the gamedesigninitiative at cornell university

Lecture 12

Sprite Graphics

Graphics Lectures

- Drawing Images
  - Coordinates & Transforms
  - Images & Colors
- Drawing Perspective
  - Camera
  - Projections
- Drawing Primitives
  - Meshes
  - Shaders

Graphics Lectures

- Drawing Images
  - Coordinates & Transforms
  - Images & Colors

bare minimum to draw graphics

- Drawing Perspective
  - Camera
  - Projections

side-scroller vs. top down

- Drawing Primitives
  - Meshes
  - Shaders

necessary for lighting & shadows

Graphics Lectures

- Drawing Images
  - Coordinates & Transforms
  - Images & Colors
- Drawing Perspective
  - Camera
  - Projections
- Drawing Primitives
  - Meshes
  - Shaders

Animation is part of AI Lectures

Graphics Lectures

- **Drawing Images**
  - Coordinates & Transforms
  - Images & Colors

bare minimum to draw graphics

- Drawing Perspective
  - Camera
  - Projections

side-scroller vs. top down

- Drawing Primitives
  - Meshes
  - Shaders

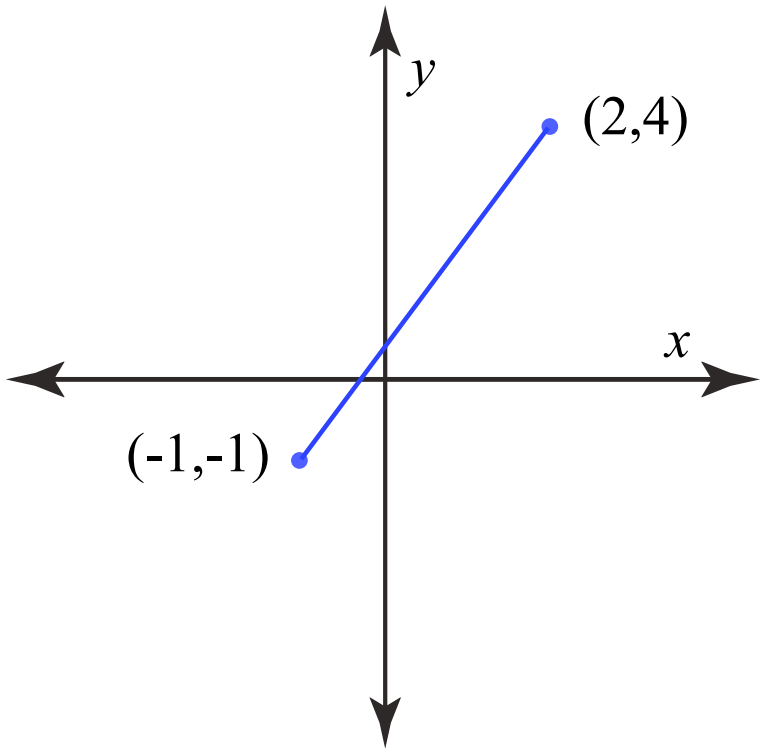
necessary for lighting & shadows

The `SpriteBatch` Interface

- In this class we restrict you to 2D graphics
  - 3D graphics are much more complicated
  - Covered in much more detail in other classes
    - Art 1701: Artist tools for 3D Models
    - CS 5625: Programming with 3D models
- In LibGDX, use the class `SpriteBatch`
  - **Sprite**: Pre-rendered 2D (or even 3D) image
  - All you do is *composite* the sprites together

Drawing in 2 Dimensions

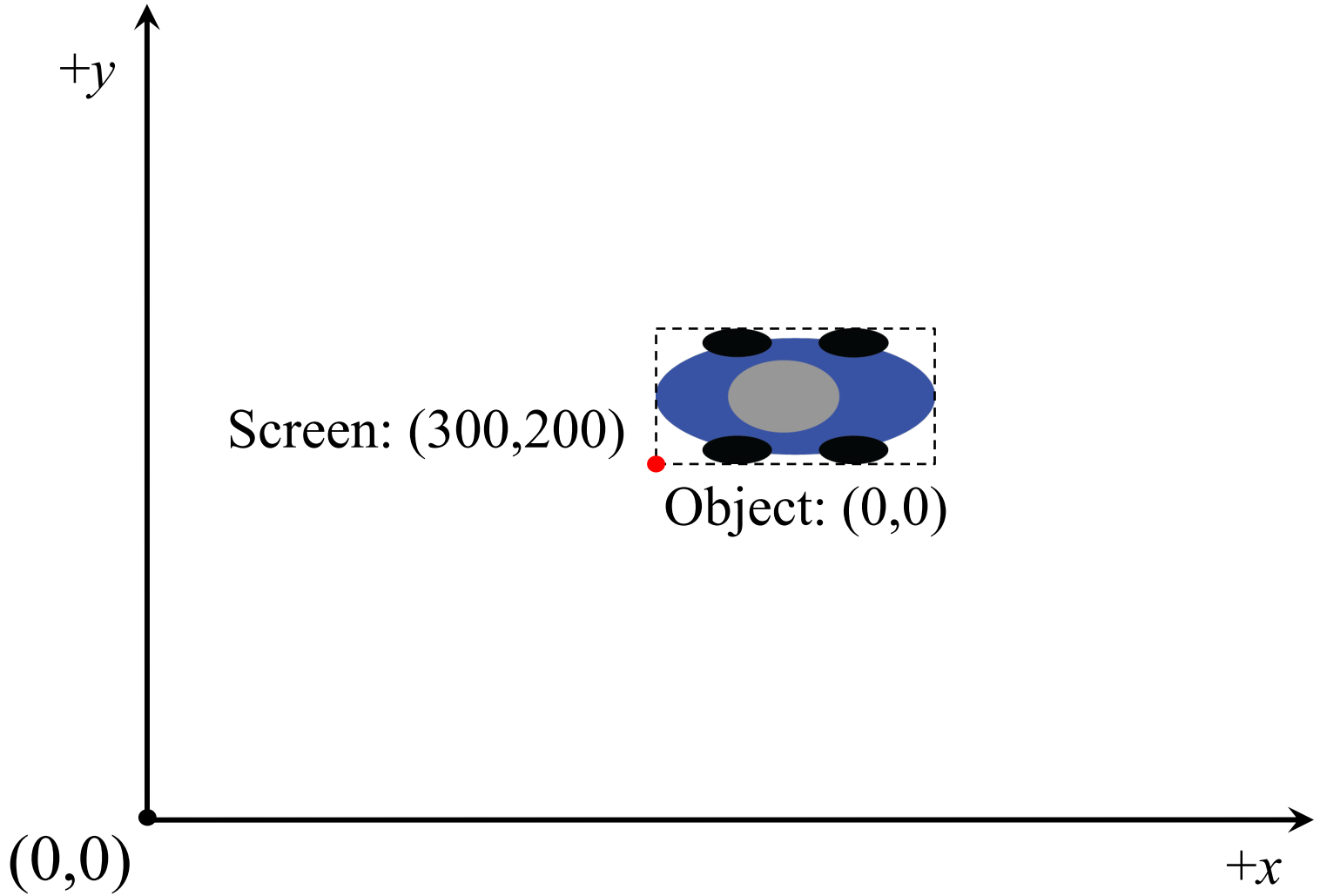
- Use **coordinate systems**
  - Each pixel has a coordinate
  - Draw something at a pixel by
    - Specifying what to draw
    - Specifying where to draw
- Do we draw each pixel?
  - Use a drawing API
  - Given an image; does work
  - What LibGDX gives us



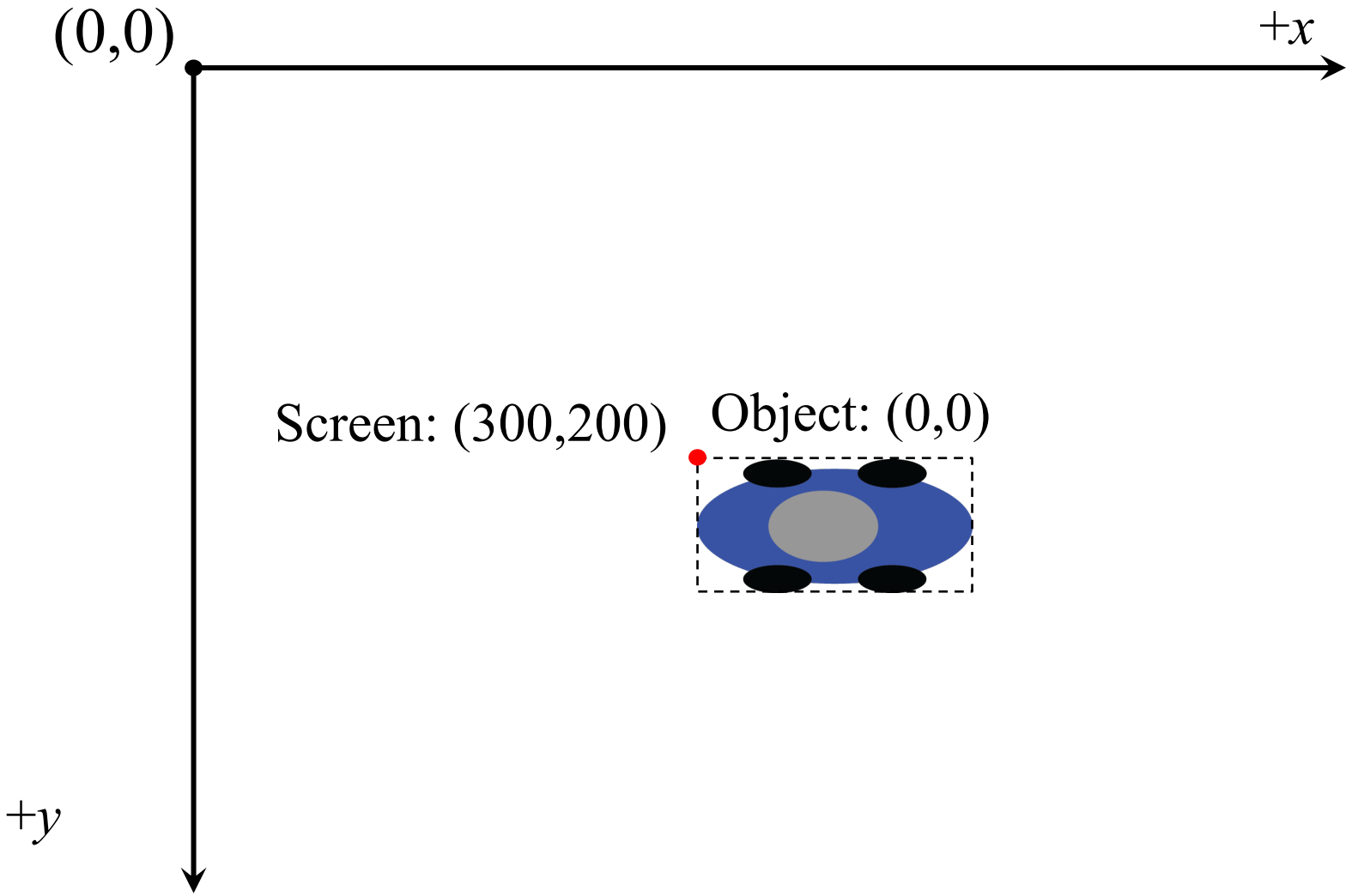
Sprite Coordinate Systems

- **Screen coordinates**: where to paint the image
  - Think screen pixels as a coordinate system
  - Very important for object *transformations*
    - **Example**: scale, rotate, translate
  - In 2D, LibGDX origin is **bottom left** of screen
- **Object coordinate**: location of pixels in object
  - Think of sprite as an image file (it often is)
  - Coordinates are location of pixels in this file
  - Unchanged when object moves about screen

