COMP3271 Computer Graphics

Geometry

2019-20

Objectives

Introduce the elements of geometry

- Scalars
- Vectors
- Points

Develop mathematical operations among them in a coordinate-free manner

Define basic primitives

- Line segments
- Polygons

Basic Elements

Geometry is the study of the relationships among objects in an n-dimensional space

- In computer graphics, we are interested in objects that exist in three dimensions

Want a minimum set of primitives from which we can build more sophisticated objects

We will need three basic elements

- Scalars
- Vectors
- Points

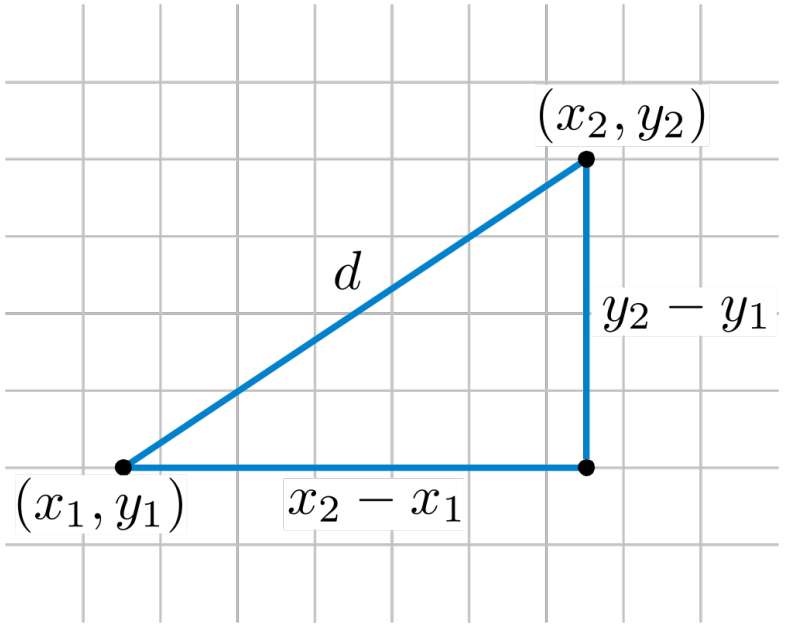
Cartesian Geometry

Points were at locations in space $\boldsymbol{p}=(x, y, z)$

We derived results by algebraic manipulations involving these coordinates

Example:
A line is given by the equation $x + 2y = 4$

Example:
To find intersection of two lines by solving a system of two linear equations
$x + 2y = 4$
$3x - y = 1$



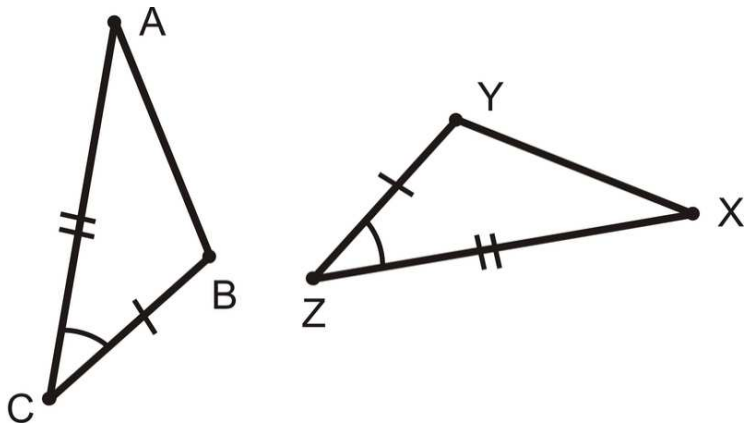
Cartesian Geometry

The Cartesian based approach was nonphysical

- Physically, points exist regardless of the location of an arbitrary coordinate system

However, most geometric results are independent of the coordinate system

- Example Euclidean geometry: two triangles are identical if two corresponding sides and the angle between them are identical



Cartesian Geometry

Example: To add two points P and Q

Let's try using coordinates to add two points.

(1) $P = (0,0)$ and $Q = (1,1)$, then $P+Q$ is $(1,1)$ which is the same as Q

(2) $P = (1,1)$ and $Q = (2,2)$, then $P+Q$ is $(3,3)$ which is a totally different point.

So it depends on the coordinate system and we cannot derive clear geometric interpretation from it.

Question: what is the geometric meaning in adding two points?

Coordinate Free Geometry

The three basic elements in CFG

- **Scalars**, **Vectors**, **Points**

Scalars

Scalars are simply real numbers which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutativity, inverses)

Scalars alone have no geometric properties

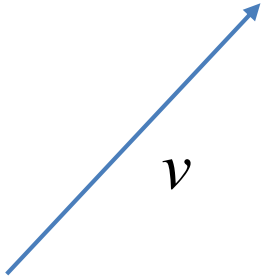
Vectors

Physical definition: a vector is a quantity with two attributes

- Direction
- Magnitude

Examples include

- Force
- Velocity
- Directed line segments
  - Most important example for graphics



Vector Operations

There is a zero vector

- Zero magnitude, undefined orientation

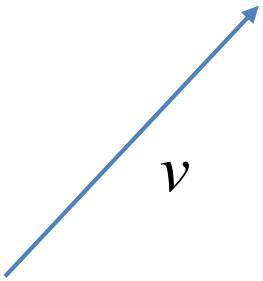
Every vector has an **inverse**

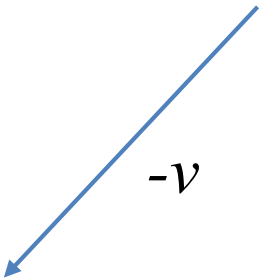
- Same magnitude but points in opposite direction

Every vector can be **multiplied by a scalar**

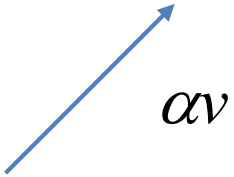
The sum of any two vectors is a vector

- Use head-to-tail axiom

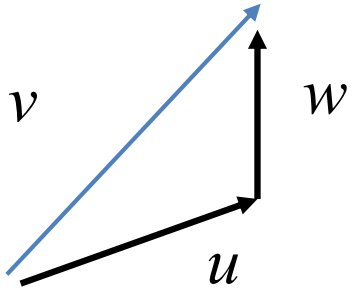




inverse



scaling



addition

Vector Operation

Magnitude of vector: $\|v\|$

Dot product: $\|v\|\|w\|\cos(\theta)$, where θ is the angle between the two vectors v and w

Cross product: $v \times w$, where v and w are 3D vectors. The cross product is a new vector perpendicular to v and w with magnitude $\|v\|\|w\|\sin(\theta)$

Linear Vector Spaces

Mathematical system for manipulating vectors

Operations

- Scalar-vector multiplication $u = \alpha v$
- Vector-vector addition: $w = u + v$

Expressions such as

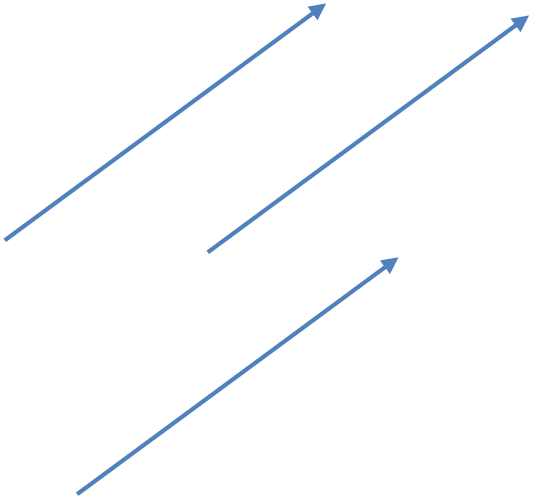
$$v = u + 2w - 3r$$

make sense in a vector space

Vectors Lack Position

These vectors are identical

- Same length and magnitude



Vectors spaces insufficient for geometry

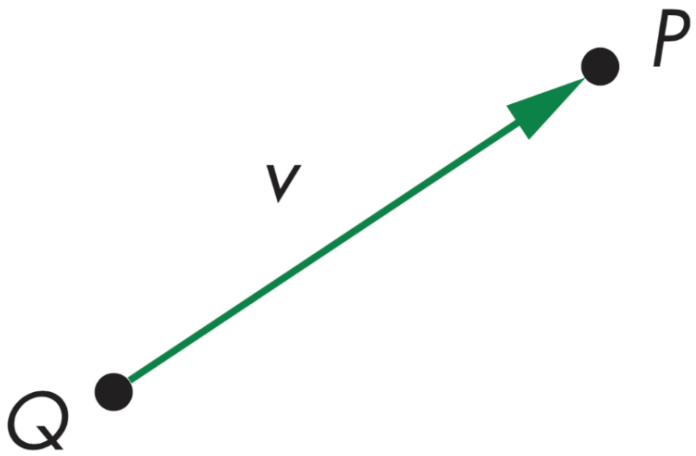
- Need points

Points

Location in space

Operations allowed between points and vectors

- Point-point subtraction yields a vector
- Equivalent to point-vector addition



$$\boldsymbol{v} = \mathrm{P} - \mathrm{Q}$$

$$\mathrm{P} = \boldsymbol{v} + \mathrm{Q}$$

Affine Spaces

Point + a vector space

Operations

- Vector-vector addition
- Scalar-vector multiplication
- Point-vector addition
- Scalar-scalar operations

For any point, define

- $1 \bullet \mathrm{P} = \mathrm{P}$
- $0 \bullet \mathrm{P} = \mathbf{0}$ (zero vector)

Coordinate-Free Geometry

Why CFG?

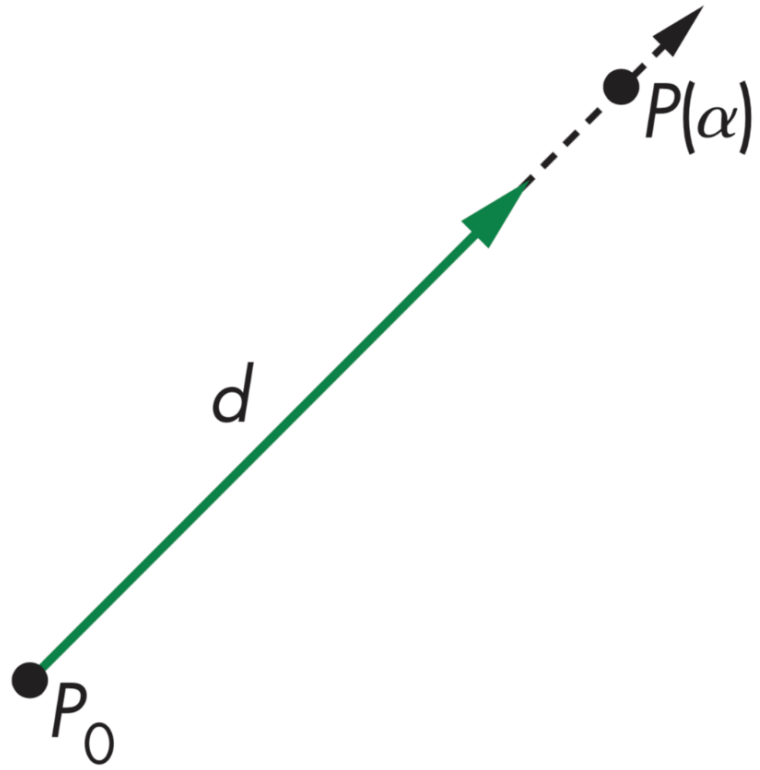
- We only care about the intrinsic geometric properties of the objects when reasoning about the geometric objects. e.g., congruent triangles
- CFG derivations usually provide much more geometric intuition and are simpler than using coordinates. e.g., dot product
- CFG provides kind-of "type checking" for geometric reasoning. e.g., point + vector

Lines

Consider all points of the form

- $P(\alpha) = P_0 + \alpha\ \mathbf{d}$
- Set of all points that pass through P_0 in the direction of the vector $\mathbf{d}$

Parametric equation



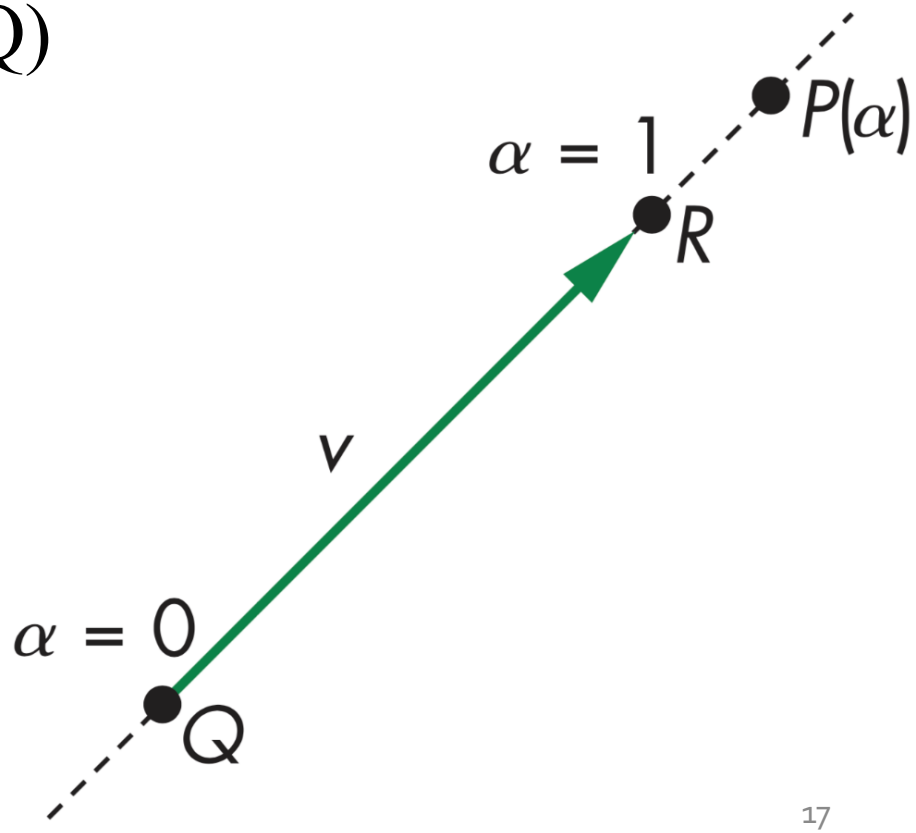
Rays and Line Segments

If $\alpha >= 0$, then $P(\alpha)$ is the ray (or half-line) leaving P_0 in the direction $\mathbf{d}$

If we use two points to define $\mathbf{v}$, then

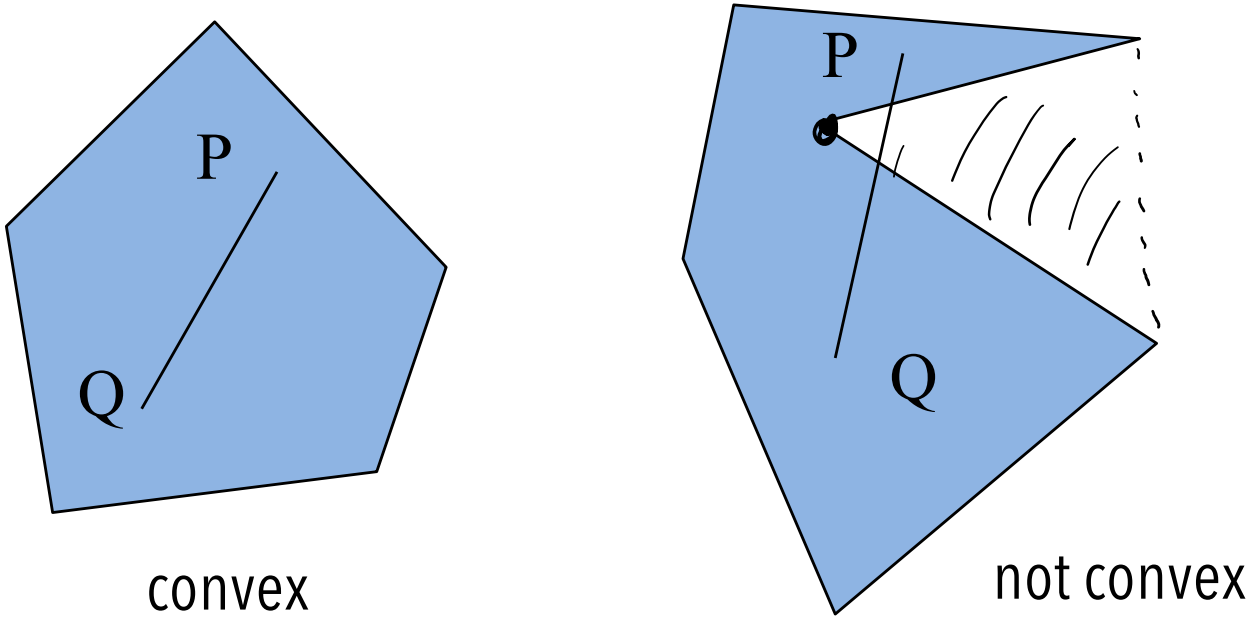
$$P(\alpha) = Q + \alpha\mathbf{v} = Q + \alpha\,(R - Q)$$
$$= \alpha R + (1 - \alpha)Q$$

For $0<=\alpha<=1$, we get all the points on the line segment joining R and Q



Convexity

An object is convex iff for any two points in the object all points on the line segment between these points are also in the object



This definition applies no matter what the object is or what the number of dimensions is

Affine Sums

The affine sum of n points $P_1, P_2, \ldots P_n$ is defined as

$$P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n$$

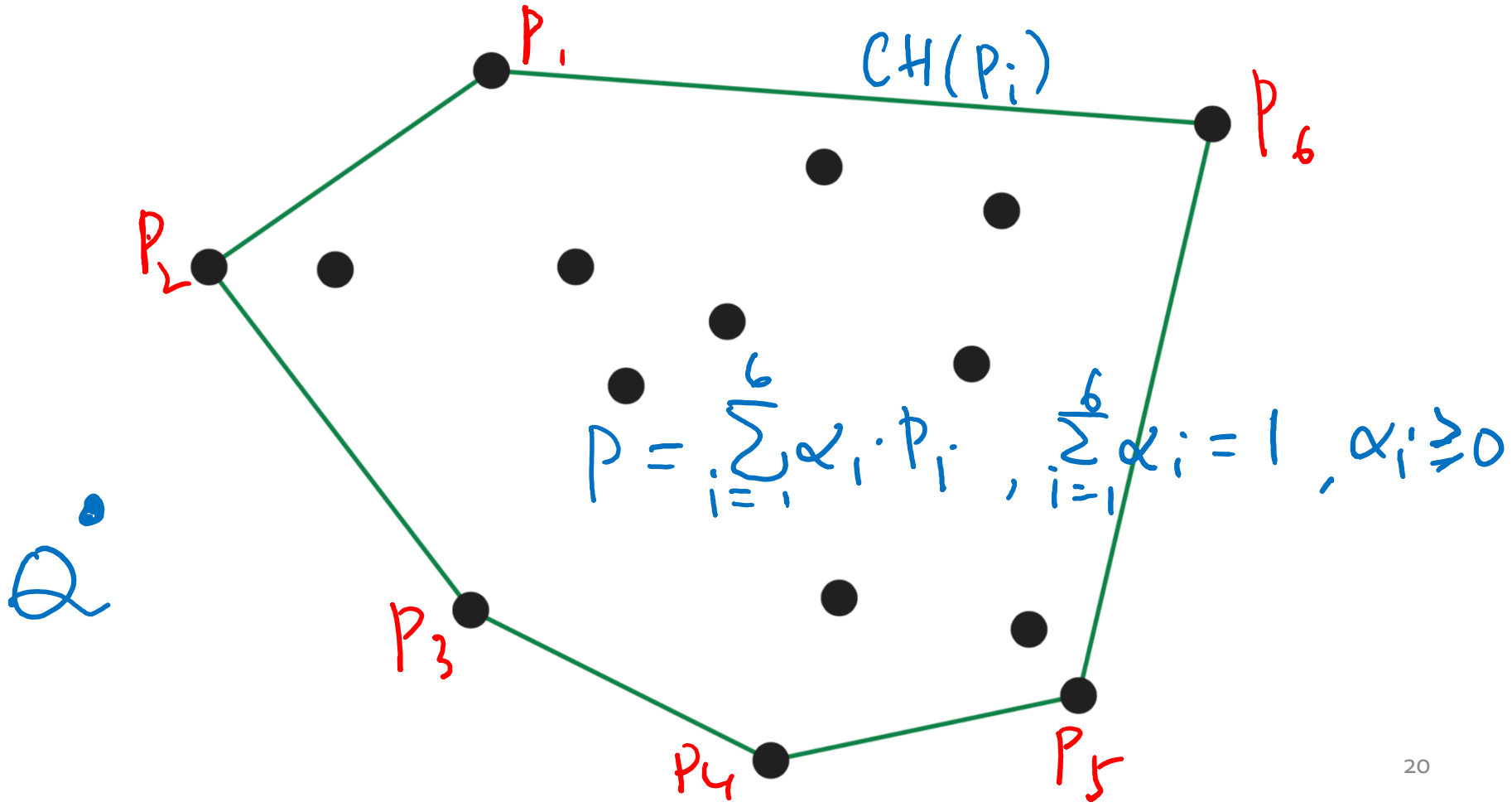
where $\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1$

If, in addition, $\alpha_i >= 0$ for all i, we have the **convex hull** of $P_1, P_2, \ldots, P_n$

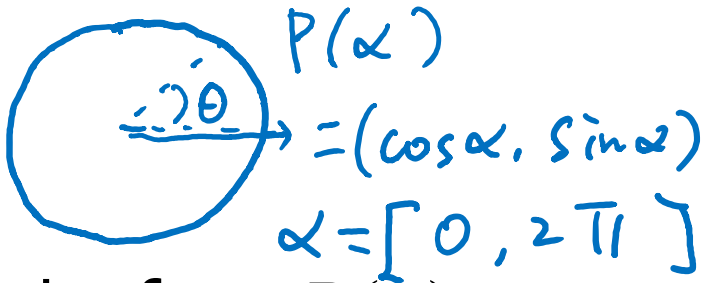
Convex Hull

Smallest convex object containing $P_1, P_2, \ldots, P_n$

Formed by "shrink wrapping" points



Curves and Surfaces

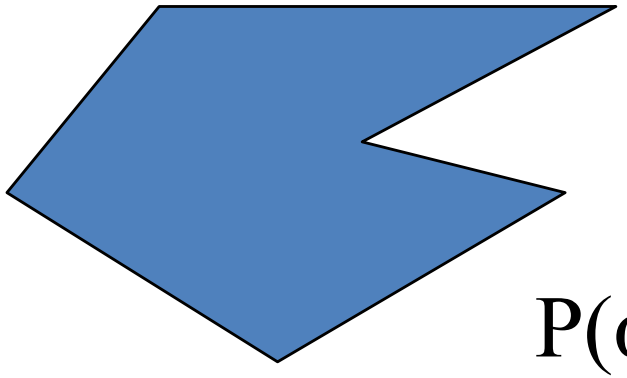


Curves are one parameter entities of the form $\mathrm{P}(\alpha)$ where the function is nonlinear

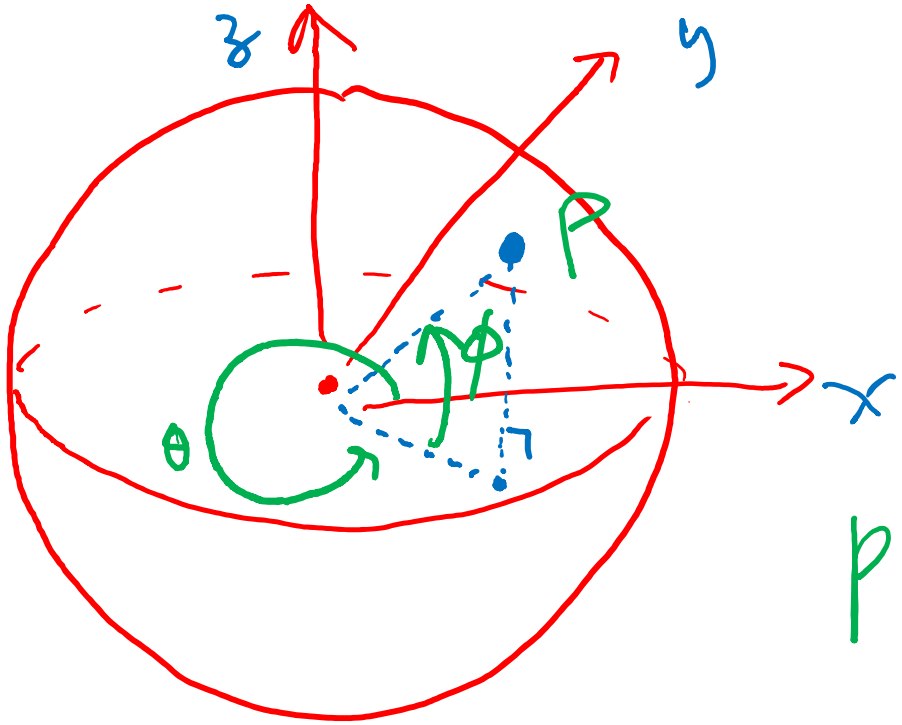
Surfaces are formed from two-parameter functions $\mathrm{P}(\alpha, \beta)$

- Linear functions give planes and polygons

$\mathrm{P}(\alpha)$



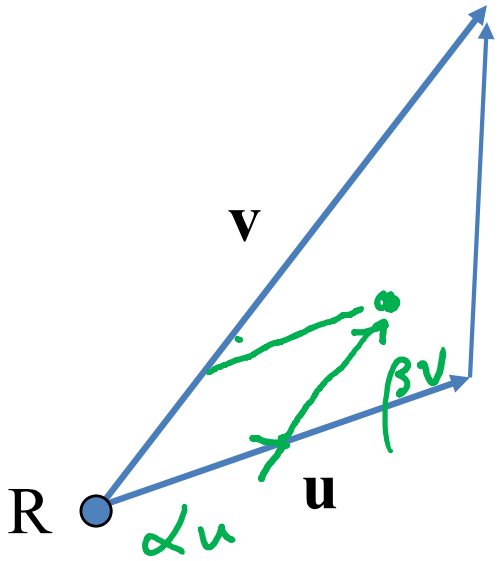
$\mathrm{P}(\alpha, \beta)$

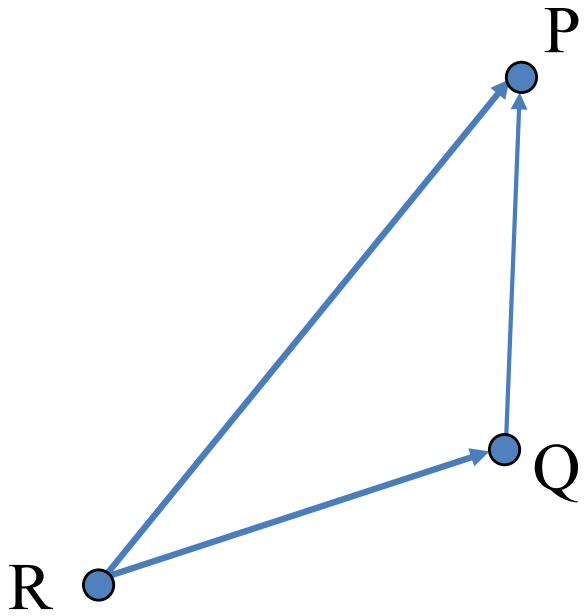


$$p(\theta,\phi) = \begin{pmatrix} \cos\theta\sin\phi \\ \sin\theta\sin\phi \\ \cos\phi \end{pmatrix}$$

Planes

A plane can be defined by a point and two vectors, or by three points





$$N(\alpha, \beta) = R+\alpha\mathbf{u}+\beta\mathbf{v}$$

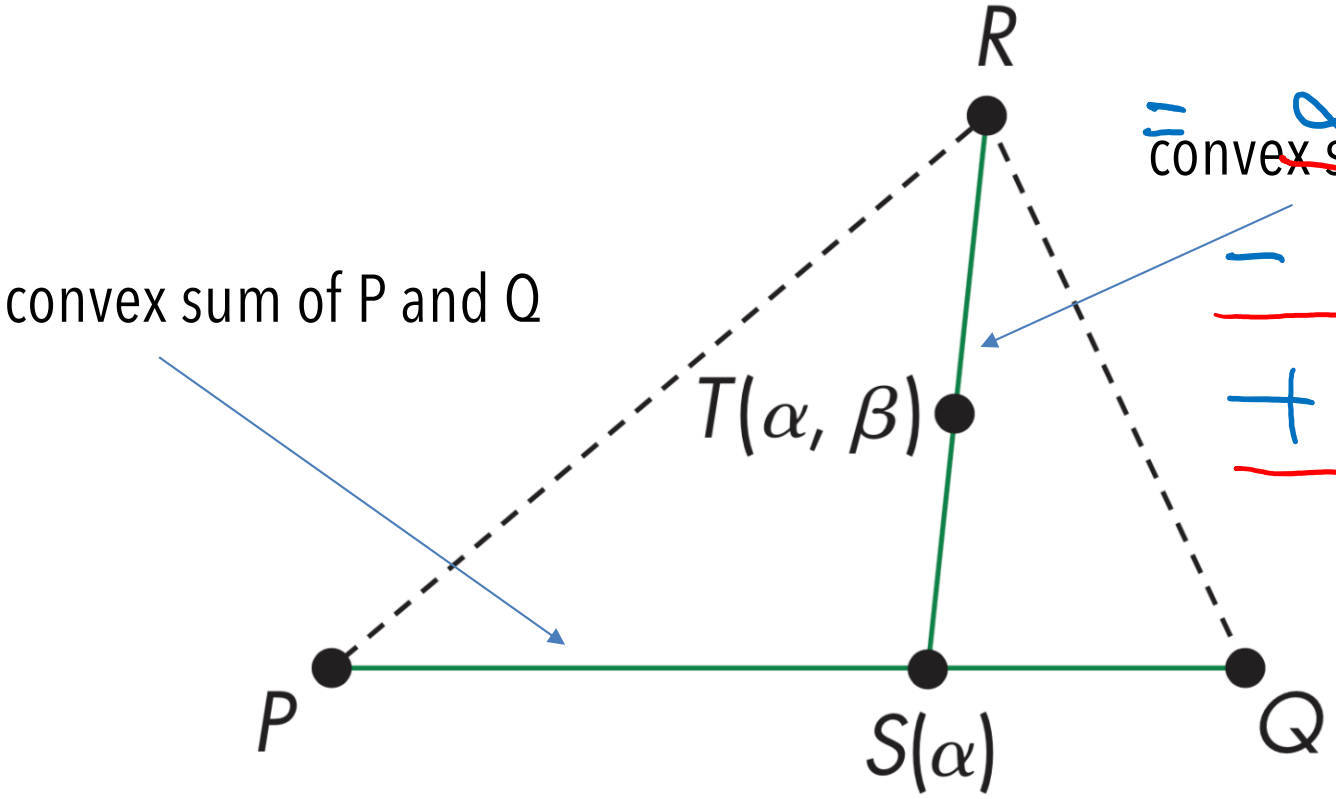
$$N(\alpha,\beta) = R+\alpha(Q-R)+\beta(P-R)$$

Triangles

$T(\alpha, \beta) = \beta\, S(\alpha) + (1-\beta) R$

$= \beta[\alpha P + (1-\alpha) Q] + (1-\beta) R, \quad \beta \in [0,1]$

$= \alpha\beta P + (1-\alpha)\beta Q + (1-\beta) R$



for $0 <= \alpha, \beta <= 1$, we get all points in triangle

$S(\alpha) = \alpha P + (1-\alpha) Q, \quad \alpha \in [0,1]$

Barycentric Coordinates

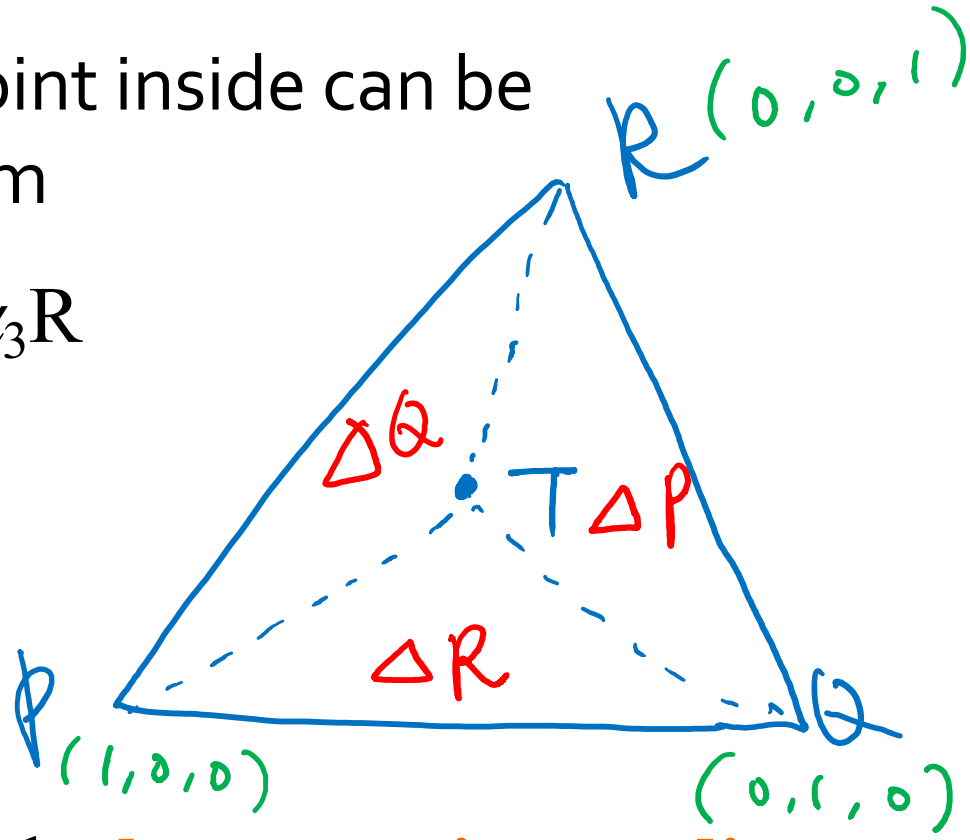
Triangle is convex so any point inside can be represented as an affine sum

$$P(\alpha_1, \alpha_2, \alpha_3) = \alpha_1 P + \alpha_2 Q + \alpha_3 R$$

where

$$\alpha_1 + \alpha_2 + \alpha_3 = 1,$$

$$\alpha_i >= 0$$



The representation is called the **barycentric coordinate** representation of P

$$\alpha_1 = \frac{\Delta P}{\Delta PQR}, \quad \alpha_2 = \frac{\Delta Q}{\Delta PQR}, \quad \alpha_3 = \frac{\Delta R}{\Delta PQR}$$

Normal to a Plane

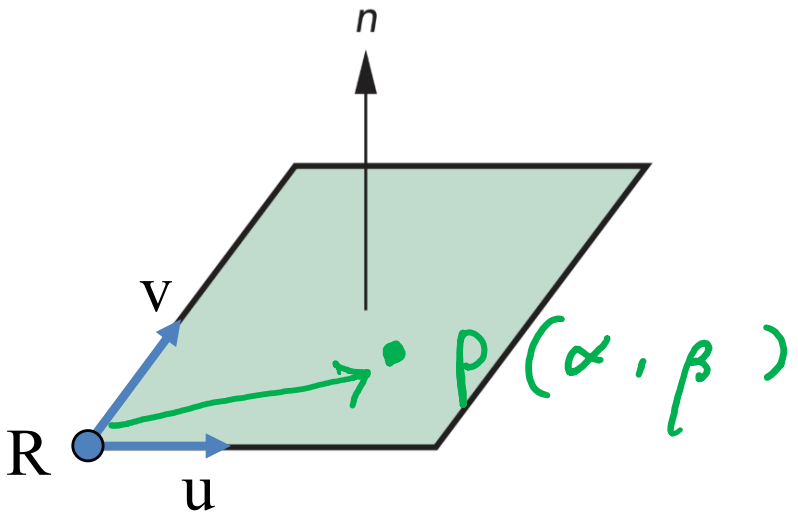
Every plane has a vector $\mathbf{n}$ perpendicular (or orthogonal) to it, and we called it the normal to the plane

parametric equation

From $\mathrm{P}(\alpha,\beta) = \mathrm{R}+\alpha\mathbf{u}+\beta\mathbf{v}$, we know we can use the cross product to find $\mathbf{n} = \mathbf{u} \times \mathbf{v}$ and the equivalent form

$$(\mathrm{P}(\alpha,\beta) - \mathrm{R}) \cdot \mathbf{n} = 0$$

implicit equation



COMP3271 Computer Graphics

Transformation

2019-20

Objectives

Introduce the three fundamental transformations

- Translation
- Scaling
- Rotation

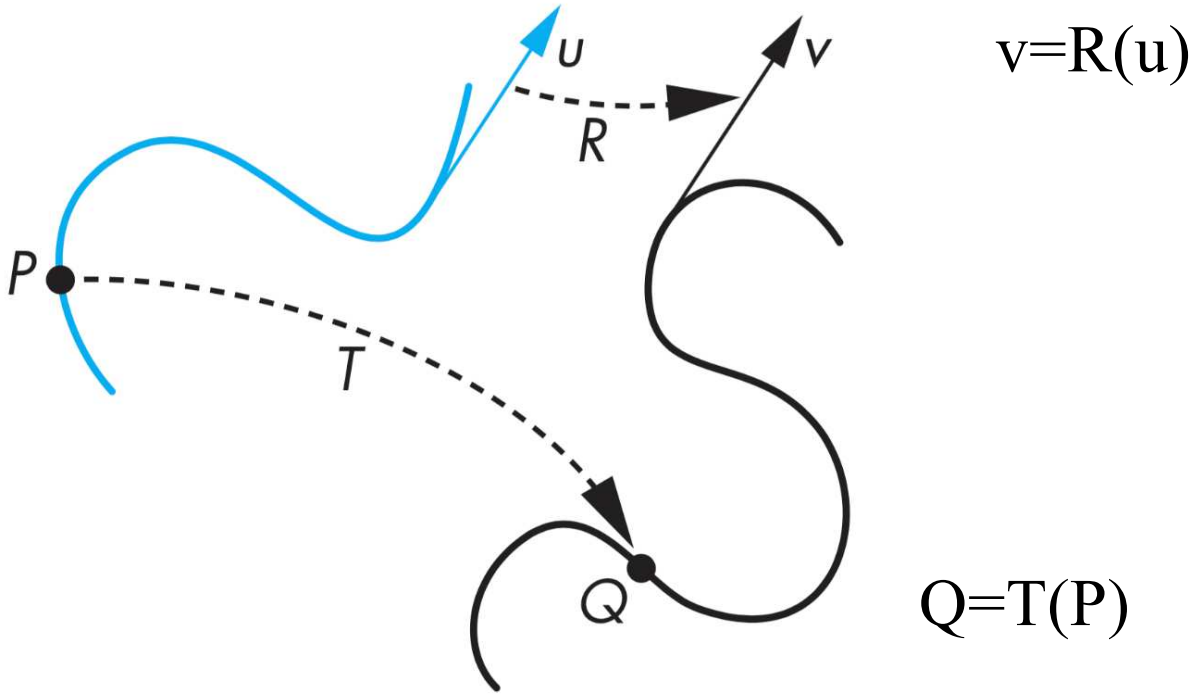
Derive homogeneous coordinate transformation matrices

Build arbitrary transformation from simple transformations

Materials adapted from textbook ppt and previous runs of the course

General Transformations

A transformation maps points to other points and/or vectors to other vectors



Provides a mechanism to manipulate objects

Why Do We Need Transformations?

Makes modeling more convenient

- for example, often easier to generate models around origin
  - gluSphere() draws a sphere of radius r about the origin
- then move them to final position with transformations

Model viewing process via transformations

- projecting 3-D to 2-D will be done this way

Animation

- transformations as a function of time creates motion

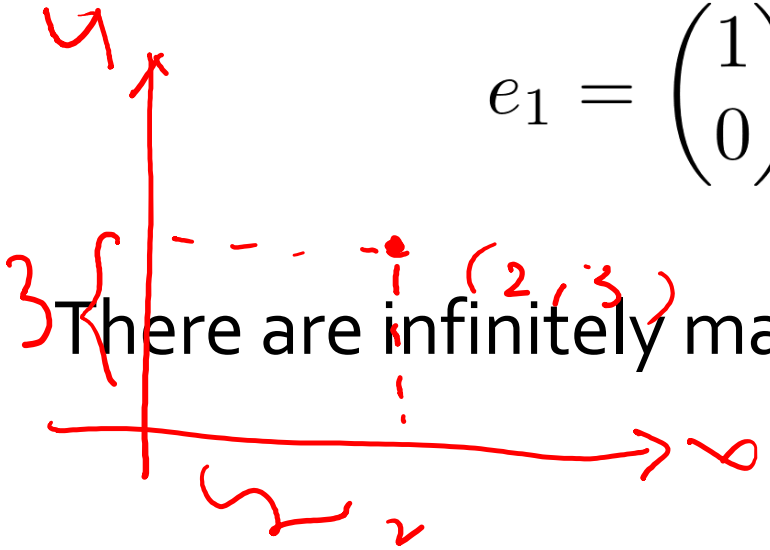
A demo: https://processing.org/examples/tree.html

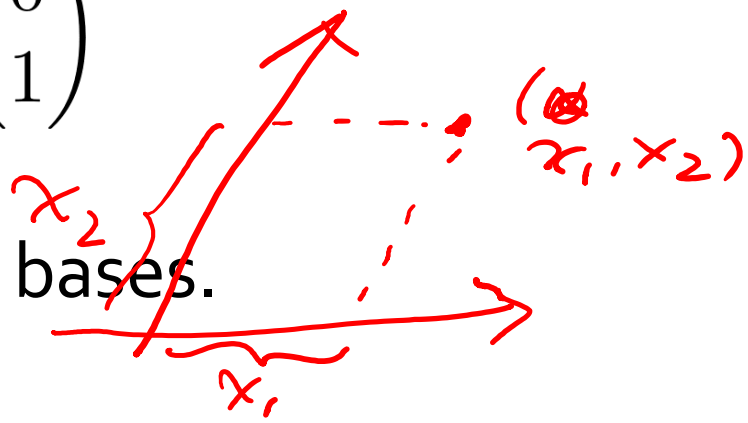
Linear Algebra (very quick review)

A linear combination of two vectors v and w is given by α v + β w, where α and are β scalars.

A **basis** for a space is a linearly independent set of vectors whose linear combinations include all vectors in the space, e.g., standard basis for 2-D plane:

$$e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$



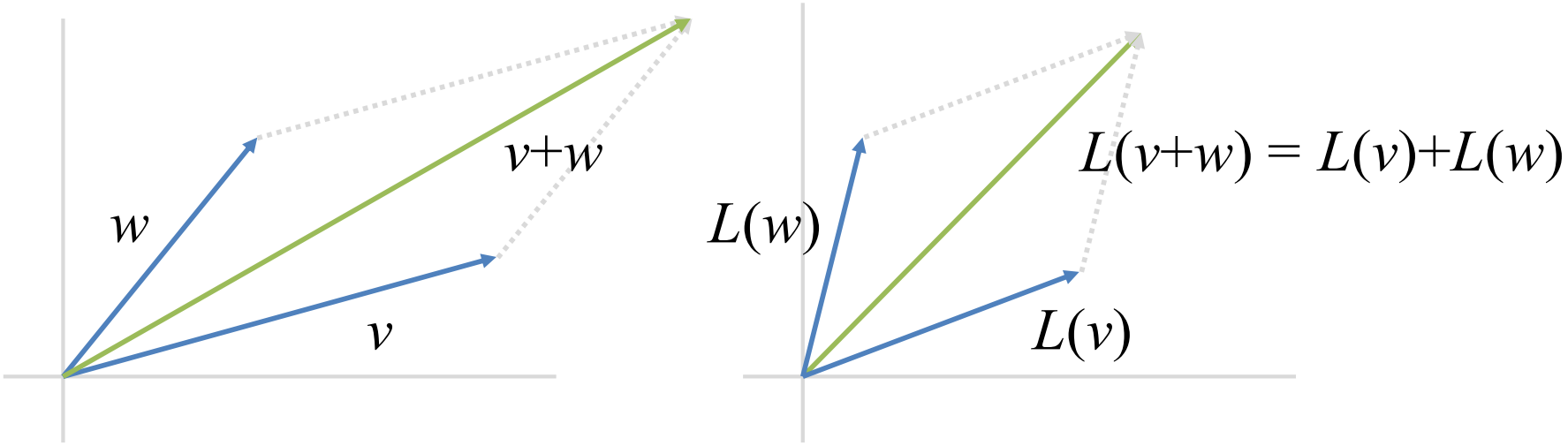


There are infinitely many possible bases.

Linear Transformation

A transformation (or mapping) L is linear when given any two vectors v and $w \in R^n$,

- $L(v + w) = L(v) + L(w)$
- $L(kv) = k\,L(v)$ for some scalar k



Linear Transformation

Considering the Cartesian coordinates, where a vector $v = (x, y)^{\mathrm{T}}$ is represented as a linear combination of the base vectors $e^1 = (1, 0)^{\mathrm{T}}$ and $e^2 = (0, 1)^{\mathrm{T}}$:

$$v = \begin{pmatrix} x \\ y \end{pmatrix} = x \begin{pmatrix} 1 \\ 0 \end{pmatrix} + y \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Applying a linear transformation to v:

$$L(\begin{pmatrix} x \\ y \end{pmatrix}) = L(x \begin{pmatrix} 1 \\ 0 \end{pmatrix} + y \begin{pmatrix} 0 \\ 1 \end{pmatrix}) = xL(\underbrace{\begin{pmatrix} 1 \\ 0 \end{pmatrix}}) + yL(\underbrace{\begin{pmatrix} 0 \\ 1 \end{pmatrix}})$$

Transformation of the base vectors

Linear Transformation

Linear transformations can be represented as **matrices**.

$$L\left(\begin{pmatrix} x \\ y \end{pmatrix}\right) = L\left(x\begin{pmatrix} 1 \\ 0 \end{pmatrix} + y\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) = xL\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}\right) + yL\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right)$$

$$= \underbrace{\left[L\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}\right) L\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right)\right]}_{\text{A 2x2 matrix}} \begin{pmatrix} x \\ y \end{pmatrix}$$

Affine Transformation

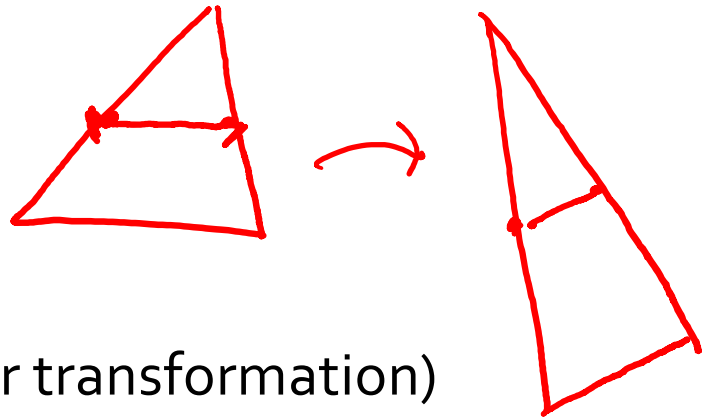
Affine transformation takes a more general form of

$$A(v) = Lv + b$$

where matrix L represents a non-singular linear transformation (i.e., $\det(L) \neq 0$) and b is a vector.

It can be viewed as a linear transformation plus a translation

Affine Transformations



Preserve geometric properties such as:

- Collinearity (lines remain lines under transformation)
- Parallelism
- Ratios of distances (e.g., mid-points remain mid-points)

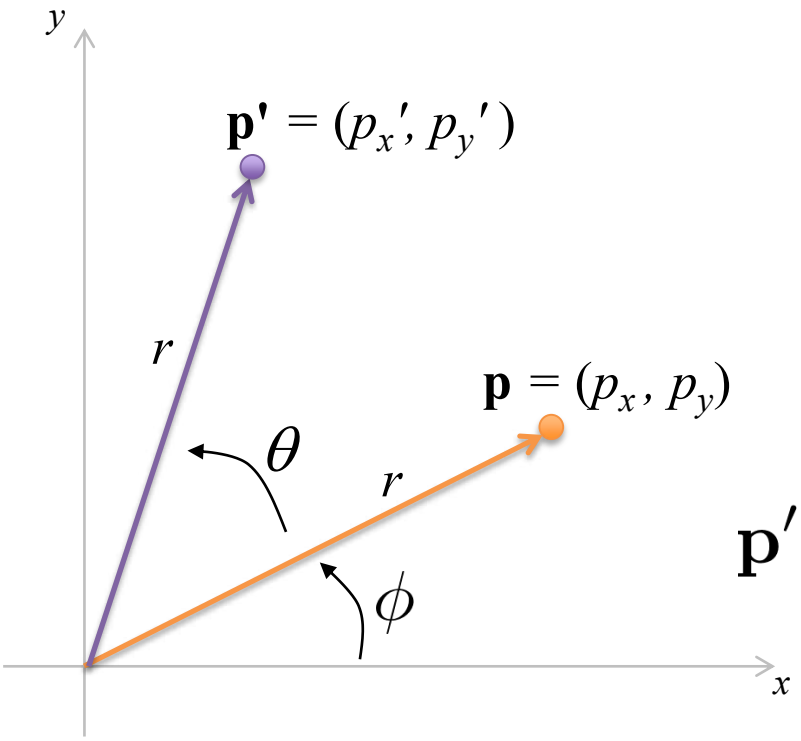
Characteristic of many physically important transformations

- Rigid body transformations: rotation, translation
- Scaling, shear

Importance in graphics: we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints

2D Rotation

$$p_x = r\cos\phi,\ p_y = r\sin\phi$$
$$p'_x = r\cos(\phi+\theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$p'_y = r\sin(\phi+\theta) = r\sin\phi\cos\theta + r\cos\phi\sin\theta$$



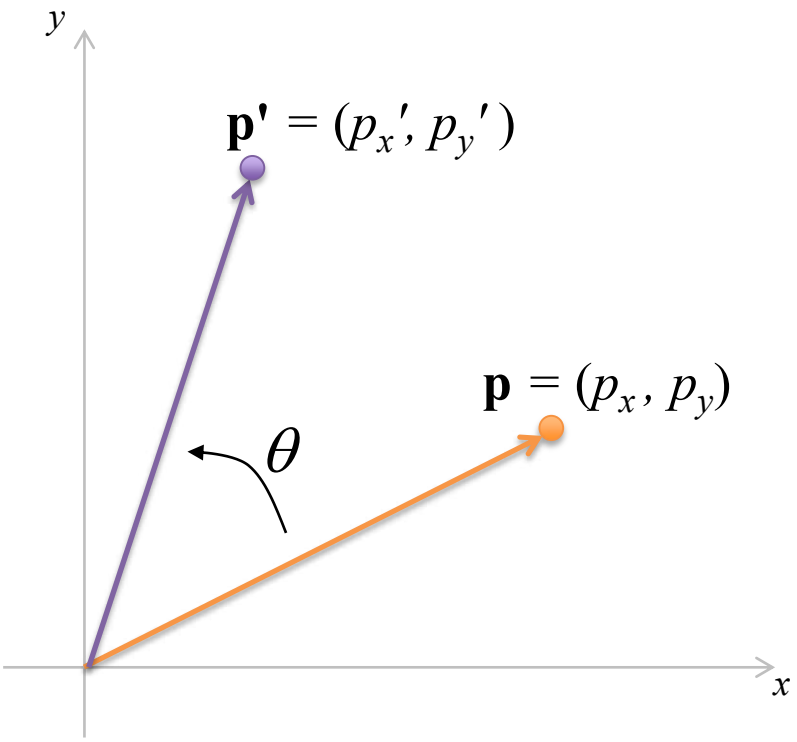
$$p'_x = p_x\cos\theta - p_y\sin\theta$$
$$p'_y = p_x\sin\theta + p_y\cos\theta$$

$$\mathbf{p}' = \begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} p_x\cos\theta - p_y\sin\theta \\ p_x\sin\theta + p_y\cos\theta \end{pmatrix}$$

2D Rotation

$$\mathbf{p}' = \begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} p_x \cos\theta - p_y \sin\theta \\ p_x \sin\theta + p_y \cos\theta \end{pmatrix}$$

$$= \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$



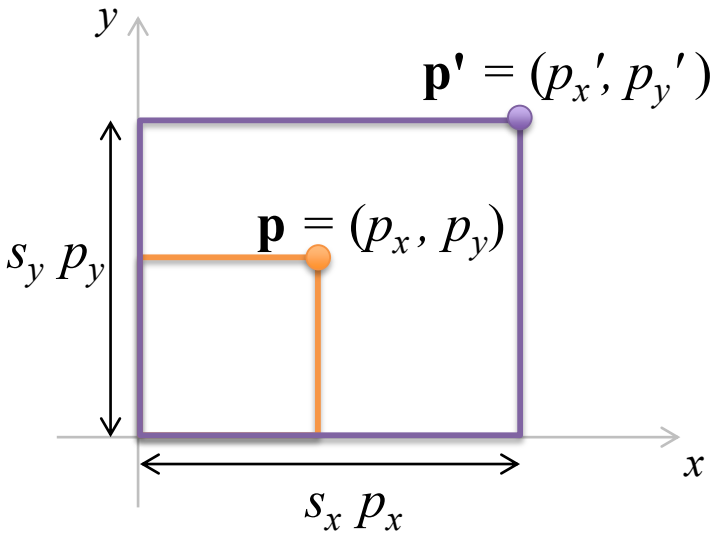
$$\mathbf{p}' = \mathbf{R}(\theta)\mathbf{p}$$

rotation

Note that the rotation is about the origin

2D Scaling

$$\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} s_x p_x \\ s_y p_y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$



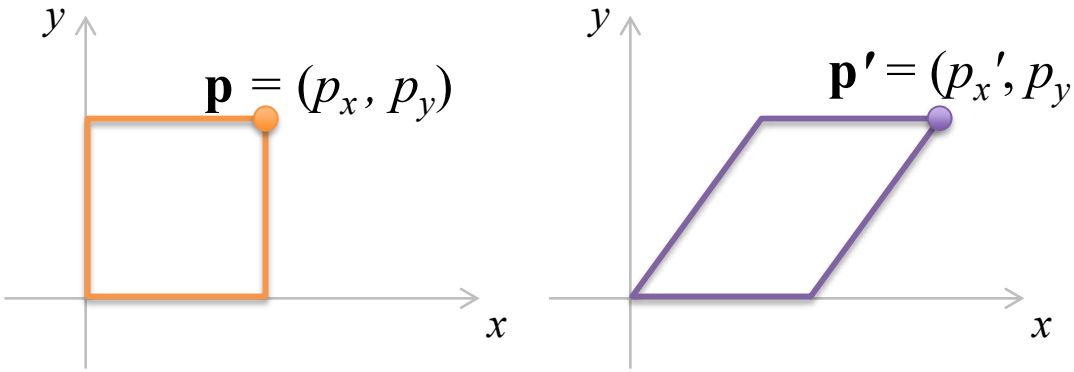
$$\mathbf{p}' = \mathbf{S}(s_x, s_y)\mathbf{p}$$

scaling

We have a uniform scaling if $s_x = s_y$

2D Shearing

X-Shear

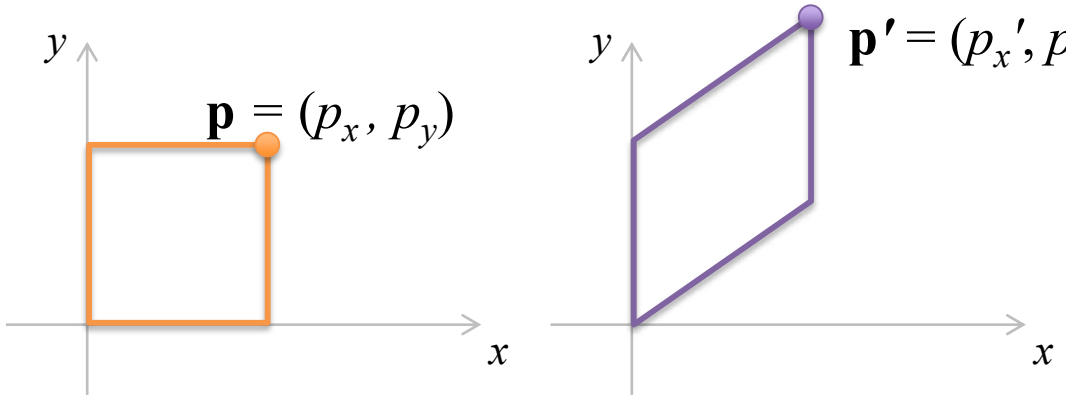


$$\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} p_x + mp_y \\ p_y \end{pmatrix} = \begin{pmatrix} 1 & m \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$

$$\mathbf{p}' = \mathbf{Sh}_x(m)\mathbf{p}$$

shearing

Y-Shear



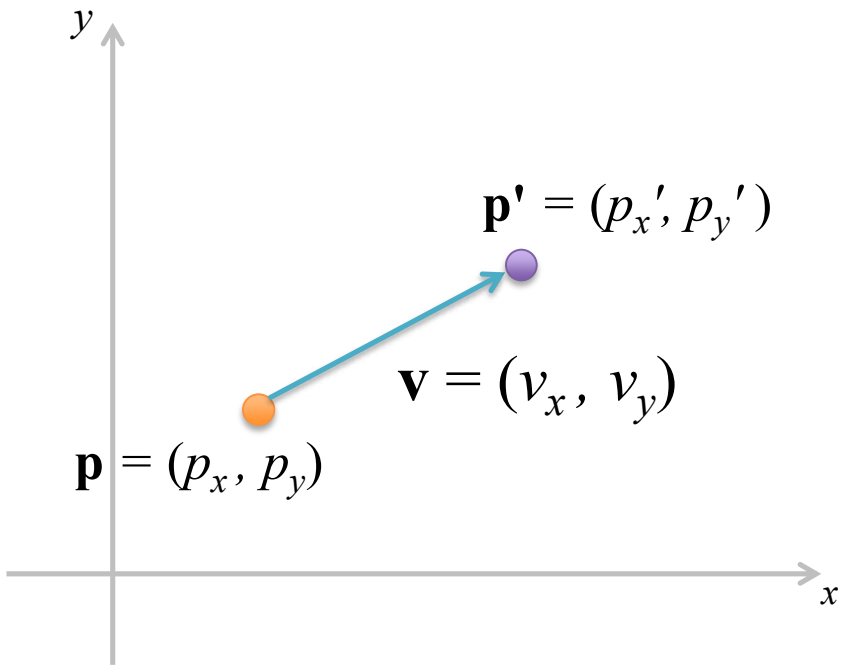
$$\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} p_x \\ mp_x + p_y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ m & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$

$$\mathbf{p}' = \mathbf{Sh}_y(m)\mathbf{p}$$

shearing

2D Translation

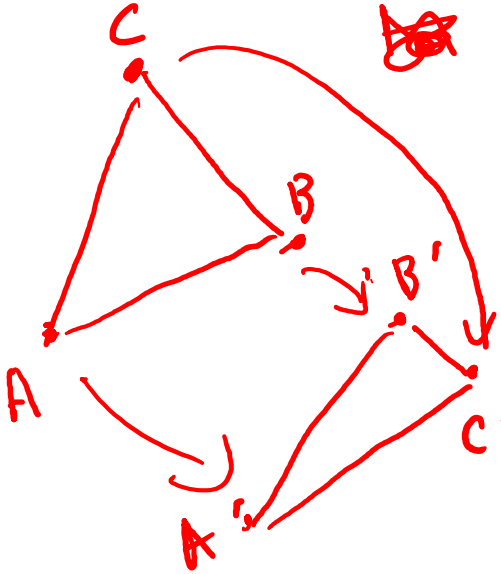
$$\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} p_x + v_x \\ p_y + v_y \end{pmatrix}$$



$$\mathbf{p}' = \mathbf{p} + \mathbf{v}$$

translation

The Basic Transformations



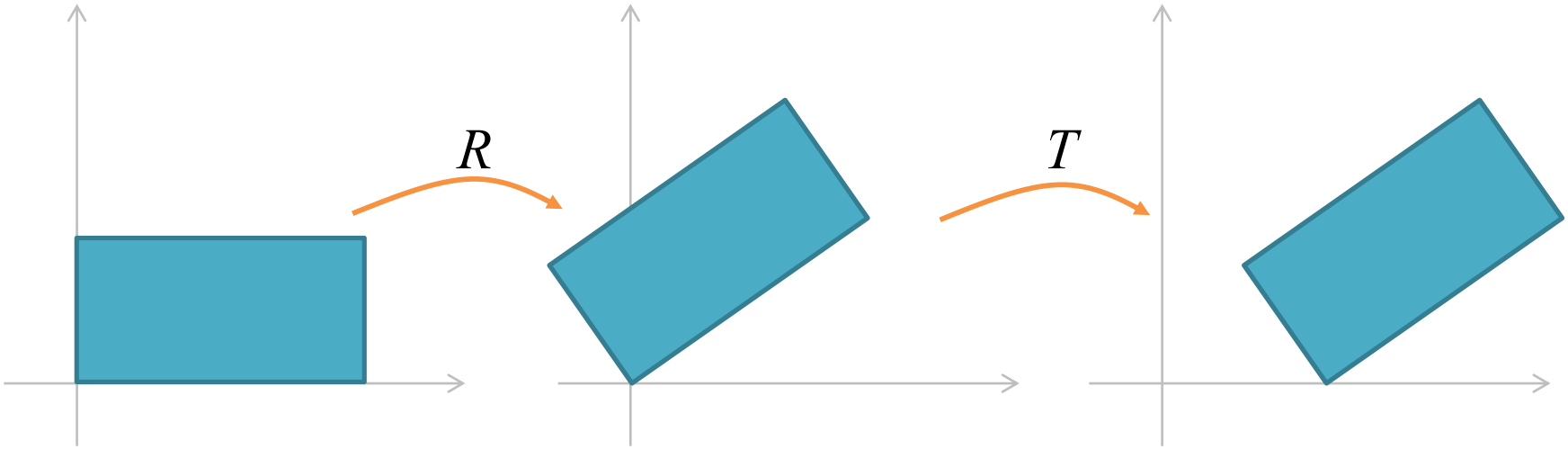
Rotation	$\mathbf{p}' = \mathbf{R}(\theta)\mathbf{p}$
Scaling	$\mathbf{p}' = \mathbf{S}(s_x, s_y)\mathbf{p}$
Shearing	$\mathbf{p}' = \mathbf{Sh}_x(m)\mathbf{p}$ $\quad \mathbf{p}' = \mathbf{Sh}_y(m)\mathbf{p}$
Translation	$\mathbf{p}' = \mathbf{p} + \mathbf{v}$

- can represent any affine transformation as a sequence of these 4
- A general rotation / scaling / shearing transformation has exactly one fixed point.
- A translation has no fixed point
- $\det(L) \neq 0$ is the scale factor of the area of a transformed region by the affine transformation
- A 2D affine transformation is uniquely determined by correspondences between three pairs of non-collinear points

Composition of Transformations

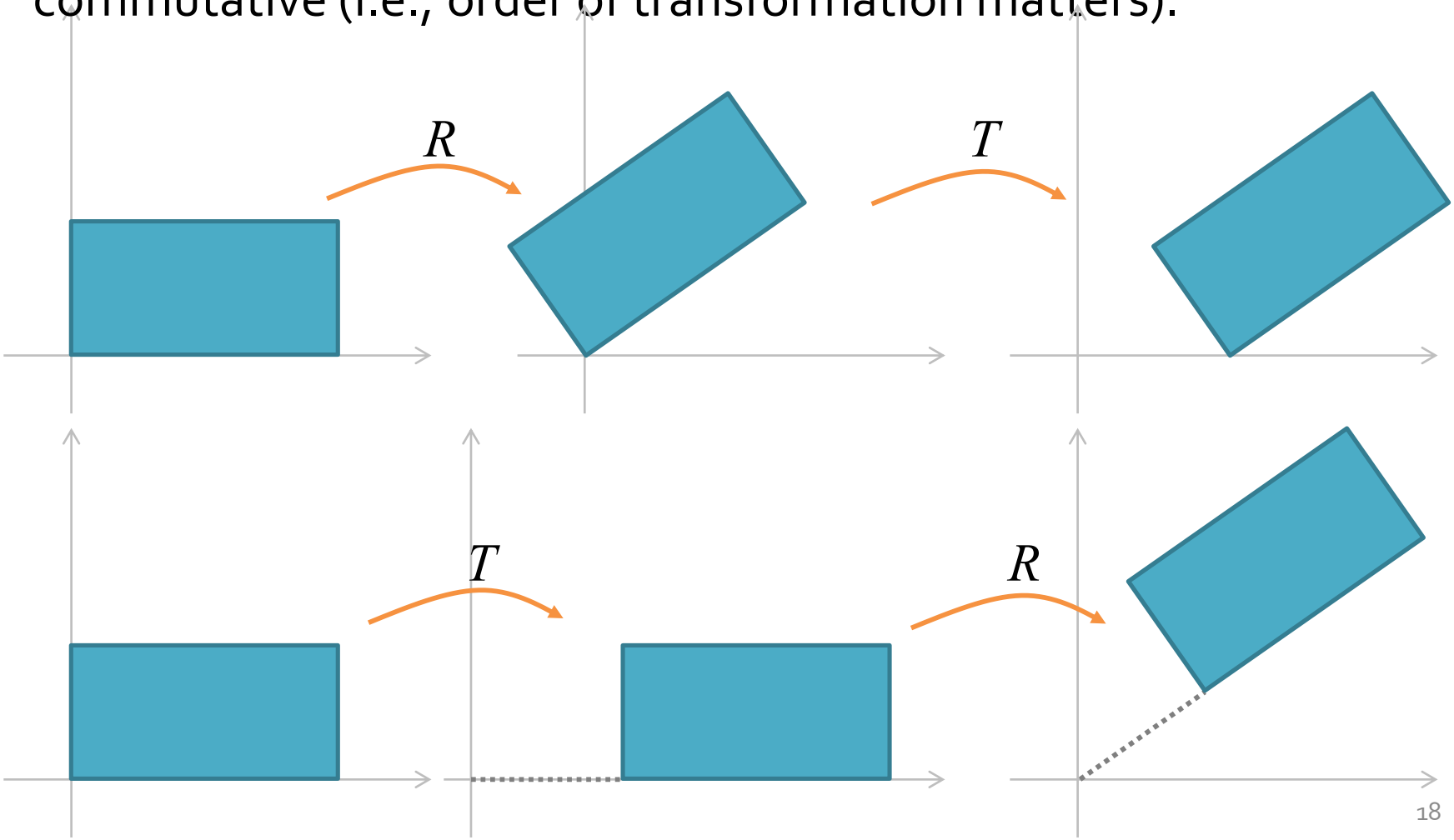
The composition of two affine transformation is also an affine transformation

Example: a rotation R followed by a translation T



Composition of Transformations

In general, the composition of affine transformations is non-commutative (i.e., order of transformation matters).



