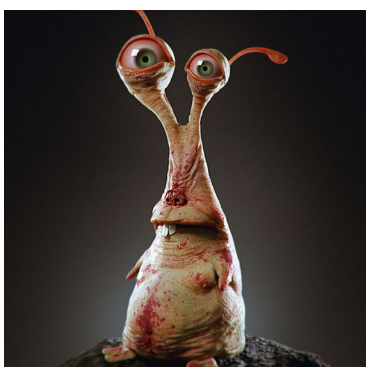
3D transformations
(deel 2)

3D Computer Graphics (Lab 8)



Applying multiple transformations to one shape

Remember …

- Assume you want to apply three different transformations.

$$A \xrightarrow{T_1} A' \xrightarrow{T_2} A'' \xrightarrow{T_3} A'''$$

- Instead of applying these three matrices T_1, T_2 en T_3 to each point, it is more efficient to compute the so called composite matrix T_c once

$$T_c = T_3.T_2.T_1$$

and apply T_c to each point $T_c.A = A'''$

Computing the composite matrix T_c requires multiplying the individual transformation matrices in the order opposite to the order in which they are applied!

In which order do we have to specify the transformations in the sdl file?
In the order in which they should be applied to the object? Or in the order in which they should be multiplied to get the composite matrix?

Conventions

1) The transformations in the sdl file should be specified in the order in which they should be multiplied to get T_c.

   So in case one wants to apply T_1, T_2 en T_3 (in that order) to a square, one has to specify in the sdl file:

   T3
   T2
   T1
   square

2) The SceneFactory class will keep track of the current composite matrix while parsing the sdl file. When it encounters a new transformation T_{new} in the sdl file, the current composite matrix T_c is updated by postmultiplying it with the new transformation.

$$T_c = T_c.T_{new}$$

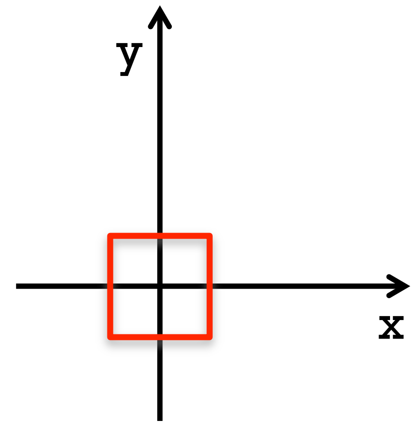
Computation of T_c

sdl file	SceneFactory
	$T_c = I$
T_3	$T_c = T_c.T_3 \quad = I.T_3 = T_3$
T_2	$T_c = T_c.T_2 \qquad = T_3.T_2$
T_1	$T_c = T_c.T_1 \qquad = T_3.T_2.T_1$

Adopting the two conventions of the previous slide gives the desired result!

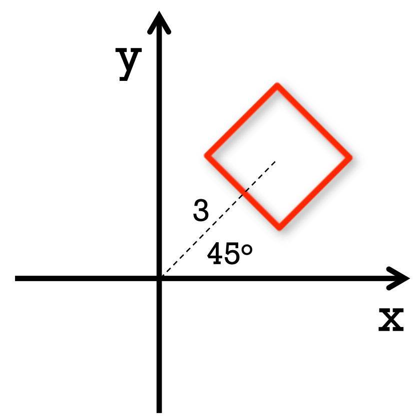
Example

- Specifying a square command without any transformation in the sdl file results in a square centered aroud the origin with sides equal to 2.