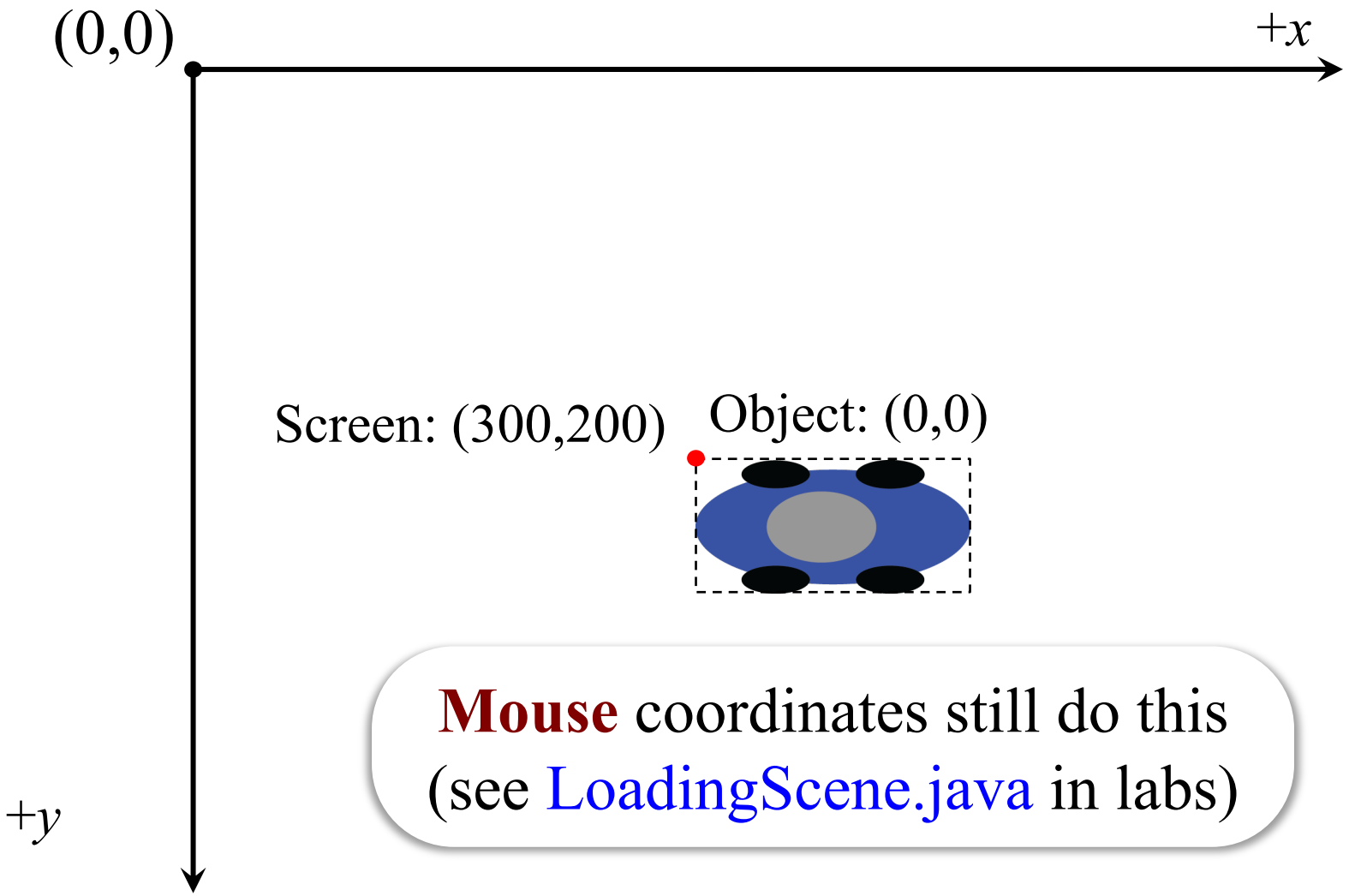
Sprite Coordinate Systems



Historical Coordinate Systems



Historical Coordinate Systems

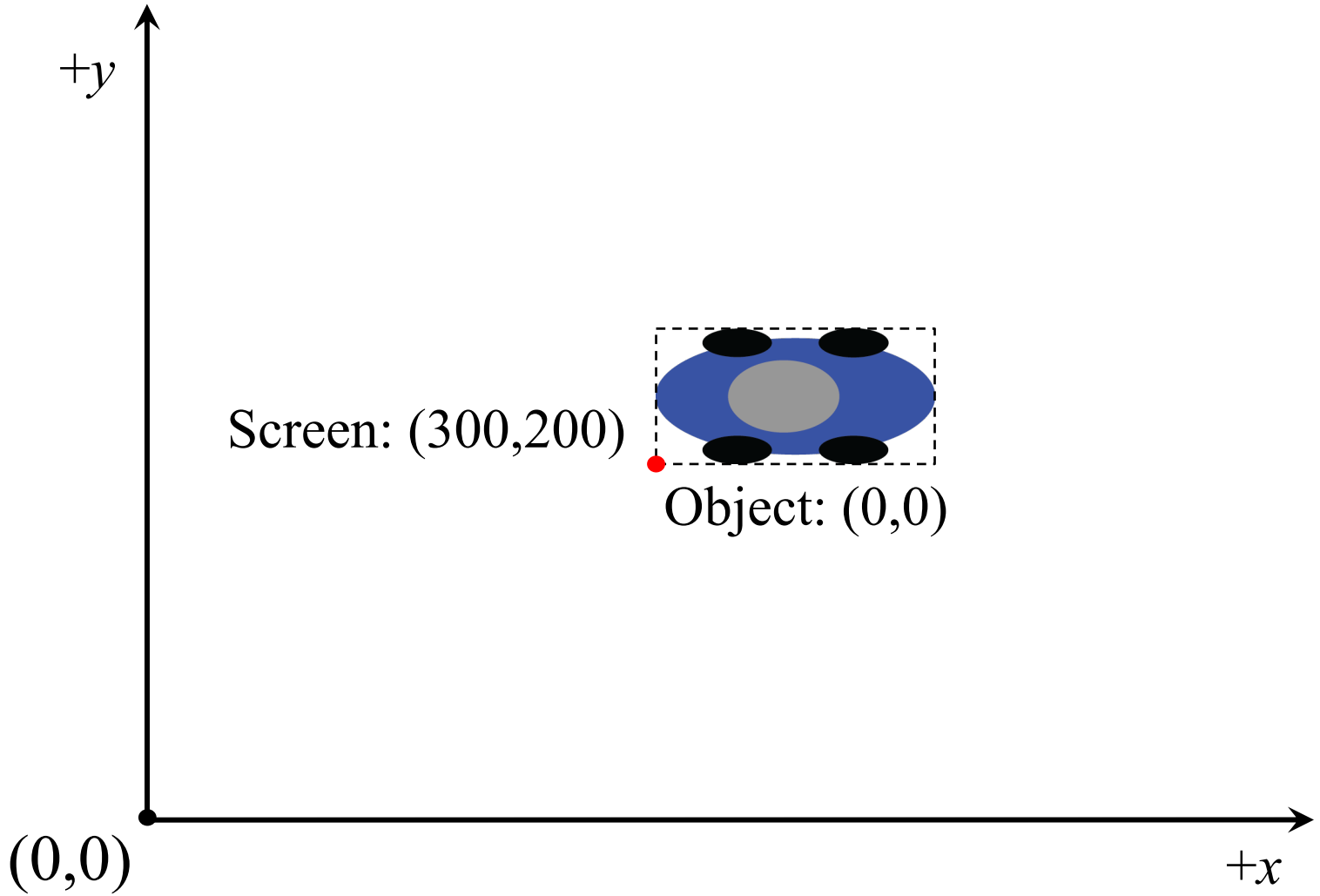


Mouse coordinates still do this
(see LoadingScene.java in labs)

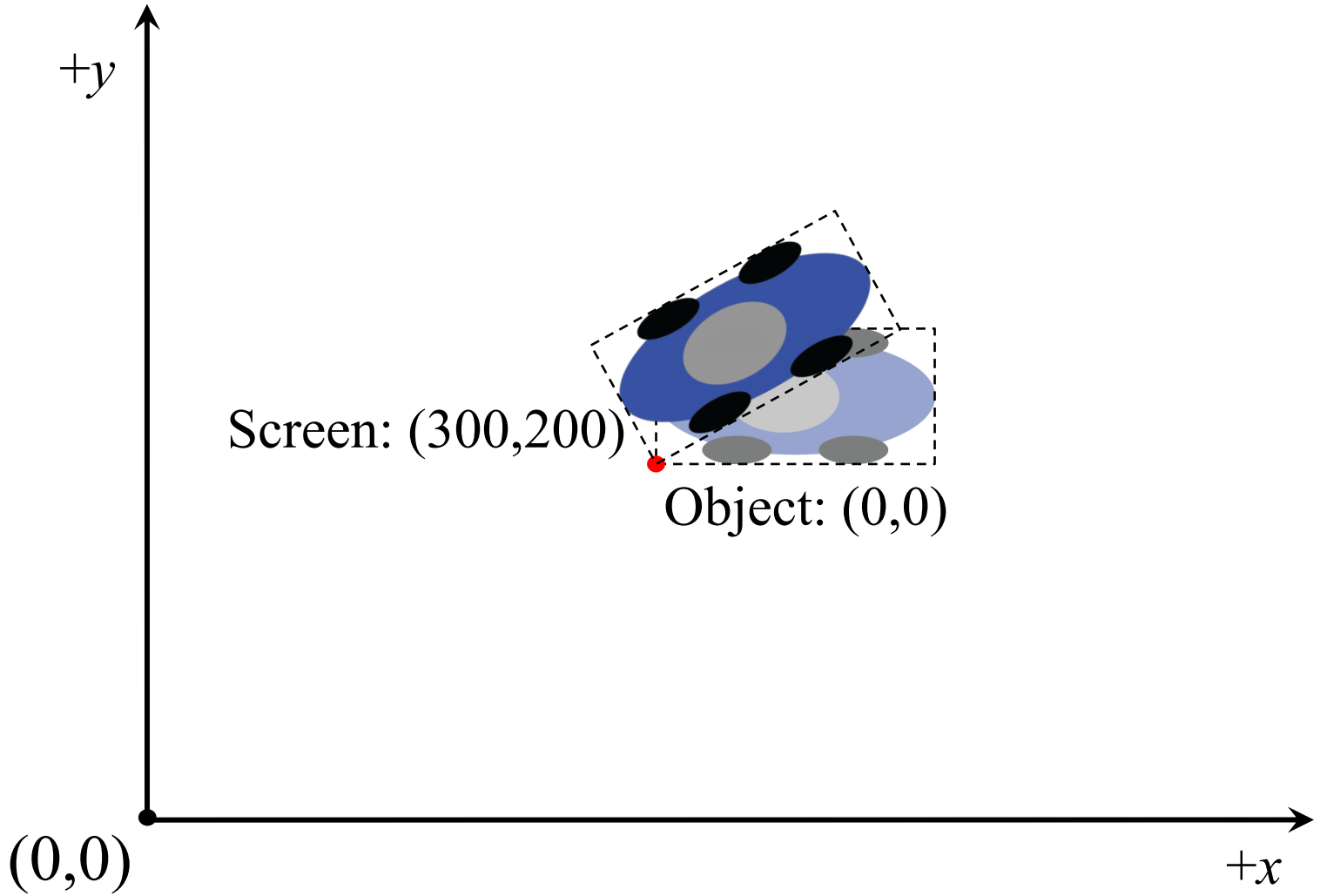
Drawing Sprites

- **Basic instructions**:
  - Set origin for the image in object coordinates
  - Give the `SpriteBatch` a point to draw at
  - Screen places origin of image at that point
- What about the other pixels?
  - Depends on transformations (rotated? scaled?)
  - But these (almost) never affect the origin
- Sometimes we can **reset** the object origin

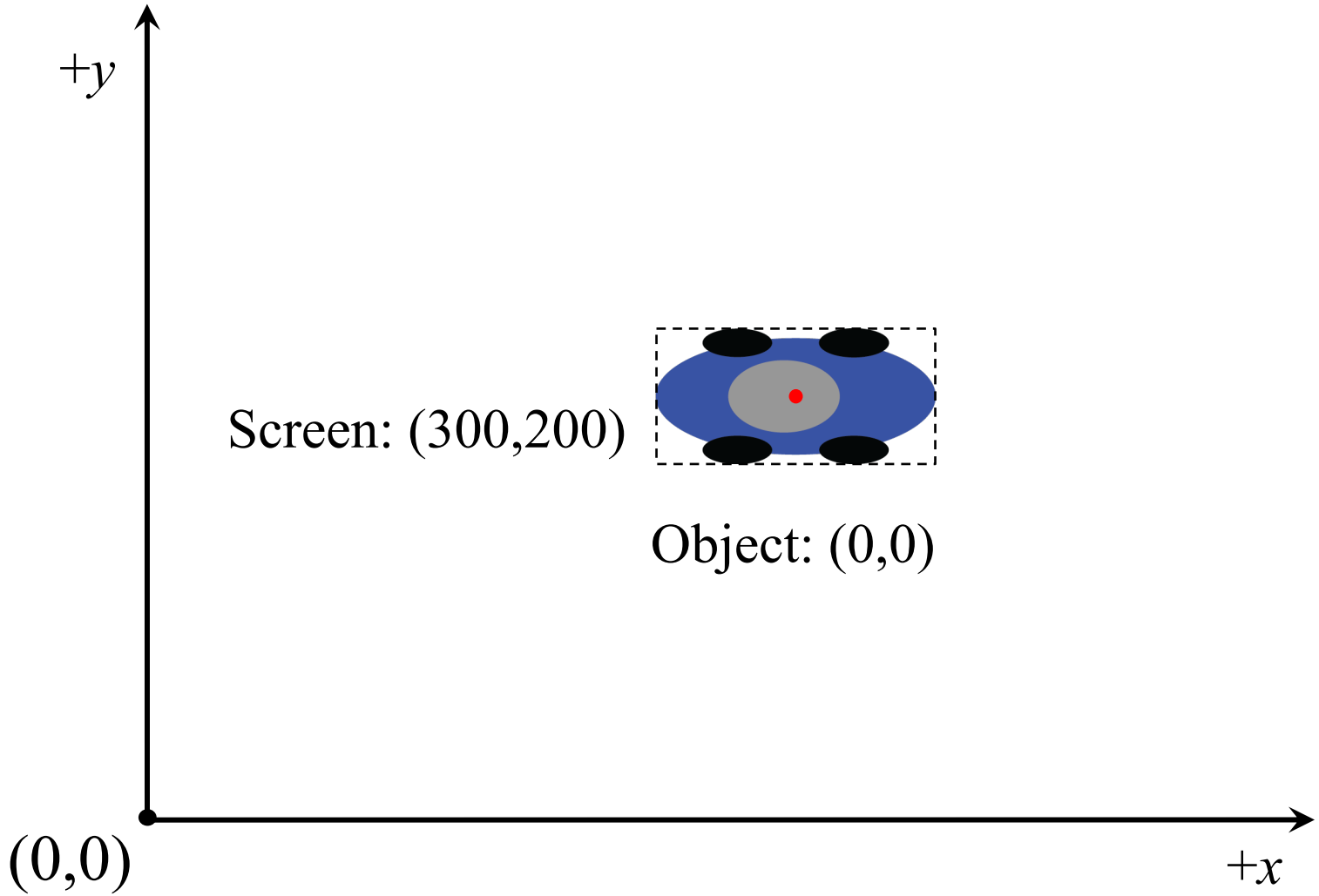
Sprite Coordinate Systems



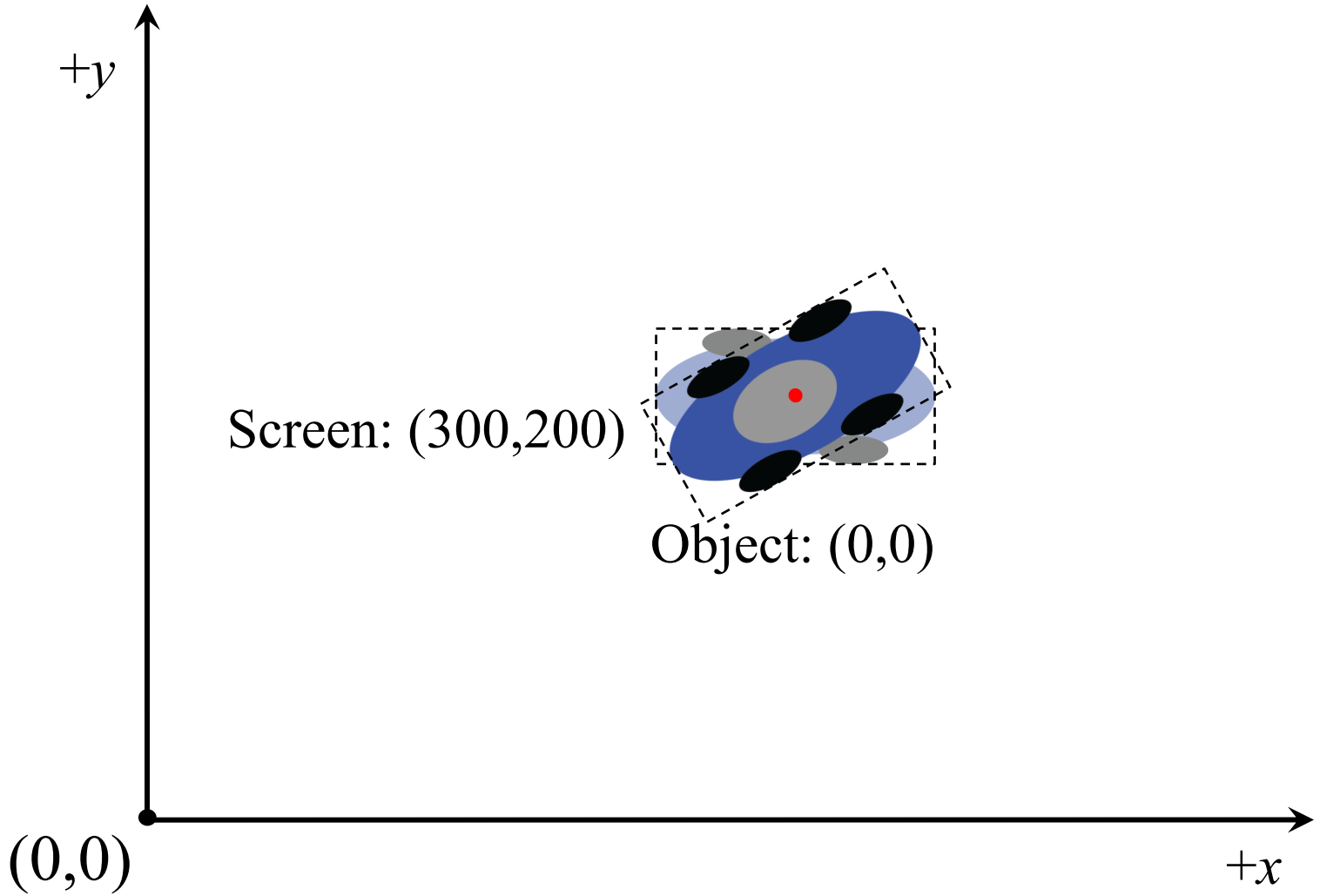
Sprite Coordinate Systems



Sprite Coordinate Systems



Sprite Coordinate Systems



Drawing with SpriteBatch

```
public void draw(float dt) {
    ...
    spriteBatch.begin(camera);
    spriteBatch.draw(image0);
    spriteBatch.draw(image1, pos.x, pos.y);
    ...
    spriteBatch.end();
    ...
}
```

