boldsymbol{l}$ = light direction, $\boldsymbol{v}$ = view direction

$I_{out} = I_{in}(\boldsymbol{l})f_r(\boldsymbol{l}, \boldsymbol{v})$ where $I_{out}(\boldsymbol{v}) = \frac{I_{light}\cos(\theta_i)}{r^2}f_r(\boldsymbol{l}, \boldsymbol{v})$

$\boldsymbol{l}$ and $\boldsymbol{v}$ must be in a local coordinate system.



BRDF will typically be graphed with $\boldsymbol{l}$ constant and then plot I_{out} as a function of $\boldsymbol{v}$. In the plot below, the distance from the center of the sphere represents f_r.



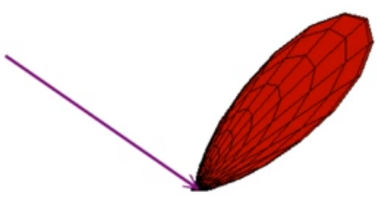
Example: Plot of "PVC" BRDF at 55° incidence

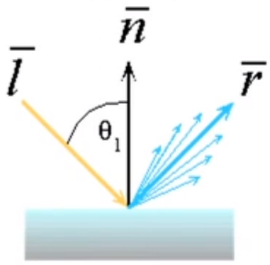
When keeping $\boldsymbol{l}$ and $\boldsymbol{v}$ fixed, if rotation of the surface around the normal does not change the reflection , the material is isotropic (ex. sphere). Surfaces with strongly oriented microgeometry elements are anisotropic (ex. brushed metal, hair, fur).

In ideal diffuse reflectance (reflectance is the same anywhere across the object), BRDF is a constant $f_r(\boldsymbol{l}, \boldsymbol{v}) = \text{const}$, where the constant is typically written as ρ/π where ρ is the albedo, a coefficient between 0 and 1 that says what fraction of light is reflected. When $\rho = 0$, the surface completely absorbs the light and no light comes out, with more light being reflected as ρ increases. The constant can also be written as $\boldsymbol{k}_d$, with one ρ value for each RGB channel, giving $\boldsymbol{k}_d = (\rho_{red}/\pi, \rho_{green}/\pi, \rho_{blue}/\pi)$.

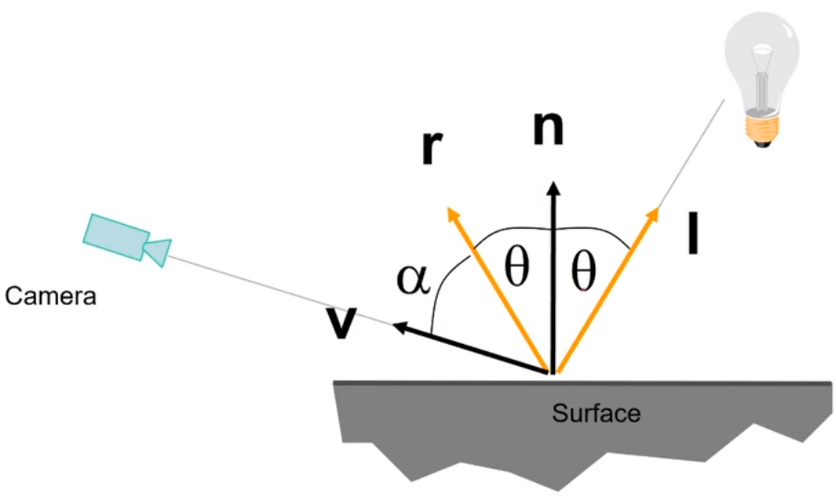
The function for ideal diffuse reflectance is given by $\boldsymbol{L}_o = \boldsymbol{k}_d \max(0, \boldsymbol{n} \cdot \boldsymbol{l}) L_i / r^2$, with $\boldsymbol{L}_o$ = shaded color, $\boldsymbol{n}$ = surface normal, $\boldsymbol{l}$ = light direction (towards light), L_i = light intensity, r = distance from point on surface to light.