Matrix Compositions

$p' = R \cdot p$

Suppose we want to first rotate a point by $\mathbf{R}_1$, then scale by $\mathbf{S}$, then rotate again by $\mathbf{R}_2$, we have:

$$\mathbf{p}' = \mathbf{R}_2 \cdot \mathbf{S} \cdot \mathbf{R}_1 \, \mathbf{p} = \mathbf{M} \, \mathbf{p}$$

Transformation matrices are applied from right to left

Now, consider first rotate a point by $\mathbf{R}_1$, then translate by $\mathbf{v}$, then rotate again by $\mathbf{R}_2$, we have:

$$\mathbf{p}' = \mathbf{R}_2 \cdot (\mathbf{R}_1\mathbf{p} + \mathbf{v}) = \mathbf{R}_2 \cdot \mathbf{R}_1\mathbf{p} + \mathbf{R}_2\mathbf{v}$$

Oops! We don't have a nice matrix composition by multiplication for the transformation because translation is not a linear transformation

Homogeneous Coordinates

A 2D point (x, y) is represented as (wx, wy, w), for any real number $w \neq 0$.

Therefore, any given point has infinitely many different homogeneous coordinate representations.

- Two points (wx, wy, w) and (ux, uy, u) in homogeneous coordinates are the same
- E.g.: the 2D points (4, 8, 2) and (20, 40, 10) are the same.

(2, 4) (2, 4)

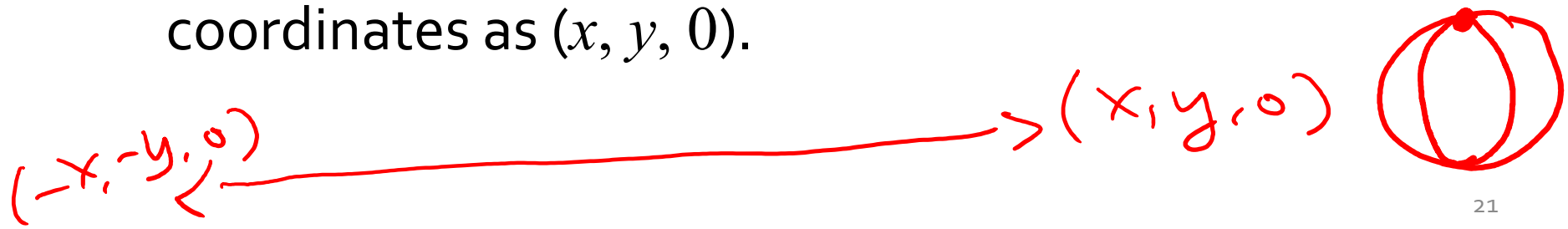
Conversely, given any homogeneous coordinates (x, y, w), $w \neq 0$, of a 2D point, the Cartesian coordinates of the point are given by $(x/w, y/w)$.

$$(x, y, w) \leftrightarrow \left(\frac{x}{w}, \frac{y}{w}\right)$$

Homogeneous Coordinates

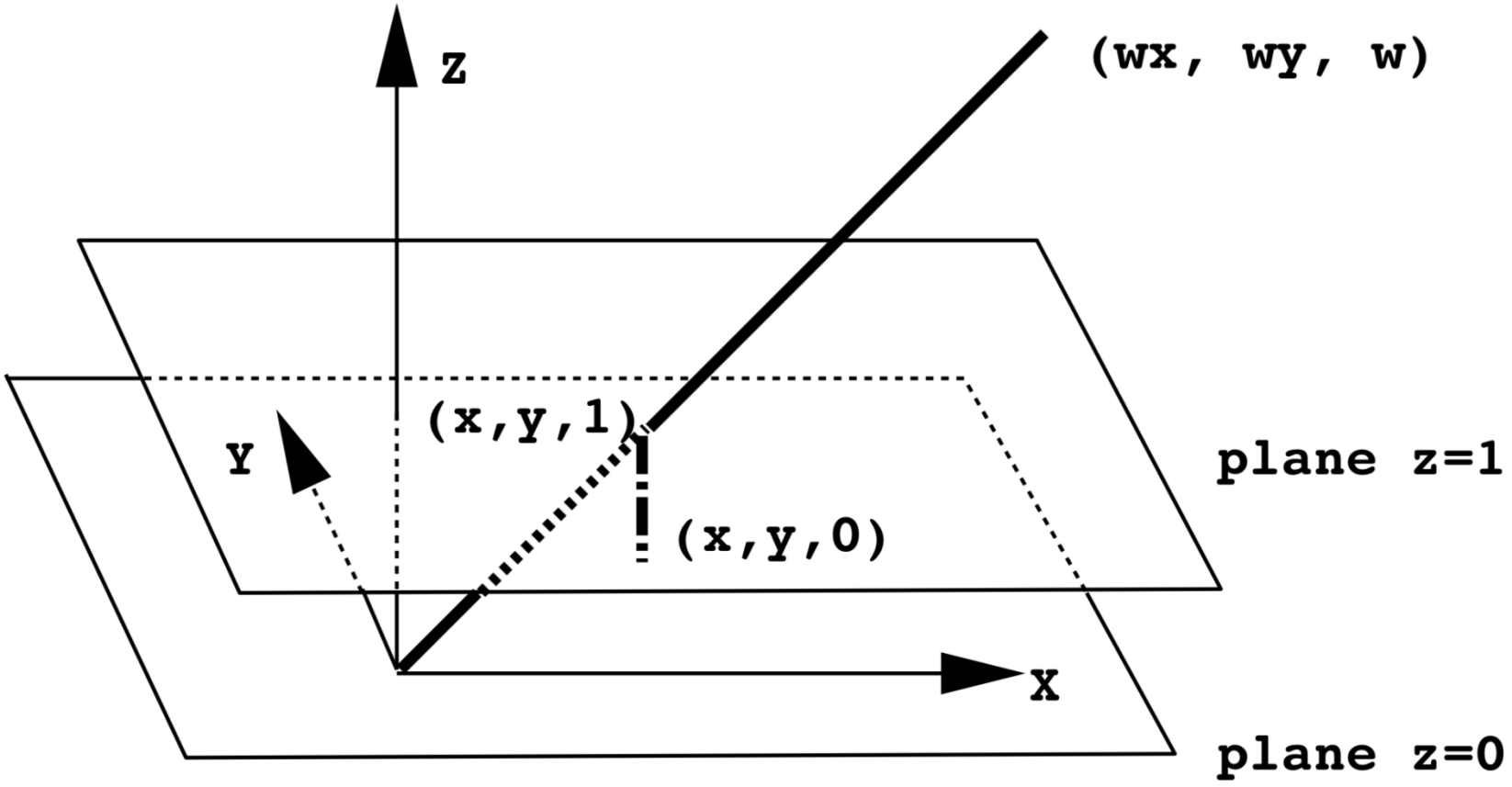
Homogeneous coordinates allow us to define points at infinity.

- $(0, 0, 0)$ does not represent a well-defined point
- When $x \neq 0$ or $y \neq 0$, $(x, y, 0)$ are the homogeneous coordinates of a point at infinity; in particular, $(x, y, 0)$ and $(-x, -y, 0)$ stand for the same point at infinity.
- A point at infinity indicates a direction and thus a 2D vector can be represented in homogeneous coordinates as $(x, y, 0)$.



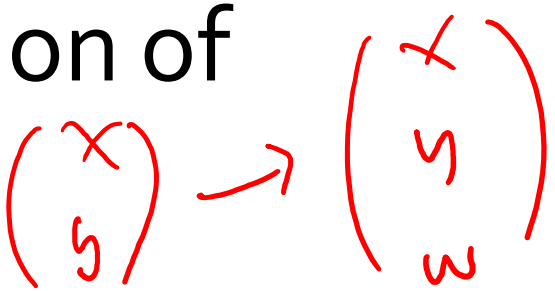
Homogeneous Coordinates

A geometric interpretation



Reading: A.4 of online appendices: "Linear Algebra & Trigonometry" from the book " Real-Time Rendering"

Homogeneous Representation of 2D Transformations



We represent 2D transformations using 3x3 matrices

Rotation

$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} p_x \cos\theta - p_y \sin\theta \\ p_x \sin\theta + p_y \cos\theta \\ 1 \end{pmatrix}$$

This is the same point as in slide 12

Homogeneous Representation of 2D Transformations

Scaling $\mathbf{p}' = \mathbf{S}\mathbf{p}$

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} = \begin{pmatrix} s_x p_x \\ s_y p_y \\ 1 \end{pmatrix}$$

Translation $\mathbf{p}' = \mathbf{T}\mathbf{p}$

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + v_x \\ p_y + v_y \\ 1 \end{pmatrix}$$

Now a translation can be represented as a matrix as well

Vector Transformations

Note that translation has no effect on vectors.

Translating a vector:

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ 0 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ 0 \end{pmatrix}$$

Translation has no effect

Composition of Transformations

With homogeneous transformations, composition of transformations can be represented by multiplication of matrices

Rotate, scale then rotate again

$$\mathbf{p}' = \underline{\mathbf{R}_2} \cdot \underline{\mathbf{S}} \cdot \underline{\mathbf{R}_1}\,\mathbf{p} = \underline{\mathbf{M}}\,\mathbf{p}$$

3x3 matrices for 2D transformations

Rotate, translate then rotate again

$$\mathbf{p}' = \mathbf{R}_2 \cdot \mathbf{T} \cdot \mathbf{R}_1\,\mathbf{p} = \mathbf{N}\,\mathbf{p}$$

3D Transformations

We represent 3D transformations using 4x4 matrices:

Rotation about x-axis

$$\mathbf{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about y-axis

$$\mathbf{R}_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about z-axis

$$\mathbf{R}_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fixed point of a 3D rotation is a straight line

Scaling

$$\mathbf{S} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Inverse Transformations

Suppose we apply a transformation **M** to a point **p**:

$$\mathbf{p}' = \mathbf{M}\mathbf{p}$$

What is the transformation needed to bring **p'** back to **p**?

$$\mathbf{p} =? \mathbf{p}'$$

We need the inverse of **M** to bring **p'** back to **p**:

$$\mathbf{M}^{-1}\mathbf{p}' = (\mathbf{M}^{-1}\mathbf{M})\mathbf{p} = \mathbf{I}\,\mathbf{p} = \mathbf{p}$$

Inverse of M

Identity transformation

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

A 3x3 identity matrix

Inverse Transformations

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

inverse of **T**?

$$\mathbf{T}^{-1} = \begin{pmatrix} 1 & 0 & 0 & -v_x \\ 0 & 1 & 0 & -v_y \\ 0 & 0 & 1 & -v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

inverse of **S**?

$$\mathbf{S}^{-1} = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

inverse of $\mathbf{R}_z$?

$$\mathbf{R}_z^{-1} = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \mathbf{R}_z^T$$

Inverse of a rotation equals its transpose

Inverse of Transformation

Suppose we have a transformation **M** given by a composition of transformations:

$$\mathbf{M} = \mathbf{R} \cdot \mathbf{S} \cdot \mathbf{T}$$

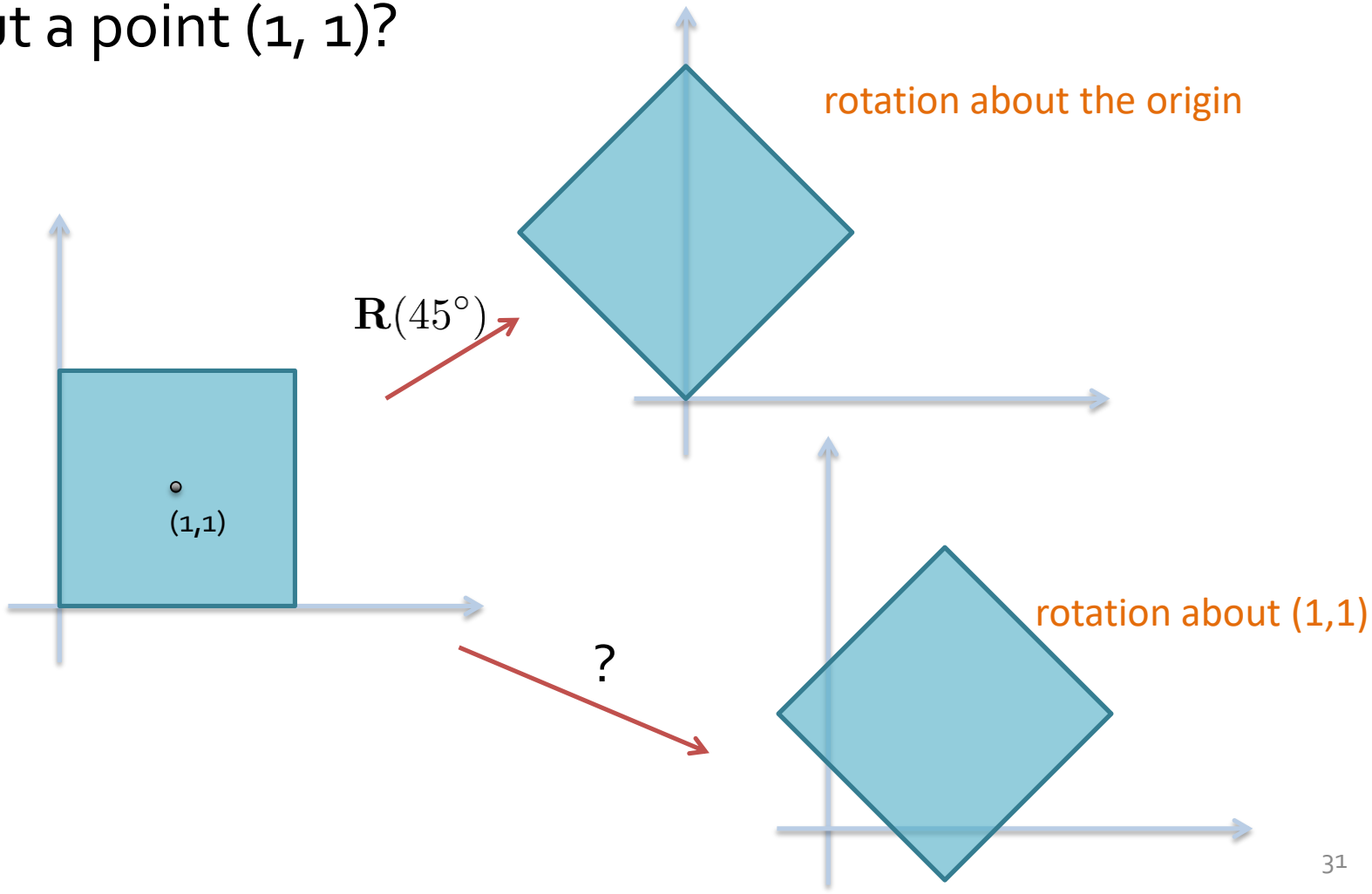
- How to find the inverse of M?

$$\begin{aligned}\mathbf{M}^{-1} &= (\mathbf{R} \cdot \mathbf{S} \cdot \mathbf{T})^{-1} \\ &= \mathbf{T}^{-1} \cdot \mathbf{S}^{-1} \cdot \mathbf{R}^{-1}\end{aligned}$$

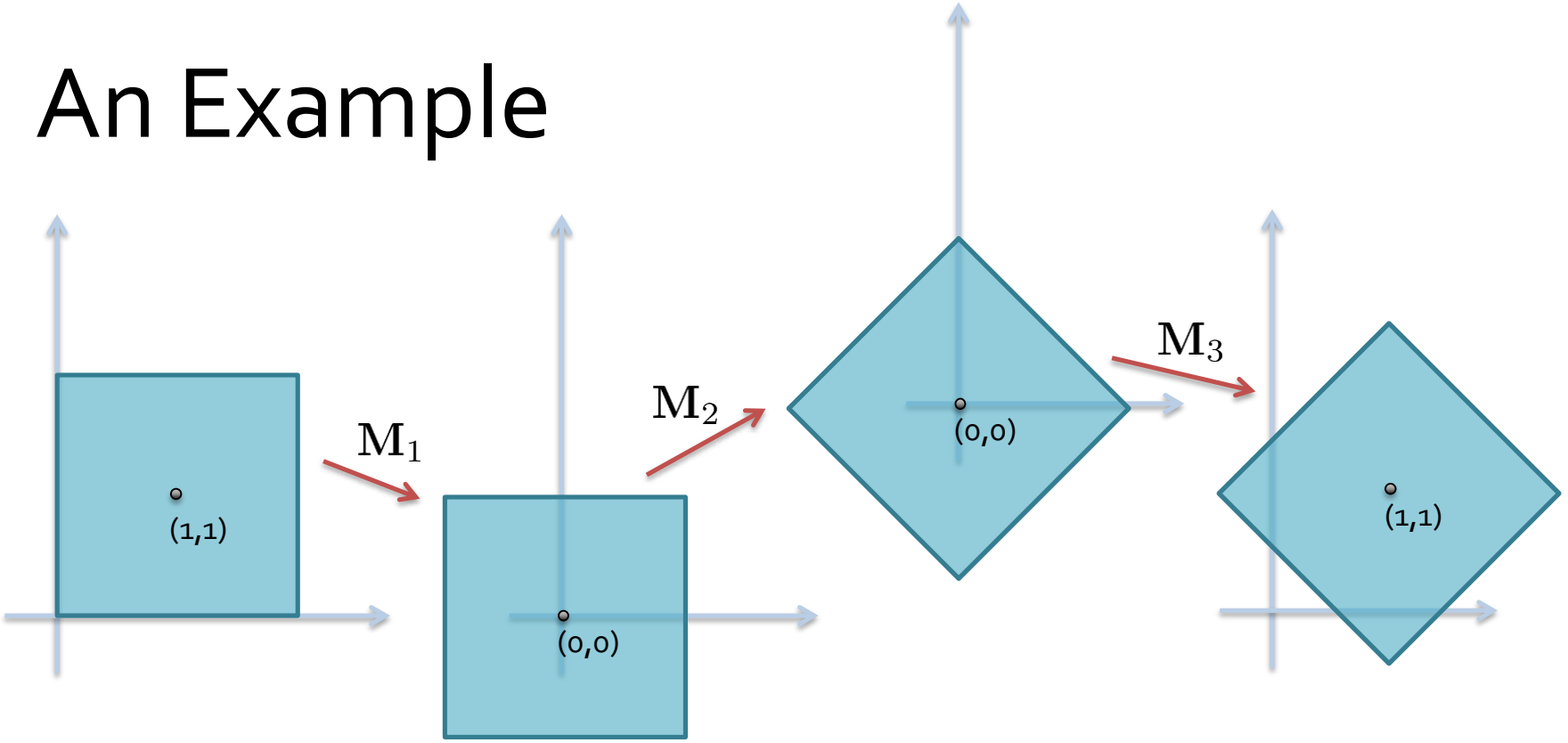
$$\begin{aligned}\mathbf{M}^{-1} \cdot \mathbf{M} &= (\mathbf{T}^{-1} \cdot \mathbf{S}^{-1} \cdot \mathbf{R}^{-1}) \cdot (\mathbf{R} \cdot \mathbf{S} \cdot \mathbf{T}) \\ &= \mathbf{T}^{-1} \cdot \mathbf{S}^{-1} \cdot (\mathbf{R}^{-1} \cdot \mathbf{R}) \cdot \mathbf{S} \cdot \mathbf{T} \\ &= \mathbf{T}^{-1} \cdot \mathbf{S}^{-1} \cdot \mathbf{I} \cdot \mathbf{S} \cdot \mathbf{T} \\ &= \cdots = \mathbf{I}\end{aligned}$$

An Example

What is the matrix representing a 2D rotation of 45° about a point (1, 1)?



An Example



translate by (1,1)

Rotate 45°

translate by (-1,-1)

$$\mathbf{M} = \mathbf{M}_3 \cdot \mathbf{M}_2 \cdot \mathbf{M}_1$$
$$= \mathbf{T}(1,1) \cdot \mathbf{R}(45°) \cdot \mathbf{T}(-1,-1)$$

An Example

$$\mathbf{M} = \mathbf{T}(1,1) \cdot \mathbf{R}(45^\circ) \cdot \mathbf{T}(-1,-1)$$

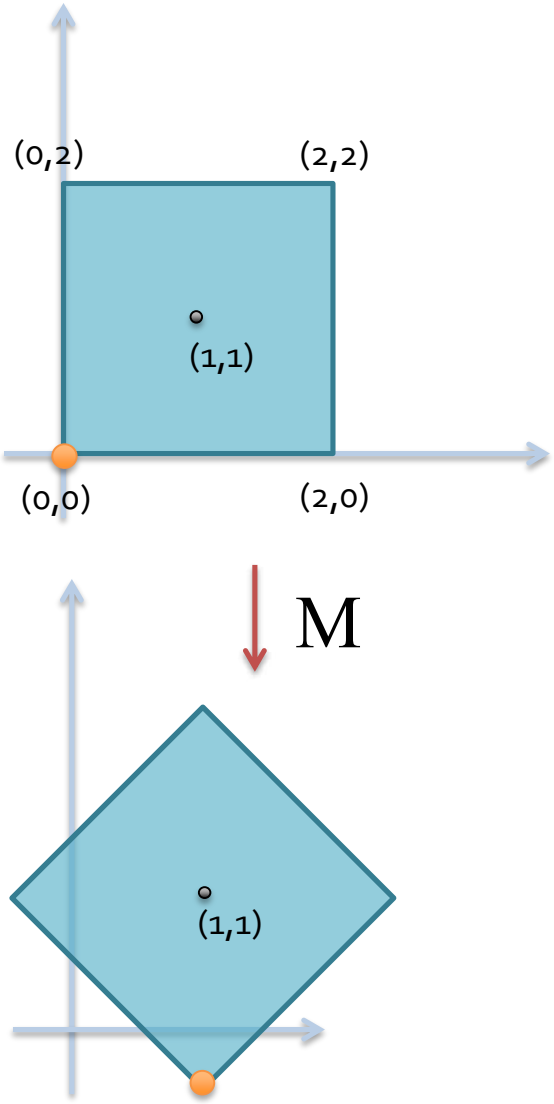
$$= \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos 45^\circ & -\sin 45^\circ & 0 \\ \sin 45^\circ & \cos 45^\circ & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos 45^\circ & -\sin 45^\circ & -\cos 45^\circ + \sin 45^\circ \\ \sin 45^\circ & \cos 45^\circ & -\sin 45^\circ - \cos 45^\circ \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos 45^\circ & -\sin 45^\circ & -\cos 45^\circ + \sin 45^\circ + 1 \\ \sin 45^\circ & \cos 45^\circ & -\sin 45^\circ - \cos 45^\circ + 1 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 1 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1 - \sqrt{2} \\ 0 & 0 & 1 \end{pmatrix}$$

Where will the four vertices of the square be transformed to?



Let's try the point $p = (0,0)^T$

$$p' = \mathbf{M}p$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 1 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1-\sqrt{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 \\ 1-\sqrt{2} \\ 1 \end{pmatrix}$$

COMP3271 Computer Graphics

Orientation Representation

2019-20

Objectives

Focus on the rotation transformation

Four orientation formats

- Rotation matrices
- Euler angles
- Axis-angle representation
- Quaternions

Comparisons of these representations

Criteria for Orientation Formats

How much storage is needed for the representation?

- How many numbers are needed to represent an orientation/rotation?

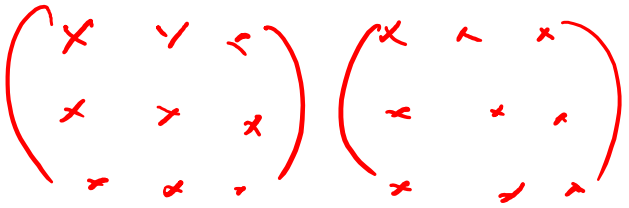
How efficient to form new orientations? $R_2 \cdot R_1$

How efficient to rotate points and vectors? $R \cdot v$

How well the representation can be interpolated?

How suitable for numeric integration (e.g. for physical simulation)?

Rotation Matrices



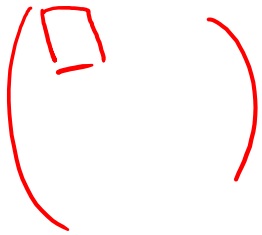
$$R = \begin{pmatrix} u_0 & v_0 & w_0 \\ u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \end{pmatrix}$$

The column vectors

$u = (u_0, u_1, u_2)^T, v = (v_0, v_1, v_2)^T,$

$w = (w_0, w_1, w_2)^T \in \mathbb{R}^3$

are three orthonormal basis vectors.



27 M

Nine numbers needed for a rotation

- Euler's rotation theorem states that we just need three numbers to represent a rotation

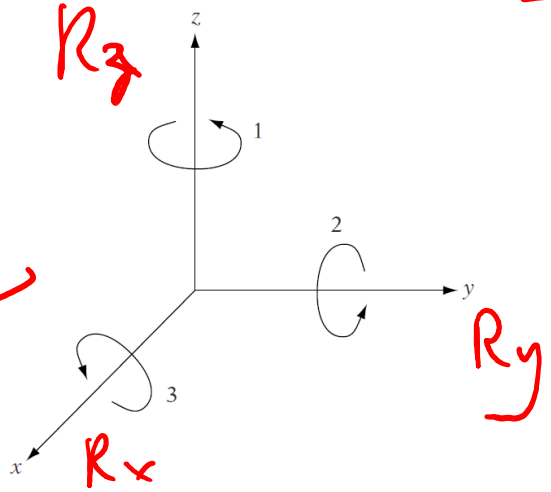
$R_2 \cdot R_1$

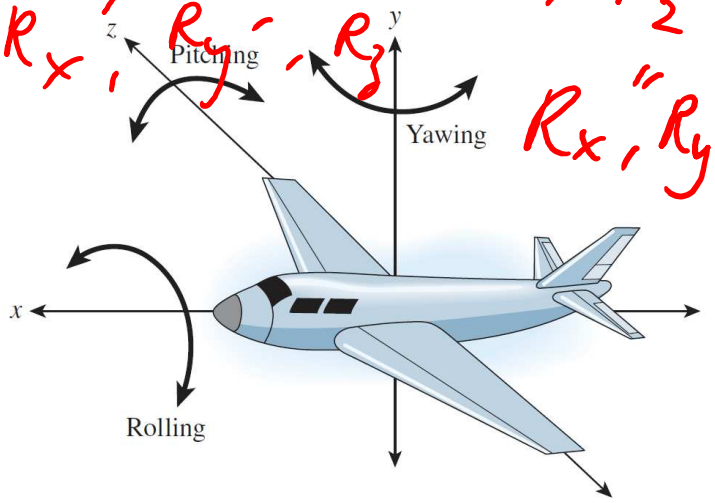
New rotations are obtained by matrix-matrix multiplication; vectors are rotated by matrix-vector multiplication

9 M → $R \cdot v$

- Can be performed quite efficiently, some hardware has built-in circuitry for the multiplications

Euler Angles





Use 3 sequential rotations about a set of orthogonal axes to specify an orientation.

- If axes are fixed, need only 3 numbers for the angles (the Euler angles)
- If we choose the standard x-,y-,z-axes, the rotations are given by R_x, R_y, R_z
- No standard order for the use of the three axes

Composition of rotations and vector rotations resort to converting back to matrix representation and therefore are not efficient

Axis-Angle Representation

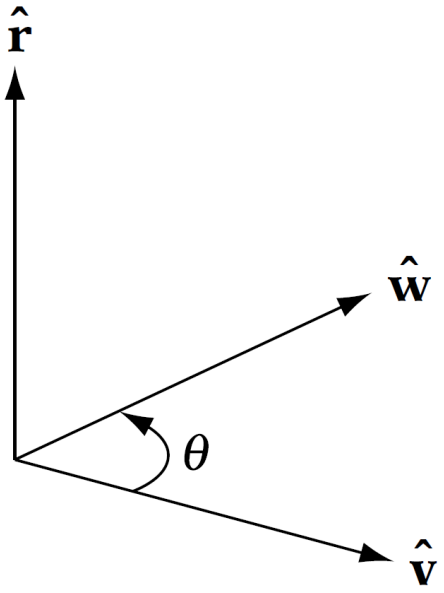
Represent a rotation by an axis of rotation $\mathbf{r}$, and the angle of rotation θ about this axis

- $\mathbf{r}$ is normalized so the degree of freedom is 3

The axis-angle rotation to bring a vector $\mathbf{v}$ to another vector $\mathbf{w}$ is given by

$$\mathbf{r} = \hat{\mathbf{v}} \times \hat{\mathbf{w}}$$
$$\theta = \arccos(\hat{\mathbf{v}} \cdot \hat{\mathbf{w}})$$

Composition of rotations and vector rotations are not trivial.



Quaternions

Mathematical object developed by Sir William Rowan Hamilton in 1843 as an extension to the complex numbers

General form of a quaternion:

$$\mathbf{q} = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

where $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ are "complex" numbers such that $\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1$

A **quaternion** can therefore be represented as a 4-dimensional vector

$$\mathbf{q} = (w, x, y, z)$$

Quaternions

$x + y\,i$

The $x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$ part is similar to a 3D vector, so we may express a quaternion as

$\mathbb{R}$ $\mathbb{R}^3$

$$\mathbf{q} = (w,\ \mathbf{v})$$

scalar 3D vector

A vector is represented as a quaternion by setting the scalar part 0:

$$\mathbf{q}_\mathbf{u} = (0,\ \mathbf{u})$$

Quaternion Normalization

Magnitude:

$$\|\mathbf{q}\| = \sqrt{(w^2 + x^2 + y^2 + z^2)}$$

Normalization:

$$\hat{\mathbf{q}} = \frac{\mathbf{q}}{\|\mathbf{q}\|}$$

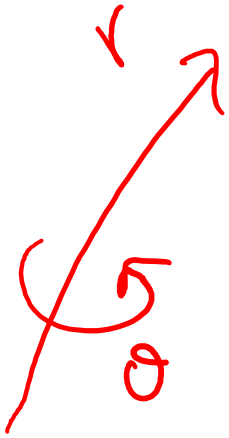
Unit Quaternions as Rotations

A unit quaternion is a quaternion $\mathbf{q} = (w, \mathbf{v})$ such that

$$w^2 + \mathbf{v} \cdot \mathbf{v} = 1$$

$\mathbf{q}$ can also be written as

$$\mathbf{q} = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\mathbf{r})$$



$\mathbf{r}$ is a unit vector representing the axis of rotation

θ is the angle of rotation

Example

$r = (r_x, r_y, r_z) = (0, 0, 1)$

$\theta = \frac{\pi}{2}$

What is the quaternion representing a rotation about the z-axis by 90 degrees?

$$w = \cos\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$$

$$x = 0 \cdot \sin\left(\frac{\pi}{4}\right) = 0 \quad (r_x)$$

$$y = 0 \cdot \sin\left(\frac{\pi}{4}\right) = 0 \quad (r_y)$$

$$z = 1 \cdot \sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \quad (r_z)$$

$$\mathbf{q} = \left(\frac{\sqrt{2}}{2}, 0, 0, \frac{\sqrt{2}}{2}\right)$$

Quaternion Operations

For addition and scalar multiplication, a quaternion behaves like a 4-vector:

$$(w_1, x_1, y_1, z_1) + (w_2, x_2, y_2, z_2)$$
$$= (w_1 + w_2, x_1 + x_2, y_1 + y_2, z_1 + z_2)$$

$$a(w, x, y, z) = (aw, ax, ay, az)$$

Given a quaternion $\mathbf{q}$, what is $-\mathbf{q}$?

(w, x, y, z) $(-w, -x, -y, -z)$

Quaternion Negation

$\cos(\pi+\phi) = -\cos\phi$

$\sin(\pi+\phi) = -\sin\phi$

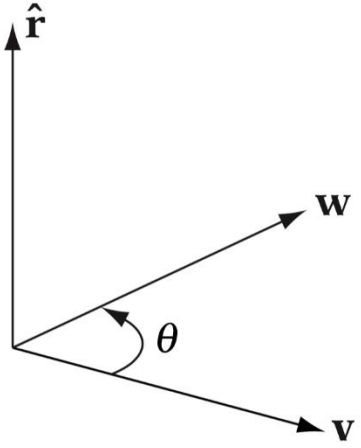
$$\mathbf{q} = (\cos\tfrac{\theta}{2}, \sin\tfrac{\theta}{2}\mathbf{r})$$

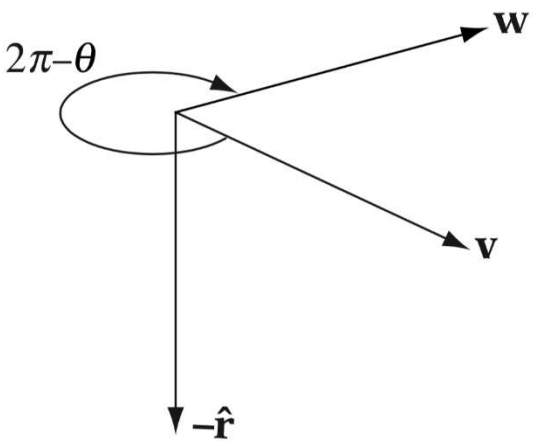
$$-\mathbf{q} = (-\cos\tfrac{\theta}{2}, -\sin\tfrac{\theta}{2}\mathbf{r})$$

$$= (\cos\tfrac{2\pi+\theta}{2}, \sin\tfrac{2\pi+\theta}{2}\mathbf{r}) \quad (1)$$

$$= (\cos\tfrac{2\pi-\theta}{2}, \sin\tfrac{2\pi-\theta}{2}(-\mathbf{r})) \quad (2)$$

q





q & -q represent the same orientation

Quaternion Composition

Let $\mathbf{q}_1$ and $\mathbf{q}_2$ be two unit quaternions representing two rotations.

$$\mathbf{q}_1 = (w_1, \mathbf{v}_1) \qquad \mathbf{q}_2 = (w_2, \mathbf{v}_2)$$

The composition of first a rotation by $\mathbf{q}_1$ and then a rotation by $\mathbf{q}_2$ is given by the multiplication of $\mathbf{q}_2$ and $\mathbf{q}_1$:

$$\mathbf{q}_2\mathbf{q}_1 = (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2,\ w_1\mathbf{v}_2 + w_2\mathbf{v}_1 + \mathbf{v}_2 \times \mathbf{v}_1)$$

Order matters!

Vector dot product

Vector cross product

Compositing two rotations using quaternions take 16 multiplications and 12 additio

Quaternion Inverse

negation: $-q$

The inverse of a quaternion $\mathbf{q}$ is denoted by $\mathbf{q}^{-1}$, such that

$$\mathbf{q}\mathbf{q}^{-1} = (1, 0, 0, 0)$$

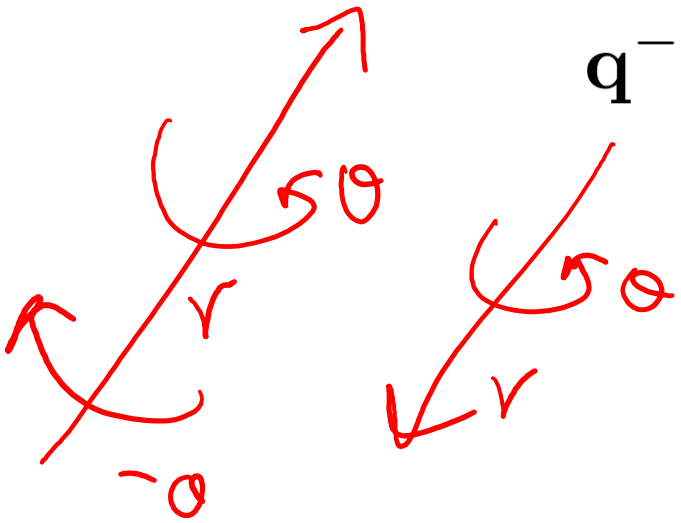
Identity quaternion, also representing zero rotation

$q_2 \cdot q_1 = (\ldots - \ldots)$

Given $\mathbf{q} = (w, \mathbf{v})$, what is $\mathbf{q}^{-1}$?

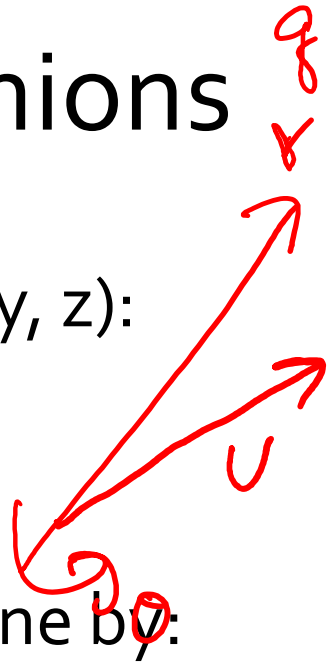
$$\mathbf{q}^{-1} = (w, -\mathbf{v})$$

Negating the axis of rotation



Inverting a quaternion is fast!

Rotating Vectors with Quaternions



Let v be a quaternion representing a vector (x, y, z):

$$\mathbf{v} = (0, x, y, z)$$

Rotating a vector **v** by a unit quaternion **q** is done by:

$$\mathbf{v}' = \mathbf{q}\mathbf{v}\mathbf{q}^{-1}$$

Further apply a rotation by a unit quaternion **p**:

$$\mathbf{v}'' = \mathbf{p}\mathbf{q}\mathbf{v}\mathbf{q}^{-1}\mathbf{p}^{-1} = \mathbf{p}\mathbf{q}\mathbf{v}(\mathbf{p}\mathbf{q})^{-1}$$

$\mathbf{pq}$ is the composite rotation

COMP3271 Computer Graphics

Viewing

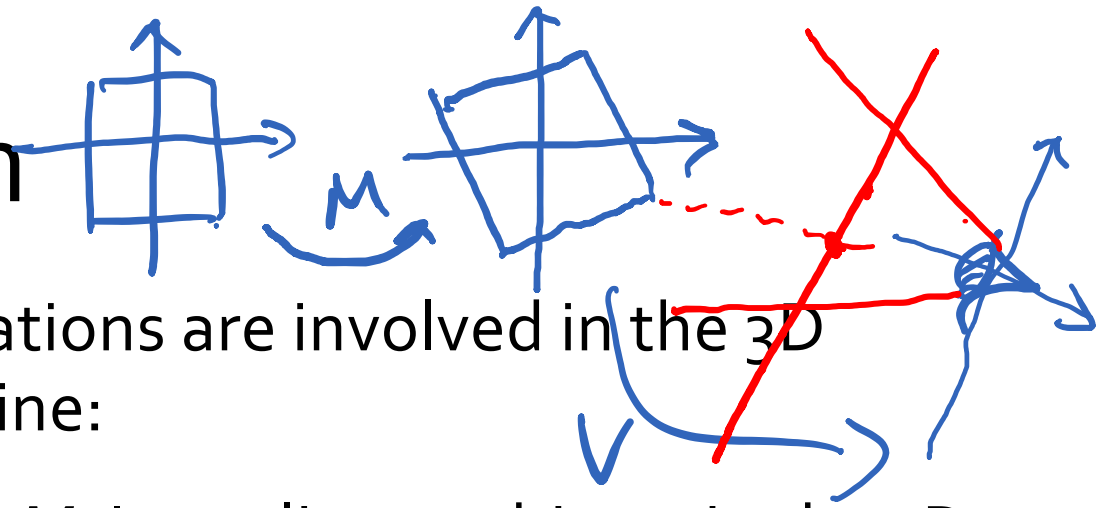
2019-20

Objectives

Understand the viewing process

Derive the projection matrices used for standard OpenGL projections

Transformation



Three kinds of transformations are involved in the 3D graphics processing pipeline:

- model transformation *M*: It applies to objects in the 3D world coordinate system (the object space);
- view transformation *V*: It maps objects from the 3D world coordinate system to the 3D eye coordinate system, with the origin at the eye-point (viewpoint);
- view projection *P*: It maps objects from the 3D eye-coordinate system to the 2D view plane.

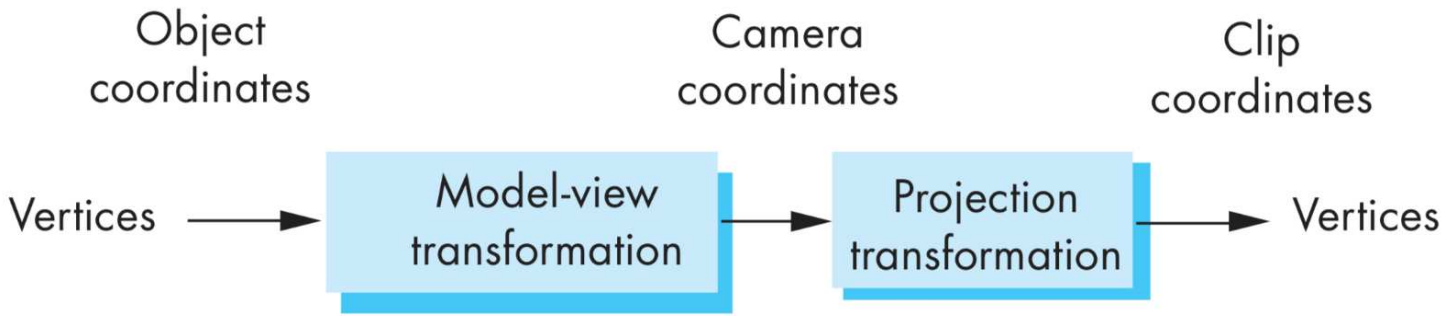
A vertex will be transformed by the concatenation of these transformations before appearing on screen

$$X_3' = P_{3\times4} V_{4\times4} M_{4\times4} X_4.$$

Viewing

There are two main steps in the viewing process:

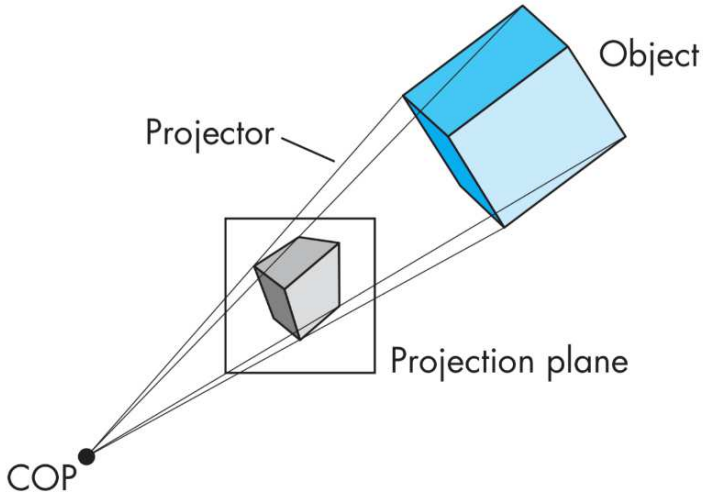
- Position and orient the camera
  - Setting the model-view matrix
  - Vertices in object coordinates will be transformed to eye or camera coordinates
- Selecting a lens
  - Setting the projection matrix
  - Normalize to a canonical view volume



Viewing

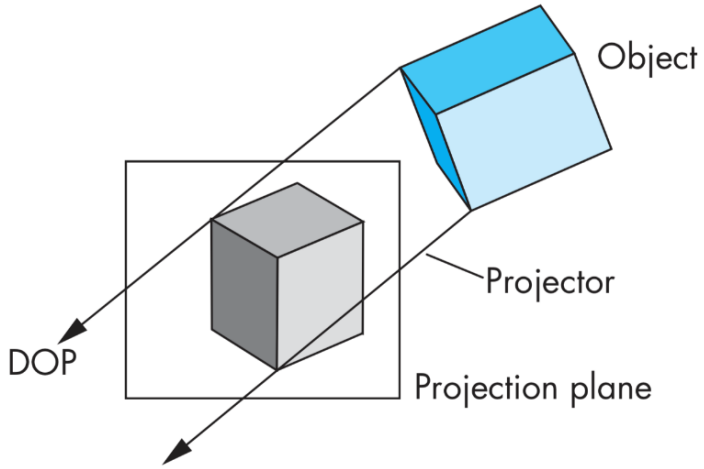
Projection determines how objects appear on screen

Perspective projection



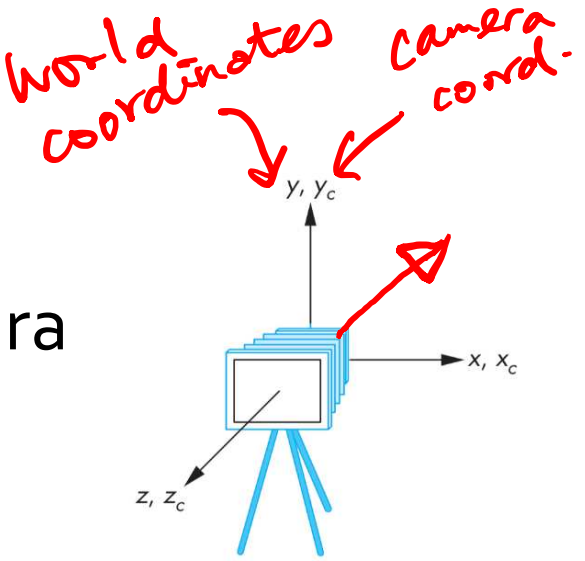
COP: Center of Projection
Original of the camera frame

Orthogonal projection



DOP: Direction of Projection
same as COP at infinity

The OpenGL Camera



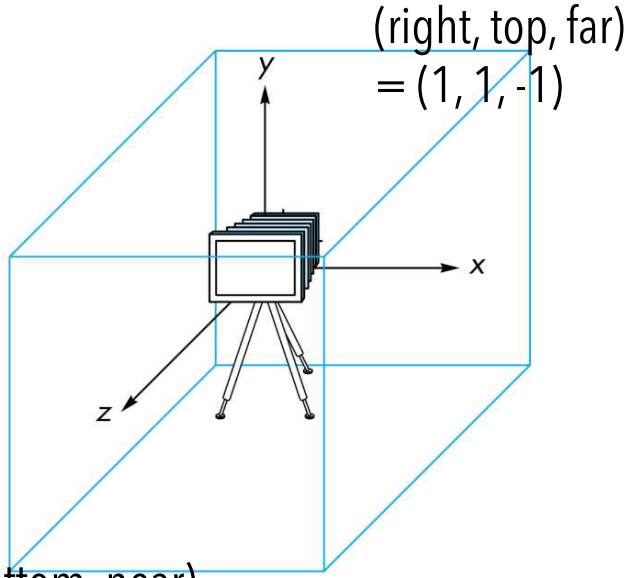
In OpenGL, initially the object and camera frames are the same

- Default model-view matrix is an identity

The camera is located at origin and points in the negative z direction

OpenGL also specifies a default canonical view volume that is a cube with sides of length 2 centered at the origin

- Default projection matrix is an identity (i.e., orthogonal projection)



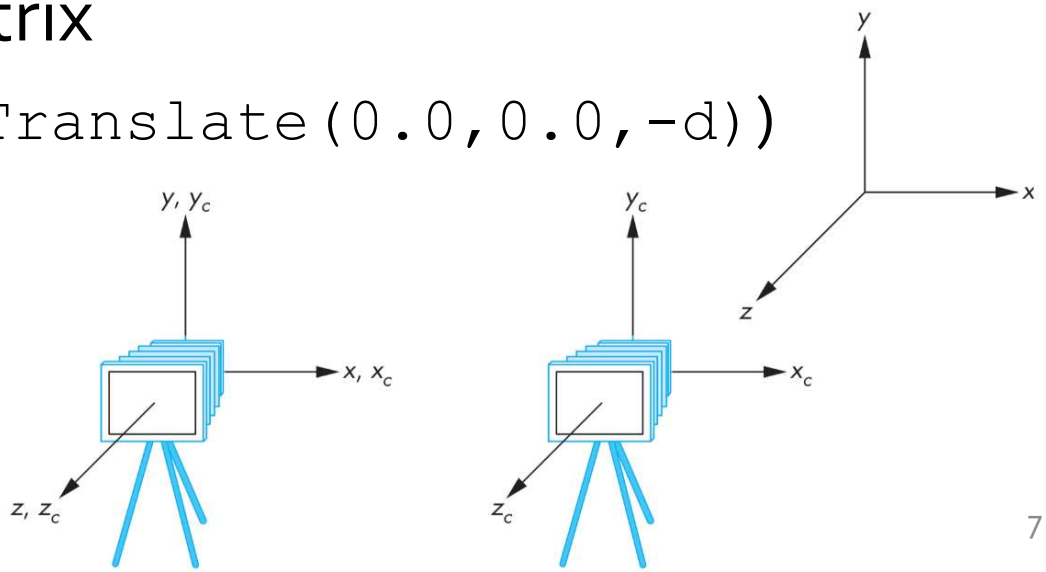
Moving the Camera Frame

Consider

- Moving the camera in the positive z direction
  - Translate the camera frame
- Moving the objects in the negative z direction
  - Translate the world frame

Both of these views are equivalent and are determined by the model-view matrix

- Want a translation (`Translate(0.0,0.0,-d)`)
- `d > 0`

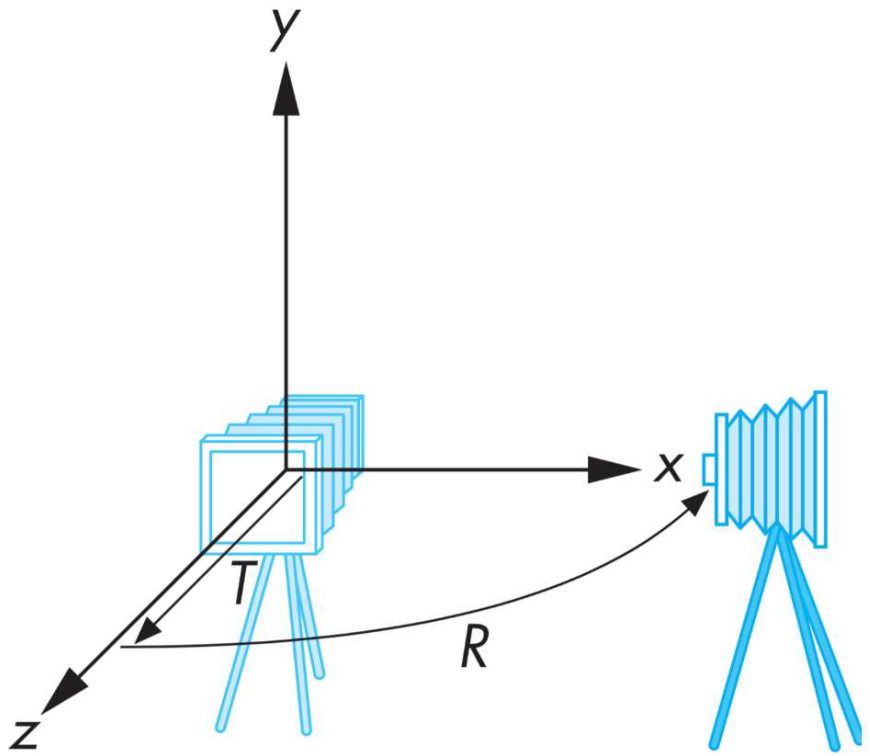


Moving the Camera Frame

We can move the camera to any desired position by a sequence of rotations and translations

Example: side view

- Rotate the camera
- Move it away from origin
- Model-view matrix C = TR



Move Camera:

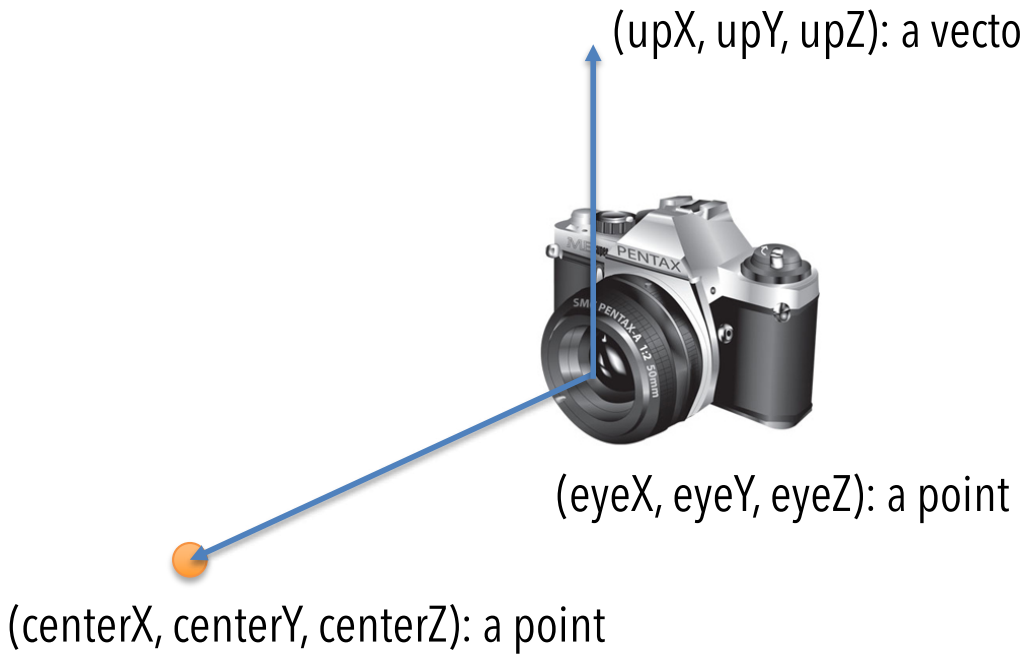
$R_y(\theta) \cdot T_z(d)$

Move World:

$T_z(-d) \cdot R_y(-\theta)$

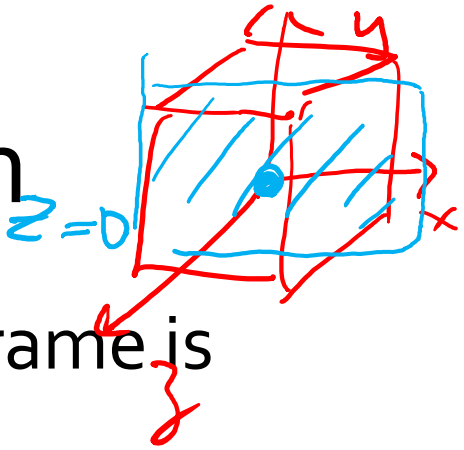
OpenGL API

```
LookAt(eyeX,eyeY,eyeZ,
 centerX,centerY,centerZ,upX,upY,upZ);
```



Note that this is a transformation that applies to the ModelView matrix

Projections and Normalization



The default projection in the eye (camera) frame is orthogonal

For points within the default view volume

$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$

Projection plane at z = 0

Most graphics systems use view normalization

- All other views are converted to the canonical view by transformations that determine the projection matrix
- Allows use of the same pipeline for all views

Default orthographic projection

$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

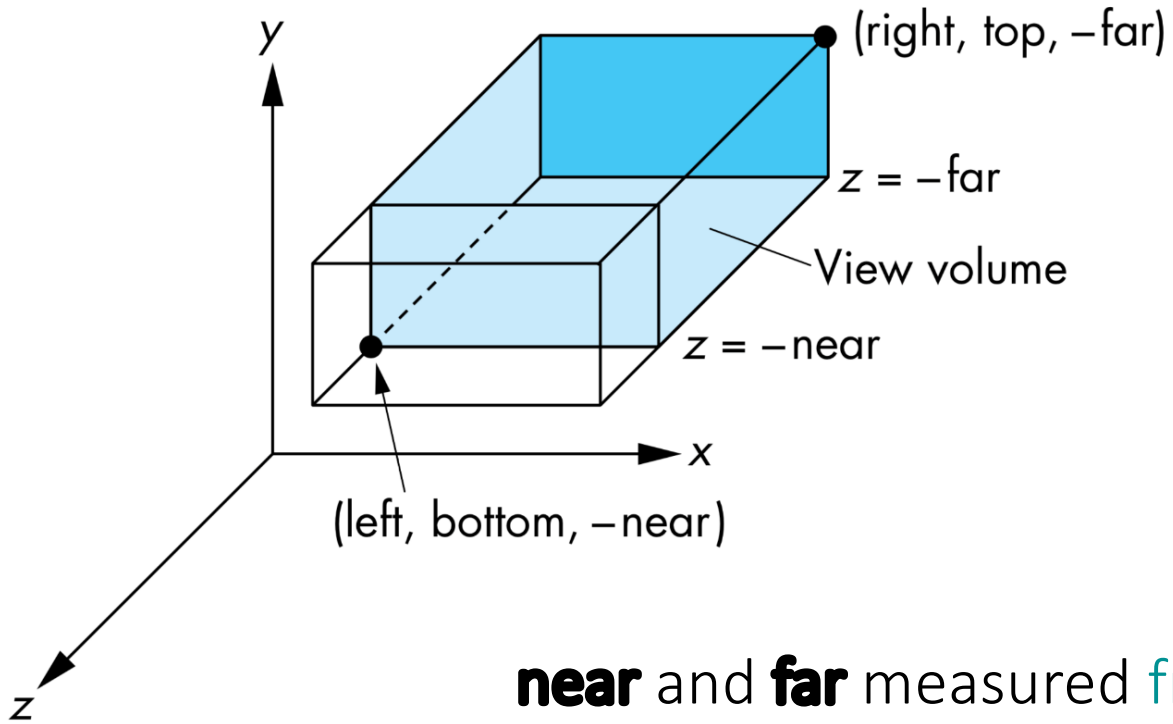
$$\mathbf{M}_{\mathbf{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{p} = (x, y, z, 1)^{\mathrm{T}}$$

$$\mathbf{p}_p = (x, y, 0, 1)^{\mathrm{T}}$$

OpenGL Orthogonal Viewing

`Ortho(left,right,bottom,top,near,far)`

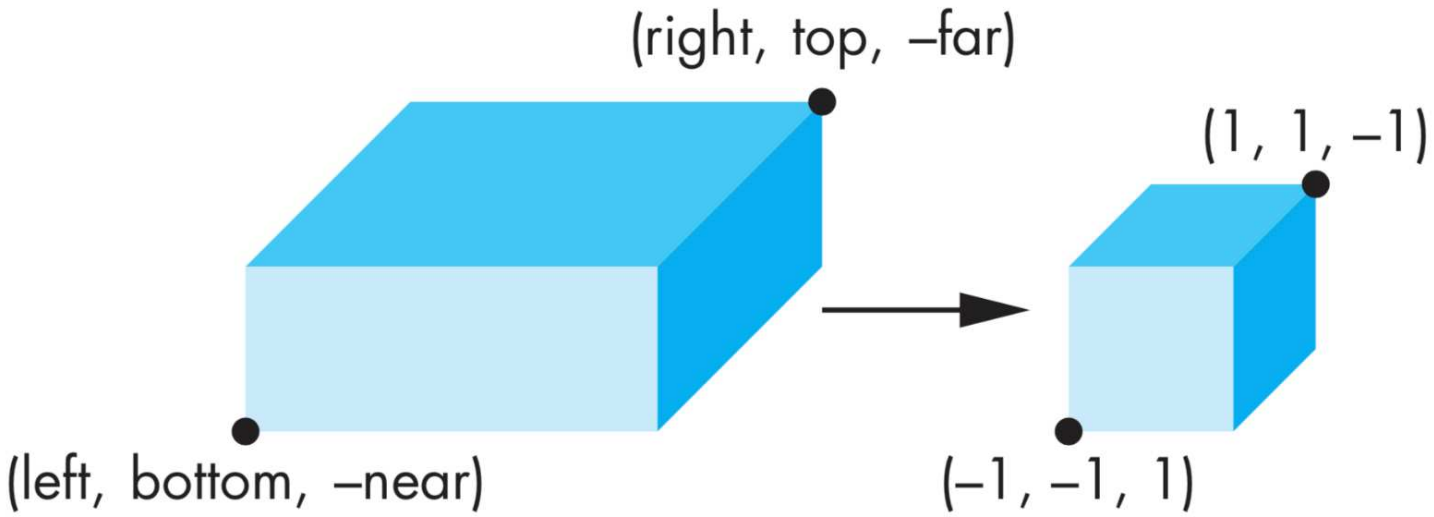


near and **far** measured from camera

How to normalize this into the canonical view?