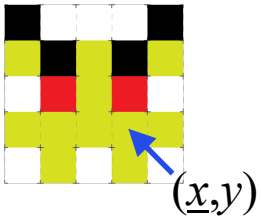
screen coordinates

2D Transforms

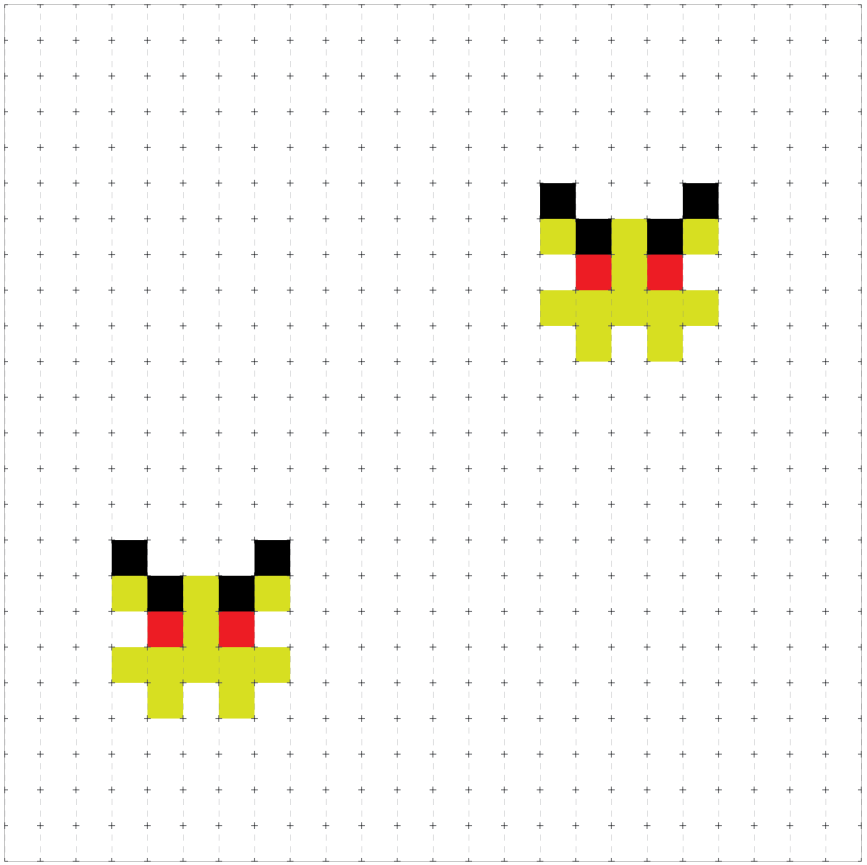
- A function $T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$
  - "Moves" one set of points to another set of points
  - Transforms one "coordinate system" to another
  - The new coordinate system is the distortion
- **Idea**: Draw on paper and then "distort" it
  - **Examples**: Stretching, rotating, reflecting
  - Determines placement of "other" pixels
  - Also allows us to get multiple images for free

The “Drawing Transform”

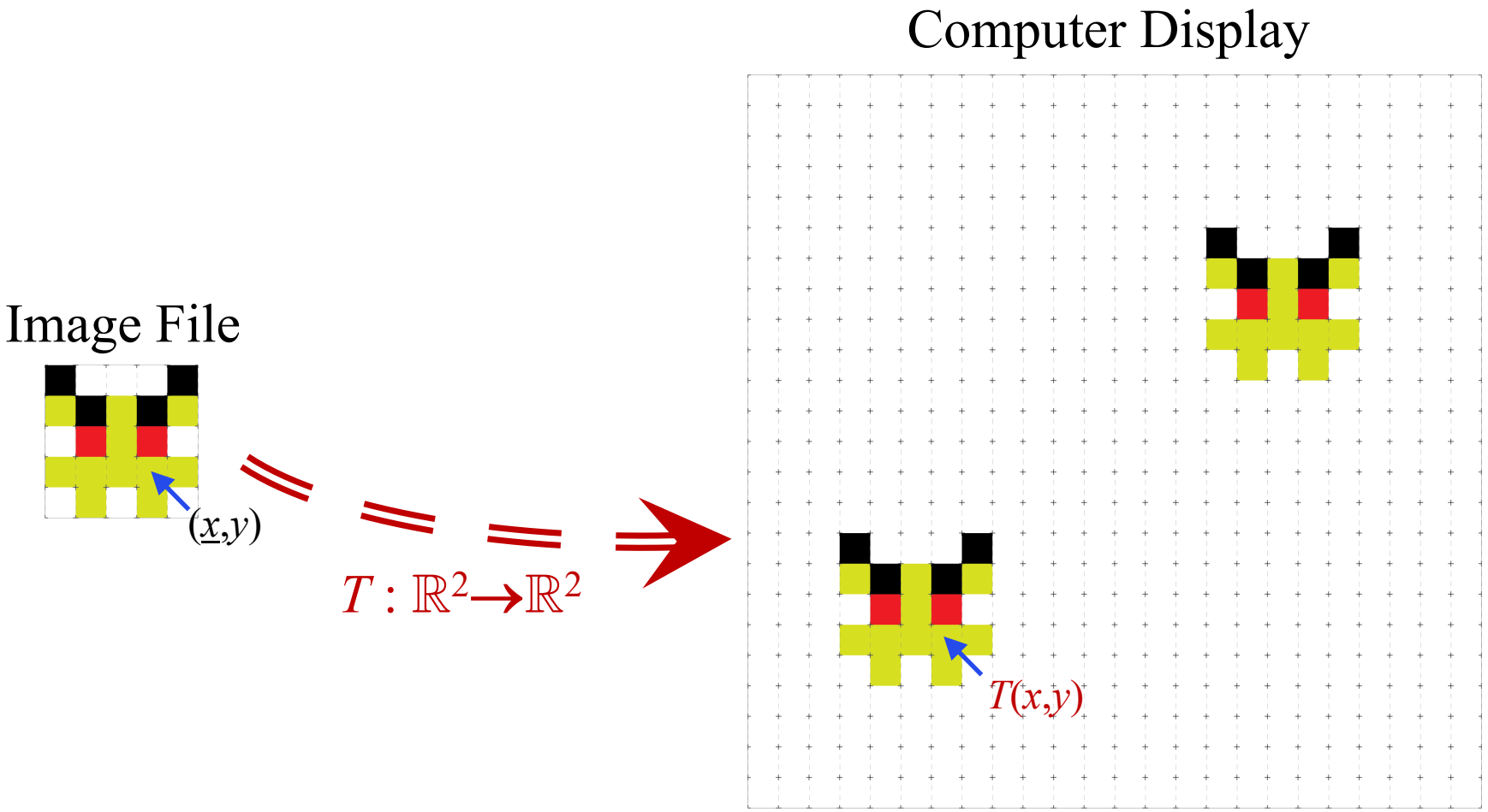
Image File



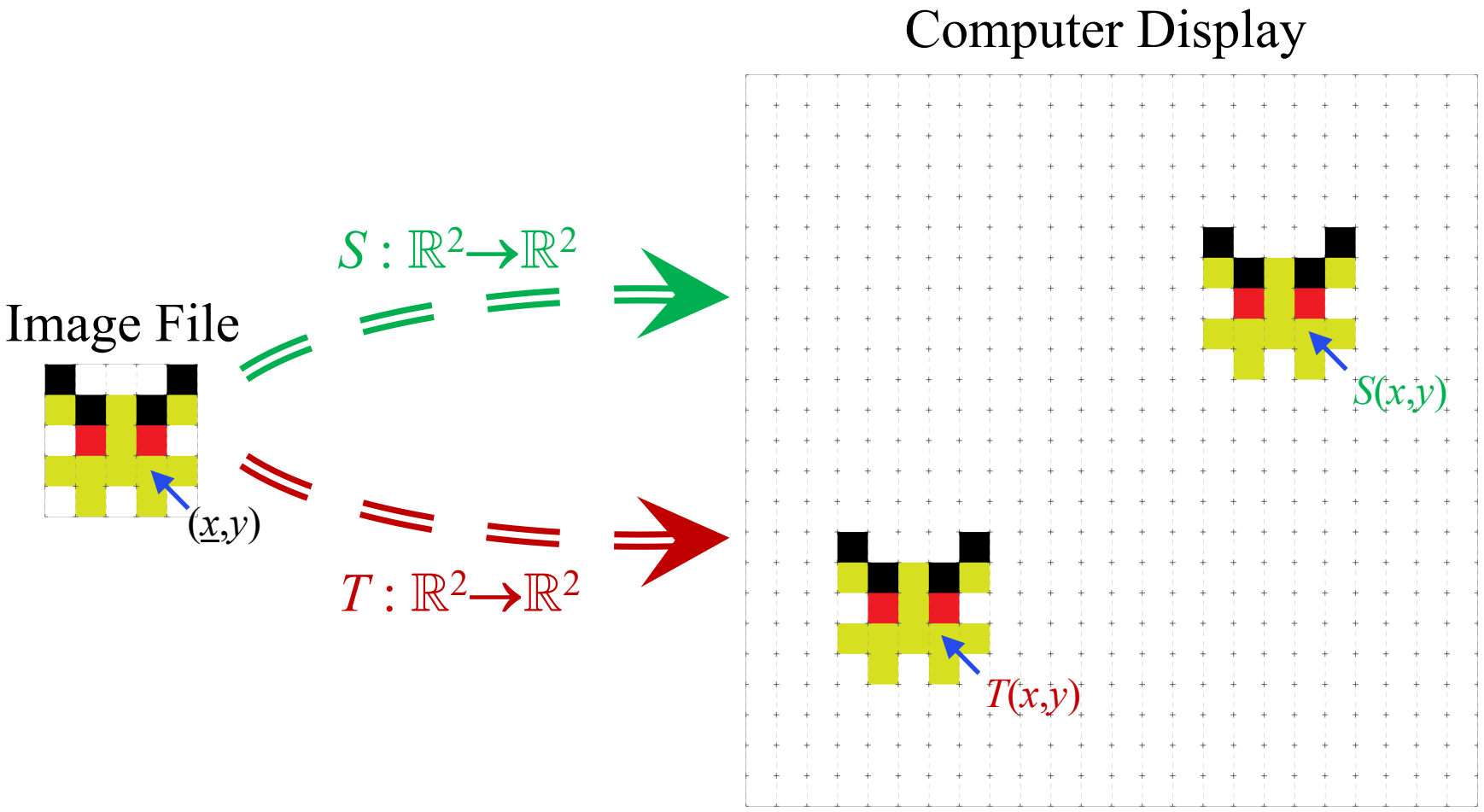
Computer Display



The "Drawing Transform"

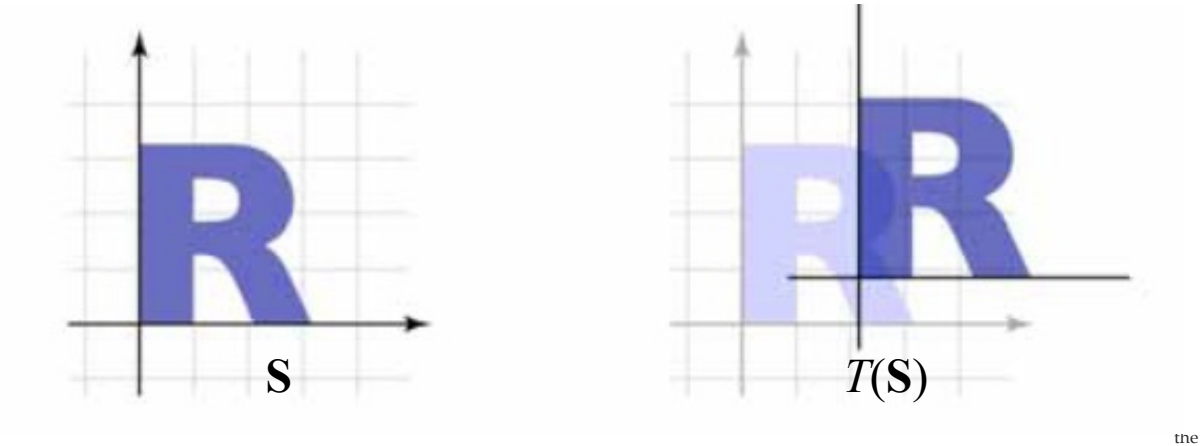


The "Drawing Transform"



Example: Translation

- Simplest transformation: $T(\mathbf{v}) = \mathbf{v} + \mathbf{u}$
  - Shifts object in direction **u**
  - Distance shifted is magnitude of **u**
- Used to place objects on screen
  - By default, object origin is screen origin
  - $T(\mathbf{v}) = \mathbf{v} + \mathbf{u}$ places object origin at **u**



