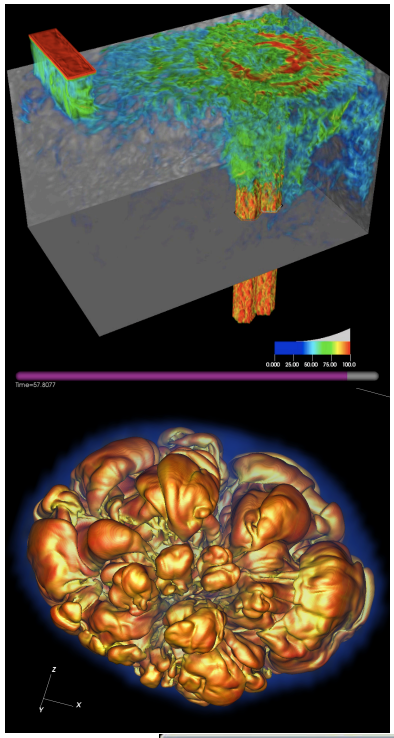
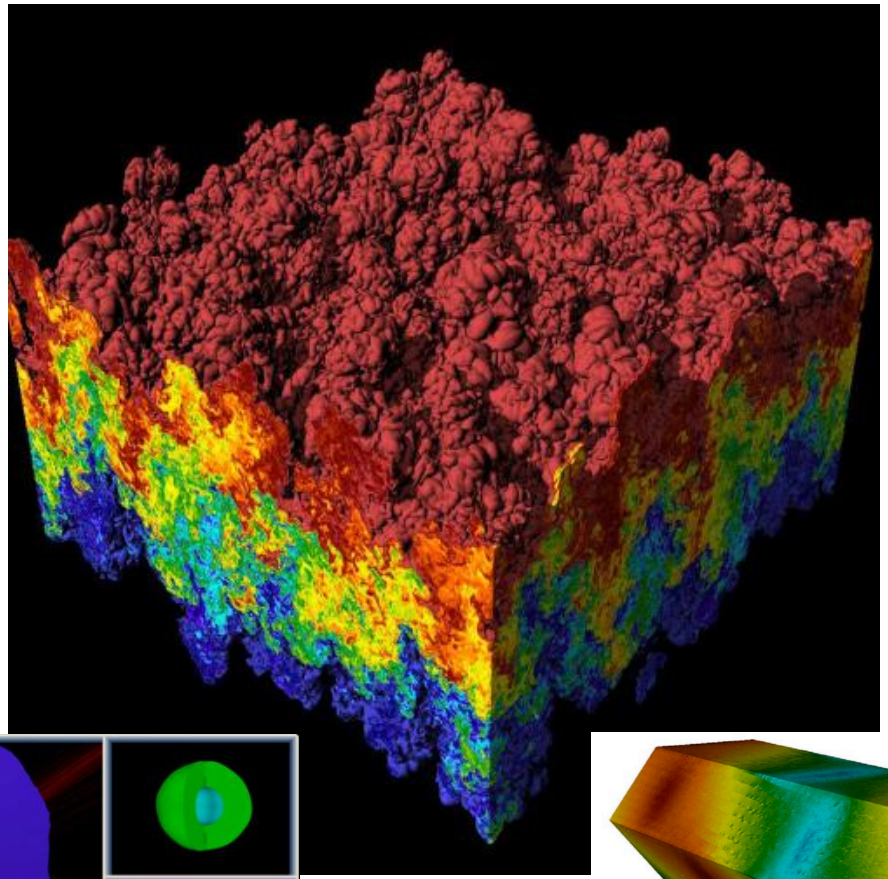
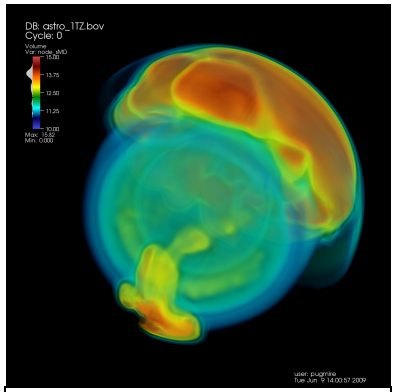
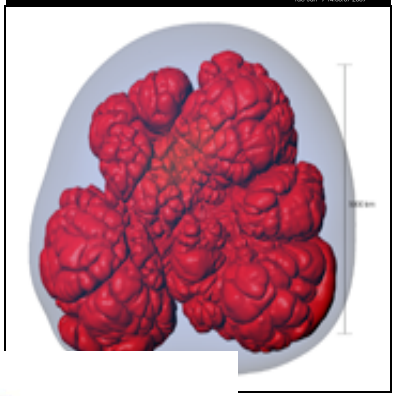
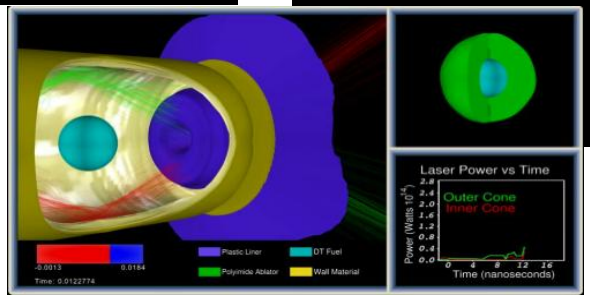
CIS 441/541: Intro to Computer Graphics
Lecture 5: Transforms

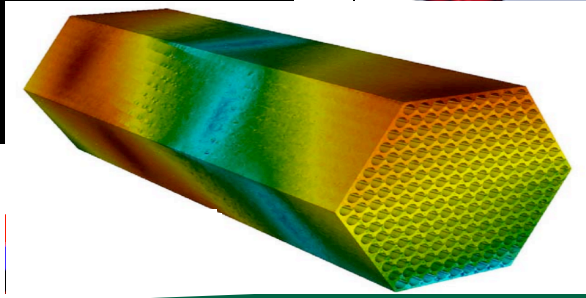












January 24, 2019

Hank Childs, University of Oregon

No Class Tuesday, 1/29

- Will definitely be a YouTube lecture to replace that one.

Office Hours: Weeks 4-10

- Monday: 1-2 (Roscoe)
- Tuesday: 1-2 (Roscoe)
- Wednesday: 1-3 (Roscoe)
- Thursday: 1130-1230 (Hank)
- Friday: 1130-1230 (Hank)

Office Hours: Week 3

- Monday: 415-530 (Hank)
- Tuesday: 1-2, 2-3 (Roscoe)
- Wednesday: 1-3 (Roscoe)
- Thursday: 1130-1230 (Hank)
- Thursday: 1230-230 (Roscoe)
- Friday: 1030-1130 (Hank)

Timeline

- ~~1C: due Weds Jan 23rd~~
- 1D: assigned ~~today~~ (LAST TUESDAY), due Thurs Jan 31st
- 1E: assigned Thurs Jan 31st, due Weds Feb 6th
  - → will be extra support with this. Tough project.
- 1F: assigned Feb 7th, due Feb 19th
  - → not as tough as 1E
- 2A: will be assigned during week of Feb 11th

Sun	Mon	Tues	Weds	Thurs	Fri	Sat
Jan 20	Jan21	Jan 22 Lec4	Jan 23 1C due	Lec 5 1D assigned	Jan 25	Jan 26
Jan 27	Jan 28	Jan 29 (YouTube)	Jan 30	Lec 6 1D due 1E assigned	Feb 1	Feb 2
Feb 3	Feb 4	Feb 5 Lec 7	Feb 6	Lec 8 1E due 1F assigned	Feb 8	Feb 9

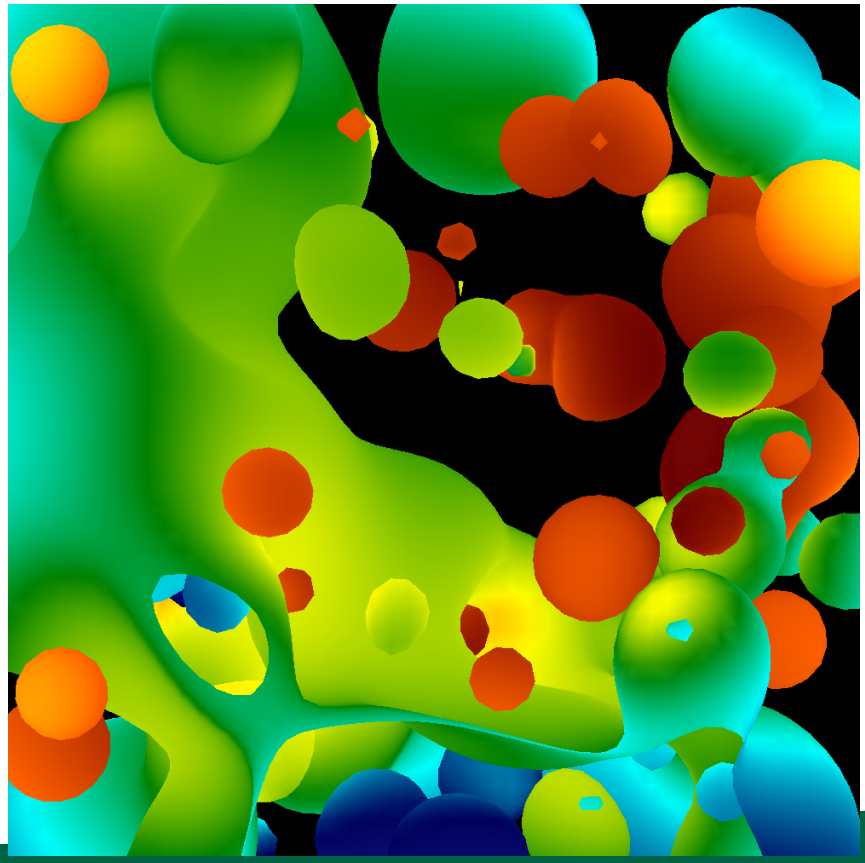
Likely: pre-SuperBowl OH

Great news!!

- No project assignment today…

Project #1D (5%), Due Thurs Jan 31st

- Goal: interpolation of color and zbuffer
- Extend your project1C code
- File proj1d_geometry.vtk available on web (1.4MB)
- File “reader1d.cxx” has code to read triangles from file.
- No Cmake, project1d.cxx



Color is now floating-point

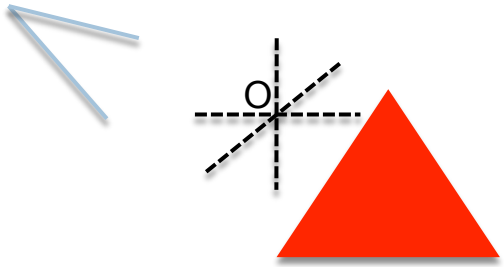
- We will be interpolating colors, so please use floating point (0 → 1)
- Keep colors in floating point until you assign them to a pixel
- Fractional colors? → use ceil_441…
  - ceil_441(value*255)

Changes to data structures

```
class Triangle
{
  public:
     double X[3], Y[3], Z[3];
     double colors[3][3];
};
```

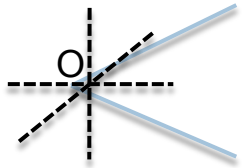
→ reader1d.cxx will not compile until you make these changes

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

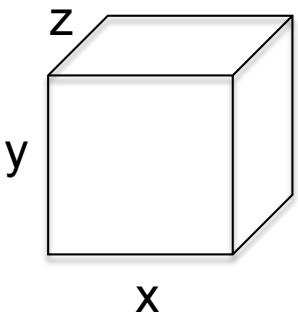
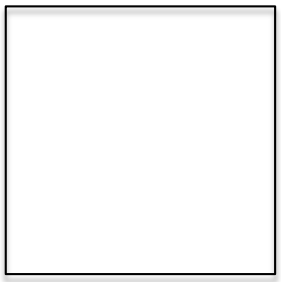


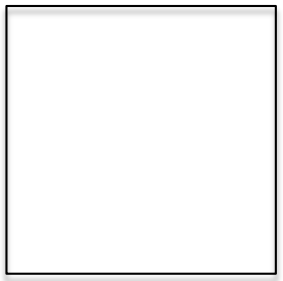
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

All viewable objects within
-1 <= x, y <= +1



Device space:

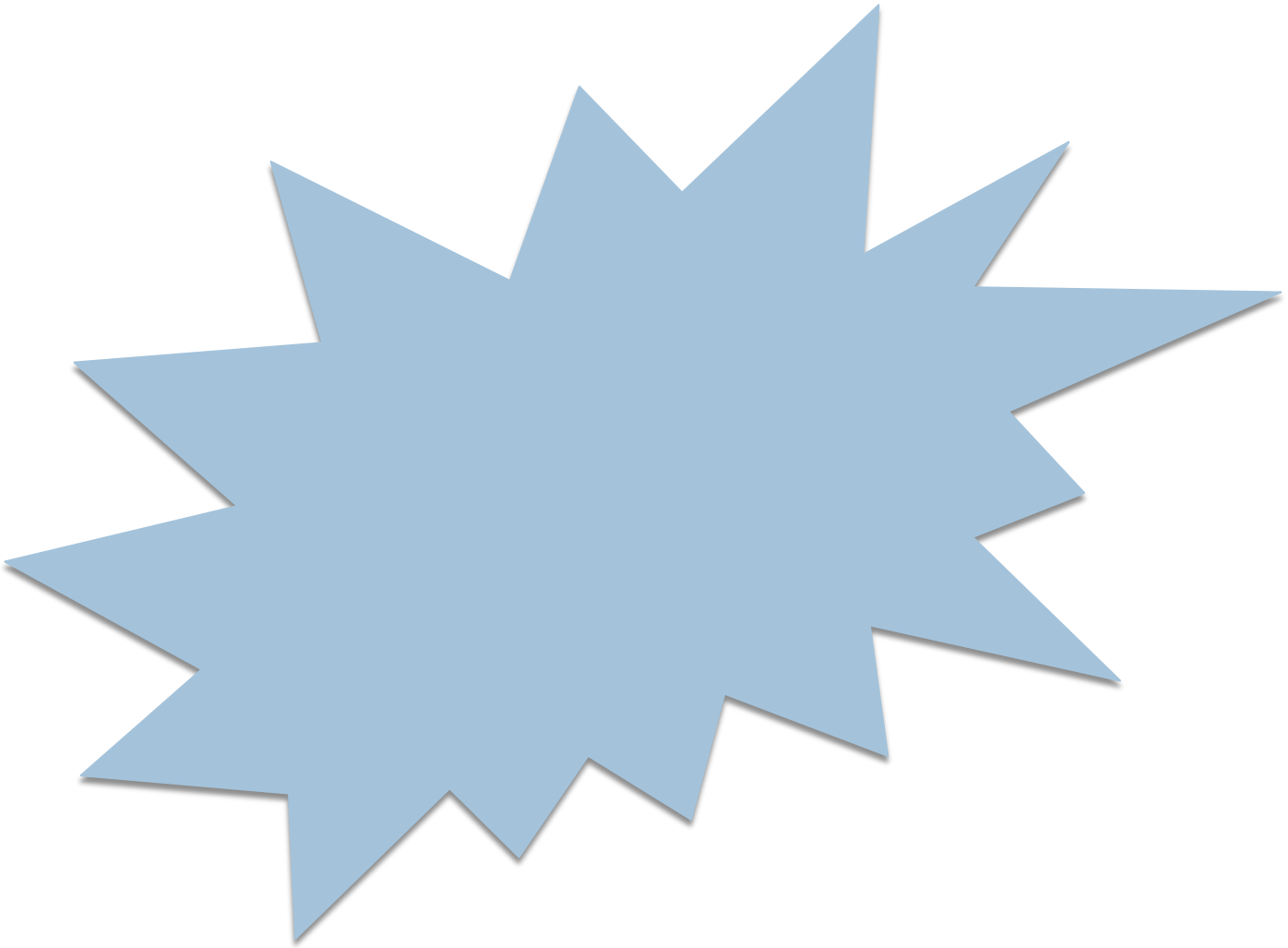
All viewable objects within
0<=x<=width, 0 <=y<=height

MATH!