Orthogonal Normalization

normalization ⇒ find transformation to convert specified clipping volume to canonical volume



Orthogonal Matrix

Two steps

- Move center to origin
  - T(-(left+right)/2, -(bottom+top)/2, (near+far)/2))
- Scale to have sides of length 2
  - S(2/(left-right), 2/(top-bottom), 2/(near-far))

$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Final Projection

Set $z = 0$

Equivalent to the homogeneous coordinate transformation

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hence, general orthogonal projection in 4D is

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$$

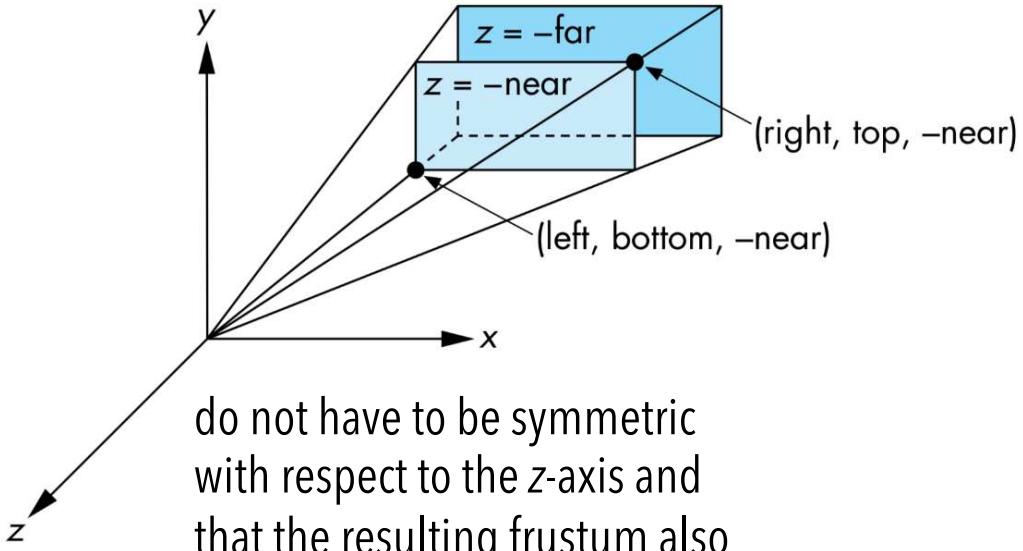
fro arbitrary orthogonal view

to canonical view

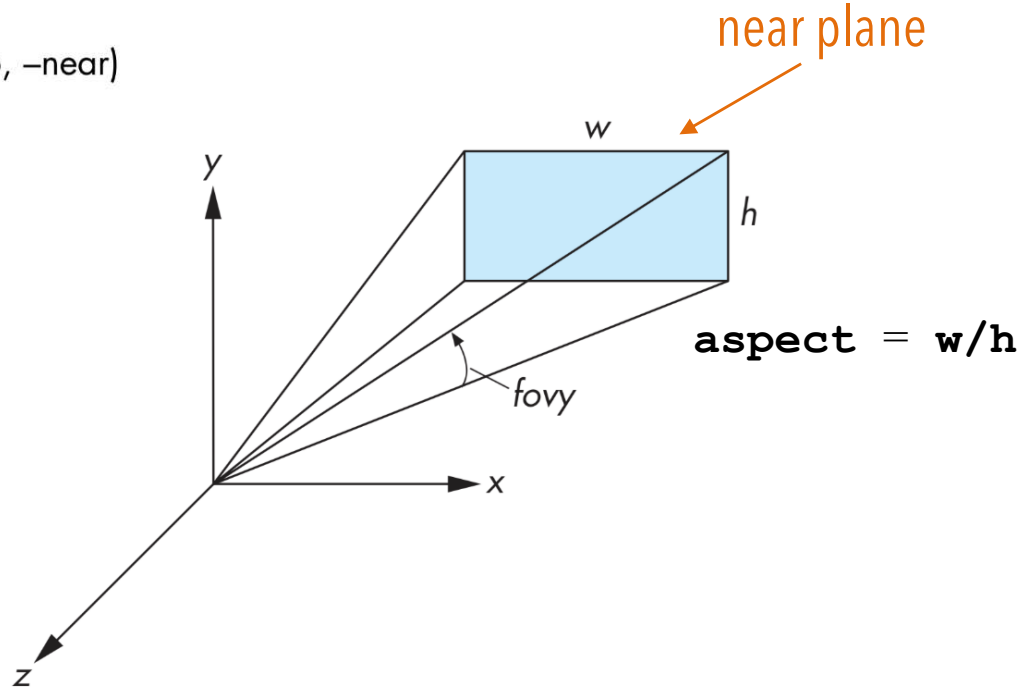
to project a 3D pt onto the view plane $z = 0$

OpenGL Perspective Viewing

`Frustum(left,right,bottom,top,near,far)`



do not have to be symmetric with respect to the *z*-axis and that the resulting frustum also does not have to be symmetric



`aspect = w/h`

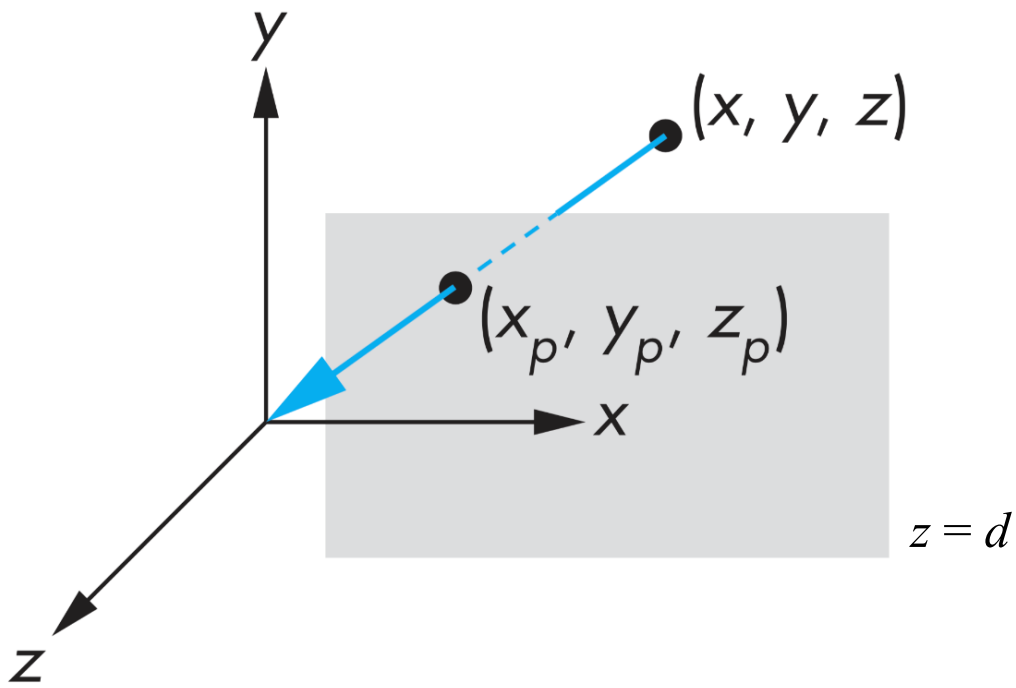
`Perpective(fovy, aspect, near, far)`

`fovy`: field of view in degrees in y direction
this often provides a better interface

Simple Perspective

Center of projection at the origin

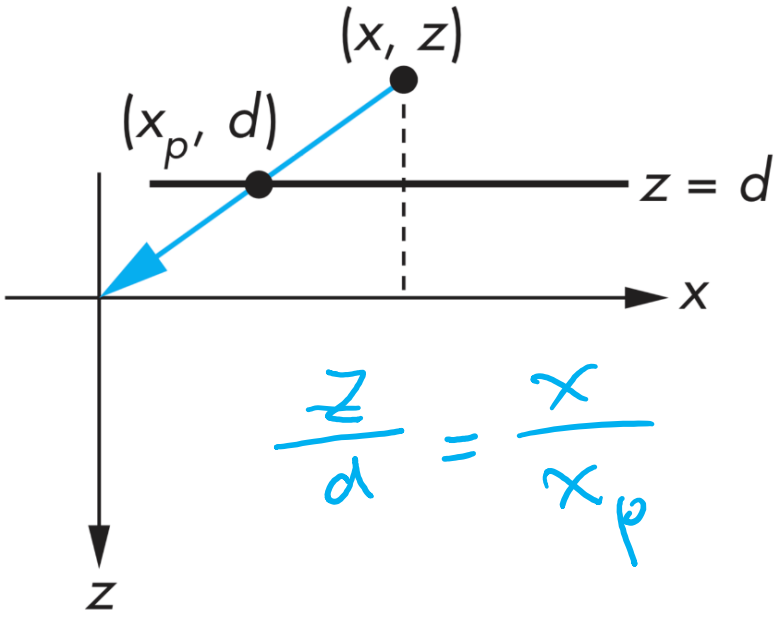
Projection plane $z = d$, $d < 0$



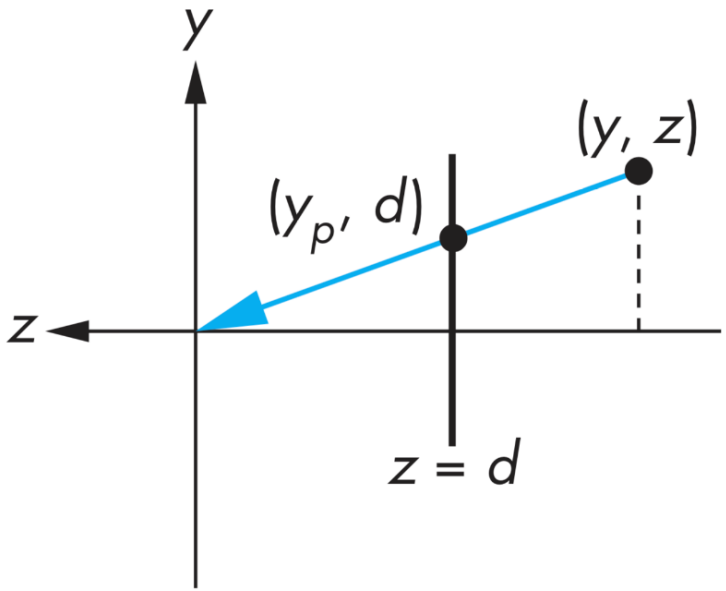
Perspective Equations

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} = M \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Consider top and side views



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



$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

Homogeneous Coordinate Form

Consider $\mathbf{p} = \mathbf{M}\mathbf{q}$ where

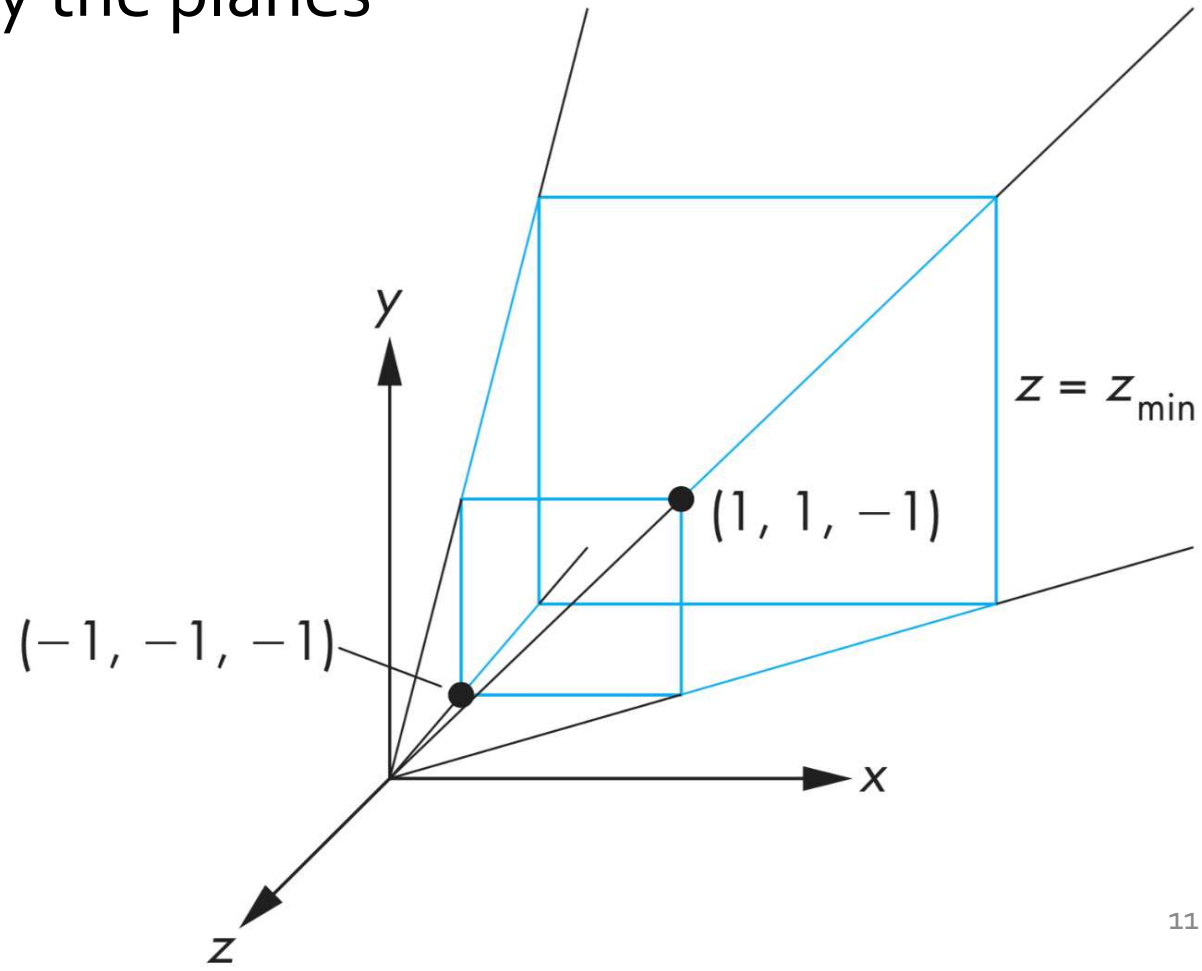
$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix}$$

$$\mathbf{q} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \Rightarrow \quad \mathbf{p} = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix} = \begin{pmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ \frac{z}{z/d} = 1 \\ 1 \end{pmatrix}$$

Simple Perspective

Consider a simple perspective with the COP at the origin, the near clipping plane at z = -1, and a 90 degree field of view determined by the planes

$$x = \pm z, y = \pm z$$



Perspective Matrices

Simple projection matrix in homogeneous coordinates

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note that this matrix is independent of the far clipping plane

Generalization

$\alpha = -1$

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

By this mapping, the point $(x, y, z, 1)$ goes to

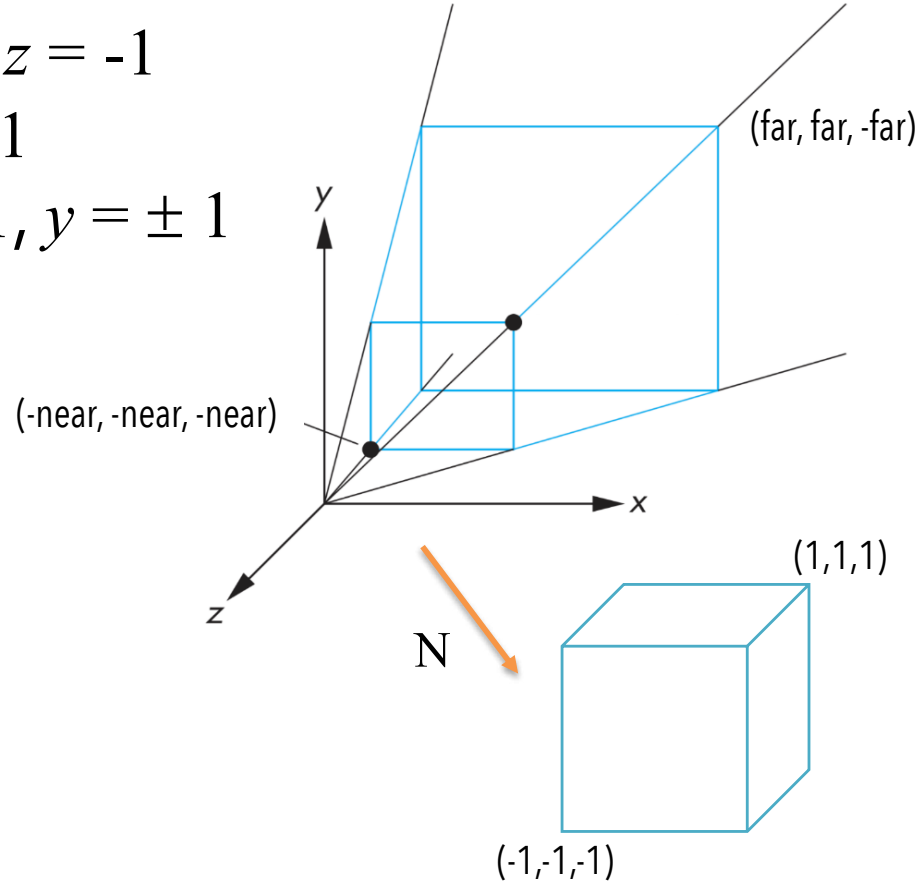
$$x'' = -x/z$$
$$y'' = -y/z$$
$$z'' = -(\alpha+\beta/z)$$

which projects orthogonally to the desired point regardless of α and β

Picking α and β

$$\begin{pmatrix} -1 \\ -1 \\ -1 \\ 1 \end{pmatrix} = N \cdot \begin{pmatrix} -n \\ -n \\ -n \\ 1 \end{pmatrix}$$

We want:
near plane z = -near be mapped to z = -1
far plane z = -far be mapped to z =1
and the sides be mapped to $x = \pm 1, y = \pm 1$



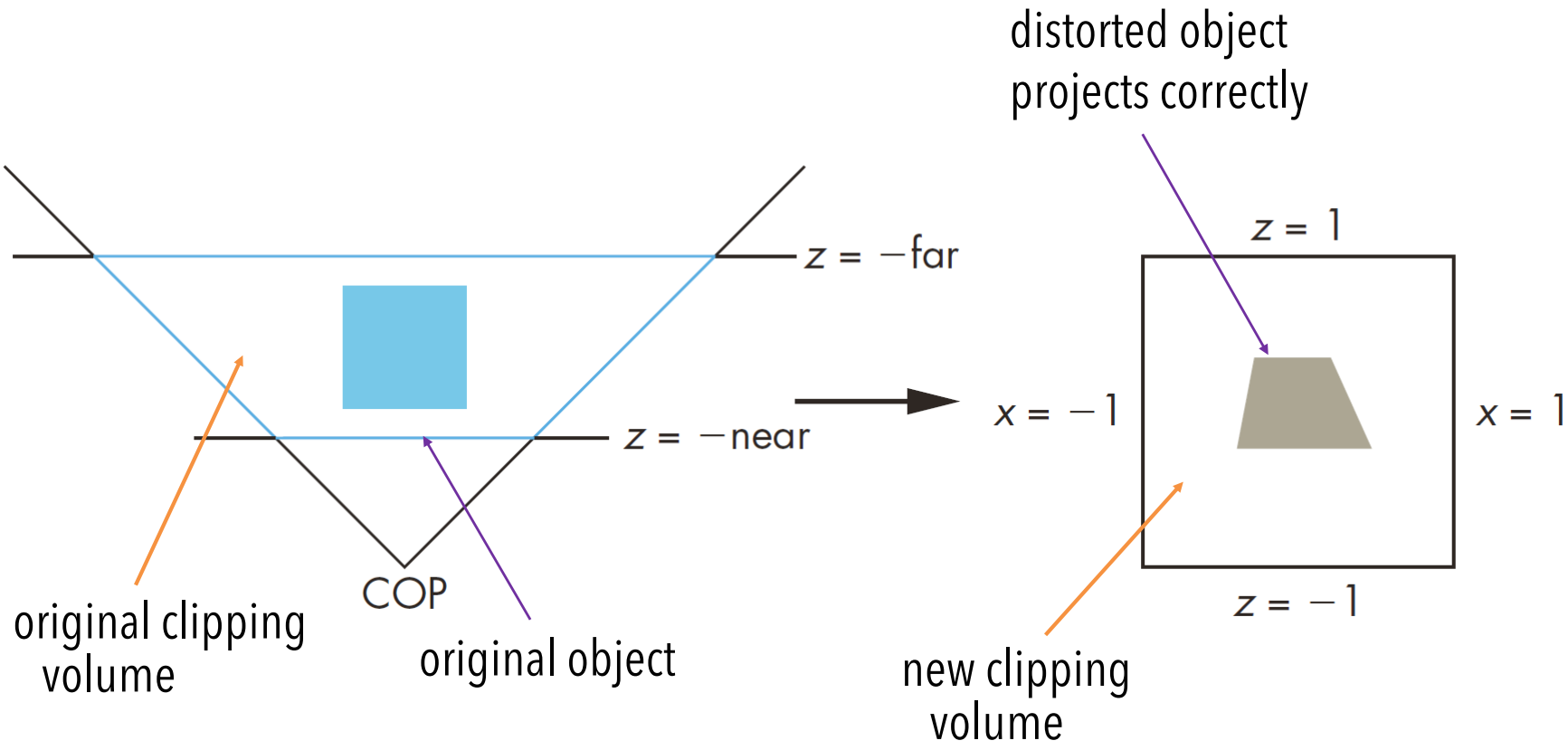
Solving two linear equations, we have

$$\alpha = \frac{\text{near} + \text{far}}{\text{near} - \text{far}}$$

$$\beta = \frac{2 \times \text{near} \times \text{far}}{\text{near} - \text{far}}$$

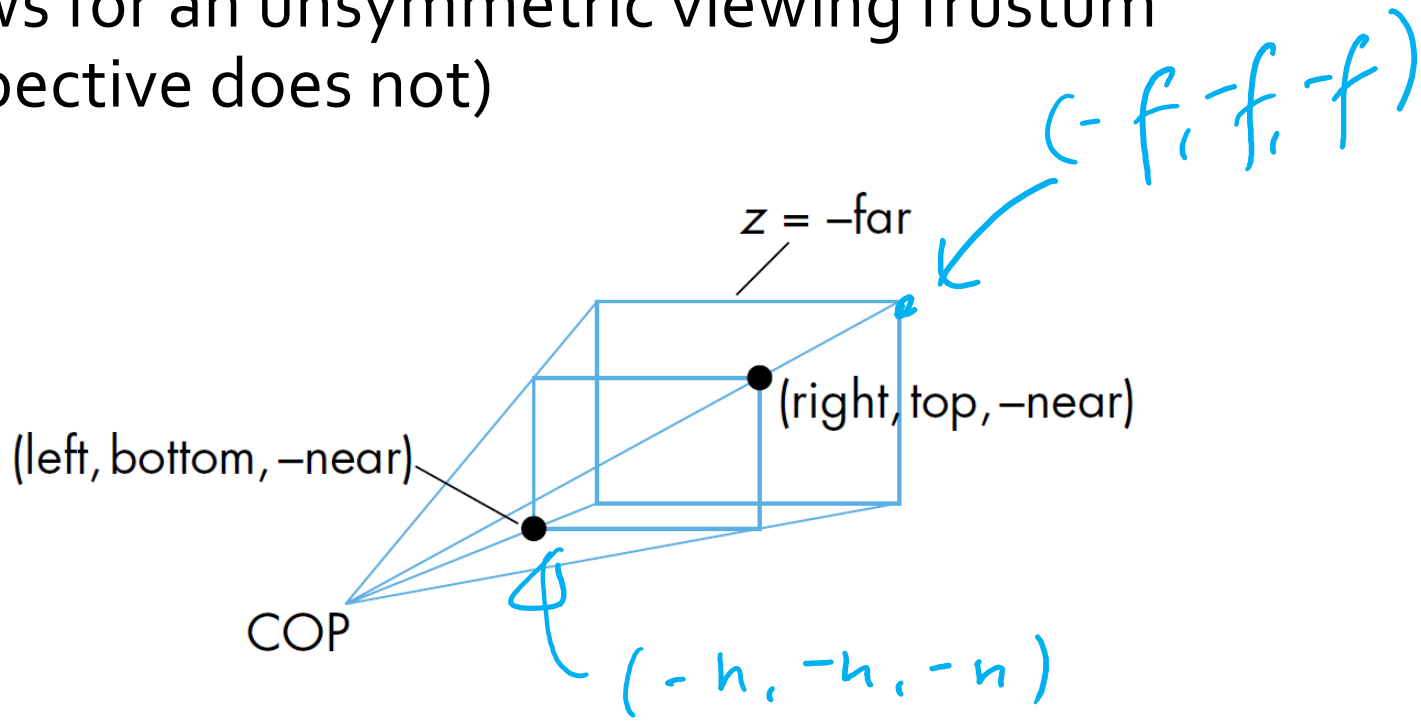
Then the new clipping volume is the canonical clipping volume

Normalization Transformation



OpenGL Perspective

glFrustum allows for an unsymmetric viewing frustum (although Perspective does not)



An unsymmetric viewing frustum can be normalized to the canonical view volume by first apply a shear and a scaling before applying N

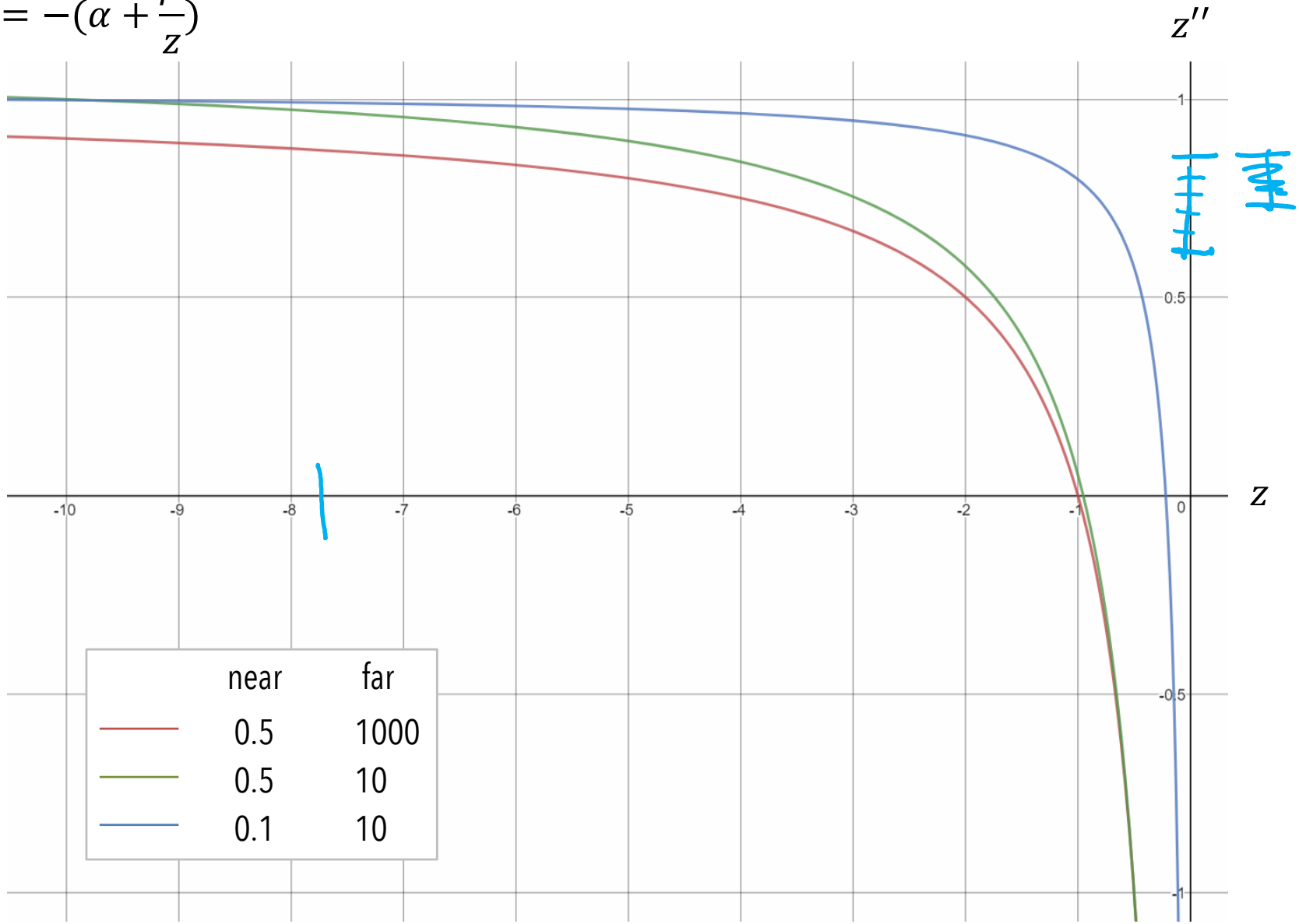
Normalization and Hidden-Surface Removal

Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if $z_1 > z_2$ in the original clipping volume then the for the transformed points $z_1' > z_2'$

Thus hidden surface removal works if we first apply the normalization transformation

However, note that the formula $z'' = -(\alpha+\beta/z)$ is nonlinear, which implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small

$$z'' = -(\alpha + \frac{\beta}{z})$$



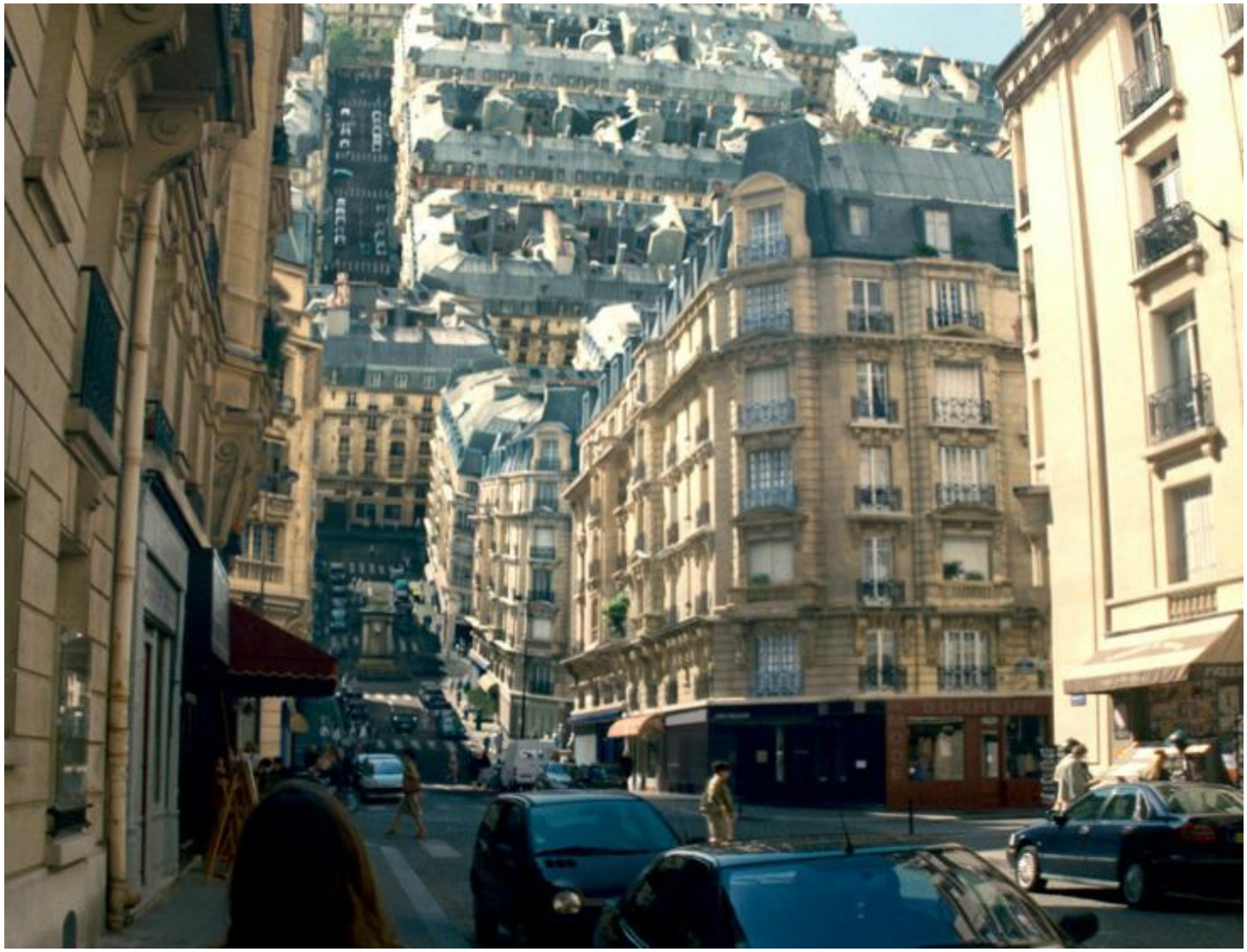
Why do we do it this way?

Normalization allows for a single pipeline for both perspective and orthogonal viewing

We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading

We simplify clipping

Special Projection Effects



The Movie "Inception"

Shading Polygons: Flat Shading

Illumination equations are evaluated at surface locations

- so where do we apply them?

Flat Shading

- do it once per polygon
- fill every pixel covered by polygon with the resulting color

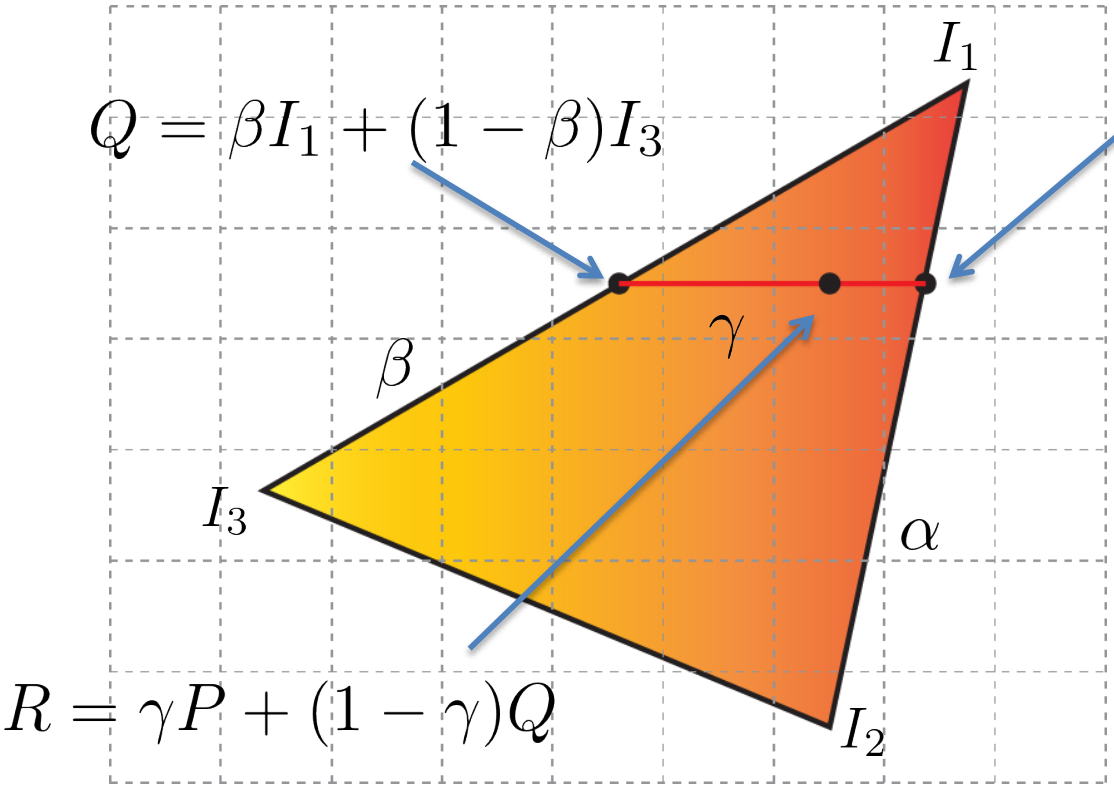
[www.dma.ufg.ac.at/app/link/Grundlagen%3A3D-Grafik/module/9728]

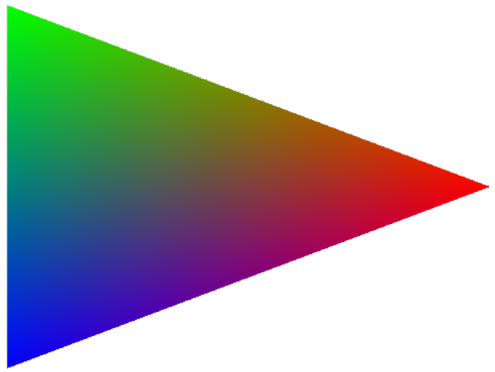
OpenGL — `glShadeModel(GL_FLAT)`

Shading Polygons: Gouraud Shading

Alternatively, we do lighting calculation once for each vertex

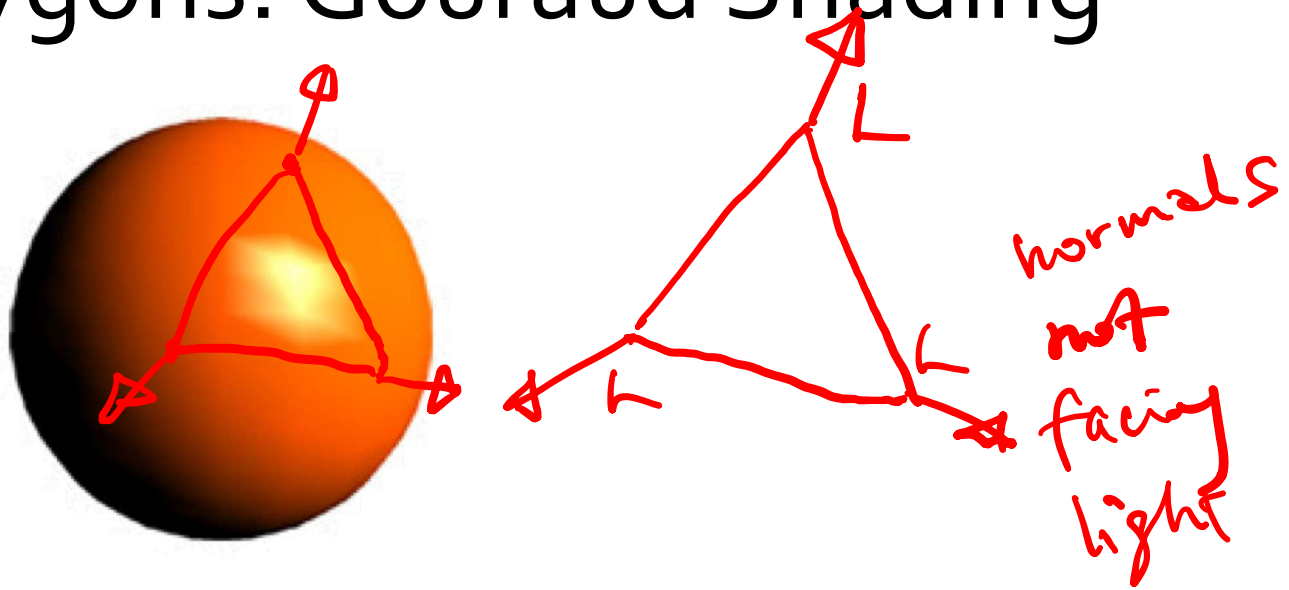
- compute color for each covered pixel
- linearly interpolate colors over polygon





Example of linear interpolation of three different colors at three vertices

Shading Polygons: Gouraud Shading



If underlying geometry is too coarse, may lead to shading artifacts

Misses details that don't fall on vertex

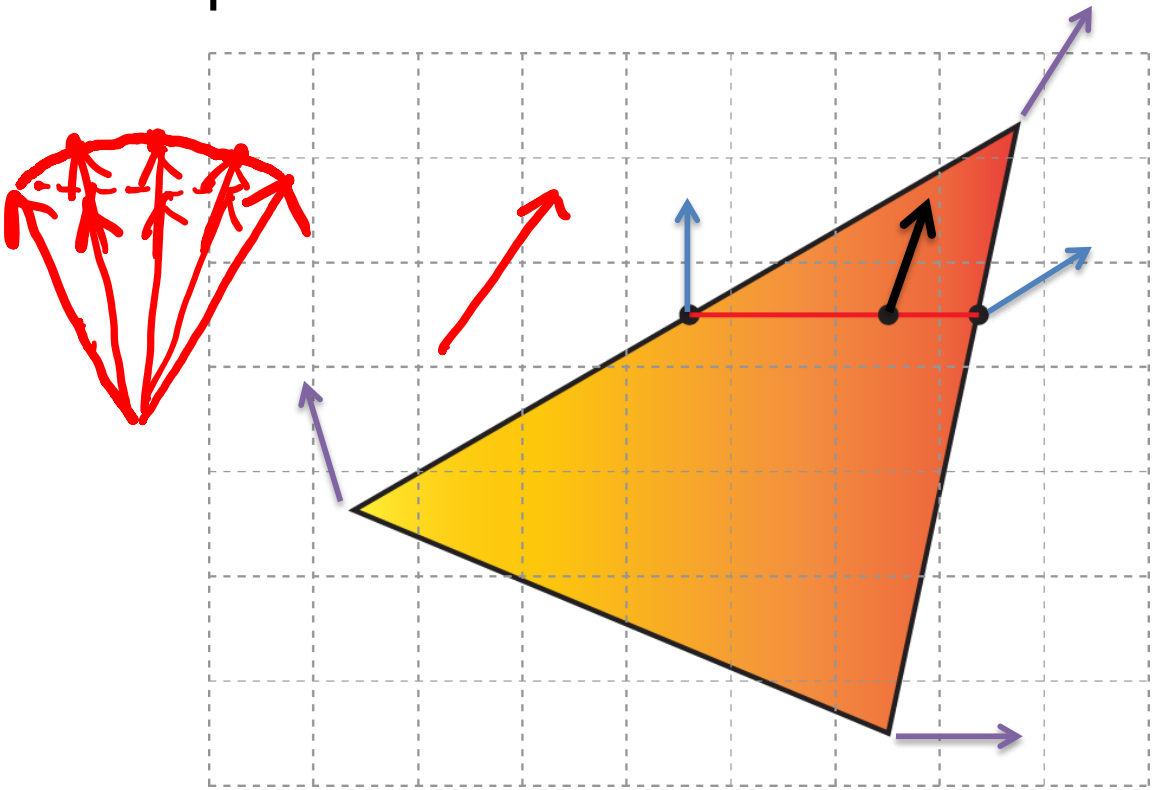
- specular highlights, for instance

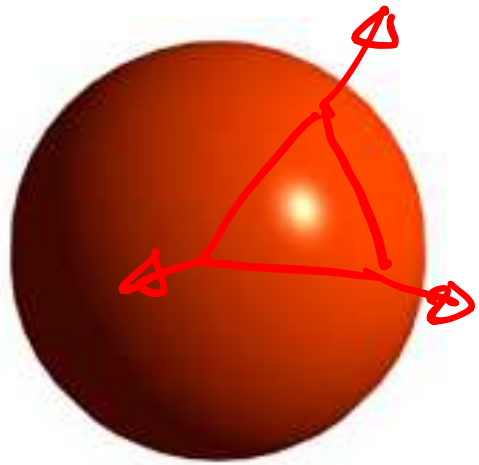
OpenGL — `glShadeModel(GL_SMOOTH)`

Shading Polygons: Phong Shading

Lighting calculation is carried out for every pixel covered by a triangle.

Surface normal at a pixel is estimated using bilinear interpolation.





Best shading but computationally intensive

OpenGL — not directly supported

Defining Materials in OpenGL

Just like everything else, there is a current material

- specifies the reflectances of the objects being drawn
- reflectances (e.g., k_d) are RGB triples

Set current values with `glMaterial(...)`

```
GLfloat tan1[] = {0.8, 0.7, 0.3, 1.0};
GLfloat tan2[] = {0.4, 0.35, 0.15, 1.0};

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, tan1);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, tan1);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, tan2);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 50.0);
```

Defining Lights in OpenGL

A fixed set of lights are available (at least 8)

- turn them on with `glEnable(GL_LIGHTx)`
- set their values with `glLight(…)`

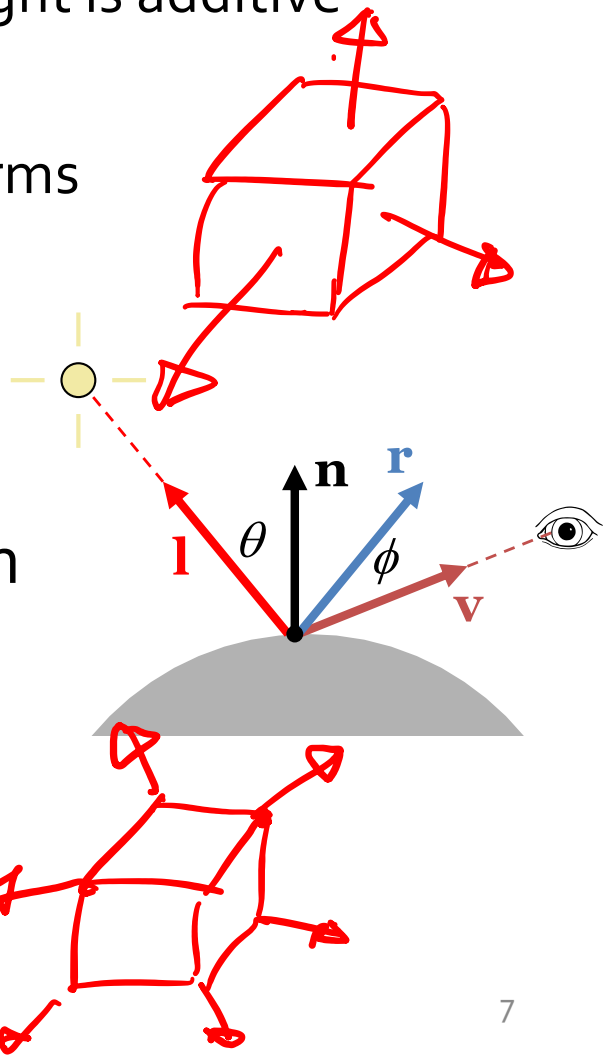
```
GLfloat white[] = {1.0, 1.0, 1.0, 1.0}
GLfloat p[] = {-2.0, -3.0, 10.0, 1.0}; // w=0 for directional light

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

glLightfv(GL_LIGHT0, GL_POSITION, p);
glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
glLightfv(GL_LIGHT0, GL_SPECULAR, white); // can be different

glEnable(GL_NORMALIZE); // guarantee unit normals
```

Summarizing the Shading Model

We describe local appearance with illumination equations

- consists of a sum of set of components — light is additive
- treat each wavelength independently
- currently: diffuse, specular, and ambient terms

$$I = I_L k_d (\mathbf{n} \cdot \mathbf{l}) + I_L k_s (\mathbf{r} \cdot \mathbf{v})^n + I_a k_a$$



Must shade every pixel covered by polygon

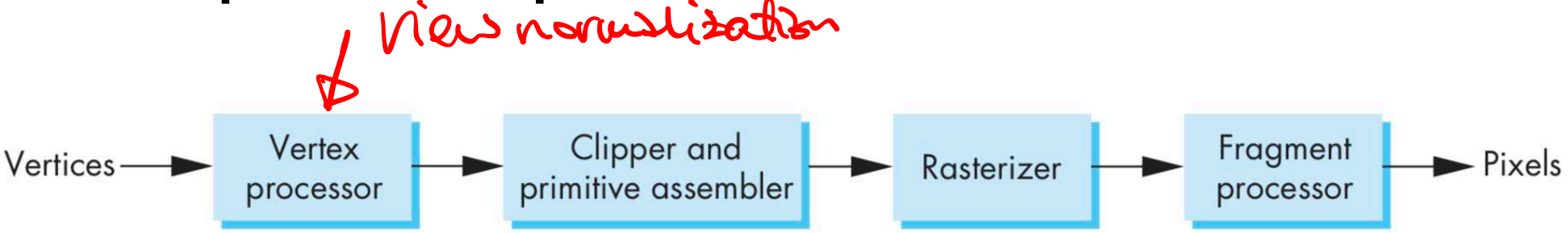
- flat shading: constant color
- Gouraud shading: interpolate vertex colors
- Phong shading: interpolate vertex normals

COMP3271 Computer Graphics

Clipping & Rasterization

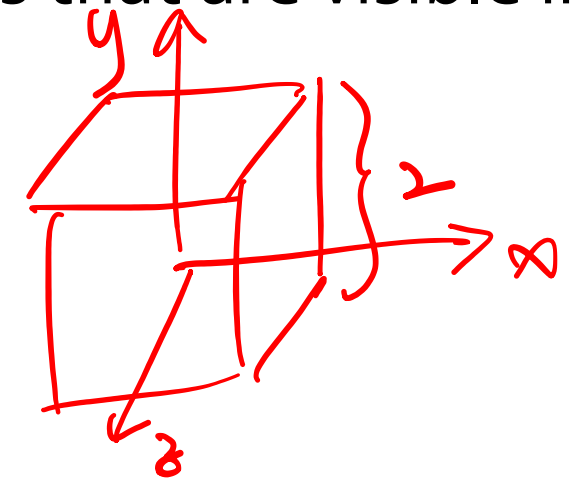
2019-20

Graphics Pipeline Overview



Clipping — To eliminate objects (or part of objects) that lie outside the viewing volume

Rasterization — To produce fragments from the remaining objects that are visible in the final image



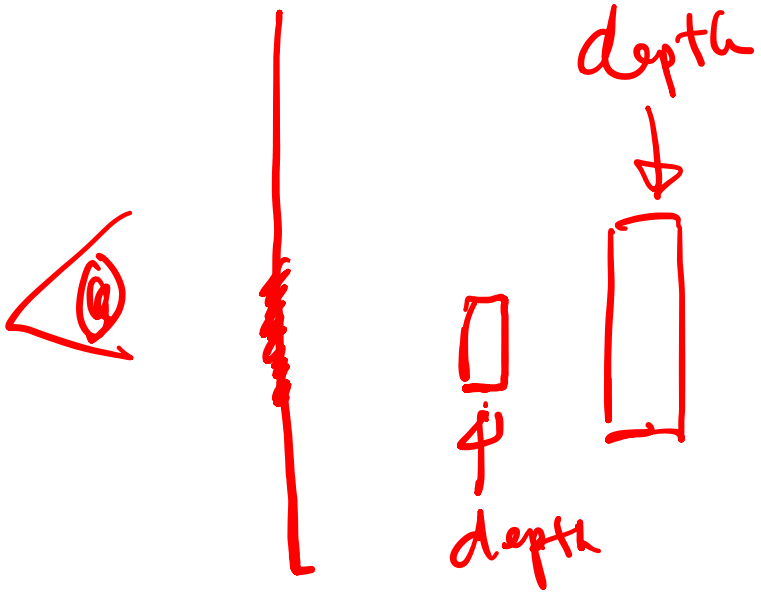
Two Rendering Approach

For every pixel, determine which object that projects on the pixel is closest to the viewer and compute the shade of this pixel

- Ray tracing paradigm

For every object, determine which pixels it covers and shade these pixels

- Pipeline approach
- Must keep track of depths



Clipping

2D against clipping window

3D against clipping volume

Easy for line segments & polygons

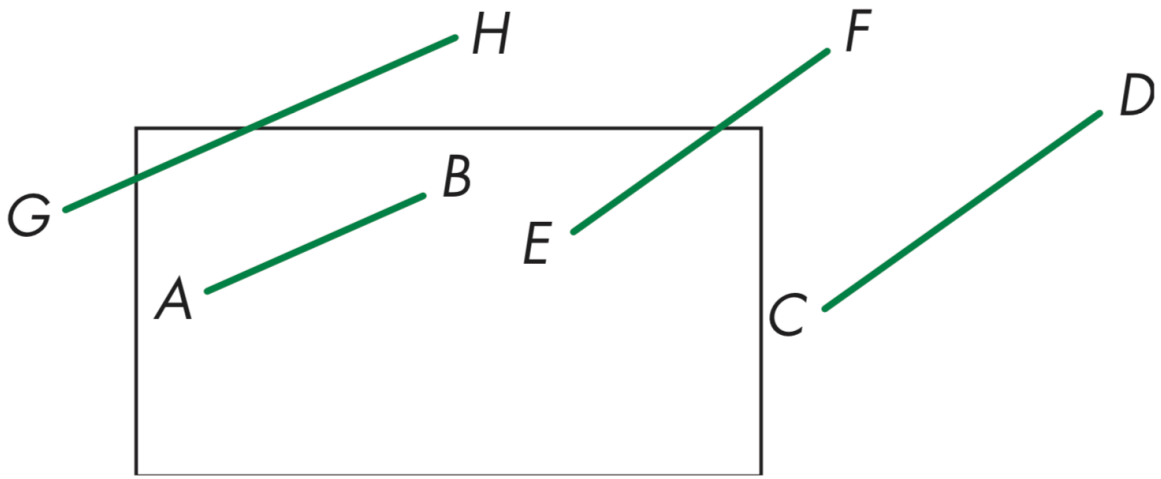
Hard for curves and text

- Convert to lines and polygons first

Clipping 2D Line Segments

Brute force approach: compute intersections with all sides of clipping window

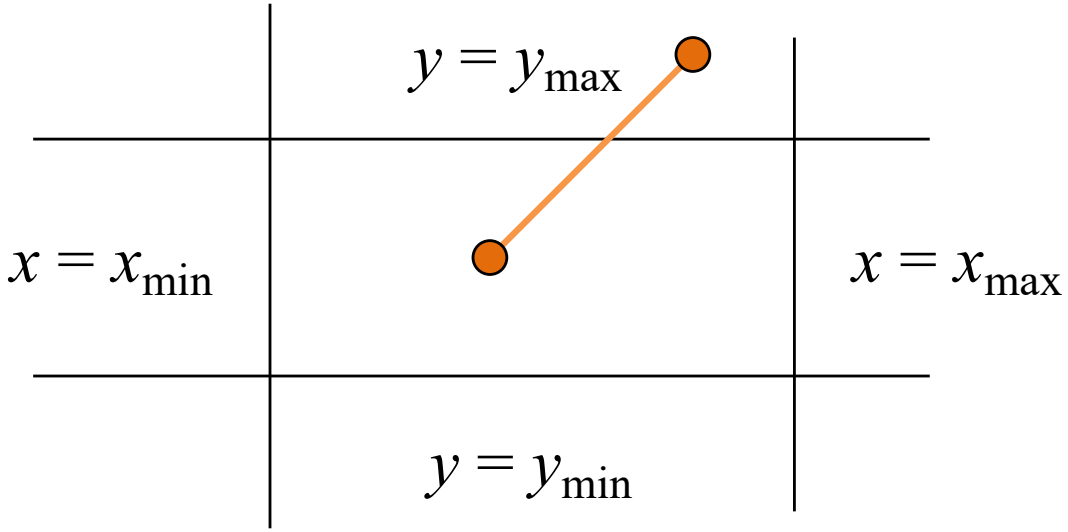
- Inefficient: one division per intersection



Cohen-Sutherland Algorithm

Idea: eliminate as many cases as possible without computing intersections

Start with four lines that determine the sides of the clipping window

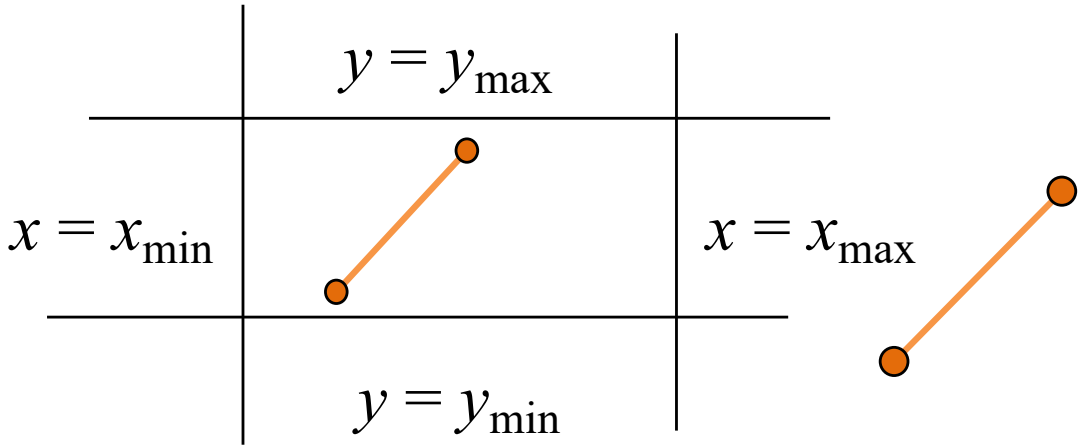


Viewing volume must be axis-aligned

The Cases

Case 1: both endpoints of line segment inside all four lines

- Draw (accept) line segment as is



Case 2: both endpoints outside all lines and on same side of a line

- Discard (reject) the line segment

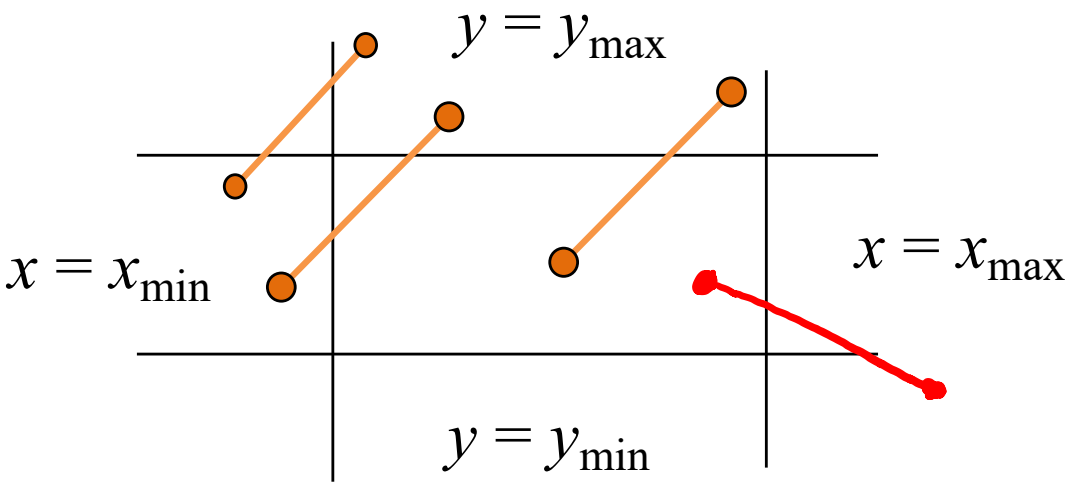
The Cases

Case 3: One endpoint inside, one outside

- Must do at least one intersection

Case 4: Both outside

- May have part inside
- Must do at least one intersection



Other Issues with Z-buffering

Z-fighting: Due to the lack of precision of depth buffer, two fragments from different objects that are close to each other (and far from the camera) will flicker back and forth on alternating frames.

