CS-536 : Notes

Charlie Stuart : src322

Fall 2021

Note: Section numbers are completely made up based on my brain understanding things

Contents

1 Math Review

1.1 Vectors, Points, Scalars

Affine Geometry : I can't find a sane definition of this, but vector geometry basically

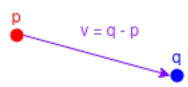
Point : Has 1 value for each dimension, represents a single spot

Vector : An n-tuple of real numbers. Uses $\vec{v}$ notation

Scalar : An value use to scale a vector

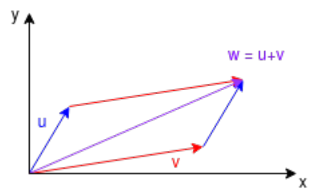
1.1.1 Operations

Point-Point Difference : Given points p and q, $q - p$ creates a vector v

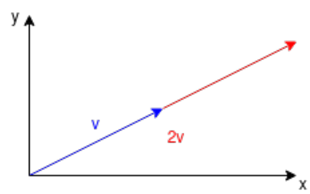


Point-Vector Difference : Given point p and vector v, $p + v$ and $p - v$ create a point q

Vector-Vector Addition : Given vectors v and u, $v + u$ and $v - u$ create a vector w



Scalar Multiplication : Given a vector v and a scalar s, sv creates a vector w



Linear Combination :

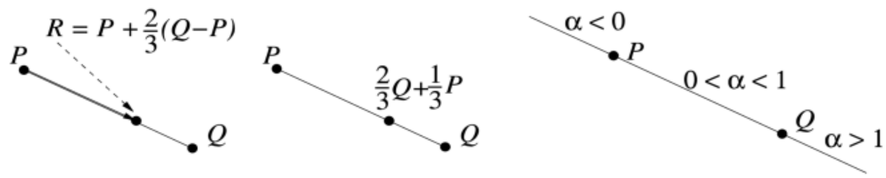
Given vectors $v_1, v_2, ..., v_n$ and scalars $\alpha_1, \alpha_2, ..., \alpha_n$ then $\alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$ is the linear combination of them.

Affine Combinations :

Given that $v_1, v_2, ..., v_n$ are vectors and $\sum_i^n \alpha_i = 1$ then:

$\alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$

For Example: $R = (1 - \alpha)P + \alpha Q$



Dot Product : Given vectors u and v,

$$u \cdot v = \sum_{i=1}^{n} u_i v_i$$

Euclidian Distance : Distance from (x, y) to $(0, 0)$

$$\begin{aligned}
&\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} && \text{Distance between two points} \\
&\sqrt{(x - 0)^2 + (y - 0)^2} && \text{Plug in values} \\
&\sqrt{x^2 + y^2} && \\
&\sqrt{x_1^2 + x_2^2 + ... + x_n^2} && \text{Generalize for } n \text{ dimensions} \\
&\sqrt{x_1 x_1 + x_2 x_2 + ... + x_n x_n} && \\
&\sqrt{\vec{x} \cdot \vec{x}} &&
\end{aligned}$$

This calculates the length of x Notation: $||\vec{x}||$

Normalization : Normalizing a vector makes it have length 1 but keeps its direction

$$\hat{v} = \frac{\vec{v}}{||\vec{v}||}$$

Orthogonal : Two vectors are orthogonal if $\vec{u} \cdot \vec{v} = 0$

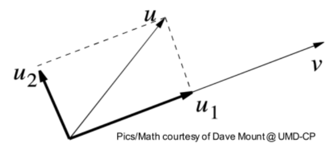
Angle Between Vectors : Given vectors u and v that share a starting point, the angle θ between them is

$$\theta = \cos^{-1} \hat{u} \cdot \hat{v}$$

Projection of Vectors : Given vector u projected onto vector v

$$\vec{u_1} = \frac{\vec{u} \cdot \vec{v}}{\vec{v} \cdot \vec{v}} \vec{v}$$

$$\vec{u_2} = \vec{u} - \vec{u_1}$$



1.2 Matrixes

Matrix : An n dimensional vector

1.2.1 Operations

Identity : The identity matrix of a square nxn matrix is one where there's 1 in the diagonal and 0s everywhere else

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

Addition :

Matrix addition is commutative and associative

Given matrices A and B of the same dimensions, then $A + B$ is:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} + \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00} + b_{00} & a_{01} + b_{01} \\ a_{10} + b_{10} & a_{11} + b_{11} \end{bmatrix}$$

Scalar Multiplication : Given a matrix A and scalar c, then cA is

$$c\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} = \begin{bmatrix} ca_{00} & ca_{01} \\ ca_{10} & ca_{11} \end{bmatrix}$$

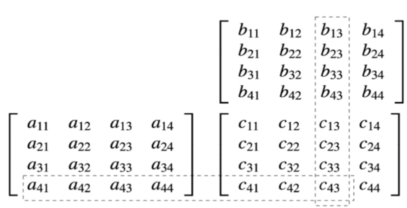
Matrix Multiplication :

Matrix multiplication is NOT commutative

Given matrices A and B, then each element of C is found with:

$$c_{ij} = \sum_{s=0}^{n} a_{is} b_{sj}$$

To visualize:



Determinant : Given the following 2x2 matrix:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The determinant is $ad - bc$

Helps us find a vector orthogonal to two other vectors and to determine the plane of a polygon

Cross Product : Given two non-parallel vectors A and B then $A \times B$ gives a vector C that is orthogonal to A and B

$$A \times B = C = [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)]$$

Matrix Transpose : Swap rows and columns

$$A = \begin{bmatrix} a & b & c \end{bmatrix}$$
$$A^T = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

Some notes:

- $(A^T)^T = A$
- $(A + B)^T = A^T + B^T$
- $(cA)^T = c(A^T)$
- $(AB)^T) = B^T A^T$

Matrix Inverse : Given a square matrix A, if $AB = BA = I$ then $B = A^{-1}$

1.3 Calculus

1.3.1 Derivatives

$$
\begin{aligned}
f(x) &= \alpha x^n \\
\frac{df(x)}{dx} &= \alpha n x^{n-1}
\end{aligned}
$$

1.3.2 Partial Derivatives

$$
\begin{aligned}
f(x) &= \alpha x^n y^m \\
\frac{\partial f(x)}{\partial x} &= \alpha n x^{n-1} y^m
\end{aligned}
$$

1.4 Combinatorics

Combinations : $\binom{x}{y}$, x choose y

$$\binom{x}{y} = \frac{x!}{y!(x-y)!}$$

2 Graphics Basic Vocab

Geometric Modeling : Mathematics and algorithms that define 2D and 3D geometric objects

Lighting and Shading : Math, physics, and algorithms that specify how light interacts with matter'

Rendering : Algorithms that take geometry, lighting, shading, and viewing information and generate an image

Visualization : Techniques for visually communicating and exploring scientific, medical, or abstract data

Perception : Study of how humans perceive light and information

Animation : Algorithms for making models change over time

Simulation : Using physics to make models move

Raster 2D Graphics : Pixels

- X11 bitmap (XBM), X11 pixmap (XPM), GIF, TIFF, PNG, JPG
- Lossy, jaggies when transforming, good for photos

Vector 2D Graphics : Drawing Instructions

- Postscript, CGM, Fig, DWG
- Non-lossy, smooth when scaling, good for line art and diagrams

3 Curves

3.1 Fuctional Representations

Polynomial : Linear combination of integer powers of x, y, z

Polynomial Degree : I found a few different definitions here

- Online says: the highest degree of a polynomial. $x^2y^3 + x^2 = 0$ has degree 5
- The slides says: the total sum of powers but the example is wrong $x^2 + y^2 + z^2 - r^2 = 0$ has degree 6

3.1.1 Explicit Functions

Representing one variable with another like $y = x^2$. Works if $\exists x$ there's only one y. What if I have a sphere? $z = \pm\sqrt{r^2 - x^2 - y^2}$

3.1.2 Implicit Functions

Curves and surfaces are represented as "the zeros"

Good for representing $(n-1)$ dimensional objects in nD space

Sphere: $x^2 + y^2 - z^2 - r^2 = 0$

3.1.3 Parametric Functions

2D Curve : Two functions of one parameter $(x(u), y(u))$

3D Curve : Three functions of one parameter $(x(u), y(u), z(u))$

3D Surface : Three functions of two parameters $(x(u,v), y(u,v), z(u,v))$

So the sphere example here is not algebraic, but is parametric

$$x(\theta, \phi) = \cos\phi\cos\theta$$
$$y(\theta, \phi) = \cos\phi\sin\theta$$
$$z(\theta, \phi) = \sin\phi$$

3.1.4 Comparisions

- Explicit isn't used in graphics
- Implicit is good for:
  - Computing ray/surface tension

- Point inclusion (inside/outside test)
- Mass and volume properties

- Parametric is good for:
  - Subdivision, faceting for rendering
  - Surface and area properties
- Parametric is popular in graphics
- Mathematical representation can be very complex
- Function to shape isn't obvious

3.1.5 Points and Curves

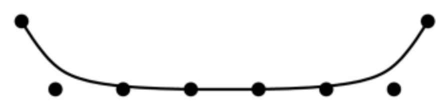
To deal with complex formulas for curves, we use **curve control points**

Lagrangian Interpolation

- $n + 1$ points for a polynomial of degree n
- Curve wiggles through each point
- Not good for smooth flat curves

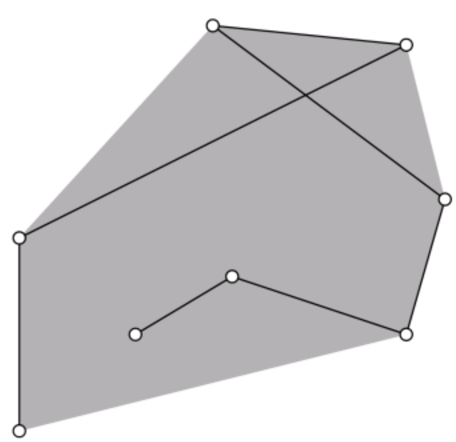
Approximation

- Points are weights that tug on the curve or surface



3.1.6 Convex Hulls

Convex Hull : The smallest convex container of a set of points



3.2 Parametric Curves

Basic Representation : $x = x(t), y = y(t)$

Properties:

- Individual functions are single valued
- Approximations are done with piecewise polynomial curves
- Each segment is given by two cubic polynomials (x, y)
- Concise

3.2.1 Mathematical Definition

The cubic polynomials that define a parametric curve segment is given by:

$$Q(t) = [x(t)y(t)z(t)]^T$$

and we know that

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y \\ z(t) &= a_z t^3 + b_z t^2 + c_z t + d_z \\ 0 \le t &\le 1 \end{aligned}$$

and that coefficients are represented with the matrices:

$$C = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix}$$
$$T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}^T$$

Then we know that:

$$Q(t) = C \cdot T$$

$Q(t)$ can be defined with four constraints if we rewrite $C = G \cdot M$

Geometry Matrix : G, a four element constraint matrix

Basis Matrix : M, a 4x4 matrix

$Q(t)$ is now a weighted sum of the columns of the geometry matrix, G, each of which represents a point of vector in 3-space. It is expanded as:

$$Q(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} G_1 & G_2 & G_3 & G_4 \end{bmatrix} \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

We can now manually multiply this out to get the following weighted sums of the elements:

$$\begin{aligned} x(t) &= (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41}) g_{1x} + (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42}) g_{2x} \\ &+ (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43}) g_{3x} + (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44}) g_{4x} \\ y(t) &= (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41}) g_{1y} + (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42}) g_{2y} \\ &+ (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43}) g_{3y} + (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44}) g_{4y} \\ z(t) &= (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41}) g_{1z} + (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42}) g_{2z} \\ &+ (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43}) g_{3z} + (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44}) g_{4z} \end{aligned}$$

Blending Functions : B, the cubic polynomial weights in t, $B = MT$ so $Q(t) = G \cdot B$

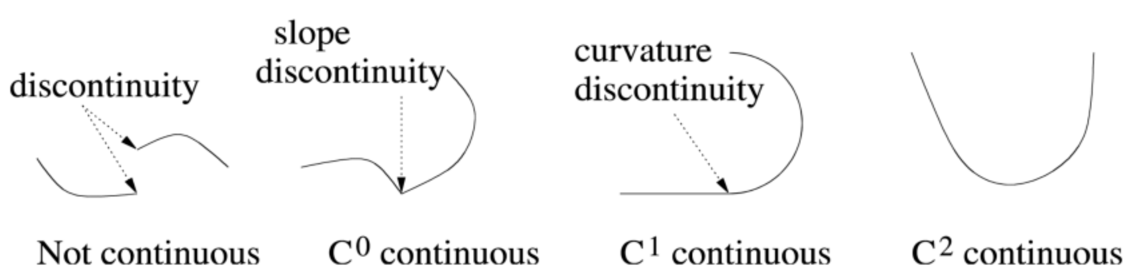
M and G matrices vary by curve

3.3 Continuity

Continuity : Two curves are C^i continuous at a point p iff the i-th derivatives of the curves are equal at p

Geometric Continuity : G^i, endpoints meet and the tangent vectors' directions are equal

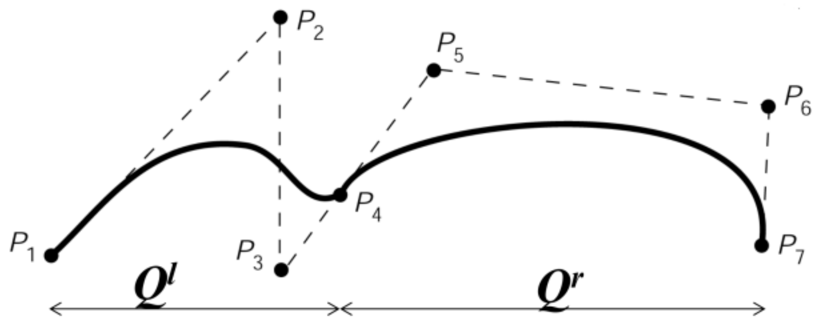
Parametric Continuity : C^i, endpoints meet and the tangent vectors' directions and magnitudes are equal, $G^i \in C^i$



Given two curves Q^l and Q^r, the condition for C^0 and C^1 continuity is that the end points and their tangent vectors are equal

$$Q^l(1) = Q^r(0)$$

$$\frac{dQ^l}{dt}(1) = \frac{dQ^r}{dt}(0)$$



We can define the parametric tangent vector of a curve with the derivative of $Q(t)$ as shown:

$$\frac{d}{dt}Q(t) = Q'(t) = \begin{bmatrix} \frac{d}{dt}x(t) \\ \frac{d}{dt}y(t) \\ \frac{d}{dt}z(t) \end{bmatrix} = \frac{d}{dt}C \cdot T = C \cdot \begin{bmatrix} 3t^2 \\ 2t \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3a_x t^2 + 2b_x t + c_x \\ 3a_y t^2 + 2b_y t + c_y \\ 3a_z t^2 + 2b_z t + c_z \end{bmatrix}$$

This means, looking back at the above image, in order to compute continuity for $Q(t)$, we need to compute it for each component individually in 3D space

$$x^l(1) = x^r(0) = P_{4_x}$$

$$\frac{d}{dt}x^l(1) = 3(P_{4_x} - P_{3_x})$$

$$\frac{d}{dt}x^r(0) = 3(P_{5_x} - P_{4_x})$$

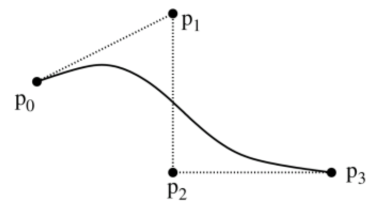
Rinse and repeat with y and z

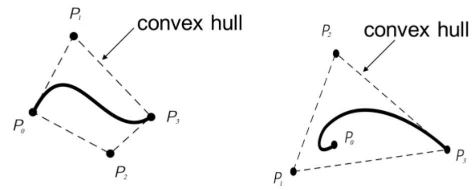
3.4 Bézier Curves

Bézier Curves : Defined with two end points plus two control points for the tangent vectors

$\mathbf{P_0}$ **:** Start point
$\mathbf{P_3}$ **:** End point
$\overline{\mathbf{P_0P_1}}$ **:** Tangent at $\mathbf{P_0}$
$\overline{\mathbf{P_2P_3}}$ **:** Tangent at $\mathbf{P_3}$
Bézier Geometry Matrix : $G_B = \begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix}$
Bézier Basis Matrix : M_B

$Q(t) = G_B \cdot M_B \cdot T$





3.4.1 Bernstein Polynomials

General Form: : i-th Bernstein polynomial for a degree k Bézier curve:

$$b_{ik}(u) = \binom{k}{i}(1-u)^{k-i}u^i$$

Properties:

- Invariant under transformations
- Form a partition of unity
  - **Partition Of Unity :** Given x functions, $\sum_{i=0}^{x} f_i(t) = 1 \forall t$
- Low degree Bernstein Polynomials (BPs) can be written as high degree BPs
- BP derivatives are a linear combination of BPs
- Form a basis for space of polynomials with degree $\leq k$

Cubic Bernstein Blending Functions : Represent the blending proportions among the control points:

$$\begin{aligned}
b_{03}(u) &= (1-u)^3 \\
b_{13}(u) &= 3u(1-u)^2 \\
b_{23}(u) &= 3u^2(1-u) \\
b_{33}(u) &= u^3
\end{aligned}$$