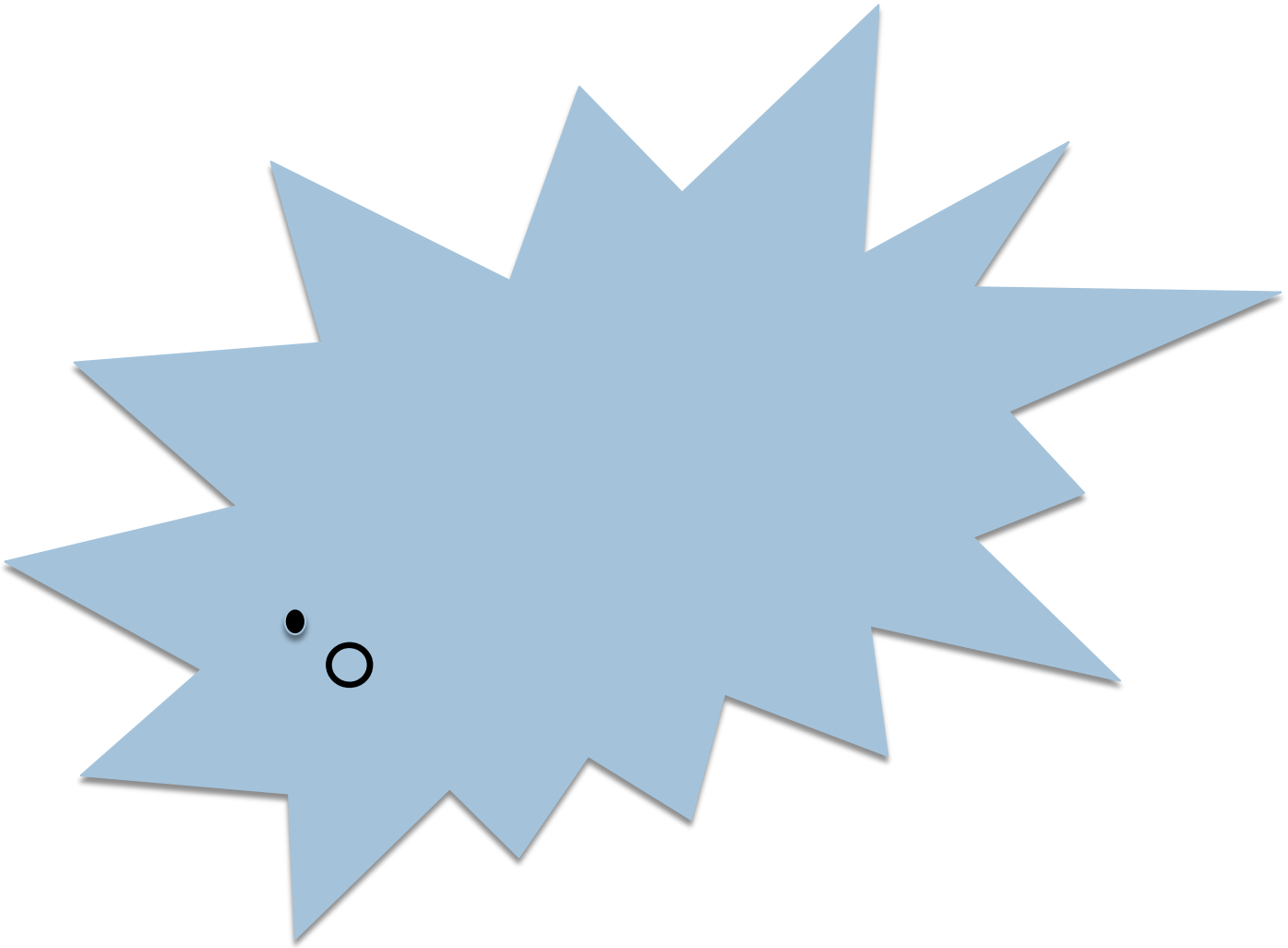
Space

- A “space” is a set of points
- Many types of spaces

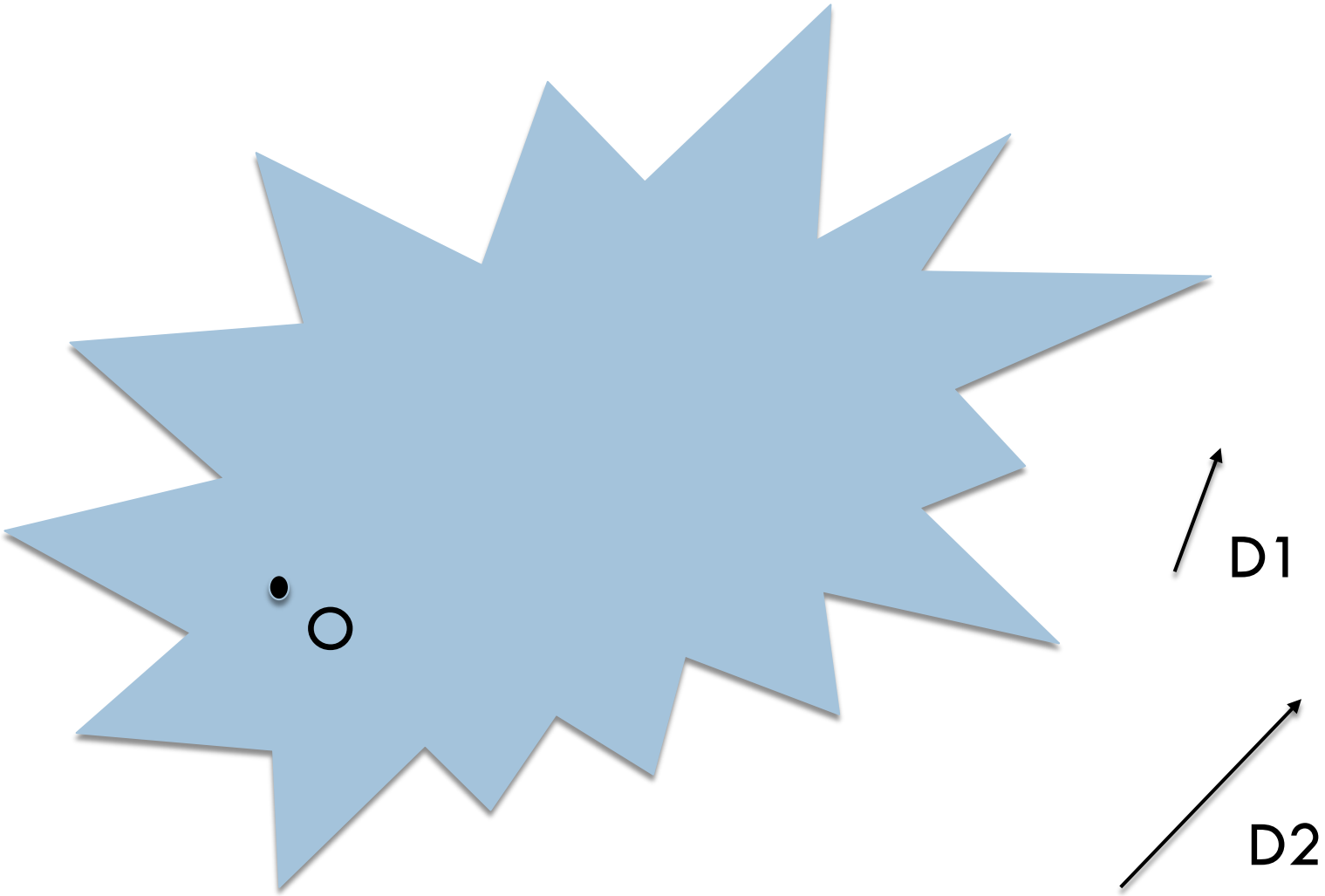
Here is a space ‘S’: the points in the blue shape



We can pick an arbitrary point in S and call it our "origin."



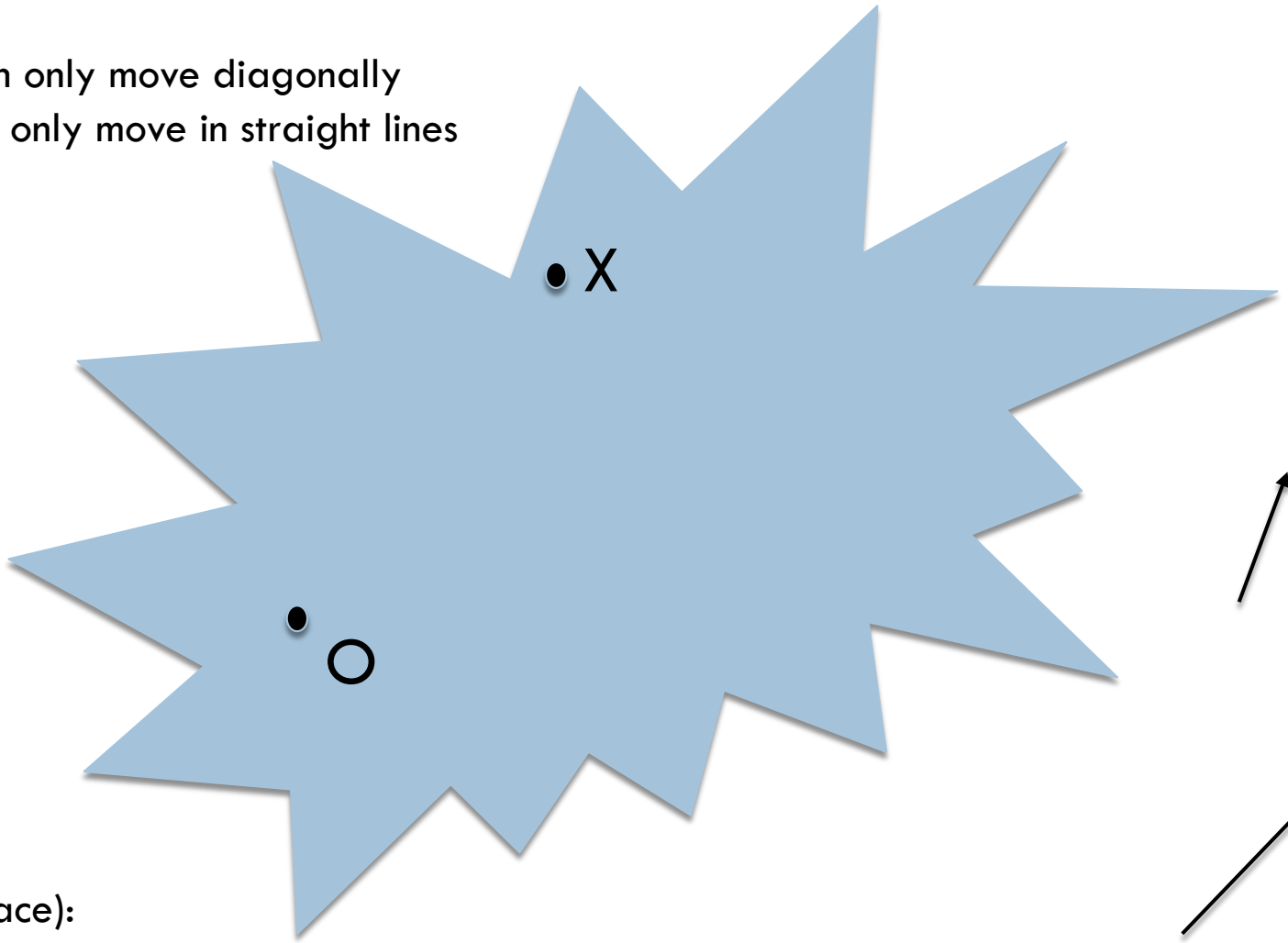
Consider two directions, D1 and D2.



