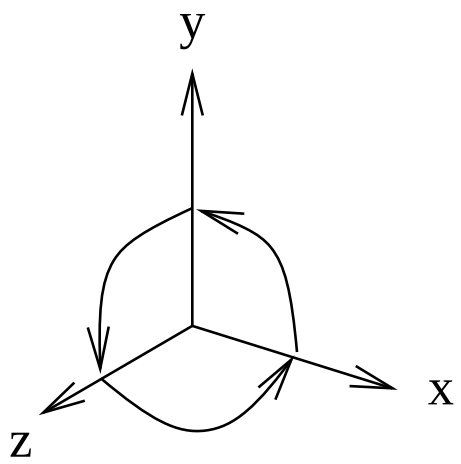
Orientation of the coordinate system

Right-handed coordinate system:

- The x-axis is mapped to the y-axis by a positive, i.e., anticlockwise, rotation of $90°$ around the z-axis.
- The y-axis is mapped to the z-axis by a positive rotation of $90°$ around the z-axis.
- The z-axis is mapped to the x-axis by a positive rotation of $90°$ around the y-axis.



Orientation of the coordinate system

Using the thumb of the right hand for the x-axis, the forefinger for the y-axis and the middle finger for the z-axis, one obtains the correct orientation of the coordinate system.

A rotation by a positive angle around an oriented axis in the three-dimensional space refers to an anticlockwise rotation for a viewer to whom the axis points.

Right-hand rule: When the thumb of the right hand points in the same direction as the rotation axis and the other fingers form a fist, then the bent fingers indicate the direction of positive rotation.

Homogeneous coordinates

$(\tilde{x}, \tilde{y}, \tilde{z}, w)$ where $w \neq 0$

represents the point

$$\left(\frac{\tilde{x}}{w}, \frac{\tilde{y}}{w}, \frac{\tilde{z}}{w}\right) \in \mathbb{R}^3.$$

$(x, y, z) \in \mathbb{R}^3$ is represented by $(x, y, z, 1)$ or, more generally, by $(x \cdot w, y \cdot w, z \cdot w, w)$ where $w \neq 0$.

Geometric transformations

Translation by the vector $(d_x, d_y, d_z)^\top$:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{pmatrix}$$

Translation matrix: $$T(d_x, d_y, d_z) = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Geometric transformations

Scaling by the factors s_x, s_y, s_z:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \\ s_z \cdot z \\ 1 \end{pmatrix}$$

Scaling matrix: $S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Geometric transformations

Rotation around the z-axis by the angle θ:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotation matrix: $$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Geometric transformations

Rotation around the x-axis by the angle θ:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotation matrix: $$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Geometric transformations

Rotation around the y-axis by the angle θ:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotation matrix: $$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Geometric transformations

Rotation around an arbitrary axis by the angle θ:

- Shift the rotation by a translation such that it passed through the origin.
- Rotation around the z-axis, such that the rotation axis is mapped to the y/z-plane.
- Rotation around the x-axis, such that the rotation axis is mapped to the z-axis.
- Rotation by the angle θ around the z-axis.
- Reverse the three first transformations.

$$T(-d_x, -d_y, -d_z) \circ R_z(-\theta_z) \circ R_x(-\theta_x) \circ R_z(\theta) \circ R_x(\theta_x) \circ R_z(\theta_z) \circ T(d_x, d_y, d_z)$$

Geometric transformations

As already in the case of 2D graphics, the composition of transformations can be implemented by matrix multiplication.

The last line for all above mentioned matrices is $(0, 0, 0, 1)$. Matrix multiplication preserves this property.

In the two-dimensional case there is exactly one transformation matrix that maps three noncollinear points to three other noncollinear points.

In the three-dimensional case there exists exactly one transformation matrix that maps four noncoplanar points to four other noncoplanar points.

Geometric transformations

Given four noncoplanar points $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4 \in \mathbb{R}^3$ and the target points $\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3, \mathbf{p}'_4$, the transformation matrix is obtained by solving the system of linear equations

$$\mathbf{p}'_i = M \cdot \mathbf{p}_i \qquad (i = 1, 2, 3, 4)$$

(in homogeneous coordinates) where

Geometric transformations

$$M = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

In this sense, transformations can be interpreted as changing from one coordinate system to another.