Aside: Matching Your Translation

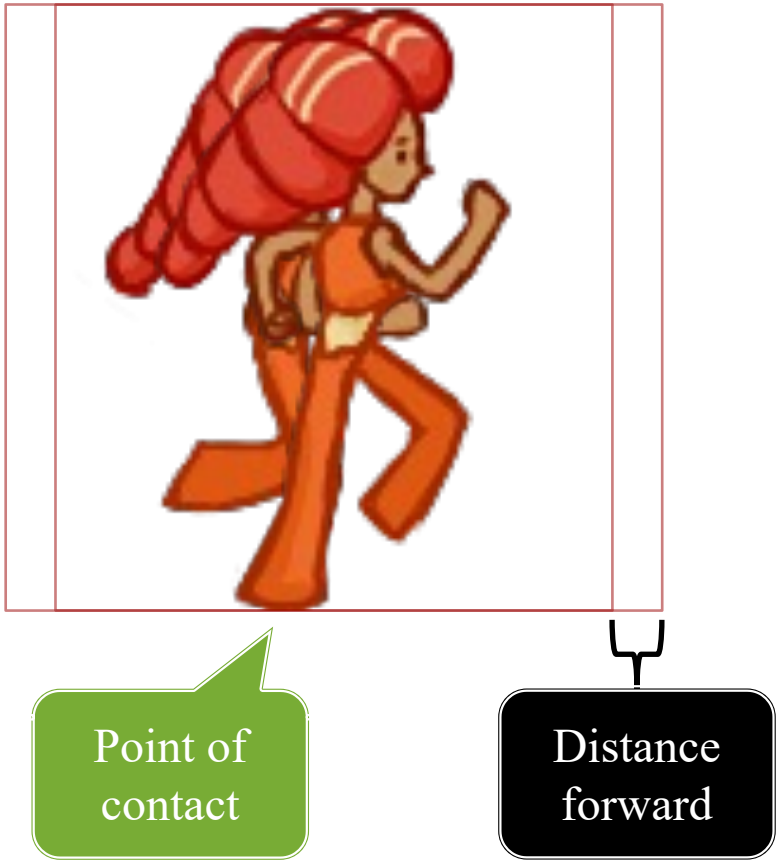
- Movement is *two* things
  - **Animation** of the filmstrip
  - **Translation** of the image
  - These two must align
- **Example:** Walking
  - Foot is point of contact
  - "Stays in place" as move
  - This constrains translation
- Make movement regular
  - Measure distance per frame
  - Keep same across frames

Aside: Matching Your Translation

- Movement is *two* things
  - **Animation** of the filmstrip
  - **Translation** of the image
  - These two must align
- **Example:** Walking
  - Foot is point of contact
  - "Stays in place" as move
  - This constrains translation
- Make movement regular
  - Measure distance per frame
  - Keep same across frames

Aside: Matching Your Translation

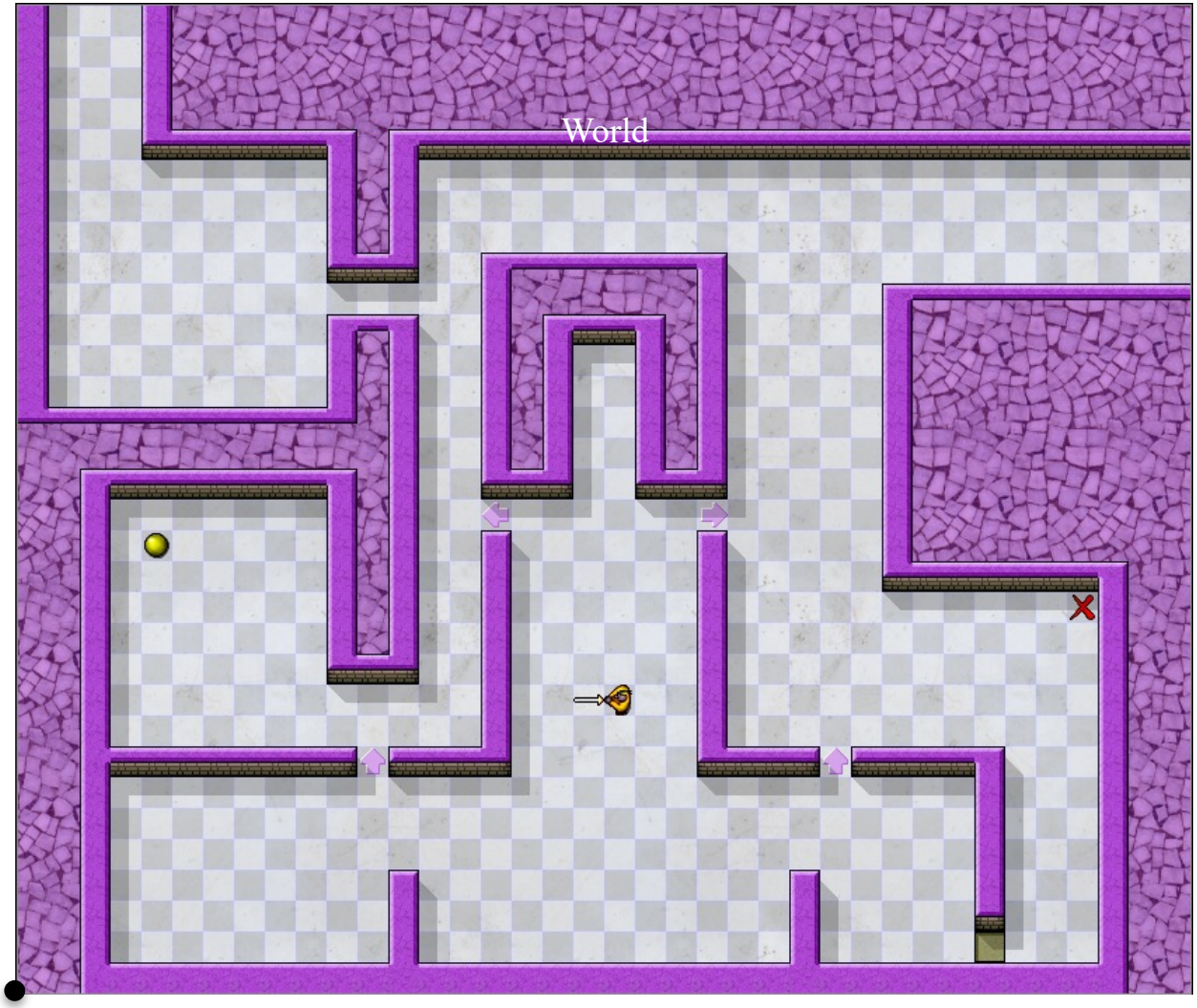
- Movement is *two* things
  - **Animation** of the filmstrip
  - **Translation** of the image
  - These two must align
- **Example:** Walking
  - Foot is point of contact
  - "Stays in place" as move
  - This constrains translation
- Make movement regular
  - Measure distance per frame
  - Keep same across frames



Composing Transforms

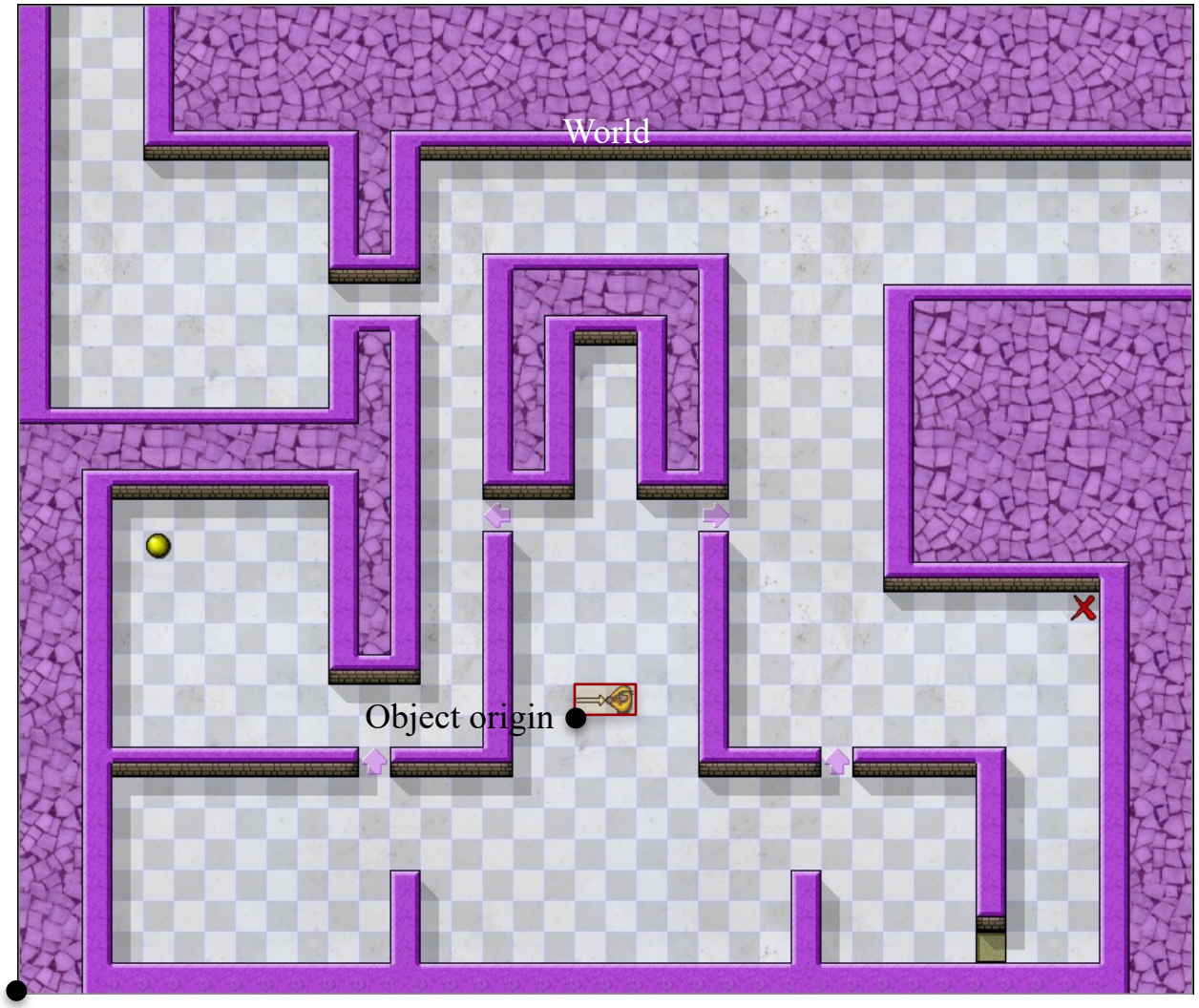
- **Example**: $T : \mathbb{R}^2 \to \mathbb{R}^2$, $S : \mathbb{R}^2 \to \mathbb{R}^2$
  - Assume pixel (a,b) in art file is blue
  - Transform T makes pixel $T(a,b)$ blue
  - Transform $S \circ T$ makes pixel $S(T(a,b))$ blue
- **Strategy**: use transforms as building blocks
  - Think about what you want to do visually
  - Break it into a sequence of transforms
  - Compose the transforms together

Application: Scrolling

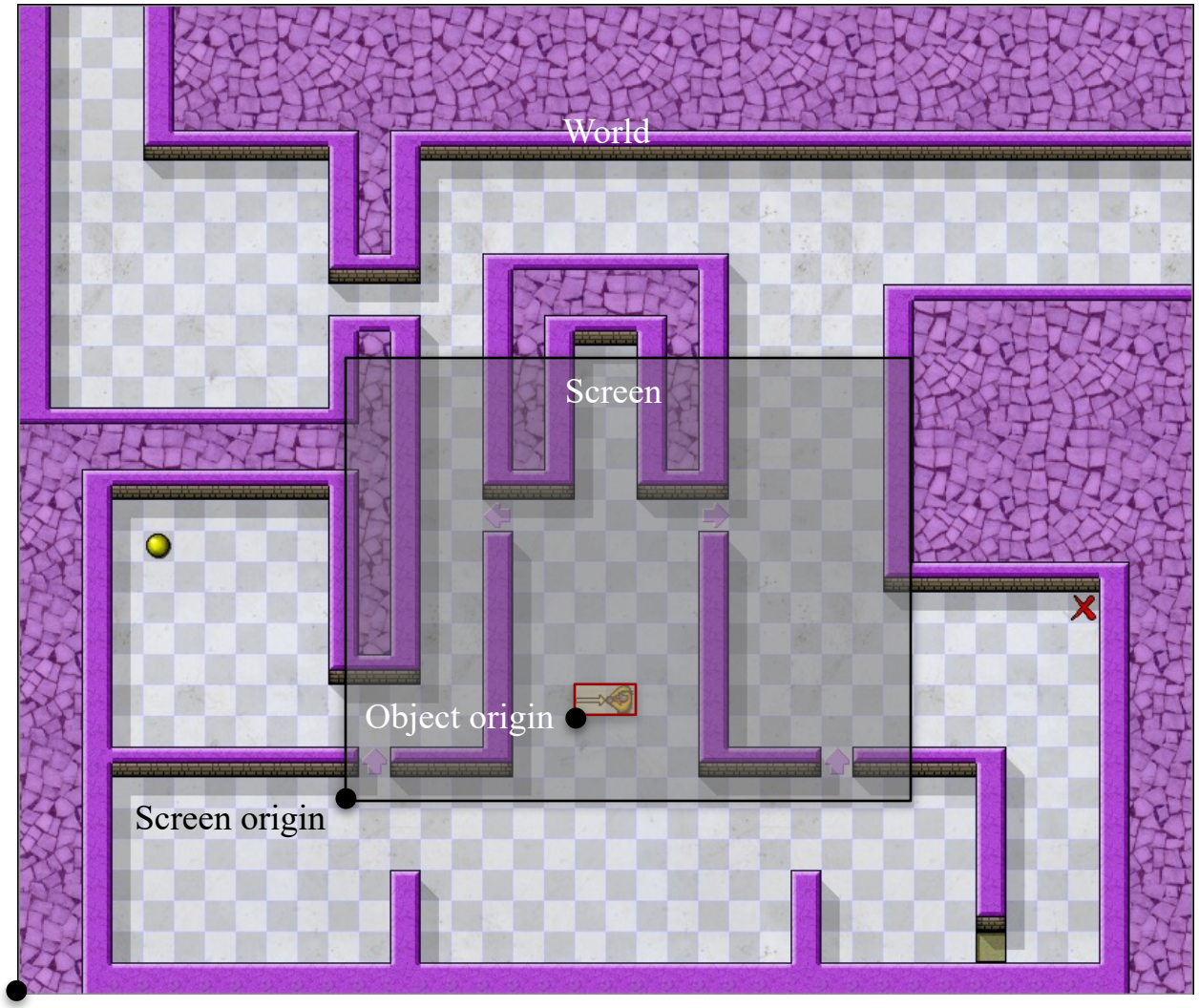


World origin

Application: Scrolling



Application: Scrolling



Scrolling: Two Translations

- Place object in the World at point $\mathbf{p} = (x,y)$
  - Basic drawing transform is $T(\mathbf{v}) = \mathbf{v}+\mathbf{p}$
- Suppose Screen origin is at $\mathbf{q} = (x',y')$
  - Then object is on the Screen at point $\mathbf{p}-\mathbf{q}$
  - $S(\mathbf{v}) = \mathbf{v}-\mathbf{q}$ transforms World coords to Screen
  - $S \circ T(\mathbf{v})$ transforms the Object to the Screen
- This separation makes scrolling easy
  - To move the object, change T but leave S same
  - To scroll the screen, change S but leave T same