Imagine you live at "O" and you want to get to "X." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



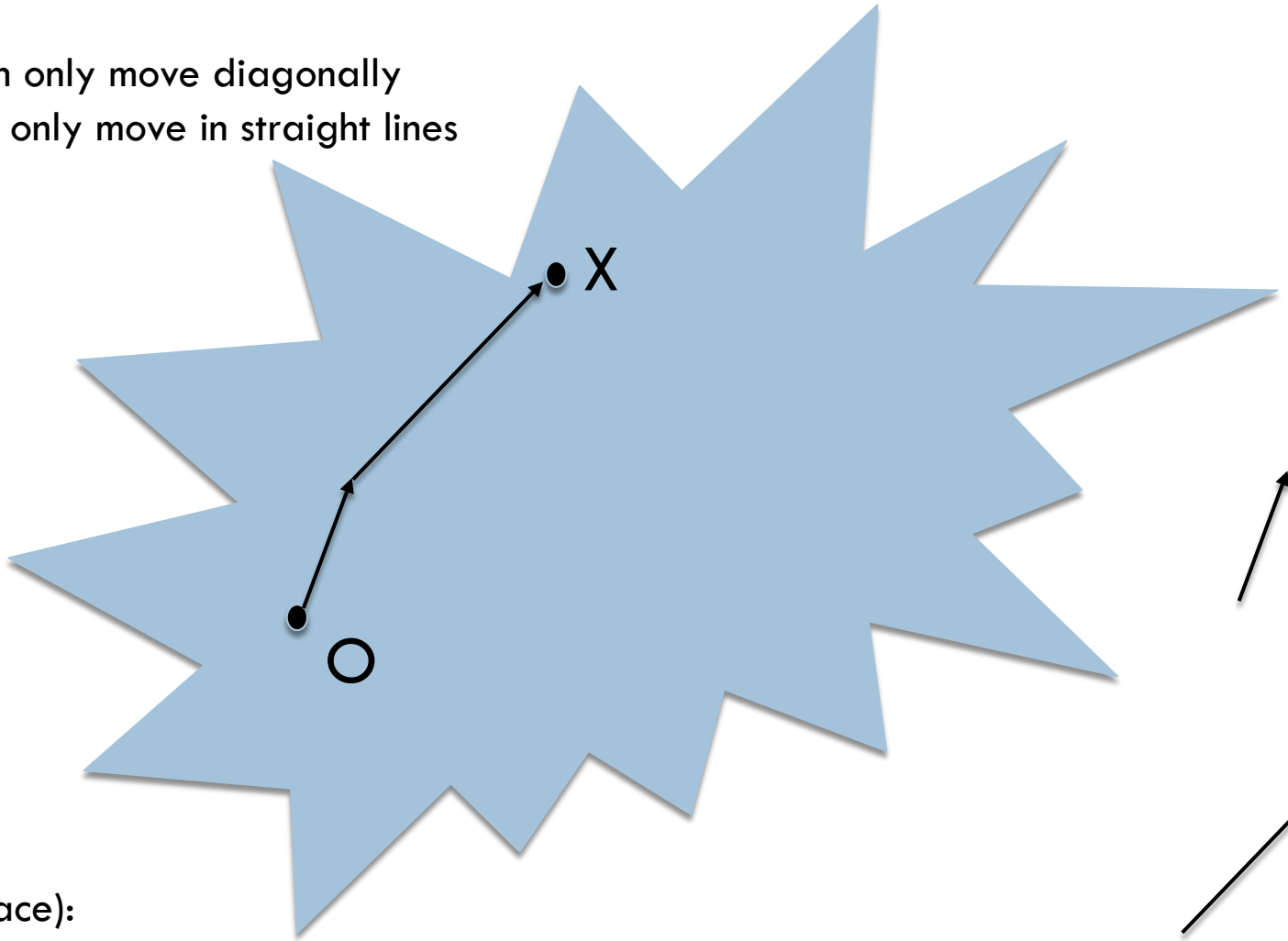
Rules (this space):

- You can only move in direction D1 or D2

Imagine you live at "O" and you want to get to "X." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



D1

D2

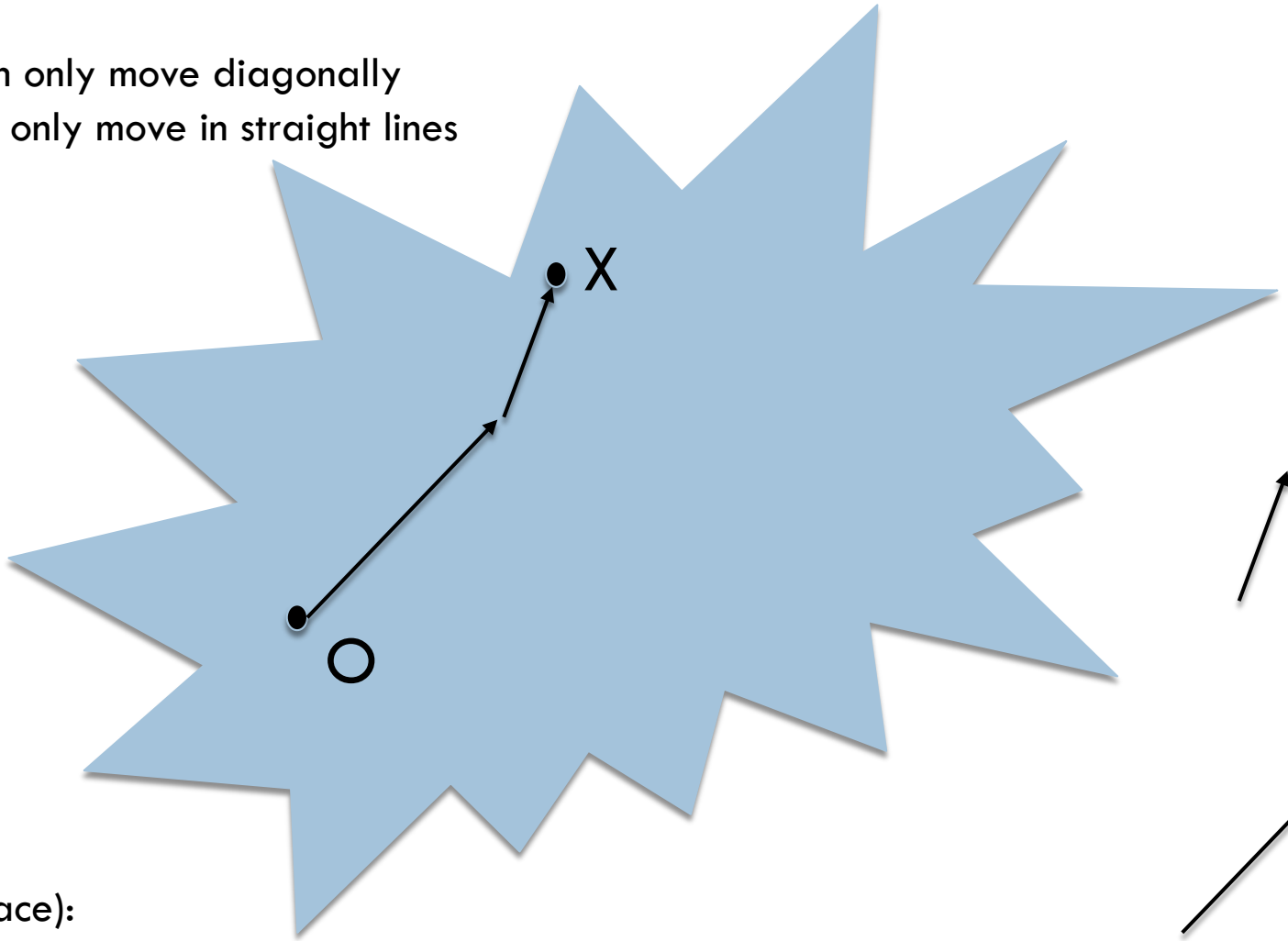
Rules (this space):

- You can only move in direction D1 or D2

Imagine you live at "O" and you want to get to "X." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



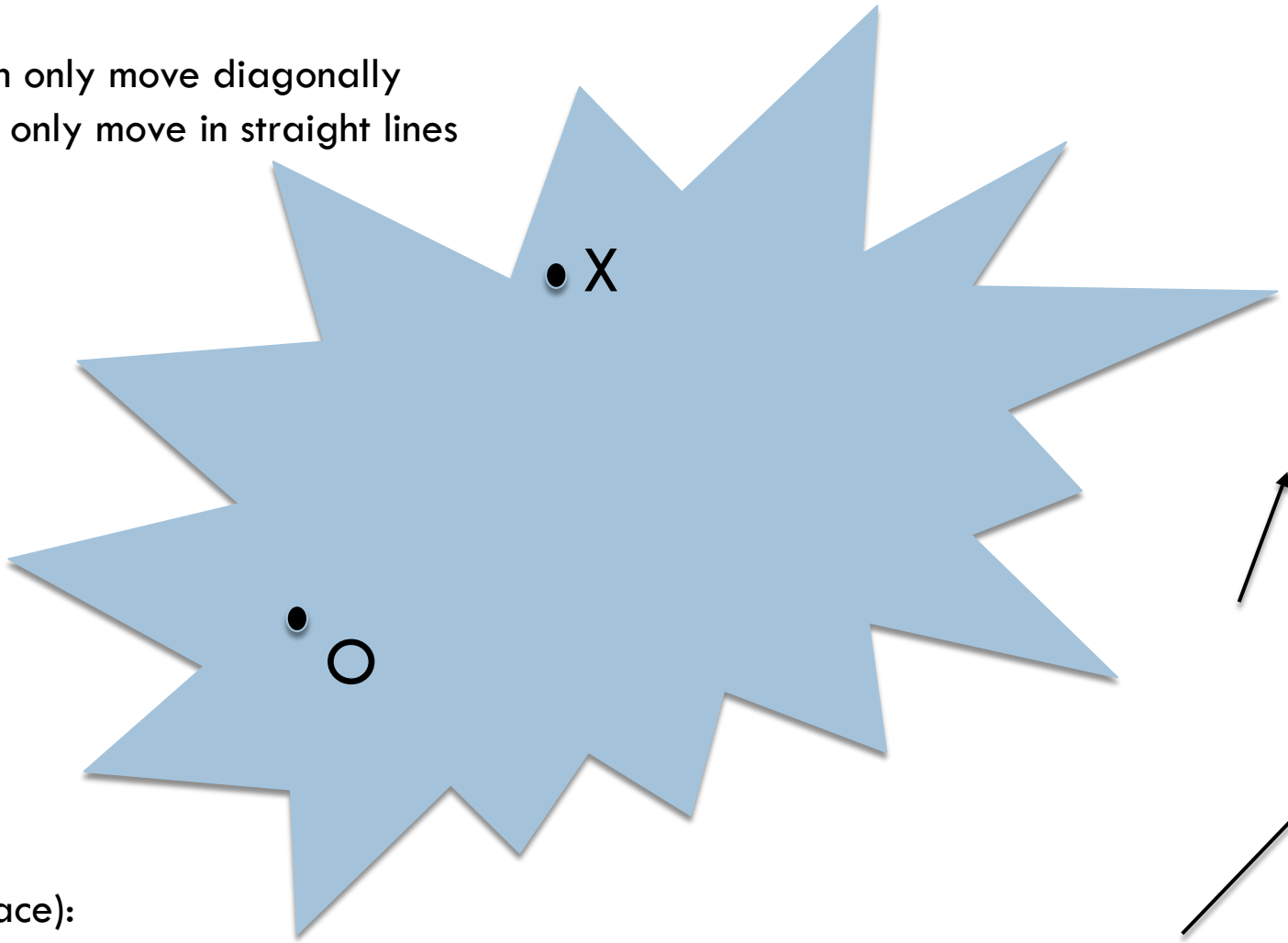
Rules (this space):

- You can only move in direction D1 or D2

Imagine you live at "O" and you want to get to "X." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



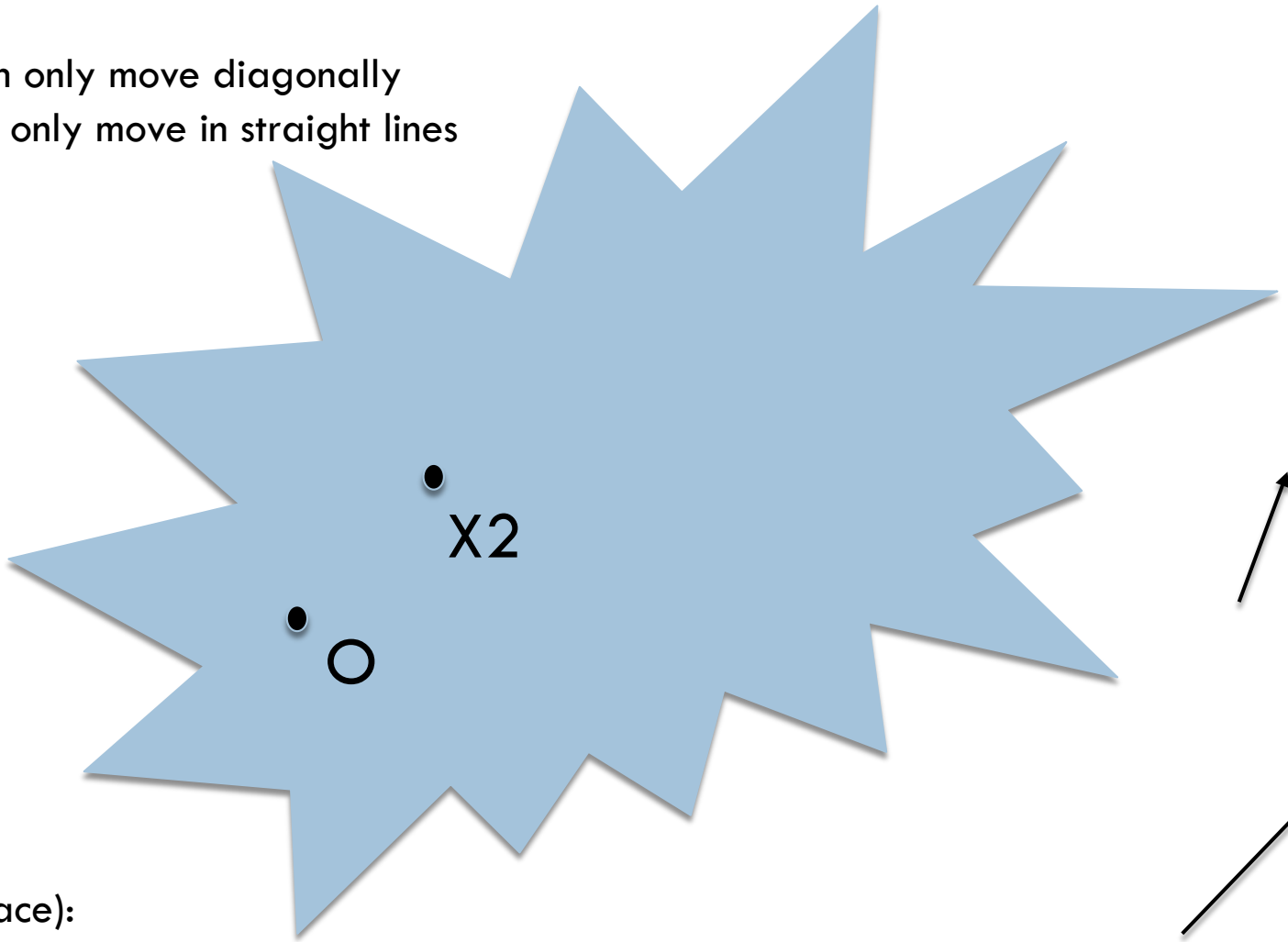
Rules (this space):

- You can only move in direction D1 or D2

Imagine you live at "O" and you want to get to "X2." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