Transformations in Java 3D

The class `Transform3D` stores 3D transformations as matrices (in homogeneous coordinates), analogous to the 2D case.

- `Transform3D tf = new Transform3D();` generates the identity transformation corresponding to the unit matrix.
- `tf.rotX(theta);` defines `tf` as a rotation by the angle `theta` around the x-axis.
  `rotY` and `rotZ` analogously for the y- and the z-axis.
- `tf.set(new AxisAngle4d(x,y,z,theta));` defines a rotation by the angle `theta` around the axis in the direction of the `float`-vector $(x, y, z)^\top$.

Transformations in Java 3D

- A translation by the `float`-vector $(x, y, z)^\top$ is specified by

  ```
  tf.setTranslation(new Vector3f(x,y,z));
  ```

- The method

  ```
  tf.setScale(new Vector3f(x,y,z));
  ```

  leads to the scaling $S(x, y, z)$.

  `tf.setScale(factor)` defines a scaling with the same scaling factor `factor` for the x-, y- and z-direction.

Transformations in Java 3D

- Arbitrary transformations can be defined by `tf.set(matrix);`

  where `matrix` is a one-dimensional `double`-array with 16 values specifying the entries in the matrix.
- `tf.get(matrix);` stores the matrix associated with the transformation `tf` in the (one-dimensional) `double`-array `matrix`.
- Composition of transformations (matrix multiplication):
  - `tf.mul(tf1,tf2);`
  - `tf1.mul(tf2);`

Java 3D coordinate system

As long as no additional transformations are applied or the viewer's point remains unchanged,

- the x-axis in the window points to the right,
- the y-axis upward and
- the z-axis to the front.

Elementary geometric objects

For 3D objects it is not sufficient to outline their geometry only. A colour or texture must be assigned to the surface.

In addition, refection properties like shininess must be assigned to the surface.

In Java 3D the class `Appearance` is responsible for such properties of surfaces.

The class `Appearance` will be discussed in detail in connection with illumination.

Elementary geometric objects

For the first objects a very simple `Appearance` is defined by

```
Appearance myApp = new Appearance();
setToMyDefaultAppearance(myApp,
                         new Color3f(r,g,b));
```

where the three `float`-value `r,g,b` define the colour.

The `setToMyDefaultAppearance` is not a standard method in Java 3D. It was defined for the purposes here.

Elementary geometric objects

Unless mentioned otherwise, real-valued parameters are always given `float`-values.

Box: A box is generated by

```
Box xyzBox = new Box(x,y,z,myApp);
```

The box has the size $(2x) \times (2y) \times (2z)$ and is centred in the origin of the coordinate system.

Sphere:

`Sphere rSphere = new Sphere(r,myApp);` defines a sphere with radius `r` and midpoint in the origin of the coordinate system.

Elementary geometric objects

Cylinder: A cylinder with radius `r` and height h whose centre point is in the origin of the coordinate system is generated by

