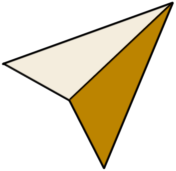


COMP4610/COMP6461

Week 3 - 2D Transformations and Hierarchical Modeling

<Print version>

[Labs]

First Lab is due at the end of this week. Remember...

- It's due 5pm Friday.
- Clarification on 'nearly perfect solution'.
- **Commit** and **push**
- **Add** me as developer (u6857890)
- (working on script to automate this for future labs...)

[Q&A]

- Same time, but now moved to my office.
- We'll be doing an online Q&A on Piazza.

[Assignment 1]

- This assignment is to be completed individually.
- Details are up on the course website.
- The assignment is due end of Week-6.

Transformations

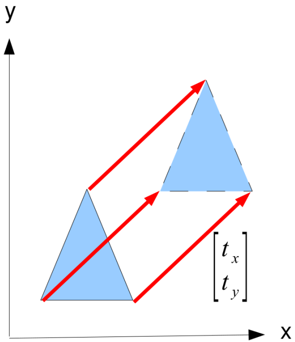
2D Transformations

Transformations are useful for modelling and viewing a scene.

- They can be used to *construct* complex objects, and also to *move* those objects around in a scene.
- Transformations are often combined using a hierarchy. (e.g rotating an arm should also move the hand, but rotating the hand should not rotate the arm.)
- Such transformations are useful and powerful tools for computer graphics applications, however, at times they can be tricky to get working properly.

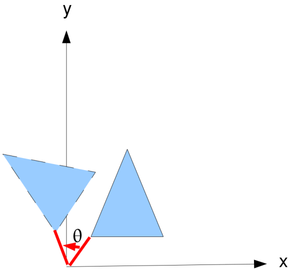
Translation

$$P' = P + \begin{bmatrix} t_x \\ t_y \end{bmatrix}.$$



Rotation (about origin)

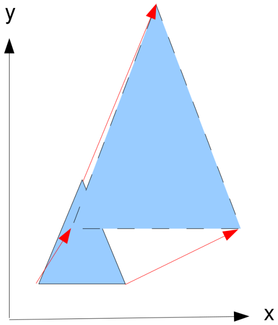
$$P' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} P$$



What if we want to rotate around a point other than the origin?

Scaling

$$P' = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} P$$



Homogeneous Coordinates

By expanding to a 3x3 matrix we can combine all these transformations into a *single* matrix multiplication.

- The Cartesian point (x, y) is represented by the homogeneous coordinate (wx, wy, w), where w is often set to 1.0 so that (x, y) is represented by $(x, y, 1)$.
- Also, when $w \neq 0$ the homogeneous coordinate (x, y, w) represents the Cartesian point $(x/w, y/w)$. A single *point* in Cartesian space is represented by a *line* in homogeneous space. **[Why might this be useful?]**

Homogeneous Coordinates

Using Homogeneous coordinates, all of the transformations we have looked at can be represented by a single matrix:

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Advantages of Homogeneous Coordinates

- One common way to apply all transformations.
- We can now combine transformations (and invert them too).
- Will allow for projections later on...
- Allows **points** and **vectors** to be represented together in one space.

Composite Transformations

- Any sequence of (linear) transformations is also a linear transformation. Therefore they can be combined into a *single* transformation matrix.
- The *order* of transformations within the sequence (usually) matters.
- Rules for composition are as expected:
  - $T(a,b)T(c,d) = T(a+c, b+d)$
  - $R(\theta)R(\phi) = R(\theta+\phi)$
  - $S(a,b)S(c,d) = S(ac, bd)$

Transformations

Other transformations

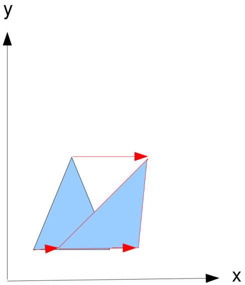
Reflection

- Reflection around the x-axis: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- Reflection around y-axis $\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- Reflection about both (same as $R(180^\circ)$) $\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

[How would you reflect around an arbitrary axis?]

Shear (uncommon)

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



Identity

Does not change the input.