- Alleviation: use 24-bit or 32-bit depth buffers or adjust near and far clipping planes for a smaller view frustum

Take a look at the example: https://youtu.be/9AcCrF_nX-I

Z-buffering is done on a pixel-by-pixel basis. If an entire object is blocked by another, it still needs to undergo all the lighting calculations and rasterization before all its pixels are discarded.

- Solutions: object culling algorithms such as BSP (binary spatial partitioning), occlusion volumes, etc.

Painter's Algorithm

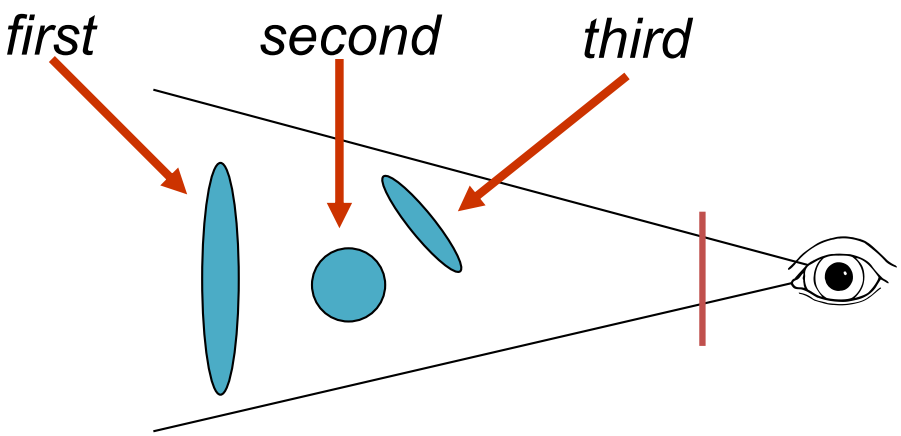
Developed thousands of years ago

- probably by cave dwellers

Draws every object in depth order

- from back to front
- near objects overwrite far objects

What could be simpler?



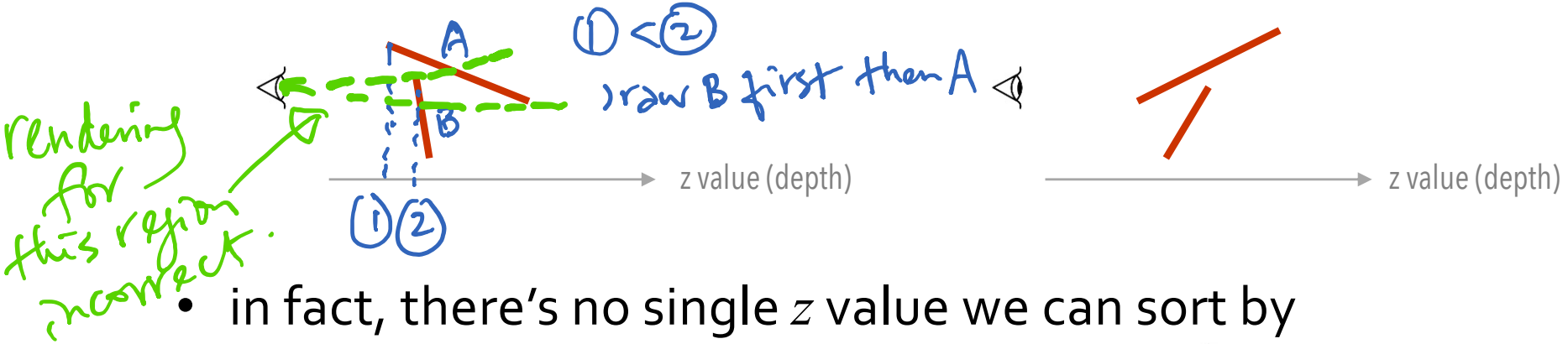
Painter's Algorithm:

```
sort objects back to front

loop over objects
  rasterize current object
  write pixels
```

But the Catch is in the Depth Sorting

What do we sort by?

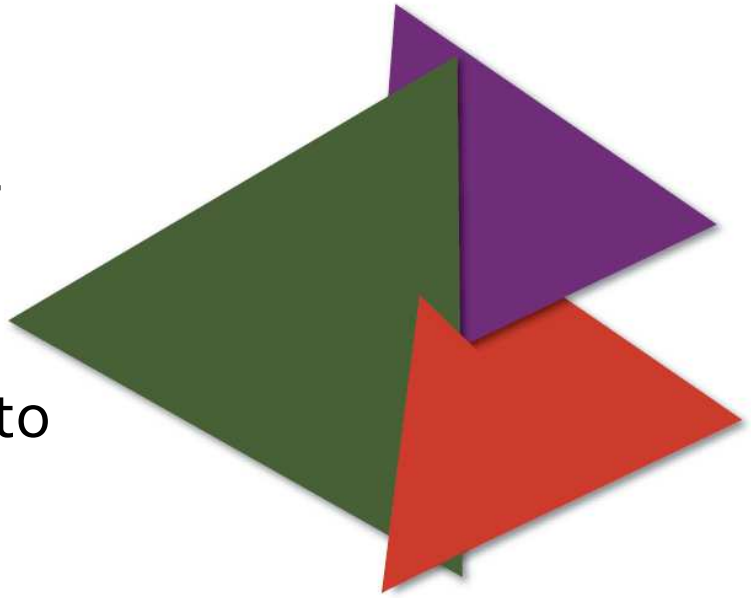
- minimum z value — no maximum z value — no



- in fact, there's no single z value we can sort by

Worse yet, depth ordering of objects can be cyclic

- may need to split polygons to break cycles



Looking at Painter's Algorithm

It has some nice strengths

- the principle is very simple
- handles transparent objects nicely
  - just composite new pixels with what's already there

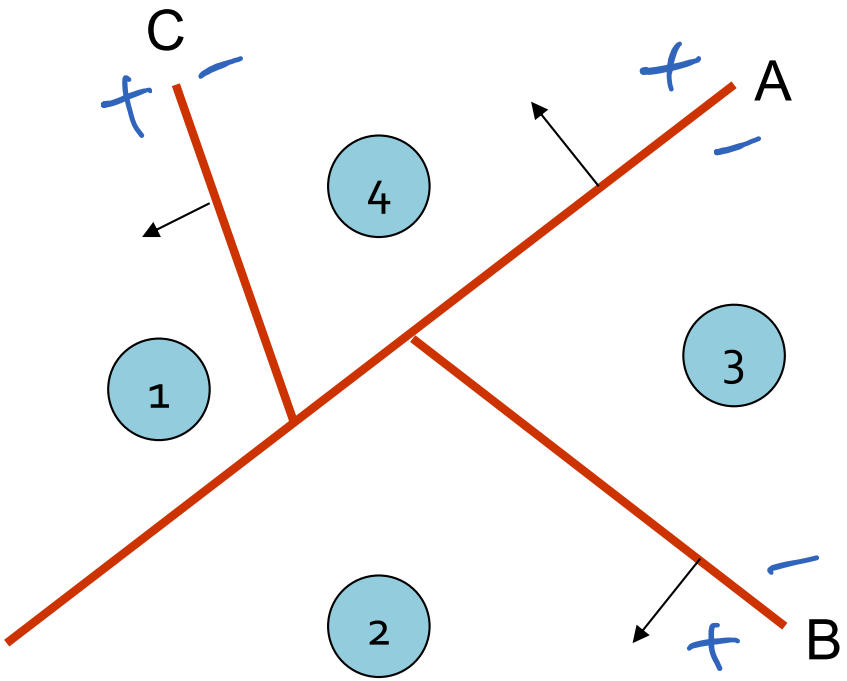
But it also has some noticeable weaknesses

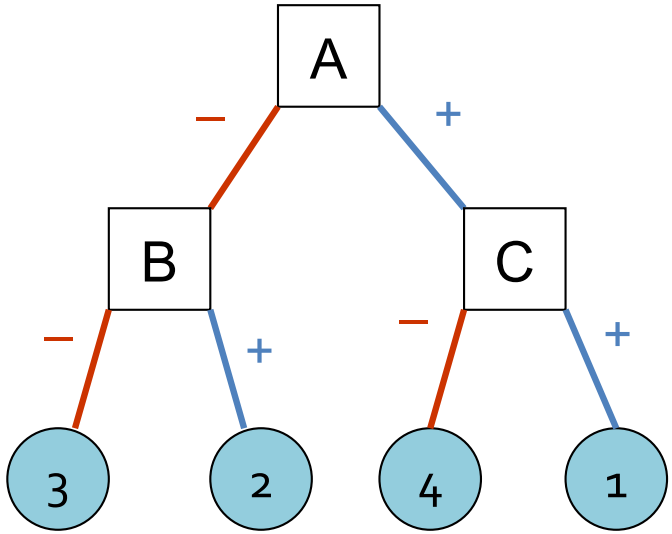
- general sorting is a little expensive — worse than *O(n)*
- need to do splitting for depth cycles, interpenetration, …

A Quick Look at BSP Trees

Recursively partition space with planes

- this defines a binary space partitioning tree
- need to split objects hit by planes

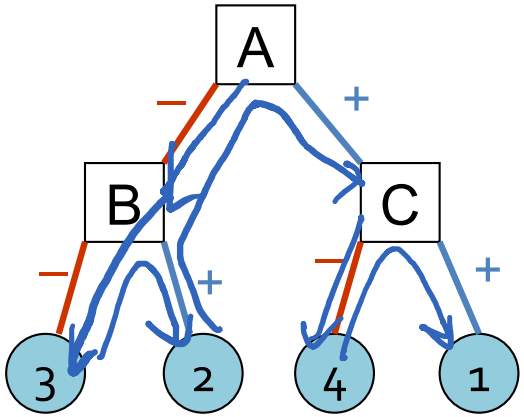




A Quick Look at BSP Trees

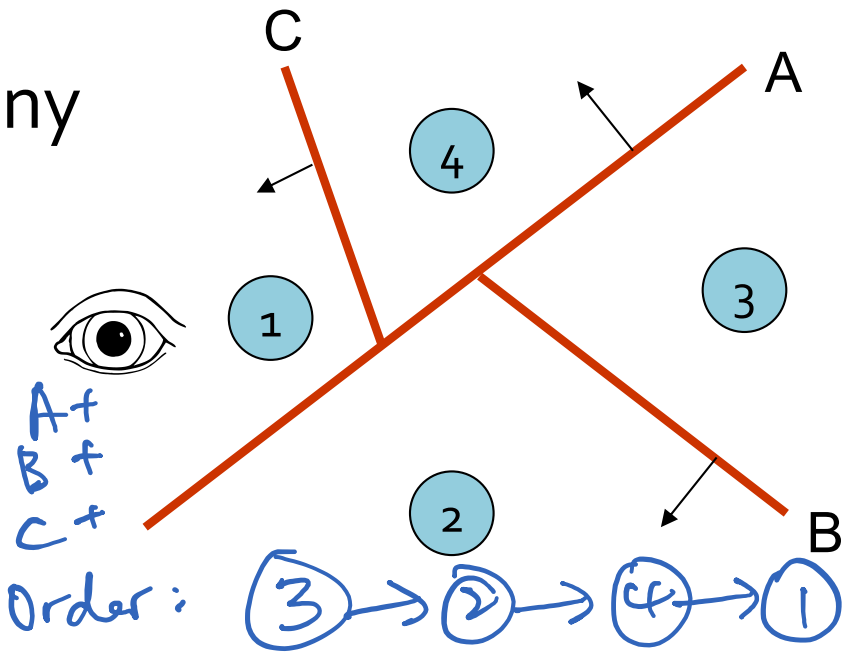
Can use this to draw scene in order (back to front for instance)

- start at root plane
- figure out which side the viewpoint is on
- descend on the opposite first
- do this recursively



Can use one BSP tree for any possible viewpoint.

View Independent Data Structures



A Quick Look at BSP Trees

Can use this to draw scene in order (back to front for instance)

- start at root plane
- figure out which side the viewpoint is on
- descend on the opposite first
- do this recursively

Originally developed in early 80's

- for Painter's Algorithm, for instance
- resurrected by PC game programmers in the early 90's
  - e.g., by John Carmack for Doom
- it's quite handy if you don't have a z-buffer

Ray Casting

This is a very general algorithm

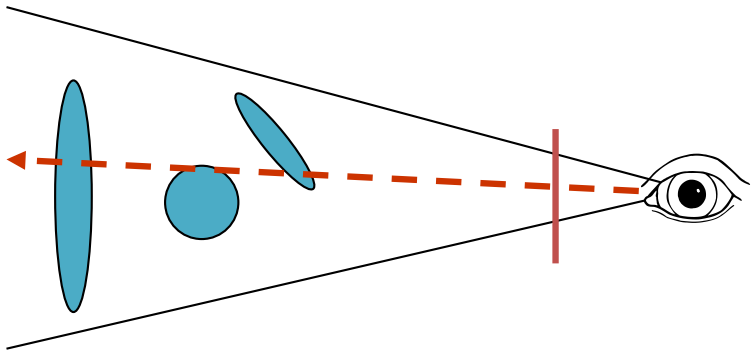
- works with any primitive we can write intersection tests for
- but it's hard to make it run fast

We'll come back to this idea later

- can use it for much more than visibility testing
- shadows, refractive objects, reflections, motion blur, …

Ray Casting:

```
loop over every pixel (x,y)
  shoot ray from eye through (x,y)
  intersect with all surfaces
  find first intersection point
  write pixel
```



A Classification of Visibility Algorithms

Image-space

- Z-buffer
- Ray-casting

Object-space

- Painter's algorithm
- BSP

They are all view-dependent except BSP trees.

COMP3271 Computer Graphics

Texture Mapping

2019-20

Objectives

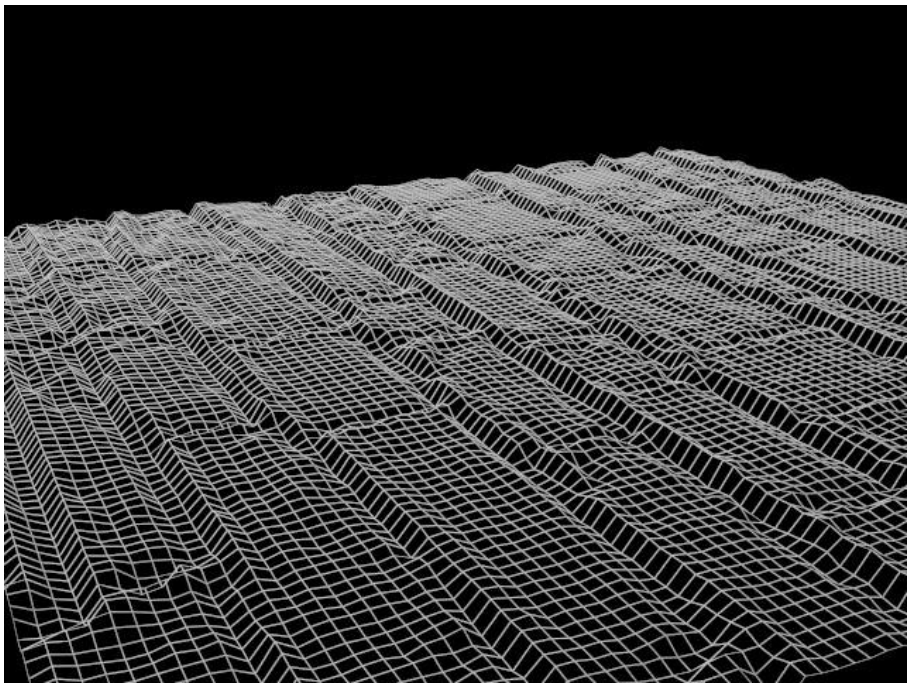
What texture mapping is for

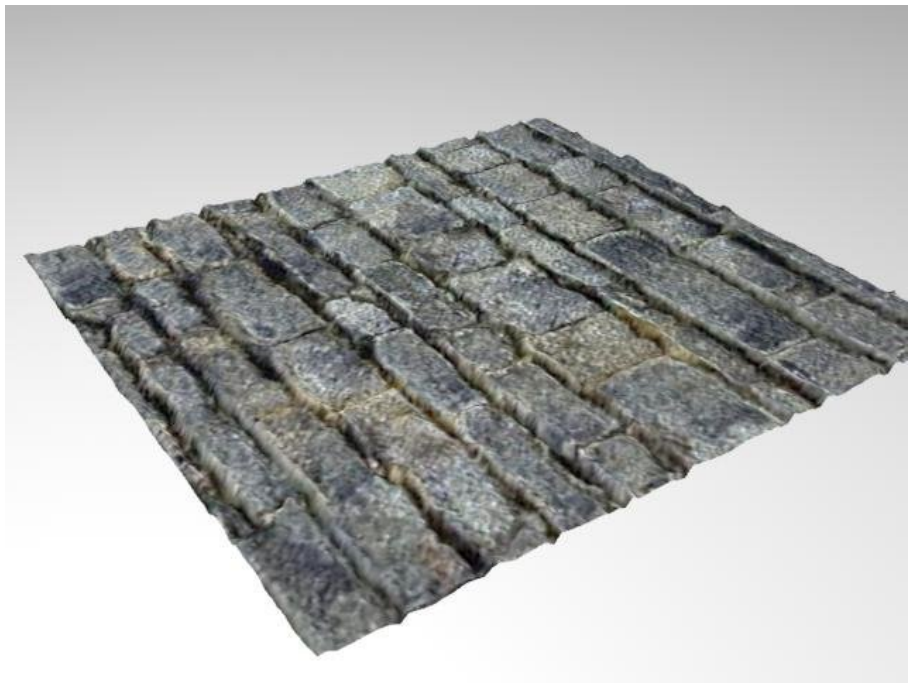
Understand the basic texture mapping process

How Do We Model Intricate Surface Detail?

Approach #1: Explicit geometric representation

- actual polygons that model all the surface variations
- up to some finest level of detail
- may generate a *lot* of polygons



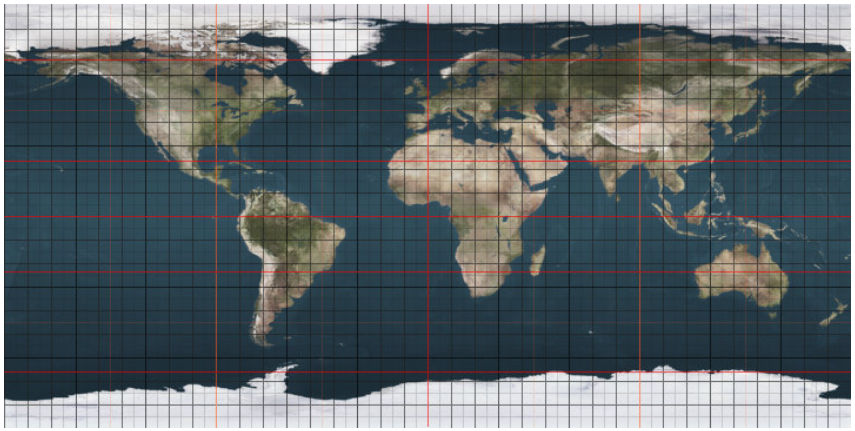


https://ch.3dexport.com/3dmodel-3d-stone-wall-55616.htm

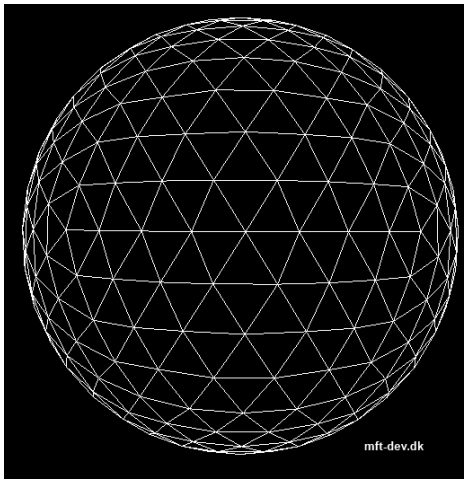
How Do We Model Intricate Surface Detail?

Approach #2: Geometry + texture images

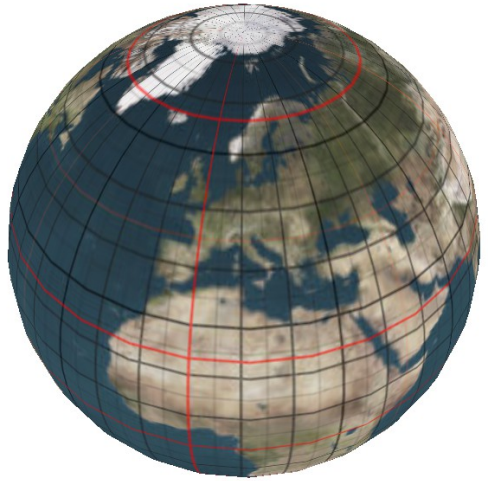
- geometry only describes the general shape of the object
- paste an image onto the surface to give the appearance with fine details



+



=



https://commons.wikimedia.org/wiki/File:OpenGL_Tutorial_Textured_Spheres.jpg
https://mft-dev.dk/uv-mapping-sphere/

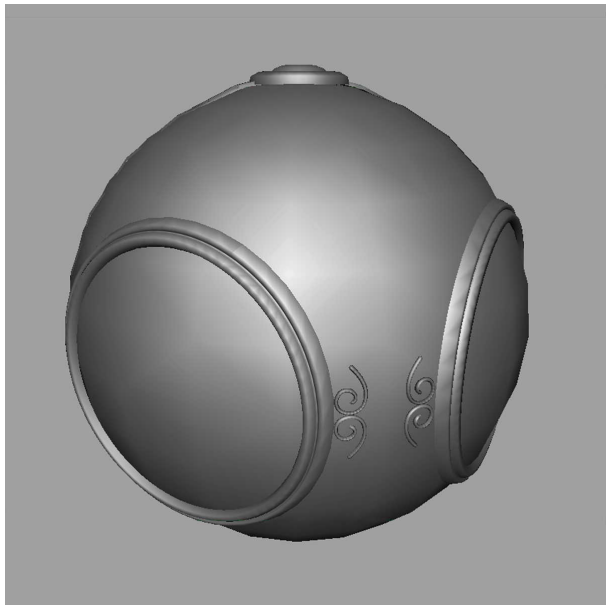
The Importance of Surface Texture

Objects in the real world have rich, detailed surface textures

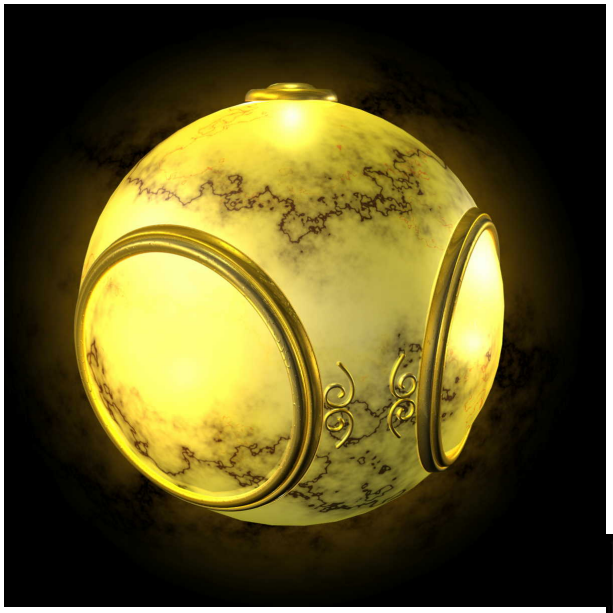
- to produce as-realistic-as-possible scenes, we must replicate this detail
- uniformly colored surfaces only get us so far

Generated with Blue Moon Rendering Tools – www.bmrt.org

Texture Mapping



geometric model



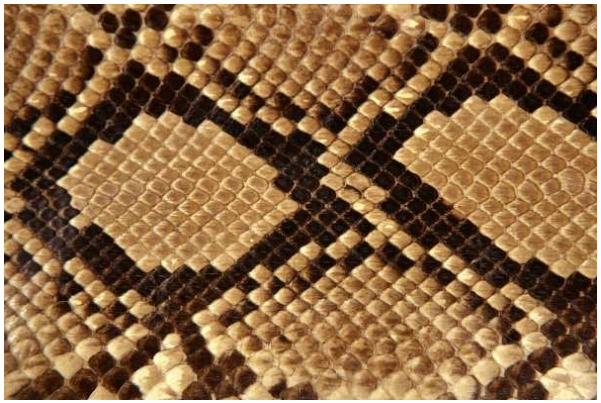
texture mapped

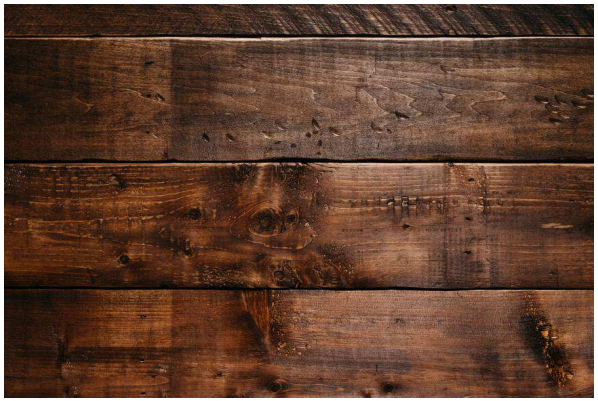
Often We Use Simple Patterns

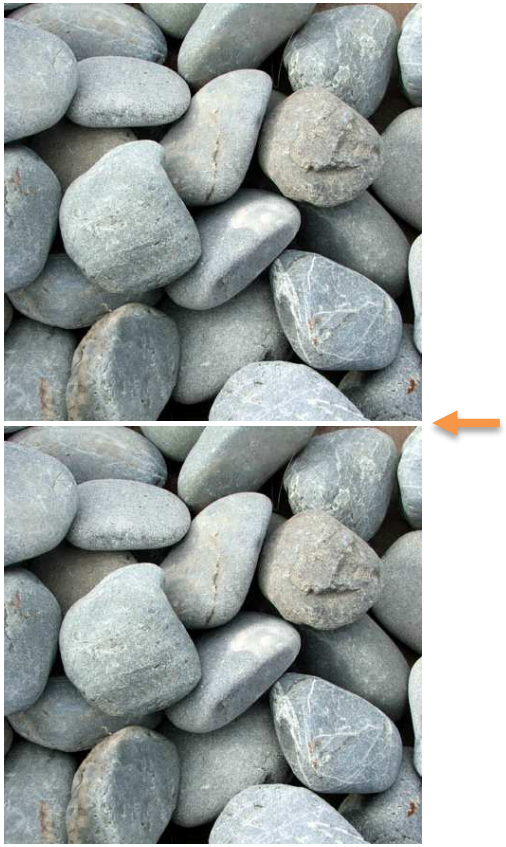
Generally useful for skin, bricks, stucco, granite, …

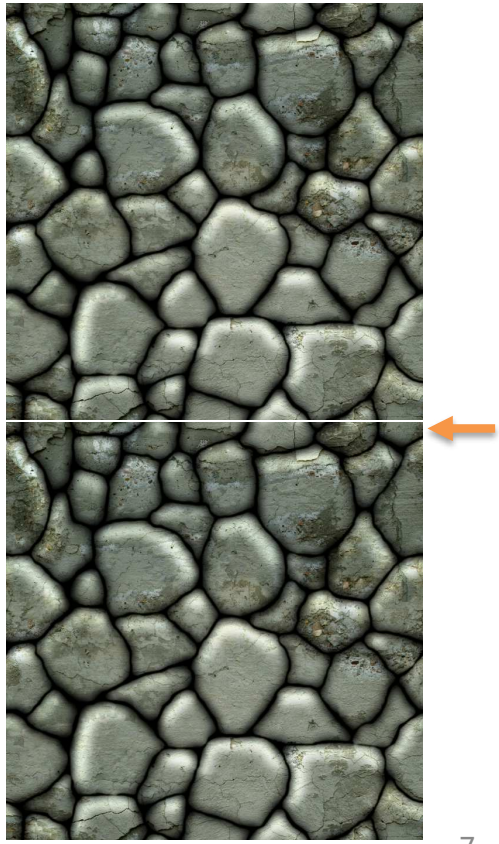
Typically need to repeat texture over the object

- must make sure there are no seams when texture is tiled









Or Given a Model and a Single Texture

Wrap the Texture onto the Model

+

=

Sample model from www.cyberware.com

Framework for Texture Mapping

The texture itself is just a 2-D raster image

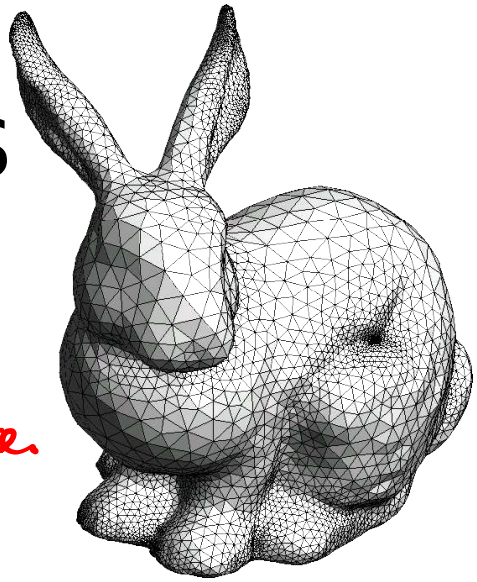
- acquired from reality, hand-painted, or procedurally generated

Establish a correspondence between surface points & texture

When shading a particular surface point

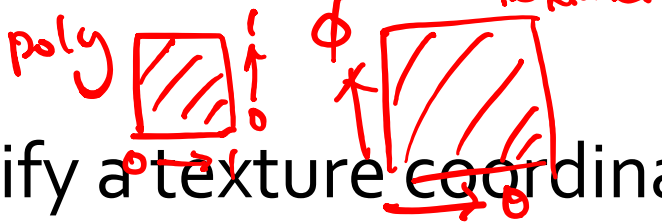
- look up the corresponding pixel in the texture image
- final color of surface point will be a function of this pixel

Texturing Polygonal Models



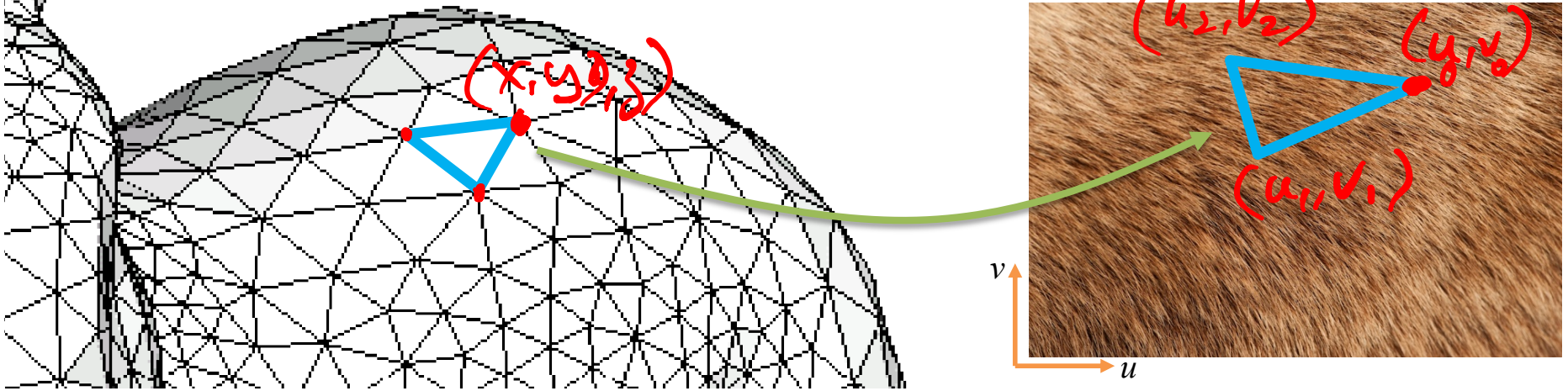
Polygonal models are not so easy

- they don't have a natural 2-D parameterization
- we need to create one

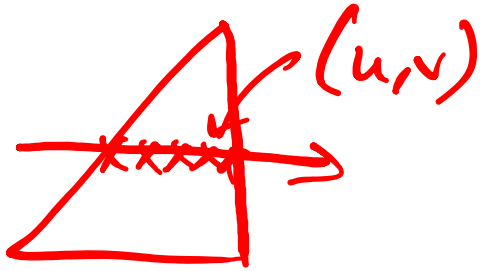


For each vertex, we specify a texture coordinate

- a (u,v) pair that maps that point into the texture image
- a triangle on the surface will be mapped to a triangle in texture
- can interpolate texture coordinates over the triangle
- note that the size of the triangle may be quite different

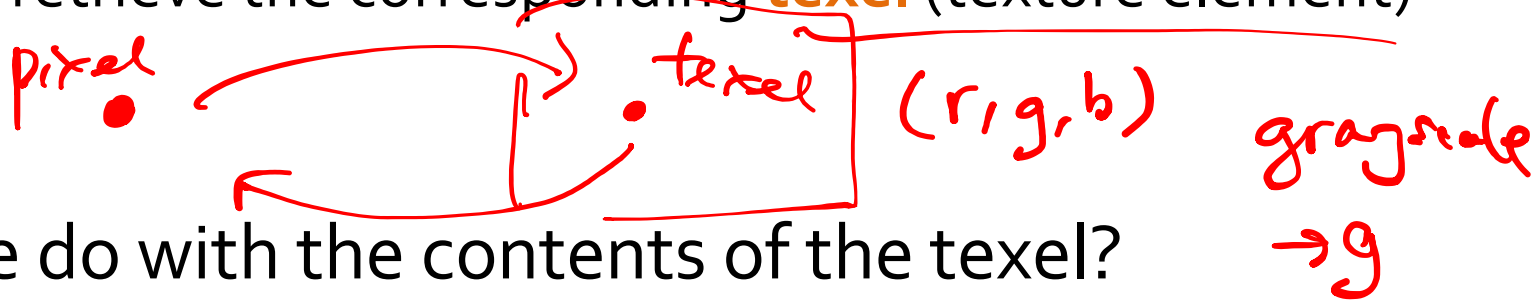


Texturing and Rasterization



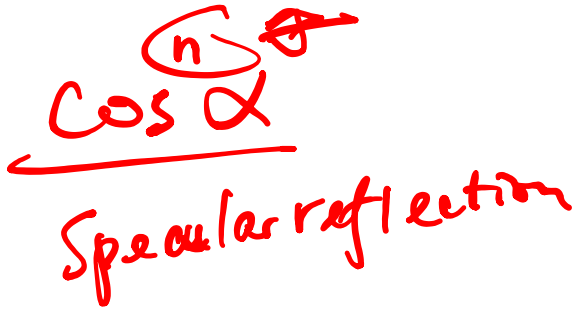
During rasterization, we traverse the pixels of a triangle

- at each pixel we interpolate the correct texture coordinate
- and we retrieve the corresponding **texel** (texture element)



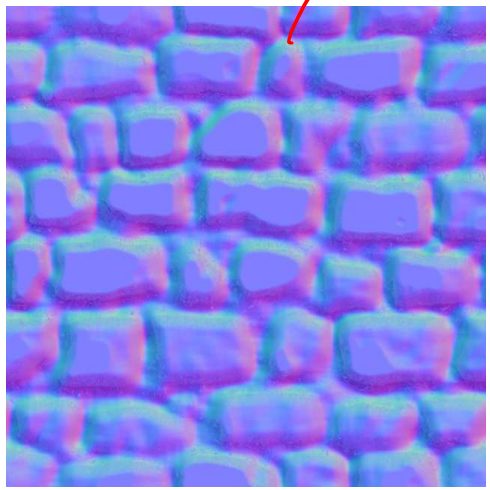
What do we do with the contents of the texel?

- color — use it to fill in the current pixel
- reflectance — coefficient for illumination equation (e.g., k_d)
- transparency — an alpha value
- and many others

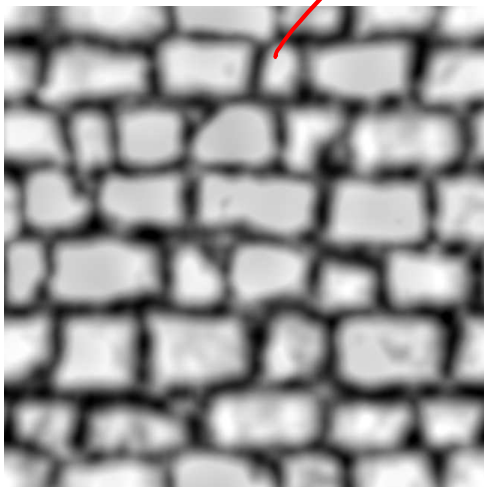


Examples of Other Mappings

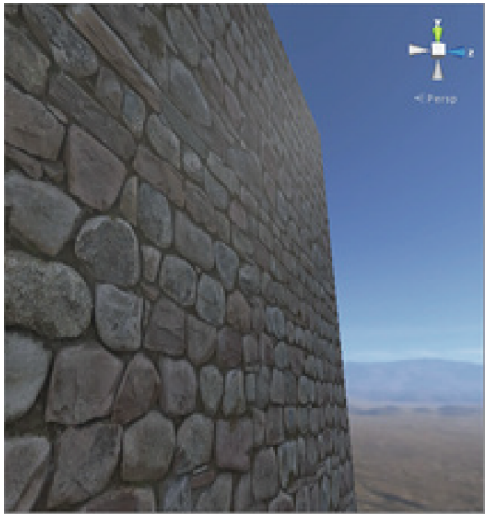
Texture map
texel as diffuse color



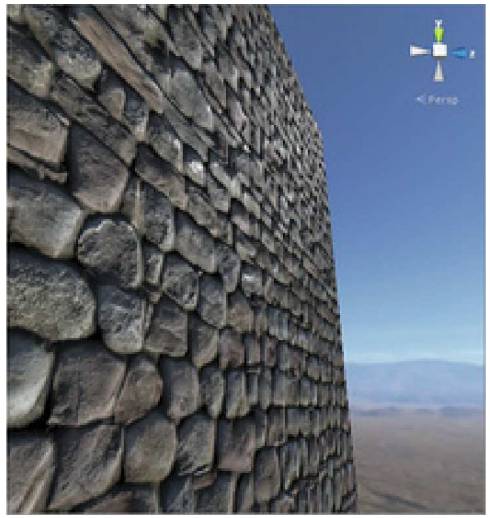
Normal map
texel as normal vectors



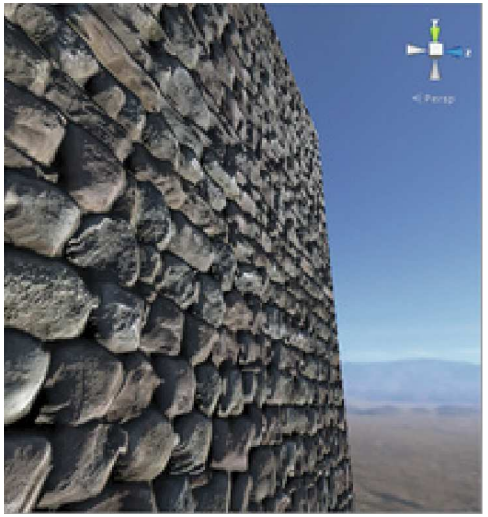
Displacement map
texel as distance from surface



Texture map only



Texture map + normal map



Texture map + normal map
+ displacement map

[Unity]

Texturing with OpenGL

First, turn on texturing — glEnable(GL_TEXTURE_2D)

Next, pass the actual texture image to OpenGL

- glTexImage2D(GL_TEXTURE_2D, *level*, *channels*, *width*, *height*, *border*, *format*, *type*, *image*)
- for now, *level*=0 and *border*=0
- *channels* is usually 3 (RGB)
  - with *format*=GL_RGB and *type*=GL_UNSIGNED_BYTE

Have lots of options to control texturing behavior

- see glTexEnvf() and glTexParameterf() for details
  - how is the color of the texture applied to the surface?
  - texture coordinates clamped to [0,1] or do they wrap around?

OpenGL Texture Modes

How is texture color applied to surface point?

OpenGL Texture modes:

- Determines how the contents of the texture are interpreted

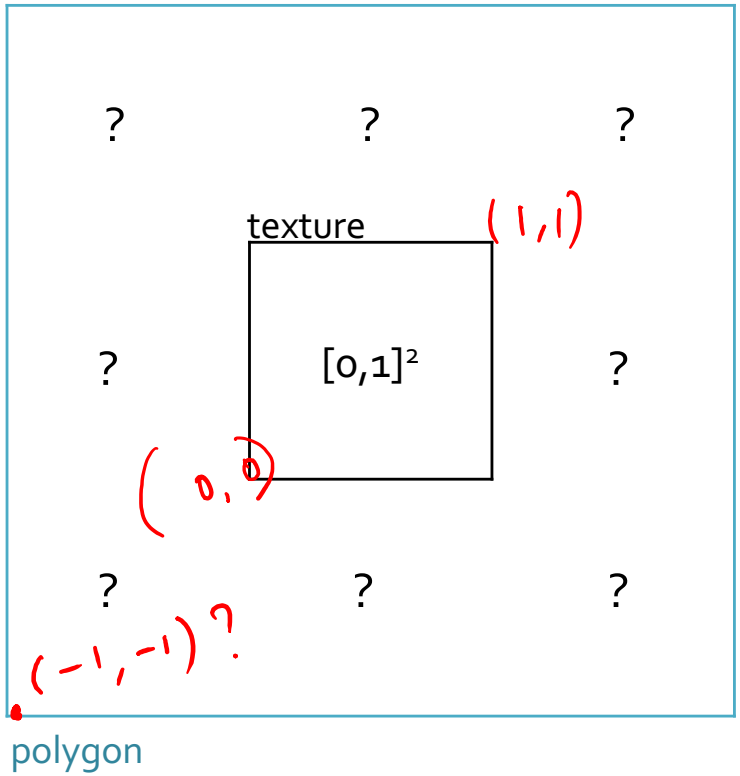
For RGB images:

GL_MODULATE — multiply together with surface color $p = p * t$

GL_BLEND — use as a *t* value to blend surface color and a predetermined color $p = p + t * c$

GL_DECAL and GL_REPLACE — use texture color directly $p = t$

OpenGl Texture Repeat Mode



GL_REPEAT

GL_MIRRORED_REPEAT

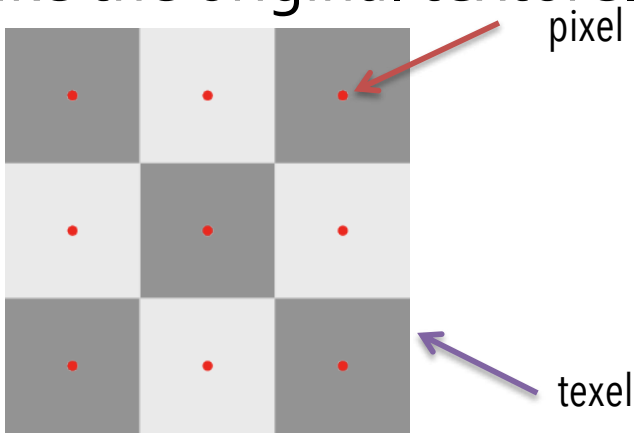
GL_CLAMP_TO_EDGE

GL_CLAMP_TO_BORDER

https://open.gl/textures

Texture Sampling

A 256x256 texture on a square that occupies 256x256 pixels on screen looks almost like the original texture.



What if the square covers a screen area of ten times as many pixels as the texture (i.e., a texel covers multiple pixels)?

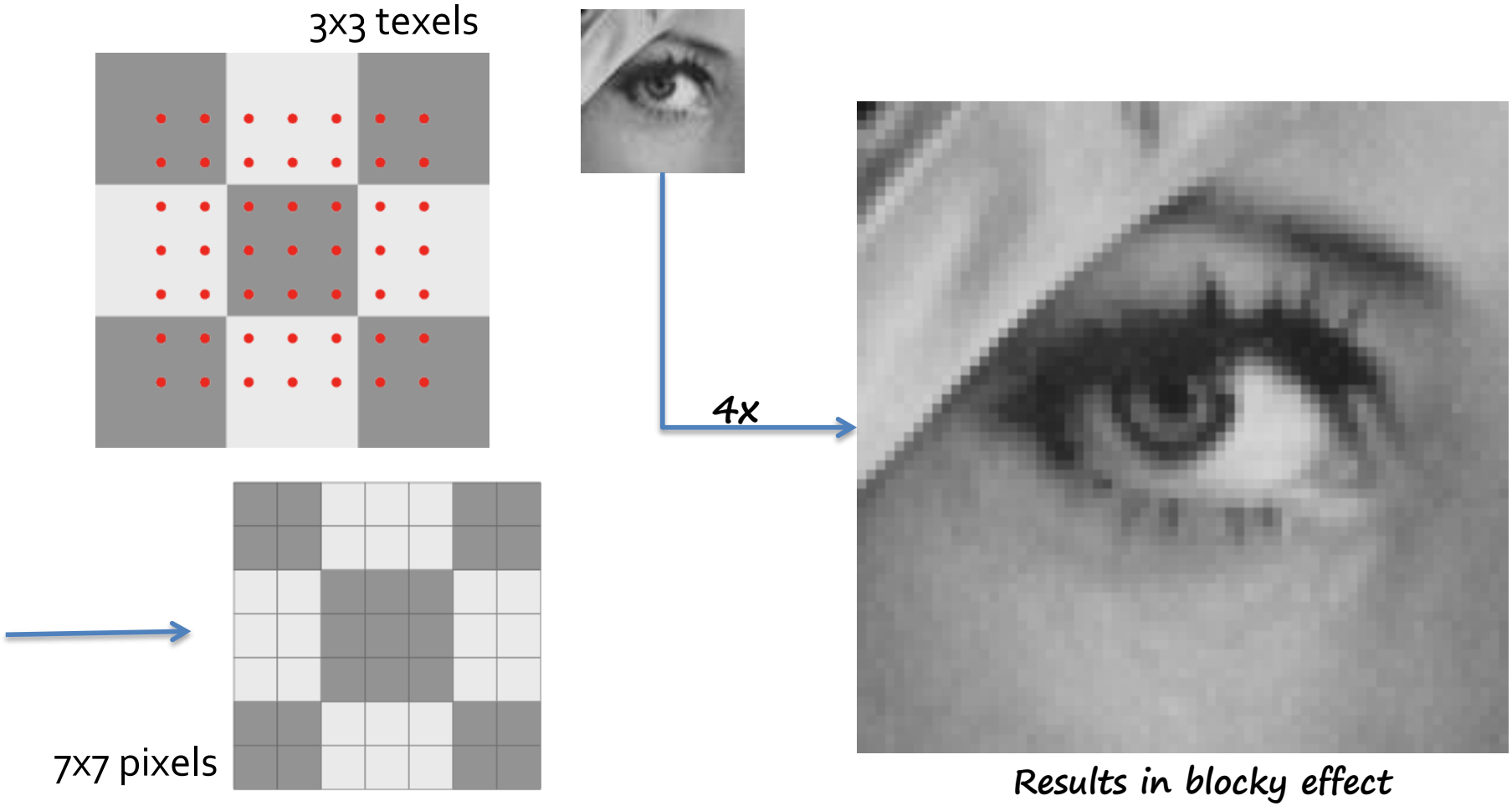
texture magnification

What if the square covers a screen area of only a few pixels (i.e., a pixel covers multiple texels)?

texture minification

Texture Magnification

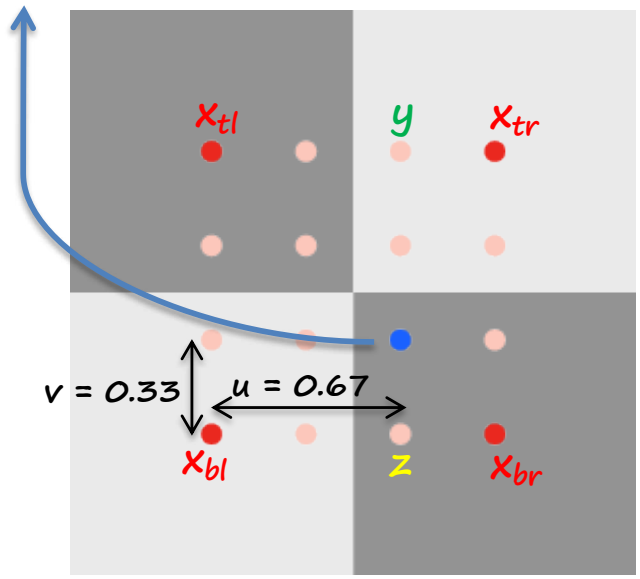
Nearest neighbor: select the nearest texel to each pixel



Texture Magnification

Bilinear interpolation: finds the four neighboring texels and performs linear interpolation in two dimensions

What should be the texture color for this pixel **p**?



1. Determine the four neighbour texels: $\mathbf{x_{tl}}$, $\mathbf{x_{tr}}$, $\mathbf{x_{bl}}$, $\mathbf{x_{br}}$
2. Determine the interpolating parameter (**u**, **v**)
3. Let **t(x)** be the color of a pixel **x**. Then we have

   $t(y) = (1-u) * t(x_{tl}) + u * t(x_{tr})$

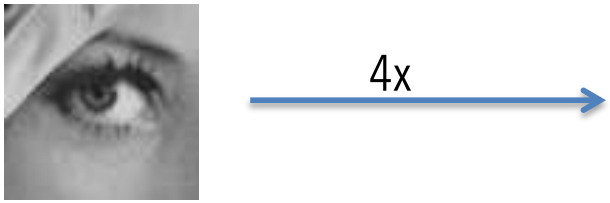
   $t(z) = (1-u) * t(x_{bl}) + u * t(x_{br})$

   and

   $t(p) = (1-v) * t(z) + v * t(y)$

Texture Magnification

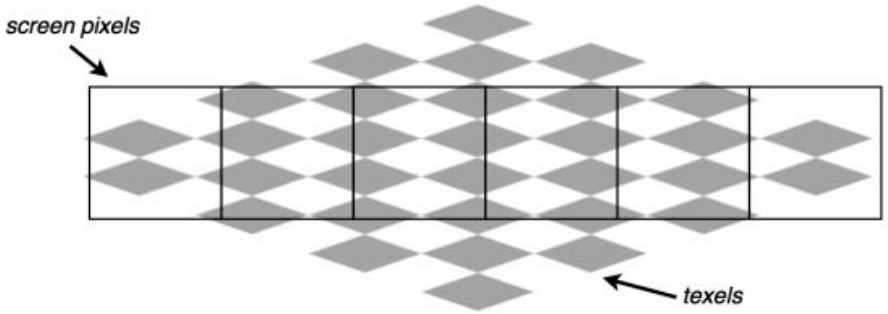
Bilinear interpolation: finds the four neighboring texels and performs linear interpolation in two dimensions



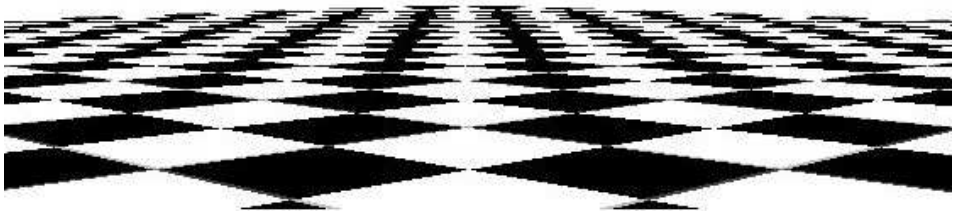
Bilinear interpolation results in a blurred image

Texture Minification

When a texture is minimized, a pixel is covered by several (fractional) texels, but we're only selecting one of them to use in the pixel



Using nearest neighbor sampling and bilinear interpolation leads to severe **aliasing** (i.e., jagged edges) problems.



nearest neighbor sampling

bilinear interpolation works only slightly better

[image from: www.adobe.com/devnet/flashplayer/articles/mipmapping.html]

Texturing with OpenGL

Here's an example setup

```
glEnable(GL_TEXTURE_2D);

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);


glTexImage2D(GL_TEXTURE_2D, 0, 3, width, height, 0,
                    GL_RGB, GL_UNSIGNED_BYTE, image);
```

This configures the texturing system to

- combine (modulate) the texture color with the surface color
- wrap texture coordinates around outside unit square
- linearly average texels when "magnifying" and "minifying"

Texturing with OpenGL

When drawing, just assign texture coordinates to vertices

```
glBegin(GL_TRIANGLES);
  glNormal3fv(n1);
  glTexCoord2f(s1, t1);
  glVertex3fv(v1);

  glNormal3fv(n2);
  glTexCoord2f(s2, t2);
  glVertex3f(v2);

  glNormal3fv(n3);
  glTexCoord2f(s3, t3);
  glVertex3fv(v3);
glEnd();
```

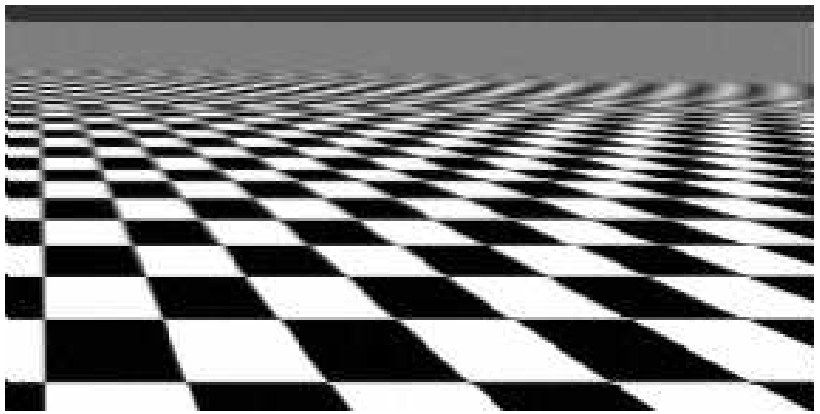
Texture Antialiasing

The antidote is to average (filter) all covered texels together

- need to choose appropriate averaging method

Removes objectionable artifacts

- but it's not magic
- very high frequency details just get smoothed over completely (e.g., gray on horizon)



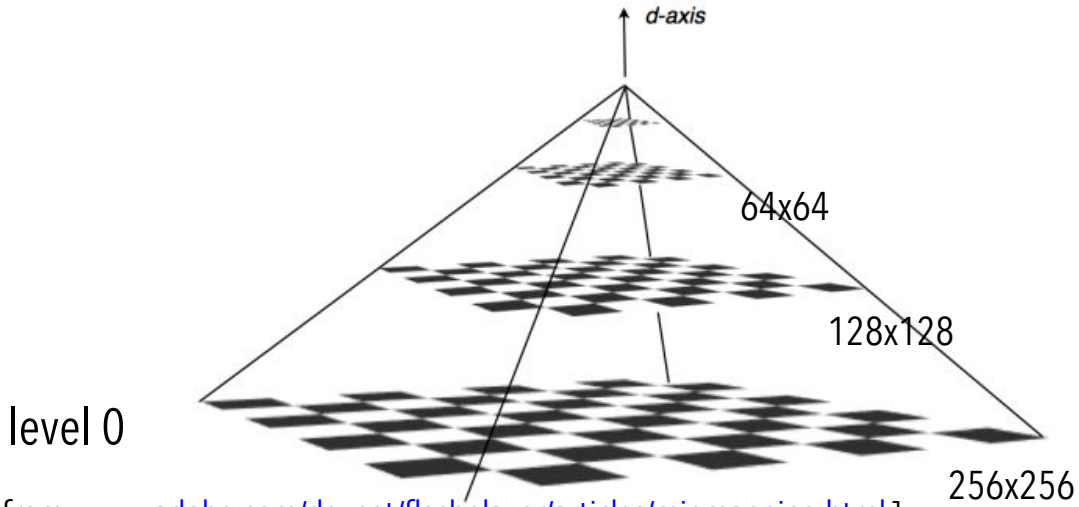
Unfortunately, there's a significant drawback here

- averaging covered texels can be very expensive
- for every pixel, might have to visit $O(n)$ texels
- this would really hurt rendering performance

Mipmaps (Image Pyramids)

An efficient method for **antialiasing** of textures.