In ideal specular reflectance, the BRDF function will typically look like a lobe:



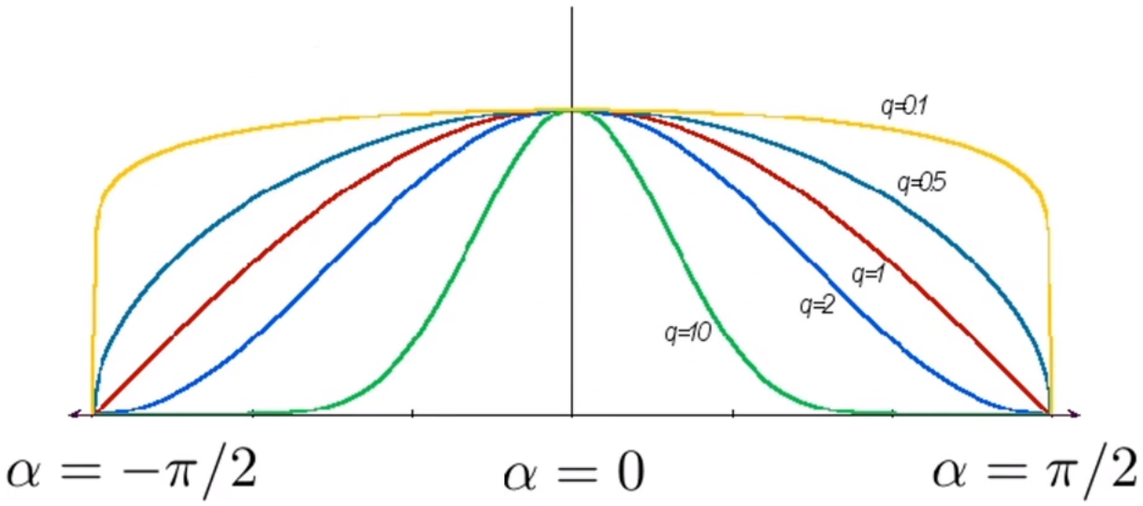


The amount of light reflected is dependent on the angle between the view direction and the reflected light direction:



The function for ideal specular reflectance can then be given by $\boldsymbol{L}_o = \boldsymbol{k}_s(\boldsymbol{v} \cdot \boldsymbol{r})^q L_i / r^2$, with $\boldsymbol{L}_o$ = output color to screen, $\boldsymbol{k}_s$ = specular reflection coefficient, q = specular exponent, L_i = light intensity, r = distance from point on surface to light.

The specular exponent's effect can be seen in the graph below, with α being the angle between $\boldsymbol{v}$ and the $\boldsymbol{r}$ vector (above diagram) and the vertical axis representing the output intensity:



2.2.7 Phong lighting model

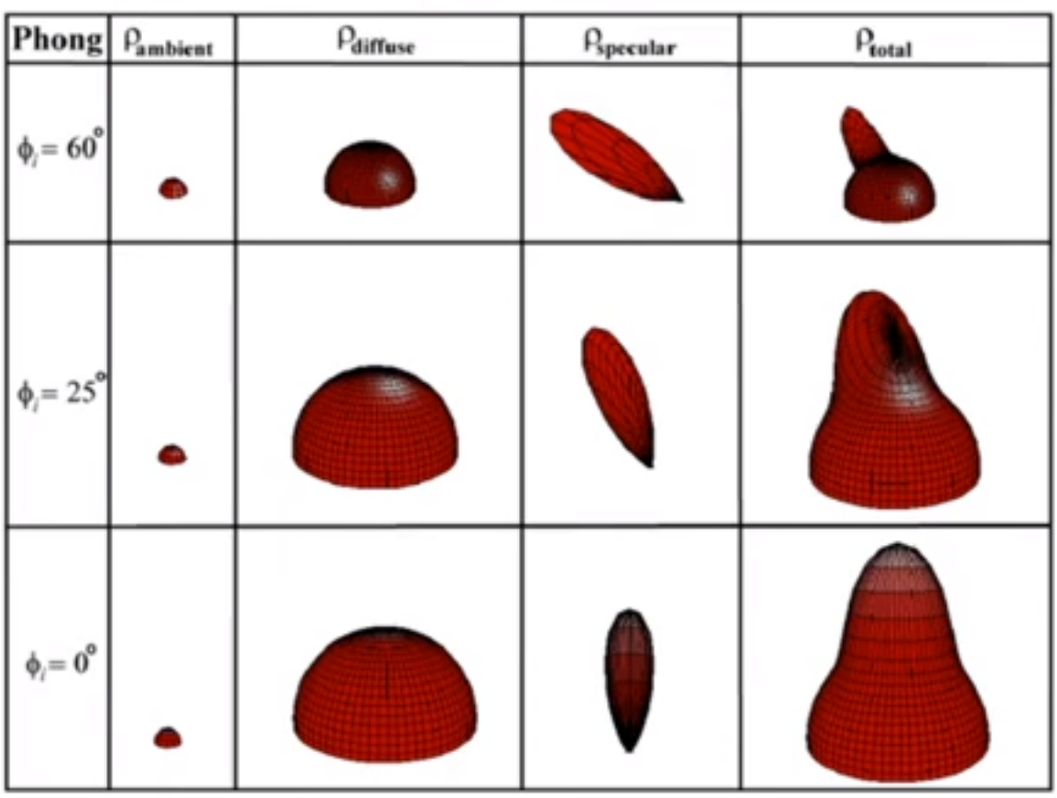
Phong lighting model is made up of ambient, diffuse, and specular light. Ambient light is the base color of objects when all lights are off, diffuse is light intensity and angle dependent which gives a rough surface look, specular light adds shininess.