Note: The coefficients follow Pascal's triangle

Which gives us:

$$
\begin{aligned}
Q(t) &= G \cdot B \\
Q(t) &= b_{03}(t)G_1 + b_{13}(t)G_2 + b_{23}(t)G_3 + b_{33}(t)G_4 \\
Q(t) &= (1-u)^3 P_0 + 3u(1-u)^2 P_1 + 3u^2(1-u)P_2 + u^3 P_3
\end{aligned}
$$

3.4.2 General Form

Given:

- Control points $P_0, P_1, ..., P_k$
- $0 \leq t \leq 1$

Then:

$$Q(t) = \sum_{i=0}^{k} P_{i+1} \binom{k}{i} (1-t)^{k-i} t^i$$

3.4.3 Properties

- k control points defines a single curve of degree $k-1$
- Affine invariance
  - Invariance under affine parameter transformations
- Convex Hull Property
  - Curve lies completely within the convex hull of control points
- Endpoint interpolation
- Intuitive for design
  - Curve mimics the control polygon

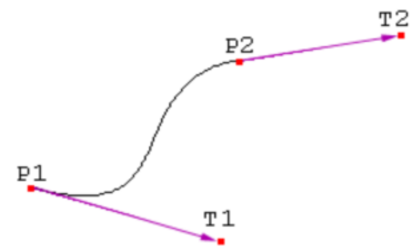
3.4.4 Issues

- More complex curves require more control points
  - Potentially high degree polynomial with many wiggles
- Bézier blending functions have global support over the whole curve
  - Moving one point changes the whole curves

Can be improved by linking many low degree Bézier curves together

3.5 Hermite Curves

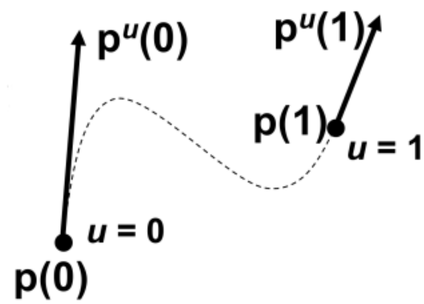
- 3D curve of polynomial bases
- Geometrically defined by positions and tangents at end points
- Not guaranteed to exist within a convex hull
- Supports tangent-continuous (C^1) composite curves



3.5.1 Algebraic Form

Knowing that derivatives give us a tangent line at a point, we can build a line in the following form:

$\mathbf{P(u)}$: The line over time u
$\mathbf{P^u(u)}$: The derivative of $P(u)$
: $P^u(u) = P'(u) = \frac{dP}{du}(u)$
$\mathbf{P(0)}$: Control/End Point 0
$\mathbf{P(1)}$: Control/End Point 1
$\mathbf{P^u(0)}$: Control Tangent Vector 0
$\mathbf{P^u(1)}$: Control Tangent Vector 1



The curve is very simple, just a parametric algebraic polynomial. A cubic curve is given by the following formulas with coefficients a, b, c, d:

$$P(u) = au^3 + bu^2 + cu + d$$
$$P^u(u) = P'(u) = 3au^2 + 2bu + c$$

We can now calculate the control points and control tangents:

$$P(0) = a0^3 + b0^2 + c0 + d$$
$$\mathbf{P(0) = d}$$
$$P(1) = a1^3 + b1^2 + c1 + d$$
$$\mathbf{P(1) = a + b + c + d}$$
$$P^u(0) = 3a0^2 + 2b0 + c$$
$$\mathbf{P^u(0) = c}$$
$$P^u(1) = 3a1^2 + 2b1 + c$$
$$\mathbf{P^u(1) = 3a + 2b + c}$$

Using the values of the control points and tangents, we can calculate the constants:

$$
\begin{aligned}
a &= 2P(0) - 2P(1) + P^u(0) + P^u(1) \\
b &= -3P(0) + 3P(1) - 2P^u(0) - P^u(1) \\
c &= P^u(0) \\
d &= P(0)
\end{aligned}
$$

Now we can put this all together and plug it in (this is messy sorry):

$$
\begin{aligned}
P(u) &= au^3 + bu^2 + cu + d \\
P(u) &= (2P(0) - 2P(1) + P^u(0) + P^u(1))u^3 \\
&+ (-3P(0) + 3P(1) - 2P^u(0) - P^u(1))u^2 \\
&+ P^u(0)u \\
&+ P(0) \\
P(u) &= 2P(0)u^3 - 2P(1)u^3 + P^u(0)u^3 + P^u(1)u^3 \\
&- 3P(0)u^2 + 3P(1)u^2 - 2P^u(0)u^2 - P^u(1)u^2 \\
&+ P^u(0)u \\
&+ P(0) \\
P(u) &= 2P(0)u^3 - 3P(0)u^2 + P(0) \\
&- 2P(1)u^3 + 3P(1)u^2 \\
&+ P^u(0)u^3 - 2P^u(0)u^2 + P^u(0)u \\
&+ P^u(1)u^3 - Pu(1)u^2 \\
P(u) &= (2u^3 - 3u^2 + 1)P(0) \\
&+ (-2u^3 + 3u^2)P(1) \\
&+ (u^3 - 2u^2 + u)P^u(0) \\
&+ (u^3 - u^2)P^U(1)
\end{aligned}
$$

This gives us the full equations:

$$
\begin{aligned}
P(u) &= (2u^3 - 3u^2 + 1)P(0) + (-2u^3 + 3u^2)P(1) + (u^3 - 2u^2 + u)P^u(0) + (u^3 - u^2)P^u(1) \\
P^u(u) = P'(u) &= (6u^2 - 6u)P(0) + (-6u^2 + 6u)P(1) + (3u^2 - 4u + 1)P^u(0) + (3u^2 - 2u)P^u(1)
\end{aligned}
$$

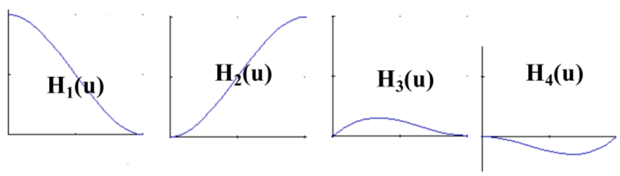
3.5.2 Blending/Basis Curves

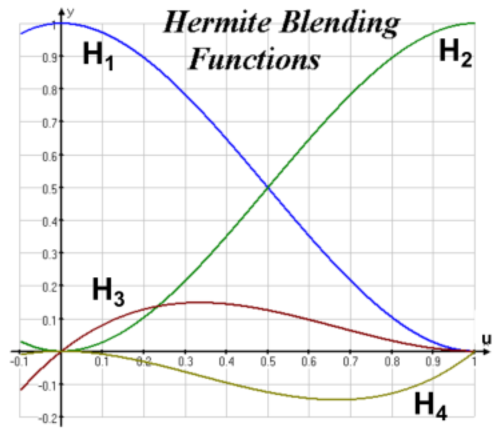
Given we transform the messy $P(u)$ function into:

$$
P(u) = H_1(u)P(0) + H_2(u)P(1) + H_3(u)P^u(0) + H_4(u)P^u(1)
$$

The blending/basis functions are then:

$\mathbf{H_1(u)} = 2u^3 - 3u^2 + 1$
$\mathbf{H_2(u)} = -2u^3 + 3u^2$
$\mathbf{H_3(u)} = u^3 - 2u^2 + u$
$\mathbf{H_4(u)} = u^3 - u^2$





We now see the following behavior:

At $u = 0$:
$$0 = H_2, H_3, H_4, H_1', H_2', H_4'$$
$$1 = H_1, H_3'$$

At $u = 1$:
$$0 = H_1, H_3, H_4, H_1', H_2', H_3'$$
$$1 = H_2, H_4'$$

3.5.3 Matrix Form

We can define the Hermite blending functions as the following where M_H is the Hermite characteristic matrix

$$H = \begin{bmatrix} H_1(u) & H_2(u) & H_3(u) & H_4(u) \end{bmatrix} = \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} u^3 \\ u^2 \\ u \\ 1 \end{bmatrix} = M_H U = B_H$$

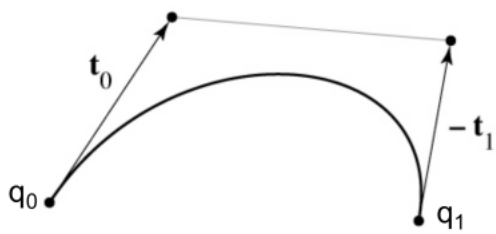
We then collect the control points and tangents into the following geometry matrix G:

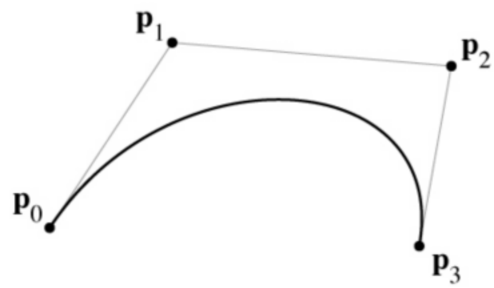
$$G = \begin{bmatrix} P(0) & P(1) & P^u(0) & P^u(1) \end{bmatrix}$$

Putting this together gives us the following matrix definition of $P(u)$:

$$P(u) = G M_H U = G B_H$$

3.5.4 Hermite and Bézier





Bézier to Hermite Transformation:

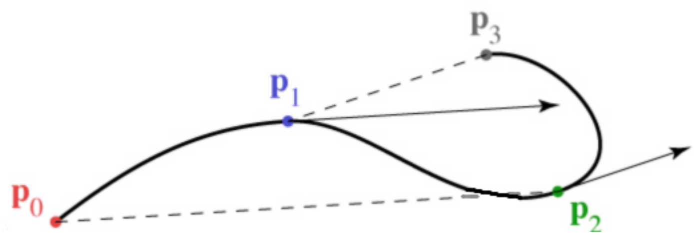
$$
\begin{aligned}
q_0 &= p_0 \\
q_1 &= p_3 \\
t_0 &= 3(p_1 - p_0) \\
t_1 &= 3(p_2 - p_3)
\end{aligned}
$$

Hermite to Bézier Transformation:

$$
\begin{aligned}
p_0 &= q_0 \\
p_1 &= q_0 + \frac{1}{3}t_0 \\
p_2 &= q_1 - \frac{1}{3}t_1 \\
p_3 &= q_1
\end{aligned}
$$

Note: Derivative is defined as 3 times offset

3.6 Catmull-Rom Splines



$\mathbf{n}$: Number of points

$\mathbf{p_0}$: Start point

$\mathbf{p_n}$: End point

$\mathbf{T_0}$: Tangent at 0 (Given)

$\mathbf{T_n}$: Tangent at n (Given)

$\mathbf{T_k}$: Tangent at any point p_k where $k \neq 0, n$ is $\frac{p_{k+1} - p_{k-1}}{2}$

3.6.1 Tension

To apply Tension $\mathcal{T}$ to a Catmull-Rom spline, adjust the tangents at interior joint points p_k with:

$$T_k = (1 - \mathcal{T})\frac{p_{k+1} - p_{k-1}}{2}$$

When $\mathcal{T} = 0$, we get a standard Catmull-Rom spline

When $\mathcal{T} = 1$, the tangent is 0

We can scale user-given tangent vectors with tension:

$$T_0' = (1 - \mathcal{T})T_0$$
$$T_n' = (1 - \mathcal{T})T_n$$

3.7 B-Splines

Rational : B-splines are defined as a ratio of cubic polynomials

Control Points : P_i

Blending Function : Bi Points defined by blending the control points

$$P(t) = \sum_{i=0}^{m} B_{i,d}(t) P_i$$

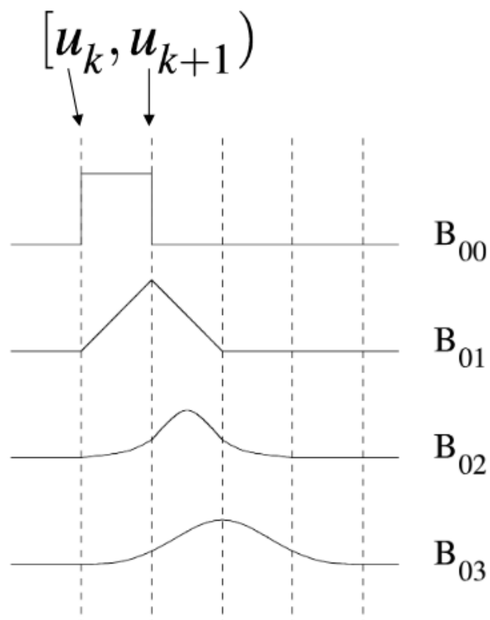
There are no limits on the value of t. but $B_i(t)$ is mostly 0.

Cox-deBoor Recursion : Defines the blending function $B_{i,d}$ where i is the point and d is the degree of the curve

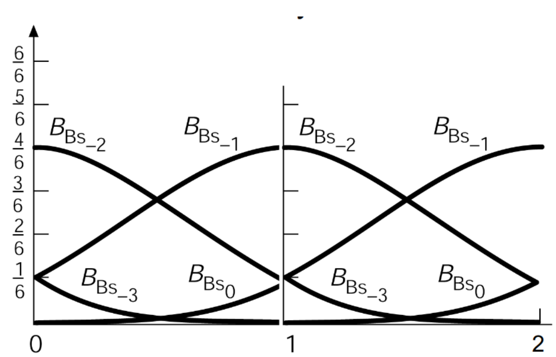
$$B_{i,0}(t) = \begin{cases} 1 & t_i \leq t < t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{i,d}(t) = \frac{t - t_i}{t_{i+d} - t_i} B_{i,d-1}(t) + \frac{t_{i+d+1} - t}{t_{i+d+1} - t_{i+1}} B_{i+1,d-1}(t)$$

$\mathbf{B_{i,0}(t)}$: Step function. Either 0 or 1
$\mathbf{B_{i,1}(t)}$: Piecewise function that spans two intervals. Goes from 0 to 1
$\mathbf{B_{i,2}(t)}$: Piecewise quadratic function that spans four intervals. Goes from 0 to $\frac{1}{4}$, $\frac{1}{4}$ to $\frac{3}{4}$ then back
$\mathbf{B_{i,3}(t)}$: Piecewise cubic function that spans four intervals. Goes from 0 to $\frac{1}{6}$, $\frac{1}{6}$ to $\frac{2}{3}$ then back

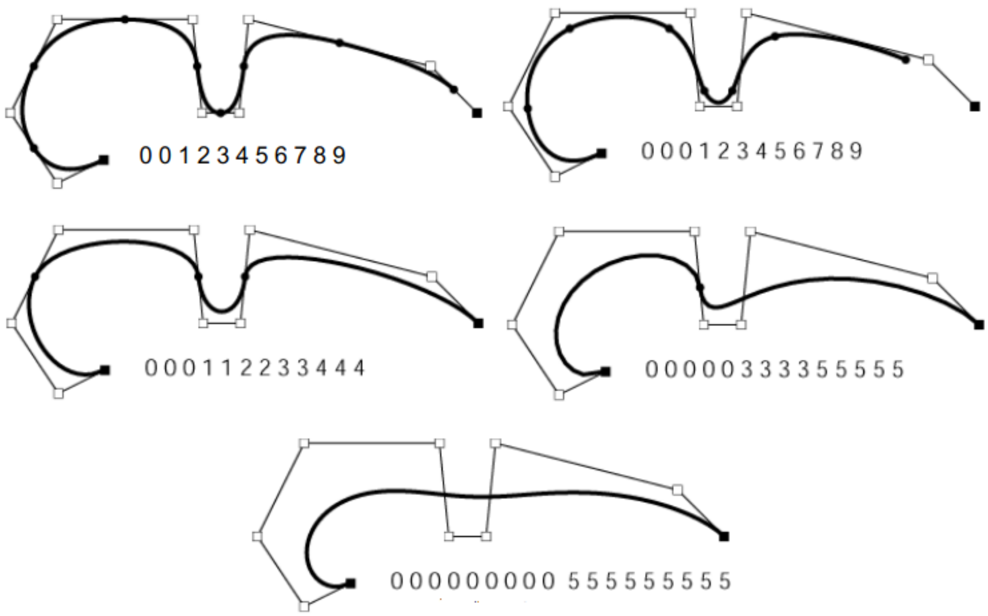


3.7.1 Knots



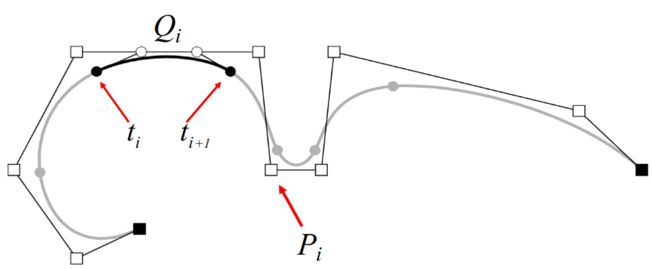
Notice how from $t = 0 tot = 1$, all the functions are not zero, this means the sum comes to 1. The convex hull property holds for all segments of a B-spline. Notice how segments connect to each other and transisition into each other.

Knots need to be chosen uniformly to get a uniform B-Spline. The closer knots are, the more weight given to them:



For example, the knots $\{0,0,0,0,1,1,1,1\}$ creates a Bezier curve.