Scrolling: Practical Concerns

- Many objects will exists outside screen
  - Can draw if want; graphics card will drop them
  - It is expensive to keep track of them all
  - But is also unrealistic to always ignore them
- In graphics, drawing transform = matrix
  - Hence composition = matrix multiplication
  - Details beyond the scope of this course
  - LibGDX handles all of this for you (sort of)

Using Transforms in LibGDX

- LibGDX has methods for creating transforms
  - Two types depending on application
  - `Affine2` for transforming 2D sprites
  - `Matrix4` for transforming 3D objects
- Parameters fill in details about transform
  - **Example**: Position (x,y) if a translation
  - The most math you will ever need for this
  - Just read the class APIs

Using Transforms in LibGDX

- LibGDX has methods for creating transforms
  - Two types depending on application
  - `Affine2` for transforming 2D sprites
  - `Matrix4` for transforming 3D objects
- Parameters Never Use details about transform
  - **Example**: Position (x,y) if a translation
  - The most math you will ever need for this
  - Just read the class APIs

Positioning in LibGDX

```
public void draw(float dt) {

    Vector2 pos = object.getPosition();




    spriteBatch.begin();
        spriteBatch.draw(image,pos.x,pos.y);
    spriteBatch.end();
}
```

Positioning in LibGDX

```
public void draw(float dt) {
    Affine2 oTran = new Affine2();
    oTran.setToTranslation(object.getPosition());

    spriteBatch.begin();
        spriteBatch.draw(image,oTran);
    spriteBatch.end();
}
```

Translate origin to position in world.

Positioning in LibGDX

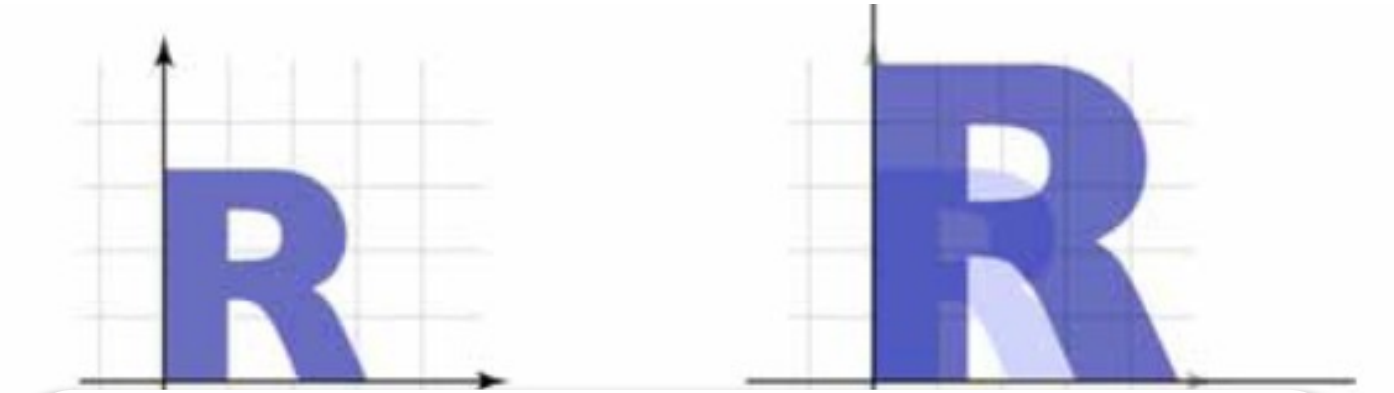
```
public void draw(float dt) {
    Affine2 oTran = new Affine2();
    oTran.setToTranslation(object.getPosition());
    Affine2 wtran = new Affine2();
    Vector2 wPos = viewWindow.getPosition();
    wTran.setToTranslation(-wPos.x,-wPos.y);
    oTran.mul(wTran);
    spriteBatch.begin();
        spriteBatch.draw(image,oTran);
    spriteBatch.end();
}
```

scrolling support

Transform Gallery

- Uniform Scale: $\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx \\ sy \end{bmatrix}$

$$\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$



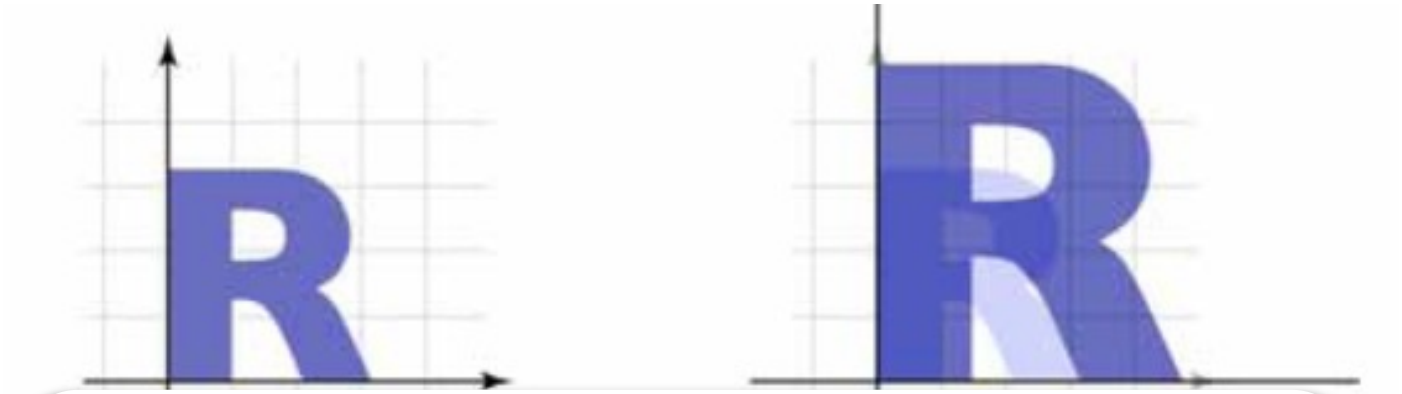
```
affine.setToScaling(s,s);
```

Transform Gallery

- Uniform Scale: $\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx \\ sy \end{bmatrix}$

Represent as 2x2 matrix

$$\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$



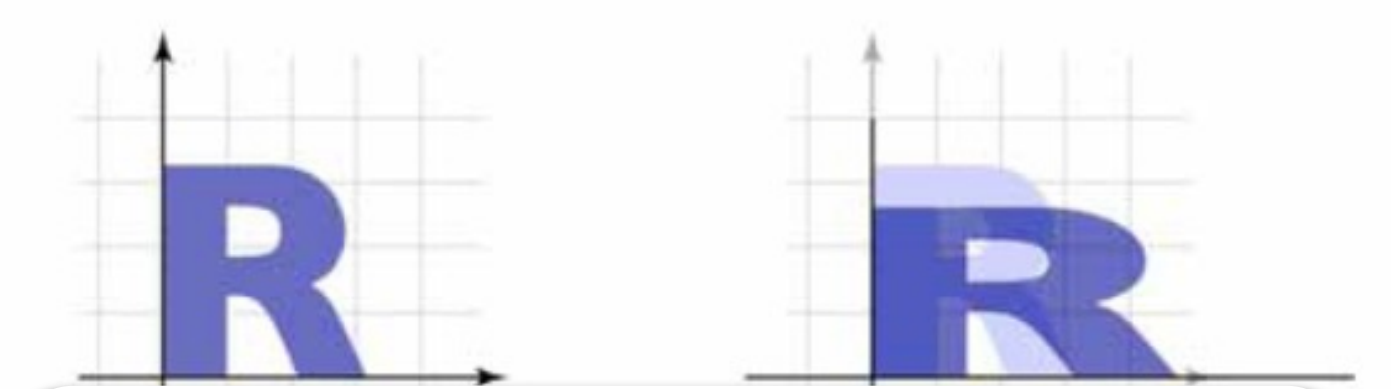
```
affine.setToScaling(s,s);
```

Matrix Transform Gallery

- Nonuniform Scale:

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$$

$$\begin{bmatrix} 1.5 & 0 \\ 0 & 0.8 \end{bmatrix}$$



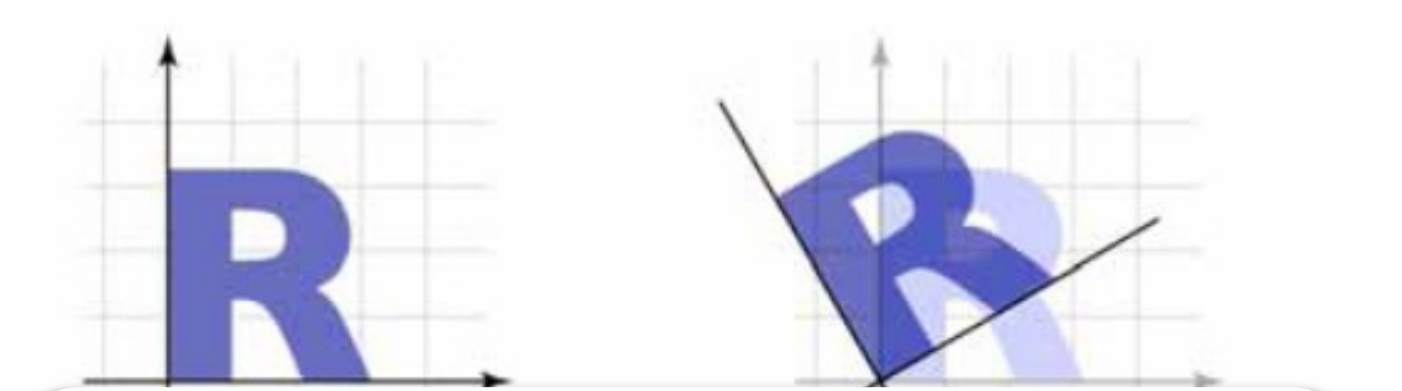
```
affine.setToScaling(sx,sy);
```

Matrix Transform Gallery

- Rotation:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

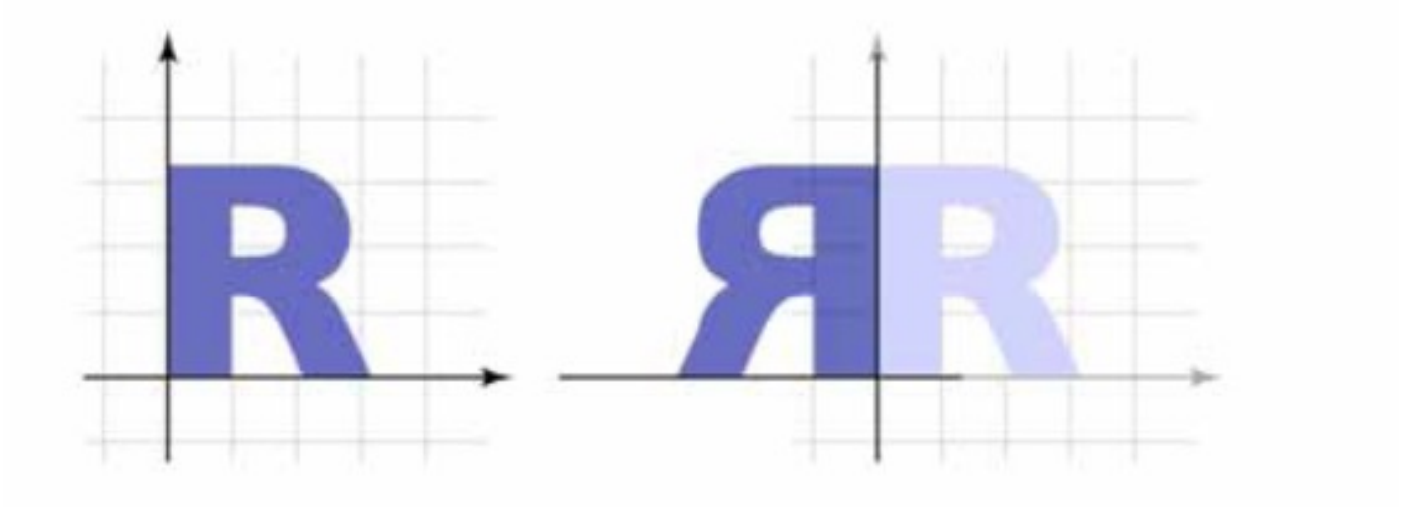
$$\begin{bmatrix} 0.866 & -0.5 \\ 0.5 & 0.866 \end{bmatrix}$$



```
affine.setToRotationRad(angle);
```

Matrix Transform Gallery

- Reflection: $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -x \\ y \end{bmatrix}$
  - View as special case of Scale $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$



Translation Revisited

- Translation is **not** a linear transform
  - To be linear, $T(\mathbf{v}+\mathbf{w}) = T(\mathbf{v})+T(\mathbf{w})$
  - Translation transform is $T(\mathbf{v}) = \mathbf{v}+\mathbf{u}$
  - $T(\mathbf{v})+T(\mathbf{w}) = (\mathbf{v}+\mathbf{u})+(\mathbf{w}+\mathbf{u}) = \mathbf{v}+\mathbf{w}+2\mathbf{u} \neq T(\mathbf{v}+\mathbf{w})$
- But LibGDX treats it like one
  - `Affine2` transforms support translation
  - `Matrix4` supports `matrix.set(affine)`
- What is going on here?