The equation for the phong lighting model can be given by:

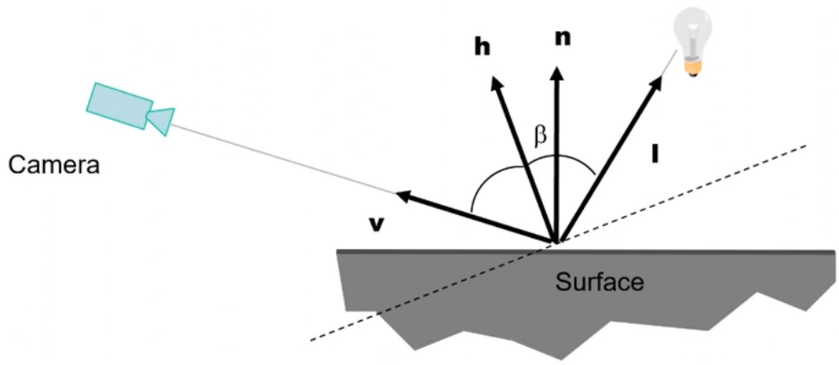
$$\boldsymbol{L}_o = (\boldsymbol{k}_a + \boldsymbol{k}_d(\boldsymbol{n} \cdot \boldsymbol{l}) + \boldsymbol{k}_s(\boldsymbol{v} \cdot \boldsymbol{r})^q) \frac{L_i}{r^2}$$

What the variables represent are listed in the previous BRDF subsection.

The chart below shows the BRDF graphs of lights at different ϕ angles from the vertical axis:



One issue with phong is that it doesn't capture higher specular reflection at grazing angles, which is what the Blinn-Torrance phong variation attempts to solve.

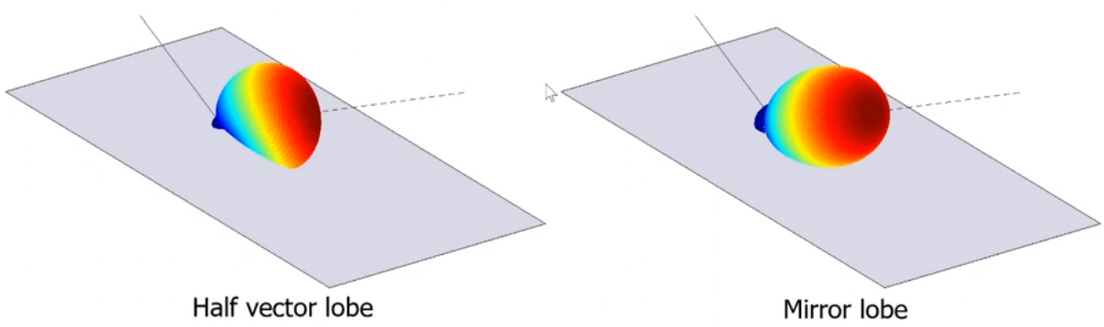


$$\boldsymbol{L}_{o,s} = \boldsymbol{k}_s(\boldsymbol{n} \cdot \boldsymbol{h})^q \frac{L_i}{r^2}$$

where

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

Specular BRDF lobe comparison (Blinn-Torrance vs phong):