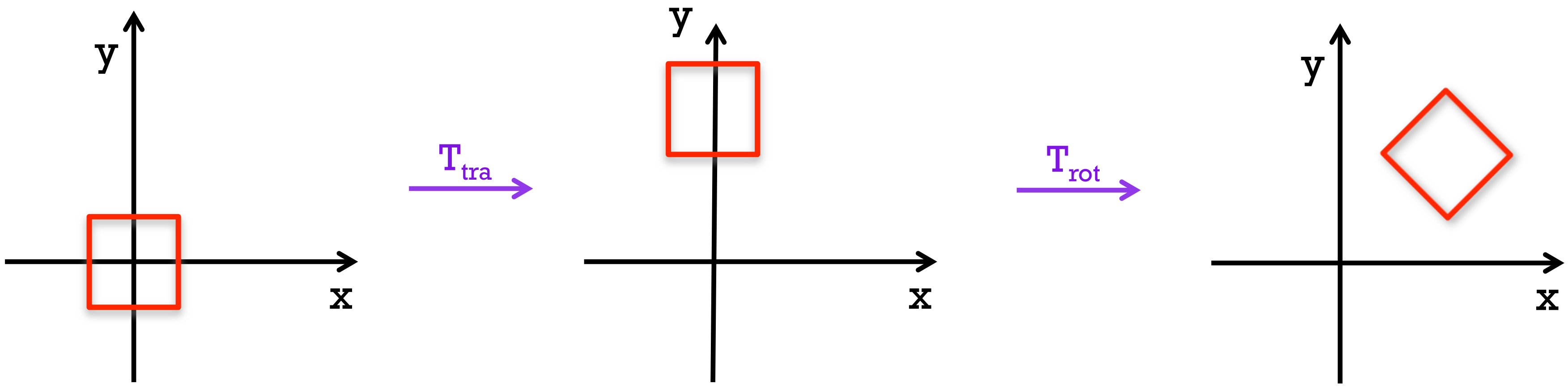


- But assume that you want to draw the square as shown in this figure:

  Which transformations do you have to apply?