- base of pyramid is the original image (level 0)
- level 1 is the image down-sampled by a factor of 2
- level 2 is down-sampled by a factor of 4, and so on
- requires that original dimensions be a power of 2
- total size = 4/3 the size of the original image

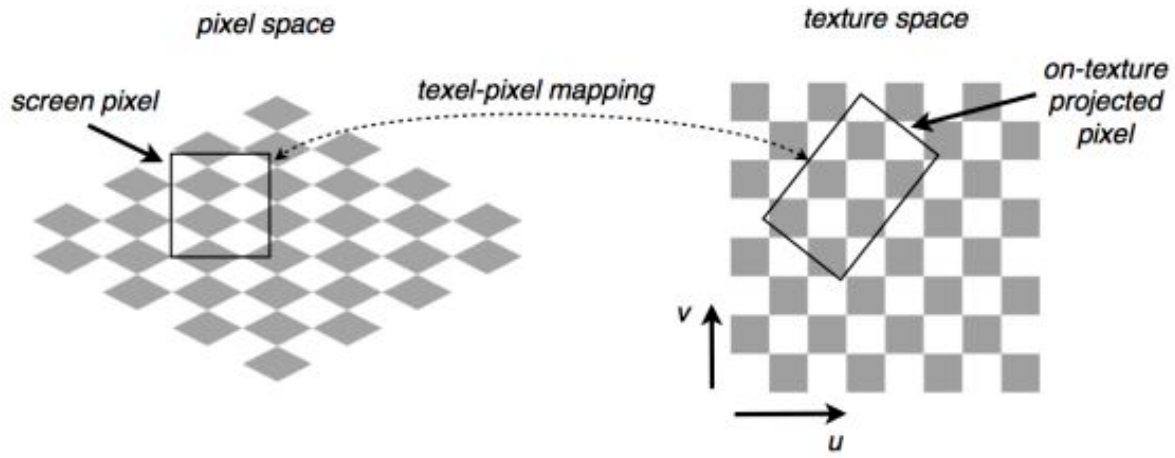


[image from: www.adobe.com/devnet/flashplayer/articles/mipmapping.html]

Mipmaps

While texturing, a screen pixel is mapped to the texture space and see how much area it covers.

If it is about the same size as a texel, the level 0 texture will be used. The larger the area it covers, the higher the texture level in the mipmap is used.



[image from: www.adobe.com/devnet/flashplayer/articles/mipmapping.html]

Antialiasing with Mipmaps

Mipmaps let us efficiently average large regions

- each texel in upper levels covers many base texels
  - at level k they are the average of 2^kx2^k texels
- can quickly assemble appropriate texels for averaging

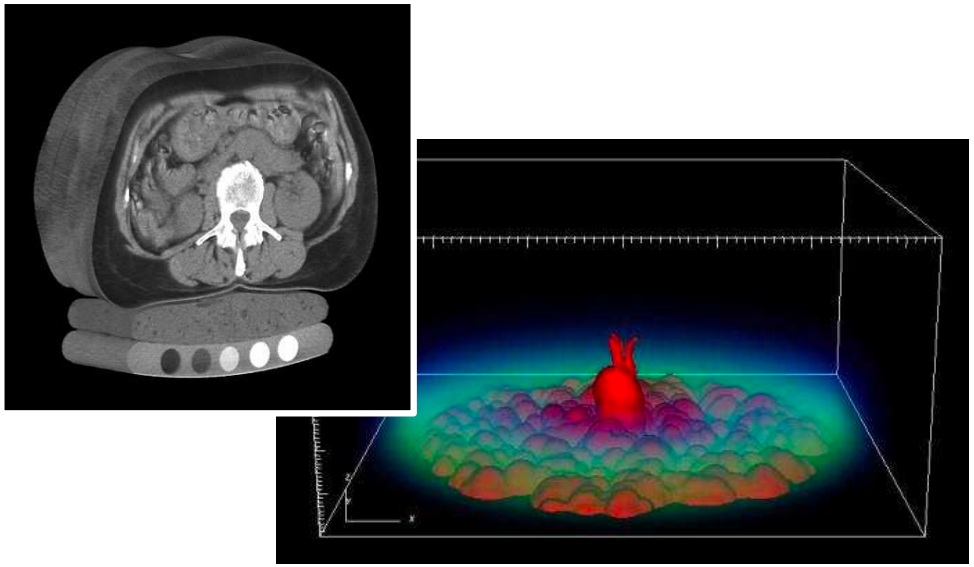
Fortunately, OpenGL can take care of most details

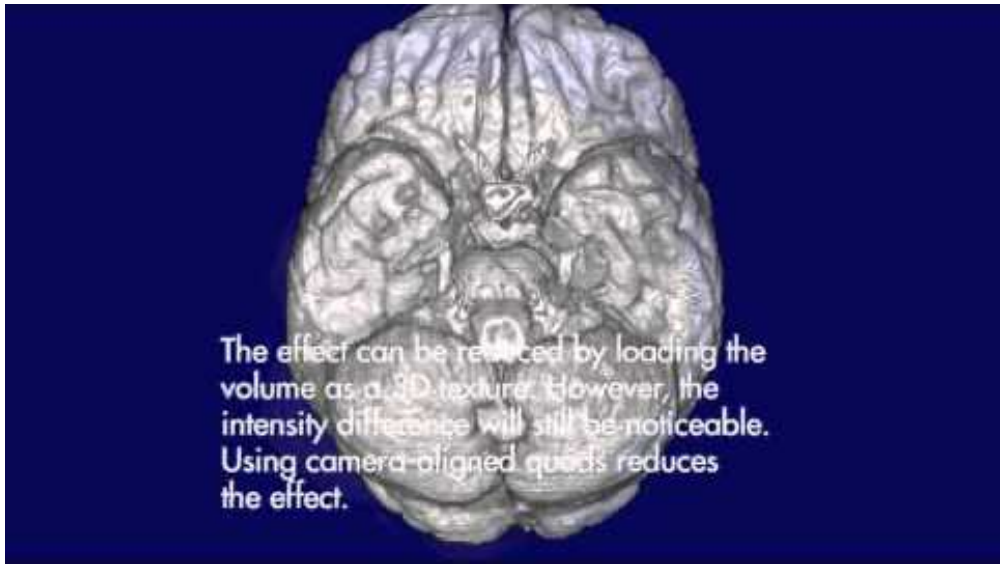
- gluBuild2DMipmaps() — automatically generate pyramid from base image
- control behavior with glTexParameter()
- OpenGL handles all filtering details during rasterization

Solid Texture

Instead of texture images, we can define texture volumes

- create a 3-D parameterization (u,v,w) for the texture map this onto the object
- the easiest parameterization $(u,v,w) = (x,y,z)$





https://youtu.be/FG-43pYZCZE

Some Texturing Applications

First, there's the obvious one: realistic surface detail

- paste a fur, marble, face scan, … on a surface

We can also support illumination precalculation

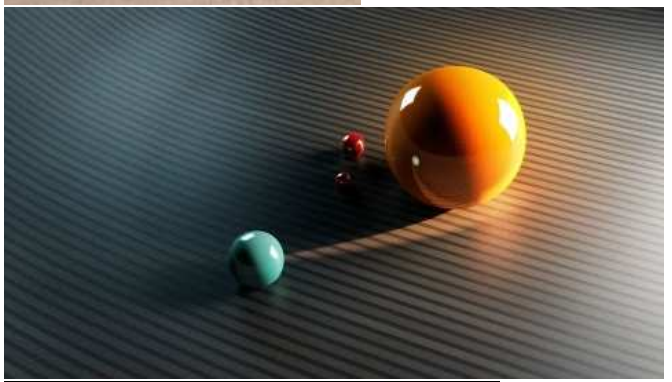
- suppose we precompute some expensive lighting effects
  - soft shadows, indirect light (e.g., radiosity)
- can hard code this lighting into texture maps

Texturing can also be handy for faking objects

- billboards — place image aon a polygon which always rotates to face the viewer (handy for things like trees)
- sprites used in video games are a similar idea

And texturing is useful for level of detail management

- can decouple resolution of texture from resolution of surface



https://youtu.be/PeogP2uIEZU
https://youtu.be/di16x6ovABY

COMP3271 Computer Graphics

Curves & Surfaces (I)

2019-20

Objectives

Different representations for curves and surfaces

Design criteria

Parametric curves & surfaces

Interpolation

Escaping Flatland

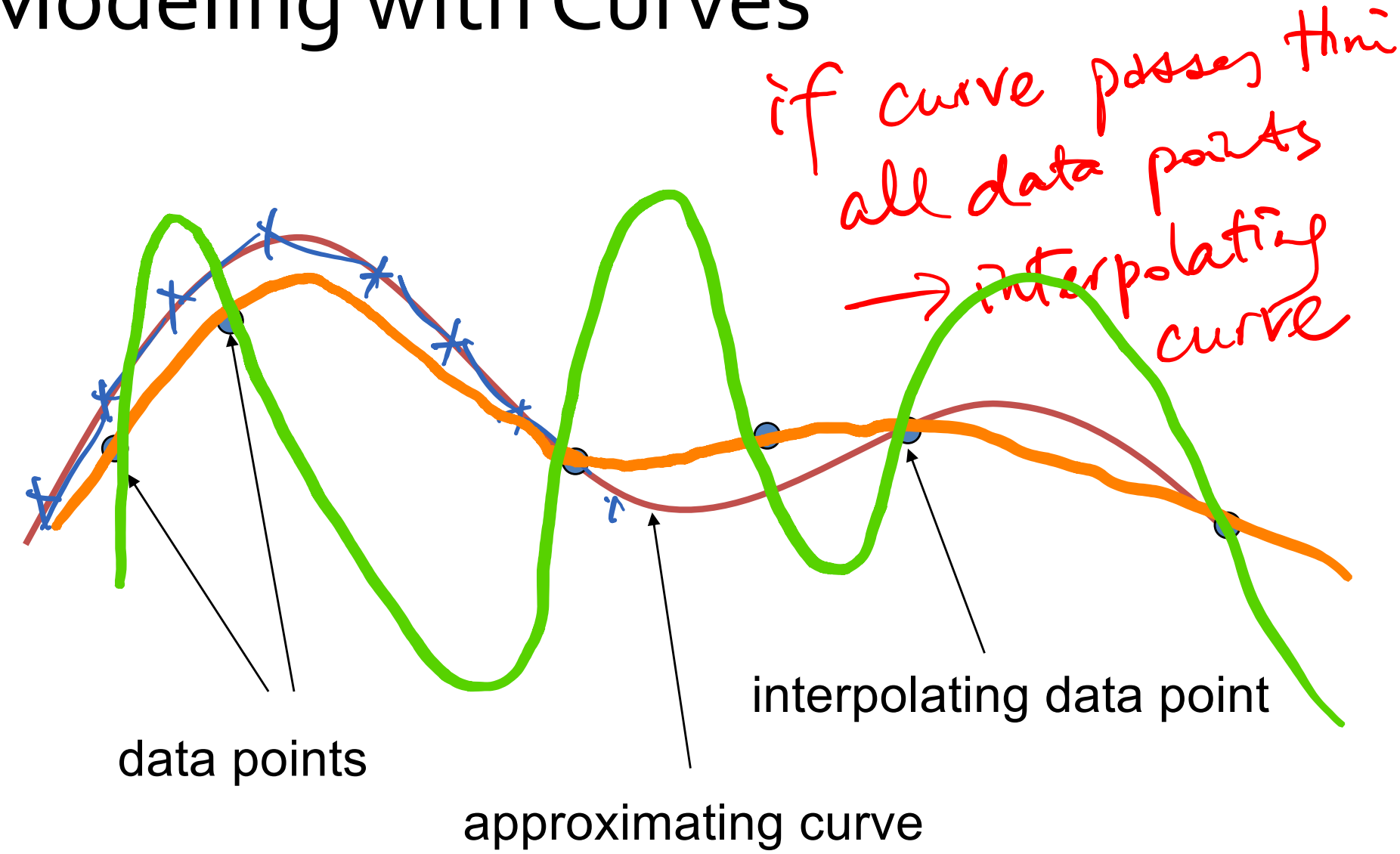
Until now we have worked with flat entities such as lines and flat polygons

- Fit well with graphics hardware
- Mathematically simple

But the world is not composed of flat entities

- Need curves and curved surfaces
- Implementation can render them approximately with flat primitives

Modeling with Curves



What Makes a Good Representation?

There are many ways to represent curves and surfaces

Some design criteria

- Local control of shape
- Stability
- Smoothness and continuity (in terms of derivatives)
- Ability to evaluate derivatives
- Ease of evaluation
- Ease of rendering
- Must we interpolate or can we just come close to data?

Explicit Representation

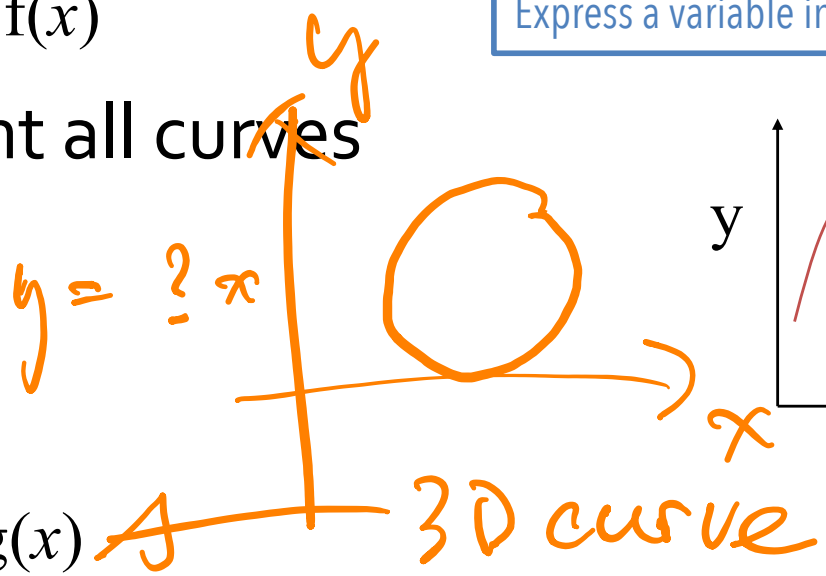
Most familiar form of curve in 2D

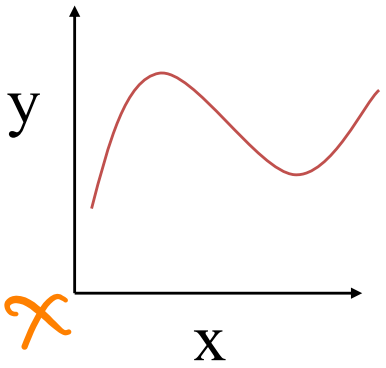
$$y = f(x)$$

Express a variable in terms of other variables

Cannot represent all curves

- Vertical lines
- Circles

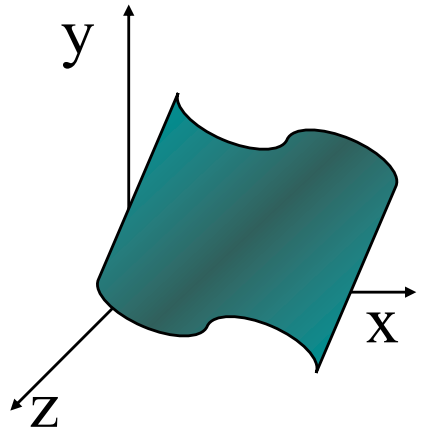




Extension to 3D

- $y = f(x), z = g(x)$
- The form $z = f(x,y)$ defines a surface

Cannot represent a sphere in the form of $z = f(x, y)$. Why?



Implicit Representation

Two dimensional curve(s)

$$g(x, y)=0$$

Represents the membership of points on curve

Much more robust

- All lines $ax + by + c = 0$
- Circles $x^2 + y^2 - r^2 = 0$

Not unique

- $(x^2 + y^2 - r^2)^2 = 0$ and $\sqrt{x^2 + y^2} - 1 = 0$ represent the same circle as $x^2 + y^2 - r^2 = 0$.

In general, no analytic way to solve for points that satisfy the equation

Implicit Representation

$g(x,y)=0$

Three dimensions $g(x, y, z)=0$ defines a surface

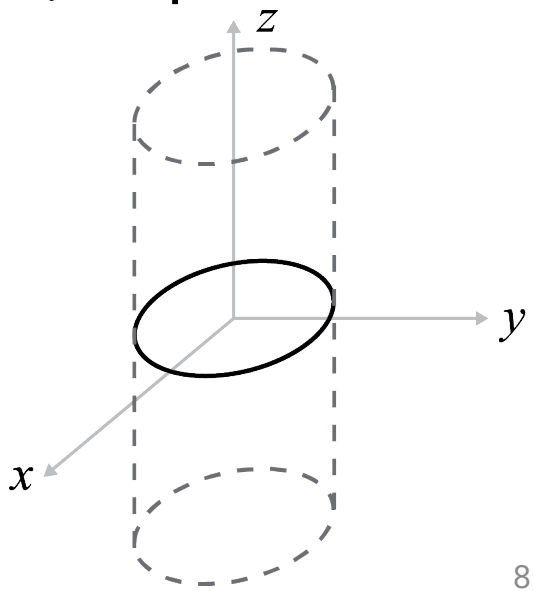
- E.g., $g(x, y, z) = x^2 + y^2 + z^2 - 1 = 0$ represents the unit sphere

Question: how to represent the unit circle centered at the origin in the xy-plane implicitly in the xyz-space?

To represent a 3D curve

- Intersect two surfaces to get a curve

$$\begin{cases} \text{cylinder} & x^2+y^2-1=0 \\ \text{plane} & z=0 \end{cases}$$

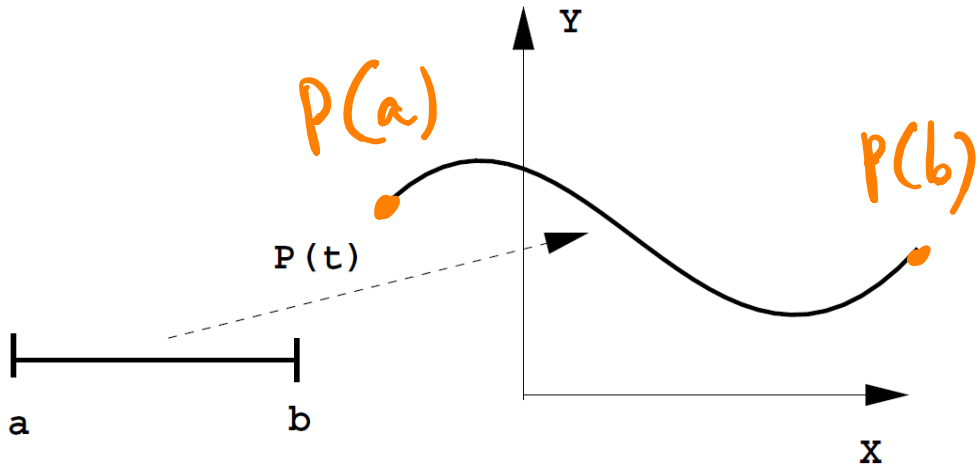


Parametric Representation

Two dimensional curves:

$$x = x(t), y = y(t), \quad t\epsilon[a,b]$$

Express the *x*,*y* values of each point on the curve explicitly in terms of an independent variable, *t*, i.e., the **parameter**, with a domain [a,b]



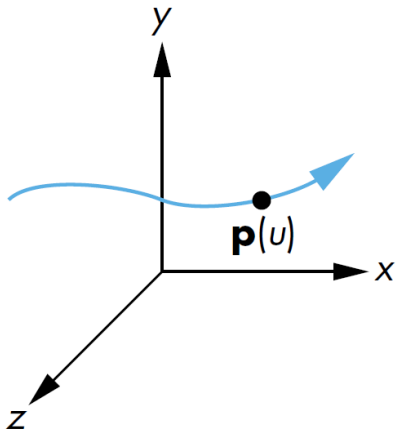
Example: Unit circle

$$P(\theta) = \big(x(\theta), y(\theta)\big) = (\cos\theta\,, \sin\theta)\,, \theta\epsilon[0,2\pi)$$

Easily extended to three dimensional curves:

$$x = x(u), y = y(u), z = z(u), \quad u\epsilon[a,b]$$

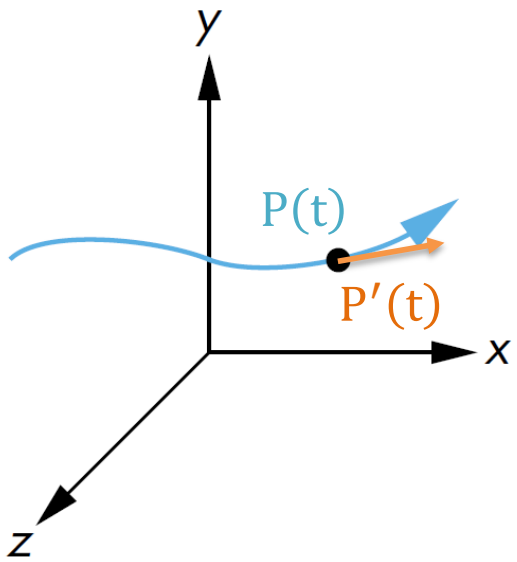
Still in one parameter, hence a curve



Parametric Representation

We trace the curve $P(t) = (x(t), y(t), z(t))$ as t varies. Hence, we can talk of the velocity of $P(t)$:

$$\mathrm{P}'(\mathrm{t}) = \frac{dP(t)}{dt} = \begin{bmatrix} \frac{dx(t)}{dt} \\ \frac{dy(t)}{dt} \\ \frac{dz(t)}{dt} \end{bmatrix}$$



This gives the tangent direction of the curve.

The speed of $P(t)$ is then $|\mathrm{P}'(\mathrm{t})|$.

Parametric Representation

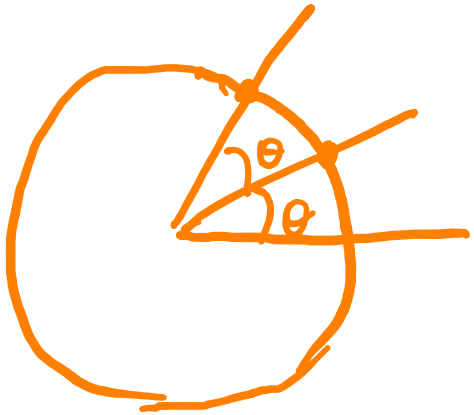
When the speed of $P(t)$ is constant or nearly constant, the computed points $P(t_i)$ on $P(t)$ are evenly or nearly evenly spaced if the parameters t_i, $i = 0, 1, 2, \ldots$, are evenly sampled.

Example. The following parametric equation of the unit circle has a constant speed.

$$P(\theta) = \big(x(\theta), y(\theta)\big) = (\cos\theta, \sin\theta), \theta \in [0, 2\pi)$$

$P'(\theta) = (-\sin\theta, \cos\theta)$

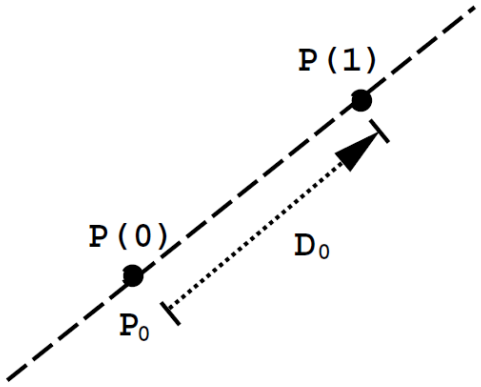
$|P'(\theta)| = 1$

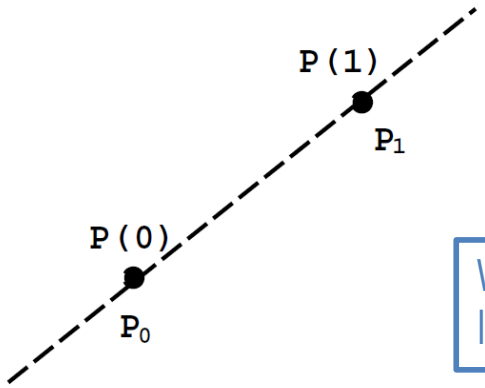


Parametric Lines

Example. A straight line passing through the point P_0 with the direction vector D_0 can be represented by

$$P(t) = \big(x(t), y(t)\big) = P_0 + tD_0, \ t \in \ (-\infty, \infty).$$





With $t \in [0,1]$, we get the straight line segment P_0 and P_1

Example. A straight line passing through two distinct points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$ is commonly represented by

$$P(t) \ = \ (1-t)P_0 \ + \ tP_1, t \in \ (-\infty, \infty).$$

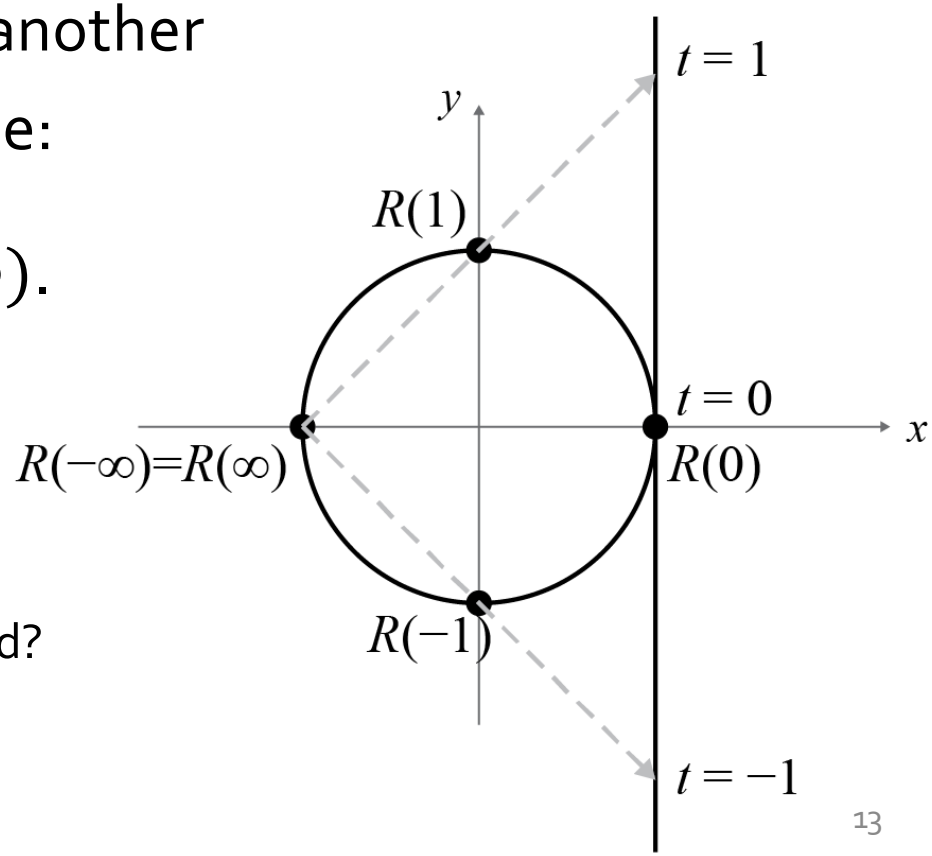
Unit Circle in Parametric Form

$$P(\theta) = \big(x(\theta), \mathrm{y}(\theta)\big) = (\cos\theta\,, \sin\theta)\,, \theta\epsilon[0,2\pi)$$

Since $\cos\theta = \frac{1-\tan^2(\theta/2)}{1+\tan^2(\theta/2)}$ and $\sin\theta = \frac{2\tan(\theta/2)}{1+\tan^2(\theta/2)}$

Substituting $t = \tan\frac{\theta}{2}$, we have another parametric from for the unit circle:

$$R(t) = \left(\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2}\right), t \in (-\infty, \infty).$$



Is this parameterization with constant speed?

Parametric Surfaces

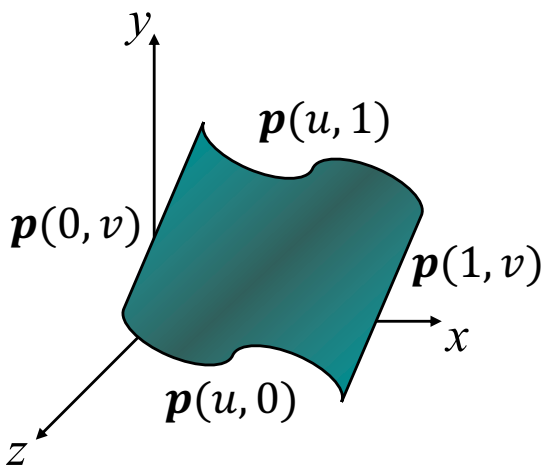
Surfaces require 2 parameters

$$x = x(u, v)$$

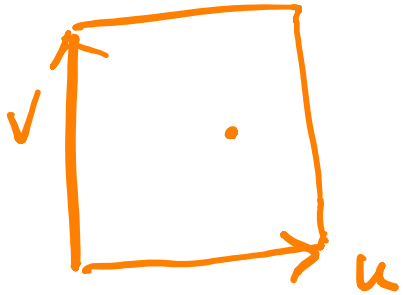
$$y = y(u, v)$$

$$z = z(u, v)$$

$$\boldsymbol{p}(u, v) = [x(u, v), y(u, v), z(u, v)]^T$$



the four boundary curves of a patch



Want same properties as curves:

- Smoothness
- Differentiability
- Ease of evaluation

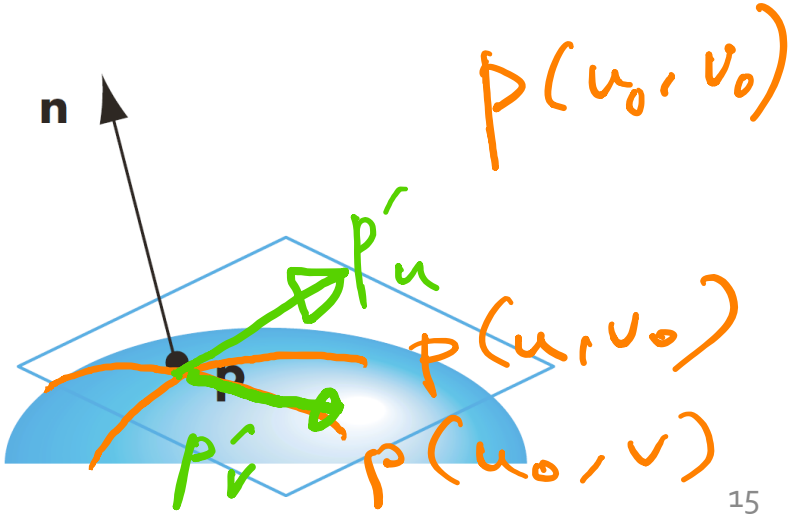
Surface Normals

We can differentiate with respect to u and v to obtain the normal at any point $\boldsymbol{p}$

$$\frac{\partial \mathbf{p}(u,v)}{\partial u} = \begin{bmatrix} \partial \mathrm{x}(u,v)/\partial u \\ \partial \mathrm{y}(u,v)/\partial u \\ \partial \mathrm{z}(u,v)/\partial u \end{bmatrix}$$

$$\frac{\partial \mathbf{p}(u,v)}{\partial v} = \begin{bmatrix} \partial \mathrm{x}(u,v)/\partial v \\ \partial \mathrm{y}(u,v)/\partial v \\ \partial \mathrm{z}(u,v)/\partial v \end{bmatrix}$$

$$\mathbf{n} = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}$$

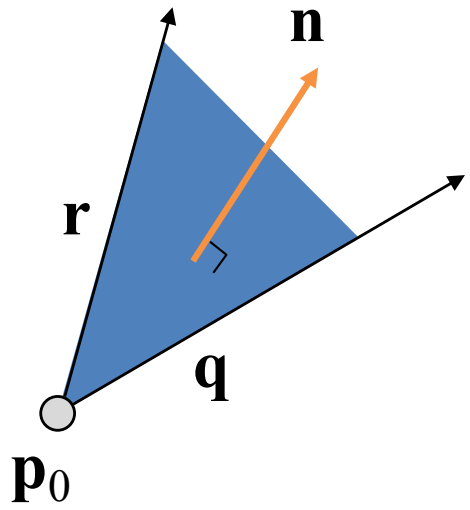


Parametric Planes

Point-vector form

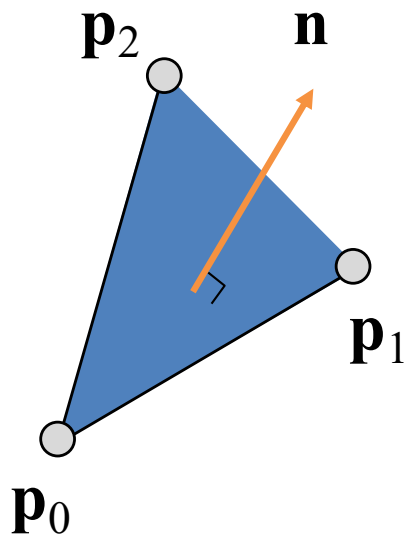
$$\mathbf{p}(u,v) = \mathbf{p}_0 + u\mathbf{q} + v\mathbf{r}$$

$$\mathbf{n} = \mathbf{q} \times \mathbf{r}$$



Three-point form

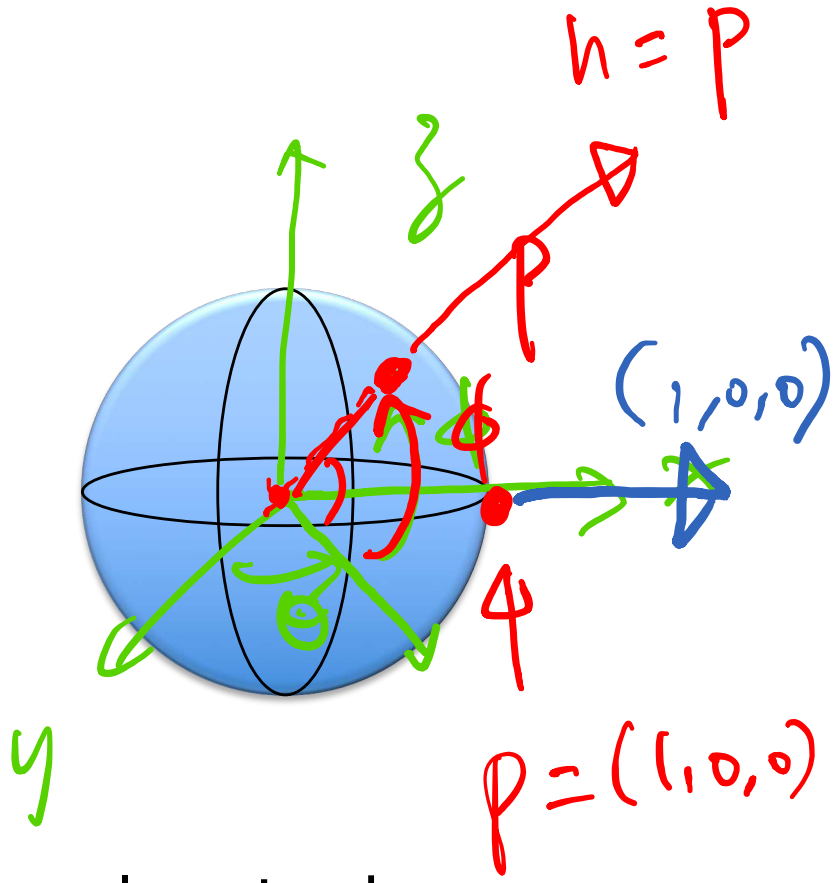
$$\mathbf{q} = \mathbf{p}_1 - \mathbf{p}_0$$
$$\mathbf{r} = \mathbf{p}_2 - \mathbf{p}_0$$



Parametric Spheres

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

$$0 \le \theta \le 2\pi$$
$$0 \le \varphi \le \pi$$



θ: constant; circles of constant longitude

φ: constant; circles of constant latitude

Exercise: differentiate to show $\mathbf{n} = \mathbf{p}$

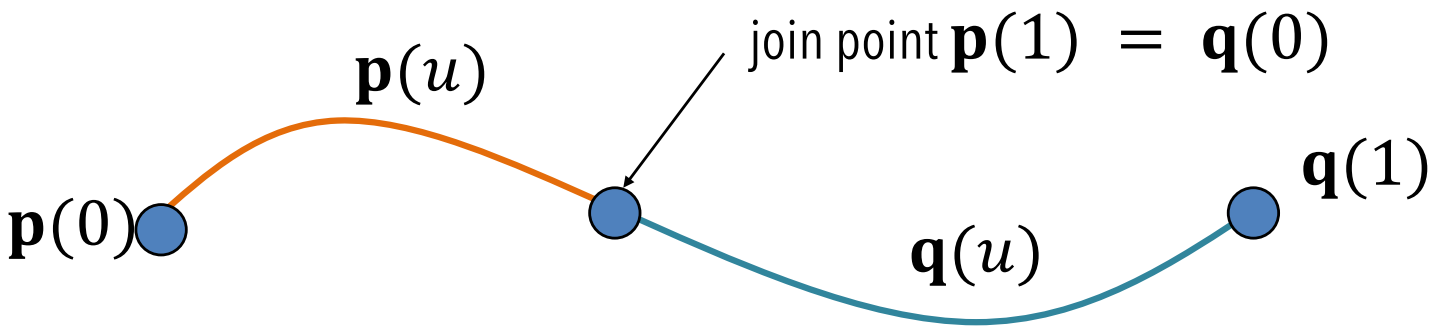
Curve Segments

After normalizing u, each curve can be written as

$$\mathbf{p}(u) = [x(u), y(u), z(u)]^T, \quad 0 \le u \le 1$$

In classical numerical methods, we design a single global curve

In computer graphics and CAD, it is better to design small connected curve segments



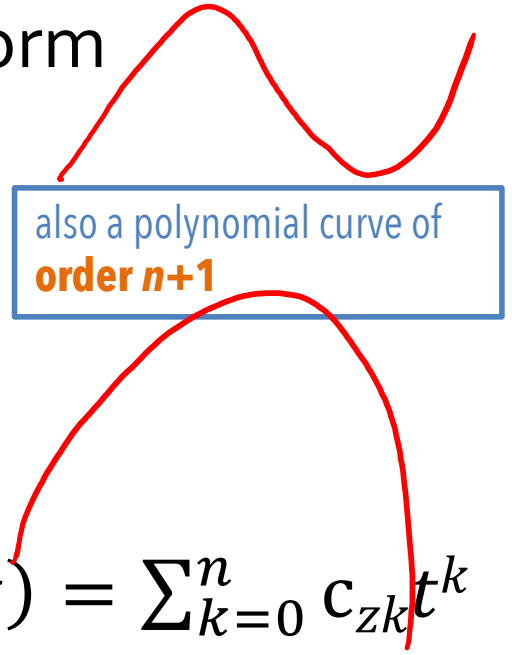
$a_n t^n + a_{n-1} t^{n-1} + \dots + a_1 t + a_0 = \sum_{i=0}^{n} a_i t^i$

Parametric Polynomial Curves

$\mathbf{p}(t) = (x(t), y(t), z(t))^T$ is called a polynomial curve if $x(t)$, $y(t)$ and $z(t)$ are polynomial functions of t.

A polynomial curve of degree n is of the form

$$\mathbf{p}(t) = \sum_{k=0}^{n} \mathbf{c}_k t^k \text{ where } \mathbf{c}_k = \begin{pmatrix} c_{xk} \\ c_{yk} \\ c_{zk} \end{pmatrix}.$$



Hence,

$$x(t) = \sum_{k=0}^{n} c_{xk} t^k,\ y(t) = \sum_{k=0}^{n} c_{yk} t^k,\ z(t) = \sum_{k=0}^{n} c_{zk} t^k$$

$t \in [0,1]$.

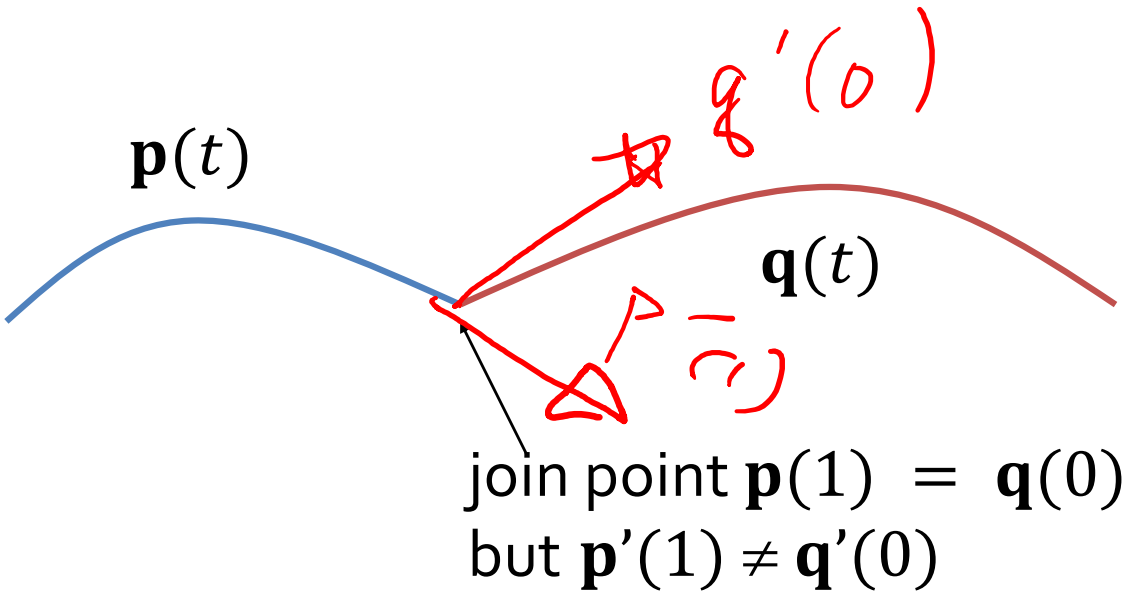
We need to determine $3(n+1)$ coefficients
Equivalently we need $3(n+1)$ independent conditions

Why Polynomials

Easy to evaluate – need only $+, -, \times, \div$ (e.g., no sin/cos)

Continuous and differentiable everywhere

- Still need to worry about continuity at join points including continuity of derivatives



Cubic Parametric Polynomials

We mostly use cubic curves which gives balance between ease of evaluation and flexibility in design

$$\mathbf{p}(t) = \sum_{k=0}^{3} \mathbf{c}_k t^k$$

$$\sum_{k=0}^{3} c_{kx} t^k$$

Four coefficients to determine for each of x, y and z

Seek four independent conditions for various values of t resulting in 4 equations in 4 unknowns for each of x, y and z

- Conditions are a mixture of continuity requirements at the join points and conditions for fitting the data

Representation in Matrix-Vector Form

$$\mathbf{p}(u) = \sum_{k=0}^{3} \mathbf{c}_k u^k = \mathbf{c}_0 + \mathbf{c}_1 u + \mathbf{c}_2 u^2 + \mathbf{c}_3 u^3$$

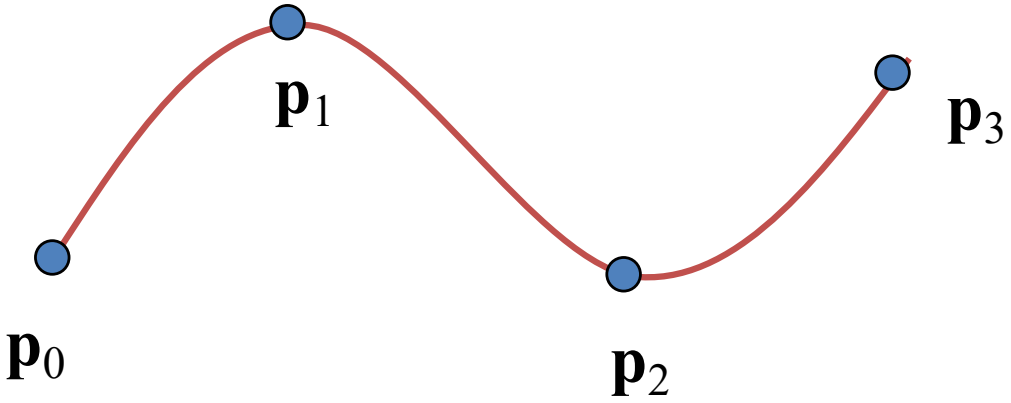
a cubic curve

$$= \mathbf{u}^T \mathbf{c} = \mathbf{c}^T \mathbf{u}$$

where

$$\mathbf{c} = \begin{pmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \mathbf{c}_3 \end{pmatrix}, \quad \mathbf{u} = \begin{pmatrix} 1 \\ u \\ u^2 \\ u^3 \end{pmatrix}, \quad \mathbf{c}_k = \begin{pmatrix} c_{kx} \\ c_{ky} \\ c_{kz} \end{pmatrix}$$

Interpolating Curve

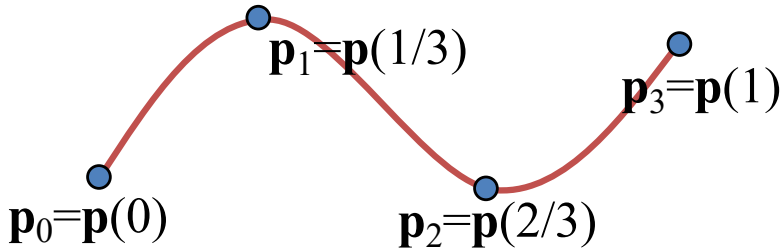


Given four data (control) points $\mathbf{p}_0$, $\mathbf{p}_1$,$\mathbf{p}_2$, $\mathbf{p}_3$
determine cubic $\mathbf{p}(u)$ which passes through them

Need to find $\mathbf{c}_0$,$\mathbf{c}_1$,$\mathbf{c}_2$, $\mathbf{c}_3$

Interpolation Equations

$$\mathbf{p}(u) = \sum_{k=0}^{3} \mathbf{c}_k u^k = \mathbf{c}_0 + \mathbf{c}_1 u + \mathbf{c}_2 u^2 + \mathbf{c}_3 u^3$$



apply the interpolating conditions at $u = 0, 1/3, 2/3, 1$

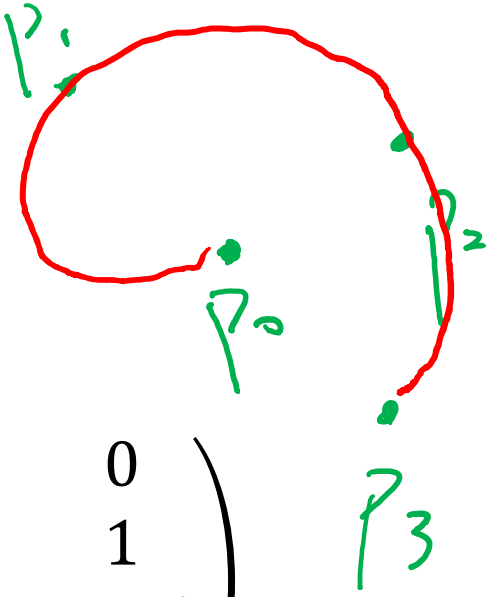
$$\mathbf{p}_0 = \mathbf{p}(0) = \mathbf{c}_0$$
$$\mathbf{p}_1 = \mathbf{p}(1/3) = \mathbf{c}_0 + (1/3)\mathbf{c}_1 + (1/3)^2\mathbf{c}_2 + (1/3)^3\mathbf{c}_2$$
$$\mathbf{p}_2 = \mathbf{p}(2/3) = \mathbf{c}_0 + (2/3)\mathbf{c}_1 + (2/3)^2\mathbf{c}_2 + (2/3)^3\mathbf{c}_2$$
$$\mathbf{p}_3 = \mathbf{p}(1) = \mathbf{c}_0 + \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_2$$

or in matrix form

$$\mathbf{p} = (\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)^T = \mathbf{Ac}, \quad \mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \left(\frac{1}{3}\right) & \left(\frac{1}{3}\right)^2 & \left(\frac{1}{3}\right)^3 \\ 1 & \left(\frac{2}{3}\right) & \left(\frac{2}{3}\right)^2 & \left(\frac{2}{3}\right)^3 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Interpolation Matrix

$$\mathbf{p} = \mathbf{A}\mathbf{c}, \qquad \mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \left(\frac{1}{3}\right) & \left(\frac{1}{3}\right)^2 & \left(\frac{1}{3}\right)^3 \\ 1 & \left(\frac{2}{3}\right) & \left(\frac{2}{3}\right)^2 & \left(\frac{2}{3}\right)^3 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

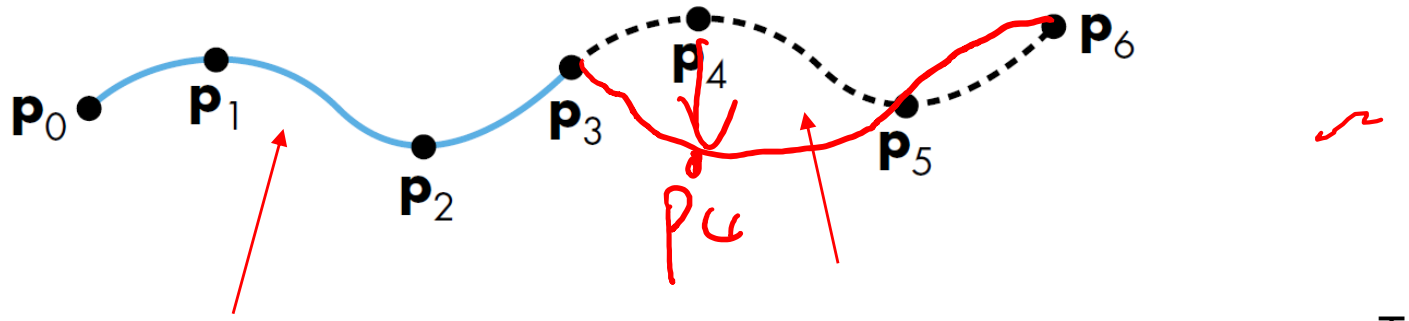


Let $\mathbf{M}_I = \mathbf{A}^{-1}$ and solving for $\mathbf{c}$,

$$\mathbf{c} = \mathbf{M}_I\mathbf{p}, \qquad \mathbf{M}_I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & 22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{pmatrix}$$

Note that $\mathbf{M}_I$ does not depend on input data and can be used for each segment in x, y, and z

Interpolating Multiple Segments



use $\mathbf{p} = [\mathbf{p}_0\ \mathbf{p}_1\ \mathbf{p}_2\ \mathbf{p}_3]^T$

use $\mathbf{p} = [\mathbf{p}_3\ \mathbf{p}_4\ \mathbf{p}_5\ \mathbf{p}_6]^T$

Get continuity at join points but not continuity of derivatives

Blending Functions

Rewriting the equation for p(u)

$$\mathbf{p}(u) = \mathbf{u}^T\mathbf{c} = \mathbf{u}^T\mathbf{M}_I\mathbf{p} = \mathbf{b}(u)^T\mathbf{p}$$

where $\mathbf{b}(u) = [b_0(u)\ b_1(u)\ b_2(u)\ b_3(u)]^T$ is an array of **blending polynomials** such that

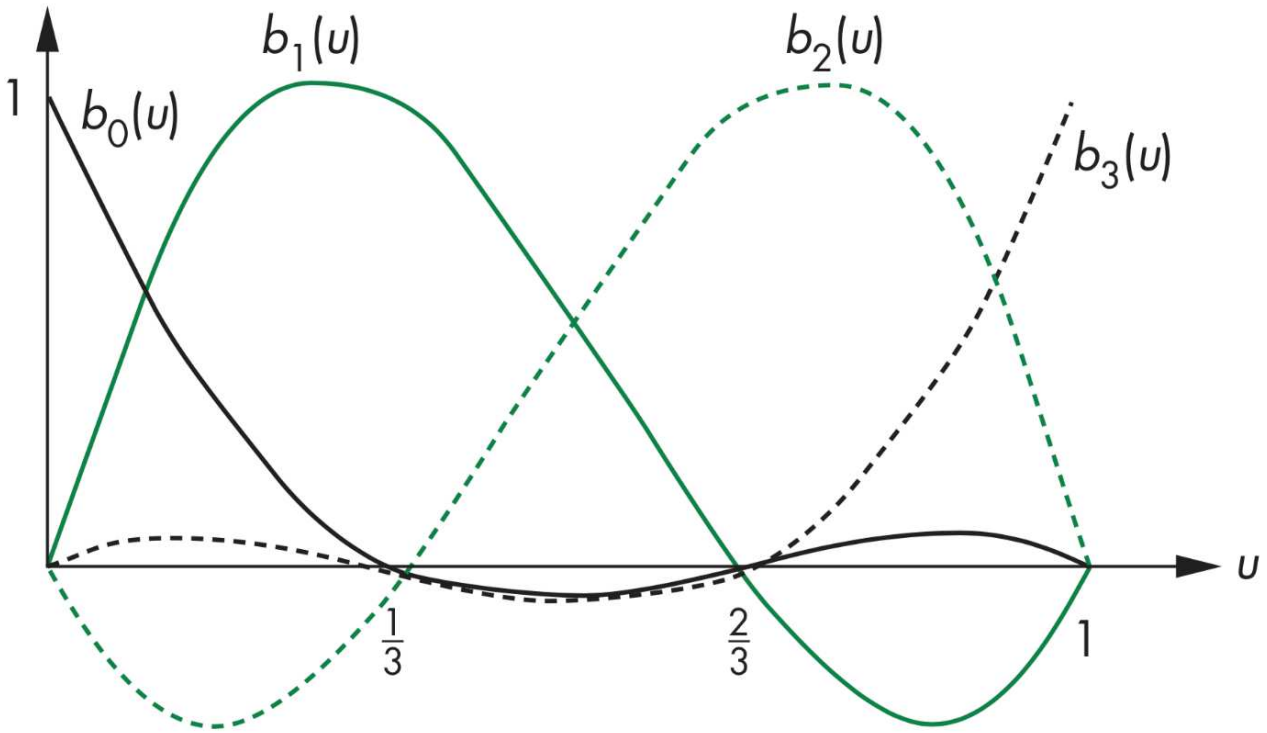
$$\mathbf{p}(u) = b_0(u)\mathbf{p}_0 + b_1(u)\mathbf{p}_1 + b_2(u)\mathbf{p}_2 + b_3(u)\mathbf{p}_3$$

$$b_0(u) = -4.5(u - \tfrac{1}{3})(u - \tfrac{2}{3})(u - 1)$$
$$b_1(u) = 13.5u\,(u - \tfrac{2}{3})(u - 1)$$
$$b_2(u) = -13.5u\,(u - \tfrac{1}{3})(u - 1)$$
$$b_3(u) = 4.5u\,(u - \tfrac{1}{3})(u - \tfrac{2}{3})$$

Blending Functions

These functions are not smooth

- Hence the interpolation polynomial is not smooth



Cubic Polynomial Surfaces

A degree $m \times n$ parametric polynomial surface is defined as

$$\mathbf{p}(u,v) = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix} = \sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{c}_{ij}\, u^i\, v^j\,, \qquad (u,v) \in [0,1]^2$$

More specifically, a cubic polynomial surface is given by

$$\mathbf{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} \mathbf{c}_{ij} u^i v^j\,, \qquad (u,v) \in [0,1]^2.$$

Need 48 coefficients (4 x 4 x 3) to determine a surface patch

Interpolating Patch

$$\mathbf{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} \mathbf{c}_{ij}\, u^i\, v^j$$

a bicubic surface

Define $\mathbf{u} = [1 \;\; u \;\; u^2 \;\; u^3]^T$, $\mathbf{v} = [1 \;\; v \;\; v^2 \;\; v^3]^T$, $\mathbf{C} = [\mathbf{c}_{ij}]$

4x4 matrix, each element is a 3-vector

We may write

$$\mathbf{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} \mathbf{c}_{ij}\, u^i\, v^j \;\; = \;\; \mathbf{u}^T\mathbf{C}\mathbf{v}$$

Interpolating Patch in Matrix Form

$$\mathbf{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} \mathbf{c}_{ij}\, u^i\, v^j \quad = \mathbf{u}^T\mathbf{C}\mathbf{v}$$

$\mathbf{u} = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}$

Need 16 conditions to determine the 16 coefficients $\mathbf{c}_{ij}$

a 3-vector

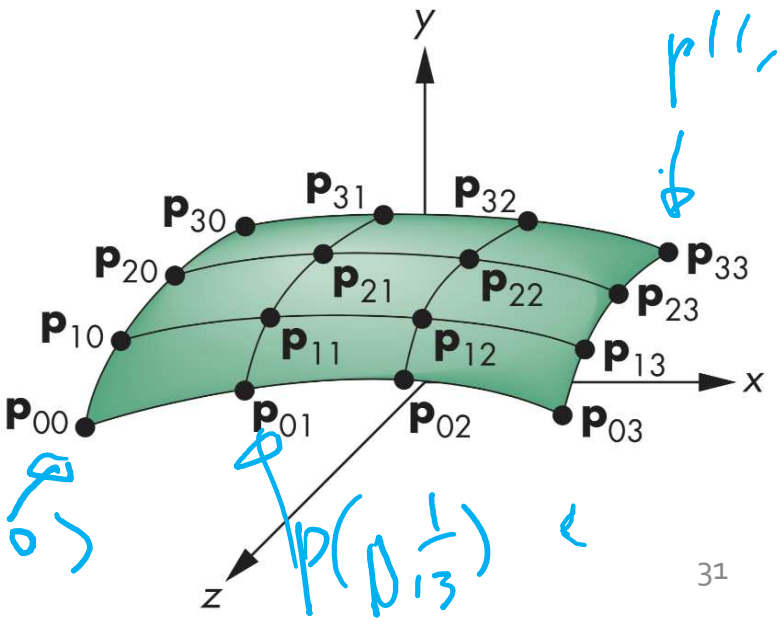
Consider $u = v = 0$, and we have

$$\mathbf{p}_{00} = [1 \quad 0 \quad 0 \quad 0]\,\mathbf{C}\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{c}_{00}$$

$u = 0$, $v = \frac{1}{3}$ $\quad p_{01} = [1\ 0\ 0\ 0]\,C\begin{bmatrix} 1 \\ \frac{1}{3} \\ (\frac{1}{3})^2 \\ (\frac{1}{3})^3 \end{bmatrix}$

Choose $u, v = 0, \frac{1}{3}, \frac{2}{3}, 1$

and we have the 16 conditions



$p(1,1)$ if

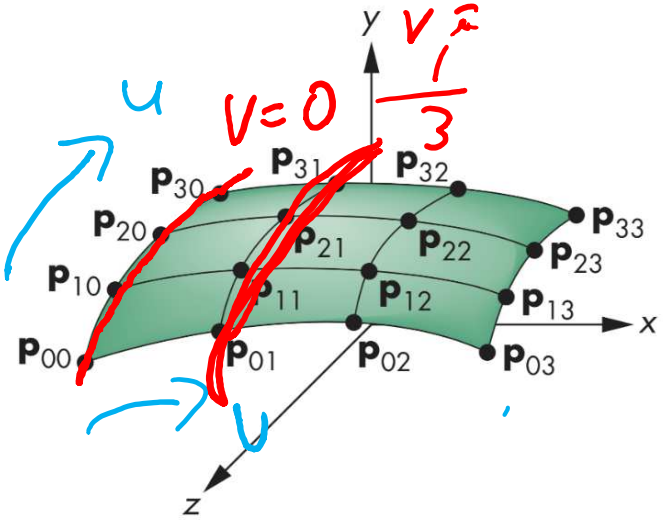
$p(0,0)$ $\quad p(0, \frac{1}{3})$

Interpolating Patch in Matrix Form

Instead of writing the 16 equations one by one and solve for $\mathbf{C}$, let's consider $v = 0$,

$$\mathbf{p}(u,0) = \sum_{i=0}^{3} \mathbf{c}_{i0} u^i = \mathbf{u}^T \mathbf{C} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{u}^T \mathbf{M}_I \begin{bmatrix} \mathbf{p}_{00} \\ \mathbf{p}_{10} \\ \mathbf{p}_{20} \\ \mathbf{p}_{30} \end{bmatrix}$$

A cubic curve interpolating $\mathbf{p}_{00}$, $\mathbf{p}_{10}$, $\mathbf{p}_{20}$, $\mathbf{p}_{30}$



Doing the same for $v = \frac{1}{3}, \frac{2}{3}, 1$, and we have 3 more equations similar to the above. Hence we have

$$\mathbf{u}^T \mathbf{C} \mathbf{A}^T = \mathbf{u}^T \mathbf{M}_I \mathbf{P} \quad \text{where } \mathbf{P} = [\mathbf{p}_{ij}] \quad \text{and therefore } \mathbf{C} = \mathbf{M}_I \mathbf{P} \mathbf{M}_I{}^T$$

$\mathbf{M}_I = \mathbf{A}^{-1}$

The bicubic surface is given by $\mathbf{p}(u,v) = \mathbf{u}^T \mathbf{M}_I \mathbf{P} \mathbf{M}_I{}^T \mathbf{v}$

Blending Patches

We can build and analyze surfaces from our knowledge of curves

$$\mathbf{p}(u,v) = \mathbf{u}^T\mathbf{M}_I\mathbf{P}\mathbf{M}_I{}^T\mathbf{v}$$

Note that $\mathbf{M}_I\mathbf{u}^T$ are the blending functions

$$\mathbf{b}(u) = [b_0(u)\ \ b_1(u)\ \ b_2(u)\ \ b_3(u)]^T$$

Hence we may rewrite the surface equation as

$$\sum_{i=0}^{3}\sum_{j=0}^{3} b_i(u)b_j(v)\mathbf{p}_{ij}$$

Each $b_i(u)\ b_j(v)$ describes a blending patch

COMP3271 Computer Graphics

Curves & Surfaces (II)

2019-20

Objectives

Introduce the Bézier curves and surfaces

Derive the required matrices

Other Types of Curves and Surfaces

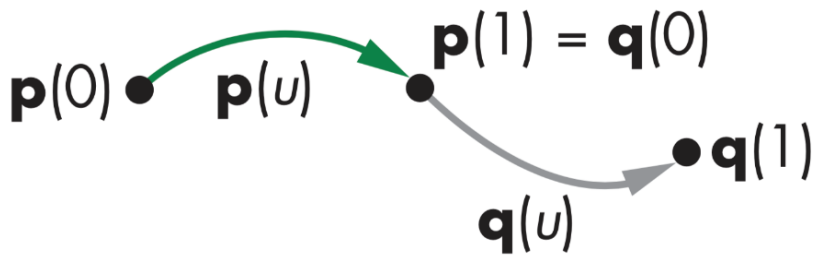
How can we get around the limitations of the interpolating form

- Lack of smoothness
- Discontinuous derivatives at join points

We have four conditions (for cubics) that we can apply to each segment

- Not necessarily using interpolating points as conditions
- Need only come close to the data

Parametric and Geometric Continuity



If $\mathbf{p}(1) = \mathbf{q}(0)$, then we have C^0 parametric continuity.