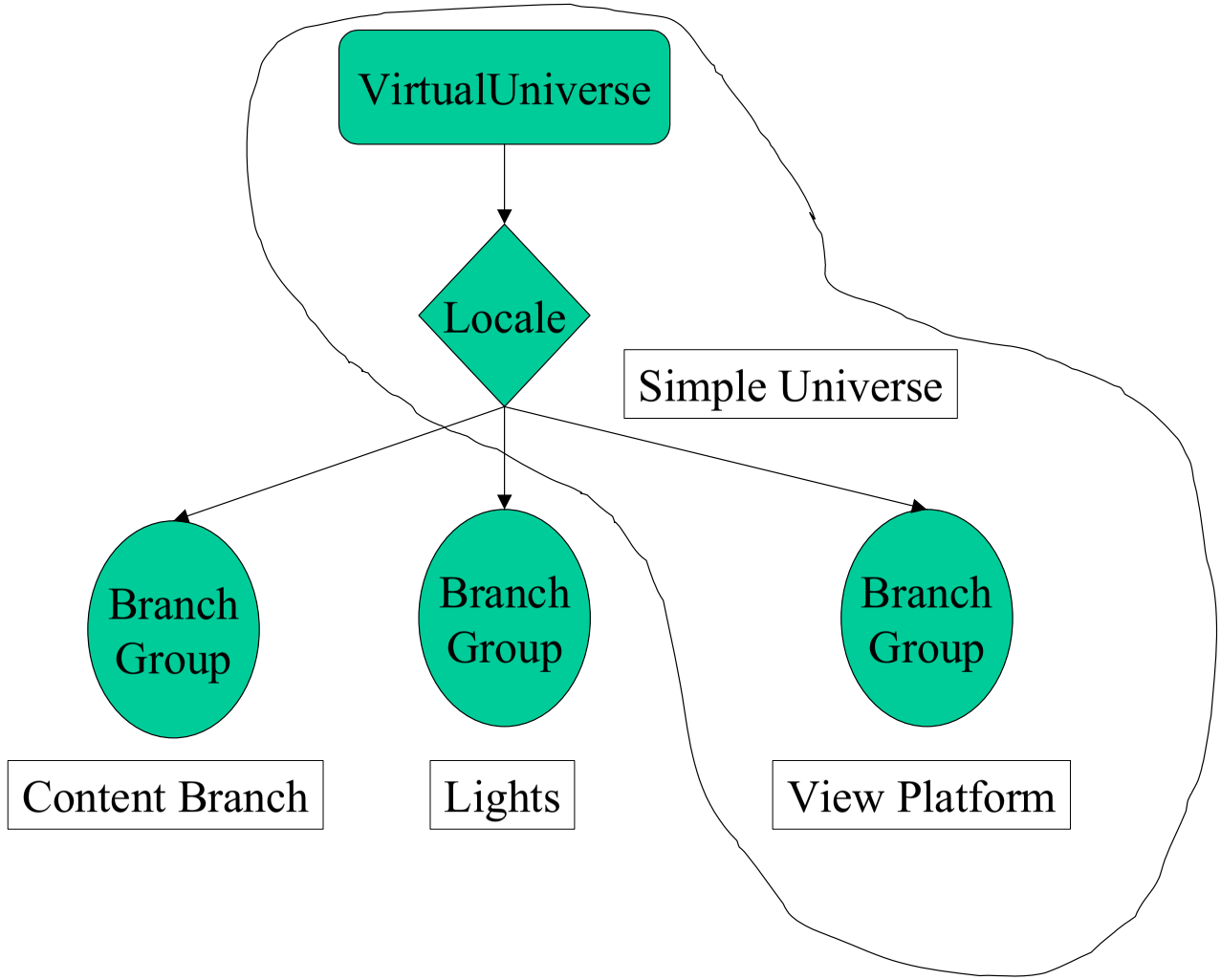
```
Cylinder rhCyl = new Cylinder(r,h,myApp);
```

Cone: A cone with radius `r` and height h is constructed by

```
Cone rhCone = new Cone(r,h,myApp);
```

The cone is positioned in the same way as the cylinder centred around the y-axis with its tip h/2 units above the x/z-plane.

The scenegraph in Java 3D



The scenegraph in Java 3D

- The geometric objects in the scene (including movements and animations) are assigned to the Content Branch.
- The View Platform (View Branch) defines viewing parameters like the position of the viewer, the direction of his view or the type of projection.
- Lights is responsible for illumination of the scene.

The `SimpleUniverse` provides a simple default View Platform automatically without the need of specifying all the parameters.

Structure of a Java 3D program

```
import ...

public class MyJava3DClass extends JFrame
{
  public Canvas3D myCanvas3D;

  public MyJava3DClass()
  {
    ...
  }

  public static void main(String[] args)
  {
     MyJava3DClass myJava3D = new MyJava3DClass();
  }
}
```

public MyJava3DClass()

```
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myCanvas3D = new Canvas3D(
        SimpleUniverse.getPreferredConfiguration());
SimpleUniverse simpUniv =
                        new SimpleUniverse(myCanvas3D);
simpUniv.getViewingPlatform(
                        ).setNominalViewingTransform();
createSceneGraph(simpUniv);
addLight(simpUniv);
setTitle("Ueberschrift");
setSize(700,700);
getContentPane().add("Center", myCanvas3D);
setVisible(true);
```

Structure of a Java 3D program

The method `createSceneGraph` must be implemented individually for each program or scene.