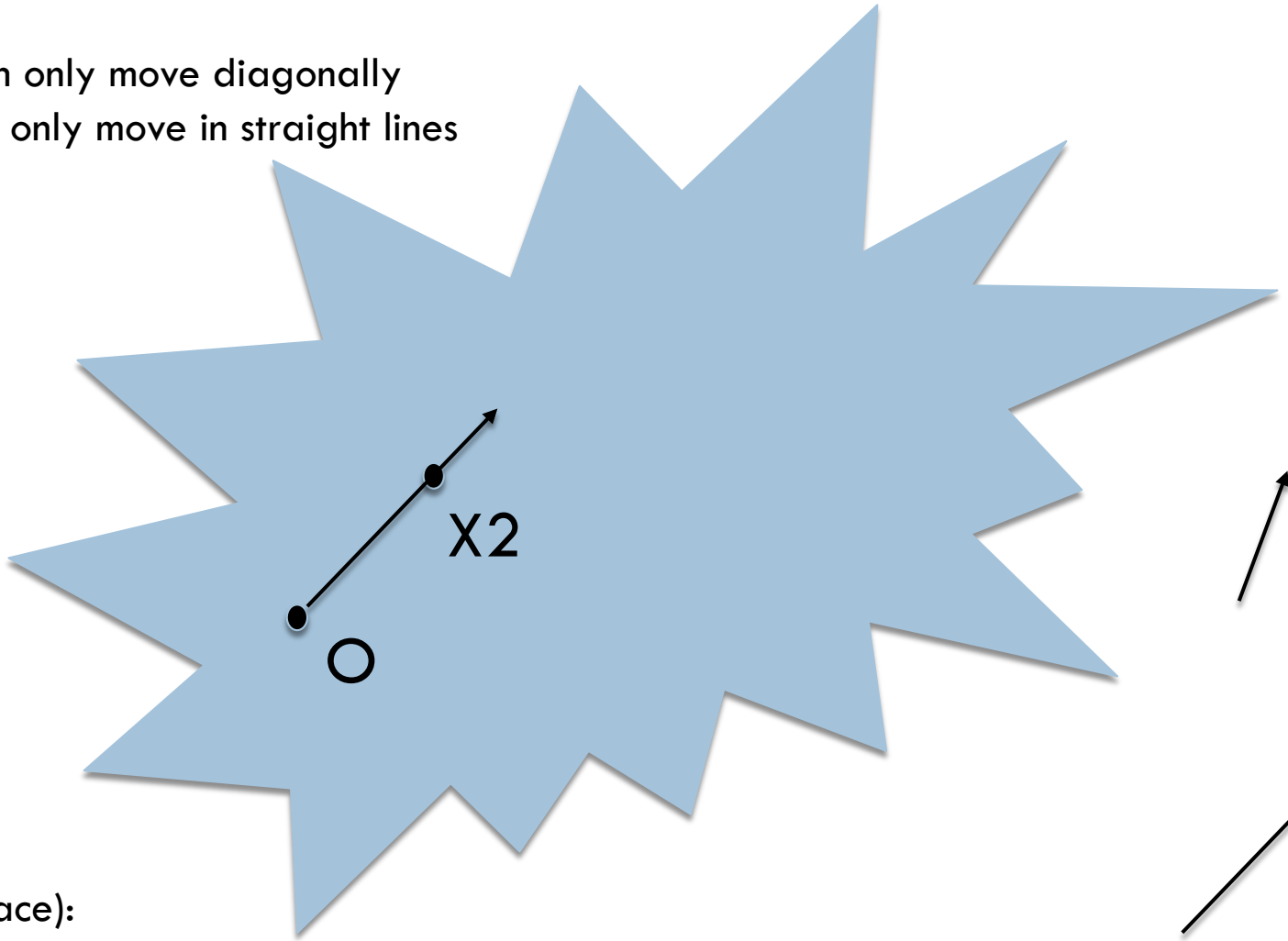
Rules (this space):

- You can only move in direction D1 or D2

Imagine you live at "O" and you want to get to "X2." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



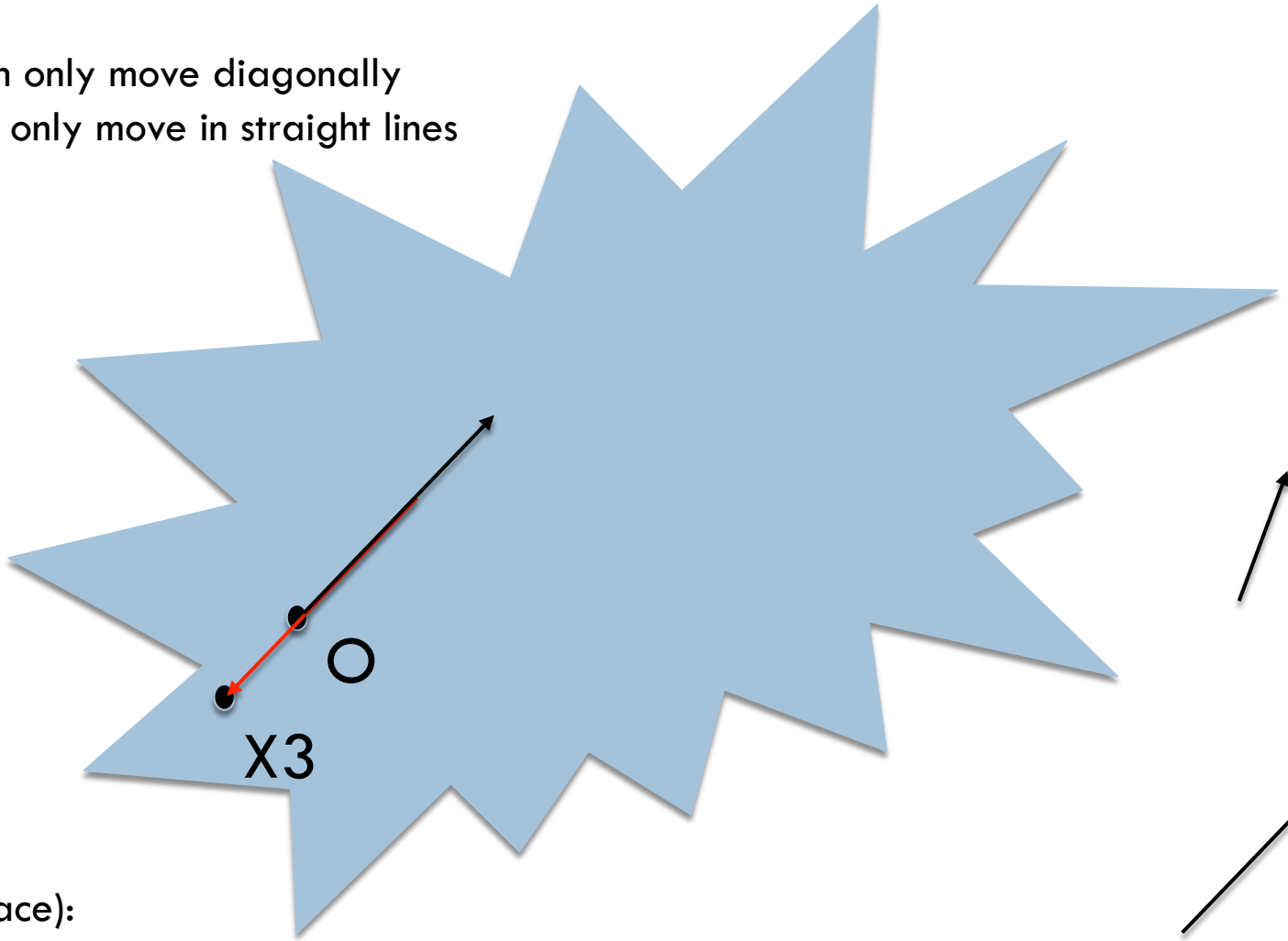
Rules (this space):

- You can only move in direction D1 or D2

Imagine you live at "O" and you want to get to "X3." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



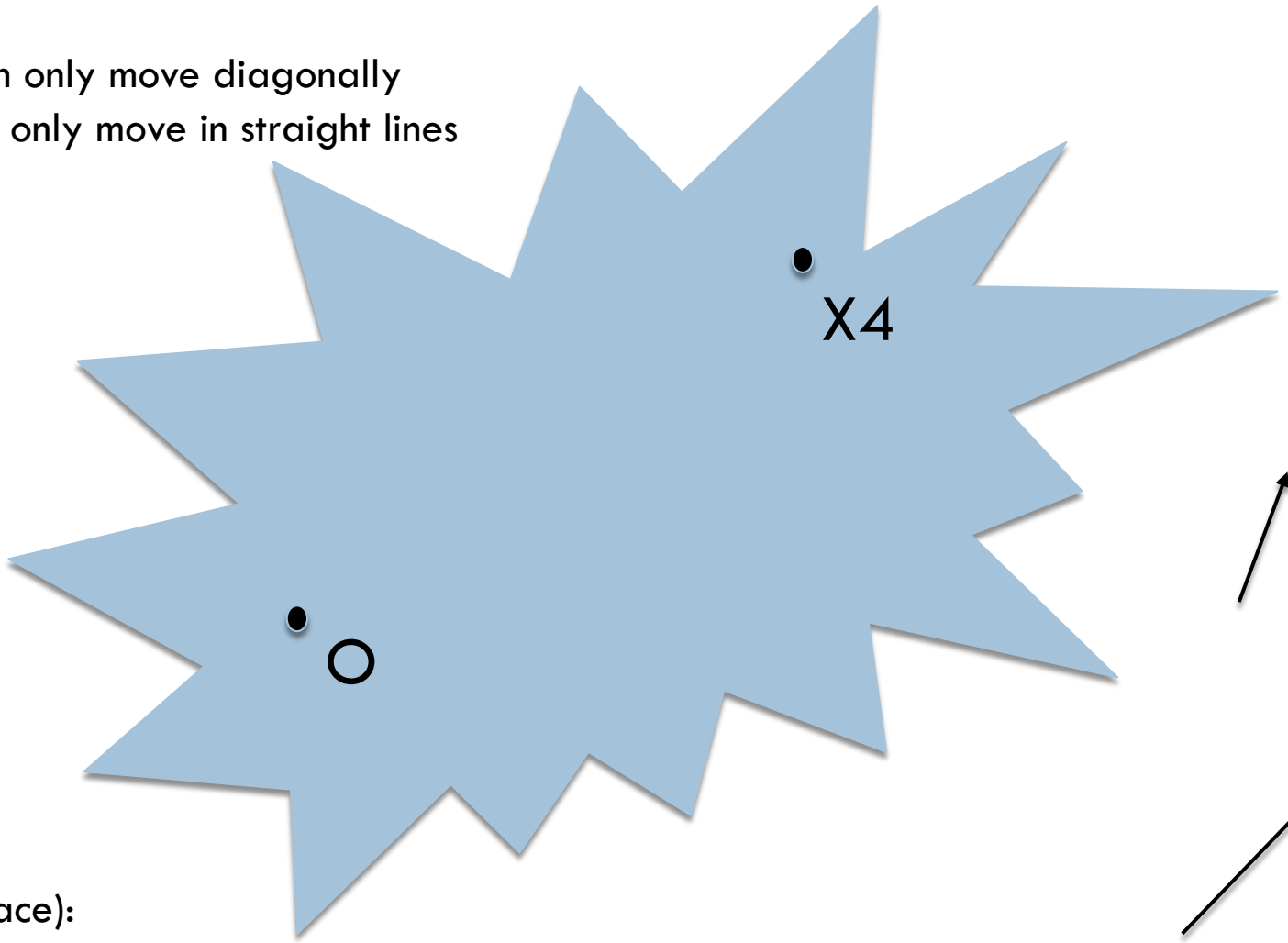
Rules (this space):

- You can only move in direction D1 or D2

Imagine you live at "O" and you want to get to "X4." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



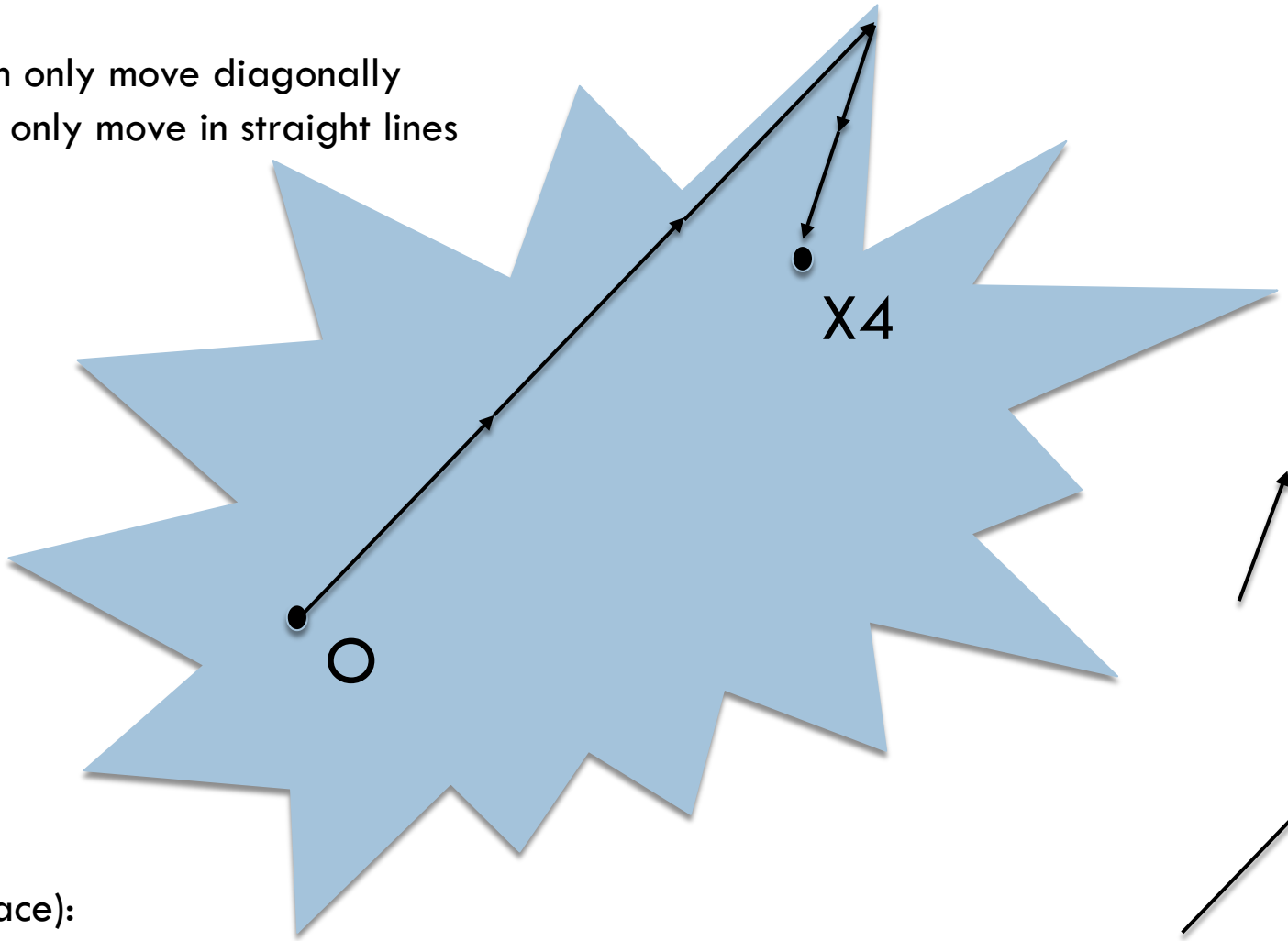
Rules (this space):

- You can only move in direction D1 or D2

Imagine you live at "O" and you want to get to "X4." Can you do it?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



D1

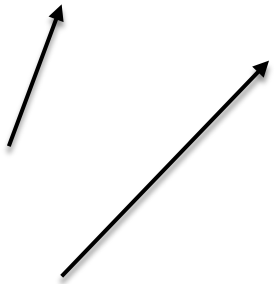
D2

Rules (this space):

- You can only move in direction D1 or D2

Conventions!