3.7.2 Drawing a Line



When building lines, don't use $0 \leq t \leq 1$. Now use the knot points: $t_{min} \leq t_0 \leq t_1 \leq ... \leq t_{m-1} \leq t_m \leq t_{max}$

When drawing, we have the following specifications:

m : Must be greater or equal to 3

$\mathbf{P_i}$: Control points $P_0 \dots P_m$. Represented by squares in the above image

$\mathbf{t_i}$: Knot points $t_3 \dots t_{m+1}$. Represented by circles in the above image

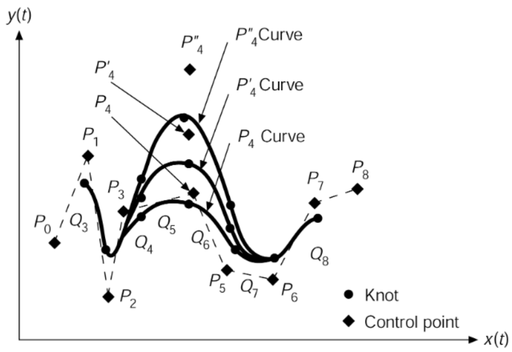
$\mathbf{Q_i}$: The cubic polynomial line segments being drawn $Q_3 \dots Q_m$

- Defined over knot interval $[t_i, t_{i+1}]$

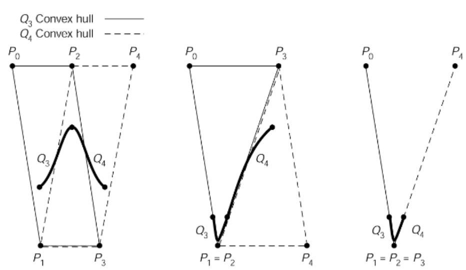
- Defined by control points $P_{i-3} \dots P_i$

3.7.3 Properties

Local Control : Since the polynomial coefficients only depend on a few points, adjusting knots only affects local curve. See below how moving only P_4 affects the local curve.



Convex Hull : B-Splines follow the convex hull property



Continuity : Since derivatives are really easy for cubics, it's easy to show C^0, C^1, and C^2

$$p(u) = \sum_{k=0}^{3} u^k c_k = c_0 + uc_1 + u^2 c_2 + u^3 c_3$$

$$p'(u) = c_1 + 2c_2 u + 3c_3 u^2$$

Benefits

Rational : Ratio of Polynomials

Since they're rational, they're invariant under:

- rotation
- scale
- translation

- perspective transformations

These transformations only redefine the control points then the curve is regenerated. Non-rationals are variant under perspective transformations.

Rational splines can also precisely define conic sections and other analytic functions. You can only approximate conics with non-rationals.

3.8 NURBS

NURBS : Non-**U**niform **R**ational **B**-splines

Different notations can be used:

Blending Function : $B_{i,d}(u)$ or $N_{i,d}(u)$

Parameter Variable : u or t

Curve : C or P or Q

Control Points : P_i or B_i

Variables for order, degree, number of control points, etc are consistently inconsistent.

When defined using homogeous coordinates, the 4th dimension of each P_i is the weight. In a 2D space, this is the 3rd dimension of P_i.

If the curve is defined as a weighted euclidian, a separate constant w_i is the weight of each control point.

Basic Idea : Four dimensional non-uniform B-splines which are normalized via homogeneous coordinates

Given functions $X(t), Y(t), Z(t)$ and $W(t)$ that are all cubic polynomials with controls points specified in homogeneous coordinates $[x, y, z, w]$ then:

$$\begin{aligned} P_i &= [x, y, z, w] \\ x(t) &= \frac{X(t)}{W(t)} \\ y(t) &= \frac{Y(t)}{W(t)} \\ z(t) &= \frac{Z(t)}{W(t)} \end{aligned}$$

In a 2D case, $Z(t) = 0$

Example : On the left is a unit circle in 3D homogeneous coordinates, the right is the rational parameterization of it

$$X(t) = 1 - t^2 \qquad x(t) = \frac{1 - t^2}{1 + t^2}$$
$$Y(t) = 2t \qquad y(t) = \frac{2t}{1 + t^2}$$
$$Z(t) = 0 \qquad z(t) = 0$$
$$W(t) = 1 + t^2$$

We can define a d-th degree NURBS curve C as:

$$C(u) = \frac{\sum_{i=0}^{n-1} w_i B_{i,d}(u) P_i}{\sum_{i=0}^{n-1} w_i B_{i,d}(u)}$$

This can also be written as:

$$C(u) = \sum_{i=0}^{n-1} P_i \frac{w_i B_{i,d}(u)}{\sum_{j=0}^{n-1} w_j B_{j,d}(u)}$$

The weights now induce a new rational basis function, R which can be defined as:

$$R_i(u) = \frac{w_i B_{i,d}(u)}{\sum_{j=0}^{n-1} w_j B_{j,d}(u)}$$

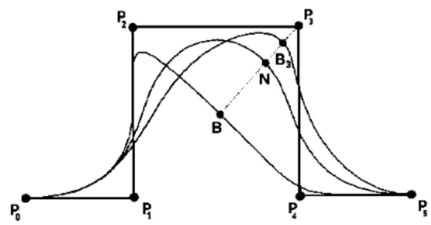
Give that $R_{i,d}(u)$ is a rational basis function on $u \in [0, 1]$ we can write the general form of the curve as:

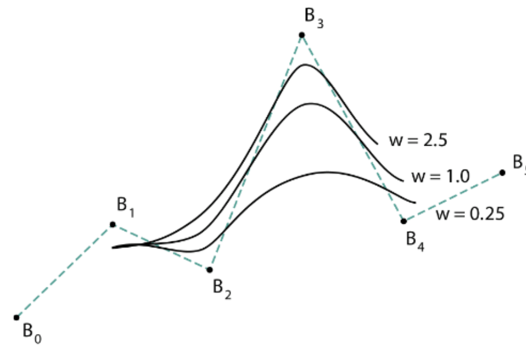
$$C(u) = \sum_{i=0}^{n-1} R_{i,d}(u) P_i$$

3.8.1 Weights

Since w_i of P_i only affects the range $[u_i, u_{i+k+1})$, the following behavior is observed:

- When $w_i = 0$, then P_i does not contribute to C
- When w_i increases, point B and curve C are pulled toward P_i and pushed away from P_j
- When w_i decreases, point B and curve C are pulled toward P_j and pushed away from P_i
- As w_i approaches infinity, B approaches 1


P0
P1
P2
P3
P4
P5
B
N
B3


B0
B1
B2
B3
B4
B5
w = 2.5
w = 1.0
w = 0.25

4 Drawing Parametic Curves

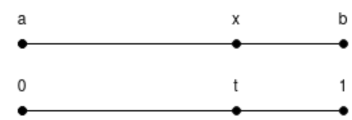
4.1 de Casteljau Algorithm

Developed by Paul de Casteljau at Citroën in the late 1950s

4.1.1 Linear Interpolations

Given a line from point a to point b, interpolating between them is an affine combination of points

$$x(t)a + (b - a)t$$



Given:

- C, a continuous curve
- P, an arbitrary plane
- PLI, a piecewise linear interpolant of C

Then the number of crossings of P by PLI is no greater than those of C

4.1.2 Drawing a Line

Base Case : Two points

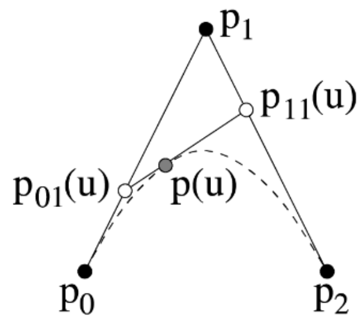
- Control points (p_0, p_1)
- Map parameter u to $\overline{p_0 p_1}$

$$p(u) = (1 - u)p_0 + up_1 \forall 0 \leq u \leq 1$$

A Step Up : Three points

- Control points (p_0, p_1, p_2)
- Interpolate $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$
- $0 \leq u \leq 1$

$$\begin{aligned} p_{01}(u) &= (1 - u)p_0 + up_1 \\ p_{11}(u) &= (1 - u)p_1 + up_2 \\ p(u) &= (1 - u)p_{01}(u) + up_{11}(u) \end{aligned}$$



Based off the fact $ratio(a, b, c) = \frac{b-a}{c-a}$, so:

$$ratio(p_0, p_{01}(u), p_1) = ratio(p_1, p_{11}(u), p_2) = ratio(p_{01}(u), p(u), p_{11}) = u$$

General Form:

Control Points: $p_0, p_1, ..., p_n \in R^3, t \in R$

$$p_{ir}(u) = (1-u)p_{i(r-1)}(u) + up_{(i+1)(r-1)}(t) \begin{cases} r = 1, ..., n \\ i = 0, ..., n-r \end{cases}$$
$$p_{i0}(u) = p_i$$
$$p_{0n}(u) = ?$$

4.1.3 Observations

- Interpolation along the curve is based only on u
- Drawing the curve's pixels requires iterating over u at sufficient refinement
- What is the right increment? Not constant
- Compute points and define a polyline

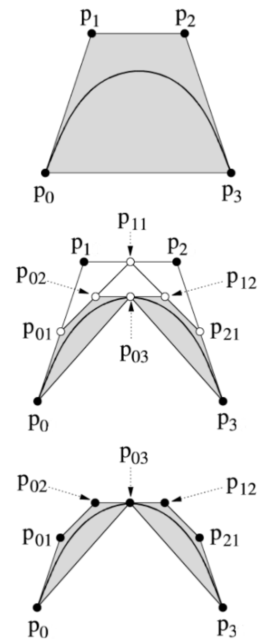
4.2 Subdivisions

Basics :

- Primitives defined by control polygons
- Set of control points is not unique
  - More than one way to compute a curve
- Subdivision refines representation of an object by adding control points

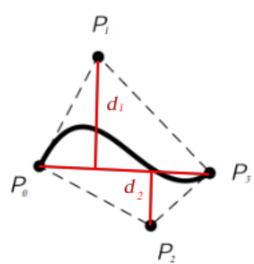
With Bezier Curves:

- Given control points $p_0, p_1.p_2, p_3$
- Calculate $p(0.5)$ to subdivide the curve into two curves
  - New control point $p_{03} = p(0.5)$
- Use edges to find new control points for each curve
  - $p_{01}(0.5) = 0.5p_0 + 0.5p_1$
  - $p_{02}(0.5) = 0.5p_{01} + 0.5p_{11}$
  - $p_{21}(0.5) = 0.5p_2 + 0.5p_3$
  - $p_{12}(0.5) = 0.5p_{11} + 0.5p_{21}$
- Two new curves
  - Control points: $p_0, p_{01}, p_{02}, p_{03}$
  - Control points: $p_{03}, p_{12}, p_{21}, p_3$
- Overall curve shape is not affected



4.2.1 Recursive Subdivisions

Curve Flatness Test : If d_1 and d_2 are both less than some ε, then the curve is flat



Distance from a Point to a Line :

1. Project point P, (p_x, p_y), onto line L, $[(l_{x0}, l_{y0}), (l_{x1}, l_{y1})]$
2. Find location of the projection

$$d(P, L) = \frac{(l_{y0} - l_{y1})p_x + (l_{x1} - l_{x_0})p_y + (l_{x0}l_{y0} - l_{x1}l_{y0})}{\sqrt{(l_{x1} - l_{x0})^2 + (l_{y1} - l_{y0})^2}}$$

The Algorithm : $DrawCurveRecSub(curve, e)$

1. if $straight(curve, e)$
   (a) $DrawLine(curve)$

2. else
    (a) $SubdivideCurve(curve, LeftCurve, RightCurve)$
    (b) $DrawCurveRecSub(LeftCurve, e)$
    (c) $DrawCurveRecSub(RightCurve, e)$

4.3 Bézier Curves

Iterate Over t with formulas:

- Increment t
- Calculate $x(t), y(t), z(t)$
- Can't easily control segment lengths and error

Iterate Over t with de Casteljau:

- Increment t
- Apply de Casteljau algorithm
- Successive interpolation of control polygon edges

Recursive Subdivision:

- Recursively subdivide de Casteljau polygons until they are approximately flat
- Use Bresenhams to draw line

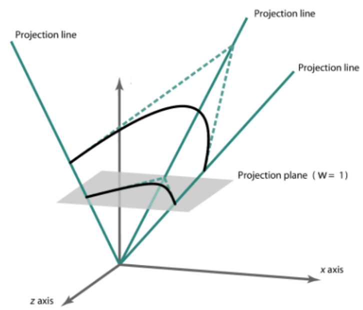
Degree Elevation:

- Given control points
- Generate additional control points
- Increase the degree of the curve
- Keep the curve the same
- In the limit, this converges to the curve defined by the original control points
- Generate control points until the points generated approximate the curve necessary

4.4 NURB Conic-Sections

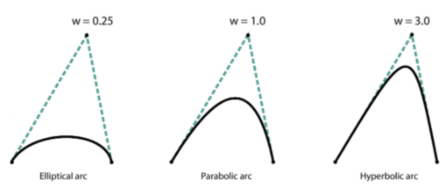
Obtained by projecting a parabola onto a plane. Assign w to each control point

- 3D Case: Rational curve is a 4D object
- 2D Case: Rational curve is a 3D object

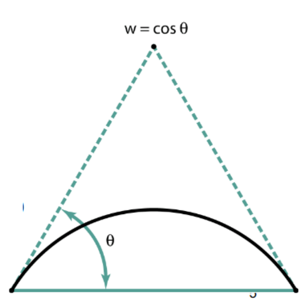


We can define the curve with three control points where the weights of the first and last control point are 1, Given the knot vector $\{0,0,0,1,1,1\}$, the weight of the center control point gives the following behavior:

- $w < 1$: Ellipse
- $w = 1$: Parabola
- $w > 1$: Hyperbola

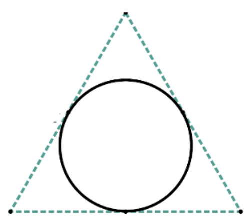


We can create a circular arc when the two lengths of the control point polygon are equal. The chord connecting the first and last control points must connect with the polygon at an angle θ which is equal to half of the angle of the arc. For example, in a 60° then $\theta = 30^\circ$. Additionally, the weight of the inner control point must be $\cos(\theta)$. The knot vector, just as above is $\{0,0,0,1,1,1\}$



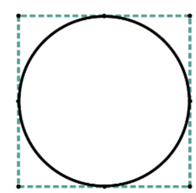
Circle : Three 120° arcs
knots $= \{0, 0, 0, 1, 1, 2, 2, 3, 3, 3\}$

$$\mathbf{B_0} = \{-0.866, 0.5, 1\}$$
$$\mathbf{B_1} = \{-1.732, -1, 0.5\}$$
$$\mathbf{B_2} = \{0, -1, 1\}$$
$$\mathbf{B_3} = \{1.732, -1, 0.5\}$$
$$\mathbf{B_4} = \{0.866, 0.5, 1\}$$
$$\mathbf{B_5} = \{0.2, 0.5\}$$
$$\mathbf{B_6} = B_0 = \{-0.866, 0.5, 1\}$$



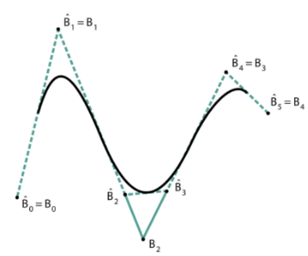
Square : Four 90° arcs
knots $= \{0, 0, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}, \frac{3}{4}, 1, 1, 1\}$

$$\mathbf{B_0} = \{1, 0, 1\}$$
$$\mathbf{B_1} = \{1, 1, \frac{\sqrt{2}}{2}\}$$
$$\mathbf{B_2} = \{0, 1, 1\}$$
$$\mathbf{B_3} = \{-1, 1, \frac{\sqrt{2}}{2}\}$$
$$\mathbf{B_4} = \{-1, 0, 1\}$$
$$\mathbf{B_5} = \{-1, -1, \frac{\sqrt{2}}{2}\}$$
$$\mathbf{B_6} = \{0, -1, 1\}$$
$$\mathbf{B_7} = \{1, -1, \frac{\sqrt{2}}{2}\}$$
$$\mathbf{B_8} = B_0 = \{1, 0, 1\}$$



4.5 Knot Insertion

Basic Idea : We want to add points but keep the same curve.



- Decide where we want to add control point
- Add knot
- Find the affected $d - 1$ control points
- Replace it with d new control points

Now looking at this more in-depth, we're given:

$P = (P_0, P_1, ..., P_n)$	Control Points
$U = (u_0, u_1, ..., u_m)$	Knots
d = degree of curve	

We want to insert a new knot u_k into the knot vector without changing the shape. If $u_k \in [u_i, u_{i+1})$, only the basis functions for $(P_i, ..., P_{i-d})$ are non-zero.