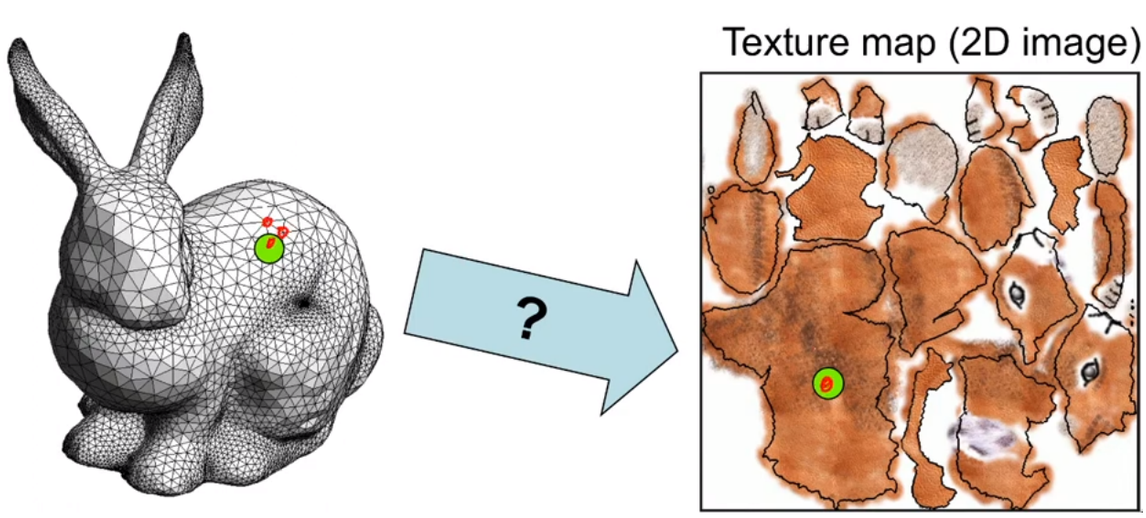
The half vector lobe is preferable because it's more consistent with what is observed in real world measurments.

2.2.8 Normal interpolation

Normal interpolation gives the illusion that a triangle mesh is smoother than it actually is. For any point on a triangle in the triangle mesh, get its barycentric coordinates and multiply them by the vertex normals of the three triangle vertices. Remember to renormalize normals after interpolation. $\boldsymbol{n}(\alpha, \beta, \gamma) = \alpha\boldsymbol{n}_0 + \beta\boldsymbol{n}_1 + \gamma\boldsymbol{n}_2$

2.2.9 Textures

Each vertex on a triangle will contain (u, v) coordinates which indicate where it is on the texture, which can then be interpolated with barycentric coordinates to give the texture at any point on the triangle.

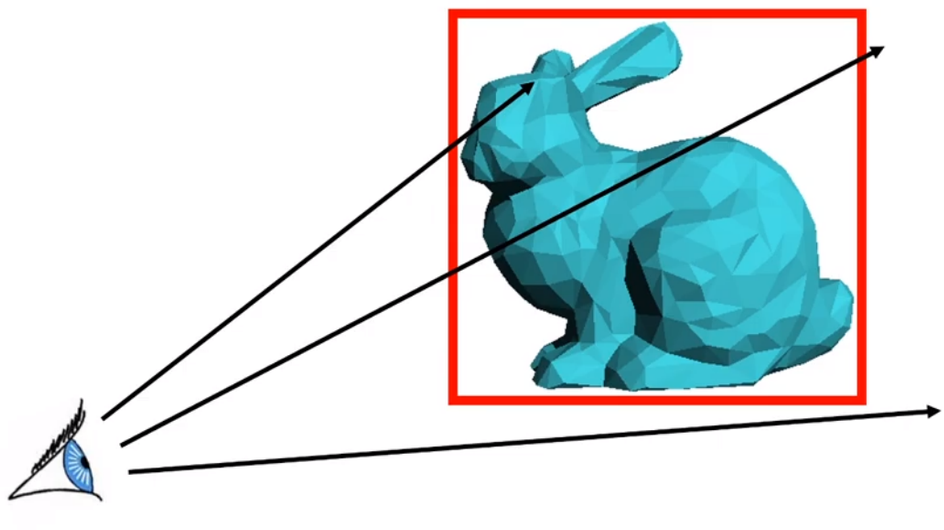


This technique of texture interpolation can also be used for normals.

2.3 Optimization

2.3.1 Bounding volume: Axis-aligned bounding box

A bounding volume encloses an object entirely, and if a ray doesn't intersect the bounding volume it will not attempt to check for intersections with the more complicated object.



Box can be described as $(x_1, y_1, z_1) \rightarrow (x_2, y_2, z_2)$ where $x_1 < x_2$, $y_1 < y_2$, $z_1 < z_2$. To check if a ray intersects with the bounding box, check intervals of t for which the ray is within the range of the bounding box.