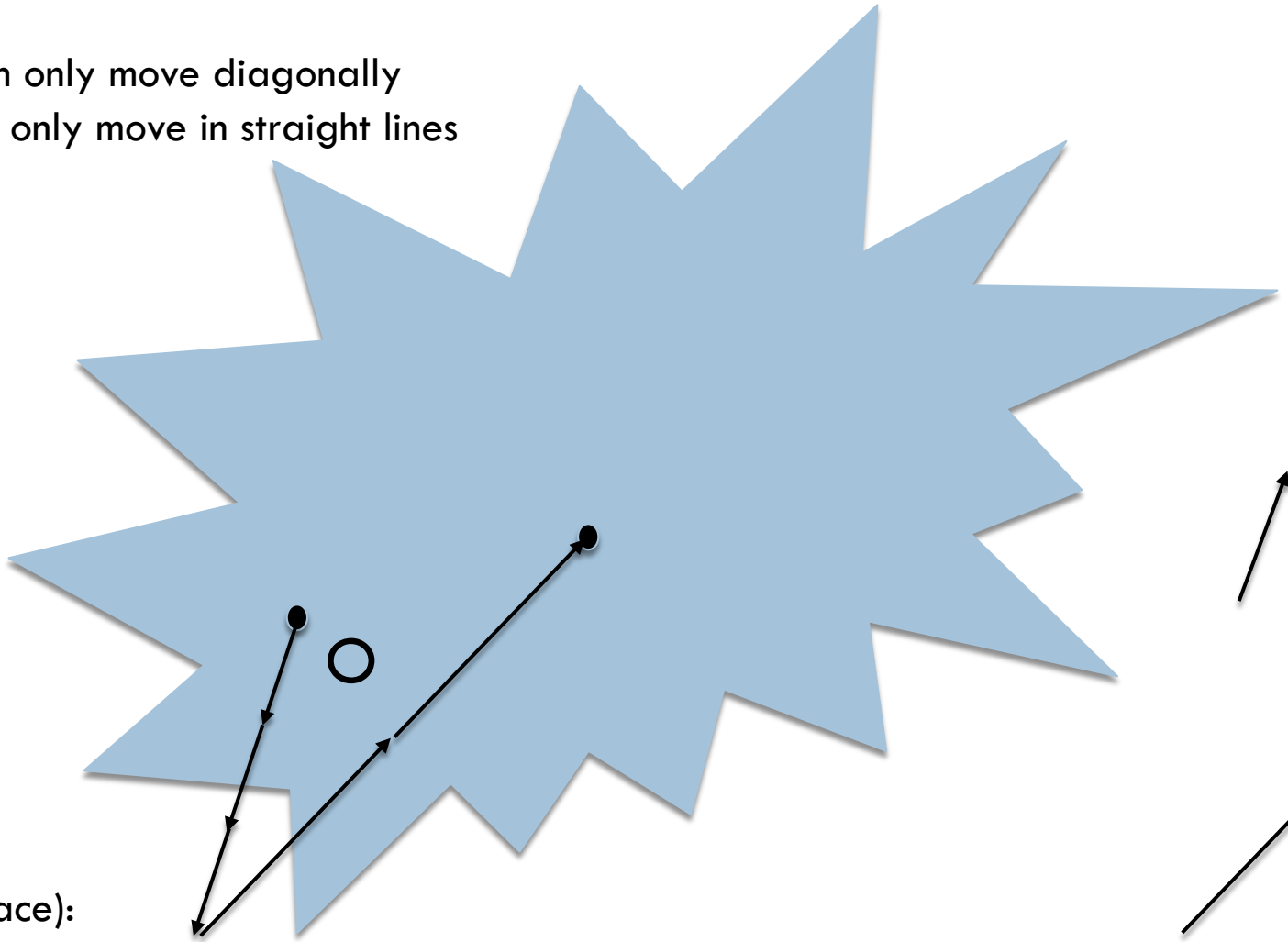
- Let (a, b) mean:
  - The number of steps ‘a’ in direction D1
  - The number of steps ‘b’ in direction D2



Where is (-3, 2)?

Rules (chess):

- Bishop can only move diagonally
- Rooks can only move in straight lines



Rules (this space):

- You can only move in direction D1 or D2

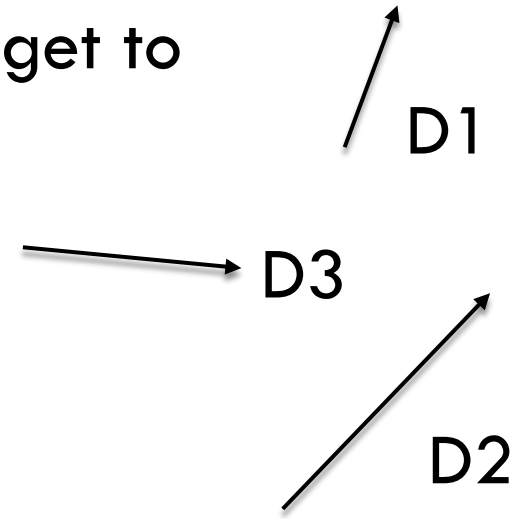
A basis

- Paraphrasing Wikipedia:
- Let B = { D1, D2 } (a set of two vectors, D1 & D2)
- Let S be our Shape 
- B is a basis for S if every element of S can be written as a **unique** linear combination of elements of B.
- The coefficients of this linear combination are referred to as components or coordinates on *B* of the vector.
- The elements of a basis are called basis vectors.

Why unique?

- Let (a, b, c) mean:
  - The number of steps ‘a’ in direction D1
  - The number of steps ‘b’ in direction D2
  - The number of steps ‘c’ in direction D3

- Then there is more than one way to get to some point X in S, i.e.,
  - (a1, b1, c1) = X and
  - (a2, b2, c2) = X



What does it mean to form a basis?

- For any vector v, there are unique coordinates (c1, …, cn) such that

  v = c1*v1 + c2*v2 + … + cn*vn

- Consider some point P.
  - The basis has an origin O
  - There is a vector v such that O+v = P
  - We know we can construct v using a combination of vi's
  - Therefore we can represent P in our frame using the coordinates (c1, c2, …, cn)

A basis

- Paraphrasing Wikipedia:
- Let B = { D1, D2 } (a set of two vectors, D1 & D2)
- Let S be our Shape 
- B is a basis for S if every element of S can be written as a **unique** linear combination of elements of B.
- The coefficients of this linear combination are referred to as components or coordinates on *B* of the vector.
- The elements of a basis are called basis vectors.

Most common basis

- D1 = X-axis (i.e., (1,0,0)-(0,0,0))
- D2 = Y-axis (i.e., (0,1,0)-(0,0,0))
- D3 = Z-axis (i.e., (0,0,1)-(0,0,0))

- Then the coordinate (2, -3, 5) means
  - 2 units along X-axis
  - -3 units along Y-axis
  - 5 units along Z-axis

But we could have other bases

- Instead of “basis 1” (B1)
  - D1 = X-axis (i.e., (1,0,0)-(0,0,0))
  - D2 = Y-axis (i.e., (0,1,0)-(0,0,0))
  - D3 = Z-axis (i.e., (0,0,1)-(0,0,0))
- Use “basis 2” (B2)
  - D1 = Y-axis (i.e., (0,1,0)-(0,0,0))
  - D2 = X-axis (i.e., (1,0,0)-(0,0,0))
  - D3 = Z-axis (i.e., (0,0,1)-(0,0,0))
- Then (a,b,c) in B1 is the same as (b,a,c) in B2

Last vocab term for a few slides: frame

- Frame:
  - A way to place a coordinate system into a specific location in a space
  - Basis + reference coordinate ("the origin")
- Cartesian example: (3,4,6)
  - It is assumed that we are speaking in reference to the origin location (0,0,0).

Example of Frames

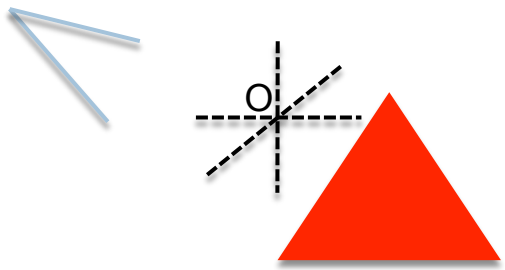
- Frame F = (v1, v2, O)
  - v1 = (0, -1)
  - v2 = (1, 0)
  - O = (3, 4)
- What are F's coordinates for the point (6, 6)?

Example of Frames

- Frame F = (v1, v2, O)
  - v1 = (0, -1)
  - v2 = (1, 0)
  - O = (3, 4)
- What are F's coordinates for the point (6, 6)?

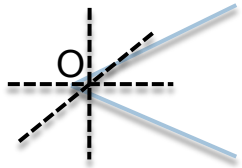
- Answer: (-2, 3)

Each box is a frame, and each arrow converts to the next frame



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

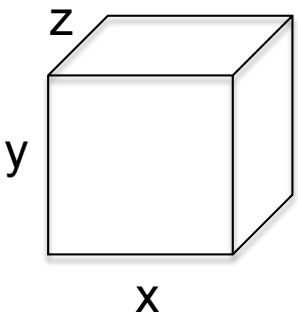
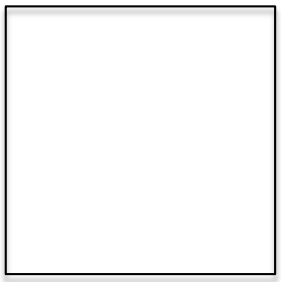


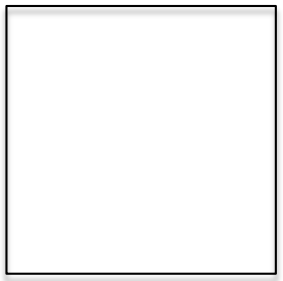
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

All viewable objects within
-1 <= x, y <= +1



Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height

Context

- Models stored in “world space” frame
  - Pick an origin, store all points relative to that origin
- We have been rasterizing in “device space” frame
- Our goal: transform from world space to device space
- We will do this using matrix multiplications
  - Multiply point by matrix to convert coordinates from one frame into coordinates in another frame

But wait! There's more…

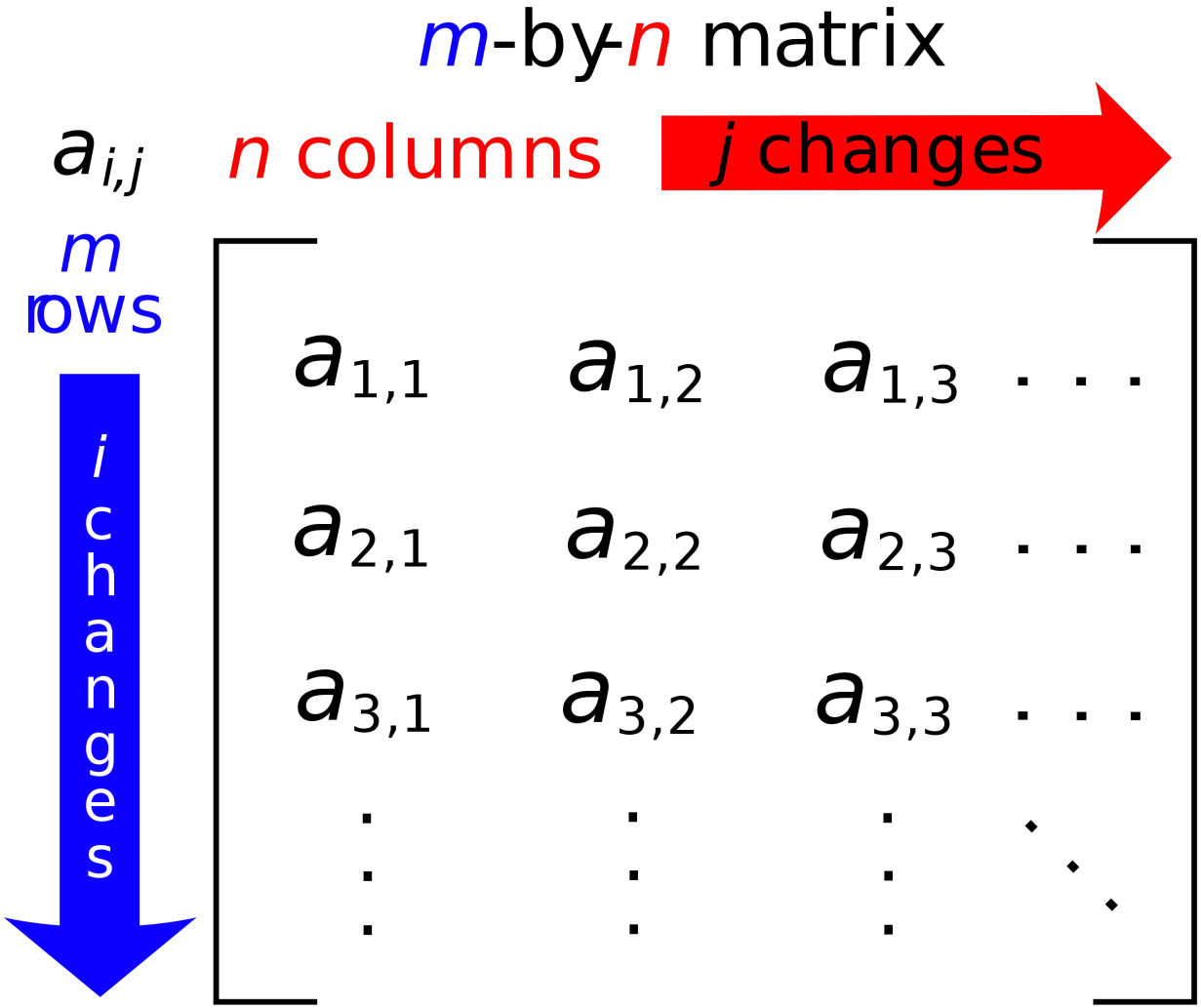
- And matrices also useful for more than frame-to-frame conversions.

- So let's get comfy with matrices.

Matrix

- Defined: a rectangular array of numbers (usually) arranged in rows and columns
- Example
  - 2D matrix
  - "two by three" (two rows, three columns)
    - [3 4 8]
    - [-1 9.2 12]

Matrix: wikipedia picture



Matrix

- What do you do with matrices?
- Lots of things
  - Transpose, invert, add, subtract
- But most of all: multiply!