Find d new control points, everything else remains unchanged:

- Q_i on edge (P_{i-1}, P_i)
- Q_{i-1} on edge $(P_{i-2} P_{i-1})$
- ...
- Q_{i-d+1} on edge $(P_{i-d} P_{i-d+1})$

We actually define our new control point as:

$$Q_j = (1 - \alpha_j) P_{j-1} + \alpha_j P_j$$

We define α as:

$$\alpha_j = \frac{u_k - u_j}{u_{j+d} - u_j}$$

4.5.1 Properties

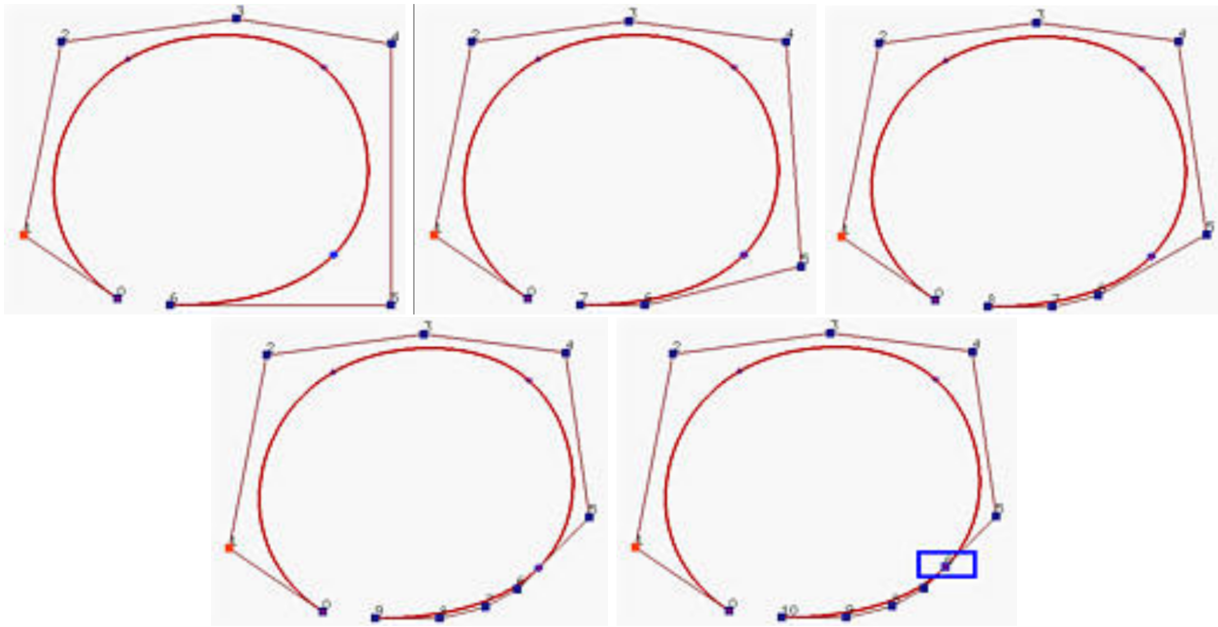
- Increasing multiplicity of a knot decreases number of non-zero basis functions at that knot
- At a knot of multiplicity d, there is only one non-zero basis function
- Corresponding curve $p(u)$ is affected by only one control point P_i

4.6 de Boor Algorithm

This is a generalization of de Casteljau's Algorithm.

Goal : Find a fast and numerically stable way for finding a point on a B-spline curve

Observation : If knot u is inserted d times to a B-spline, then $p(u)$ is the point on the curve



function DEBOORS(P_n, u_m, u)
    if $u \in [u_i, u_{i+1})$ and $u \neq u_i$ **then**
        $h := d$
    end if
    if $u = u_i$ and (u_i is a knot of multiplicity s) **then**
        $h := d - s$
    end if
    for Affected Control Point **do**
        $P_{i-s,0} := P_{i-s}$
    end for
    for $r := 1 \to h$ **do**
        for $j := i - d + r \to i - d$ **do**
            $a_{j,r} := \frac{u-u_j}{u_{j+d-r+1}-u_j}$
            $P_{j,r} := (1 - a_{j,r})P_{j-1,r-1} + a_{j,r}P_{j,r-1}$
        end for
    end for
    return $P_{i-s,d-s}$
end function

Compare To De Casteljau's Algorithm:

- De Casteljau's Algorithm
  - Dividing points are computed with a pair of numbers $(1 - u)$ and u that don't change
  - Can be used for subdivision
  - Uses all control points
- De Boor's Algorithm
  - These pairs of numbers are different and depend on the column number and control point number
  - Intermediate control points not sufficient
  - Only affected control points $(d - 1)$ are used in the computation

4.7 Oslo Algorithm

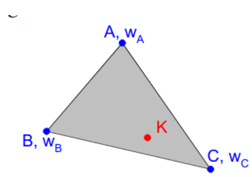
Goal : Subdivision for B-Splines

- Curve C with control points $(P_0, ..., P_m)$
- Insert knot at any point
- Two new points P_k' and P_k''
- Apply recursively on new parts:
  - $P_0, ..., P_k', P_k'', ..., P_{m-1}$
  - $P_1, ..., P_k', P_k'', ..., P_m$

4.8 Barycentric Coordinates

Given a triangle with vertices $\{A, B, C\}$ and weights $\{w_A, w_B, w_C)$, their center of gravity (barycenter) will coincide with any point K inside the triangle. This defines K as:

$$K = w_A A + w_B B + w_C C1 \qquad = w_A + w_B + w_C w_A = 0 \qquad \text{Points on } BC$$
$$w_B = 0 \qquad \text{Points on } AC$$
$$w_C = 0 \qquad \text{Points on } AB$$



To calculate the weights now, we can calculate the are of each of the subtriangles, , then use ratios:

$$w_A = \frac{SubArea(B, C, K)}{Area(A, B, C)}$$
$$w_B = \frac{SubArea(A, C, K)}{Area(A, B, C)}$$
$$w_C = \frac{SubArea(A, B, K)}{Area(A, B, C)}$$

$$SubArea(d, e, f) = \frac{|(d - f) * (e - f)|}{2}$$

Given vertices $\{A, B, C\}$ and a centroid K we can find the coordinates of the weights:

$$
\begin{aligned}
x_K &= w_A x_A + w_B x_B + w_C x_C \\
y_K &= w_A y_A + w_B y_B + w_C y_C \\
x_K &= w_A x_A + w_B x_B + (1 - w_A - w_B) x_C \\
y_K &= w_A y_A + w_B y_B + (1 - w_A - w_B) y_C \\
w_A &= \frac{(x_B - x_C)(y_C - y_K) - (x_C - x_K)(y_B - y_C)}{(x_A - x_C)(y_C - y_K) - (x_B - x_C)(y_A - y_C)} \\
w_B &= \frac{(x_A - x_C)(y_C - y_K) - (x_C - x_K)(y_A - y_C)}{(x_B - x_C)(y_C - y_K) - (x_A - x_C)(y_B - y_C)} \\
w_C &= 1 - w_A - w_B
\end{aligned}
$$

5 Surfaces

5.1 Exact vs Approximation

There are two ways to represent and model 3D objects:

1. Exactly
2. Approximately

Overview

Exact

- Wireframe
- Parametric Surface
- Solid Model (CSG, BRep, Implicit Solid Modeling)
- Precise model of object topology
- Mathematically represent all geometry

Approximate

- Facet / Mesh (Surfaces)
- Voxels (Volume Info)
- A discretization of the 3D object
- Use simple primitives to model topology and geometry

Pros

Exact

- High precision
- Lots of modeling environments
- Physical properties
- High level control
- Many applications (tool path generation, motion, etc)
- Compact

Approximate

- Easy to implement
- Easy to acquire
- Easy to render
- Many algorithms

Cons

Exact	Approximate
• Complex data structures	• Lossy
• Extensive algorithms	• Data structure sizes can get huge
• Many specific nuanced formats	• Easy to break
• Hard to acquire data	• Not good for many applications
• Requires translation for rendering	• Lots of guesswork

5.2 Wireframes

The general idea is to represent the model as the set of its edges. We can describe a cube as:

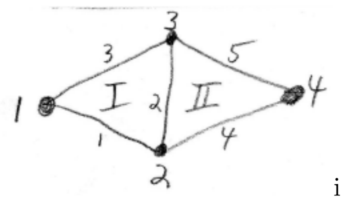
Vertices	Lines
$A : (0,0,0)$	AB
$B : (1,0,0)$	BC
$C : (1,1,0)$	CD
$D : (0,1,0)$	DA
$E : (0,0,1)$	EF
$F : (1,0,1)$	FG
$G : (1,1,1)$	GH
$H : (0,1,1)$	HE
	AE
	BF
	CG
	DH

The problem is that wire frames are visually ambiguous since there are no surfaces, the inside is hard to tell from the outside. It's also hard to model a shape wire by wire.

5.3 Surface Models

The general idea now is to represent a model as a set of faces and patches. The issue now is getting faces to "line up" nicely and accurately. Which side is "inside" and which is "outside"?

BRep Data Structure : Winged Edge Data Structure that has:



i

- Vertex
  - (x, y, z) point
  - n pointers to coincident edges
- Edge
  - 2 pointers to end-point vertices
  - 2 pointers to adjacent faces
  - Pointer to the next edge
  - Pointer to the previous edge
- Face
  - m pointers to edges

Vertices

V1	(x, y, z)	$E3$	$E1$	
V2	(x, y, z)	$E1$	$E2$	$E4$
V3	(x, y, z)	$E2$	$E3$	$E5$
V4	(x, y, z)	$E4$	$E5$	

Edges

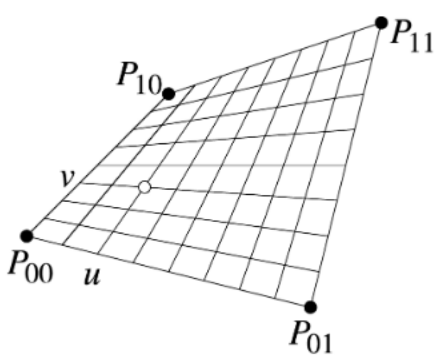
E1	$V1$	$V2$	$F1$		$E2$	$E3$
E2	$V2$	$V3$	$F1$	$F2$	$E3$	$E1$
E3	$V3$	$V1$	$F1$		$E1$	$E2$
E4	$V2$	$V4$	$F2$		$E5$	$E2$
E5	$V3$	$V3$	$F2$		$E2$	$E4$

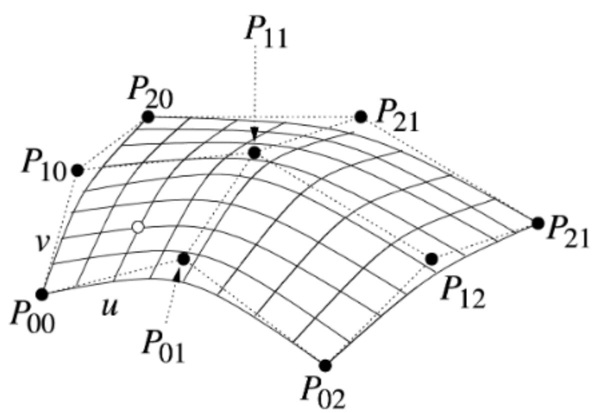
Faces

F1	$E1$	$E2$	$E3$
F2	$E2$	$E4$	$E5$

5.4 Biparametric Surfaces

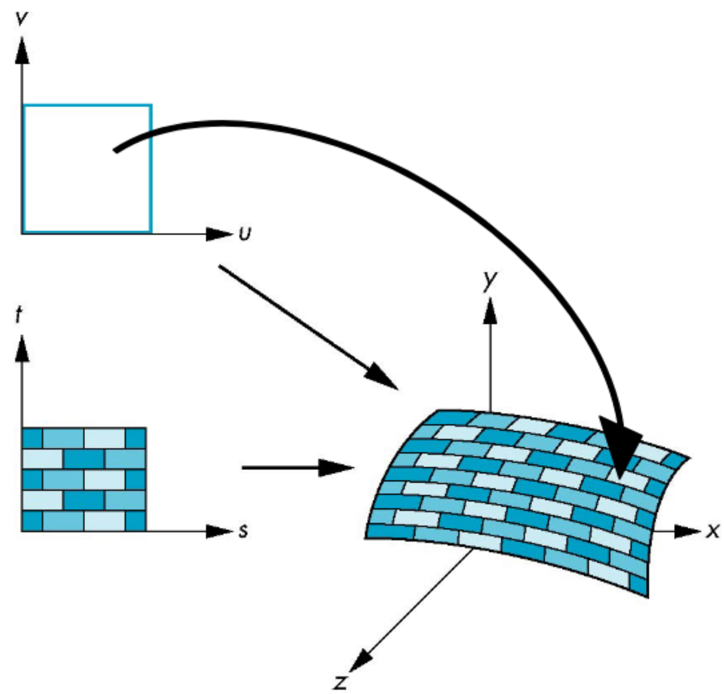
Biparametric Surfaces : Generalization of parametric curves with two parameters s and t





Biparametric Patch : (u, v) maps to a 3D point on the patch

$$F(u, v) = (x, y, z) = (x(u, v), y(u, v), z(u, v))$$

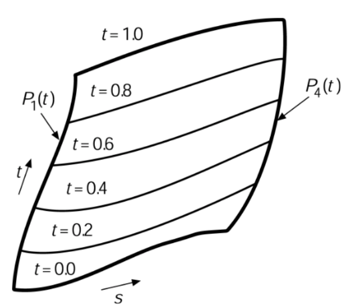


5.5 Bicubic Surfaces

In 3D, we allow the points in the geometry matrix G to vary in 3D along t:

$$Q(s,t) = \begin{bmatrix} G_1(t) & G_2(t) & G_3(t) & G_4(t) \end{bmatrix} MS$$

For a fixed t_1, then $Q(s, t_1)$ is a curve. Gradually incrementing t_1 to t_2 gives us a new curve. This combination of curves creates a surface. Above, $G_i(t)$ are 3D curves.



Each $G_i(t)$ is defined as $G_i(t) = G_i MT$ where:

$$G_i = \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix}$$

We then transpose $G_i(t)$:

$$G_i(t) = T^T M^T G_i^T$$

Then substitute $G_i(t)$ into $Q(s)$ gives:

$$Q(s,t) = T^T M^T \begin{bmatrix} g_{11} & g_{21} & g_{31} & g_{41} \\ g_{12} & g_{22} & g_{32} & g_{42} \\ g_{13} & g_{23} & g_{33} & g_{43} \\ g_{14} & g_{24} & g_{34} & g_{44} \end{bmatrix} MS$$

$Q(s,t)$ can be written over the $0 \leq s,t \leq 1$ interval as:

$$\begin{aligned} x(s,t) &= T^T M^T G_x MS \\ y(s,t) &= T^T M^T G_y MS \\ z(s,t) &= T^T M^T G_z MS \end{aligned}$$

5.6 Bicubic Bézier Patches

Also called a Bézier Surface. Defined as:

$$\begin{aligned} x(s,t) &= T^T M_B^T G_{B_x} M_B S \\ y(s,t) &= T^T M_B^T G_{B_y} M_B S \\ z(s,t) &= T^T M_B^T G_i B_z M_B S \end{aligned}$$

Given a data array $P = [p_{ij}]$:

$$\vec{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u) b_j(v) \vec{p}_{ij} = u^T M_B P M_B^T v$$

The cubic Bézier blending function is defined as:

$$b(u) = \begin{cases} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{cases}$$

Features of the Bicubic Bézier Patch include:

- Interpolates 4 corner control points

- 4 edges are Bézier curves
- Has convex hull property

Faceting : Defining triangle surfacs that tessellate the patch

The process is:

- Nested loops for u and v
- For each (u, v) from $(0, 0)$ through $(nu - 1, nv - 1)$
  - Calculate 3D point on patch
  - Keep track of linear index
  - Define triangles
    $triangle[k] = (vert[i, j], vert[i + 1, j], vert[i + 1, j + 1])$
    $triangle[k] = (vert[i, j], vert[i + 1, j + 1], vert[i, j + 1])$

5.6.1 Surface Normals

Normals are used for:

- Shading
- Interference detection in robotics
- Calculating offsets in numeric controlled machining

The general process is as follow:

1. Compute s tangent vector
2. Compute t tangent vector
3. Compute the cross product of s and t

Since s and t are tangent to the surface, their cross product is the normal vector.