Example

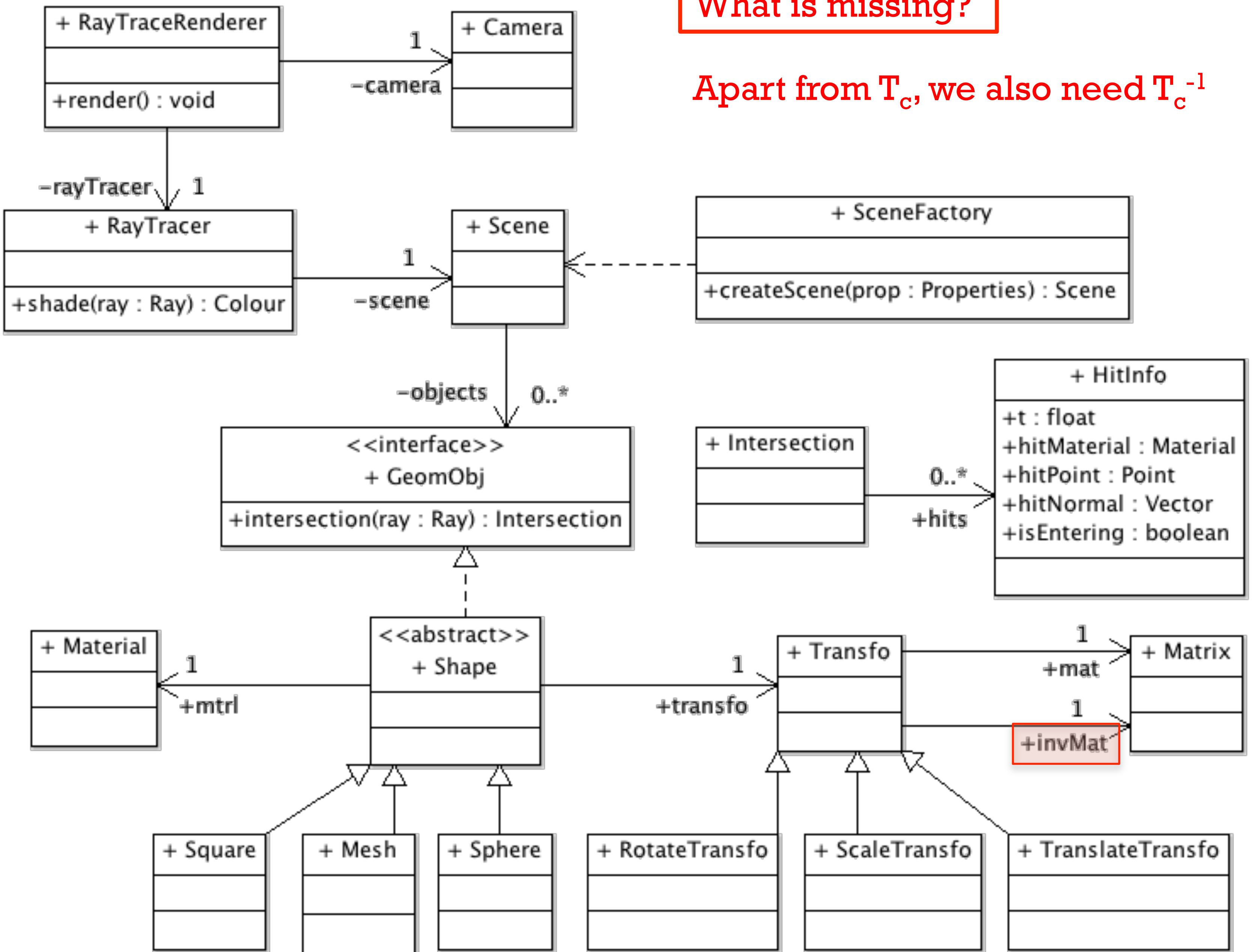


So your sdl file would look like

rotate -45 0 0 1
translate 0 3 0
square

What is missing?

Apart from T_c, we also need T_c^{-1}



The inverse of T_c

- Assume you want to apply three different transformations.

$$A \xrightarrow{T_1} A' \xrightarrow{T_2} A'' \xrightarrow{T_3} A'''$$

- $T_c = T_3.T_2.T_1$
- Property: $\boxed{\text{if} \quad T_c = T_3.T_2.T_1 \quad \text{then} \quad T_c^{-1} = T_1^{-1}.T_2^{-1}.T_3^{-1}}$

Proof:

$$\begin{aligned} T_c.T_c^{-1} &= (T_3.T_2.T_1)(T_1^{-1}.T_2^{-1}.T_3^{-1}) \\ &= T_3.T_2.(T_1.T_1^{-1}).T_2^{-1}.T_3^{-1} \\ &= T_3.T_2.T_2^{-1}.T_3^{-1} \\ &= T_3.(T_2.T_2^{-1}).T_3^{-1} \\ &= T_3.T_3^{-1} \\ &= I \end{aligned}$$

Conventions

1) The transformations in the sdl file should be specified in the order in which they should be multiplied to get T_c.

   So in case one wants to apply T_1, T_2 en T_3 (in that order) to a square, one has to specify in the sdl file:

   T3
   T2
   T1
   square

2) The SceneFactory class will keep track of the current composite matrix while parsing the sdl file. When it encounters a new transformation T_{new} in the sdl file, the current composite matrix T_c is updated by postmultiplying it with the new transformation.

$$T_c = T_c.T_{new}$$

3) The SceneFactory class will keep track of the inverse of the current composite matrix while parsing the sdl file. When it encounters a new transformation T_{new} in the sdl file, the inverse of the current composite matrix T_c is updated by premultiplying it with the inverse of the new transformation.