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

Graphics2D

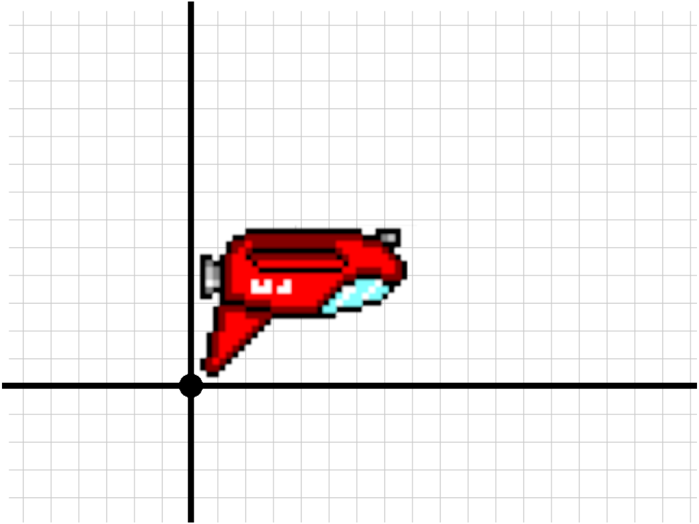
Graphics2D

- Java Graphics2D has an **AffineTransform** class that allows us to perform operations
- The bottom row is assumed to be $0, 0, 1$ so matrices are $3x2$ instead of $3x3$.
- How this works:
  - Graphics 2D keeps track of the current transformation matrix
  - User can modify it by applying transformations
  - This is very similar to how **OpenGL** works (later in the course).

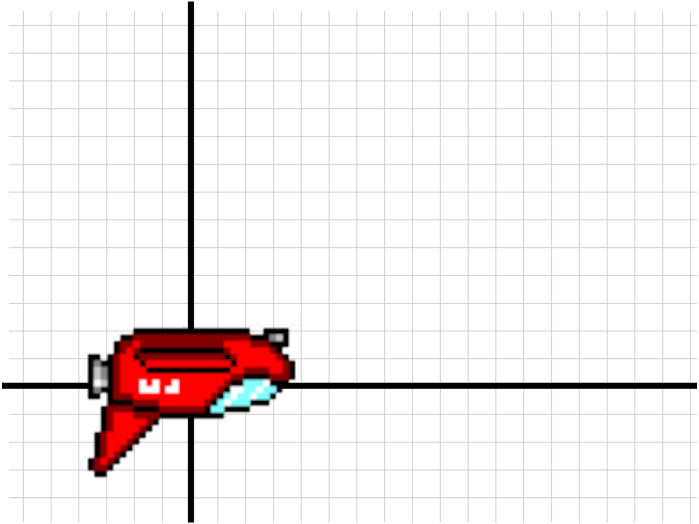
Graphics2D

Example

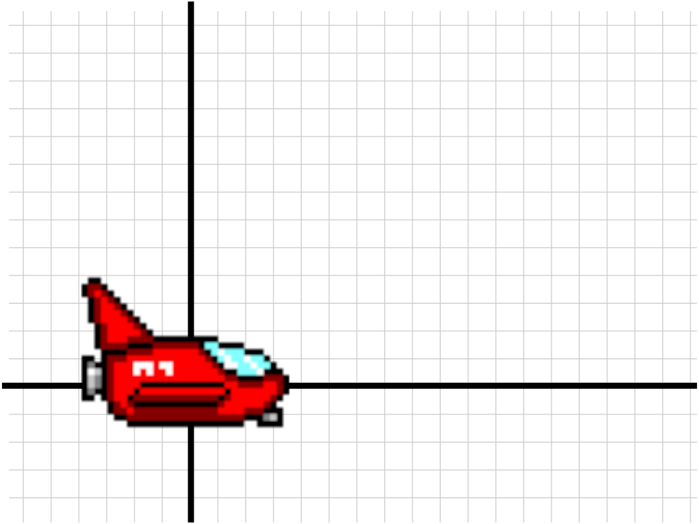
g.drawImage(...)



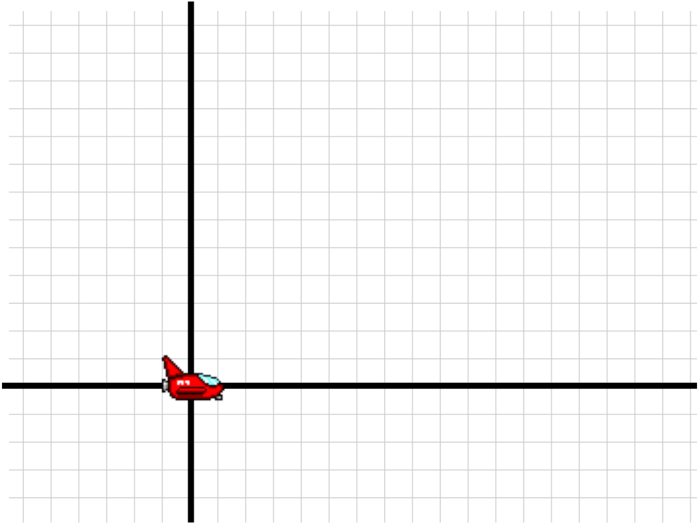
g.translate(iw/2, ih/2)