First, the s tangent vector:

$$\begin{aligned}
Q(s,t) &= T^T * M^T * G * M * S \\
\frac{\delta}{\delta s}Q(s,t) &= \frac{\delta}{\delta s}(T^T * M^T * G * M * S) \\
&= T^T * M^T * G * M * \frac{\delta}{\delta s}(S) \\
&= T^T * M^T * G * M * \begin{bmatrix} 3s^2 & 2s & 1 & 0 \end{bmatrix}
\end{aligned}$$

Then the t tangent vector:

$$
\begin{aligned}
Q(s,t) &= T^T * M^T * G * M * S \\
\frac{\delta}{\delta t} Q(s,t) &= \frac{\delta}{\delta t}(T^T * M^T * G * M * S) \\
&= \frac{\delta}{\delta t}(T^T) * M^T * G * M * S \\
&= \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix}^T * M^T * G * M * S
\end{aligned}
$$

Then their cross product:

$$
Q_N(s,t) = \frac{\delta}{\delta s} Q(s,t) \times \frac{\delta}{\delta t} Q(s,t) \tag{1}
$$

$$
Q_N(s,t) = \begin{bmatrix} y_s z_t - y_t z_s & z_s x_t - z_t x_s & x_s y_t - x_t y_s \end{bmatrix} \tag{2}
$$

$$
\tag{3}
$$

5.7 B-Spline Surfaces

We get the following repesentation for B-Spline patches:

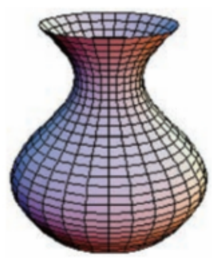
$$
\begin{aligned}
x(s,t) &= T^T * M_{Bs}^T * G_{Bs_x} * M_{Bs} * S \\
y(s,t) &= T^T * M_{Bs}^T * G_{Bs_y} * M_{Bs} * S \\
z(s,t) &= T^T * M_{Bs}^T * G_{Bs_z} * M_{Bs} * S
\end{aligned}
$$

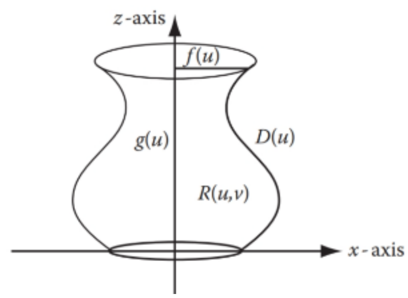
5.8 Surfaces of Revolution

Directrix : A planar curve being revolved

Axis of Revolution : The axis the directrix is revolved around, typically the z axis

Surfaces of revolution produce circular cross sections





Directrix : $D(u) = (f(u), 0, g(u))$

Surface : $S(u) = (f(u)\cos\theta, f(u)\sin\theta, g(u)), 0 \leq u \leq 1, 0 \leq \theta \leq 2\pi$

Tangents : Similar process as before:

1. $\frac{\delta}{\delta u}S(u,\theta) = (f'(u)\cos\theta, f'(u)\sin\theta, g'(u))$
2. $\frac{\delta}{\delta\theta}S(u,\theta) = (-f(u)\sin\theta, f(u)\cos\theta, 0)$
3. $N(u,\theta) = \frac{\delta}{\delta u}S(u,\theta) \times \frac{\delta}{\delta\theta}S(u,\theta)$

5.9 Drawing Parametic Surfaces

Typically done by going "patch by patch", but there are two main options:

- Render directly from the parametric description
- Approximate with a polygon mesh, then rend the mesh

Direct Rendering : Scan line by line, then pixel by pixel. Issues include:

- How to go from (x,y) "screen space" to point on the 3D patch
- Max and Min y coordinates may not lie on boundaries
- Silhouette edges result from patch bulges
- Need to track silhouettes and boundaries

5.9.1 Object Space Conversion

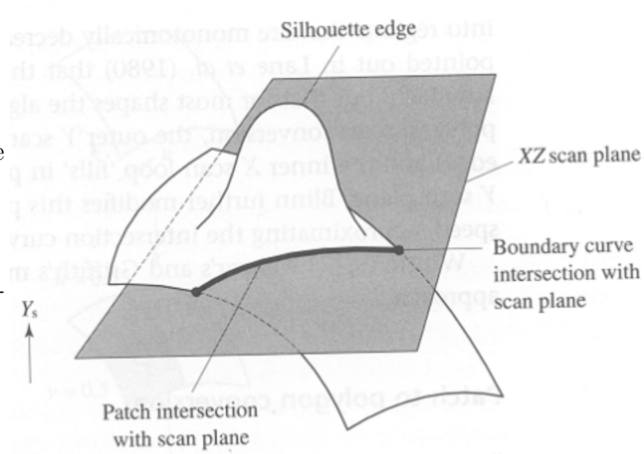
A type of patch to polygon rendering. The resolution depends on the object space. Three techniques:

- Iterative evaluation
- Uniform subdivision
- Non-uniform subdivision

Plane By Plane Rendering : Scan plane by plane iteratively

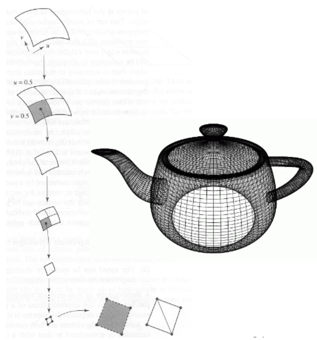
Given a patch $x = X(u, v), y = Y(u, v), z = Z(u, v)$, this looks like:

- Find the intersection of the patch with XZ plane producing a planar curve
- Draw the curve using a known algorithm
- *Note: When rendering, pixel-by-pizel color values can be computed this way*



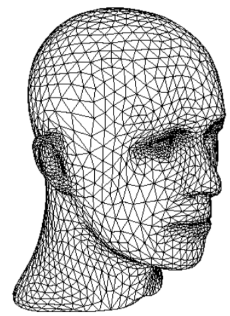
Uniform Subdivision

1. Cut parameter space into equal parts
2. Find new points on surface
3. Recurse and repeat until desired resolution
4. Split squares into triangles



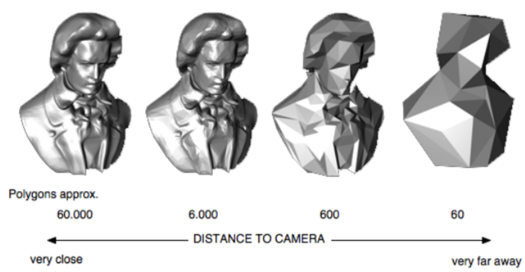
Non-Uniform Subdivision

1. More facets in areas of high curvature
2. Use change in normals to assess curvature
3. Break patch into subpatches based on curvature changes
4. Split squares into triangles



In Practice:

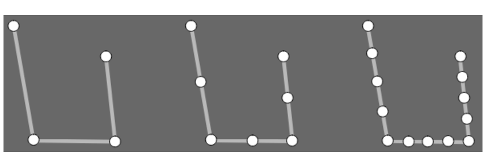
You need fewer triangles for models further away from the camera



In 1D : It's just piecewise linear subdivision

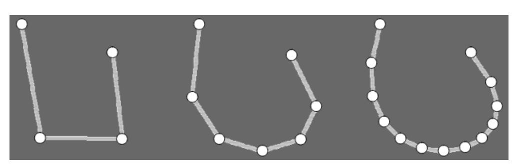
$$x_n = \frac{1}{2}(x_l + x_r)$$

$$y_n = \frac{1}{2}(y_l + y_r)$$



1D Four Points:

$$p_{2i+1,j+1} = \frac{1}{16}(-p_{i-1,j} + 9p_{i,j} + 9p_{i+1,j} - p_{i+2,j})$$



5.9.2 Image Space Conversion

A type of patch-polygon rendering. The resolution depends on the pixels and screen. Control the subdivision based on the screen criteria:

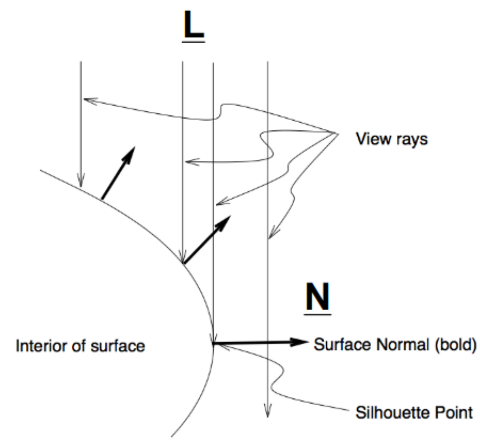
- **Minimum pixel area :** Stop when the patch is 1 pixel

- **Screen flatness :** Stop when the patch converges to a polygon
- **Screen flatness of Silhoette edges :** Stop when edge is straight or 1 pixel

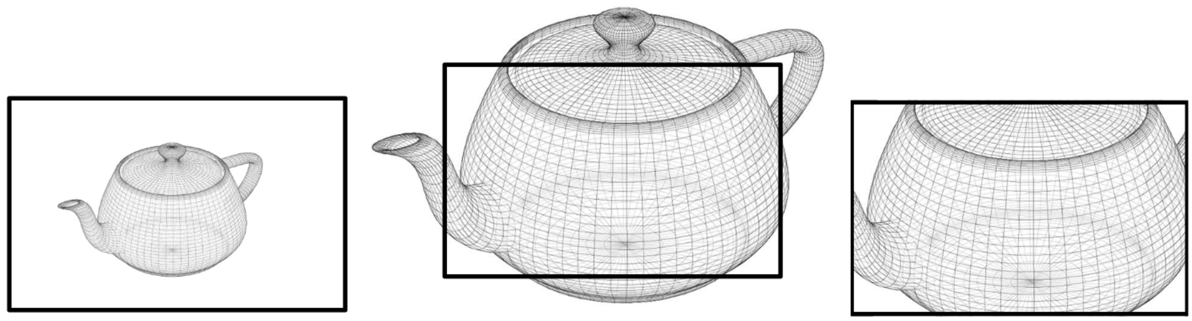
5.9.3 Silhouette Rays

An edge is a silhouette edge when the viewing ray is tangent to the point it hits on the surface. Where N is the normal, and L is the line of sight:

$$N(S) \cdot L = 0$$



6 Clipping



Goal : Only draw lines inside of the window and clip the lines to window boundary

6.1 Scissor Clipping

While scanning and converting the line:

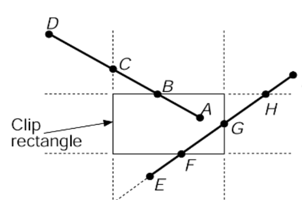
---

if $x_{min} < x < x_{max}$ and $y_{min} < y < y_{max}$ **then**
  Draw (x, y)
else
  Do Nothing
end if

The problem is this is too slow and we do more work than necessary. It's better to clip lines to window instead of calculating lines outside of the window.

6.2 Cohen-Sutherland Line Clipping

It's easy to know which lines are entirely inside and entirely outside a window. It's harder to figure out the partials.



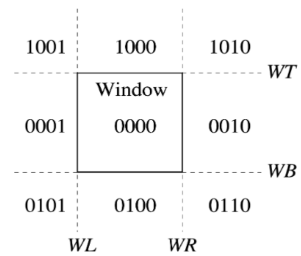
Given a straight line from $P_0 = (x_0, y_0)$ to $P_1 = (x_1, y_1)$ and a window defined by lines WT, WR, WB, WL we:

- Is the line completely in the window? Draw it
- Is the line completely outside the window? Ignore it

- Does the line intersect the window? Do more work

There's a basic four bit code we get based on the endpoints P_0 and P_1

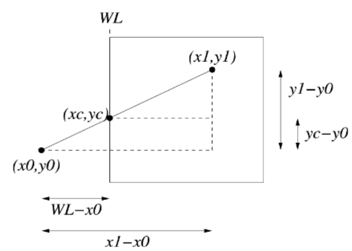
B_3 : Point above window $y > WT$
B_2 : Point below window $y < WB$
B_1 : Point left of window $x > WR$
B_0 : Point right of window $x < WL$



So now, computing $\neg(P_0 \vee P_1)$ will tell us of the line is completely visible. The line is completely outside the window when $P_0 \wedge P_1$.

Using the bits, we find the line $((x_0, y_0), (x_1, y_1))$ that intersects with the window border, then use similar triangles to find the point of intersection:

$$x_c = WL$$
$$\frac{y_c - y_0}{y_1 - y_0} = \frac{WL - x_0}{x_1 - x_0}$$
$$y_c = \frac{WL - x_0}{x_1 - x_0}(y_1 - y_0) + y_0$$



Then, replace (x_0, y_0) with (x_c, y_c), recompute the codes and continue until all lines are inside the window.

6.3 Cyrus Beck Technique

Recall a parametric line P with point on it's line t where $0 \leq t \leq 1$ and $P(0) = P_0$ and $P(1) = P_1$:

$$P(t) = P_0 + t(P_1 - P_0) = (1 - t)P_0 + tP_1$$

We can intersect two edges (P_0, P_1) and (P_2, P_3) with the following:

$$E_a = P_0 + t_a(P_1 - P_0)$$
$$E_b = P_2 + t_b(P_3 - P_2)$$
$$D_a \equiv P_1 - P_0$$
$$D_b \equiv P_3 - P_2$$

They intersect when:

$$P_0 + t_a D_a = P_2 + t_b D_b$$

Giving us:

$$
\begin{aligned}
x_0 + dx_0 t_a &= x_2 + dx_2 t_b \\
y_0 + dy_0 t_a &= y_2 + dy_2 t_b
\end{aligned}
$$

We can now solve for t_a and t_b :

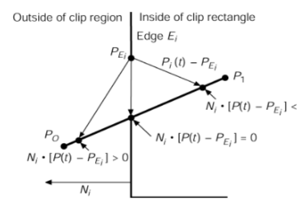
$$
\begin{aligned}
t_a &= \frac{dy_2(x_0 - x_2) + dx_2(y_2 - y_0)}{dy_0 dx_2 - dx_0 dy_2} \\
t_b &= \frac{dy_0(x_2 - x_0) + dx_0(y_0 - y_2)}{dy_2 dx_0 - dx_2 dy_0}
\end{aligned}
$$

If the denominator is 0, the lines are parallel. If $0 \le t_a, t_b \le 1$, the edges intersect.

Goal : Clip lines against convex polygons

Line : $P(t) = P_0 + t(P_1 - P_0)$
Point on Edge : P_{E_i}
Normal to Edge i **:** N_i
Displacement : $D = (P_1 - P_0)$
Note: Make sure D isn't 0 and the lines aren't parallel

$$
\begin{aligned}
0 &= N_i[P(t) - P_{E_i}] \\
0 &= N_i[P_0 + t(P_1 - P_0) - P_{E_i}] \\
0 &= N_i[P_0 - P_{E_i}] + N_i t(P_1 - P_0)
\end{aligned}
$$



The calculate t:

$$
t = \frac{-N_i[P_0 - P_{E_i}]}{N_i D}
$$

For the window edges, N_i is easy:

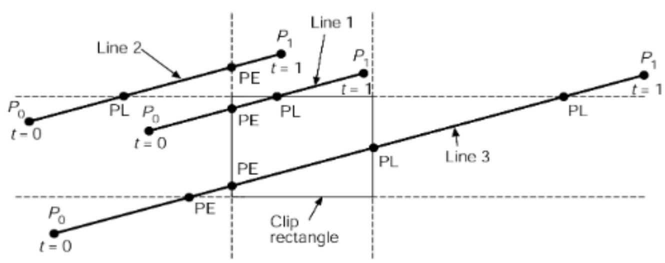
- WT : (0, 1)
- WB : (0, -1)
- WL : (-1, 0)
- WR : (1, 0)

For arbitrary edges:

$$
\begin{aligned}
E &= \frac{V_1 - V_0}{|V_1 - V_0|} && \text{Calculate edge direction} \\
N_x &= E_y && \text{Rotate } -90^\circ \\
N_y &= -E_x && \text{Rotate } -90^\circ
\end{aligned}
$$

Then to calculate the line segment:

1. Find intersection points between line and every window edge
2. Classify points as entering (PE) or leaving (PL)
   - PE if angle P_0P_1 and N_i is greater than 90°
   - PL otherwise
3. $T_e = maxt_e$
4. $T_l = maxt_l$
5. Discard if $T_e > T_l$
6. If $T_e < 0$ then $T_e = 0$
7. If $T_l > 1$ then $T_l = 1$
8. Use T_e, T_l to compute intersection coordinates (x_e, y_e) and (x_l, y_l)



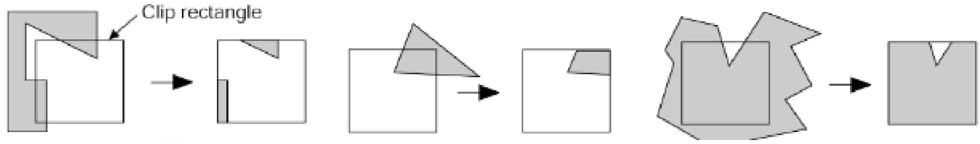
6.4 Polygon Clipping

Polygon : Ordered set of vertices, usually counter clockwise. Two consecutive vertices define an edge. Left side of an edge is inside while the right is the outside. Last vertex is implicitly connected to the first.