If $\mathbf{p}'(1) = \mathbf{q}'(0)$, i.e., each of the derivatives of x, y,and z components are continuous at join points, then we have C^1 parametric continuity.

Parametric and Geometric Continuity

Or we can only require that the tangents of the resulting curve be continuous, i.e., $\mathbf{p}'(1) = \alpha\mathbf{q}'(0)$ then we have G^1 geometric continuity)

This gives more flexibility then C^1 continuity as we need satisfy only two conditions rather than three at each join point

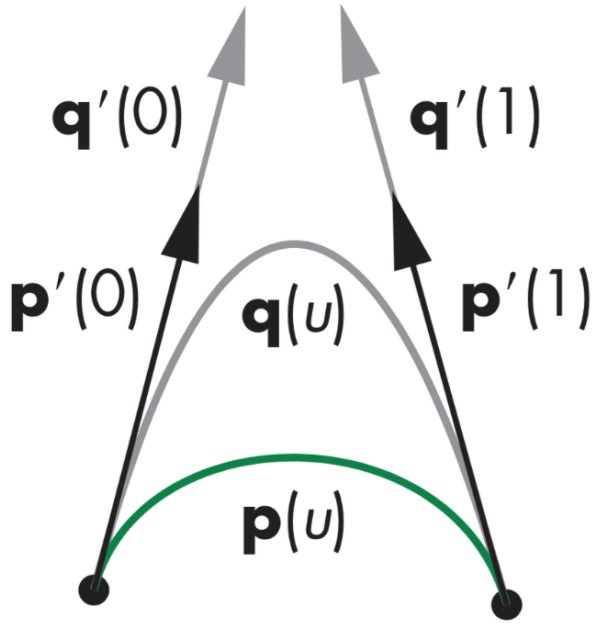
Note that G^0 is the same as C^0 continuity.

Example

Here the curves $\mathbf{p}$ and $\mathbf{q}$ have the same tangents at the ends of the segment but different derivatives

Can generate different curves by changing the "magnitude" (α) in G^1 continuity

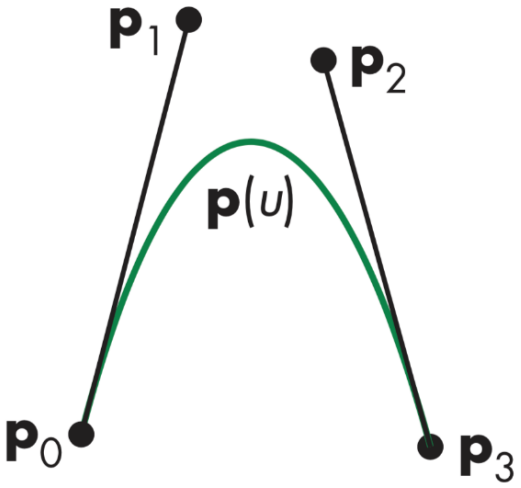
This techniques is used in drawing applications



Bézier's Idea

In graphics and CAD, we do not usually have derivative data

Bézier suggested using the same 4 data points (for cubic curve) as with the cubic interpolating curve to approximate the derivatives at the two end points



Bézier Curves

The B′ezier curve is just another representation of polynomial curves.

It is given by

$$P(t) = B_{0,n}(t)P_0 + B_{1,n}(t)P_1 + \cdots + B_{n,n}(t)P_n, t \in [0,1]$$

where

$$B_{i,n}(t) = \frac{n!}{(n-i)!\, i!}(1-t)^{n-i}t^i, \qquad i = 0,1,\ldots,n$$

Bernstein polynomials of degree n

Bézier Curve of Degree 1

$\sum_{i=0} B_{i,1}(t) P_i$

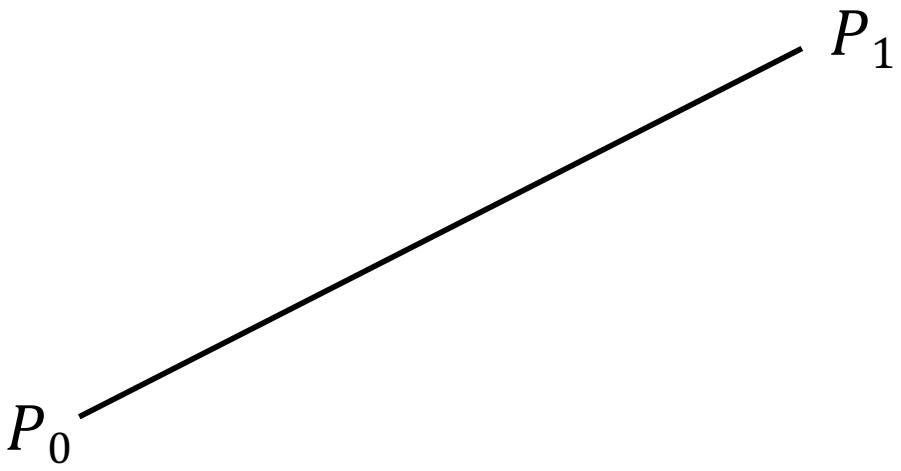
$$n = 1$$

$$P(t) = B_{0,1}(t)P_0 + B_{1,1}(t)P_1$$

$$B_{0,1}(t) = 1 - t, \; B_{1,1}(t) = t$$

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

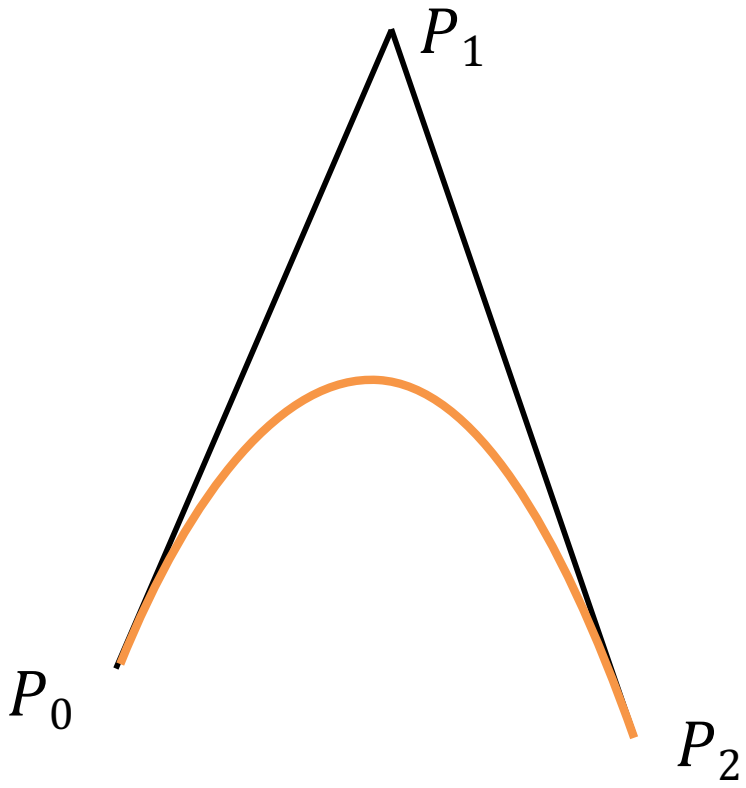
What is the curve?



Bézier Curve of Degree 2

The quadratic Bézier curve:

$$P(t) = (1-t)^2\, P_0 \; + 2(1-t)tP_1 \; + t^2 P_2, \qquad t \in [0,1].$$

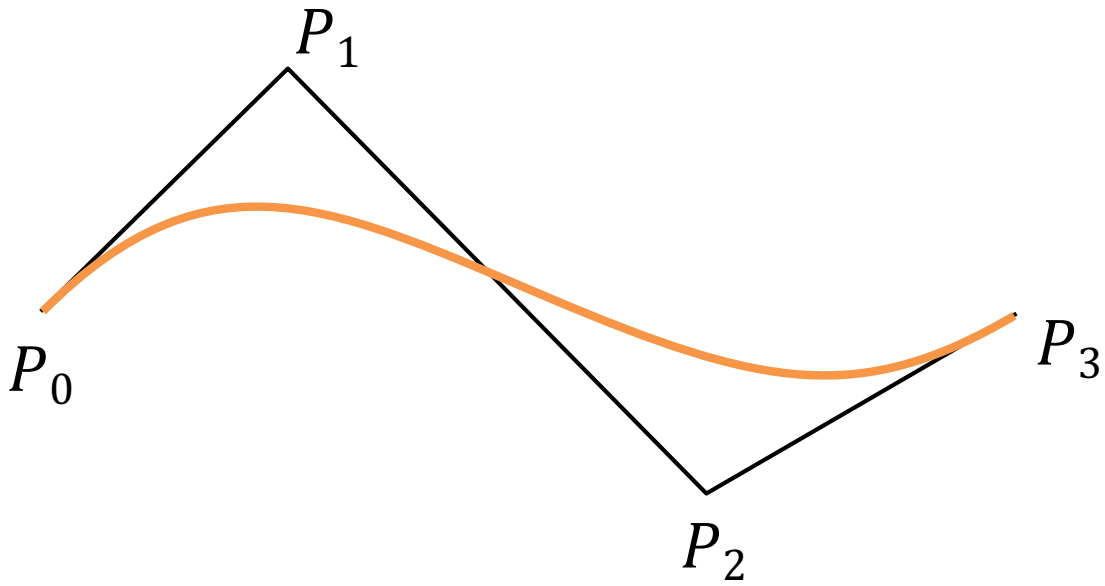


Bézier Curve of Degree 3

The cubic Bézier curve

$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_3,$$

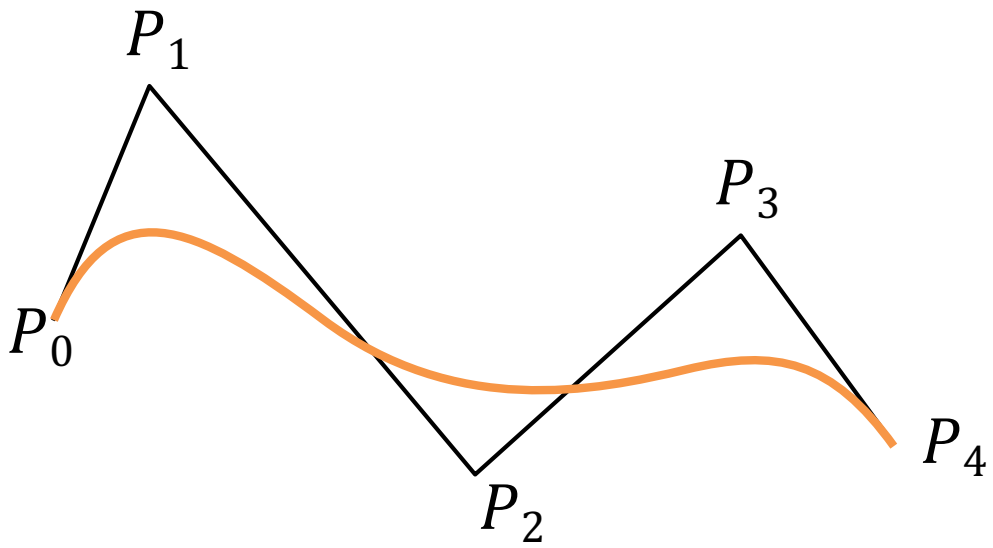
$t \in [0, 1]$.



Bézier Curve of Degree 4

The quartic Bézier curve

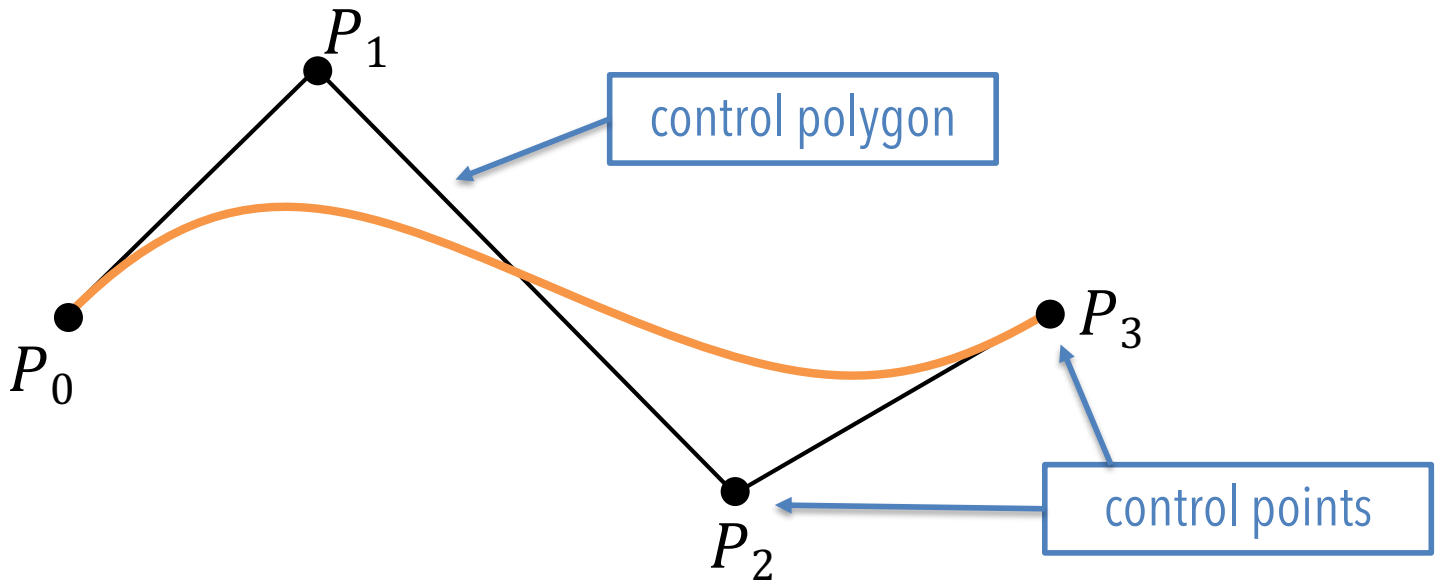
$$P(t) = (1-t)^4 P_0 + 4(1-t)^3 t P_1 + 6(1-t)^2 t^2 P_2$$
$$+4(1-t)t^3 P_3 + t^4 P_4, \ t \in [0,1].$$



Terminologies

The points P_i are called the **control points** or control vertices of the Bézier curve $P(t)$.

The polygon connecting $P_0, P_1, \ldots, P_n$, in this order, is called the **control polygon** of $P(t)$.



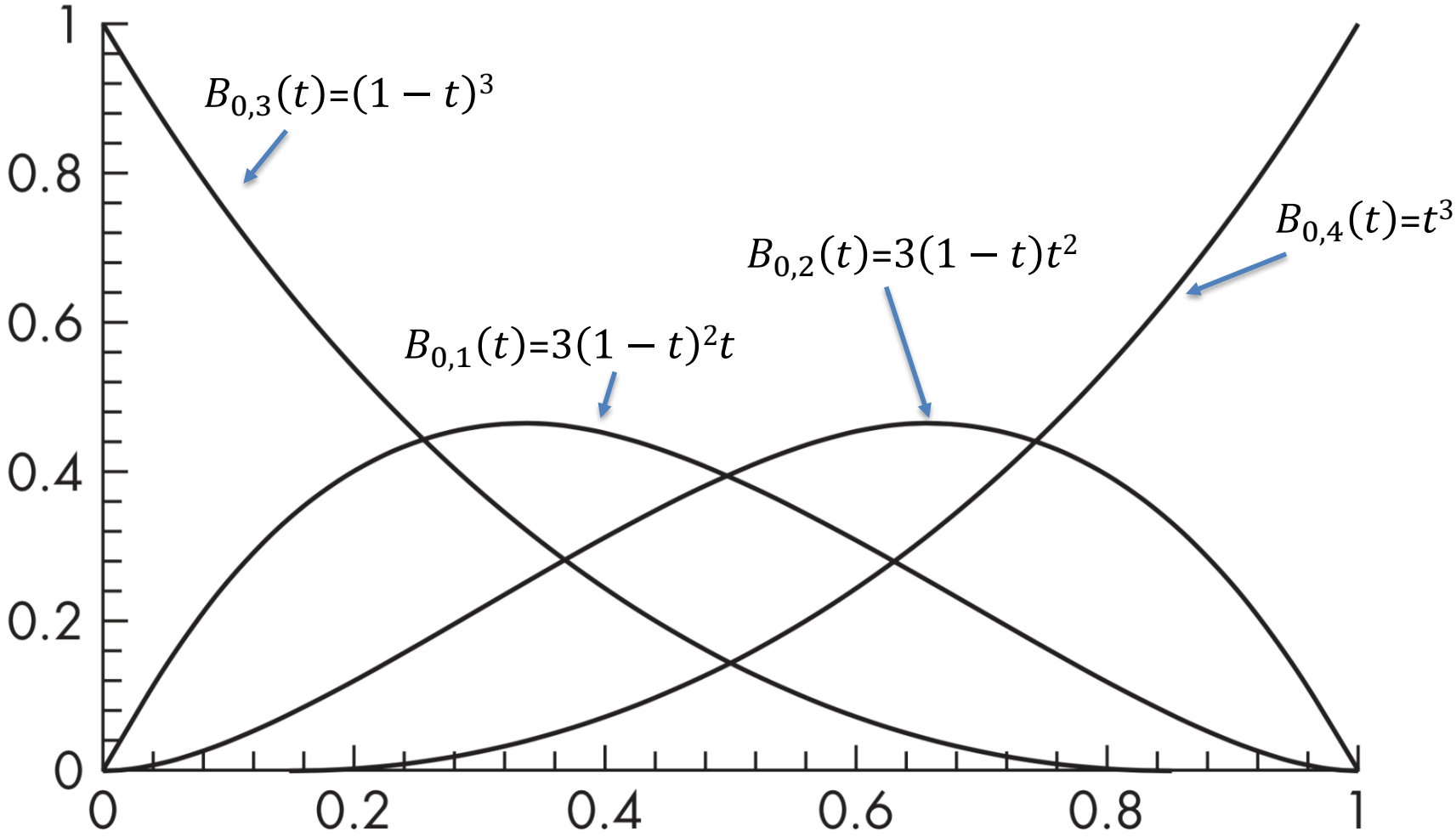
Terminologies

The points P_i are called the **control points** or control vertices of the Bézier curve $P(t)$.

The polygon connecting $P_0, P_1, \ldots, P_n$, in this order, is called the **control polygon** of $P(t)$.

The polynomials $B_{i,n}(t)$ are called the **blending functions** or **basis functions**.

Bézier Cubic Basis Functions



Properties of Bézier Curves

1. Any polynomial curve can be put in the Bézier form.

Proof:

The polynomials

$$\{B_{0,n}(t), B_{1,n}(t), \dots, B_{n,n}(t)\}$$

$$\{1, t, t^2, \dots, t^n\}$$

span the same space.

Example

$$\begin{pmatrix} B_{0,3}(t) \\ B_{1.3}(t) \\ B_{2,3}(t) \\ B_{3,3}(t) \end{pmatrix} = \begin{pmatrix} (1-t)^3 \\ 3t(1-t)^2 \\ 3t^2(1-t) \\ t^3 \end{pmatrix} = \begin{pmatrix} 1 & -3 & 3 & 1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Properties of Bézier Curves

2. A Bézier curve $P(t)$ of degree n interpolates the two endpoints P_0 and P_n.

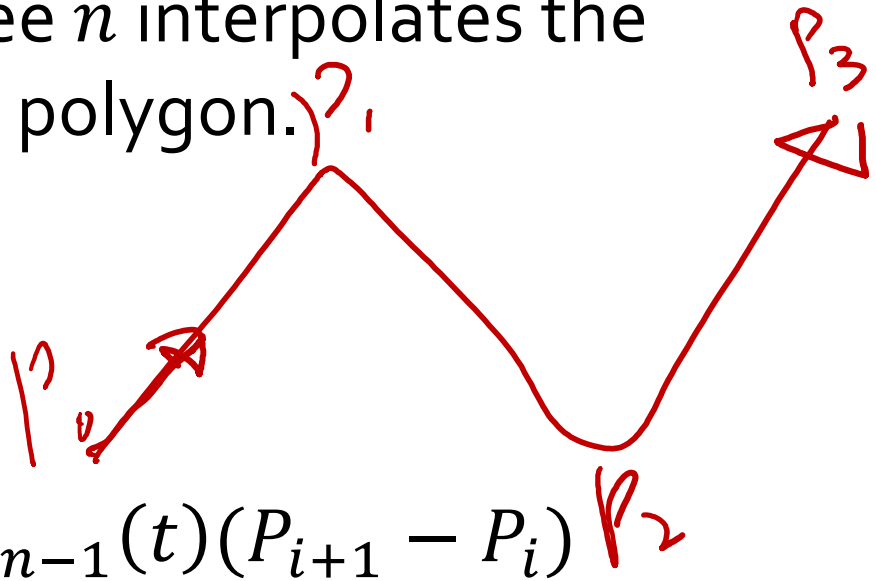
$$\sum_{i=0}^{n} B_{i,n}(t) P_i$$

Proof:

$$P(0) = P_0, \qquad P(1) = P_n$$

Properties of Bézier Curves

3. A Bézier curve $P(t)$ of degree n interpolates the two end sides of the control polygon.



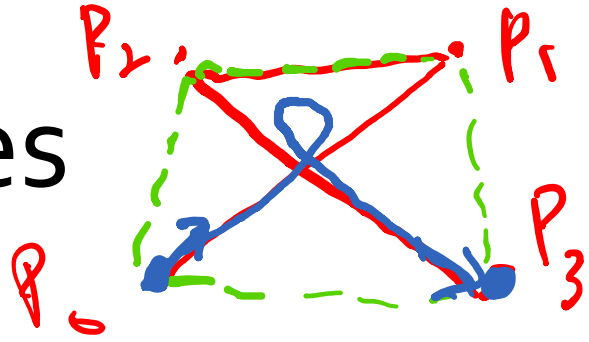
Proof:

$$P'(t) = \frac{dP(t)}{dt} = n\sum_{i=0}^{n-1} B_{i,n-1}(t)(P_{i+1} - P_i)$$

$$P'(t)|_{t=0} = n(P_1 - P_0)$$

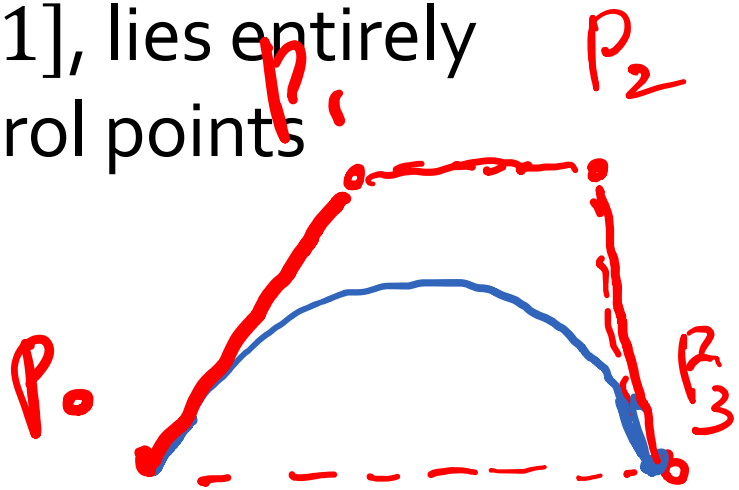
$$P'(t)|_{t=1} = n(P_n - P_{n-1})$$

Properties of Bézier Curves



4. Convex hull property

   The curve segment $P(t), t \in [0,1]$, lies entirely inside the convex hull of all control points

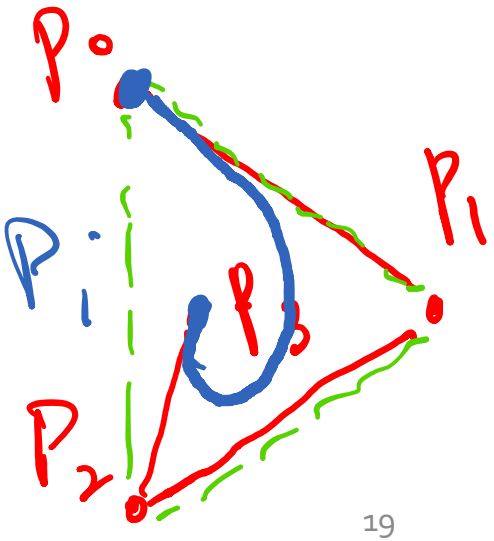


Proof:

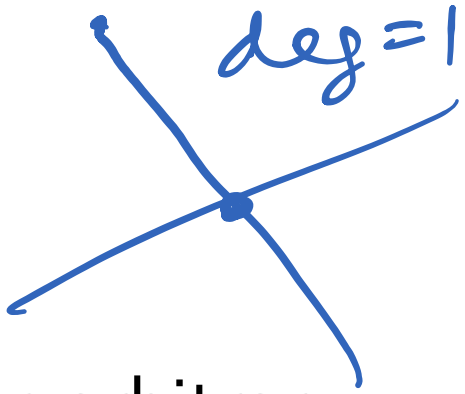
This property follows from

$$\sum_{i=0}^{n} B_{i,n}(t) \equiv 1,$$

$$B_{i,n}(t) \geq 0, \quad t \in [0,1]$$

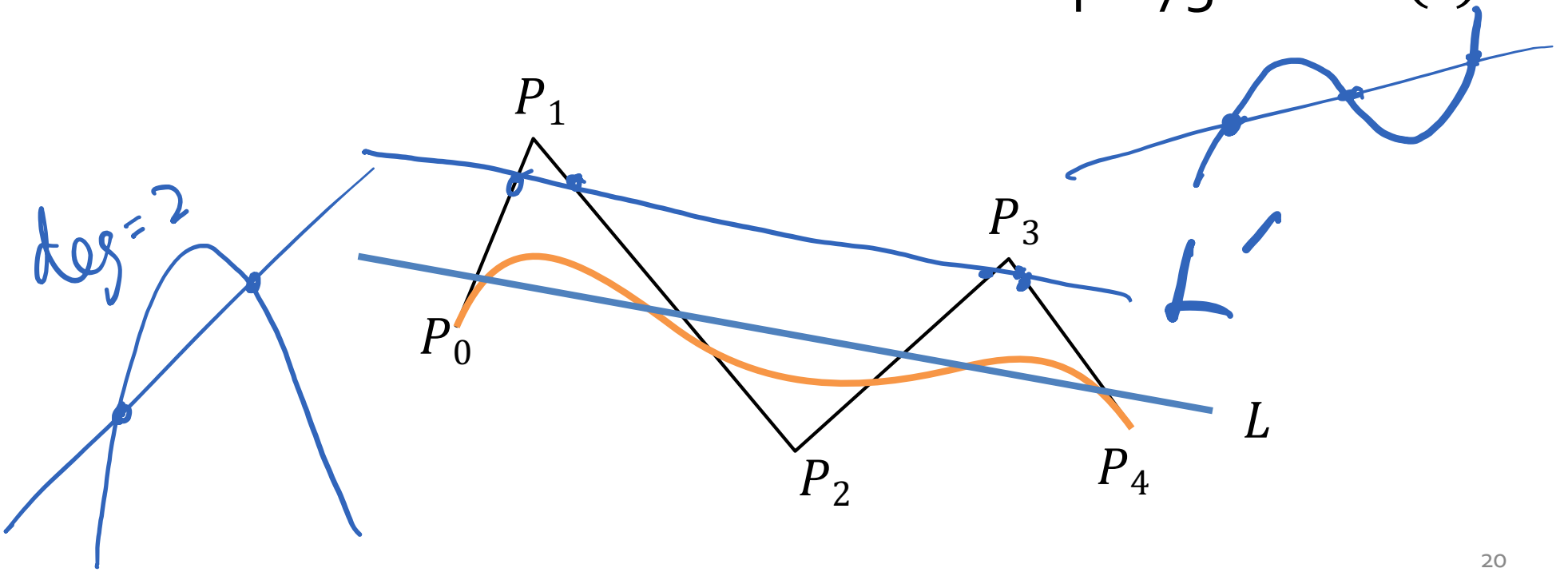


Properties of Bézier Curves

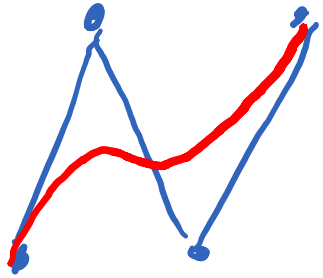


5. Variation diminishing property

   The number of intersections between an arbitrary straight line L and the Bézier curve $P(t), t \in [0,1]$, is NO greater than the number of intersections between the line L and the control polygon of $P(t)$.



Properties of Bézier Curves



6. Invariant form under affine transformations

   For any affine transformation $\mathbf{M}$: $X' = AX + b$, there is

$$\mathbf{M}\big(P(t)\big) = \sum_{i=0}^{n} B_{i,n}(t)\mathbf{M}(P_i).$$

Proof:

$$\begin{aligned}\mathbf{M}\big(P(t)\big) &= AP(t) + b \\ &= \sum_{i=0}^{n} B_{i,n}(t)AP_i + \sum_{i=0}^{n} B_{i,n}(t)b \\ &= \sum_{i=0}^{n} B_{i,n}(t)(AP_i + b) \\ &= \sum_{i=0}^{n} B_{i,n}(t)M(P_i)\end{aligned}$$

Properties of Bézier Curves

6. Invariant form under affine transformations

Thanks to this property, when a Bézier curve is transformed affinely, we can

a) transform many sampled points on the curve directly, or

b) transform the control points only and use the transformed control points to generate a new Bézier curve

These two ways yield the same transformed curve, but the latter can be more efficient when there are many points to be evaluated on the curve.

Composite Bézier Curves

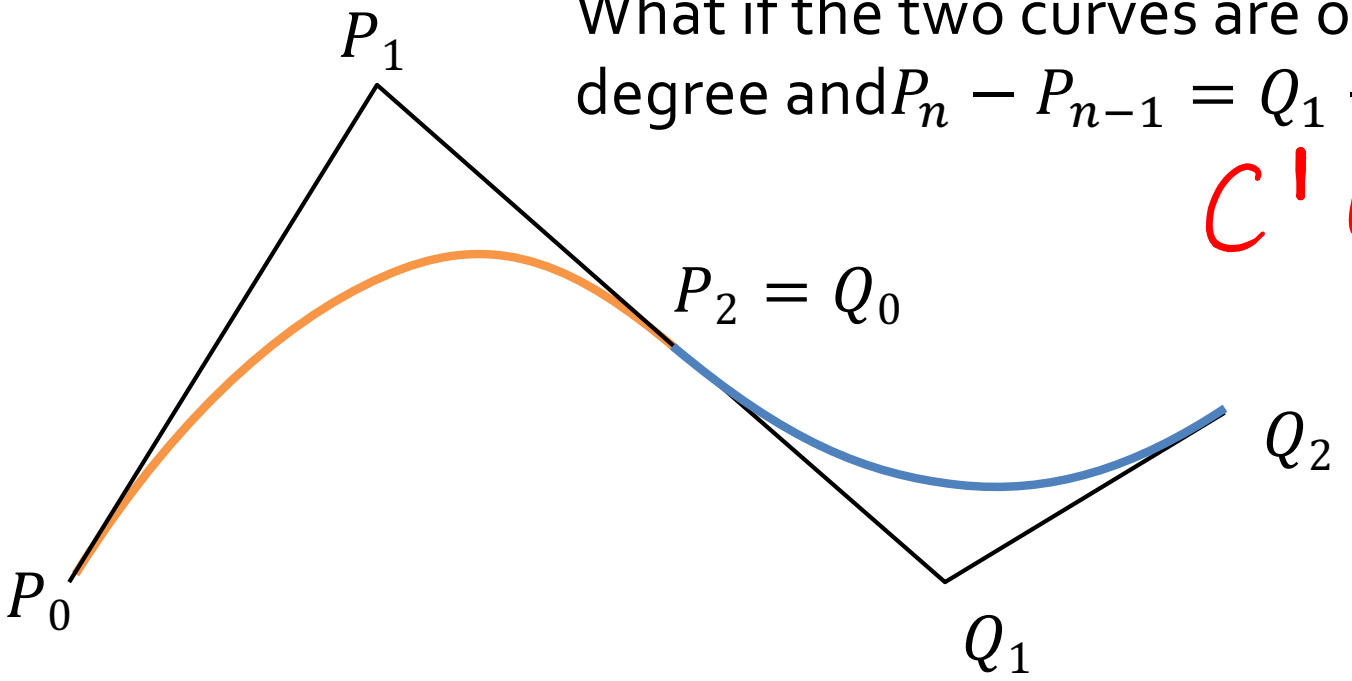
By placing the end control points of two Bézier curve collinear, we can obtain a smooth curve comprising two Bézier curves.

The curves then have G^1 continuity at P_2. Why?

What if the two curves are of the same degree and $P_n - P_{n-1} = Q_1 - Q_0$?

$|P_1P_2| \neq |Q_1P_2|$

C^1 continuity



Bézier Matrix

$$\mathbf{p}(u) = \mathbf{b}(u)^T\mathbf{p} = \mathbf{u}^T\mathbf{M}_B\mathbf{p}$$

For cubic Bézier curves

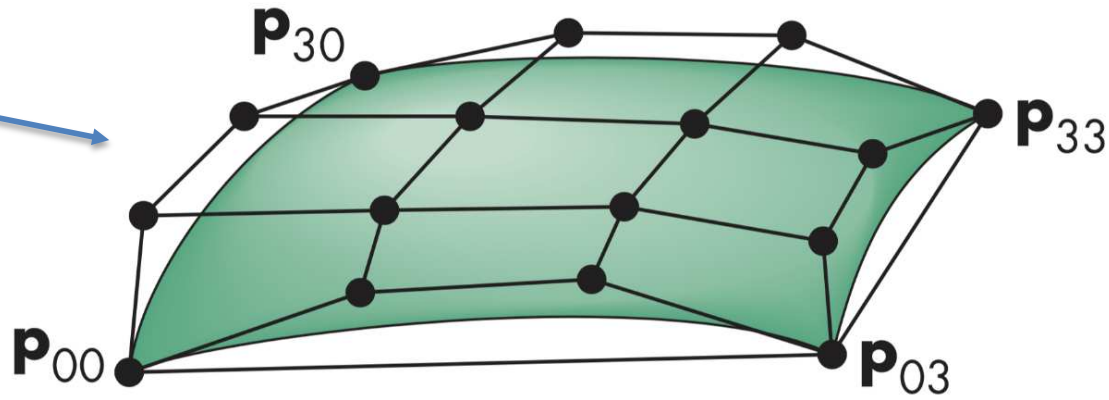
$$\mathbf{M}_B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

Bézier Patches

Using same data array $\mathbf{P} = [\mathbf{p}_{ij}]$ as with interpolating form

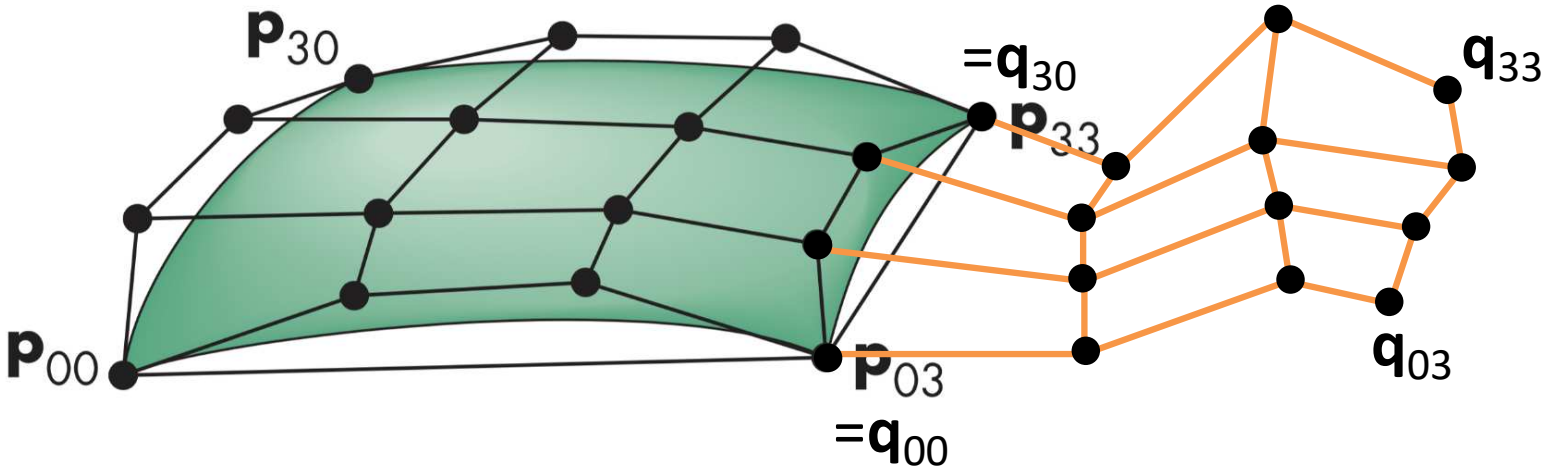
$$\mathbf{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_i(u)\, b_j(v)\, \mathbf{p}_{ij} = \mathbf{u}^T \mathbf{M}_B \mathbf{P} \mathbf{M}_B{}^T \mathbf{v}$$

Patch lies in convex hull



Bézier Bicubic Surface Patches

A 2 x 1 bicubic surface patches



COMP3271 Computer Graphics

Curves & Surfaces (III)

2019-20

Objectives

The de Casteljau algorithm for evaluating Bézier curves

Other curves and surfaces:

- Conics & Quadrics
- Extrusion surfaces
- Surface of revolutions
- Sweep surfaces

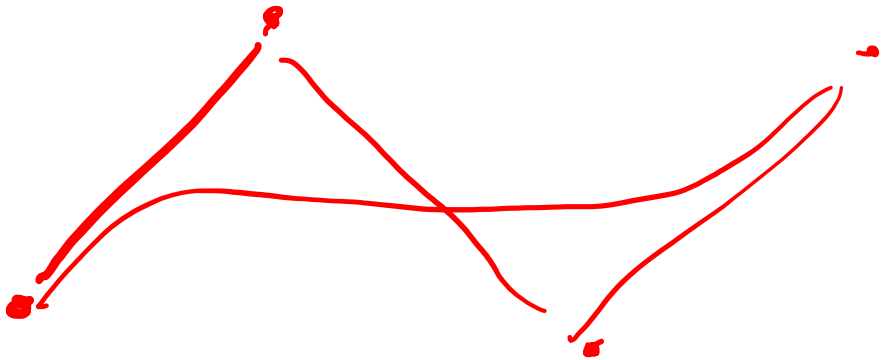
The de Casteljau Algorithm

We can use the convex hull property of Bézier curves to obtain an efficient recursive method that does not require any function evaluations

- Uses only the values at the control points

Based on the idea that "any polynomial and any part of a polynomial is a Bézier polynomial for properly chosen control data"

$$\sum_{i=0}^{3} B_{i,n}(t) P_i$$



The de Casteljau Algorithm

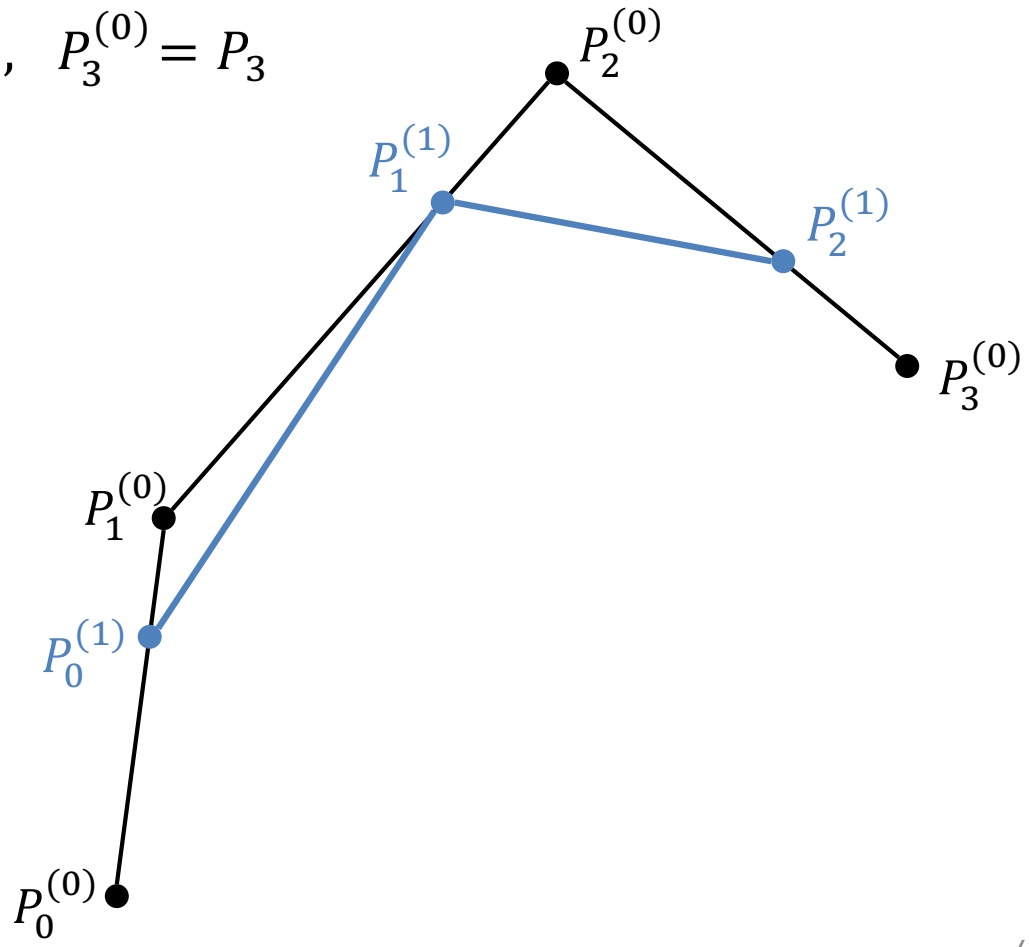
Given a cubic Bézier curve $P(t) = \sum_{i=0}^{3} P_i B_{i,3}(t)$. The following procedure produces a point $P(t)$ on the curve.

$$P_0^{(0)} = P_0,\ P_1^{(0)} = P_1,\ P_2^{(0)} = P_2,\ P_3^{(0)} = P_3$$

$$P_0^{(1)} = (1-t)P_0 + tP_1$$

$$P_1^{(1)} = (1-t)P_1 + tP_2$$

$$P_2^{(1)} = (1-t)P_2 + tP_3$$



The de Casteljau Algorithm

Given a cubic Bézier curve $P(t) = \sum_{i=0}^{3} P_i B_{i,3}(t)$. The following procedure produces a point $P(t)$ on the curve.

$$P_0^{(0)} = P_0,\ P_1^{(0)} = P_1,\ P_2^{(0)} = P_2,\ P_3^{(0)} = P_3$$

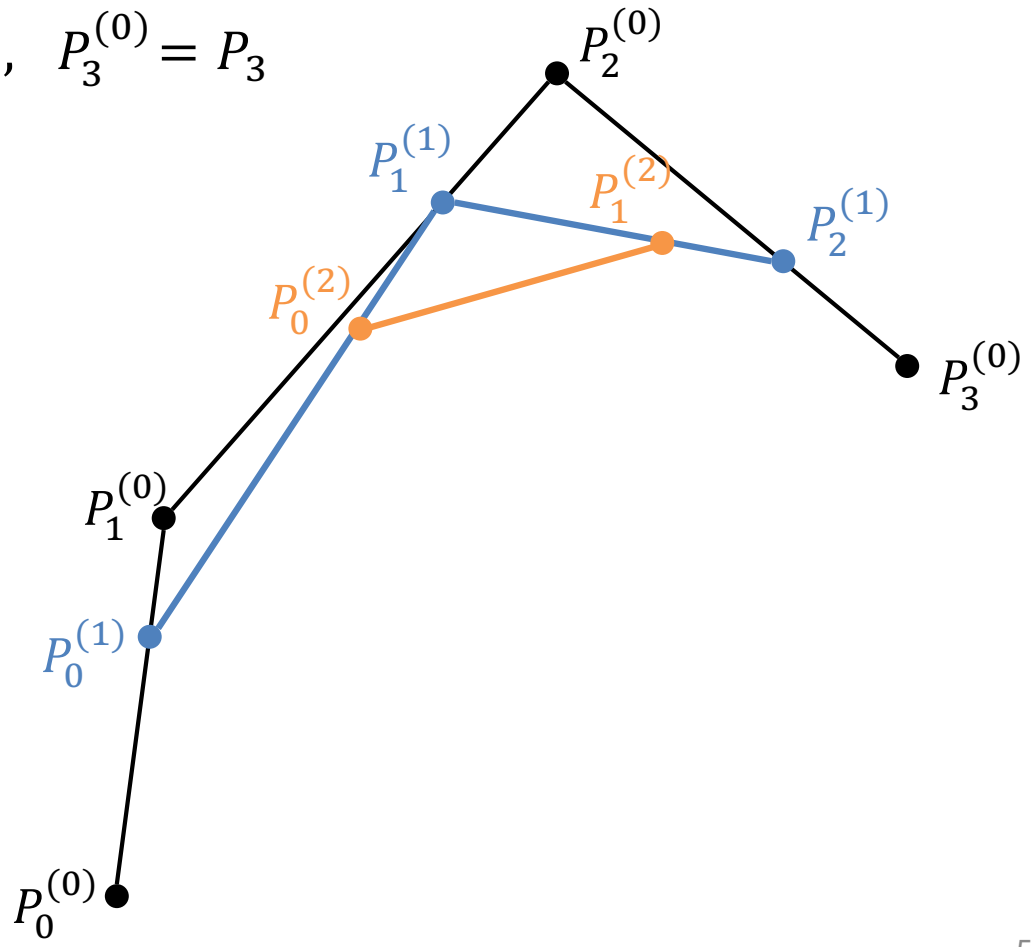
$$P_0^{(1)} = (1-t)P_0 + tP_1$$

$$P_1^{(1)} = (1-t)P_1 + tP_2$$

$$P_2^{(1)} = (1-t)P_2 + tP_3$$

$$P_0^{(2)} = (1-t)P_0^{(1)} + tP_1^{(1)}$$

$$P_1^{(2)} = (1-t)P_1^{(1)} + tP_2^{(1)}$$



The de Casteljau Algorithm

Given a cubic Bézier curve $P(t) = \sum_{i=0}^{3} P_i B_{i,3}(t)$. The following procedure produces a point $P(t)$ on the curve.

$$P_0^{(0)} = P_0,\ P_1^{(0)} = P_1,\ P_2^{(0)} = P_2,\ P_3^{(0)} = P_3$$

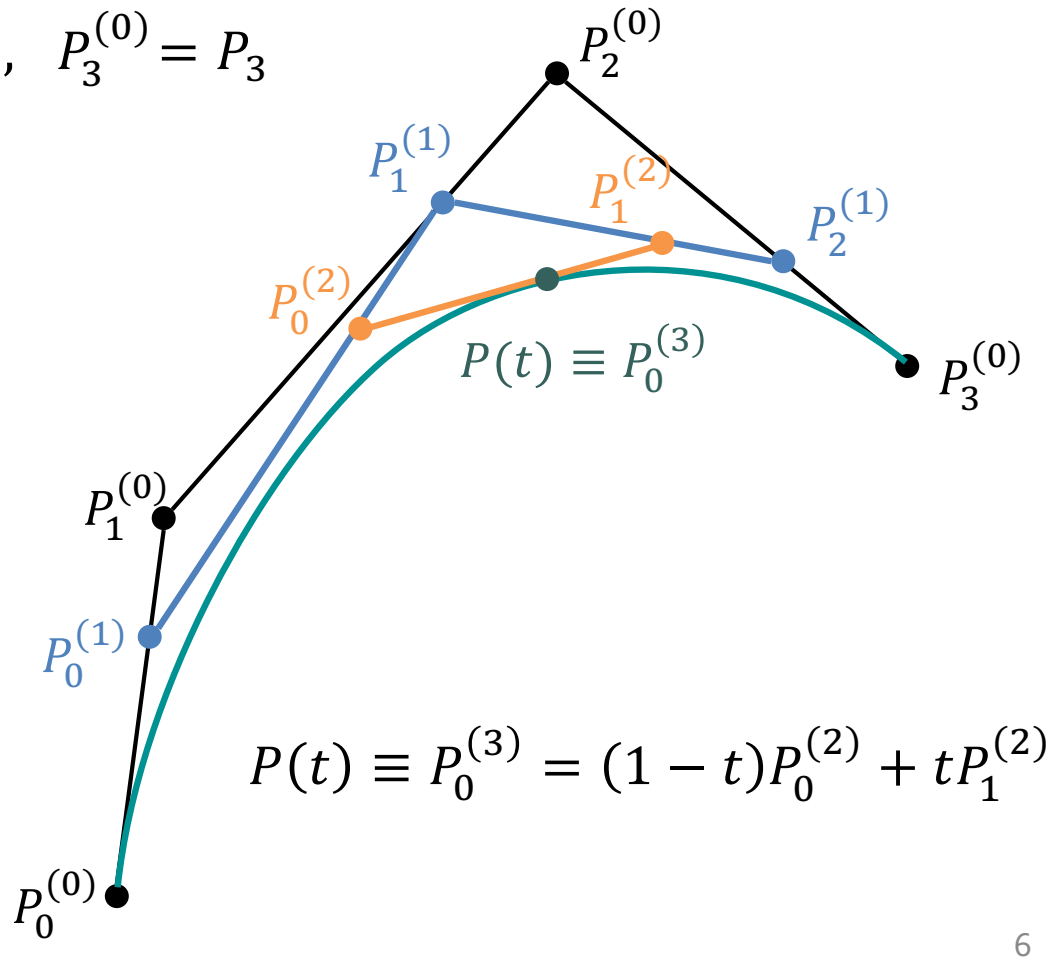
$$P_0^{(1)} = (1-t)P_0 + tP_1$$

$$P_1^{(1)} = (1-t)P_1 + tP_2$$

$$P_2^{(1)} = (1-t)P_2 + tP_3$$

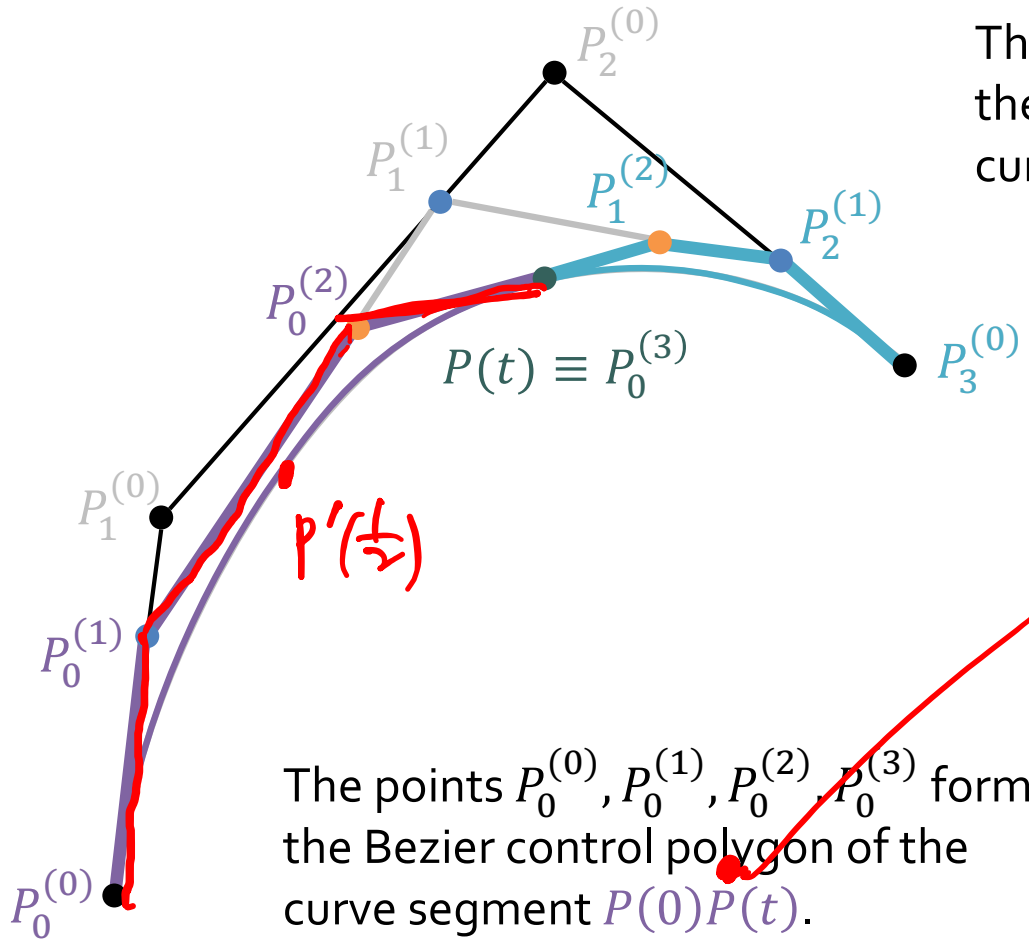
$$P_0^{(2)} = (1-t)P_0^{(1)} + tP_1^{(1)}$$

$$P_1^{(2)} = (1-t)P_1^{(1)} + tP_2^{(1)}$$



$$P(t) \equiv P_0^{(3)} = (1-t)P_0^{(2)} + tP_1^{(2)}$$

Splitting the Control Polygon



The points $P_0^{(3)}, P_1^{(2)}, P_2^{(1)}, P_2^{(0)}$ form the Bezier control polygon of the curve segment $P(t)P(1)$.

The points $P_0^{(0)}, P_0^{(1)}, P_0^{(2)}, P_0^{(3)}$ form the Bezier control polygon of the curve segment $P(0)P(t)$.

Middle-point Subdivision

When setting $t = \frac{1}{2}$, the arithmetic operations of the above algorithm are simplified to

$$P_i^{(k+1)} = \frac{P_i^{(k)} + P_{i+1}^{(k)}}{2}.$$

So only addition and right shift (division by 2) are required.

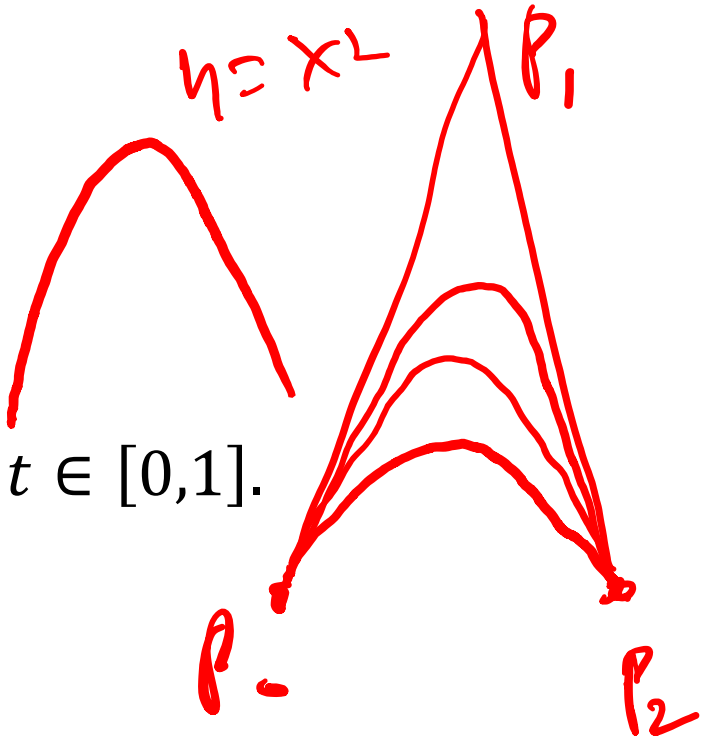
The middle-point subdivision scheme works by computing the Bézier control polygons of the two sub-curves of $P(t), t \in [0,1]$, over subintervals $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 0]$. Then each sub-curve is recursively subdivided.