Multiplying two 2x2 matrices

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a*e+b*g & a*f+b*h \\ c*e+d*g & c*f+d*h \end{pmatrix}$$

Multiplying two 2x2 matrices

$$\begin{pmatrix} a & b \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a*e+b*g & a*f+b*h \end{pmatrix}$$

One usage for matrices:

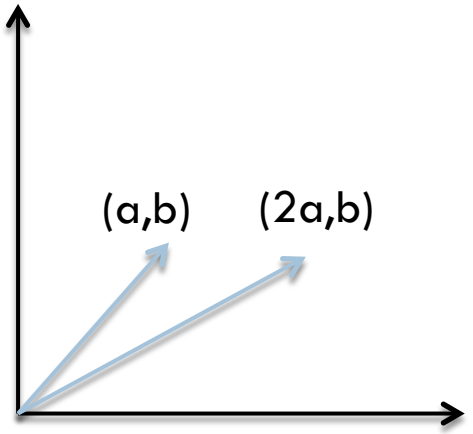
Let (a, b) be the coordinates of a point

Then the 2x2 matrix can transform (a,b) to a new location – (a*e+b*g, a*f+b*h)

Identity Matrix

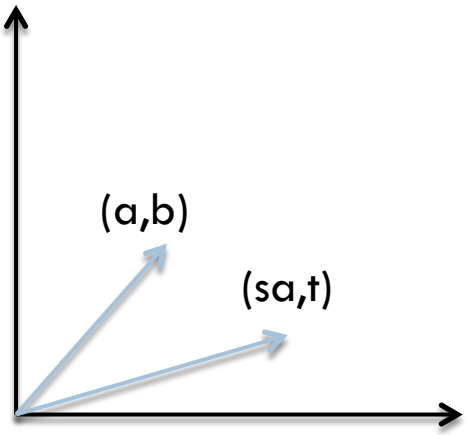
$$\begin{pmatrix} a & b \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} a & b \end{pmatrix}$$

$$\begin{pmatrix} a & b \end{pmatrix} \times \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2a & b \end{pmatrix}$$



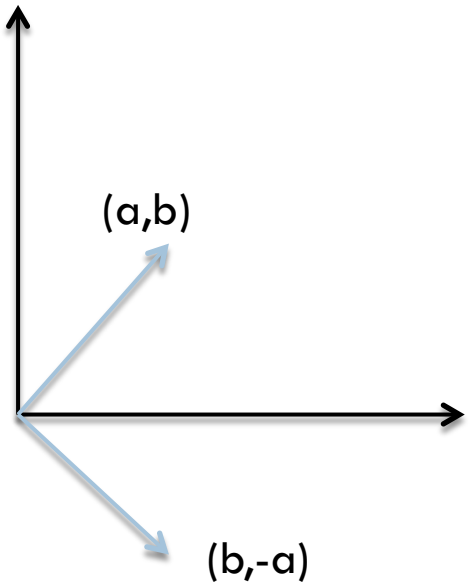
Scale in X, not in Y

$$\begin{pmatrix} a & b \end{pmatrix} \times \begin{pmatrix} s & 0 \\ 0 & t \end{pmatrix} = \begin{pmatrix} sa & tb \end{pmatrix}$$



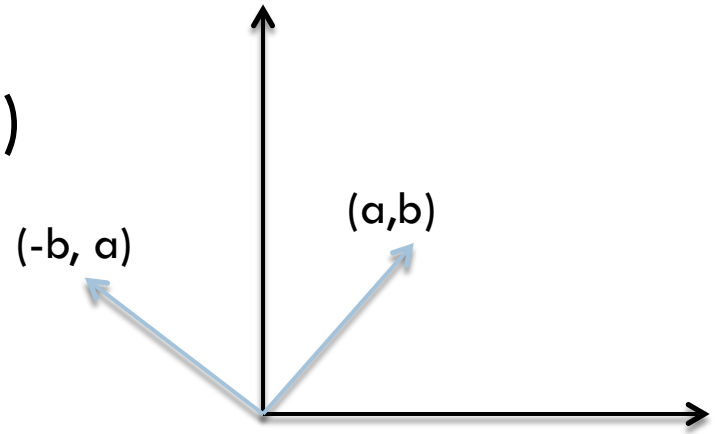
Scale in both dimensions

$$\begin{pmatrix} a & b \end{pmatrix} \times \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} b & -a \end{pmatrix}$$



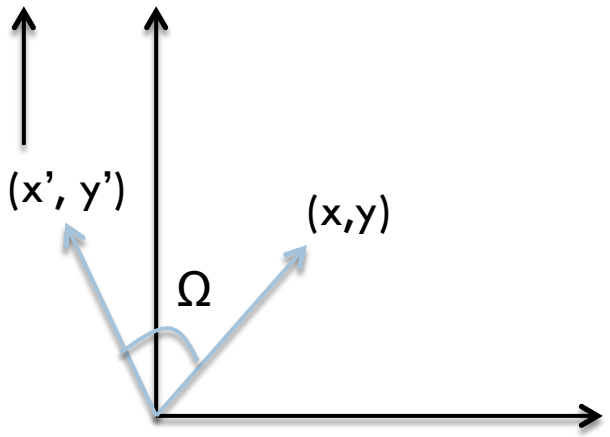
Rotate 90 degrees counter-clockwise

$$\begin{pmatrix} a & b \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} = \begin{pmatrix} -b & a \end{pmatrix}$$



Rotate 90 degrees counter-clockwise

$$(a \quad b) \quad \begin{pmatrix} \cos(\Omega) & -\sin(\Omega) \\ \sin(\Omega) & \cos(\Omega) \end{pmatrix} = (\cos(\Omega)*a + \sin(\Omega)*b,\ -\sin(\Omega)*a + \cos(\Omega)*b)$$



Rotate "Ω" degrees counter-clockwise

Combining transformations

- How do we rotate by 90 degrees clockwise and then scale X by 2?
  - Answer: multiply by matrix that multiplies by 90 degrees clockwise, then multiple by matrix that scales X by 2.
  - But can we do this efficiently?

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

Combining transformations

- How do we scale X by 2 and then rotate by 90 degrees clockwise?
  - Answer: multiply by matrix that scales X by 2, then multiply by matrix that rotates 90 degrees clockwise.

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

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

Rotate then scale
Order matters!!

Translations

- Translation is harder:

$$\begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a+c \\ b+d \end{pmatrix}$$

But this doesn't fit our nice matrix multiply model…
What to do??

Homogeneous Coordinates

$$(x \quad y \quad 1) \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = (x \quad y \quad 1)$$

Add an extra dimension.
A math trick … don't overthink it.

Homogeneous Coordinates

$$\begin{pmatrix} x & y & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{pmatrix} = \begin{pmatrix} x+dx & y+dy & 1 \end{pmatrix}$$

Translation

We can now fit translation into
our matrix multiplication system.

Graphics

- Two really important operations:
  - Transform from one frame to another
  - Transform geometry (rotate, translate, etc)

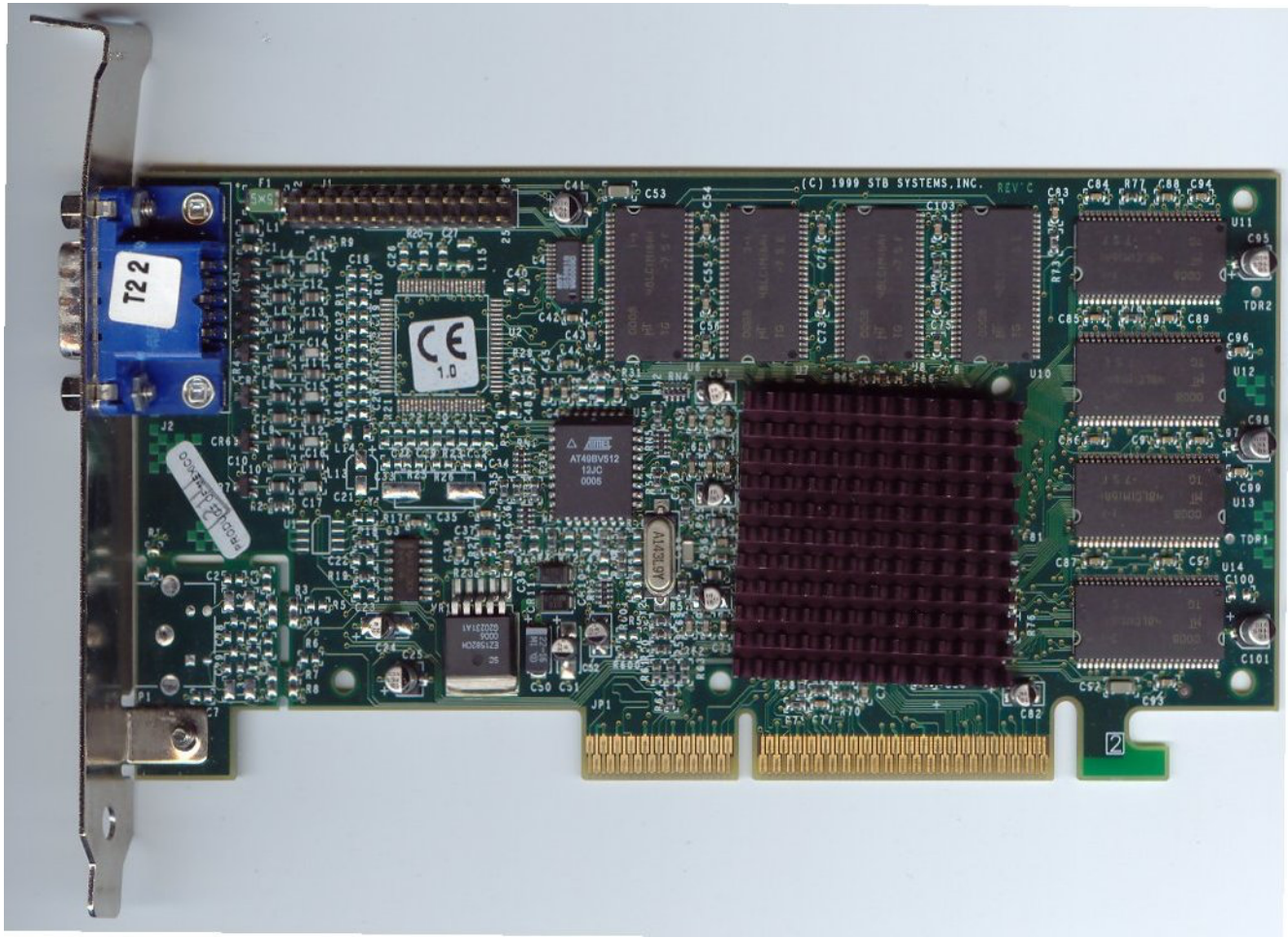
- Both can be done with matrix operations
- In both cases, need homogeneous coordinates

- Much of graphics is accomplished via 4x4 matrices
  - And: you can compose the matrices and do bunches of things at once (EFFICIENCY)

Silicon Graphics, Inc.

Former type	Public
Traded as	NYSE: SGI OTC Pink: SGID.pk NASDAQ: SGIC
Industry	Computer hardware and software
Fate	Chapter 11 bankruptcy; assets acquired by Rackable Systems, which renamed itself Silicon Graphics International Corp.
Founded	November 9, 1981; 37 years ago Mountain View, California, U.S.[1]
Defunct	May 11, 2009
Headquarters	Sunnyvale, California, U.S.
Key people	Jim Clark, Kurt Akeley, Ed McCracken, Thomas Jermoluk
Products	High-performance computing, visualization and storage
Website	www.sgi.com/

3dfx Voodoo (source: wikipedia)



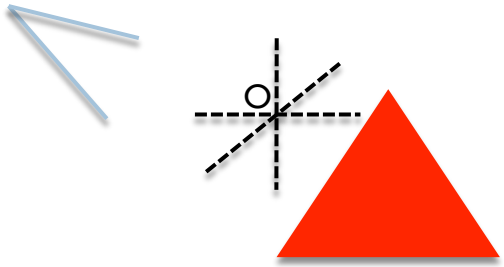
Early GPUs

- Special hardware to do 4x4 matrix operations
- A lot of them (in parallel)

GPUs now

- Many, many, many cores
- Each code less powerful than typical CPU core

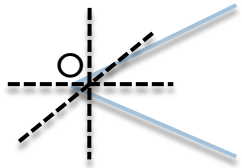
Our goal



World space:

Triangles in native Cartesian coordinates

Camera located anywhere



Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

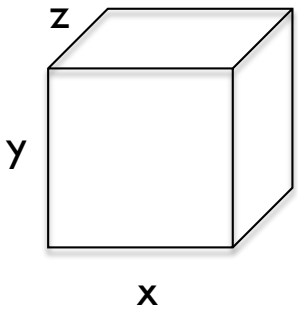
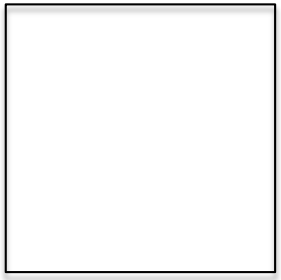


Image space:

All viewable objects within

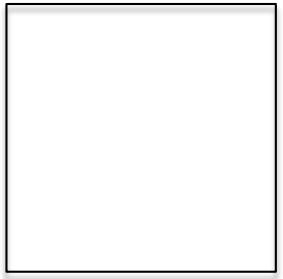
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



Device space:

All viewable objects within

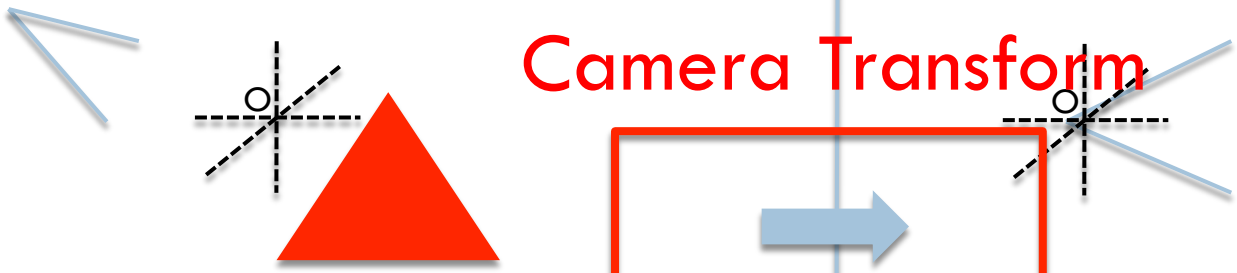
0<=x<=width, 0

<=y<=height

World Space

- World Space is the space defined by the user's coordinate system.
- This space contains the portion of the scene that is transformed into image space by the camera transform.
- Many of the spaces have "bounds", meaning limits on where the space is valid
- With world space 2 options:
  - No bounds
  - User specifies the bound