The edges of polygon need to be tested against the clipping rectangle. This creates a few cases we should be prepared to handle:

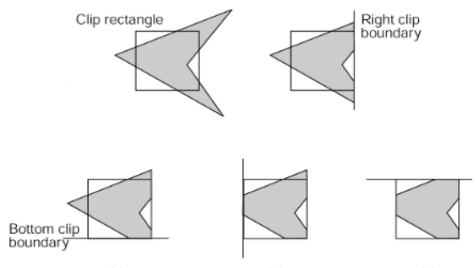
- May need to discard edges
- May need to add new edges
- May need to divide edges

- One polygon may become many



6.5 Sutherland-Hodgman Algorithm

Basic Idea : Clip single polygon using a single infinite clip edge 4 times

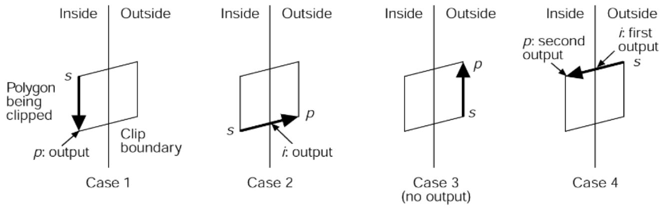


Given the vertices of the polygon $(v_1, v_2, ..., v_n)$ and a single infinite clip edge with inside and outside information, we can:

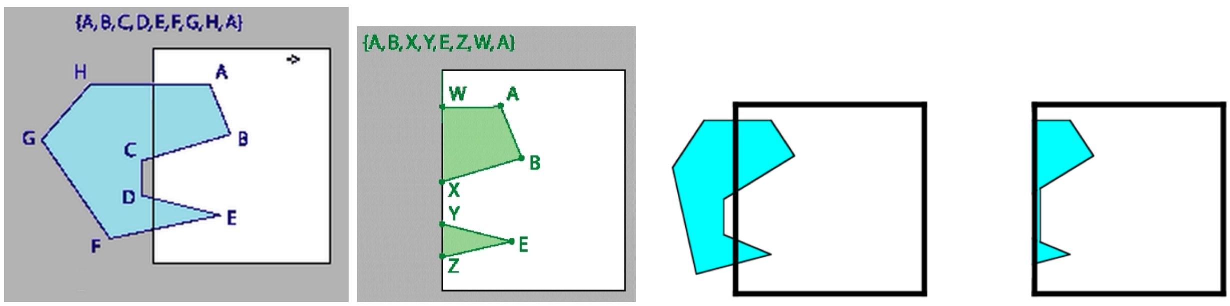
---

function SutherlandHodgeman(P, W)
    for $\forall$ polygons $P_i \in P$ **do**
        for $\forall$ clipping edges $E_c \in W$ **do**
            for $\forall$ edges $E_p in P_i$ **do**
                // Check clipping cases
                if Case 1 Applies **then**
                    Output v_{i+1}
                else if Case 2 Applies **then**
                    Output intersection point
                else if Case 3 Applies **then**
                    No output
                else if Case 4 Applies **then**
                    Output intersection point and v_{i+1}
                end if
            end for
        end for
    end for
end function

Then a visualization of the four clipping cases:



This produces the following input and output. Note the edges (X, Y) and (Z, W) exist. We have an issue where instead of two convex polygons created, we have one concave polygon.



6.6 Weiler-Atherton Algorithm

Given polygons A and B as linked lists, Weiler-Atherton starts with the following. This starting point is used for any of the sub functions of the algorithm.

1. Find all the edge intersections and
2. Place into another list. Insert as "intersection" nodes
3. Determine inside and outside nodes

The intersection special cases are:

- The edges are parallel? Ignore
- The intersection is a vertex and the vertex should be replaced with an intersection node
  - V_a is on E_b
  - E_a runs through E_a

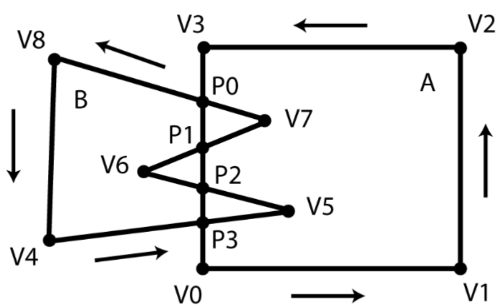
From here, there are two cases, union and intersection.

6.6.1 Intersection

1. Start at intersection node
2. If it's connected to an "inside" vertex, go there
3. Else go to an intersection point

4. If neither stop
5. Traverse linked list
6. At each intersection point switch to other polygon and remove intersection point from list
7. Repeat until returning to starting intersection point
8. If the intersection list isn't empty, pick another point
9. All visited vertices and nodes define the overlapping polygon

The below image is traversed in the order: $\{V_1.V_7.P_0\}$ and $\{P_3, V_5.P_2\}$



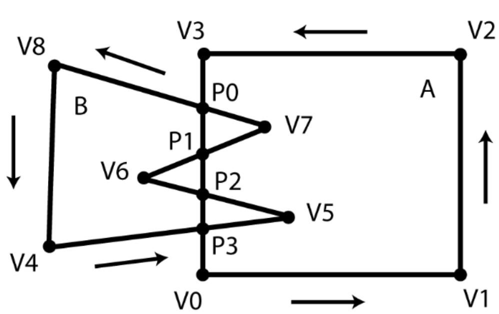
Special Cases : When the polygons don't intersect

- If one is inside the other, return the inner polygon
- Otherwise, return none

6.6.2 Union

1. Find an "outside" vertex
2. Traverse linked list
3. At each intersection, switch to the other polygon
4. Repeat until back at start
5. If there are still unvisited "outside" edges, repeat

The below image is traversed in the order: $\{V_0.V_1.V_2, V_3, P_0, V_8, V_4, P_3, V_0\}$ and $\{V_6, P_1.P_2\}$



Special Cases : When the polygons don't intersect

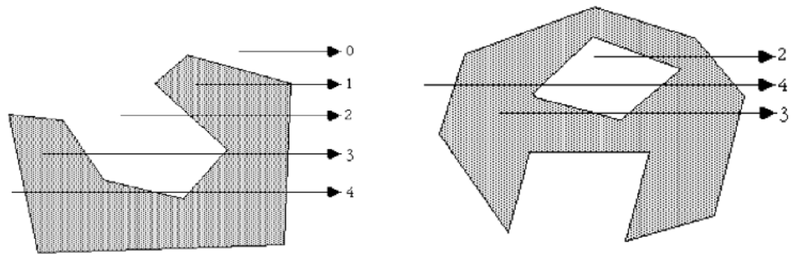
- If one is inside the other, return the outer polygon
- Otherwise, return both

```
function UnionTwoSimpleConvexPolygons(A, B)
    P0 := A
    P1 := B
    v_i := a vertex from A outside of B
    Output v_i
    E_c := (v_i, v_{i+1})
    while (||Output|| < 2) or (Output.first ≠ Output.last) do
        Intersect E_c with all edges in P1
        // There can be at most 2 intersections
        if Intersections = 0 then
            Output v_{i+1}
            E_c := E_c.next
        else
            Output intersection with lowest t value along E_c
            Output last vertex of P1's intersected edge
            E_c := P1.next
            T := P1
            T := P1
            P1 := P0
            P0 := T
        end if
    end while
end function
```

To find if a point is inside a polygon?

Jordan Curve Theorem : A point p is inside polygon P if for any ray shot from p to the outside of P, there are an odd number of edges crossed.

```
function InteriorPoint?(p, P)
    p' := known point outside P
    pp' := the edge between p and p'
    Intersect pp' with all polygon edges and count intersections
    if count is even then
        False
    else
        True
    end if
end function
```


0
1
2
3
4
2
4
3

7 Filling

Two main parts:

1. Which pixels do I fill?
2. What values do I fill them with?

Coherence : Make everything go together, but in what way?

- **Spatial :** Pixels are the same from pixel to pixel and scan-line to scan-line
- **Span :** All pixels on a span get the same value
- **Scan-Line :** Consecutive scan lines are the same
- **Edge :** Pixels are the same along edges

7.1 Rectangles

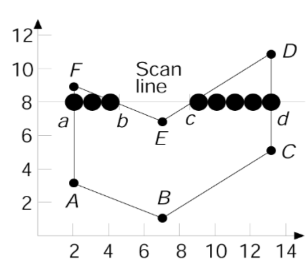
It's an easy algorithm:

function FillRectangle(R)
    for $x_i := x_{min} \to x_{max}$ **do**
        for $y_i := y_{min} \to y_{max}$ **do**
            Color (x_i, y_i)
        end for
    end for
end function

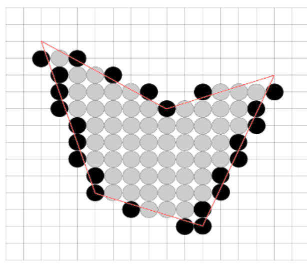
What happens if two rectangles share an edge? What color do we color the boundary pixels? So as a general rule of thumb only color pixels in $[x_{min}, x_{max})$ and $[y_{min}, y_{max})$.

7.2 Polygons

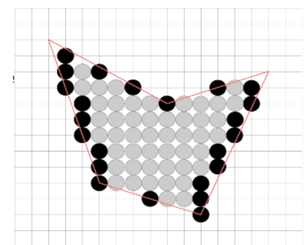
We can take a similar approach where we take a scan-line through each height y and fill the interior xs. Two issues though. First, notice below the intersections a and d are integer values and fall perfectly on the line. The intersections b and c do not fall precisely on the line. Second, shooting off this, which pixels are interior?



Option 1 : Midpoint Algorithm
Use the midpoint algorithm on each edge, fill in pixels between the found extrema points
Issue: The midpoint algorithm has no sense of in and out, so many extrema pixels are outside the polygon.

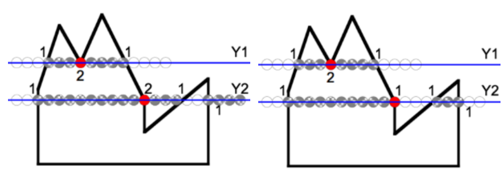


Option 2 : Strict Inside
Find intersections of scan line with edges, sort intersections by increasing x, fill pixels based on a parity bit B_p
B_p is initially even (off). At each intersection, invert the bit. Draw when odd, don't draw when even.
Issue: What do we do with fractional x? What about intersections at vertices? Shared vertices? Vertices that define a horizontal edge?



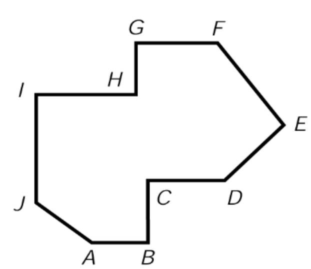
In option 2, we come across an issue where vertices will be counted twice, and flip the parity bit B_p twice. We need to account for the following by comparing the y value with the y value of neighboring vertices:

- Both neighboring vertices on the same side of the scan line? Ignore
- Both neighboring vertices on different sides of the scan line? Count Once



Then for filling horizontal edges, how do we handle this?

1. Apply open and closed status to vertices to other edges
   - y_{min} is closed, y_{max} is open
2. On AB, A is at y_{min} for JA. AB does not contribute. B_p is odd and draws AB
3. Edge BC has y_{min} at B, but AB does not contribute. B_p becomes even and drawing stops
4. Start drawing at IJ. B_p is odd
5. C is y_{max} for BC. B_p stays odd
6. D is y_{min} for DE. B_p becomes even. Stop drawing and ignore CD
7. I is y_{max} for IJ. B_p is even. No drawing occurs
8. B_p still even, ignore IH
9. H is y_{min} for GH. B_p becomes off. Draw FE
10. Ignore GF



function FillPolygon(P)
  for $\forall$ edges $E_i \in P.E$ **do**
    if $x_{min_i} == x_{max_i}$ **then**
      Ignore
    end if
    if y_{max} is on a scanline **then**
      Ignore
    end if
    if $y_{min} \leq y_s \leq y_{max}$ **then**
      Add E_i to scan line y_s's edge list
    end if
  end for
  for $\forall$ scanline $y_s \in [y_{min}, y_{max}]$ **do**
    Calculate intersections with edges on list
    Sort intersections by x
    Perform parity bit scan line filling
    Check intersection special cases
  end for
  Clear scan lines edge list
end function

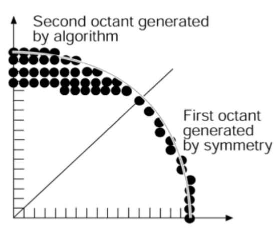
How do we handle slivers though? Places where the polygon is thinner than the width of a pixel?

It says anti-aliasing but doesn't describe it!!!

7.3 Curved Objects

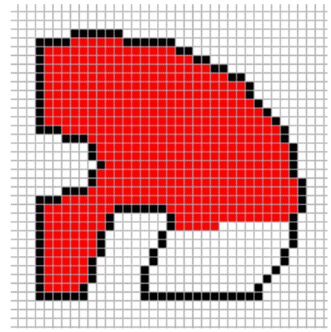
It's hard to do this in a general case, but circles and ellipses are easy.

Use the midpoint algorithm to generate boundary points. Fill in with horizontal pixel spans, then use symmetry.



7.4 Boundary Fill Algorithm

Start with an internal point (x, y), color it, check neighbors for filled or border color. Recurse on valid neighbors
However, it may make mistakes if parts of the space were already filled with the fill color. It also requires a huge stack size to keep track of all the recursion.



function BoundaryFill($x, y, fill, bound$)
    if Color(x, y) $\neq fill$ and Color(x, y) $\neq bound$ **then**
        Color(x, y) $:= fill$
        BoundaryFill($x + 1, y, fill, bound$)
        BoundaryFill($x, y + 1, fill, bound$)
        BoundaryFill($x - 1, y, fill, bound$)
        BoundaryFill($x, y - 1, fill, bound$)
    end if
end function

8 Color

8.1 Vocab

Hue : The flavor of color we see. Red, green, blue, etc. Based on the dominant wavelengths

Saturation : How much of the flavor we see. Based on excitation purity

Lightness : Self reflecting objects. Based on luminance

Brightness : Self luminous objects. Based on luminance

8.2 Physics and Eyes

Electromagnetic spectrum

wavelengths

what do we percieve?

highest degree of sensitivity is also where sun is

8.3 Intensity

Achromatic : Light without color. Defined in terms of three types of energy, intensity, luminance, and brightness

If we have a limited number of shades, how do we decide how to distribute them?

Bad Idea 1:
128 levels between 0.0 and 0.9
128 levels between 0.9 and 1.0
This creates discontinuities at 0.9 and an uneven distribution of samples

Bad Idea 2:
Distribute them evenly
This isn't how the human eye works. It deals in relatives, not absolutes (like a sith would). The intensity change between 0.10 and 0.11 looks like the change from 0.50 and 0.55 since they're both 10%.

Good Idea : Start with I_0, build up to $I_{255} = 1$

$$
\begin{aligned}
I_0 &= \text{Given} \\
I_1 &= rI_0 \\
I_2 &= rI_1 = r^2 I_0 \\
\ldots &= \ldots \\
I_{255} &= r^{255} I_0 = 1 \\
r &= (\frac{1}{I_0})^{1/255} = I_0^{-1/255} \\
r^j &= I_0^{-j/255} \\
I_j &= r^j I_0 = I_0^{(1-j/255)} = I_0^{(255-j)/255} \\
r &= (\frac{1}{I_0})^{1/n} \\
I_j &= r^j I_0 = I_0^{(n-j)/n}
\end{aligned}
$$

When selecting intensities, take into account the dynamic range of the device and choose a minimum intensity such that its $[\frac{1}{500}, \frac{1}{200}]$ of the maximum.

Gamma Correction : Adjusting intensities to compensate for a device. This requires a look up table

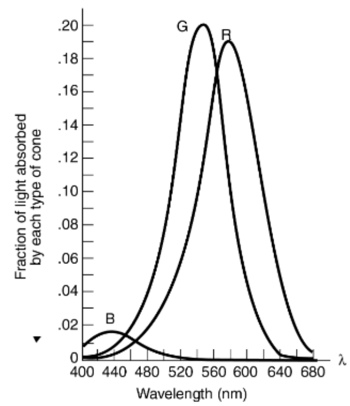
$$
\begin{aligned}
I &= v^{\gamma} \\
\gamma &= 2 \rightarrow 2.5
\end{aligned}
$$

But how many intensities are enough? The human eye cannot see changes $< 1\%$:

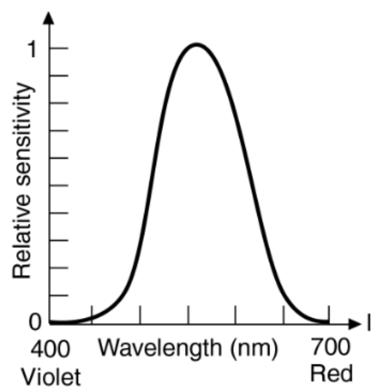
$$
\begin{aligned}
1.01 &= (\frac{1}{I_0})^{1/n} \\
n &= \log_{1.01}(\frac{1}{I_0}) \\
I_0 &= \frac{1}{200} \\
n &= 532
\end{aligned}
$$

8.4 Physics Background

Tristimulus Theory : The human retina has three color sensors called cones. These cones are tuned to **red**, **green**, and **blue** wavelengths.



Luminous Efficiency Function : The eye's response to light of constant luminance as the dominant wavelength is varied



Eyes can distinquish 100,000s of colors side by side. When the colors only differ in hue, colors are only distinguishable when the wavelength difference is [2nm, 10nm], but most within 4nm. This means 128 fully saturated hues can be distinguished. Less saturation makes us less sensitive to changes in hues. We are more sensitive at spectrum extremes to changes in saturation. There are about 23 distinguishable saturation grades.

8.5 Color Models

8.5.1 RGB

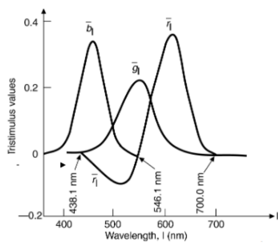
RGB is an additive model that defines colors in weighted sums of **red**, **green**, and **blue**. Some colors may need values less than 0 to match the wavelengths, so some colors cannot be represented this way.