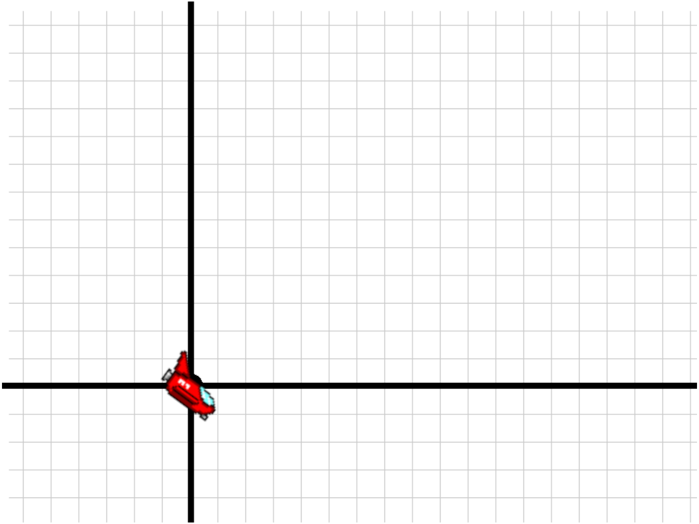
g.scale(1, -1)



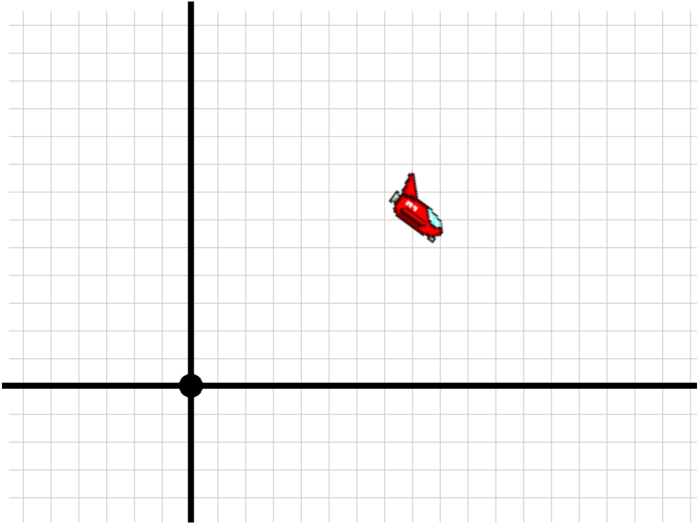
g.scale(0.1, 0.1)



g.rotate(-0.4)



g.translate(100,100)



Aspect Ratio

Aspect Ratio

- The aspect ratio of a device/image is the ratio between the width and the height of that device/image. This ratio is commonly given by two numbers the width and then the height (with a colon in between). Note that the width and height are often simplified and only describe the ratio not the actual length, or number of pixels. For example $1920 : 1080 \rightarrow 16 : 9$.
- Modern devices generally have square pixels. This makes drawing shapes like circles and squares simpler (if this is not the case then you need to add appropriate scaling transformation).

Aspect Ratio

Applications will often have to draw to different device or window aspect ratios. There are several approaches you can use to deal with this, including:

- Draw in *device coordinates*.
- Draw in a *fixed user coordinate* system, which you transform onto the device.
- Draw in a user coordinate system but have different approaches for different aspect ratios.

Aspect Ratio

Suppose you wish to draw in a user coordinate system that has $(0, 0)$ at the centre of the screen, positive y going up, $(-10, 10)$ as the top left coordinate and $(10, -10)$ the bottom right. Now suppose the device you are drawing to is 640x480 (aspect 4:3), with g being the Graphics2D object you are using for drawing. Then you could use the following transformations:

```
g.scale(640.0/20.0, 480.0/20.0);
g.translate(10.0,10.0);
g.scale(1.0,-1.0);
```

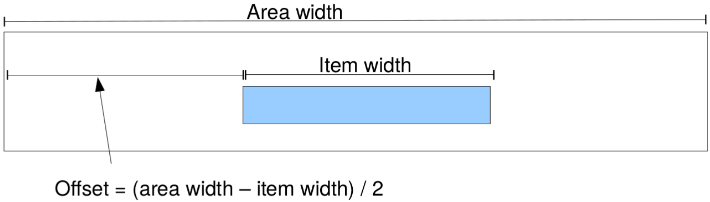
The problem with the above approach is it will squash what you draw. Another approach would be to scale the user coordinate area such that it only uses a part of the screen. This could be done with:

```
g.translate((640.0-480.0)/2.0,0.0);
g.scale(480.0/20.0, 480.0/20.0);
g.translate(10.0,10.0);
g.scale(1.0,-1.0);
```

Centering

Centering and Spacing Items

To position an item you are drawing within the center of a scene you can do some simple math to calculate the offset from the side.