Our goal



Camera Transform

World space:

Triangles in native Cartesian coordinates

Camera located anywhere

Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

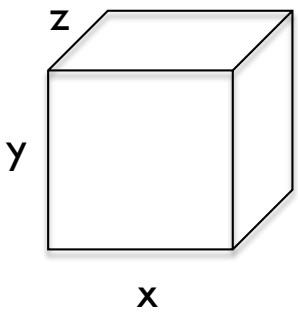
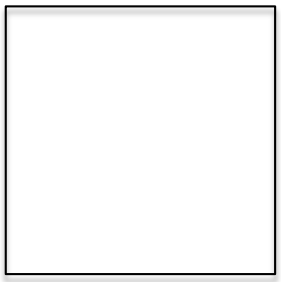


Image space:

All viewable objects within

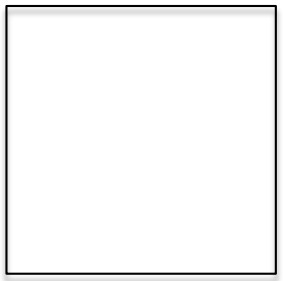
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



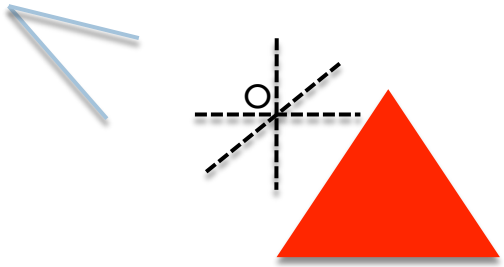
Device space:

All viewable objects within

0<=x<=width, 0

<=y<=height

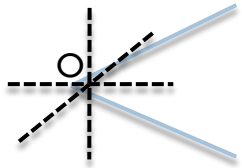
Our goal



World space:

Triangles in native Cartesian coordinates

Camera located anywhere



Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

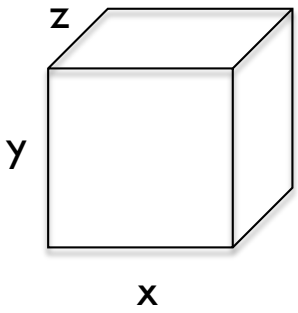
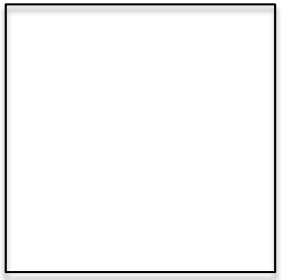


Image space:

All viewable objects within

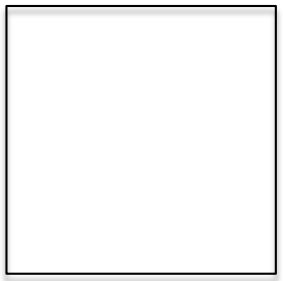
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



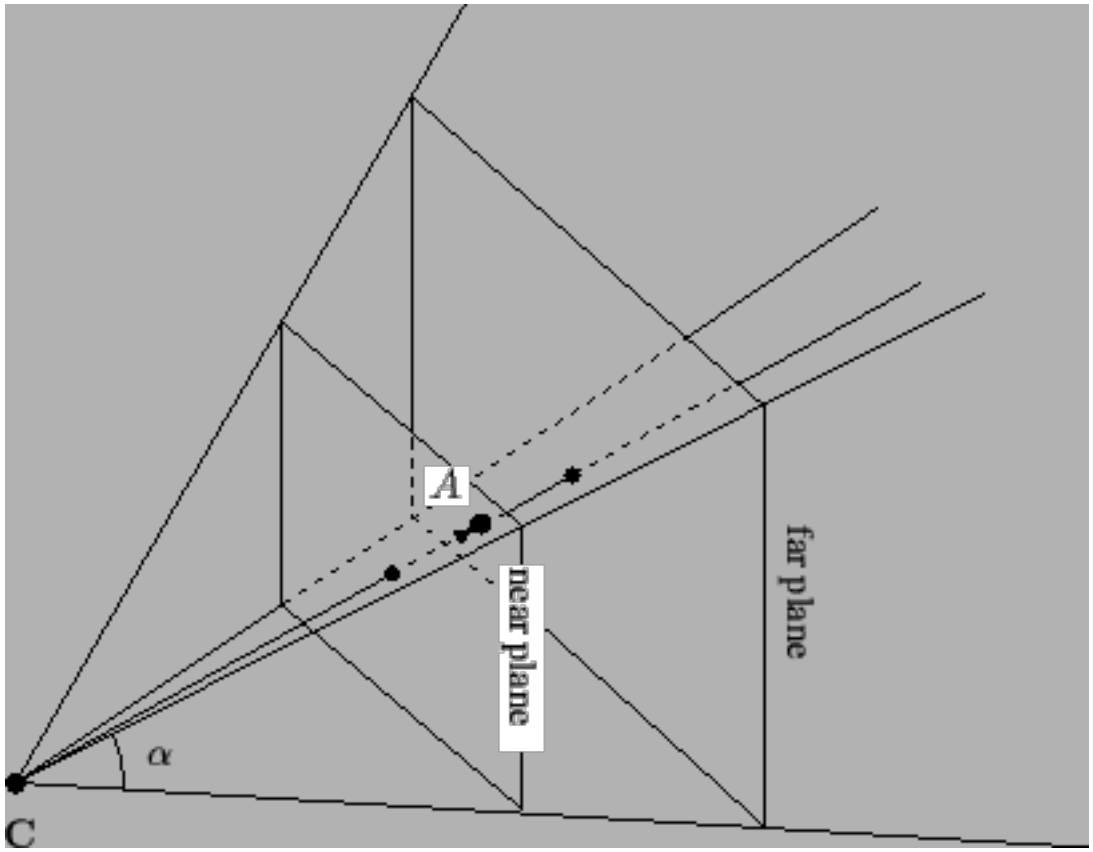
Device space:

All viewable objects within

0<=x<=width, 0

<=y<=height

How do we specify a camera?

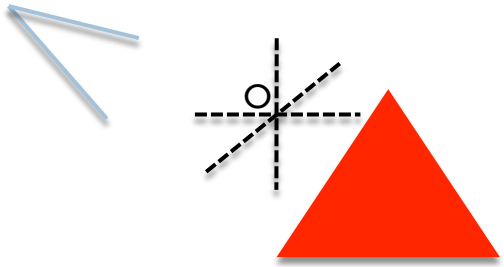


The "viewing pyramid" or "view frustum".

Frustum: In geometry, a frustum (plural: frusta or frustums) is the portion of a solid (normally a cone or pyramid) that lies between two parallel planes cutting it.

```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];
};
```

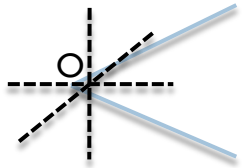
Our goal



World space:

Triangles in native Cartesian coordinates

Camera located anywhere



Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

View Transform

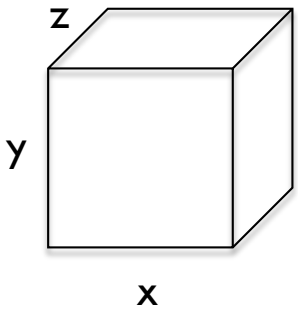
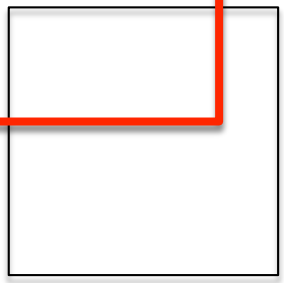


Image space:

All viewable objects within

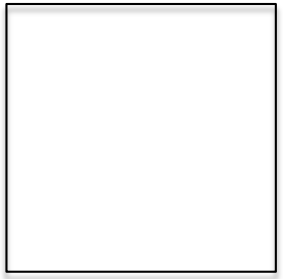
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



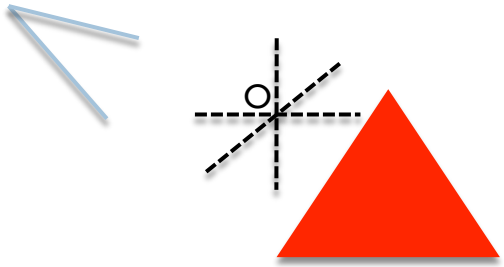
Device space:

All viewable objects within

0<=x<=width, 0

<=y<=height

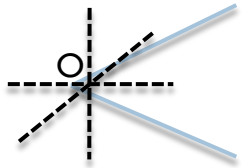
Our goal



World space:

Triangles in native Cartesian coordinates

Camera located anywhere



Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

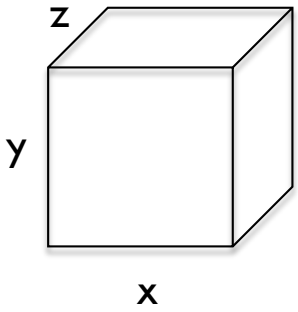
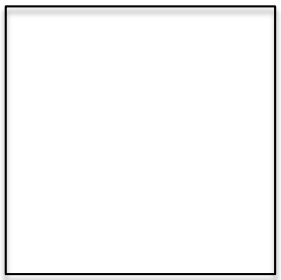


Image space:

All viewable objects within

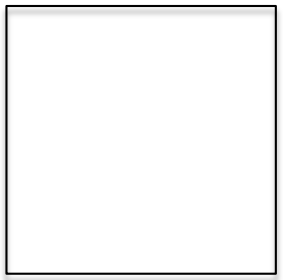
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



Device space:

All viewable objects within

0<=x<=width, 0

<=y<=height

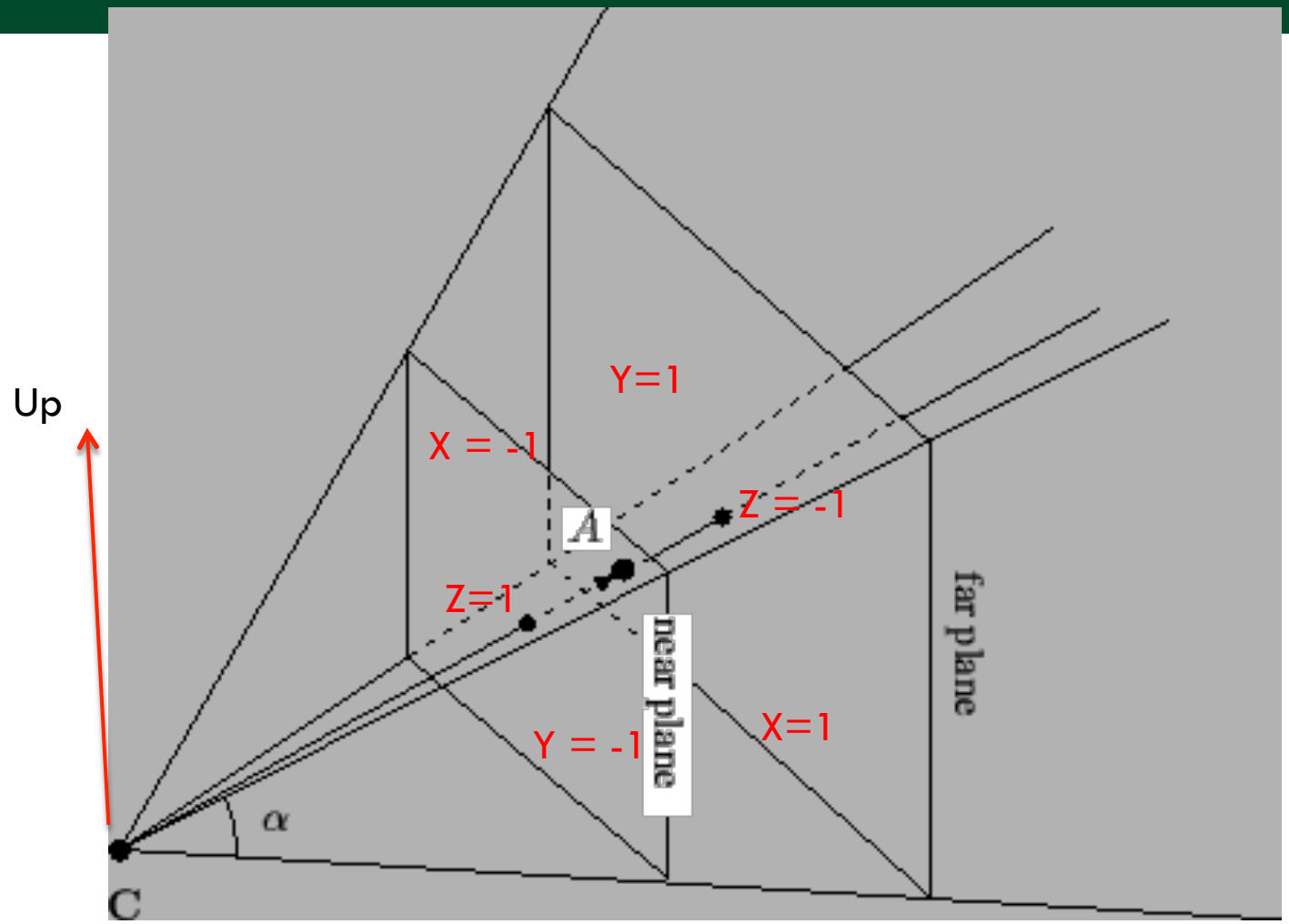
Image Space

- Image Space is the three-dimensional coordinate system that contains screen space.
- It is the space where the camera transformation directs its output.
- The bounds of *Image Space* are 3-dimensional cube.