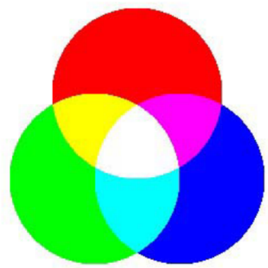
Primary colors are obviously **red**, **green**, and **blue**, which gives us the secondary colors:

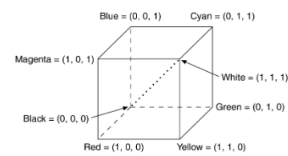
- yellow = **red** + **green**
- cyan = **green** + **blue**
- magenta = **red** + **blue**

- white = **red** + **green** + **blue**
- black = none

RGB is primarily used in monitors and TVs and things that emit light



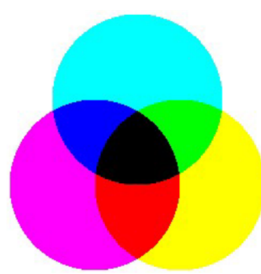
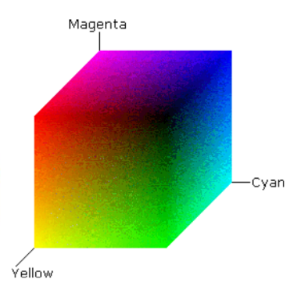




8.5.2 CMY(K)

On the flipside, **CMY**K describes a hard copy color output. Since it's mostly used in printing ink, the colors are reflected light. This makes **CMY**K a subtractive color model. For example **cyan** ink absorbs **red** light and reflects **green** and **blue**. This gives the following secondary colors:

- **blue** = **cyan** + **magenta**
- **red** = **magenta** + **yellow**
- **green** = **cyan** + **yellow**
- black (theory) = **cyan** + **magenta** + **yellow**
  - In practice, black is it's own ink instead of wasting and mixing all three
- white = no ink

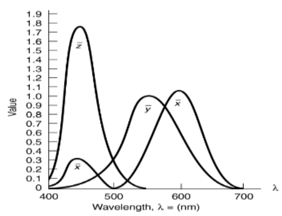



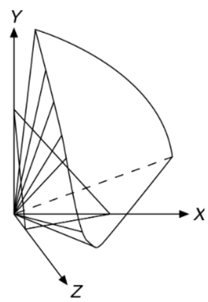
8.5.3 XYZ

This isn't actually real colors here. It's a standard defined by the International Commission on Illumination (CIE) in 1931 to avoid negative weights.

$$X = \int P(\lambda)\bar{x_\lambda}d\lambda$$
$$Y = \int P(\lambda)\bar{y_\lambda}d\lambda$$
$$Z = \int P(\lambda)\bar{z_\lambda}d\lambda$$

We can create a cone of visible colors in CIE space as shown on the $X + Y + Z = 1$ plane. Since there is constant luminance, it only depends on wavelength and saturation.



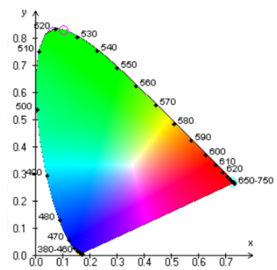


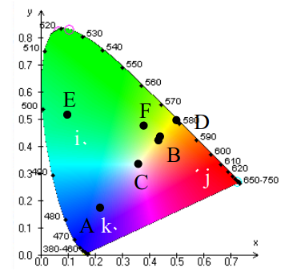
We can plot colors on this $X + Y + Z = 1$ plane and normalize by brightness to get the CIE chromaticity diagram

$$X = \frac{X}{X + Y + Z}$$
$$Y = \frac{Y}{X + Y + Z}$$
$$Z = \frac{Z}{X + Y + Z}$$

To use this diagram:

- C is "white" and close to $x = y = z = \frac{1}{3}$
- E and F can be mixed to produce any color along the line EF
- Dominant wavelength of B is where the line from C through B meets the spectrum (D)
- $\frac{BC}{DC}$ gives the saturation
- A and B are complementary colors and combine to form white light
- Colors inside ijk are linear combinations of i, j, and k





8.5.4 YIQ

This one was developed by the National Television System Committeee (NTSC)

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.528 & 0.311 \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{G} \\ \mathbf{B} \end{bmatrix}$$

Y is the same as the XYZ model and represents brightness. It uses 4MHz of bandwidth. This is the only signal black and white TVs use.

I contains the **orange**-**cyan** hue information (skin tones) and uses about 1.5MHz of bandwidth.

Q contains the **green**-**magenta** hue information. It also uses about 1.5MHz of bandwidth.

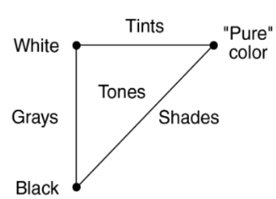
8.5.5 HS[B—V] and HSL

Both use a relationship of tints, shades, and tones

Tints : Mixture of a color with white

Shades : Mixture of a color with black

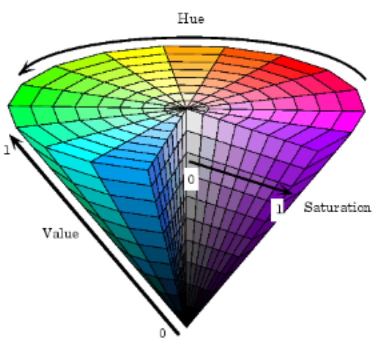
Tones : Mixture of a color with white and black



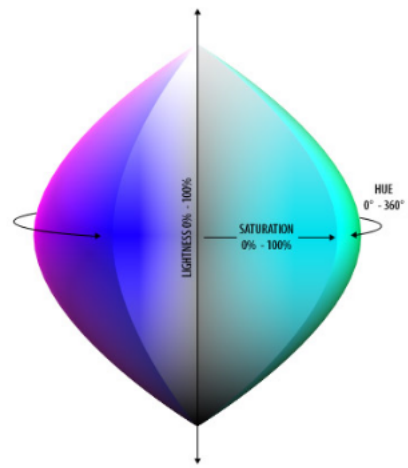
HS[B—V] : Hue, Saturation, Brightness/Value
Hue is the actual color measured in degrees around the cone

- **red** = 0 = 360
- **yellow** = 60
- **green** = 120
- ...

Saturation is the purity of the color, measured in percent from the center of the cone. 0% is white and hue is meaningless. 100% will give a pure shade of the color.
Brightness is also measured in percent. It's measured from the tip of the cone. At 0% brightness, neither hue nor saturation matter. At 100% bright, we have a pure tint of the color.



HS[B—V] : Hue, Saturation, Lightness
Hue is defned the same as in HSB with complimentary colors 180 degrees apart.
Saturation once again the same as the HSB model with the percentage from the center.
Lightness is now a gray scale accross the axis from 0, pure white, to 1, pure black. Pure hues now lie where $L = 0.5$.
It's like we took the HSB model, duplicated it and changed the black to white, then flipped it over and combined the two



9 Solid Modeling

Now, we're gonna introduce a mathematical theory of a solid shape. This theory includes the following components:

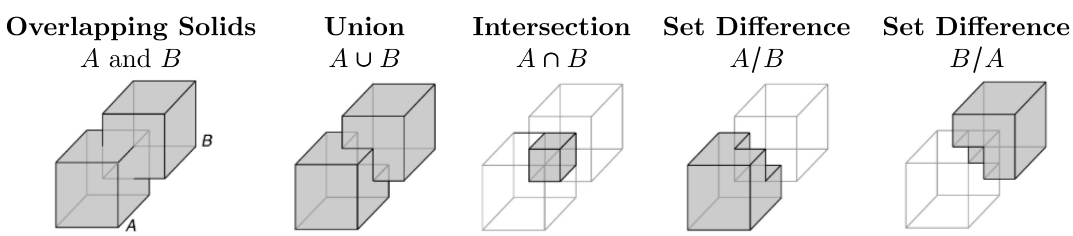
- A domain of objects
- Each object has a clearly defined inside and outside
- A set of operations on the domain
- Unambigious, accurate, unique, compact, and efficient representation

Solids : A set of points defined as interiors and boundaries

Boundary Points : Points where the distance to the object and it's complement is zero

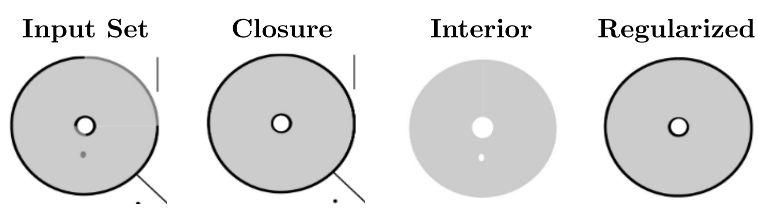
Interior Points : All other points in the set

Closure : Union of interior and boundary points (another word for solid)



By performing these operations on 3D objects, we can also creat "non-3D objects" or objects with non-uniform dimensions. Objects need to be regularized.

Regularization : Taking the closure of the interior

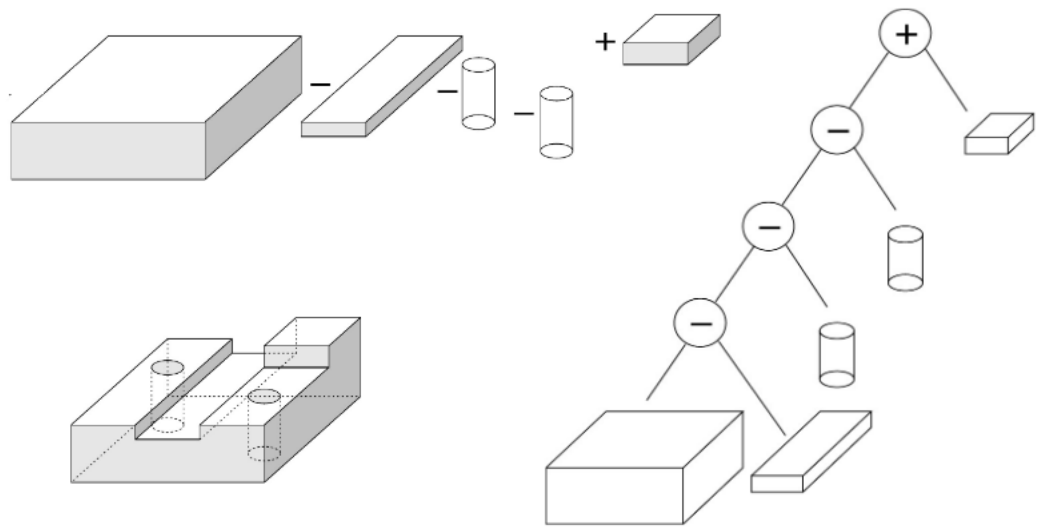


Another example:

9.1 Constructive Solid Geometry (CSG)

CSG : A tree structure combining primitives via regularized operations

It can also be represented as a topologically sorted DAG



Issues:

- **Non Uniqueness :** More than one way to build the same model
- **Primitive Choice :** Minor changes in the primitives can drastically change a model
- **Simple Models :** How would we sculpt surfaces? or deform them?

Benefits:

- Found in basically every CAD system
- Elegant, conceptually and algorithmically appealing
- Good for:
  - Rendering
  - Ray Tracing
  - Simulation
  - BRL CAD

Then to evaluate points on the surface of a CSG:

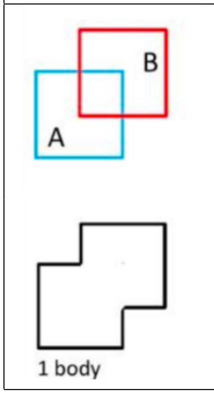
1. Compute points on the surfaces of the primitives
2. Test if points will be on the surface of the evaluated CSG model
3. Use rules based on the inside outside status of the points

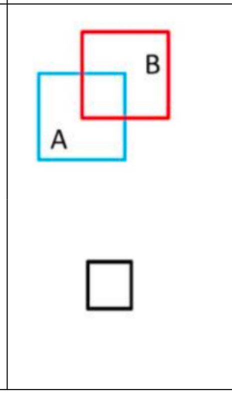
4. Display valid points

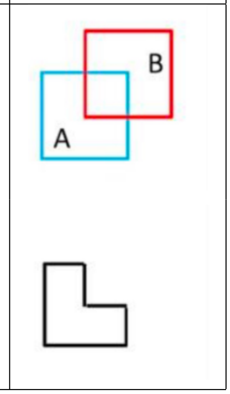
Rules:

$\mathbf{A \cup B}$	$A_{boundary} \wedge (\neg B_{interior})$ $B_{boundary} \wedge (\neg A_{interior})$
$\mathbf{A \cap B}$	$A_{boundary} \wedge (B_{interior} \vee B_{boundary})$ $B_{boundary} \wedge (A_{interior} \vee A_{boundary})$
$\mathbf{A/B}$	$A_{boundary} \wedge (\neg B_{interior})$ $B_{boundary} \wedge (A_{interior} \vee A_{boundary})$
$\mathbf{B/A}$	$A_{boundary} \wedge (B_{interior} \vee B_{boundary})$ $B_{boundary} \wedge (\neg A_{interior})$

Union	**Intersection**	**Difference**



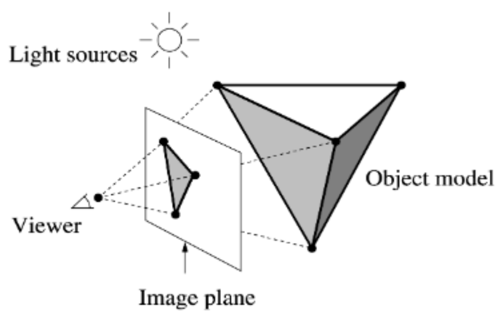


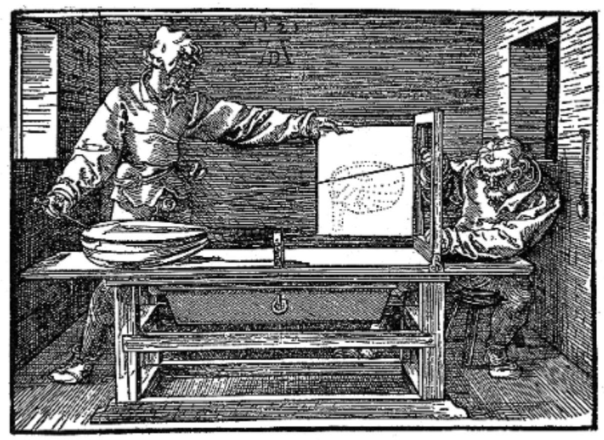


10 3D Viewing

10.1 Projection

Similar to clipping 2D objects to fit a window, we can't display 3D objects entirely on a screen. How do we project a 3D object onto a 2D plane to display it.



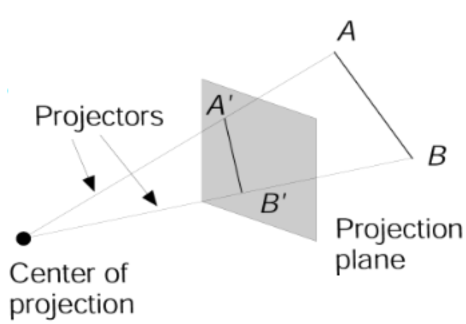


The general process for 3D viewing is as follows:

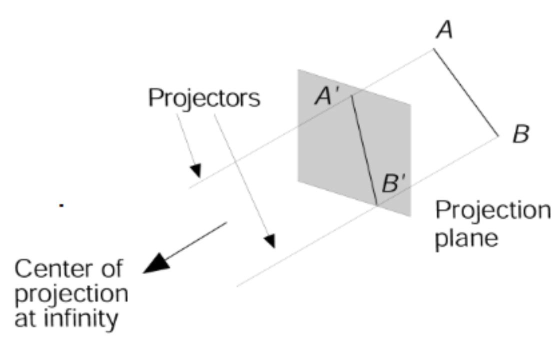
1. Clip the 3D world coordinates and output primitves against the view volume
2. Project the clipped world coordinates onto the projection plane
3. Transform the projection plane into the viewpoint 2D coordinates for displaying

10.1.1 Planar Geometric Projections

Perspective Projection
Similar to a photograph, there's a single viewing location that rays intersect with

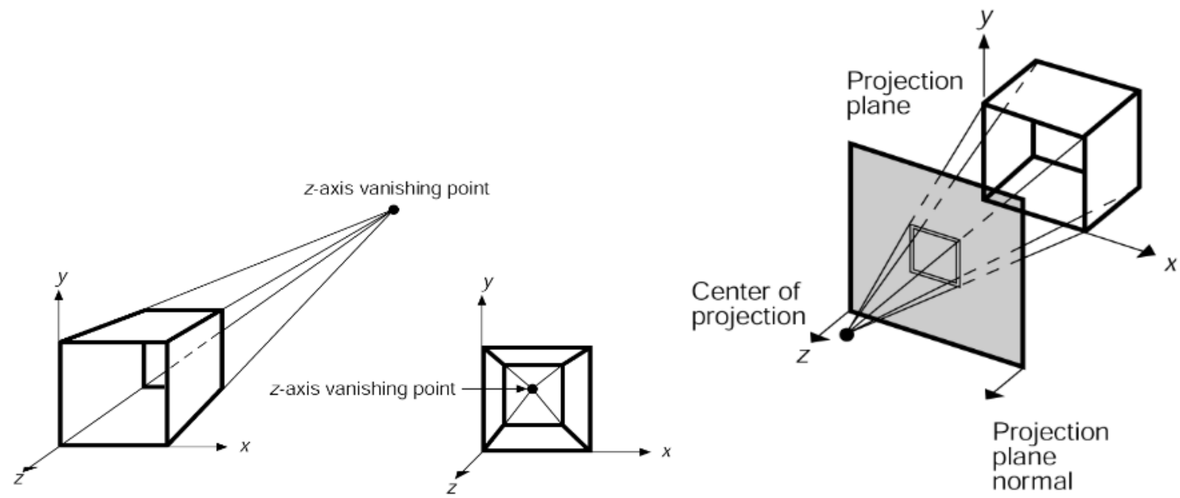


Parallel Projection
The viewing location is at ∞ and it's good for capturing shape and dimensions.