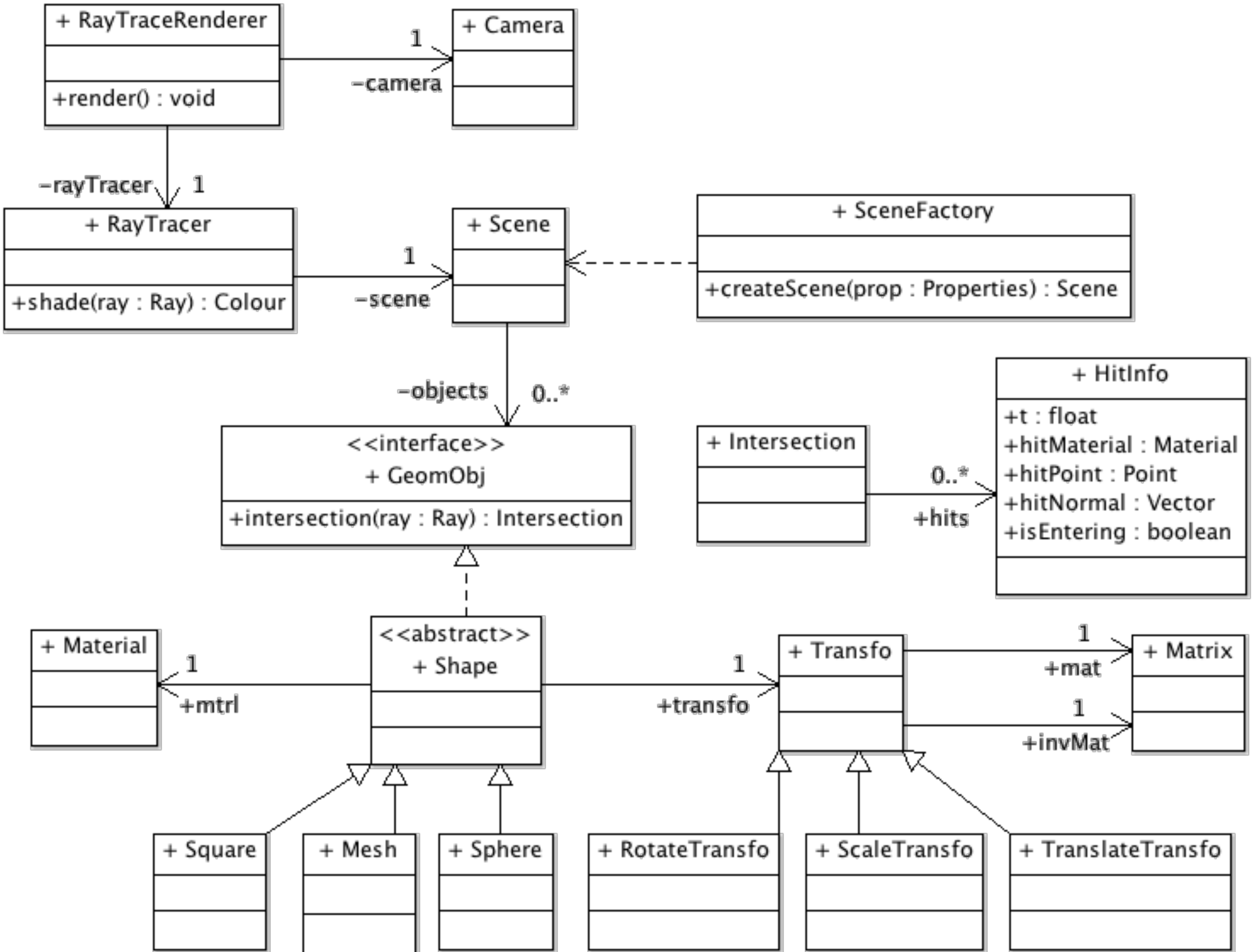
$$T_c^{-1} = T_{new}^{-1}.T_c^{-1}$$

Computation of T_c

sdl file | SceneFactory

sdl file	SceneFactory		
	$T_c = I$	$T_c^{-1} = I$	
T_3	$T_c = T_3$	$T_c^{-1} = T_3^{-1}. T_c^{-1}$	$= T_3^{-1}. I = T_3^{-1}$
T_2	$T_c = T_3.T_2$	$T_c^{-1} = T_2^{-1}.T_c^{-1}$	$= T_2^{-1}.T_3^{-1}$
T_1	$T_c = T_3.T_2.T_1$	$T_c^{-1} = T_1^{-1}.T_c^{-1}$	$= T_1^{-1}.T_2^{-1}.T_3^{-1}$

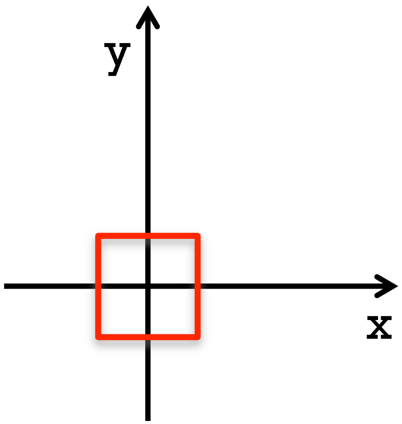
Adopting the three conventions of the previous slide gives the desired result!