Homogenous Coordinates

- Add an **extra dimension** to the calculation.
  - An extra component w for vectors
  - For affine transformations, can keep $w = 1$
  - Add extra row, column to matrices (so 3×3)
- Dimension is for calculation only
  - We are not in 3D-space **yet**
  - 3D transforms need 4D vectors, 4×4 matrices
- **Don't need to understand the math!**

Homogenous Coordinates

- Linear transforms have dummy row and column

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax+by \\ cx+dy \\ 1 \end{bmatrix}$$

- Translation uses extra column

$$\begin{bmatrix} 1 & 0 & t \\ 0 & 1 & s \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+t \\ y+s \\ 1 \end{bmatrix}$$

Homogenous Coordinates

- Linear transforms have dummy row and column

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax+by \\ cx+dy \\ 1 \end{bmatrix}$$

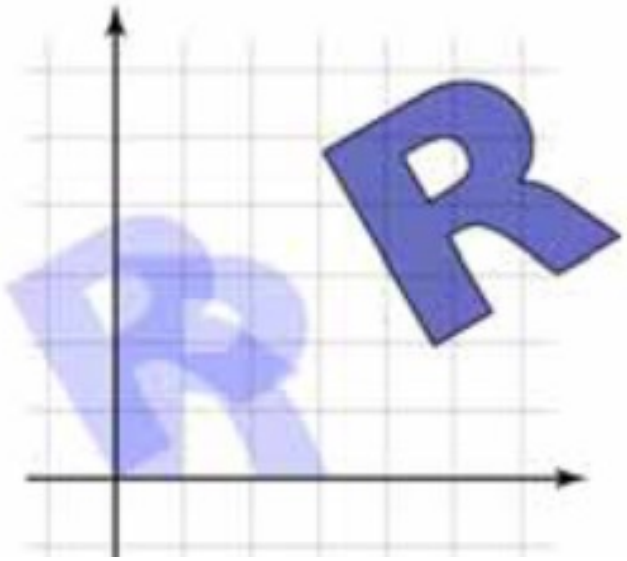
- Translation

See Affine2

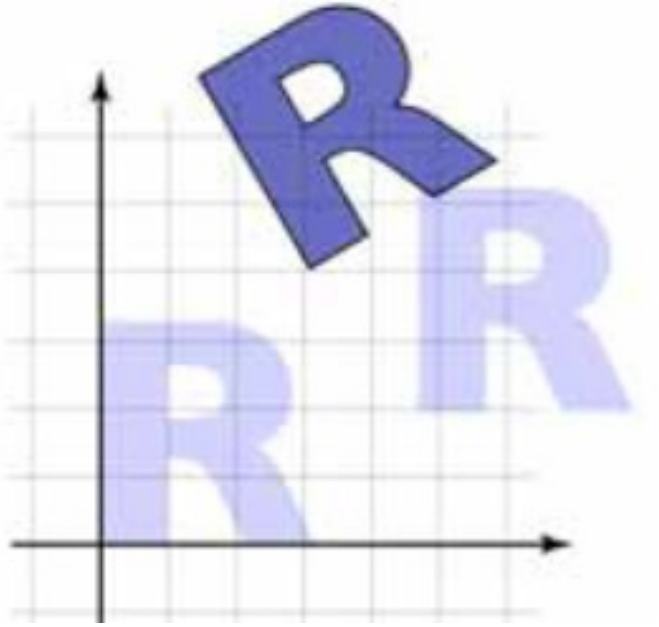
$$\begin{bmatrix} 1 & 0 & t \\ 0 & 1 & s \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+t \\ y+s \\ 1 \end{bmatrix}$$

Compositing Transforms

- Not usually commutative: order matters!



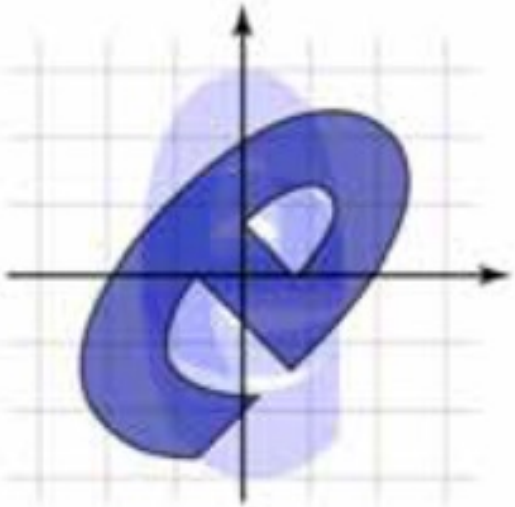
rotate, then translate



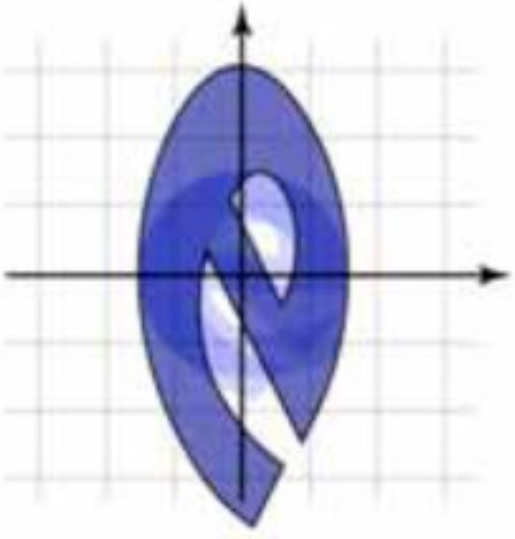
translate, then rotate

Compositing Transforms

- Not usually commutative: order matters!

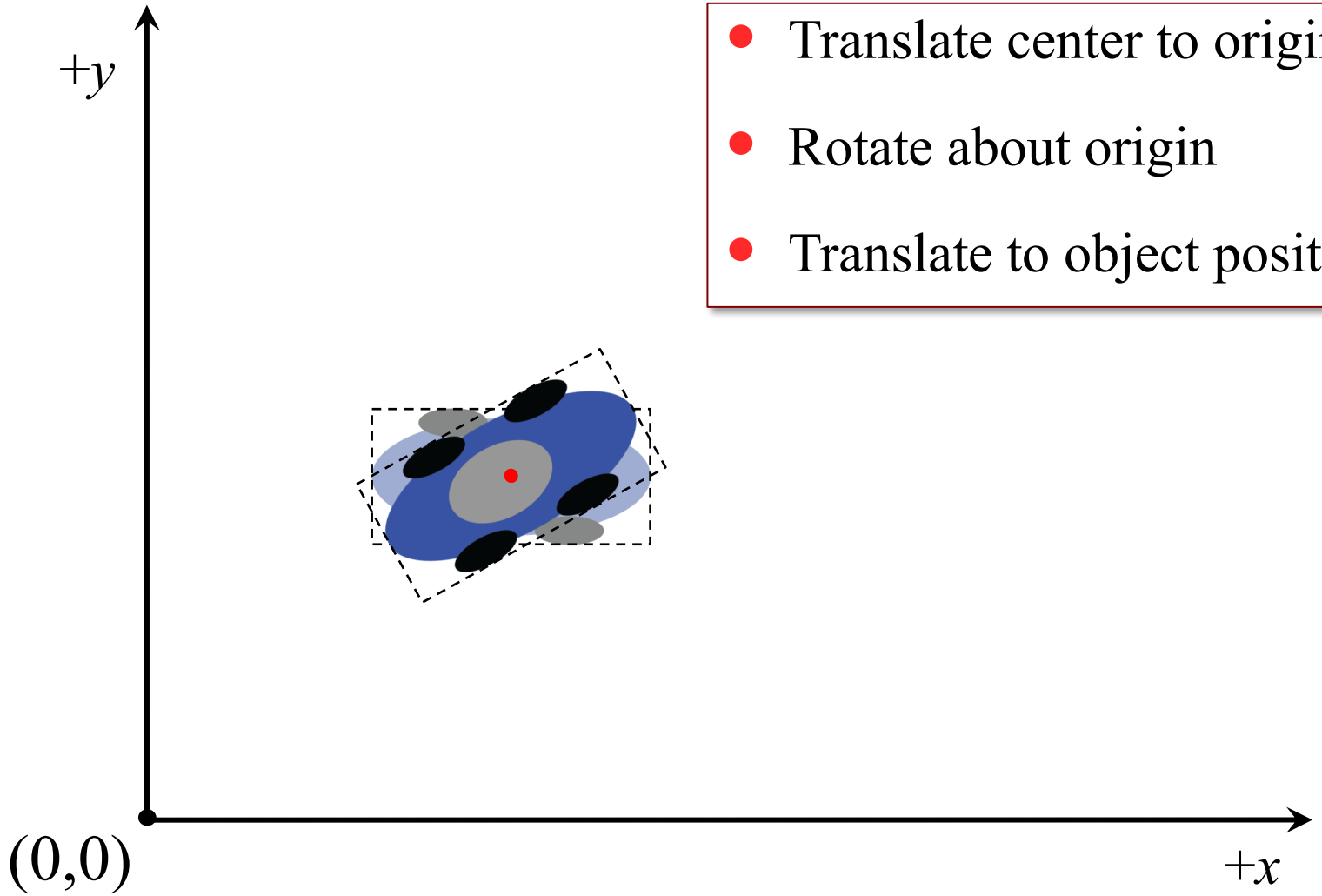


scale, then rotate



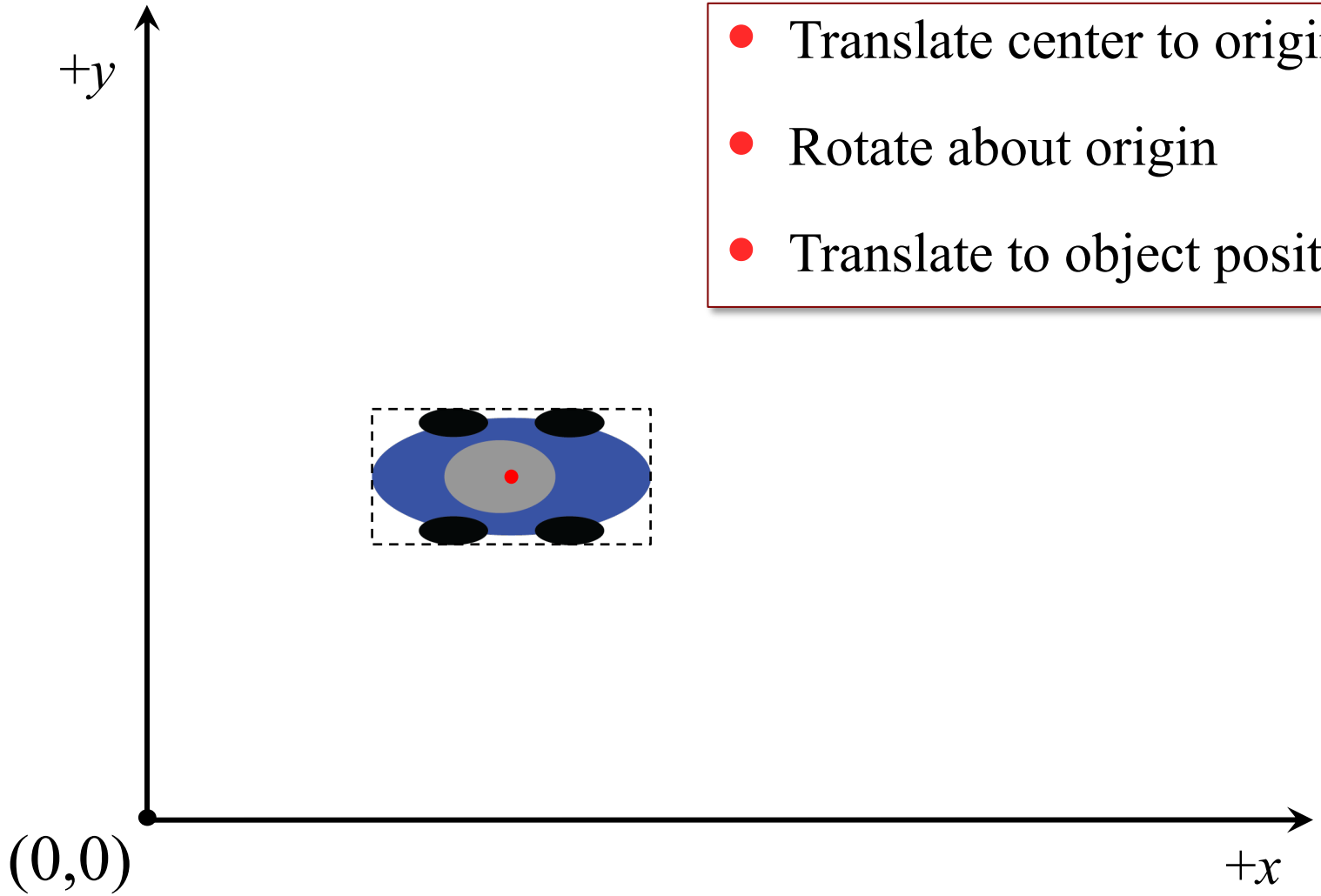
rotate, then scale

Rotating Object About Center



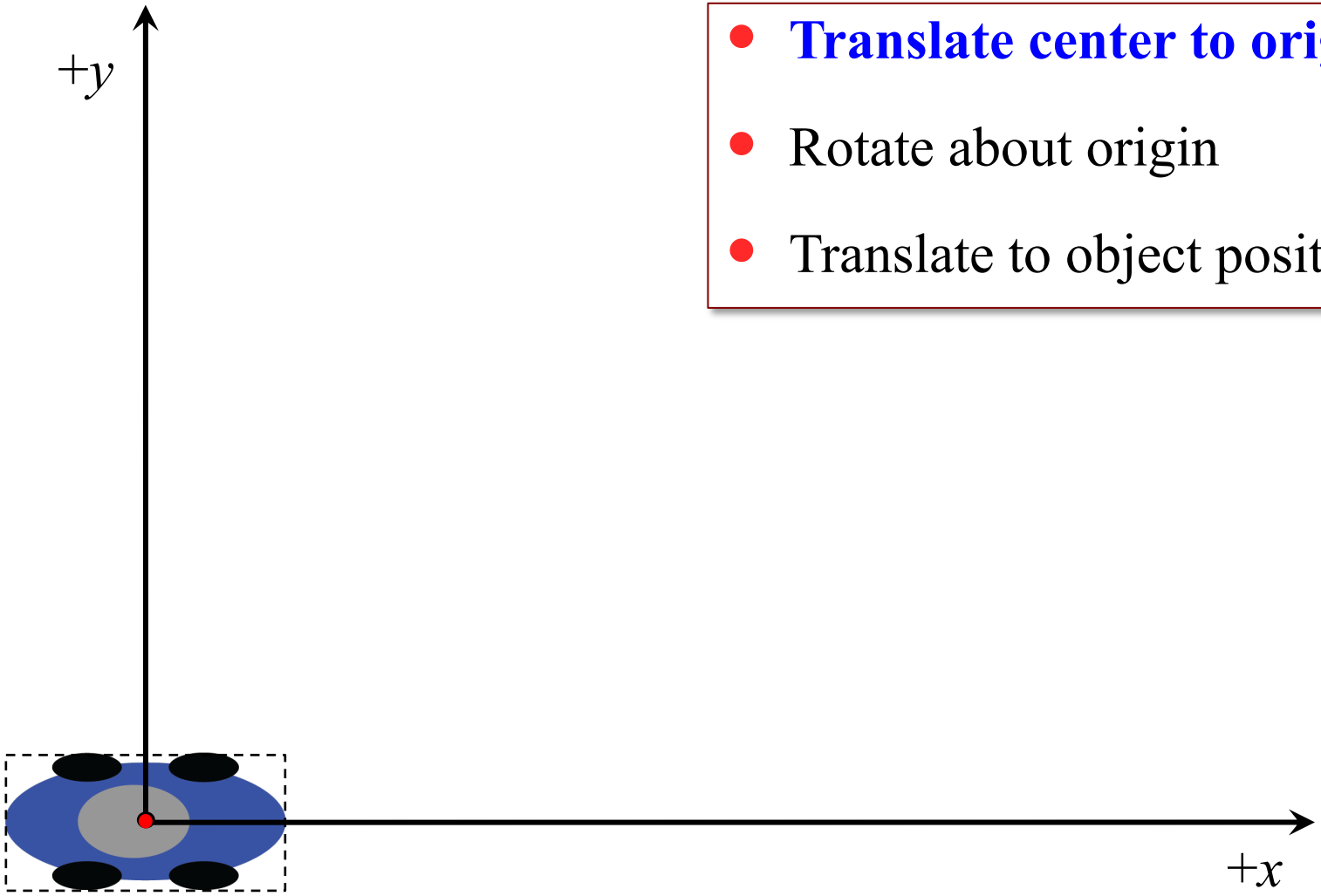
- Translate center to origin
- Rotate about origin
- Translate to object position