Note that the depth of subdivision can be made adaptive to the local curvature or the error of approximation.

Rational Bézier Curves

A rational Bézier curve is represented by

$$P(t) = \frac{\sum_{i=0}^{n} w_i B_{i,n}(t) P_i}{\sum_{i=0}^{n} w_i B_{i,n}(t)}, \qquad t \in [0,1].$$

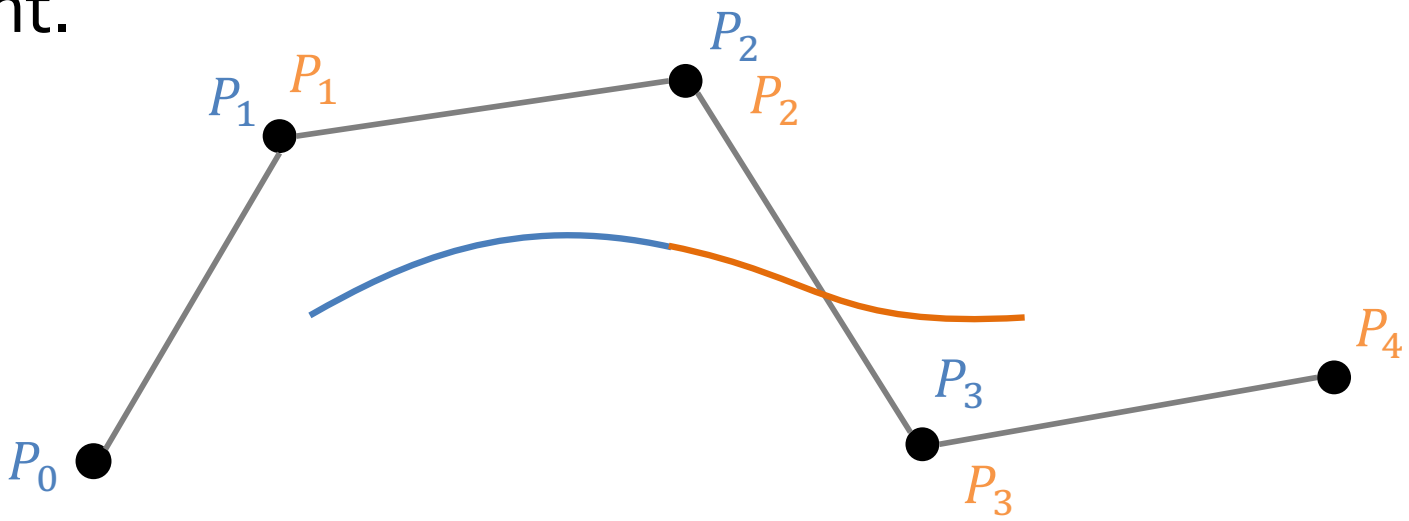


- w_i can be thought of weights
- Increasing the weight w_i pulls a portion of $P(t)$ towards the control point P_i, and decreasing w_i pushes $P(t)$ away from P_i
- An advantage of the rational curve is that it encompasses all conic sections (with $n = 2$).

B-Splines

Basis splines: use the control points $p_{i-2}, p_{i-1}, p_i, p_{i+1}$ to define curve only between p_{i-1} and p_i (for cubic curves)

Allows us to apply more continuity conditions to each segment.



NURBS: Nonuniform Rational B-Spline curves and surfaces

Conic Sections

A common class of curves with a very long history

- defined by intersections of a plane with a cone
- defined implicitly by the 2nd degree polynomial

$$F(x,y) = ax^2 + 2bxy + 2cx + dy^2 + 2ey + f$$

In matrix form

$$F(x,y) = \mathbf{v}^{\mathsf{T}}\mathbf{Q}\mathbf{v} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0$$

This describes several generally useful kinds of curves

- circles, ellipses, parabolas, hyperbolas, and lines

Quadric Surfaces

Quadrics are the 3D analogue of conics

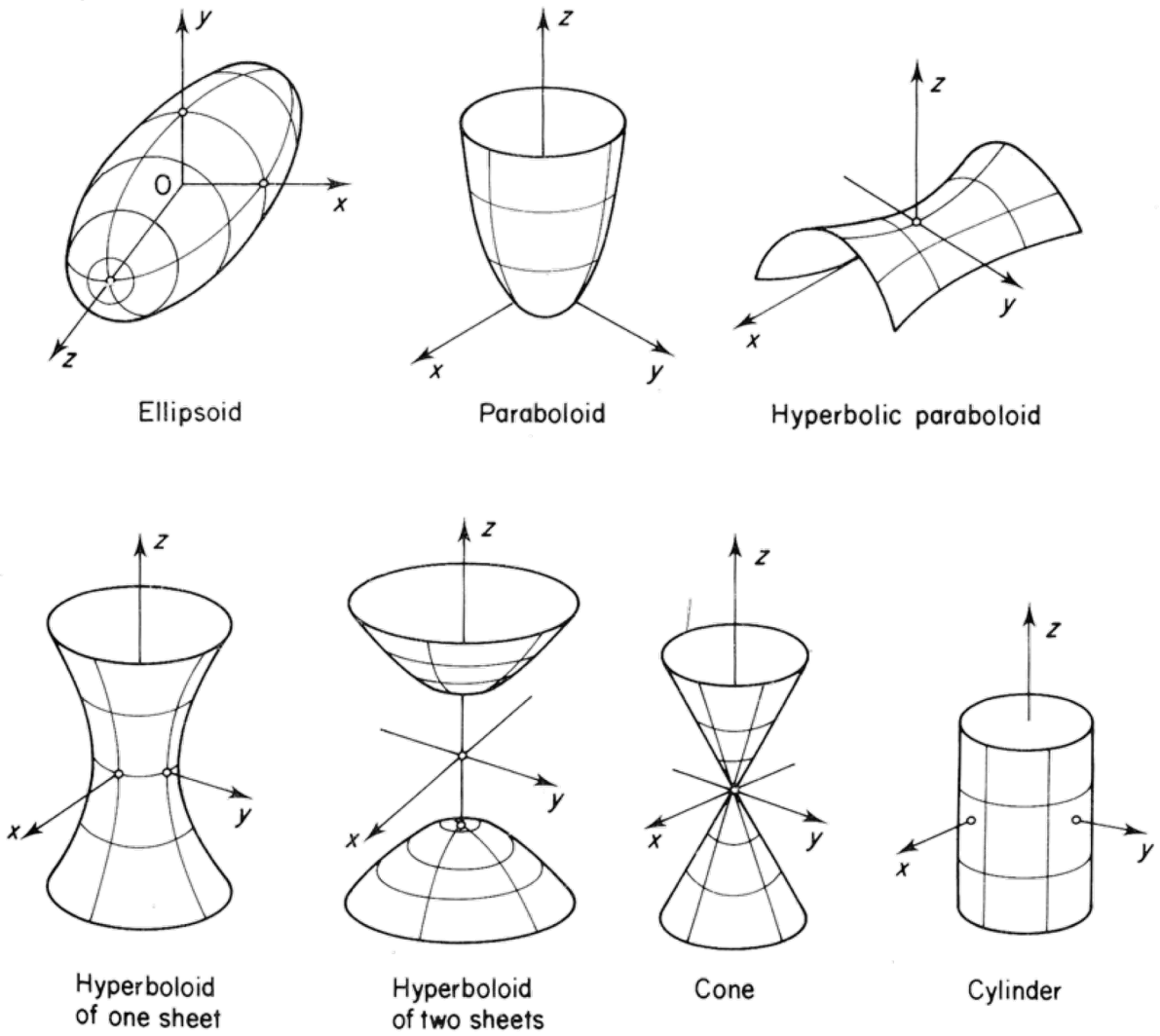
- defined by the 2nd degree polynomial

$$F(x,y,z) = ax^2 + 2bxy + 2cxz + 2dx + ey^2 + 2fyz + 2gy + hz^2 + 2iz + j$$

- or in matrix form

$$F(x,y,z) = \mathbf{v}^{\mathsf{T}}\mathbf{Q}\mathbf{v} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

Quadric Surfaces



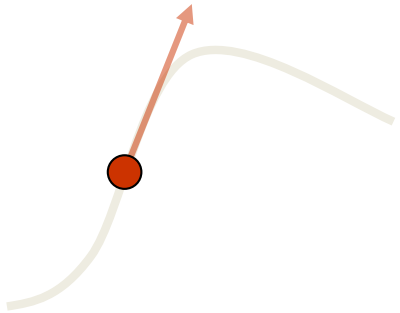
Quadric classes (from Paul Heckbert).

Sweeping out Surfaces

We view space curves as being swept out by a moving point

$$\mathbf{p}(u) = \begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix}$$

- as we vary u the point moves through space
- the curve is the path the point takes



Essentially looked at surfaces the same way

$$\mathbf{p}(u,v) = \begin{bmatrix} x(u,v) & y(u,v) & z(u,v) \end{bmatrix}$$

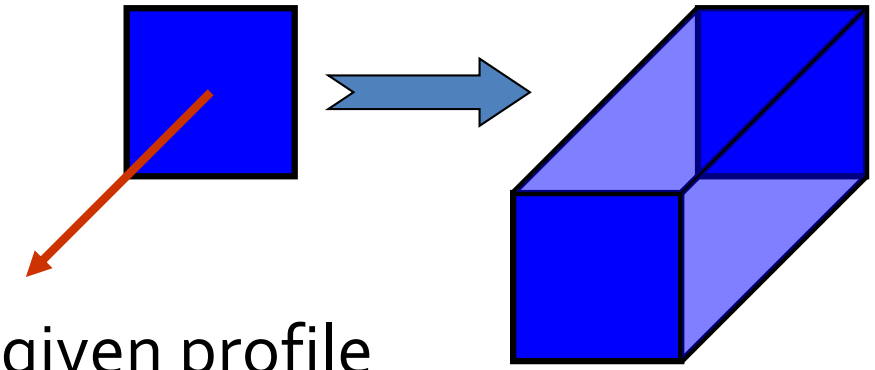
Now let's think about sweeping curves through space instead

- this will define a surface
- the set of all points visited by the curve during its motion

Extrusion Surfaces

Here's a particularly simple method

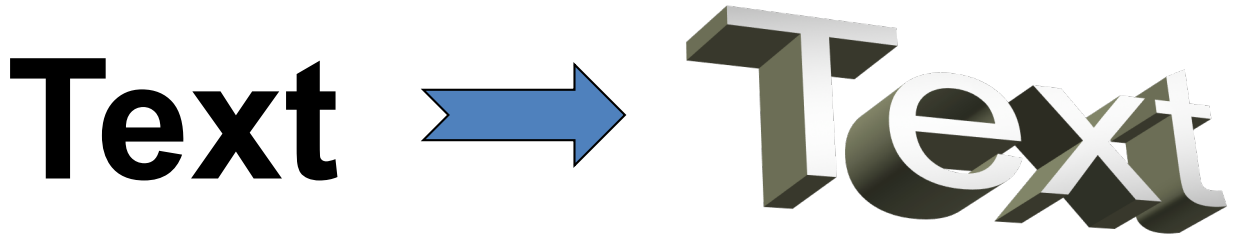
- specify initial (closed) curve
- pick an axis to move along
- and a distance to move



Sweeps out something with the given profile

- open curve defines a surface with an open boundary
- closed curve defines something like a cylinder

This is a common technique used to create 3D text



Extrusion Surfaces

Can be generated by translating a 2D cross-section curve along a fixed direction.

Let the cross-section curve be $C(u)$, and the given direction vector be D. Then the surface is defined by

$$E(u, v) = vD + C(u), \qquad v \in [v_0, v_1].$$

Surfaces of Revolution

$C(\theta) = (\cos\theta, \sin\theta, 0), \theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$

$S(\theta, \phi) = R_y(\phi) C(\theta)$

$$\begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}$$

Extrusion moves curves via translation

- we can just as easily use rotation

Start with some curve

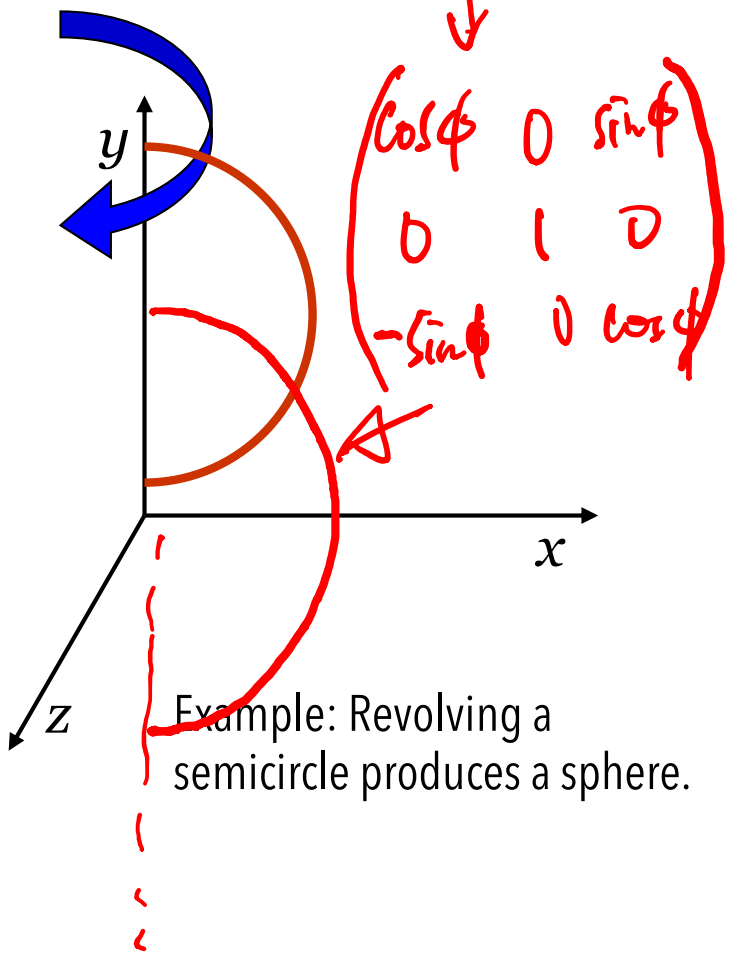
- pick an axis of rotation
- rotate about axis by 360°

Characteristics of revolved surfaces

- closed if endpoints on axis
- open otherwise
- but we can always fill in top/bottom
- by construction, they're symmetric

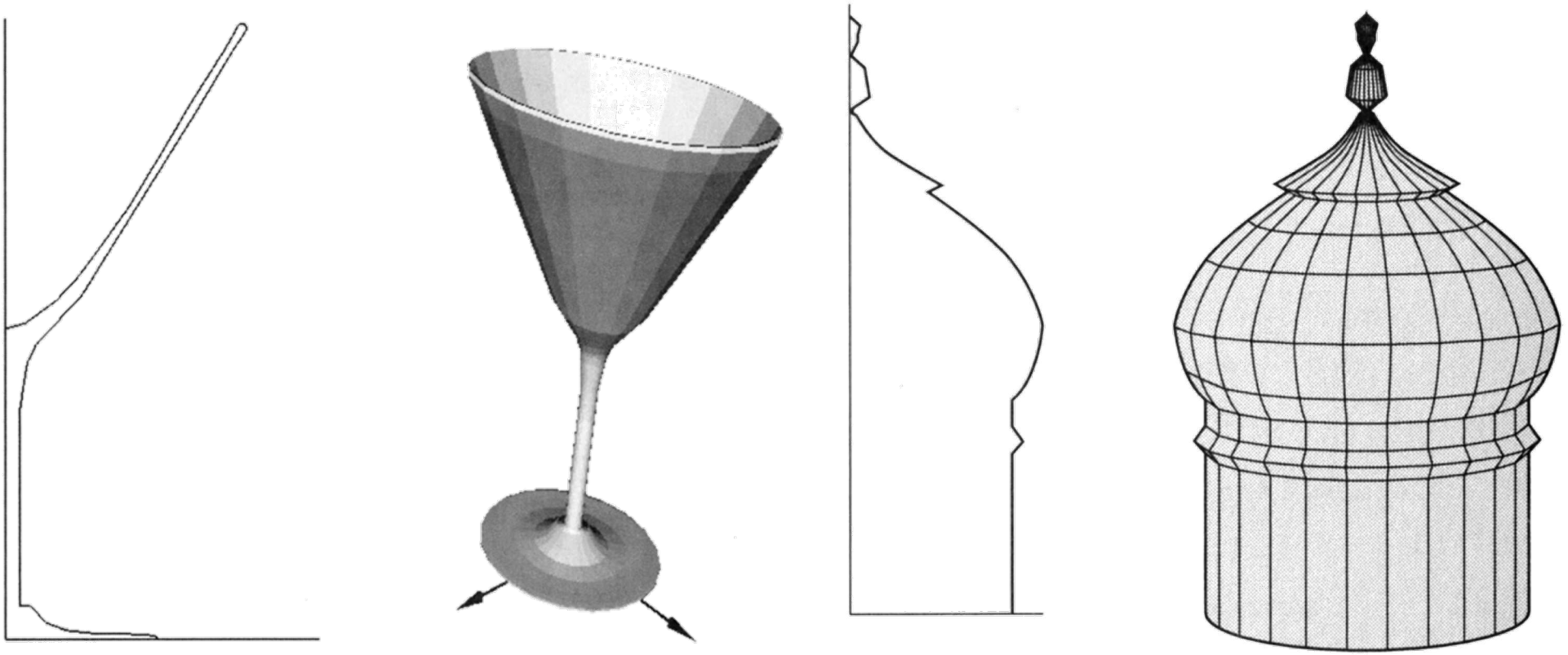
Lots of other easy examples:

- cylinder, cone, paraboloid, …



Example: Revolving a semicircle produces a sphere.

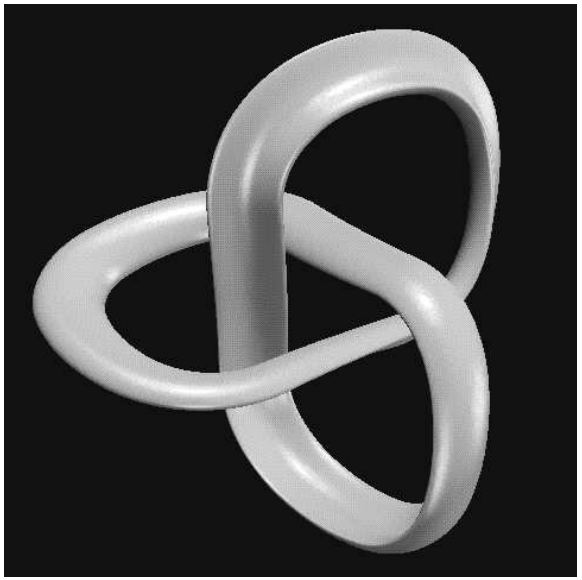
More Complex Examples of Revolution



Sweep Surfaces

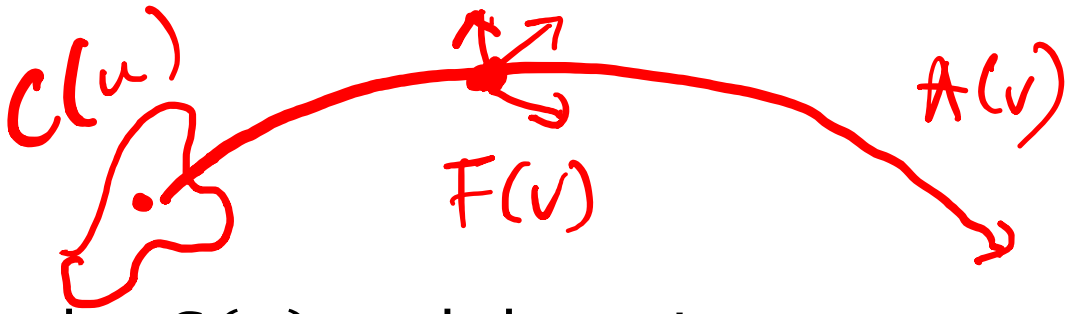
The sweep surface is generated by sweeping a 2D cross-section curve along an axis curve.

The plane of the cross-section curve is usually kept perpendicular to the tangent of the axis curve.



A sweep surface along a cubic spline curve.

Sweep Surfaces



Let the cross-section curve be $C(u)$ and the axis curve be $A(v)$.
Let $F(v)$ be the matrix representing a rotation frame attached at $A(v)$. Then the sweep surface is defined by

$$S(u,v) = A(v) + F(v)C(u), \qquad v \in [v_0, v_1].$$

To use $s(v)$ to change the size of the cross-section curve while sweeping it, we have

$$S(u,v) = A(v) + s(v)F(v)C(u), \qquad v \in [v_0, v_1].$$

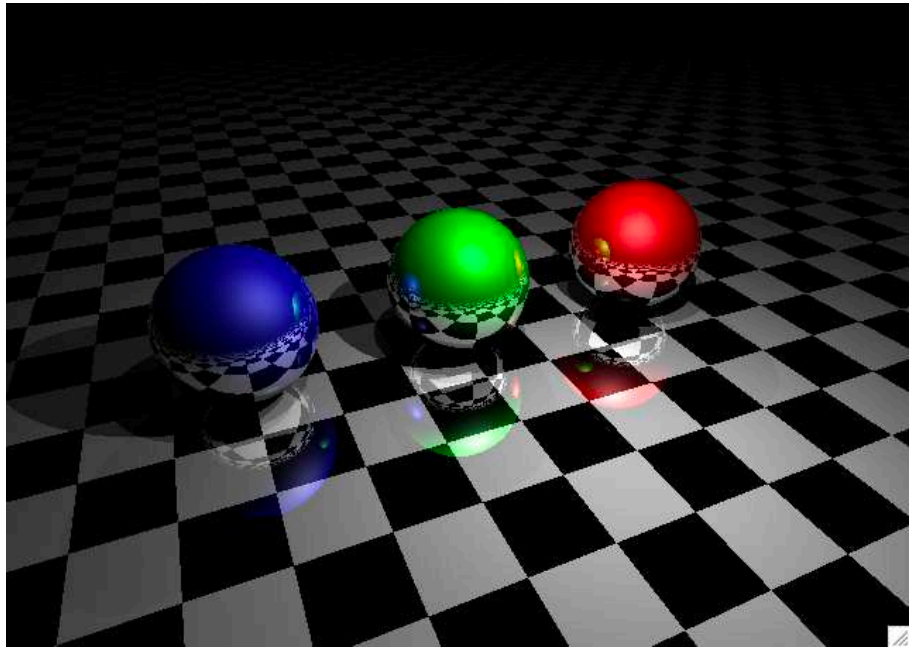
COMP3271 Computer Graphics

Ray Tracing

2019-20

Objectives

- Ray Casting
- Ray Tracing
- Speedup Techniques
- Intersection Tests



- POV-Ray - an open sourced raytracer:
  - http://www.povray.org/
  - https://en.wikipedia.org/wiki/POV-Ray

Looking Back at Image Formation

Light is a stream of photons

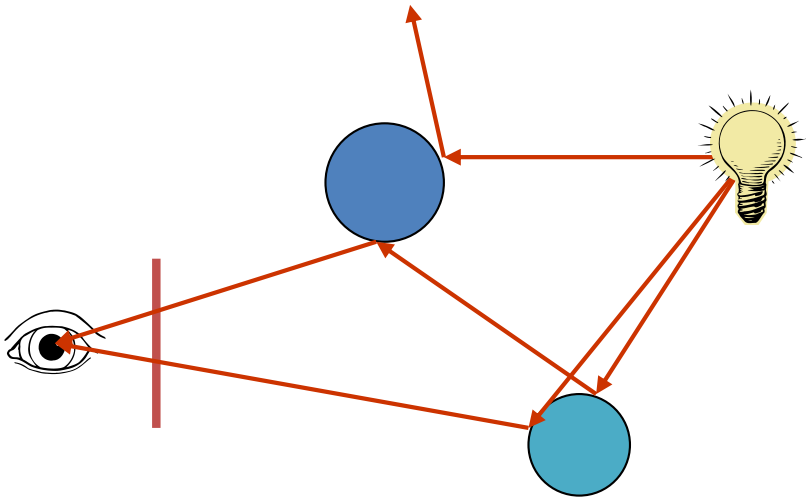
- which move in straight lines
- propagating from lights
- we ignore wave nature of light

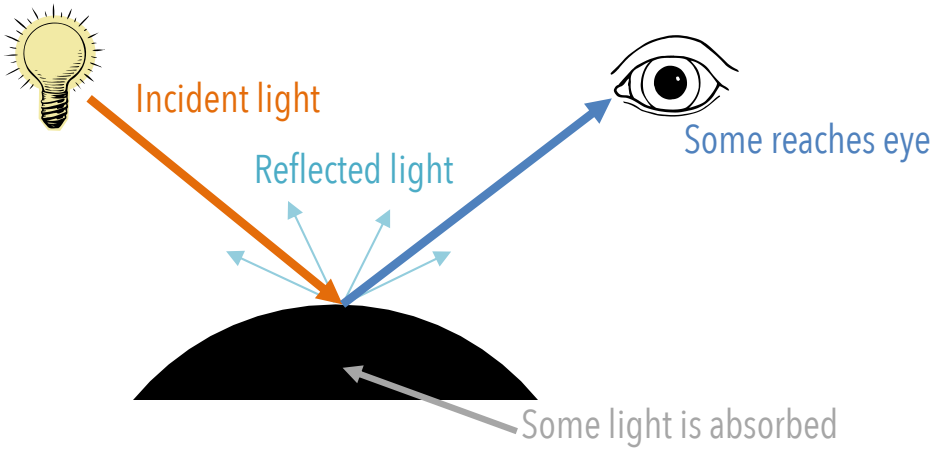
Some rays strike the eye

- (passing through image plane)
- these rays form the image

Light interacts with surfaces

- objects absorb some light
- reflect some of the light
- may bounce off many objects





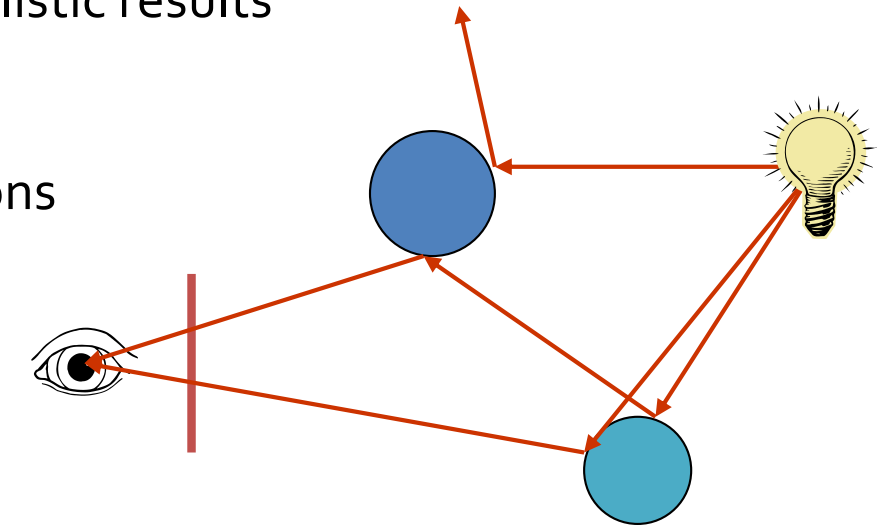
Simulating Rays of Light in the World

We can render the scene by simulating physical light transport

- we hope that this produces more realistic results

Simulation would look like this:

- light source shoots rays in all directions
- rays bounce when they hit surfaces
- can ignore rays when
  - they fly off into empty space
  - almost all of their energy is absorbed
- record rays that strike the image plane
- we call this kind of simulation **forward ray tracing**



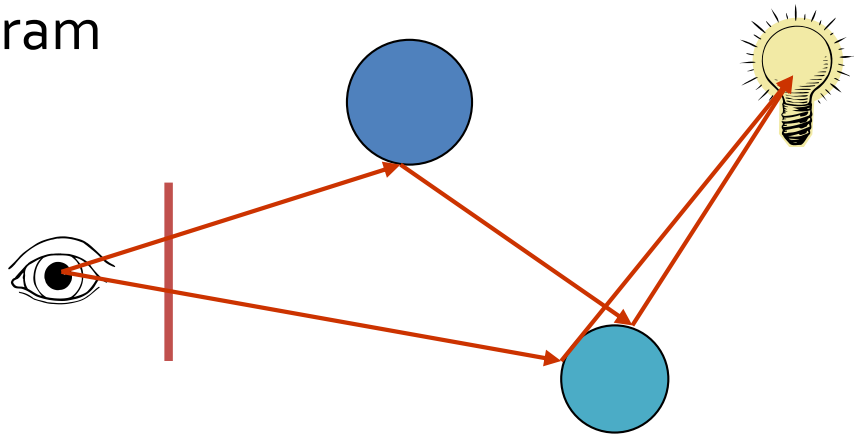
But there's a problem with this

- it can be extremely slow
- only a tiny fraction of light rays actually strike the eye

(Backward) Ray Tracing

Fortunately, there's a simple solution to this problem

- we only care about light rays that eventually strike the eye
- so shoot rays from the eye out into the world
- just reverse arrows on the ray diagram



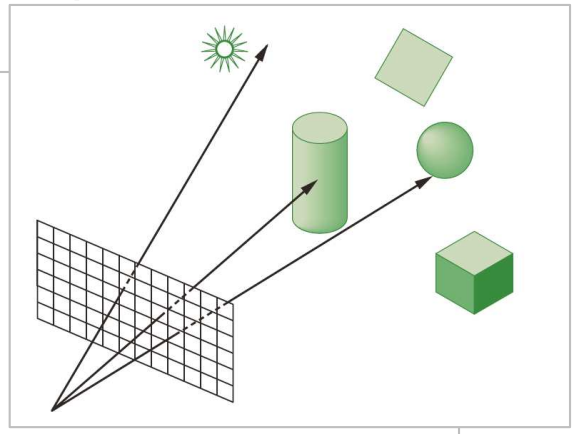
Traditionally, most ray-based renderers take this approach

- so we usually drop the "backward" from the name
- but keep in mind there are alternatives that don't

Ray Casting: Simple Ray-Based Rendering

We can formulate a very simple rendering algorithm

- we'll ignore all this business about rays bouncing around
- just shoot rays into world, see what they strike, and shade



Ray Casting Algorithm:

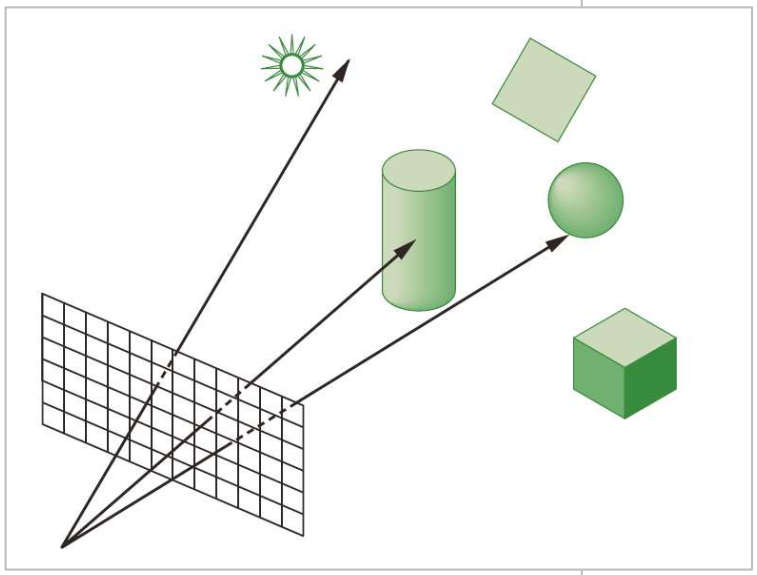
```
for all pixels (x,y)
    compute ray from eye through (x,y)
    compute intersections with all surfaces
    find surface with closest intersection
        shade this surface point (standard illumination equation)
        write this color into pixel (x,y)
```

What we need to resolve

- how to represent rays & generate rays through screen
- how to compute intersections with objects in the world

Pseudo-Code Outline of a Minimal Ray Caster

```
void raycast()
  for all pixels (x,y)
     image(x,y) = trace(compute_eye_ray(x,y))


rgbColor trace(ray r)
  for all surfaces s
    t = compute_intersection(r, s)
    closest_t = MIN(closest_t, t)

  if( hit_an_object )
    return shade(s, r, closest_t)
  else
    return background_color


rgbColor shade(surface s, ray r, double t)
  point x = r(t)
  // evaluate (Phong) illumination equation
  return color
```



From Ray Casting to Ray Tracing

We developed the simple ray casting algorithm

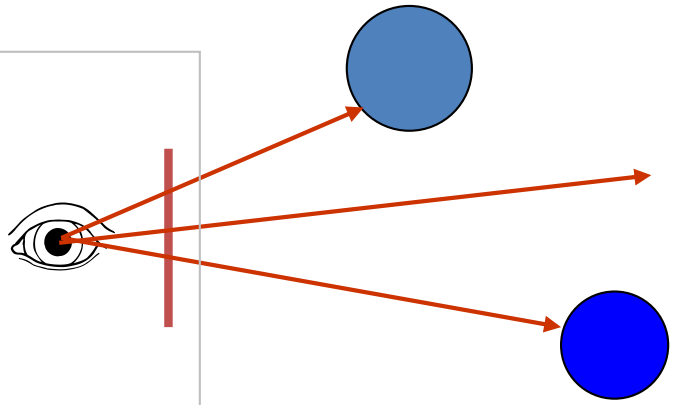
- essentially, replacing z-buffer+rasterization with ray probes
- we still used the Phong illumination model
- thus the results look like OpenGL, only much slower

We want to extend this simple algorithm

- our main focus will be on developing a better shading model

Ray Casting Algorithm:

for all pixels (*x,y*)
    compute ray from eye through (*x,y*)
    compute intersections with all surfaces
    find surface with closest intersection
        compute color using Phong illumination model
        write this color into pixel (*x,y*)

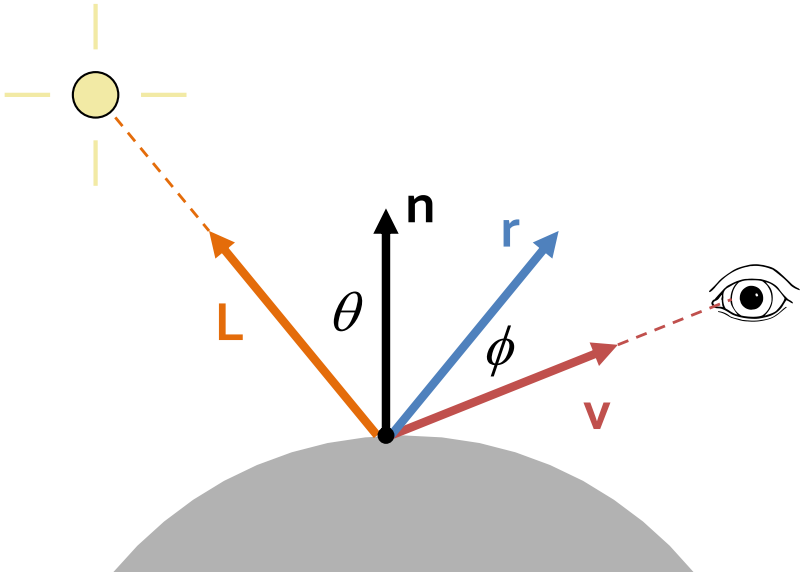


Looking at the Shading Model

Recall our current Phong illumination model

$$I = \sum_{\text{all lights L}} I_L k_d (\mathbf{n} \cdot \mathbf{L}) + I_L k_s (\mathbf{r} \cdot \mathbf{v})^n$$

diffuse reflection — specular highlight



- sums contributions from all lights
- plus the global ambient glow

We'll add three important new components

- shadows, specular reflections, and specular transmission
- our general strategy: recursively trace rays to evaluate shading
- hence we call the method **(recursive) ray tracing**

Adding Shadows

We've found the nearest surface hit for a given ray

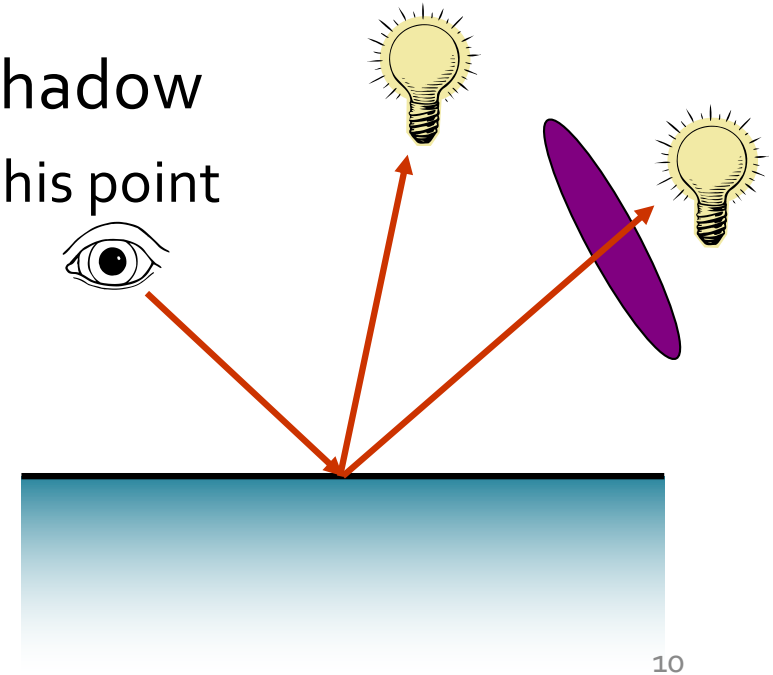
- now we want to shade this point
- for each light, we evaluate our diffuse & specular terms

For a given light, the point may or may not be in shadow

- if it is in shadow, don't add contribution for this light

We can easily test whether we're in shadow

- trace a new **shadow ray**, starting at this point
- heading towards the given light
- hit any surface closer than light?
  - yes: we're in shadow
  - else: no shadow



Computing Correct Reflections

If we're tracing rays, we can easily compute correct reflections

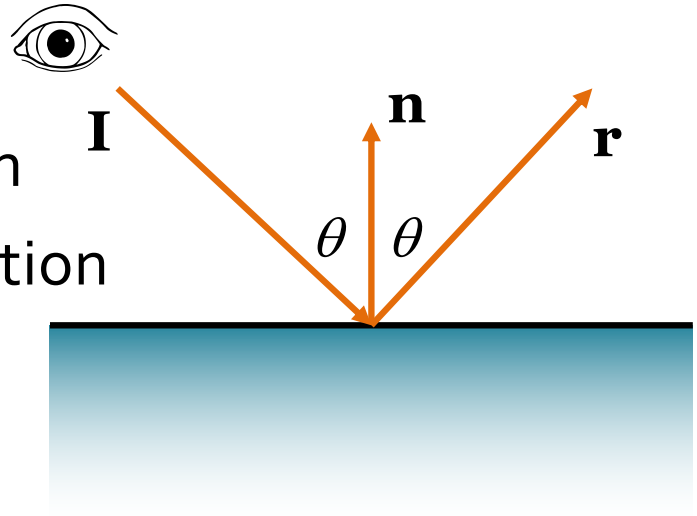
- at a given point, the perfect reflected direction is

$$\mathbf{r} = \mathbf{I} - 2(\mathbf{n} \cdot \mathbf{I})\mathbf{n}$$

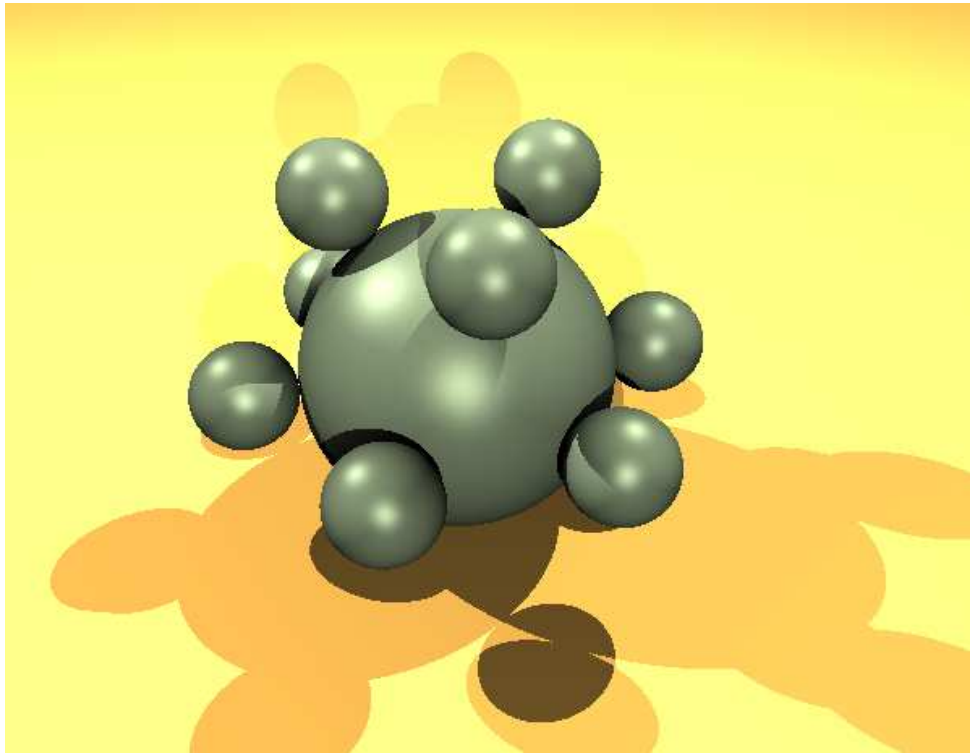
- trace a ray going in that direction
- the returned color is the reflection

Remember to keep in mind

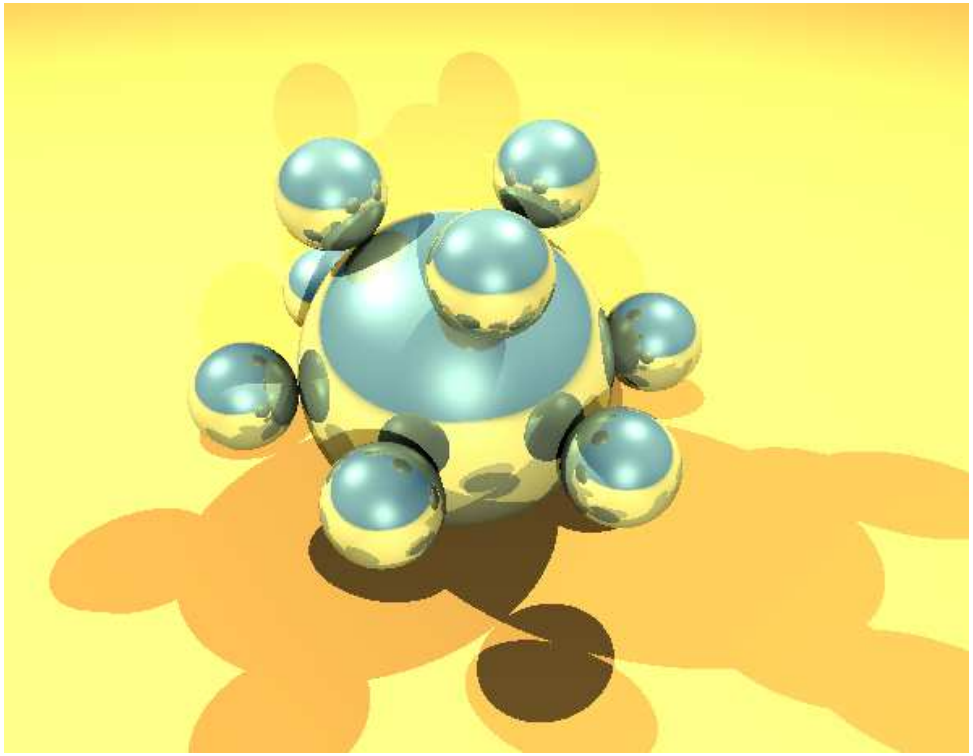
- these arrows are the tracing direction
- light is propagating in opposite direction



An Example: Adding Reflected Rays

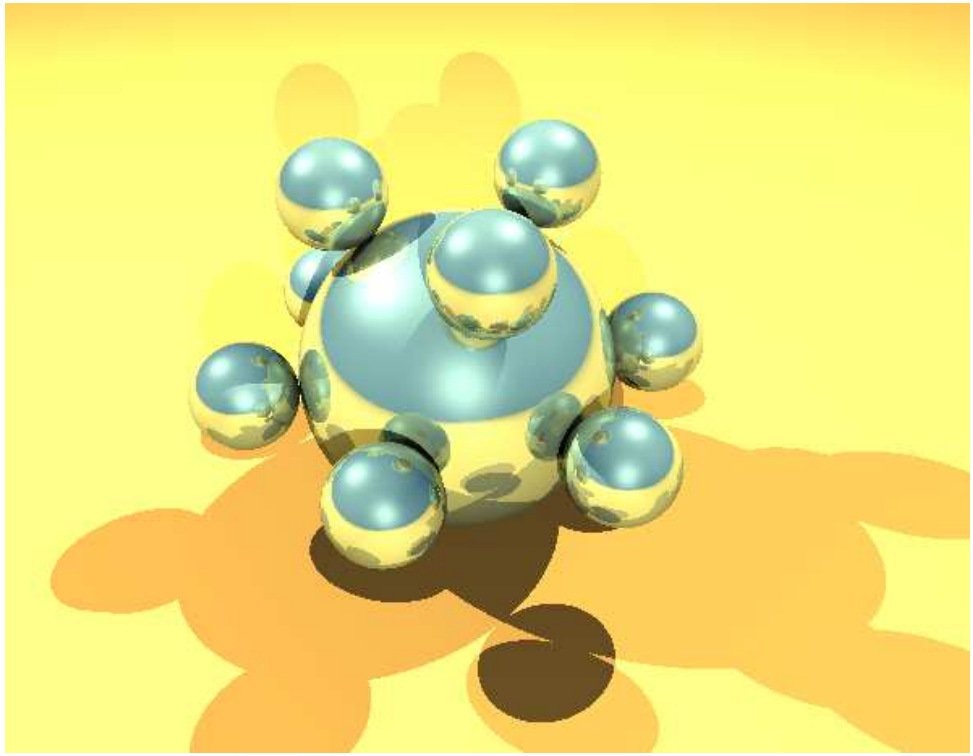


No recursive rays

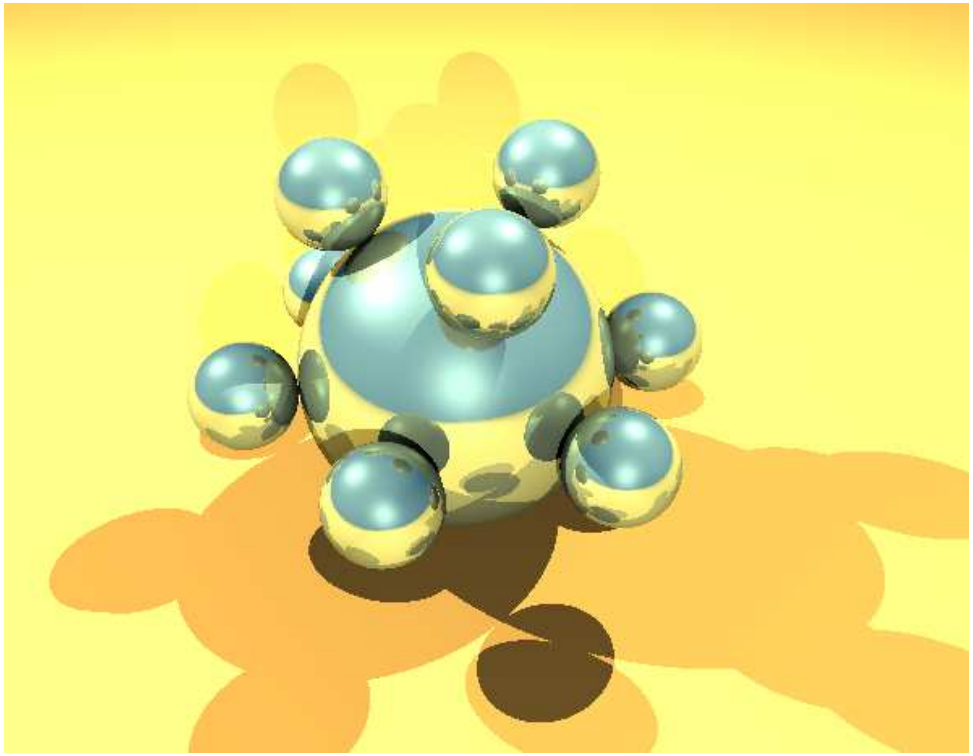


1 level of recursive reflection

An Example: Adding Reflected Rays



2 levels of recursive reflection



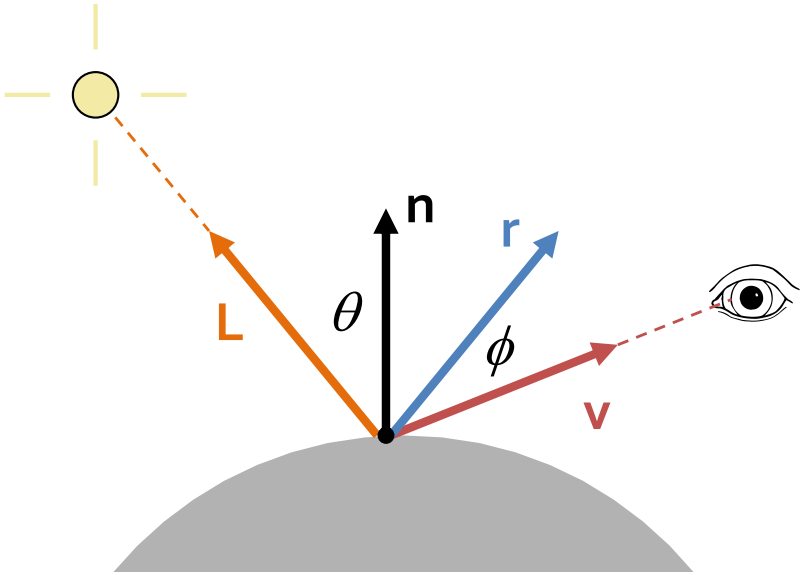
1 level of recursive reflection

Looking at the Shading Model

Recall our current Phong illumination model

$$I = \sum_{\text{all lights L}} I_L k_d (\mathbf{n} \cdot \mathbf{L}) + I_L k_s (\mathbf{r} \cdot \mathbf{v})^n$$

diffuse reflection — specular highlight



- sums contributions from all lights
- plus the global ambient glow

We'll add three important new components

- shadows, specular reflections, and specular transmission
- our general strategy: recursively trace rays to evaluate shading
- hence we call the method **(recursive) ray tracing**

Refractive Transparency

OpenGL supports limited transparency

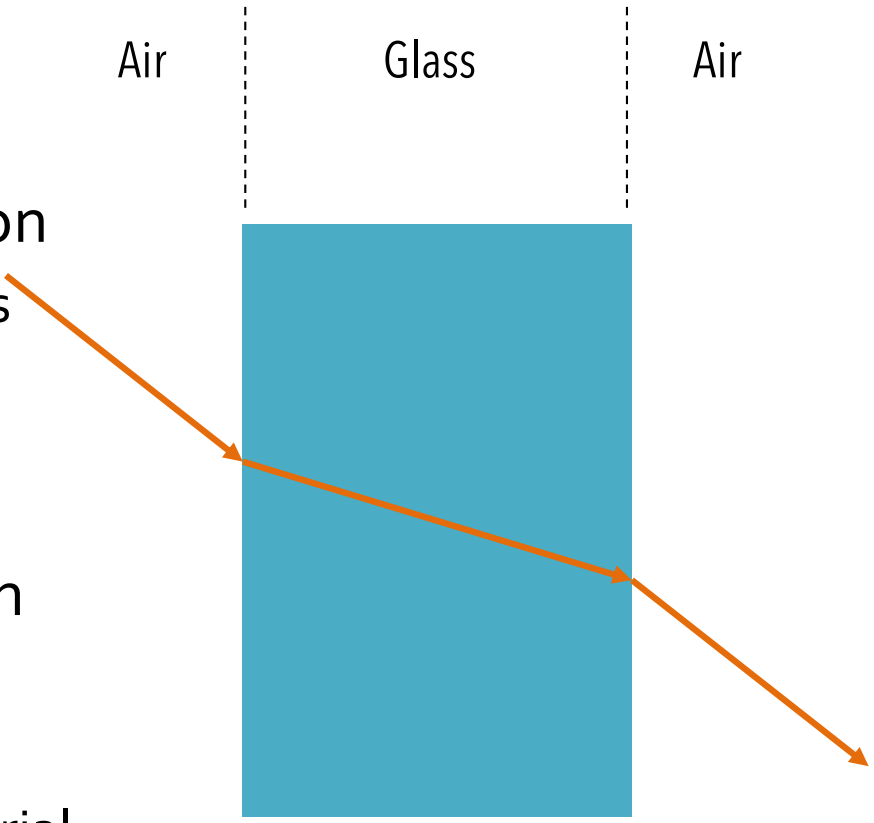
- enable alpha blending
- render objects back to front

OpenGL doesn't account for refraction

- light rays bent at material boundaries
- accounts for lenses among other

Ray Tracing can account for refraction like reflection

- when shading a given point
- trace a transmitted ray into the material
- need to compute refracted direction

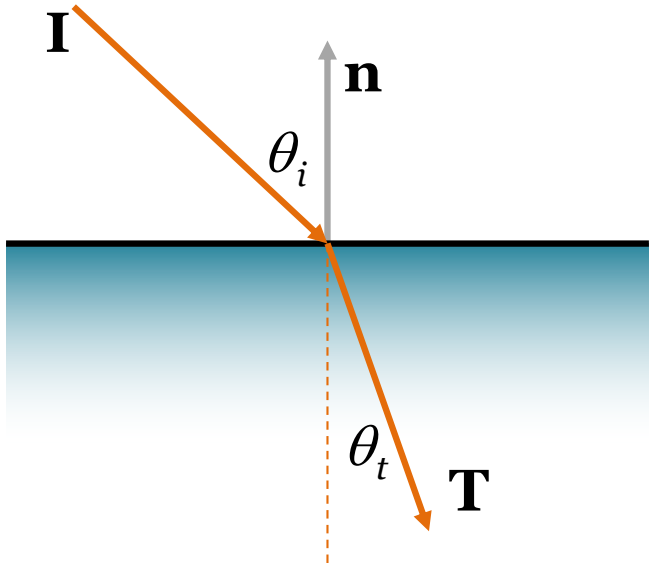


Refraction of Light

Rays transiting between materials are bent around normal

- every material has an index of refraction

Material	Index of Refraction
vacuum	1.0
ice	1.309
water	1.333
ethyl alcohol	1.36
glass	1.5–1.6
diamond	2.417



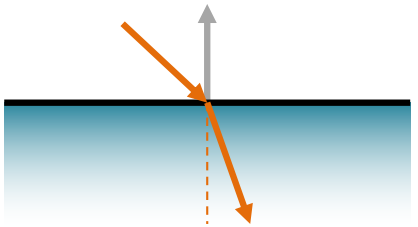
Angles with surface normal obey Snell's Law

$$\frac{\sin\theta_i}{\sin\theta_t} = \eta_{ti} = \frac{\eta_t}{\eta_i} \quad \text{where } \eta \text{ are indices of refraction}$$

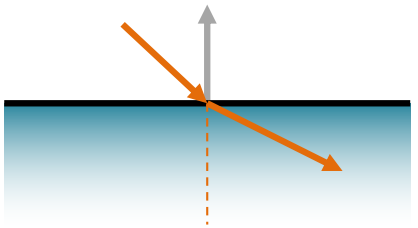
Refraction of Light

Refractive indices determine amount of bending

- going from low index to higher index
  - ray is bent towards the normal
  - for example: air to glass



- going from high index to lower index
  - ray is bent away from the normal
  - for example: glass to water



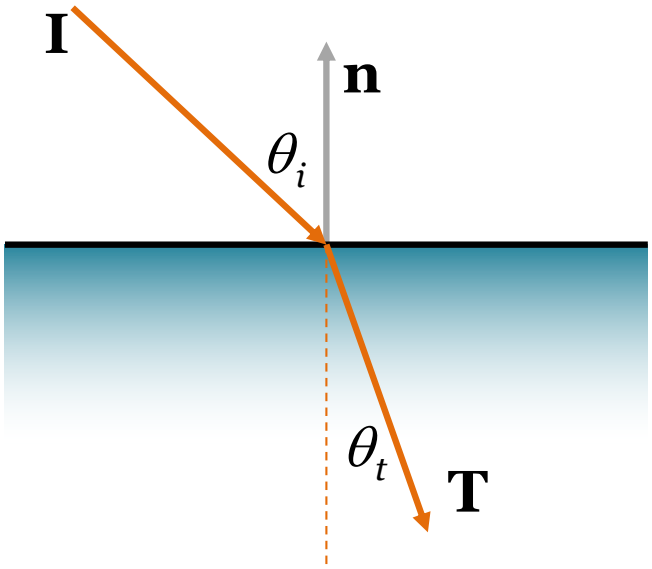
Technically, this is a function of wavelength

- that's why prisms work (and why you see rainbows)
- but for our purposes here, we'll ignore this detail

Computing the Transmitted Ray

Angles of the incoming & transmitted rays obey Snell's Law

$$\frac{\sin\theta_i}{\sin\theta_t} = \eta_{ti} = \frac{\eta_t}{\eta_i}$$



- this isn't terribly convenient
- need transmitted direction vector

With a little math, we can derive the transmitted direction

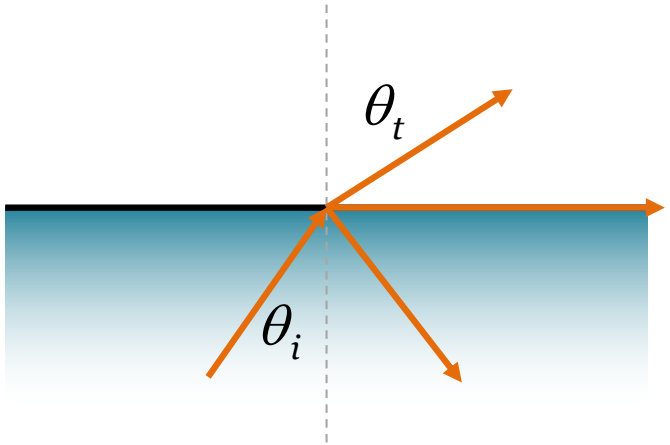
$$\mathbf{T} = \eta\mathbf{I} + \left(\eta c - \sqrt{1+\eta^2(c^2-1)}\right)\mathbf{n}$$

$$\text{where} \quad c = \cos\theta_i = -\mathbf{n}\cdot\mathbf{I} \quad \text{and} \quad \eta = \frac{\eta_i}{\eta_t}$$

One Last Refractive Detail

When entering material of lower index

- ray bends outward from normal
- what if the angle is more than 90°?
  - ray is actually reflected off the boundary
  - this is called **total internal reflection**
  - and it's how fiber optics work



Total internal reflection occurs when

$$\theta_i > \theta_{\text{critical}} \quad \text{where} \quad \theta_{\text{critical}} = \sin^{-1}\frac{\eta_t}{\eta_i}$$

- just need to check for this critical angle
- if above it, use specular reflection for "transmission"
- if we're exactly at the critical angle, things are a little weird