+ RayTraceRenderer
+render() : void
+ Camera
1
−camera
−rayTracer
1
+ RayTracer
+shade(ray : Ray) : Colour
+ Scene
1
−scene
+ SceneFactory
+createScene(prop : Properties) : Scene
−objects
0..*
<<interface>>
+ GeomObj
+intersection(ray : Ray) : Intersection
+ Intersection
0..*
+hits
+ HitInfo
+t : float
+hitMaterial : Material
+hitPoint : Point
+hitNormal : Vector
+isEntering : boolean
+ Material
1
+mtrl
<<abstract>>
+ Shape
1
+transfo
+ Transfo
1
+mat
1
+invMat
+ Matrix
+ Square
+ Mesh
+ Sphere
+ RotateTransfo
+ ScaleTransfo
+ TranslateTransfo

Applying multiple transformations to multiple shapes

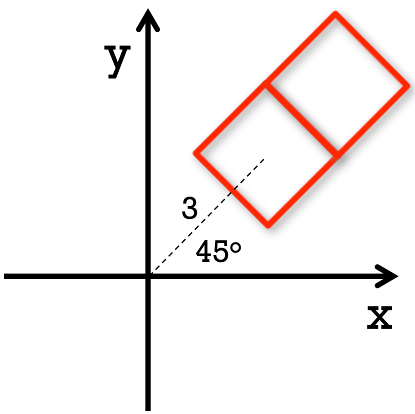
Example 2

- Specifying a square command without any transformation in the sdl file results in a square centered aroud the origin with sides equal to 2.



- But you want to draw a scene as shown in this figure:

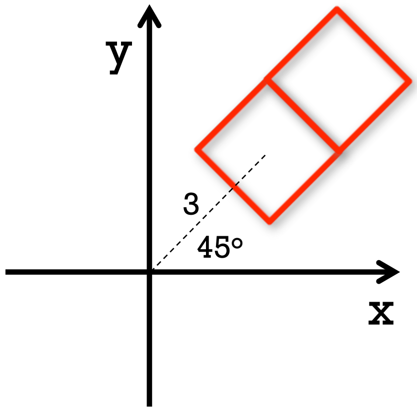
  Which commands do you have to specify in the sdl file?



Example 2

Let's assume T_c is reset to the unit matrix after an object command is given in the sdl file.

```
rotate -45 0 0 1
translate 0 3 0
square
rotate -45 0 0 1
translate 0 5 0
square
```

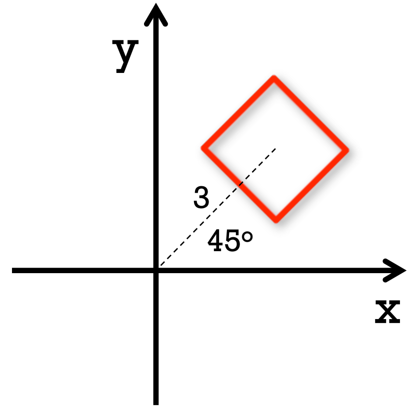


This method is highly inefficient because we have to start from scratch to transform every object in the scene.

There is a better way but it requires looking at the problem in another way.

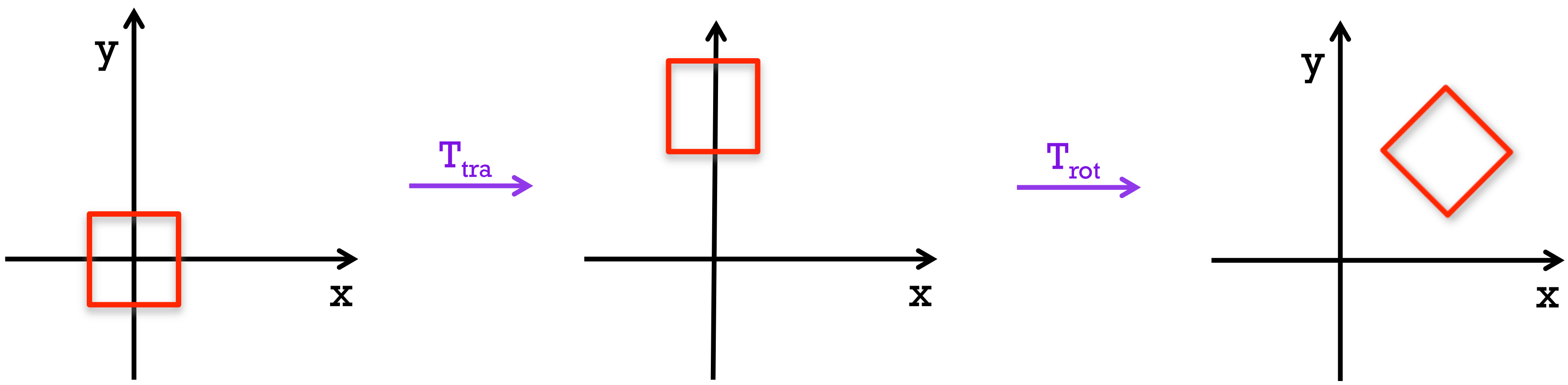
Remember example 1

- We found out that one can draw this figure by means of the following commands




```
rotate -45 0 0 1
translate 0 3 0
square
```

- We arrived at this result by reasoning how the original square has to be transformed with respect to the given coordinate system and by writing these transformations down in reverse order.