Rotating Object About Center



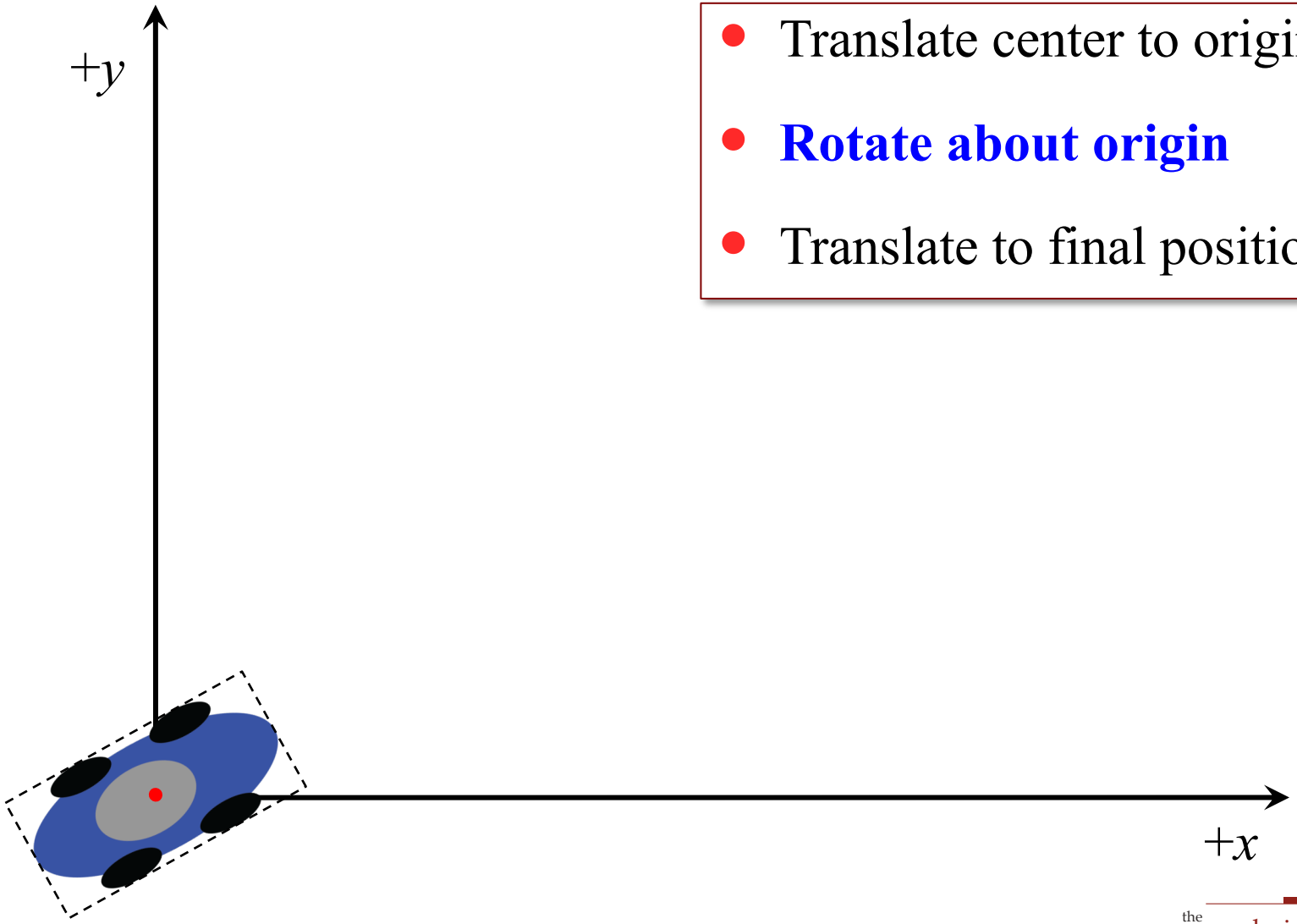
- Translate center to origin
- Rotate about origin
- Translate to object position

Rotating Object About Center



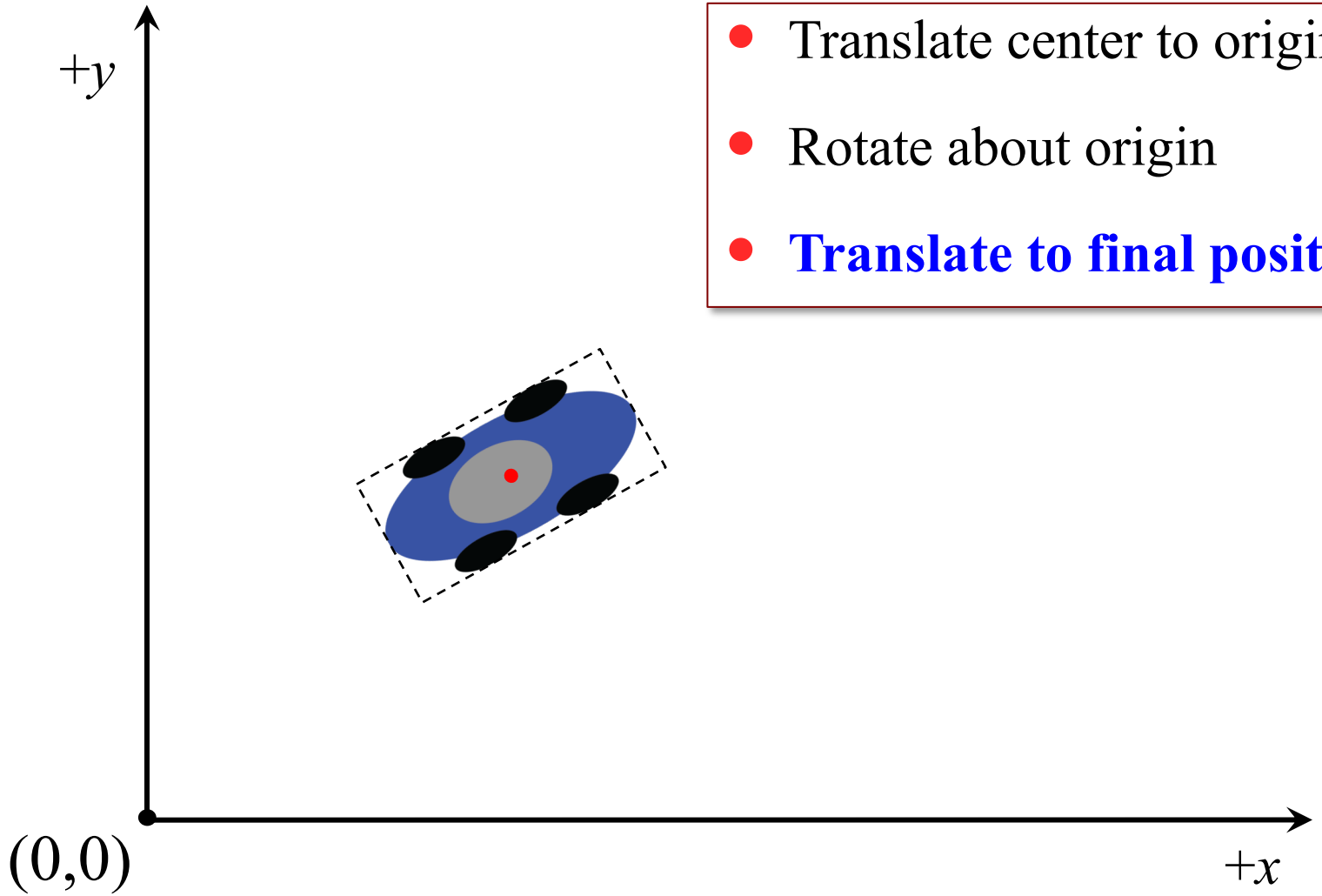
- **Translate center to origin**
- Rotate about origin
- Translate to object position

Rotating Object About Center



- Translate center to origin
- **Rotate about origin**
- Translate to final position

Rotating Object About Center



- Translate center to origin
- Rotate about origin
- **Translate to final position**

Drawing Multiple Objects

- Objects are on a **stack**
  - Images are *layered*
  - Drawn in order given
- Uses **color composition**
  - Often just draws last image
  - What about **transparency**?
- We need to understand…
  - How color is represented
  - How colors combine

Color Representation

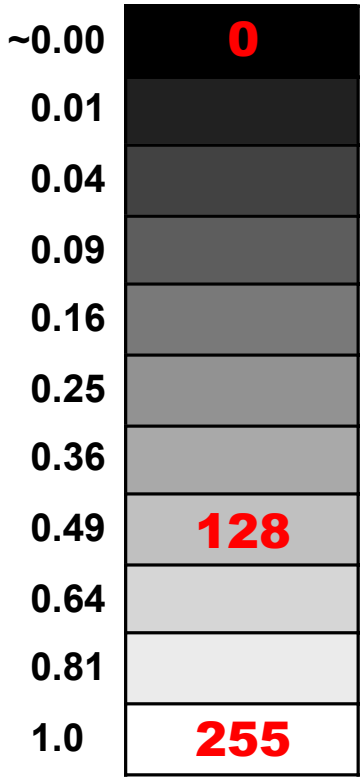
- Humans are **Trichromatic**
  - Any color a blend of three
  - Images from only 3 colors
- Additive Color
  - Each color has an intensity
  - Blend by adding intensities
- Computer displays:
  - Light for each "channel"
  - Red, green and blue
- Aside: Subtractive Color
  - Learned in primary school
  - For pigments, not light

[Cornell CS 465 Slides]

Color Blending Example

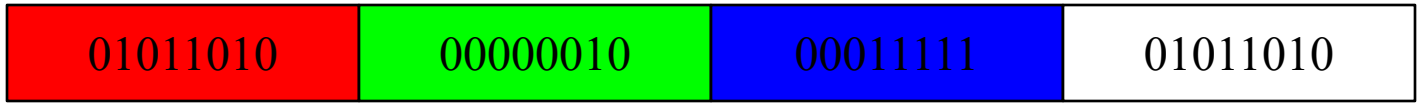
Color Representation

- Each color has an **intensity**
  - Measures amount of light of that color
  - 0 = absent, 1 = maximum intensity
- Real numbers take up a lot of space
  - **Compact representation**: one byte (0-255)
  - As good as human eye can distinguish
- But graphics algorithms require [0,1]
  - Use [0,255] for *storage only*
  - intensity = bits/255.0
  - bits = floor(intensity*255)



Color Representation

- Intensity for three colors: 3 bytes or 24 bits

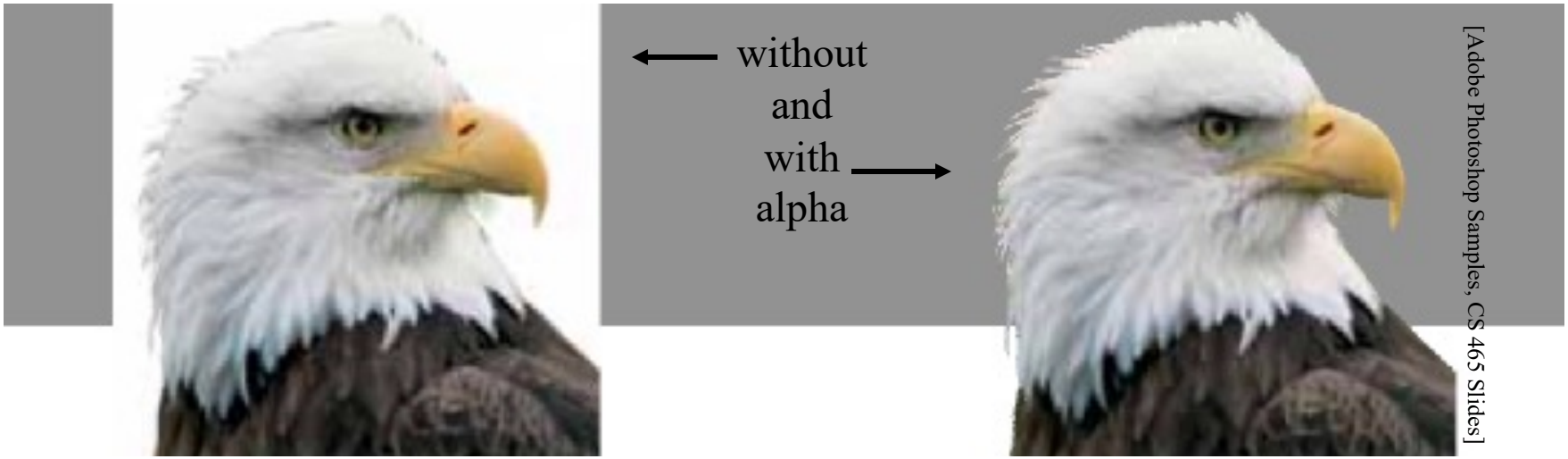


HTML Color	#5A	02	1F	Not Supported

- Store as a 32 bit int; use bit ops to access
  - red: `0x000000FF & integer`
  - green: `0x000000FF & (integer >> 8)`
  - blue: `0x000000FF & (integer >> 16)`
- Most integers are actually 4 bytes; what to do?