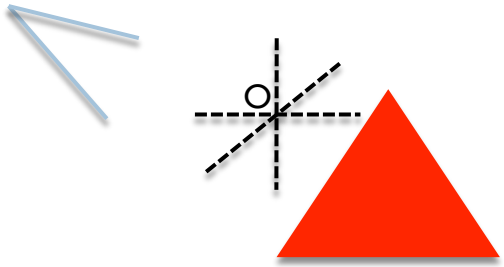
$$\{(x,y,z) : -1 \le x \le 1, -1 \le y \le 1, -1 \le z \le 1\}$$

(or $-1 \le z \le 0$)

Image Space Diagram



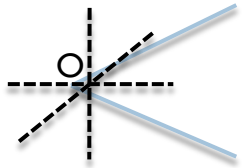
Our goal



World space:

Triangles in native Cartesian coordinates

Camera located anywhere



Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

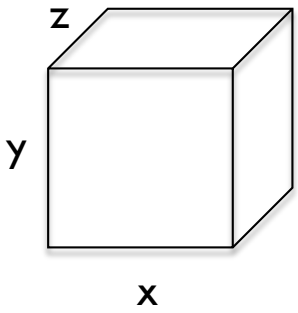
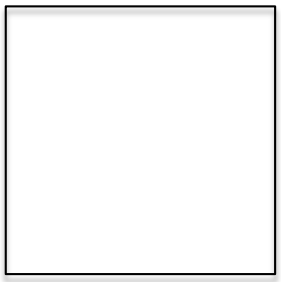


Image space:

All viewable objects within

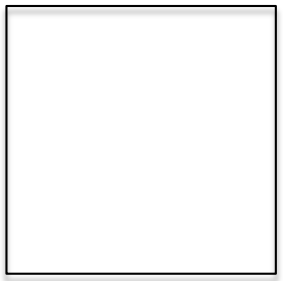
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



Device space:

All viewable objects within

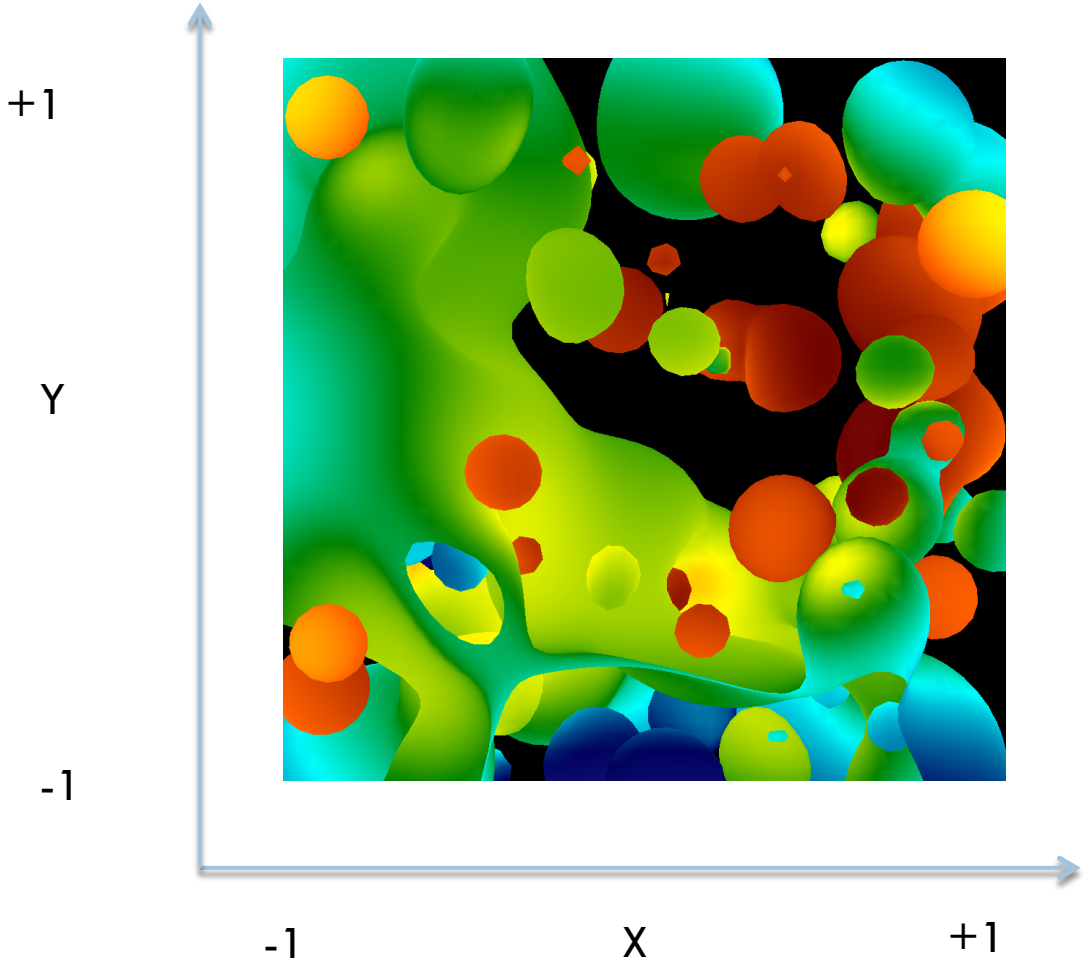
0<=x<=width, 0

<=y<=height

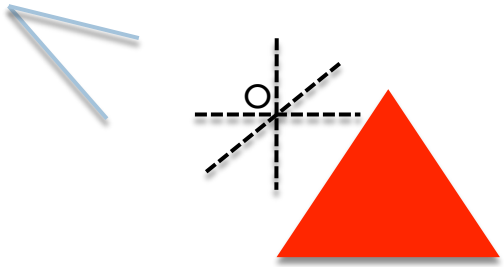
Screen Space

- Screen Space is the intersection of the xy-plane with Image Space.
- Points in image space are mapped into screen space by projecting via a parallel projection, onto the plane $z = 0$.
- Example:
  - a point (0, 0, z) in image space will project to the center of the display screen

Screen Space Diagram



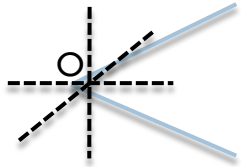
Our goal



World space:

Triangles in native Cartesian coordinates

Camera located anywhere



Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

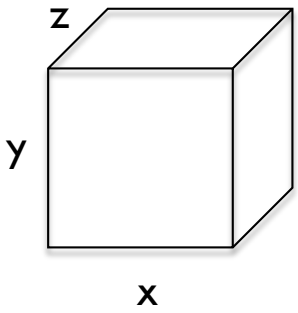
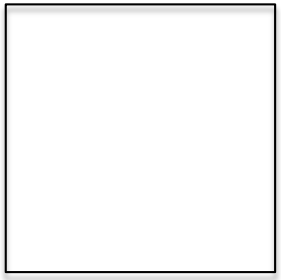


Image space:

All viewable objects within

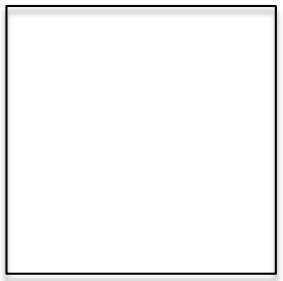
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



Device space:

All viewable objects within

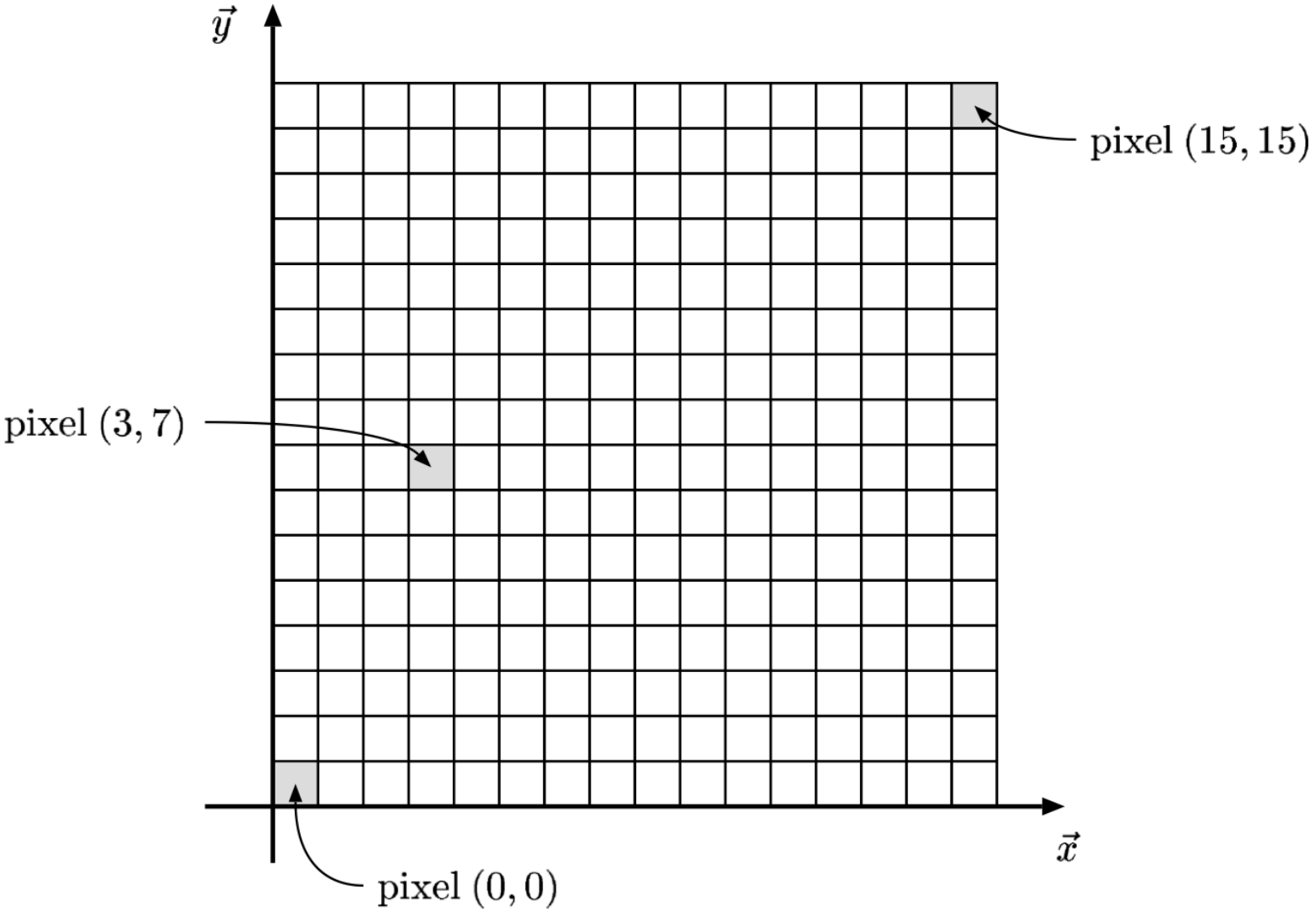
0<=x<=width, 0

<=y<=height

Device Space

- Device Space is the lowest level coordinate system and is the closest to the hardware coordinate systems of the device itself.
- Device space is usually defined to be the n × m array of pixels that represent the area of the screen.
- A coordinate system is imposed on this space by labeling the lower-left-hand corner of the array as (0,0), with each pixel having unit length and width.

Device Space Example



Device Space With Depth Information

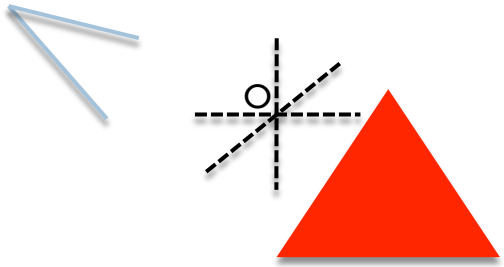
- Extends Device Space to three dimensions by adding z-coordinate of image space.
- Coordinates are (x, y, z) with

$$0 \leq x \leq n$$

$$0 \leq y \leq m$$

z arbitrary (but typically $-1 \leq z \leq +1$ or $-1 \leq z \leq 0$)

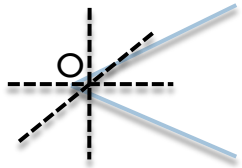
Easiest Transform



World space:

Triangles in native Cartesian coordinates

Camera located anywhere



Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

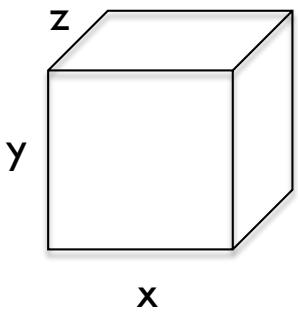
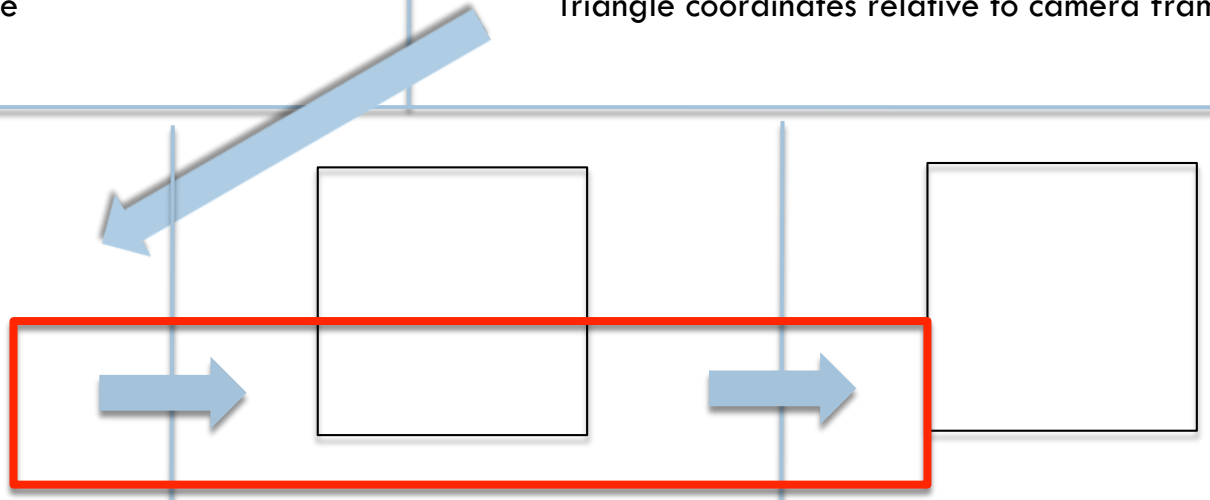


Image space:

All viewable objects within

-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1

Device space:

All viewable objects within

0<=x<=width, 0

<=y<=height

Image Space to Device Space

- (x, y, z) → (x', y', z'), where
  - x' = n*(x+1)/2
  - y' = m*(y+1)/2
  - z' = z
  - (for an n x m image)
- Matrix:

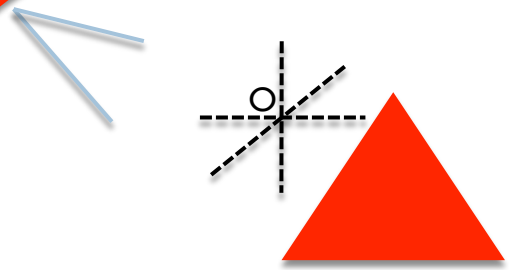
  (x' 0 0 0)

  (0 y' 0 0)

  (0 0 z' 0)

  (0 0 0 1)

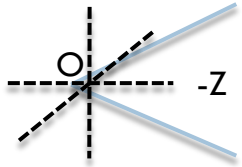
Coming Up on YouTube Lecture



World space:

Triangles in native Cartesian coordinates

Camera located anywhere



Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame

- Need to construct a Camera Frame
- Need to construct a matrix to transform points from Cartesian Frame to Camera Frame
  - Transform triangle by transforming its three vertices