Idea

- Previous reasoning:

  How does the object has to be transformed with respect to the given coordinate system to get the desired result?

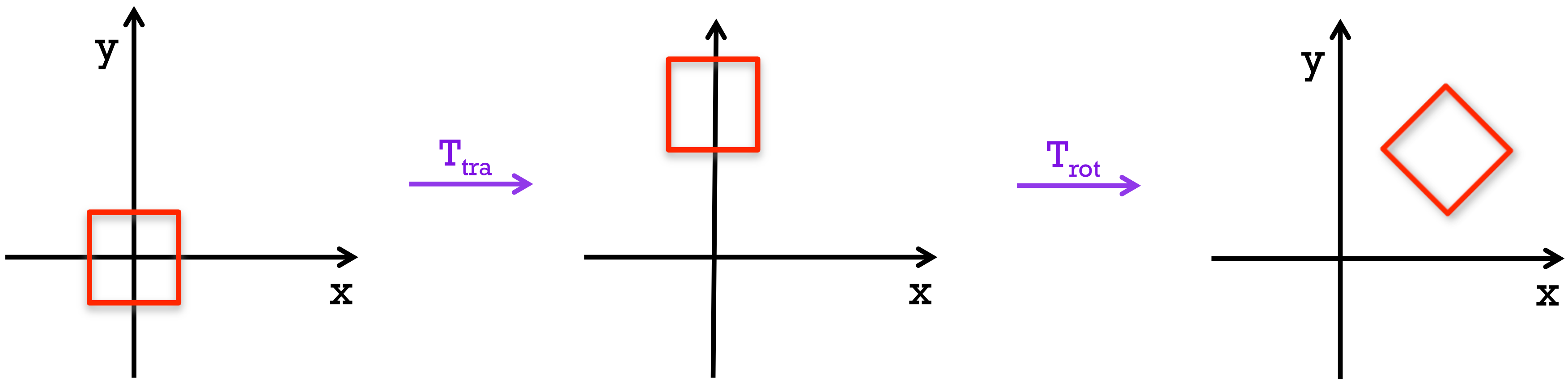
- New reasoning:

  How does the coordinate system has to be transformed with respect to the previous coordinate system to get the desired result?

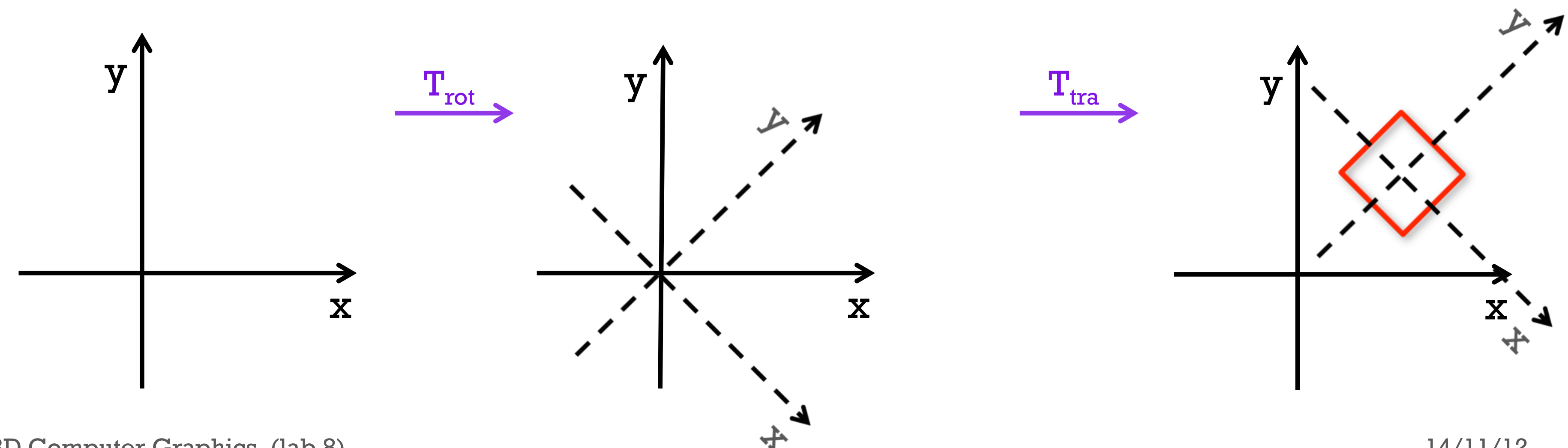
Example 1


```
rotate -45 0 0 1
translate 0 3 0
square
```