The Alpha Channel

- Only used in **color composition**
- Does *not* correspond to a physical light source
  - Allows for transparency of overlapping objects
  - Without it the colors are written atop another



Color Composition

- Trivial example: Video crossfade
  - Smooth transition from one scene to another.

[Chuang et al/ Corel]

$$r_C = tr_A + (1-t)r_B$$
$$g_C = tg_A + (1-t)g_B$$
$$b_C = tb_A + (1-t)b_B$$

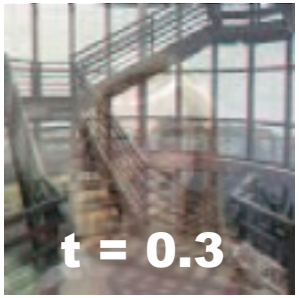
per pixel calculation

  - Note sums weight to 1.0
    - No unexpected brightening or darkening
    - No out-of-range results
  - This is an example of **linear interpolation**

Color Composition

- Trivial example: Video crossfade
  - Smooth transition from one scene to another.

[Chuang et al/ Corel]



$$r_C = tr_A + (1-t)r_B$$
$$g_C = tg_A + (1-t)g_B$$
$$b_C = tb_A + (1-t)b_B$$

per pixel calculation

  - Note sums weight to 1.0
    - No unexpected brightening or darkening
    - No out-of-range results
  - This is an example of **linear interpolation**

Color Composition

- Trivial example: Video crossfade
  - Smooth transition from one scene to another.

[Chuang et al/ Corel]

$$r_C = tr_A + (1-t)r_B$$
$$g_C = tg_A + (1-t)g_B$$
$$b_C = tb_A + (1-t)b_B$$

per pixel calculation

  - Note sums weight to 1.0
    - No unexpected brightening or darkening
    - No out-of-range results
  - This is an example of **linear interpolation**

Color Composition

- Trivial example: Video crossfade
  - Smooth transition from one scene to another.

[Chuang et al/ Corel]

$$r_C = tr_A + (1-t)r_B$$
$$g_C = tg_A + (1-t)g_B$$
$$b_C = tb_A + (1-t)b_B$$

per pixel calculation

  - Note sums weight to 1.0
    - No unexpected brightening or darkening
    - No out-of-range results
  - This is an example of **linear interpolation**

Color Composition

- Trivial example: Video crossfade
  - Smooth transition from one scene to another.

[Chuang et al/ Corel]

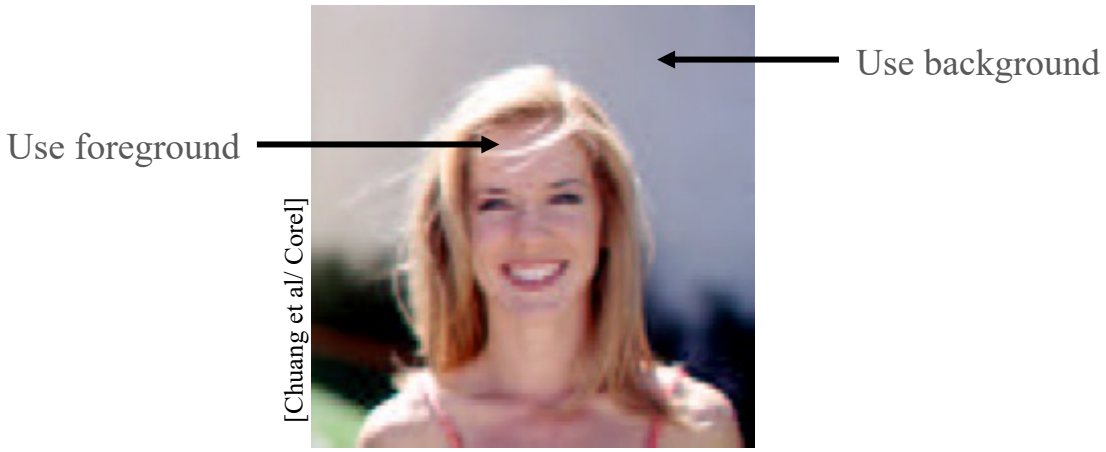
$$r_C = tr_A + (1-t)r_B$$
$$g_C = tg_A + (1-t)g_B$$
$$b_C = tb_A + (1-t)b_B$$

per pixel calculation

  - Note sums weight to 1.0
    - No unexpected brightening or darkening
    - No out-of-range results
  - This is an example of **linear interpolation**

Foreground and Background

- In many cases, just adding is not enough
  - Want some elements in composite, not others
  - Do not want transparency of crossfade
- How we compute new image varies with position.



- Need to store a tag indicating parts of interest

Binary Image Mask

- First idea: Store one bit per pixel
  - Answers question "Is this pixel in foreground?"

[Chuang et al/ Corel] [Cornell PCG]

  - Does not work well near the edges

Binary Image Mask

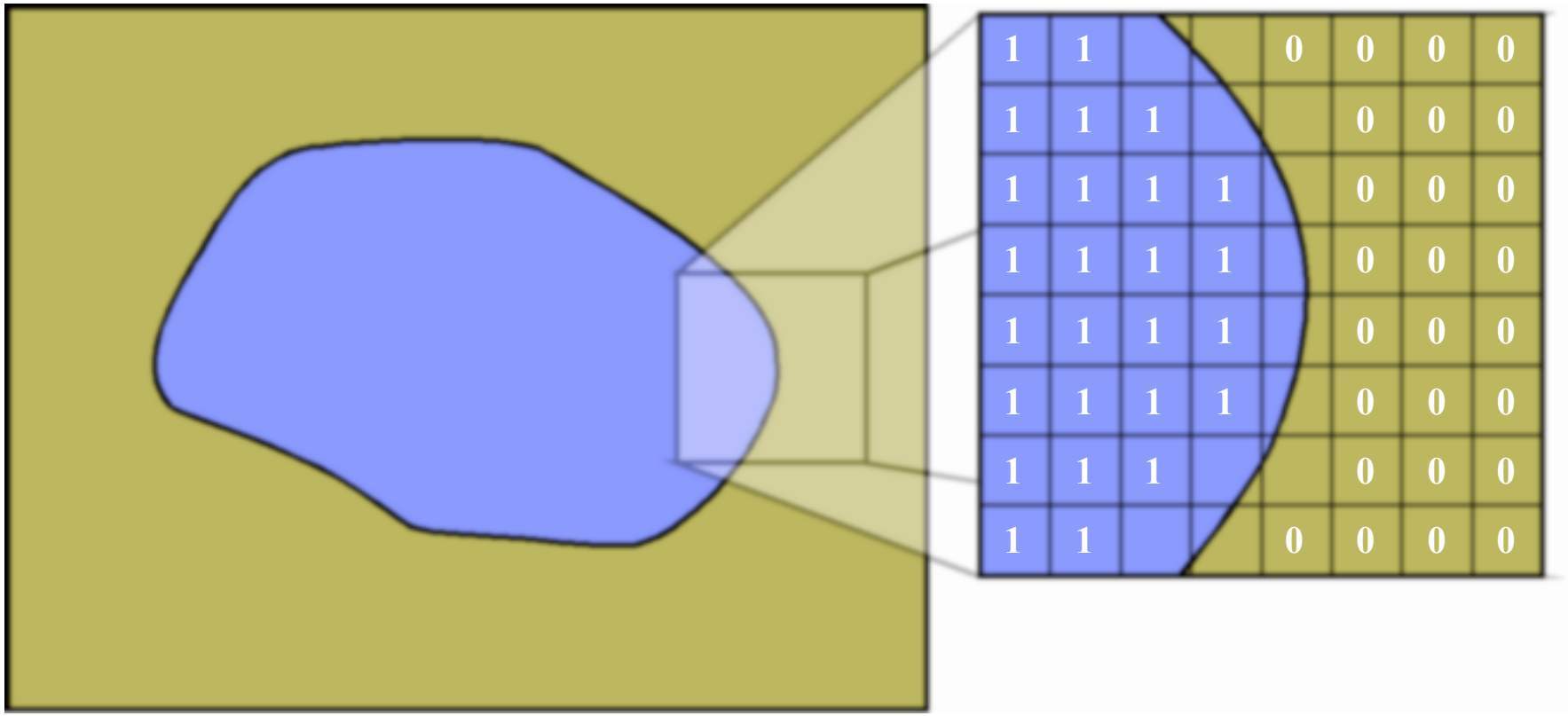
- First idea: Store one bit per pixel
  - Answers question "Is this pixel in foreground?"

[Chuang et al/ Corel] [Cornell PCG]

  - Does not work well near the edges

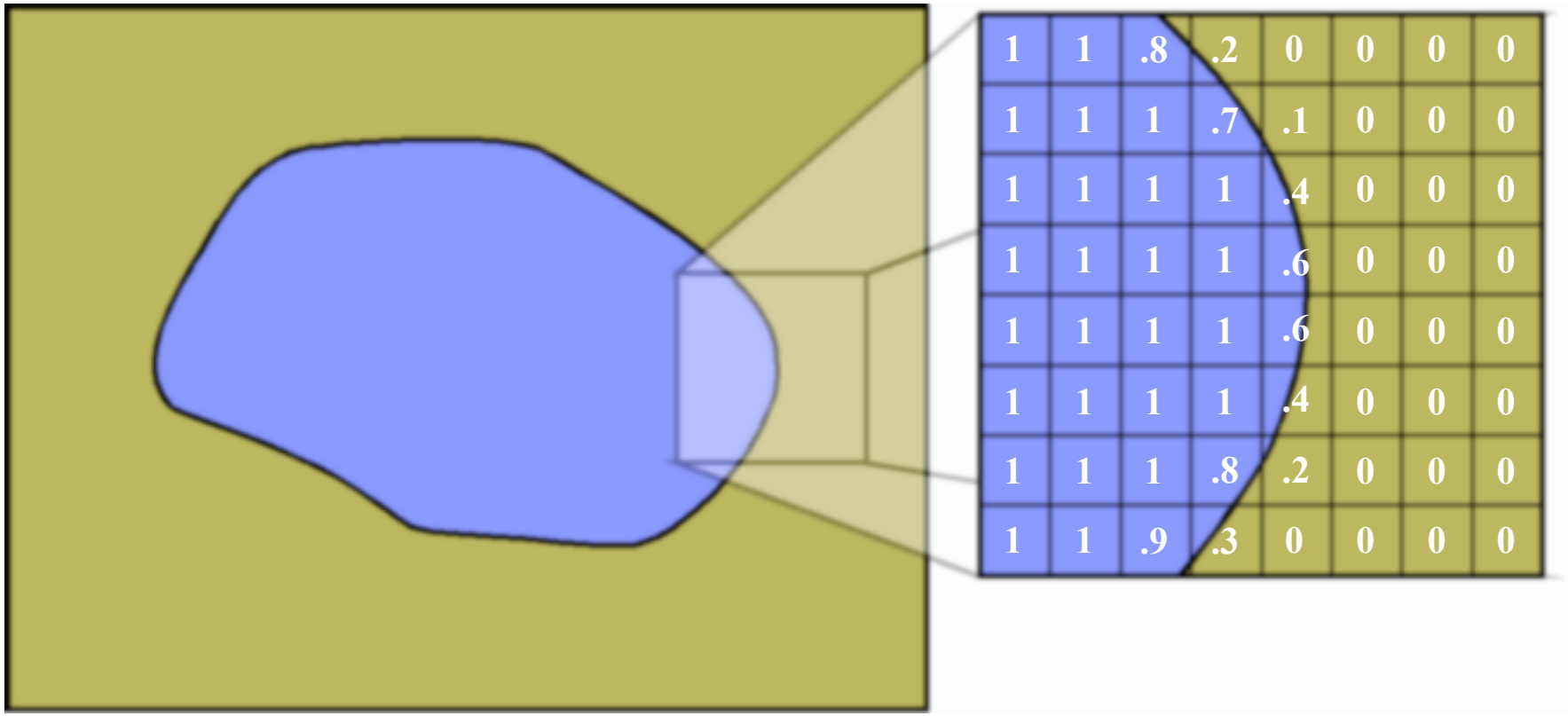
Partial Pixel Coverage

Problem: Boundary neither foreground nor background



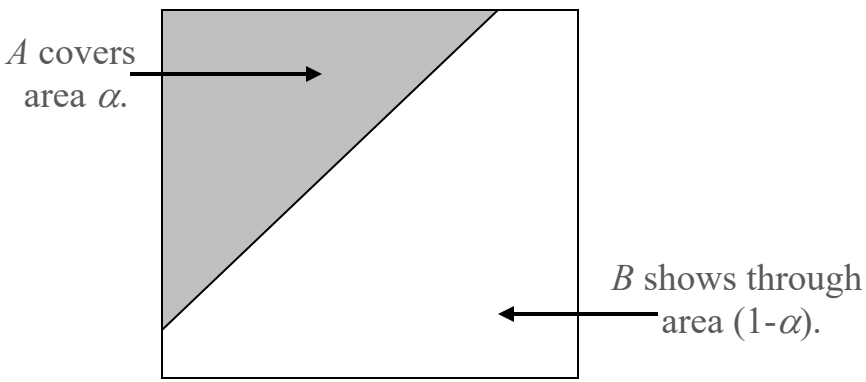
Partial Pixel Coverage

Solution: Interpolate on the border (Not exact, but *fast*)



Alpha Compositing

- Formalized in 1984 by Porter & Duff
- **Store fraction of pixel covered**; call it α



$$C = A \textbf{ over } B$$

$$r_C = \alpha_A r_A + (1 - \alpha_A) r_B$$

$$g_C = \alpha_A g_A + (1 - \alpha_A) g_B$$

$$b_C = \alpha_A b_A + (1 - \alpha_A) b_B$$

- Clean implementation; 8 more bits makes 32
  - 2 multiplies + 1 add for compositing

Alpha Compositing Example

- Repeat previous with grey scale mask
  - Edges are much better now

[Chuang et al/ Corel] [Cornell PCG]

Alpha Compositing Example

- Repeat previous with grey scale mask
  - Edges are much better now

[Chuang et al/ Corel] [Cornell PCG]

Summary

- Drawing is all about coordinate systems
  - Object coords: Coordinates of pixels in image file
  - Screen coords: Coordinates of screen pixels
- Transforms alter coordinate systems
  - "Multiply" image by matrix to distort it
  - Multiply transforms together to combine them
- Sprites combined via **color compositing**
  - Have three visible channels: red, green, blue
  - Alpha = percentage color in foreground