If you have k items you wish to space evenly within an area then the space between these items will be:

$$\frac{\text{width} - \sum_i = 1^k w_i}{k+1}$$

Hierarchical Modeling

Hierarchical Modeling

Model the structure of your seen through function calls.

- Each method modifies the current transform, then restores it once it's done.
- Every method assumes it's being draw in it's 'natural' co-ordinate system.
- Transforms applied at parent level, automatically transferred to children.
- Components can be reused (i.e. a wheel on a bike or car).

Issues

- Structure is not explicit.
- Transformations are tied to drawing.
- Each method *must* clean up the transform.

Hierarchical Modeling: Code example

```java
private void drawCar(Graphics2D g, double x, double y) {
    AffineTransform af = g.getTransform();
    g.translate(x, y)
    drawBody(g);
    g.translate(-1.0, 1.0)
    drawWheel(g); // drawn at (-1, 1)
    g.translate(2.0, 0)
    drawWheel(g); // drawn at (+1, 1)
    g.setTransform(af);
}
```

Hierarchical Modeling

- Care needs to be taken such that any transformations that an object uses for drawing itself are undone. Otherwise, it becomes difficult to position subsequent objects properly.
- One approach that may be used is to...
    - have a method for each object,
    - objects are drawn at coordinates centred on (0,0),
    - at the beginning of the method, the current transformation is stored (pushed) and then restored (popped) before the method returns.

Inverting Affine Transformations

Inverting Affine Transformations

As we draw objects to the scene coordinates are transformed from **user** coordinates to **device** coordinates. However, if a user interacts with our drawing (via a device such as a touch screen or a mouse). Then the coordinate our program obtains are **device** coordinates, thus, if you wish to interact with objects in **user** coordinates then you need to transform these **device** coordinates by inverting the **user** $\rightarrow$ **device** transformation.

Java's AffineTransform

- In Java the **AffineTransform** class provides a method that will invert an affine transform. There are also methods that will apply an inverted transformation to individual points.
- Inverted affine transforms are also affine transforms. Note: some affine transformations can not be inverted. **[which ones?]**.

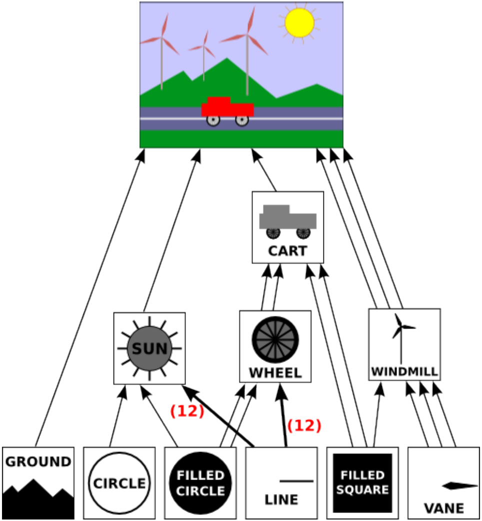
Scene Graph

Scene Graph

Another (better) approach is to use a **scene graph**.

- Nodes in a scene graph have attributes, transformations, and a method for drawing that node.
- Nodes can have a list of children, who inherit their transform.
- With this approach converting from **device** $\rightarrow$ **user** coordinates can be done *without* executing the drawing code, rather, the transformations in the scene graph can be traversed and inverted.
- We can easily traverse up (and down) the graph to calculate a a 'toLocal' matrix transform, and it's inverse 'toWorld'.

Scene Graph


CART
SUN
WHEEL
WINDMILL
(12)
(12)
GROUND
CIRCLE
FILLED
CIRCLE
LINE
FILLED
SQUARE
VANE

[Course Representatives]

Roles and responsibilities:

- Act as the official liaison between your peers and convener.
- Be creative, available and proactive in gathering feedback from your classmates.
- Attend regular meetings, and provide reports on course feedback to your course convener and the Associate Director (Education).
- Close the feedback loop by reporting back to the class the outcomes of your meetings.

Sign up here -> https://anu.au1.qualtrics.com/jfe/form/SV_3L5pd93Aq9k21iS