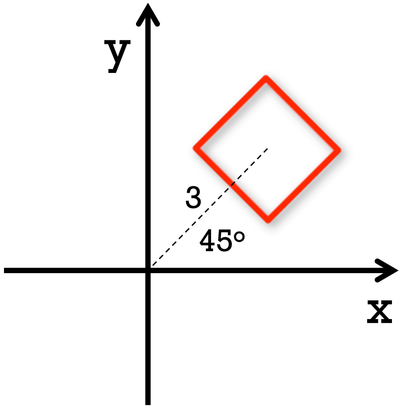
Previous reasoning



New reasoning



Summary



- We found out that one can draw this figure by means of the following commands

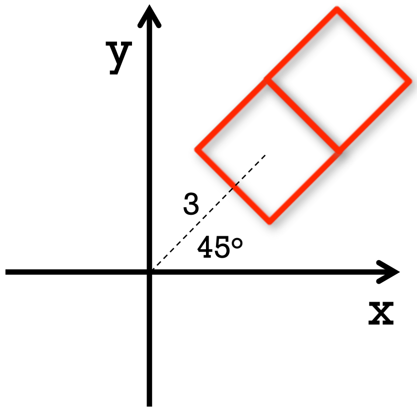
```
rotate -45 0 0 1
translate 0 3 0
square
```

- We arrived at this result by reasoning how the original square has to be transformed with respect to the given coordinate system and by writing these transformations down in reverse order.
- Alternatively, one could reason how the coordinate system has to be transformed with respect to the previous coordinate system and writing these transformations down in the same order.

Both reasonings lead to the same code and hence, to the same result!

Use the alternative reasoning!

Example 2



Let's assume T_c is reset to the unit matrix as soon as one object is drawn.

```
rotate -45 0 0 1
translate 0 3 0
square
rotate -45 0 0 1
translate 0 5 0
square
```

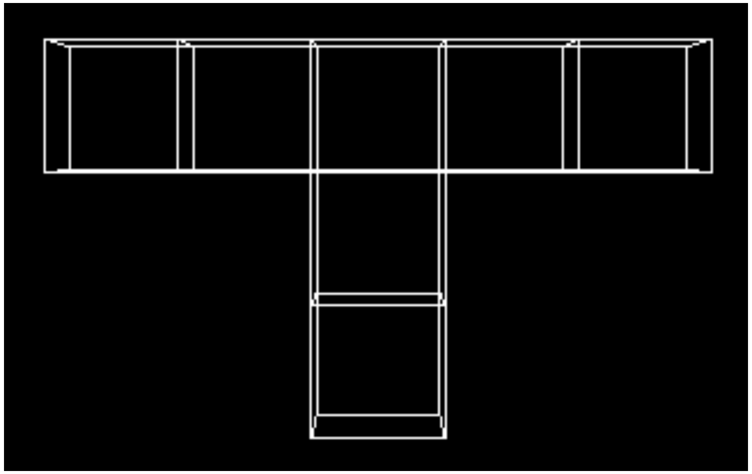
This method is highly inefficient because we have to start from scratch to transform every object in the scene.

A better way by using the alternative reasoning:

```
rotate -45 0 0 1
translate 0 3 0
square
translate 0 2 0
square
```

Example 3

- Assume we have a cube.txt file containing the polygonal mesh data of a unit cube centered around the origin.
- What commands do we have to specify in the sdl file to model the following figure?



- You may assume that the cube at the base of the figure is centered around the origin.