After the three intervals of t for dimensions x, y, and z have been determined, the starting point of intersection t_{start} is the max of the mins and the ending point of the intersection t_{end} is the min of the maxes. If $t_{start} > t_{end}$, there is no intersection.

2.3.2 Bounding volume hierarchy (BVH)

A bounding volume might not be enough for optimization if the object it encloses has a large number of triangles, which a BVH solves by splitting objects into two and calculating bounding volumes of the smaller sections of the object. Each sub bounding volume will then create its own two smaller bounding volumes, and eventually a binary tree will be created of bounding volumes.

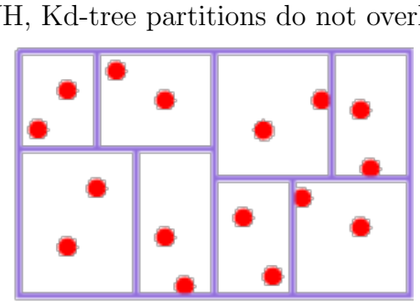
The best way to split bounding volumes is typically to sort all triangles according to one dimension and splitting that list in half to create two new bounding volumes.

When intersecting with bounding volumes, the first bounding volume intersected is not necessarily the only one that needs to be intersected. What determines that is if the ray intersects with a triangle.

Bounding volumes are allowed to intersect with each other in a BVH.

2.3.3 Kd-trees

Each space splits itself into two parts somewhere that neatly divides objects into two spaces. Unlike BVH, Kd-tree partitions do not overlap.



Kd-tree construction

- Start with scene AABB
- Decide which dimension to split on (longest dimension is typically good)
- Decide distance to split
  - It may be impossible to get a clean split, in which case any objects overlapping the boundary should be considered to be part of both spaces
- Stop when a minimum number of primitives is reached
  - Other criteria can be used as well

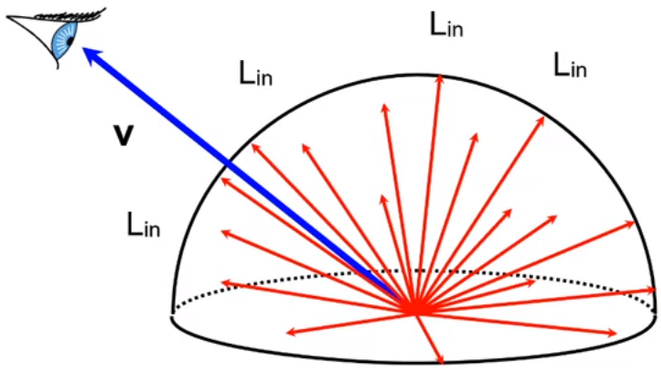
Kd-tree traversal

- If leaf, intersect with contained primitives. If no intersection continue as normal.
- If any of the two children, intersect with those children recursively

2.4 Advanced techniques

2.4.1 Global illumination

In simple raytracing, a ray is cast towards each light from the intersection point. In global illumination, a ray is cast towards every direction on the hemisphere.



Reflectance equation, with L_{out} = outgoing light in direction $\boldsymbol{v}$, $\boldsymbol{x}$ = intersection point,

$$L_{out} = \int_{\Omega} L_{in}(\boldsymbol{l}) f_r(\boldsymbol{x}, \boldsymbol{l}, \boldsymbol{v}) \cos\theta \, \mathrm{d}\boldsymbol{l}$$

The full rendering equation just includes light sources in E_{out}, which is 0 if not a light source:

$$L_{out} = \int_{\Omega} L_{in}(\boldsymbol{l}) f_r(\boldsymbol{x}, \boldsymbol{l}, \boldsymbol{v}) \cos\theta \, \mathrm{d}\boldsymbol{l} + E_{out}(\boldsymbol{x}, \boldsymbol{v})$$

Analytical solution to rendering equation is usually impossible; however, there are many ways to solve it approximately.

In Monte-Carlo raytracing, every ray intersection generates multiple new rays, which then generate multiple new rays until a recursion limit is hit, and then the contribution of each light source is added. If a ray directly intersects with a light source, the light contribution is directly added and the ray is stopped. However, this method is extremely computationally expensive since the number of rays grows exponentially with each intersection, which is what Monte- Carlo pathtracing solves. Monte-Carlo pathtracing casts a single ray from each intersection point but casts multiple rays for each pixel ($>$ 10 samples/pixel is reasonable).