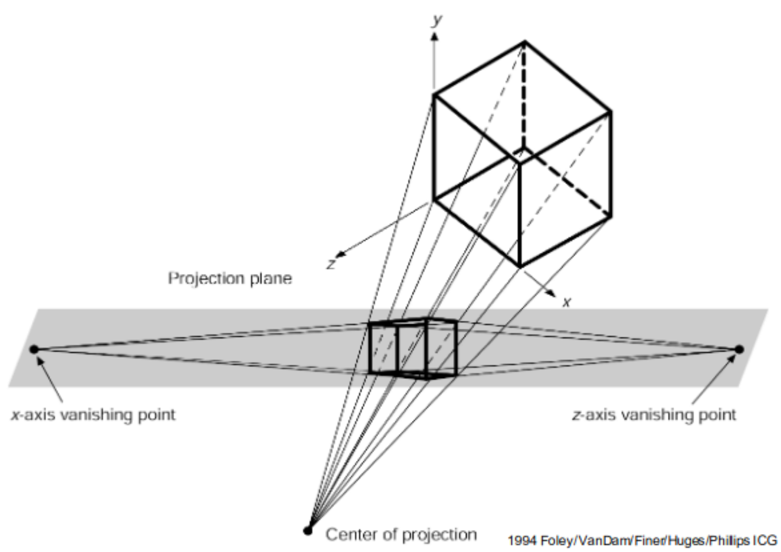


10.2 Perspective Projections

Vanishing Point : The point at which all viewing rays converge to. These lines are not parallel to the projection plane



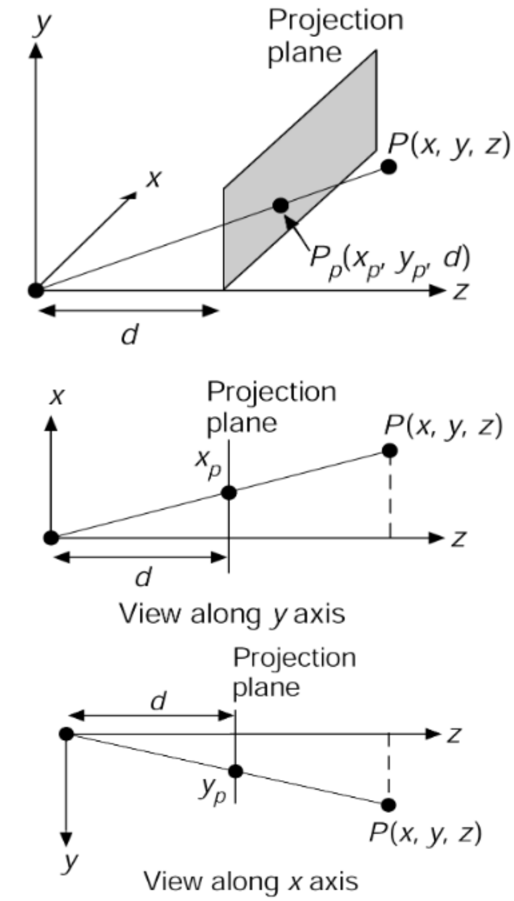
In addition to a single perpespective point, we can cut two axis at the same time with two vanishing points.



Now how do we actually compute this?

Given point P, we can project it onto a plane as point P_P using similar triangles.

Assuming that the projection normal is the z-axis:
x **Direction Ratio :** $\frac{z}{d} = \frac{x}{x_p}$
y **Direction Ratio :** $\frac{z}{d} = \frac{y}{ybp}$



$$x_p = \frac{x}{z/d}$$

$$y_p = \frac{y}{z/d}$$

$$M_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

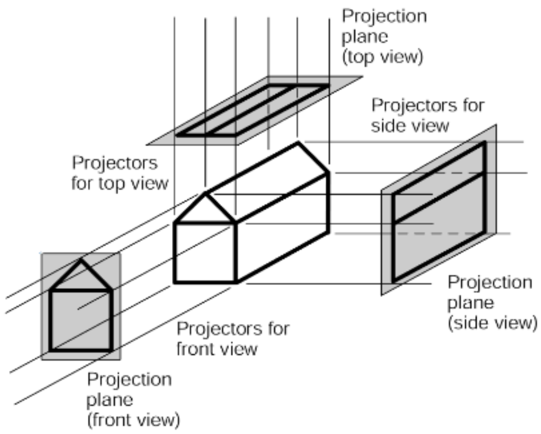
10.3 Parallel Projections

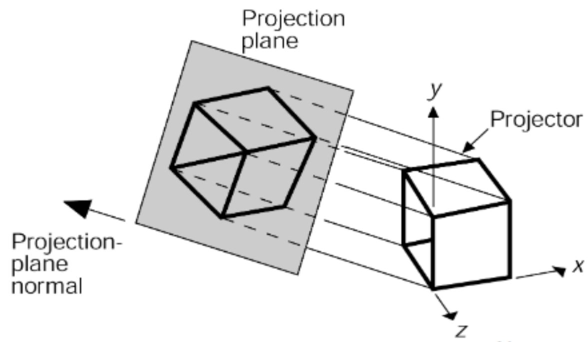
Orthographic Projections : Projection direction vector and projection plane normal are the same vectors

These are primarily used for engineer diagrams. These are good for keeping the dimensions of a model and faces of a model.

There are three 3 axonometric orthogonal projections for each x, y, and z axis. There are 8 isometric projections for each octant where the angles to each axis are equal.

$$M_{ort} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



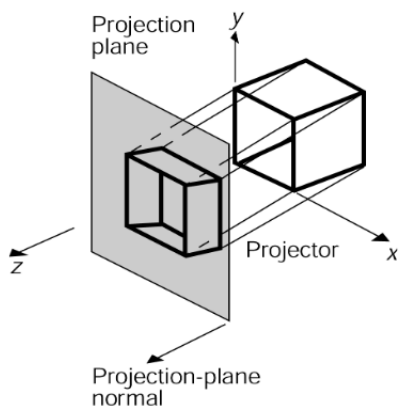


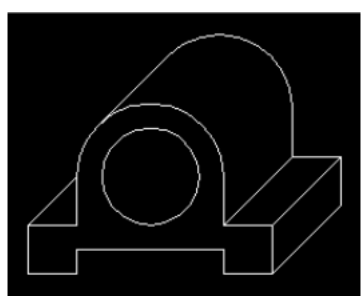
Oblique Projections : Projection direction vector and projection plane normal are different vectors

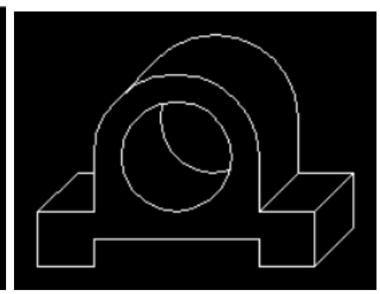
This preserves certain angles and distances. These are better for illustrations and movement.

Cavalier : All lines are their true lengths (middle)

Cabinet : Receding lines are shorted by one-half of their true length to approximate perspective foreshortening (right)





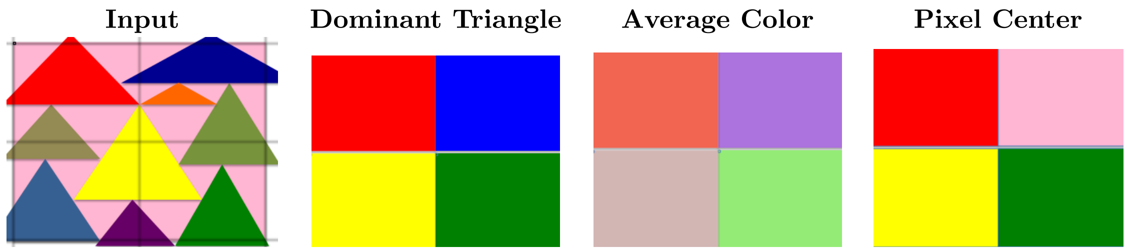


10.4 Scanline Rendering

Ray Tracing : Cast a ray for every pixel and see what geometry it intersects

Rasterization : Examine every triangle and see which pixels it covers

When choosing the color of the pixel to fill, we have options:



When using scanline rendering for rasterization, the process is exactly the same for 2D objects and finding the pixels of best match based on vertices.

Supersamping : Using the scanline algorithm a bunch of times to get an average picture

10.5 Back Face Culling

Since drawing all these triangles can get heavy and sometimes, not all of them are necessary, we use back-face culling to cut down on unnecessary shapes.

Assumptions for running this process:

- Object approximated as closed polyhedron
- Polyhedron interior is not exposed by the front cutting plane
- The eye pint is not inside the object
- Right hand vertex ordering defines a normal outside

Recall the normal N of a triangle $\{p_0, p_1.p_2\}$ is:

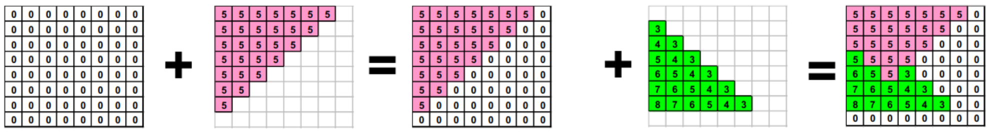
$$N = \frac{(p_1 - p_0)(p_2 - p_0)}{||(p_1 - p_0)(p_2 - p_0)||}$$

Then, the process goes as follows:

function BackFaceCulling
    Perform canonical transformation
    Examine the normal to the face $N_k = (x_k, y_k, z_k)$
    // This is the only test necessary for single convex polyhedrons
    // More general cases compare $N_k \circ V$
    if $z_k >= 0$ **then**
        Draw the face
    else
        Face is a Back-Face, do not draw
    end if
end function

10.6 Z-Buffering

Z buffering, also called depth buffering, is a visible surface detection algorith. It's very simple, assuming we have the polygons rasterized pixels. We just walk through the pixels, and draw them if they're in front of the pixel already on the screen.



function ZBUFFER
    for $y := 0 \rightarrow YMAX$ **do**
        for $x := 0 \rightarrow XMAX$ **do**
            $F[x][y] := BACKGROUND_COLOR$
            $Z[x][y] := ZMIN$
        end for
    end for
    for $\forall$ polygons P_i **do**
        for $\forall$ pixels $p_i \in P_i$ **do**
            $p_z := z$ value of p_i
            if $p_z > Z[x][y]$ **then** *// New pixel is closer*
                $F[x][y] :=$ COLOR(p_i)
                $Z[x][y] := p_z$
            end if
        end for
    end for
end function

function ZBUFFER-FRONTBACKCLIPPING
    for $y := 0 \rightarrow YMAX$ **do**
        for $x := 0 \rightarrow XMAX$ **do**
            $F[x][y] := BACKGROUND_COLOR$
            $Z[x][y] := -1$
        end for
    end for
    for $\forall$ polygons P_i **do**
        for $\forall$ pixels $p_i \in P_i$ **do**
            $p_z := z$ value of p_i
            if $p_z > Z[x][y]$ and $p_z < FRONT$ **then**
                // New pixel is closer and behind front plane
                $F[x][y] :=$ COLOR(p_i)
                $Z[x][y] := p_z$
            end if
        end for
    end for
end function

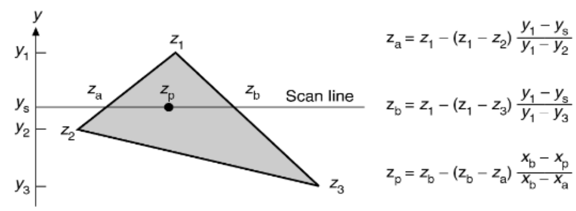
Z Interpolation : Simplify the calcuation of z by exploiting the fact a triangle is planar. Two parts:

1. Interpolate z values along the edges

2. Interpolate z values along the scan line

Three Special Cases:

- Horizontal edge
- Degenerate triangle
- Single point



10.7 Depth Cueing

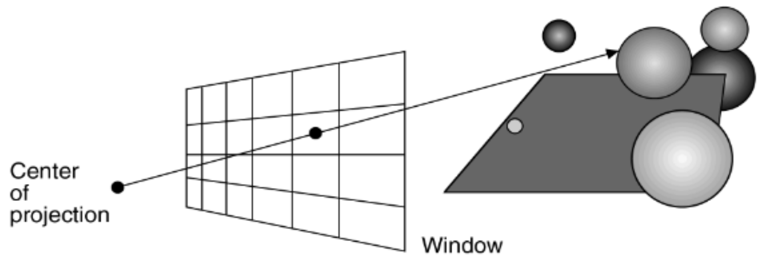
Objects that are farther away are darker, objects that are closer are brighter.

$$Color' = Color\frac{z - far}{near - far}$$

10.8 Ray Tracing

Also called Ray Casting. This determines the visible surface by tracing rays of light from the viewers eye to the objects. This allows for more rendering like reflections and such.

```
function RayTracing
    for ∀ scanlines s ∈ S do
        for ∀ pixels p ∈ s do
            Find a ray from the center of the projections through the pixel
            C_o := The closest object in the scene
            C_l := The location of the intersection with the closest object
            for ∀ objects o ∈ O do
                if o is closer than C_o then
                    C_o := o
                    C_l := The location of the intersection with o
                end if
            end for
            Color(p) := Color(C_o[C_l])
        end for
    end for
end function
```



To actually compute this, we call the center of project (x_0, y_0, z_0) and the point on the window (x_1, y_1, z_1) to get the following from the parametric equation of a line:

$$\begin{aligned}x &= x_0 + t\Delta x\\ y &= y_0 + t\Delta y\\ z &= z_0 + t\Delta z\end{aligned}$$

Intersection : Sphere

$$\begin{aligned}r^2 &= (x-a)^2 + (y-b)^2 + (z-c)^2\\ r^2 &= (x_0 + t\Delta x - a)^2 + (y_0 + t\Delta y - b)^2 + (z_0 + t\Delta z - c)^2\\ 0 &= (x_0 + t\Delta x - a)^2 + (y_0 + t\Delta y - b)^2 + (z_0 + t\Delta z - c)^2 - r^2\\ 0 &= (\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + (\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - a))2t + (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2\end{aligned}$$

Since the equation (albeit messy) is quadratic in terms of t, we can solve it with the quadratic formula and get the following cases:

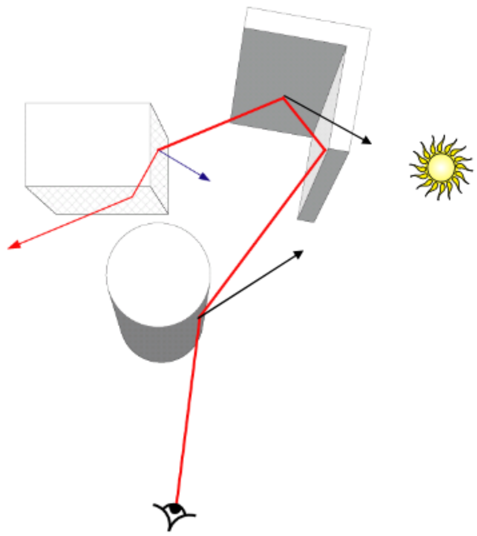
- No real roots: No intersections
- One real root: Ray grazes the sphere
- Two real roots: Two points of intersection

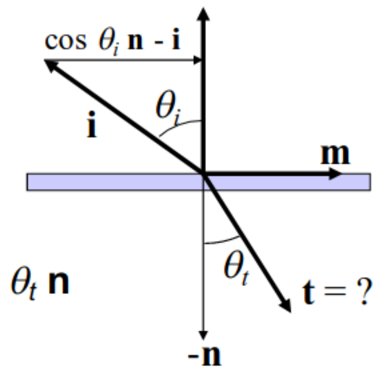
Intersection : Polygon

$$\begin{aligned}0 &= Ax + By + Cz + D && \text{Plane of intersection}\\ t &= -\frac{Ax_0 + By_0 + Cz_0 + D}{A\Delta x + B\Delta y + C\Delta z} && \text{Substitution}\end{aligned}$$

If the denominator is 0, the ray is parallel to the plane. Otherwise, project the polygon and point orthographically on the coordinate plane, then perform the point on Polygon test.

The actual ray tracing comes from when there are reflections and shadows and transparencies and refractions to take into account.





To measure refraction:

Snell's Law : $\eta_i \sin\theta_i = \eta_t \sin\theta_t$

$$
\begin{aligned}
\eta &= \frac{\eta_i}{\eta_t} \\
\eta &= \frac{\sin\theta_i}{\sin\theta_t} \\
m &= \frac{n\cos\theta_i - i}{\sin\theta_i} \\
\cos\theta_t &= \sqrt{1-\sin^2\theta_t} \\
\cos\theta_t &= \sqrt{1-\eta^2\sin^2\theta_t} \\
t &= m\sin\theta_t - n\cos\theta_t \\
&= \frac{\sin\theta_t}{\sin\theta_i}(n\cos\theta_i - i) - n\cos\theta_t \\
&= n(\eta\cos\theta_t - \cos\theta_t) - i\eta \\
t &= n(ni\eta - \sqrt{1-\eta^2(1-(ni)^2)}) - i\eta
\end{aligned}
$$