Example 3

mesh cube.txt

translate 0 1 0

mesh cube.txt

translate 0 1 0

mesh cube.txt

translate -1 0 0

mesh cube.txt

translate -1 0 0

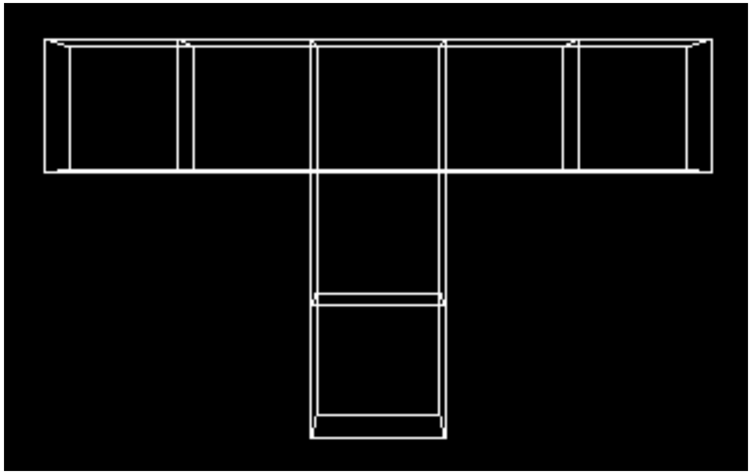
mesh cube.txt

translate 3 0 0

mesh cube.txt

translate 1 0 0

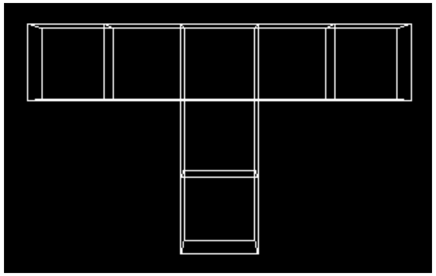
mesh cube.txt



One issue: you need to compute which transformation has to be carried out to go back to a previous location.

We want to avoid this by implementing a mechanism which allows to return to a previous location.

Example 3



mesh cube.txt
translate 0 1 0
mesh cube.txt
translate 0 1 0
mesh cube.txt

translate -1 0 0
mesh cube.txt
translate -1 0 0
mesh cube.txt

translate 3 0 0
mesh cube.txt
translate 1 0 0
mesh cube.txt

Better →

mesh cube.txt
translate 0 1 0
mesh cube.txt
translate 0 1 0
mesh cube.txt
push
translate -1 0 0
mesh cube.txt
translate -1 0 0
mesh cube.txt
pop
translate 1 0 0
mesh cube.txt
translate 1 0 0
mesh cube.txt

Save the current composite transformation

Restore the last saved composite transformation.

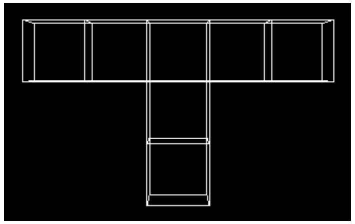
How will we implement this push and pop functionality?

Stack implementation

- We will use a stack implementation.
- Each stack element consists of a Transfo object which contains two 4x4-matrices: mat and invMat.
- The top of the stack represents the current composite matrix T_c (and its inverse).
- When the SceneFactory class encounters a new transformation T_{new} in the sdl file, the current composite matrix T_c is updated by postmultiplying it with the new transformation:

$$T_c = T_c . T_{new}$$

In the stack implementation, this means that the top of the stack is postmultiplied with T_{new}.



mesh cube.txt
translate 0 1 0
mesh cube.txt
translate 0 1 0
mesh cube.txt
push
translate -1 0 0
mesh cube.txt
translate -1 0 0
mesh cube.txt
pop
translate 1 0 0
mesh cube.txt
translate 1 0 0
mesh cube.txt

Stack:

mat: Identity matrix

Stack:

mat: Translation (0,1,0)

Stack:

mat: Translation (0,2,0)

Stack:

mat: Translation (0,2,0)
mat: Translation (0,2,0)

Stack:

mat: Translation (-1,2,0)
mat: Translation (0,2,0)

Stack:

mat: Translation (-2,2,0)
mat: Translation (0,2,0)

Stack:

mat: Translation (0,2,0)

…

Questions?