Take a look at the YouTube video: https://youtu.be/NAaHPRsveJk

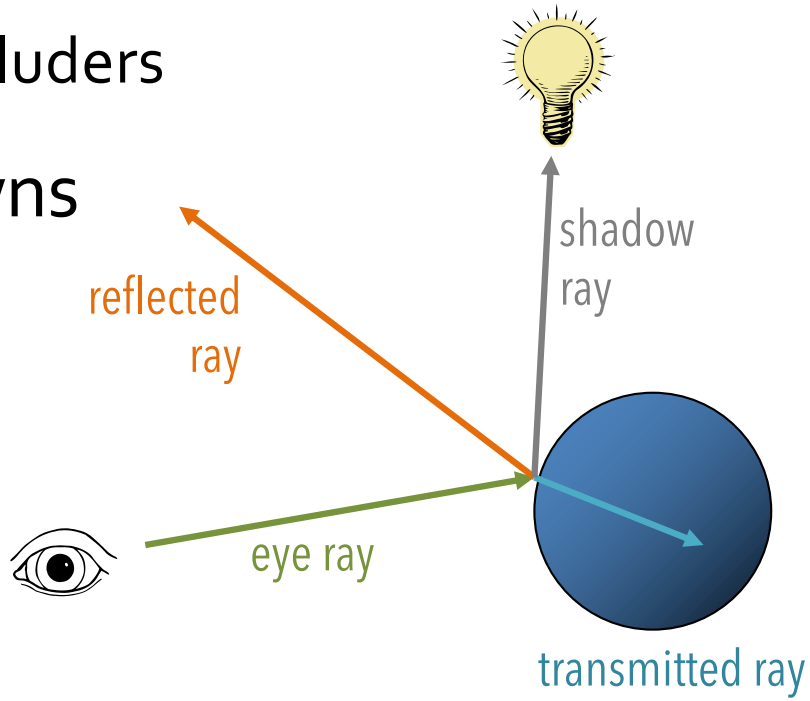
Classification of Rays

We've now seen four kinds of rays in the world

- eye rays that leave the eye through a pixel
- reflected rays that bounce off surfaces
- transmitted rays that travel through them
- shadow rays which test for occluders

Every surface intersection spawns

- 1 reflected ray
- 1 transmitted ray
- 1 shadow ray per light

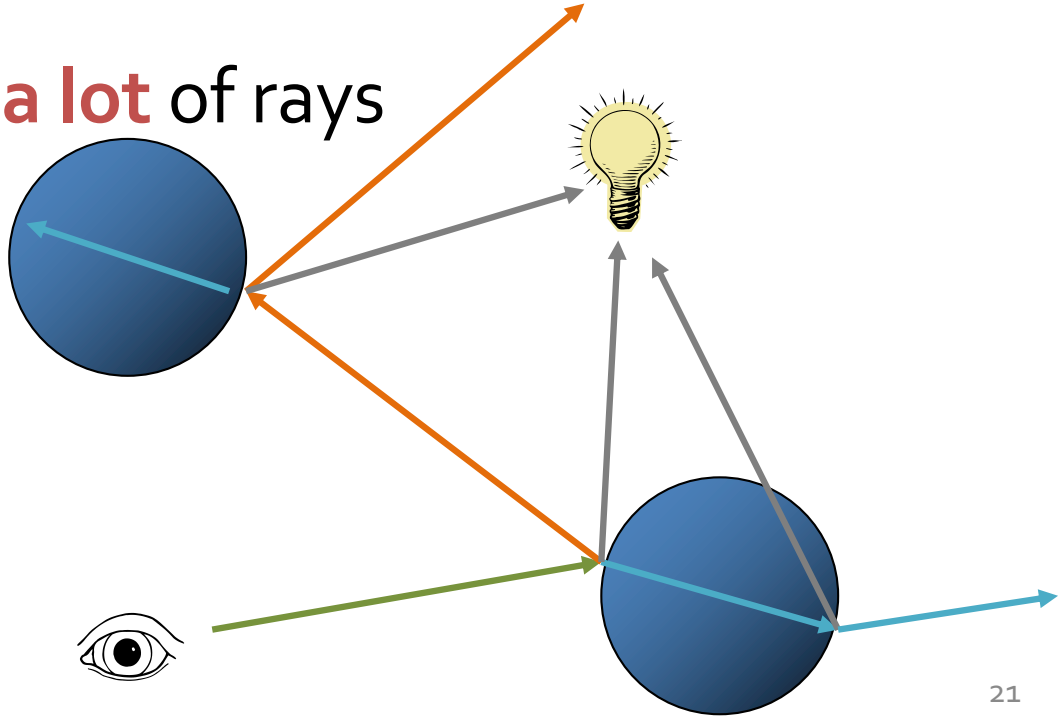


Ray Recursion

Recursive ray tracing spawns a whole tree of rays

- when eye ray hits a surface, we spawn reflected & transmission rays
- when either of them hits a surface, they spawn 2 more
- typically impose maximum recursion limit

We will wind up tracing **a lot** of rays



Structure of a Simple Ray Tracer

```
void raytrace()
 for all pixels (x,y)
     image(x,y) = trace(compute_eye_ray(x,y))


rgbColor trace(ray r)
  for all surfaces s
    t = compute_intersection(r, s)
    closest_t = MIN(closest_t, t)

  if( hit_an_object )
    return shade(s, r, closest_t)
  else
    return background_color
```

This all looks identical to a simple ray caster

Structure of a Simple Ray Tracer

```
rgbColor shade(surface s, ray r, double t)
  point x = r(t)
  rgbColor color = black

  for each light source L
    if( closest_hit(shadow_ray(x, L)) >= distance(L) )
      color += shade_phong(s, x)

  color += k_specular * trace(reflected_ray(s,r,x))

  color += k_transmit * trace(transmitted_ray(s,r,x))

  return color
```

Here's where ray tracing is different from ray casting

- it's the recursive calls to trace()
- to resolve shadows, reflection, and transmission

Ray Tracing Efficiently

The primary path to efficiency

- avoid tracing rays whenever possible
- and above all, avoid ray–surface intersection tests

A ray tracing system can be easily overcome with rays

- minimum 1 eye ray per pixel, many more with supersampling
- recursion depth k yields $2^{k+1} - 1$ rays traced per eye ray
  - counting reflection & transmission rays but *not* shadow rays

https://www.siggraph.org/education/materials/HyperGraph/aliasing/alias2c.htm

Ray Tracing Efficiently

Consider this example

- image resolution of 1024 × 768 = 786,432 pixels
- 3 × 3 supersampling = 7 million eye rays
- recursion depth 5 = 63 × 7 = 441 million rays
- each tested against 10,000 polygons
- 4.4 trillion intersection tests (ignoring shadow rays)

Spatial Data Structures

Probably the single most important efficiency improvement

- divide space into cells
- record what geometry lies in each cell
- first test rays against cell
- only check geometry within cell if the ray actually hits the cell

Several data structures in common practice

- hierarchical bounding volumes
- BSP trees
- octrees
- regular 3-D grids

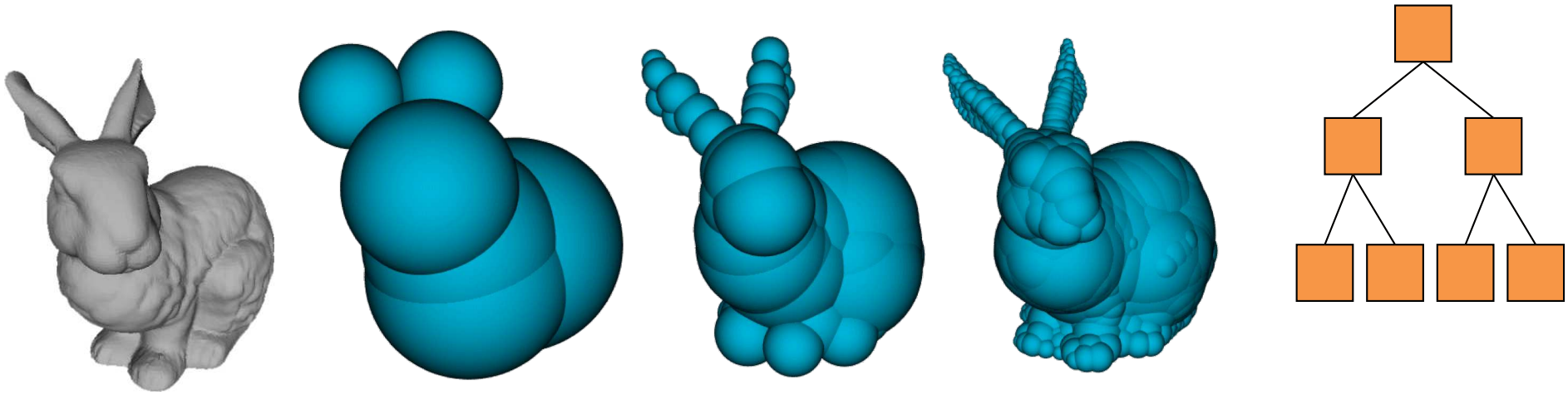
Hierarchical Bounding Volumes

Begin by intersecting ray with the root volume

- if no intersection, ignore all child volumes
- otherwise, recursively test child volumes

Test only objects whose bounding volume is actually hit by ray

- hopefully ignoring most of the scene
- but this depends a lot on the structure of the hierarchy



[isg.cs.tcd.ie/spheretree/]

BSP Trees, Octrees, and Grids

Use BSP tree to traverse scene from front to back

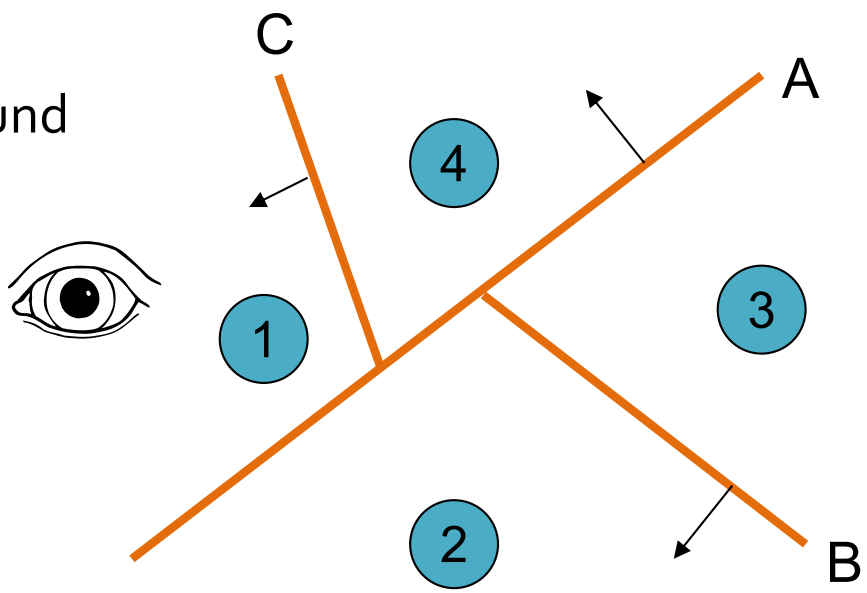
- start at root plane
- figure out which side the viewpoint is on
- descend on that side first; do this recursively

Helps us avoid unnecessary work

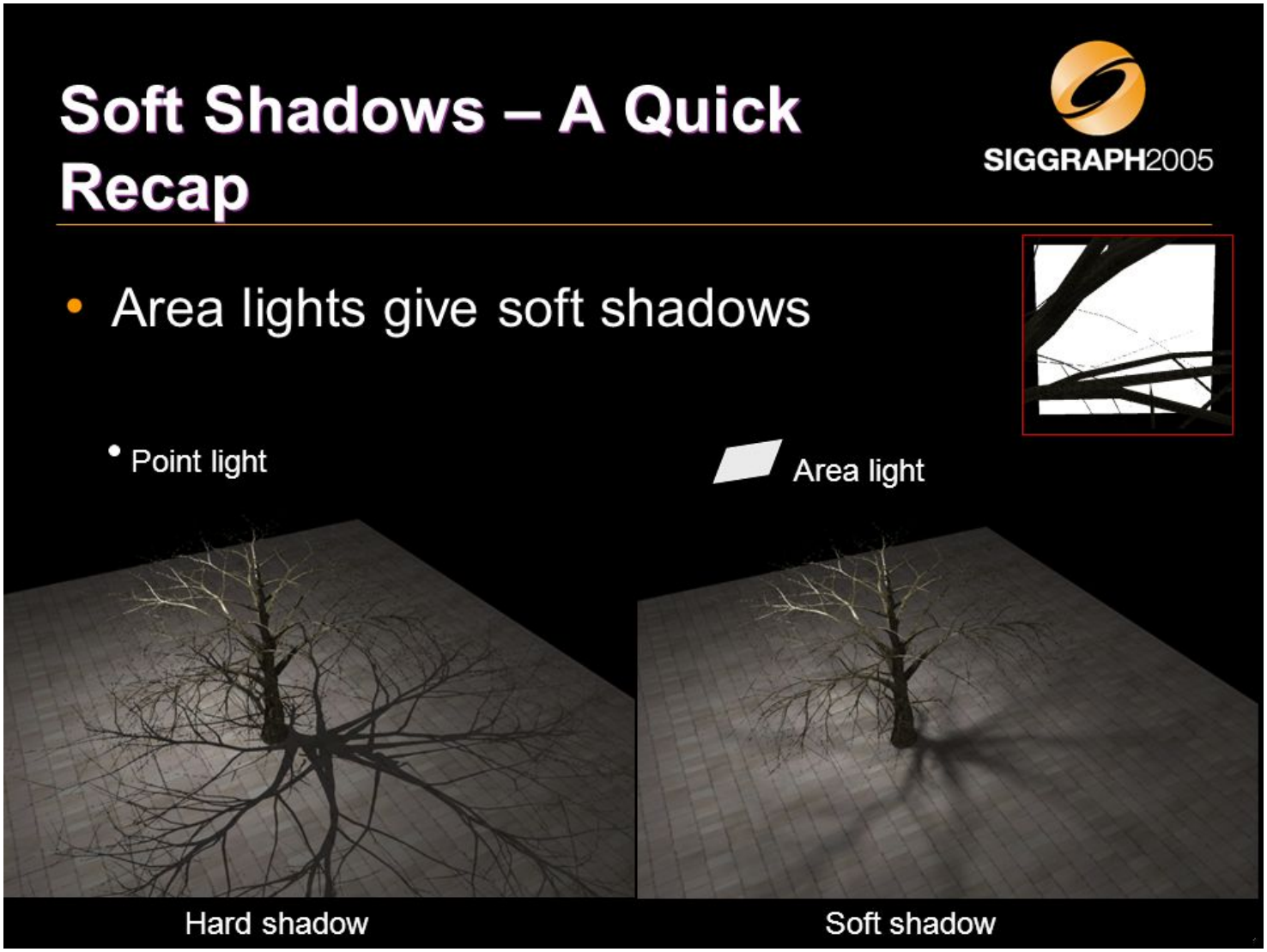
- can tell when closest intersection found
- because we're going front to back

Can use grids & octrees similarly

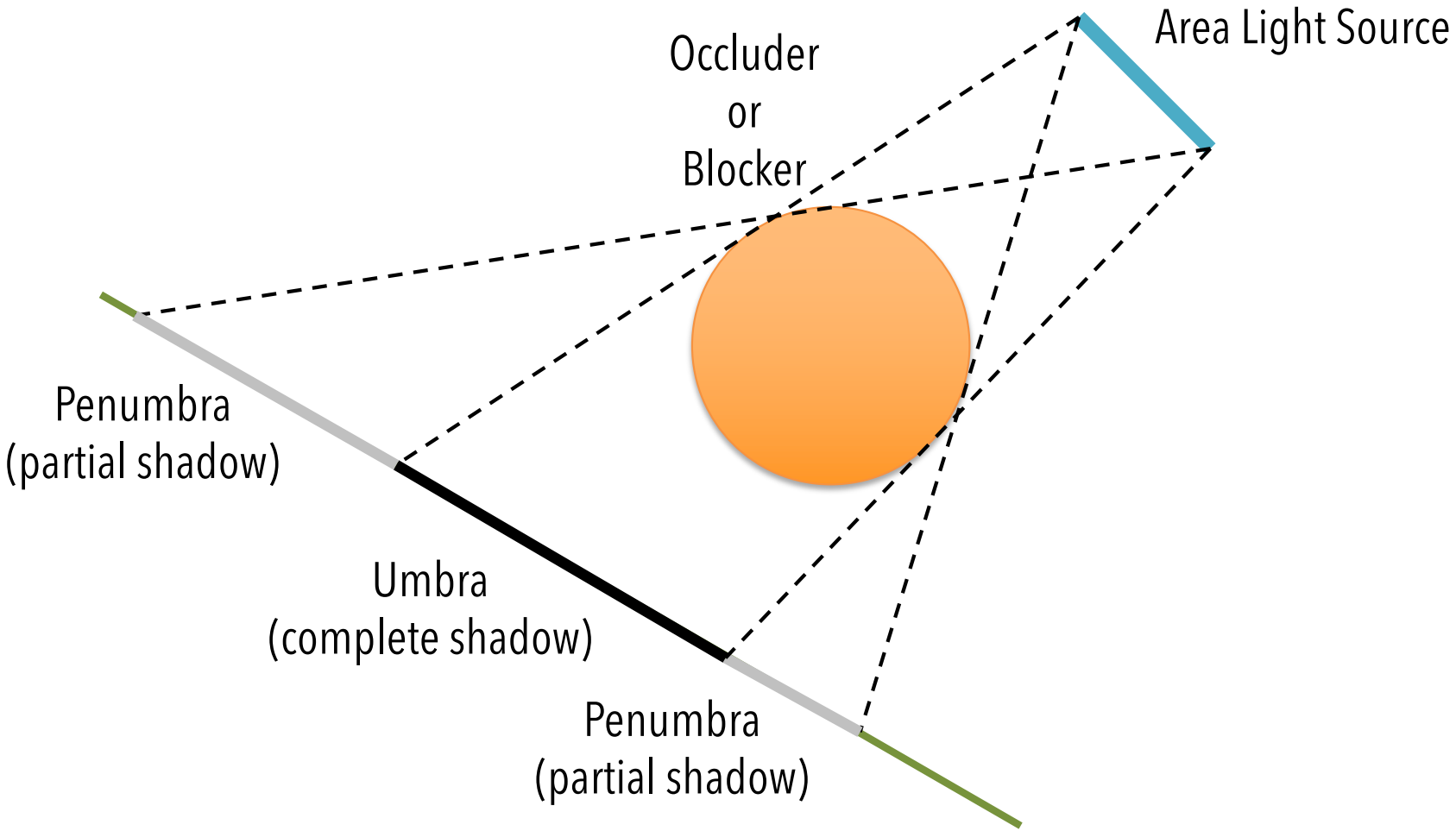
- traverse grid with 3D DDA algorithm
- octree a little trickier



Shadows


Soft Shadows – A Quick Recap
SIGGRAPH2005
Area lights give soft shadows
Point light
Area light
Hard shadow
Soft shadow

The Structure of Soft Shadows



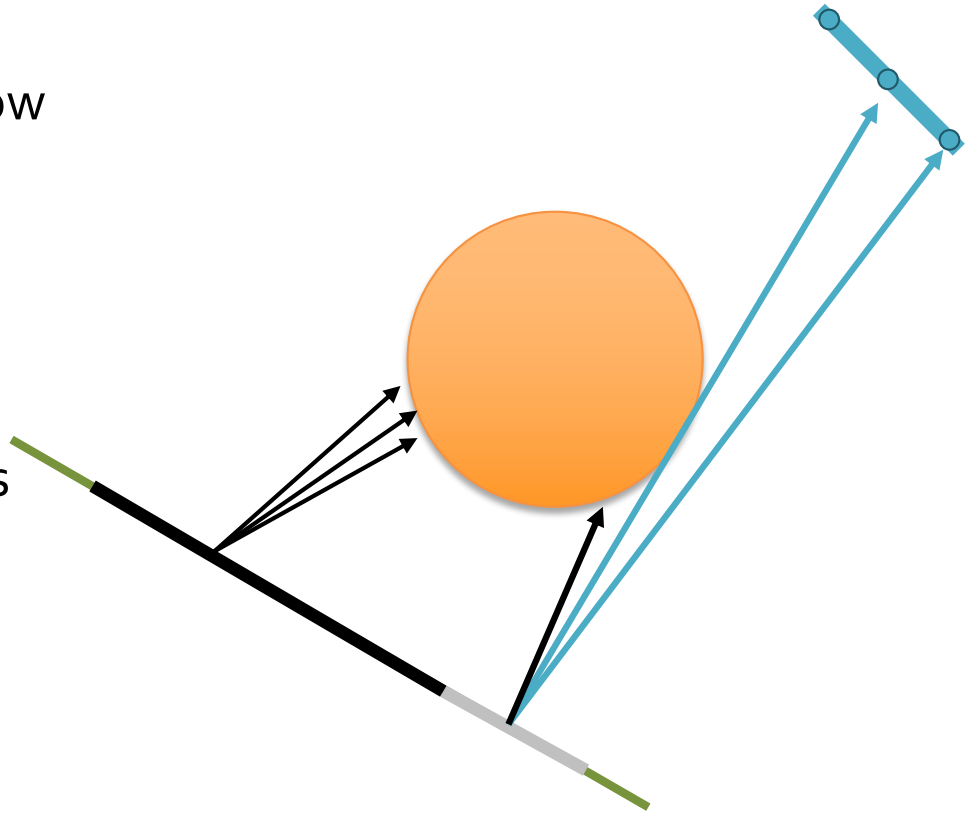
Soft Shadows

Our previous shadow method

- for the point we're shading
- cast a ray towards point light
- hit surface before light = shadow
- otherwise no shadow

Extends directly to area lights

- sample multiple spots on light
- look at fraction hitting surfaces
- indicates level of shadow
  - none hit = full illumination
  - all hit = full shadow
  - some hit = partial shadow

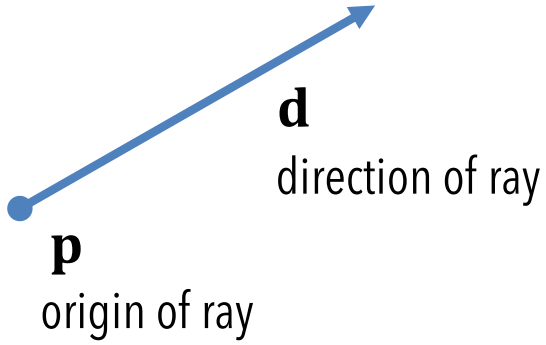


Representing Light Rays

Geometrically, a ray is just a starting point plus a direction

- the set of all points described by

$$\mathbf{x}(t) = \mathbf{p} + t\mathbf{d} \quad \text{where} \quad t > 0$$



- implementation tip:
  make sure $\mathbf{d}$ is unit vector

Each ray will return some amount of light from the world

- for implementation purposes, an RGB color

Computing Ray–Surface Intersections

We start with an equation of our ray

$$\mathbf{x}(t) = \mathbf{p} + t\mathbf{d} \quad \text{where} \quad t > 0$$

General idea:

write equation for point on both ray & surface

- for an implicit surface $f(\mathbf{x}) = 0$
  - substitute ray equation

$$f(\mathbf{p} + t\mathbf{d}) = 0$$

- for a parametric surface $\mathbf{x} = S(u, v)$
  - find location where distance between ray and surface is o

$$S(u, v) - (\mathbf{p} + t\mathbf{d}) = 0$$

in general, both approaches can require expensive root finding

Ray–Plane Intersection

For specific surfaces, can derive more efficient methods

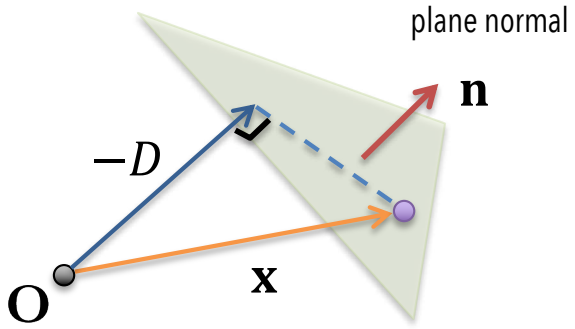
We start with the equation for the plane $\mathbf{n} \cdot \mathbf{x} + D = 0$

Then substitute the ray equation into it and solve for t

$$\mathbf{n} \cdot (\mathbf{p} + t\mathbf{d}) + D = 0$$

$$\mathbf{n} \cdot \mathbf{p} + t\mathbf{n} \cdot \mathbf{d} + D = 0$$

$$t_0 = \frac{-(\mathbf{n} \cdot \mathbf{p} + D)}{\mathbf{n} \cdot \mathbf{d}}$$



The ray hits the plane if and only if $t_0 \geq 0$

To find the actual intersection point of the ray with the plane

- substitute the solution t_0 back into the ray equation

Ray–Polygon Intersection

First, find the intersection of the ray and the polygon's plane

Then we just need to determine whether this point is in the polygon

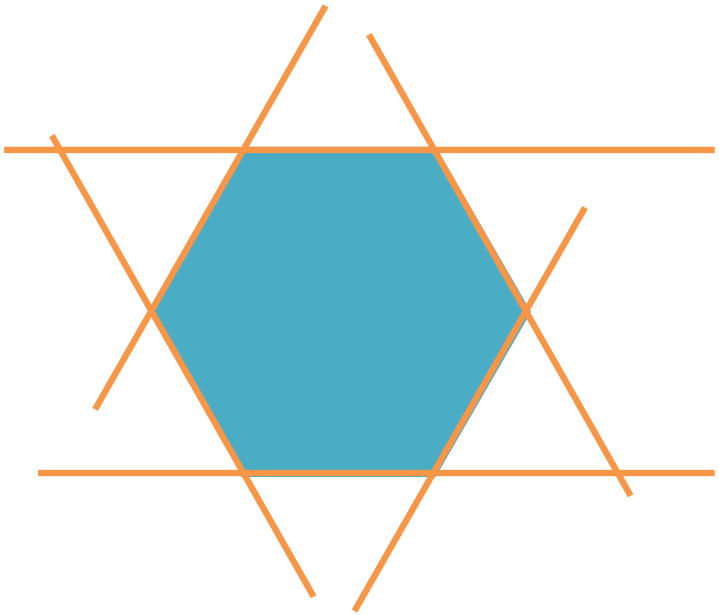
- there are many approaches to point-in-polygon testing

Point-in-Polygon Test for Convex Polygons

For general convex polygons, we can use half-space tests

- construct lines $a_i x + b_i y + c_i = 0$ through each edge of the polygon
- must make sure that normals are consistently oriented
  - either they all point in or they all point out
- the point (x, y) is in the polygon if

  $a_i x + b_i y + c_i$ all have the same sign



Note that this also works in 3-D

- construct planes through each edge
- perpendicular to polygon plane
- point must lie on inside of all of them

Point-in-Triangle Tests

With triangles, can make good use of barycentric coordinates

- all points in triangle satisfy the equation

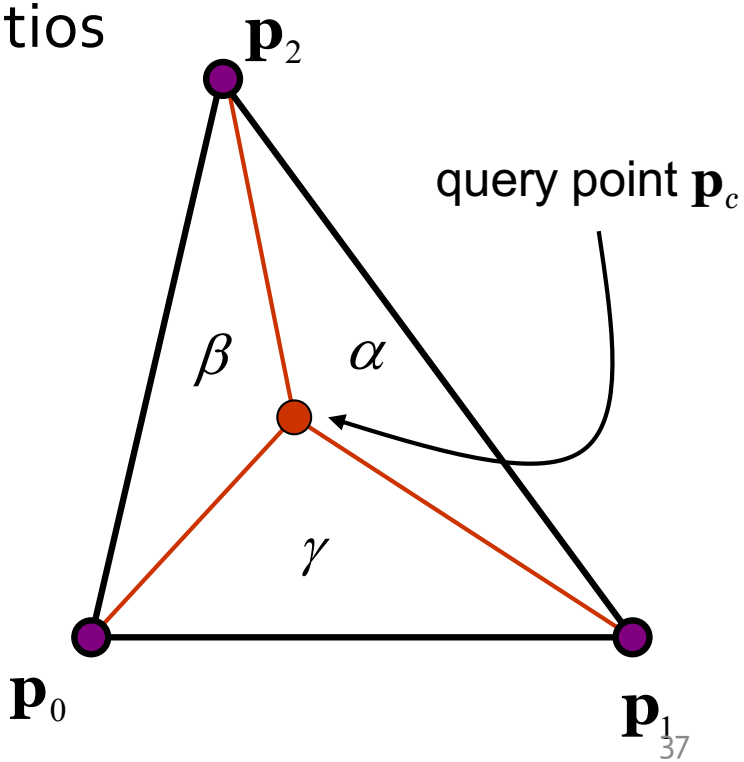
$$\mathbf{p} = \alpha\mathbf{p}_0 + \beta\mathbf{p}_1 + \gamma\mathbf{p}_2 \text{ where } \alpha + \beta + \gamma = 1$$

- these coefficients are triangle area ratios

$$\alpha = \frac{\text{Area}(\mathbf{p}_c, \mathbf{p}_1, \mathbf{p}_2)}{\text{Area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)}$$

$$\beta = \frac{\text{Area}(\mathbf{p}_c, \mathbf{p}_2, \mathbf{p}_0)}{\text{Area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)}$$

$$\gamma = \frac{\text{Area}(\mathbf{p}_c, \mathbf{p}_0, \mathbf{p}_1)}{\text{Area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)} = 1 - \alpha - \beta$$



Point-in-Triangle Test

We can compute the 2-D area of a triangle as

$$\text{Area}(\mathbf{p}_0,\mathbf{p}_1,\mathbf{p}_2)=\frac{1}{2}\begin{vmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{vmatrix}=\frac{(x_1-x_0)(y_2-y_0)-(x_2-x_0)(y_1-y_0)}{2}$$

- note: this is the signed area of the triangle
  - it's positive if points are in counter-clockwise order
  - and negative if they're in clockwise order

So, to figure out if a given point is in the triangle

- compute it's three barycentric coordinates
- point is inside the triangle exactly when $\alpha,\beta,\gamma>0$
- note that this can also be made to work directly in 3-D

Ray–Sphere Intersection

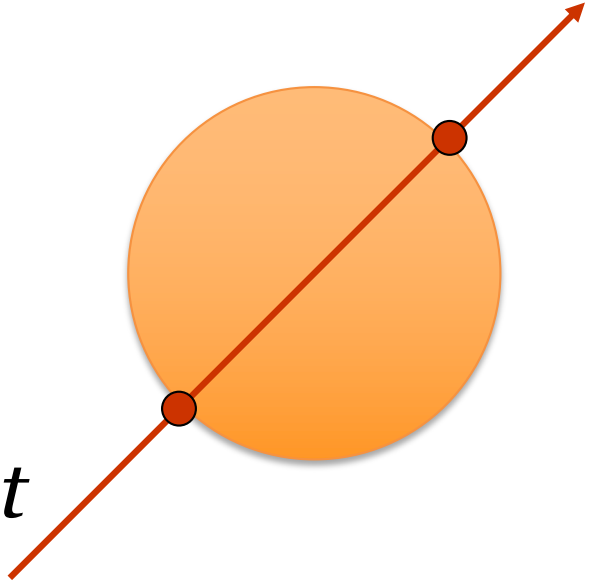
Consider a sphere centered at the origin

$$x^2 + y^2 + z^2 - r^2 = \mathbf{x} \cdot \mathbf{x} - r^2 = 0$$

We substitute the ray equation

$$(\mathbf{p} + t\mathbf{d}) \cdot (\mathbf{p} + t\mathbf{d}) - r^2 = 0$$

$$\mathbf{p} \cdot \mathbf{p} + 2t\mathbf{p} \cdot \mathbf{d} + t^2 \mathbf{d} \cdot \mathbf{d} - r^2 = 0$$



Which gives us a quadratic equation in t

$$At^2 + Bt + C = 0$$

where $A = \mathbf{d} \cdot \mathbf{d} = 1$ ($\mathbf{d}$ is unit vector)

$$B = 2\mathbf{p} \cdot \mathbf{d}$$

$$C = \mathbf{p} \cdot \mathbf{p} - r^2$$

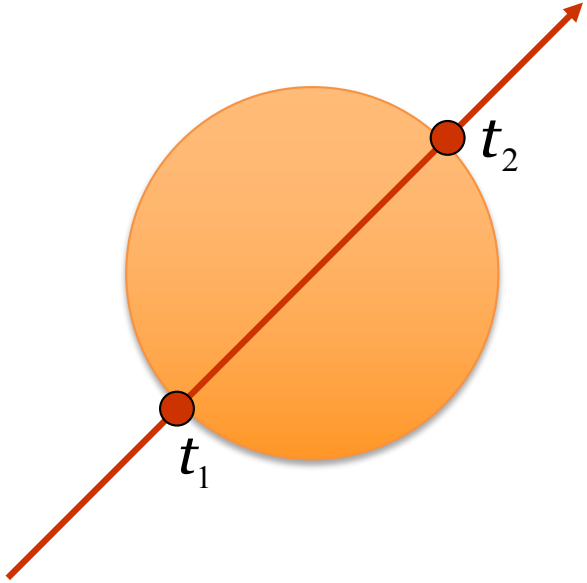
Ray–Sphere Intersection

We can directly solve this quadratic equation

$$At^2 + Bt + C = 0 \qquad \text{where} \quad A = \mathbf{d} \cdot \mathbf{d} = 1 \;\; (\mathbf{d} \text{ is unit vector})$$

$$B = 2\mathbf{p} \cdot \mathbf{d}$$

$$C = \mathbf{p} \cdot \mathbf{p} - r^2$$



For the two intersection locations

$$t_1 = \frac{-B - \sqrt{B^2 - 4C}}{2} \qquad t_2 = \frac{-B + \sqrt{B^2 - 4C}}{2}$$

- the smaller (non-negative) one is the closest ray intersection
- a negative discriminant means that the ray missed the sphere

Ray-Quadrics Intersection

For general quadrics with equation Q: $\mathbf{X}^T\mathbf{A}\mathbf{X} = 0$ where $\mathbf{X}$ is the homogeneous coordinates $(x, y, z, 1)^T$, and $\mathbf{A}$ is a 4×4 real symmetric matrix,

substituting the ray equation $\mathbf{p} + t\mathbf{d}$ to Q, we have

$$(\mathbf{p} + t\mathbf{d})^T\mathbf{A}(\mathbf{p} + t\mathbf{d}) = 0$$

$$\mathbf{p}^T\mathbf{A}\mathbf{p} + 2\mathbf{p}^T\mathbf{A}\mathbf{d}t + \mathbf{d}^T\mathbf{A}\mathbf{d}t^2 = 0 \quad (*)$$

The determinant of this equation is

$$\Delta = 4[(\mathbf{p}^T\mathbf{A}\mathbf{d})^2 - (\mathbf{p}^T\mathbf{A}\mathbf{p})(\mathbf{d}^T\mathbf{A}\mathbf{d})].$$

Ray intersects the quadric

if and only if $\Delta \geq 0$ and a root of $(*)$ is non-negative

COMP3271 Computer Graphics

Color Spaces

2019-20

Objectives

Human Visual System

Color Representations

- Physical color spaces: RGB, CMY
- Perceptual color space: HSV

Examining Color Representation

We've generally treated colors solely as RGB triples

- with an optional alpha value
- perform almost all operations on each channel separately
- but usually treat them as identical
  - trace one transmission ray, despite wavelength dependence

But RGB is not the only representation and we'll explore other color representations

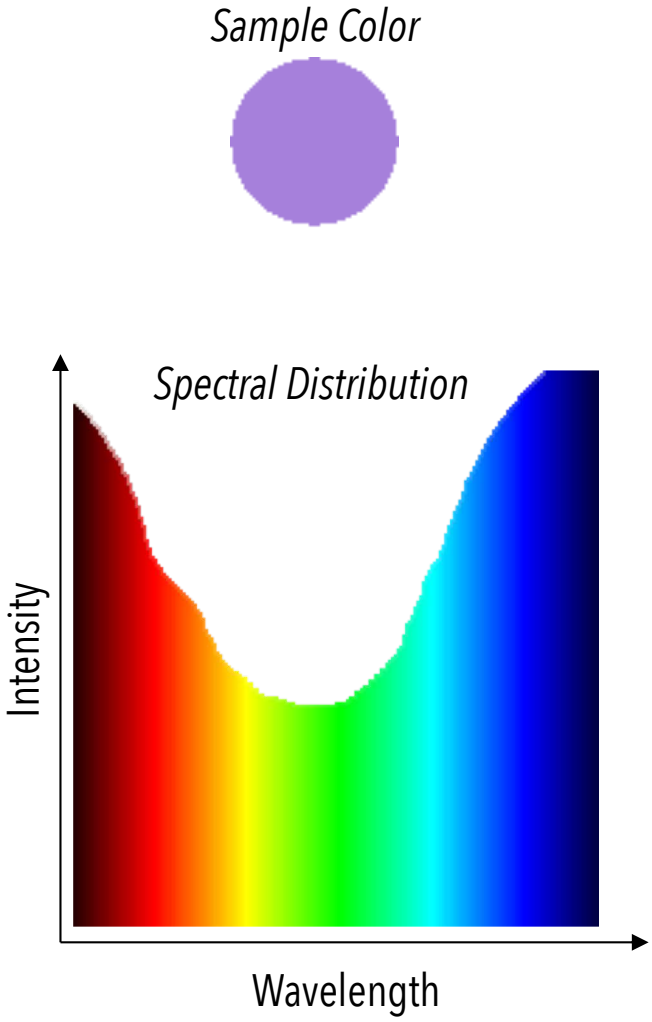
Spectral Distribution of Light

"Light" is a mixture of many wavelengths

- represented by continuous function
  $P(\lambda)$ =intensity at wavelength λ
- spectral distribution: intensity as a function of wavelength over the entire spectrum

We perceive these distributions as colors

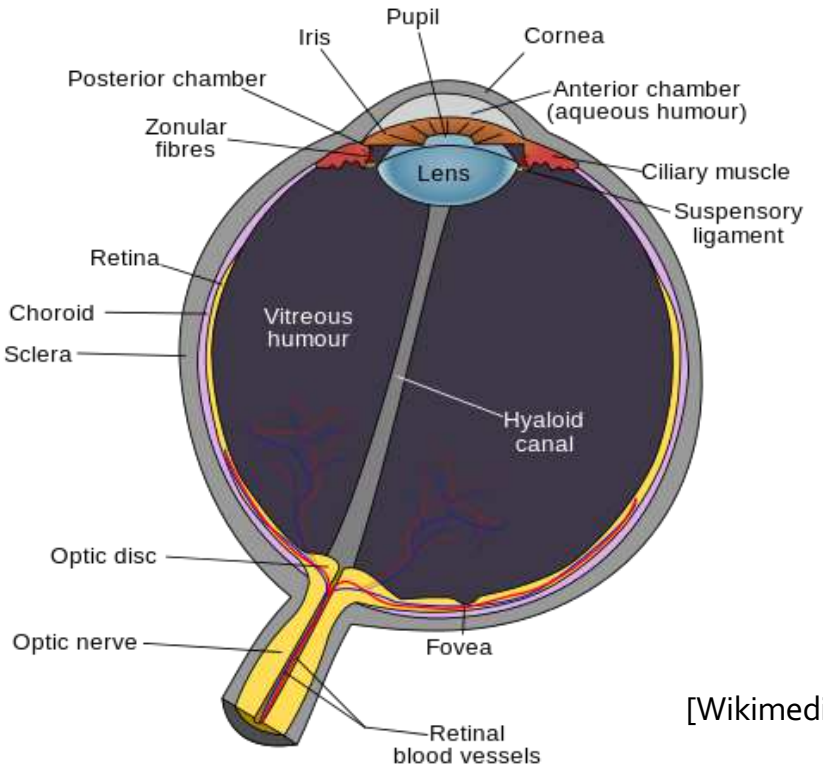
- largely an artifact of our visual system



The Visual System

How do we see?

- Light travels through cornea, pupil, lens, retina, optical nerves, then brain
- Imaging sensors on the retina: rods & cones



[Wikimedia Commons]

Rods and Cones

Cones

- active at normal light levels
- color vision involves cone only

Rods

- sensitive at low light levels
- not sensitive to color
- responsible for our dark-adapted vision
- low influence on color perception

There is an uneven distribution of cones and rods in the retina

Cone Response

Three kinds of cones: S, L, and M

- S cones respond to blue
- M cones respond to green
- L cones respond to red
- much less sensitive to blue

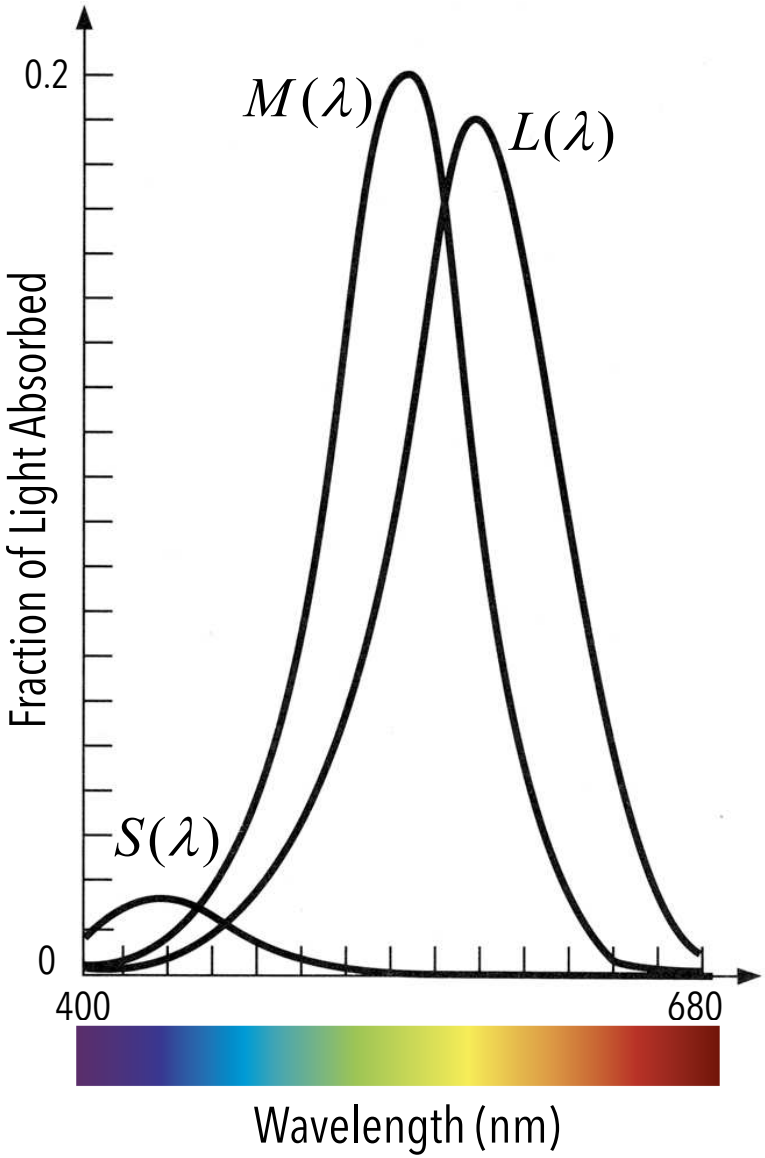
Response levels to illumination are

$$s = \int S(\lambda)P(\lambda)d\lambda$$

$$m = \int M(\lambda)P(\lambda)d\lambda$$

$$l = \int L(\lambda)P(\lambda)d\lambda$$

This implies that we humans perceive light as a 3D space only because we have 3 cone types



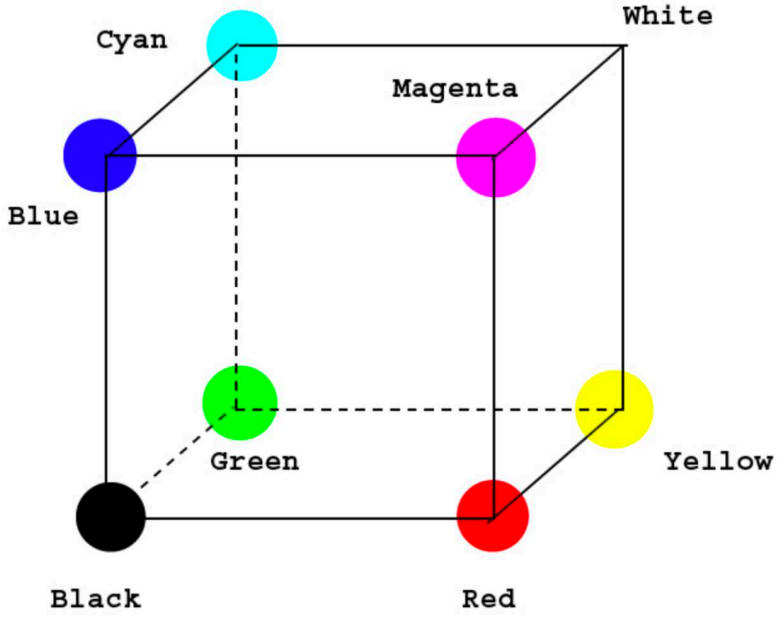
The RGB Color Space

We've represented colors as combinations of three primaries

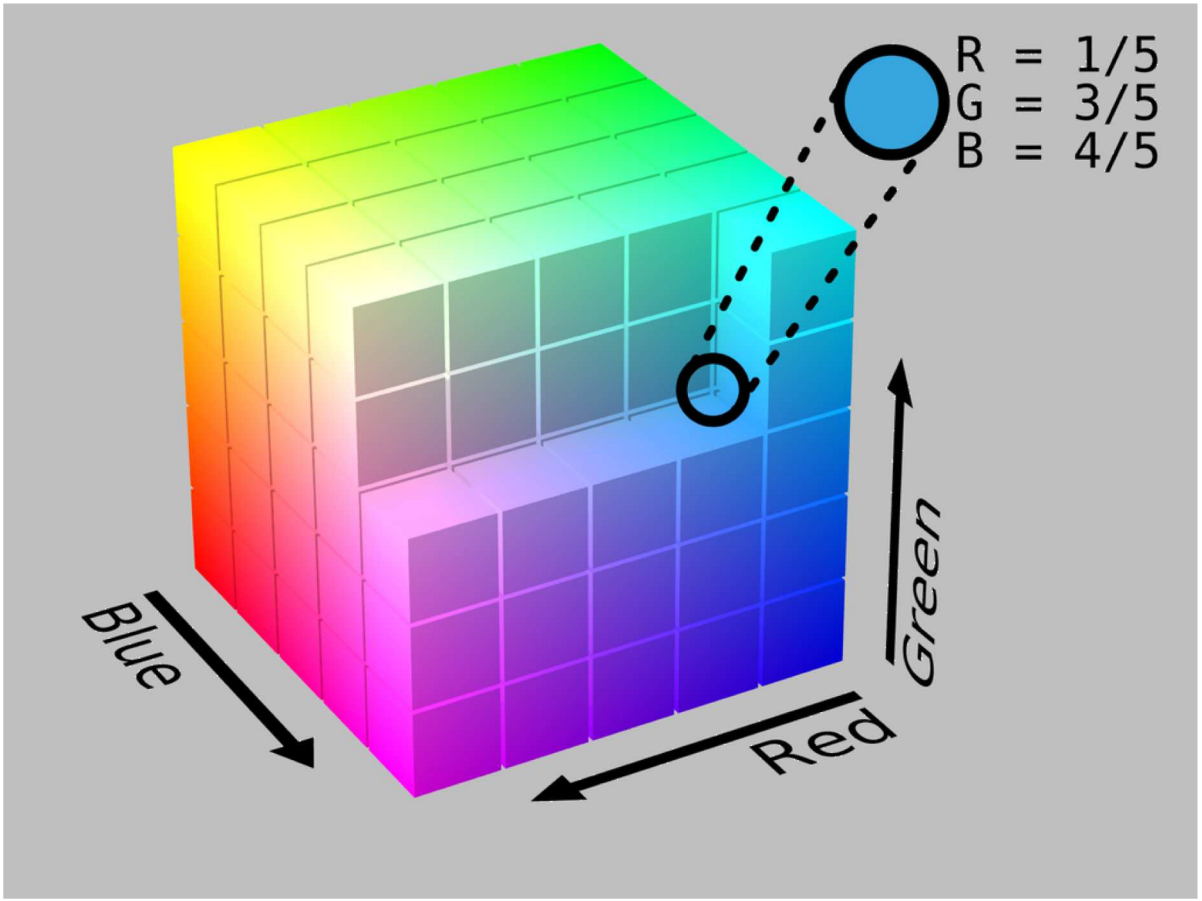
- established 3 special colors (red, green, blue)
- compose all colors by weighted combinations of these

$$C = rR + gG + bB, \text{ where } (r, g, b) \in [0,1]^3$$

- and why did we pick red, green, and blue?
  - essentially because of the structure of our visual system
  - roughly correspond to peaks of cone response curves



The RGB Color Space



Not intuitive to describe a color

Looking Towards Other Color Spaces

Our choice of RGB color space is fairly arbitrary

- it's loosely based on our perceptual system

We could in principle select any 3 primaries we like

- and continue to represent colors as weighted combinations

We can also construct other 3-D color spaces

- where the dimensions are no longer primary colors
- but have some other physical meaning

As we'll see, RGB color space is not always the best choice

- different color spaces lend themselves to different tasks

Other Primary Colors

Red, Green, Blue

- liquid crystal, CRT displays

Red, Yellow, Blue

- paint

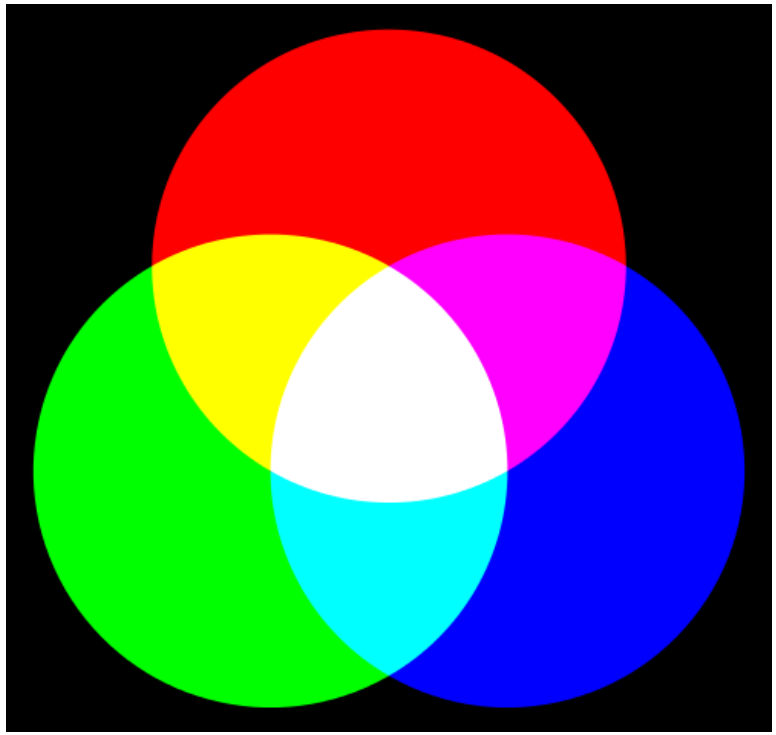
Cyan, Magenta, Yellow

- color printing

Orange, Green, Violet

- color photography

Physical Color Spaces



Additive, RGB

For media that emits light

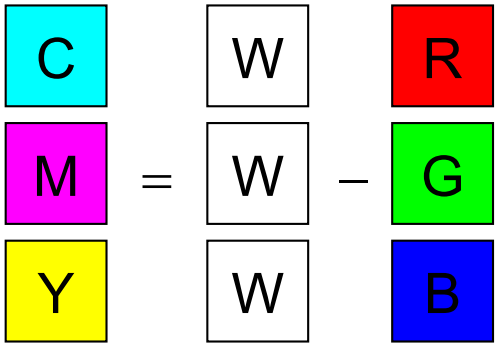
Subtractive, CMY

For media that reflects light

The CMY Color Space

The other most common set of primaries besides RGB

- cyan, magenta, and yellow — complements of red, green, blue



These are the so-called subtractive primaries

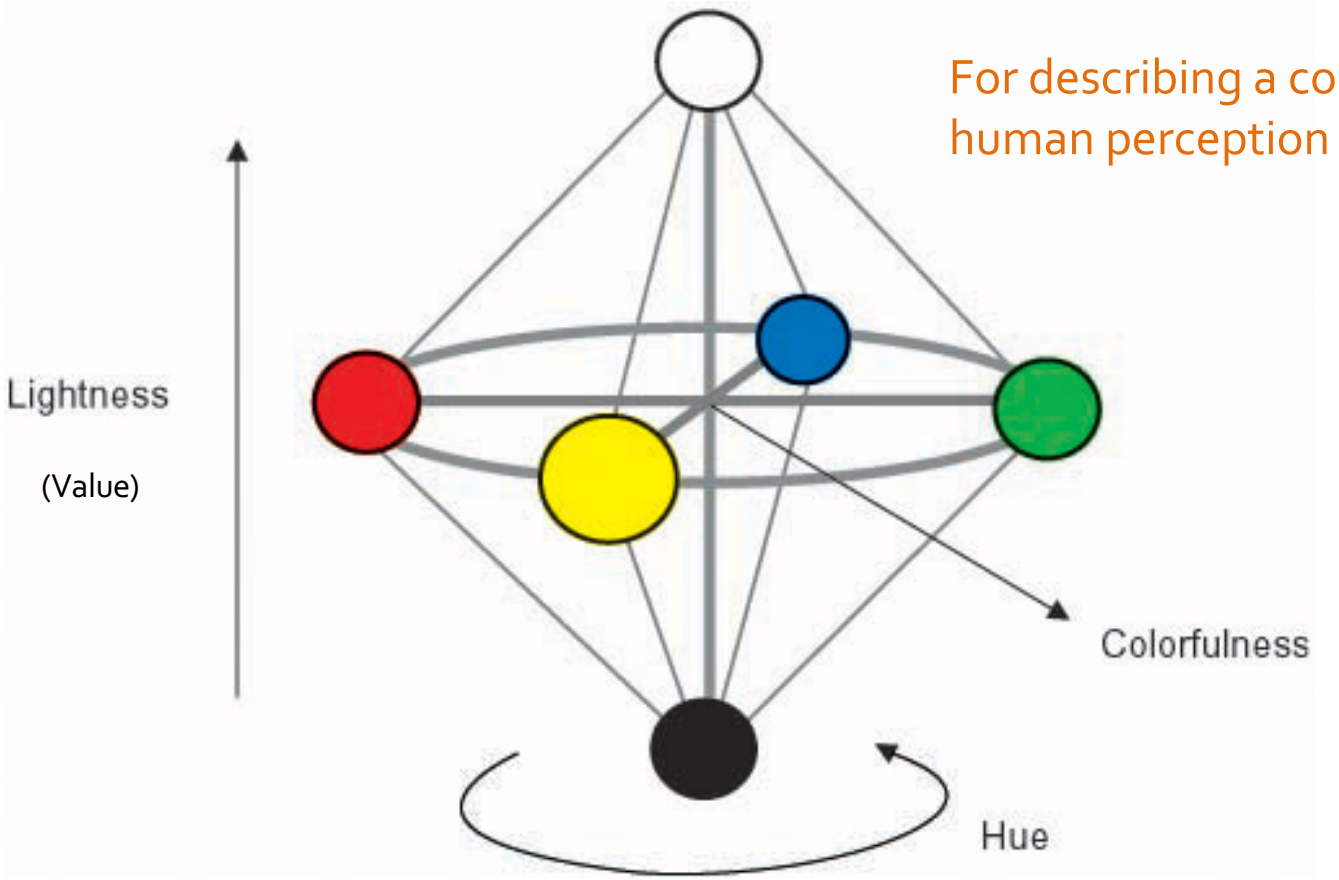
- RGB are additive primaries — start with black, add up to white
  - appropriate when dealing with emitted light
- CMY — start with white and subtract colors from white
  - appropriate when dealing with inks/pigments
  - each ink absorbs some part of the spectrum (subtracts light)

A Color Puzzle

A piece of paper appears green in yellow light, cyan in cyan light, and black in red light.

What color does it appear to have in magenta light?

Perceptual Color Spaces

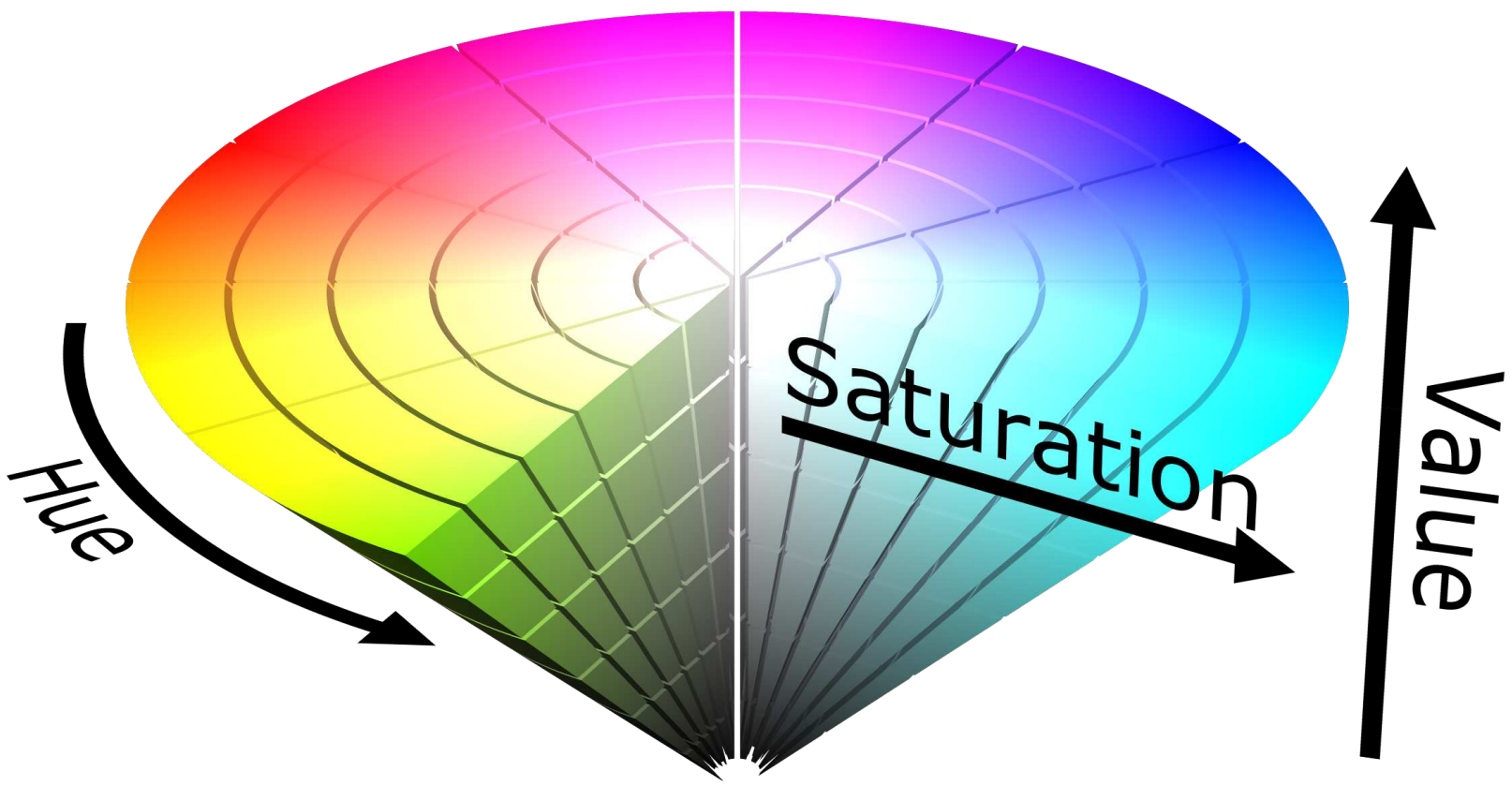


For describing a color based on human perception

[Stone]

How human perceive a color:
hue, **lightness** and **colorfulness (color saturation)**

The HSV Color Space



HSV is Common in User Interfaces