All geometric objects, information about their surfaces and dynamic changes can be incorporated in this method, representing the content branch.

The method `addLight` defines the illumination of the scene and it can also be implemented differently for each virtual scene.

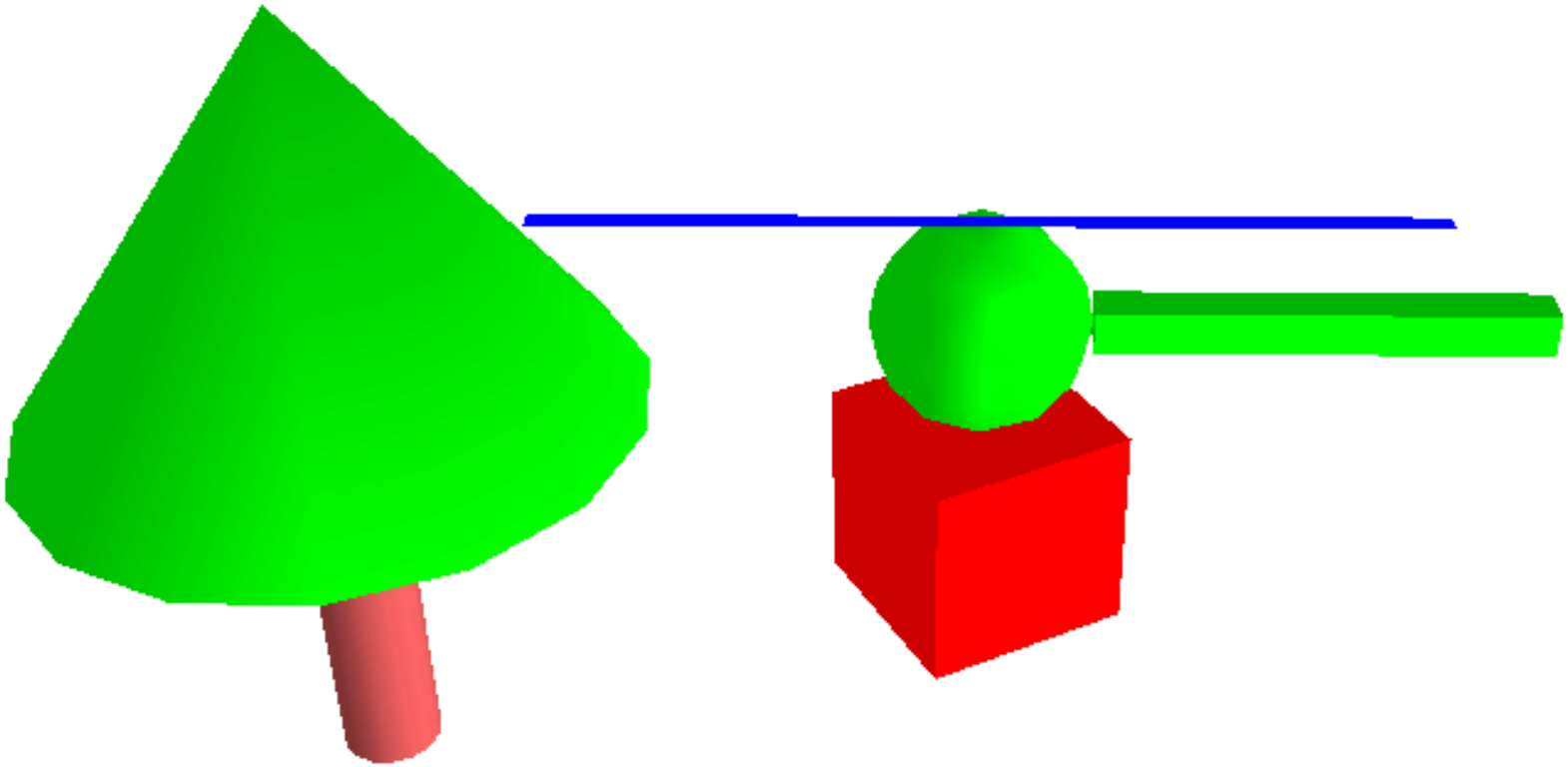
Static content branch

- The content branch is structured in the form of a tree.
- Its root is usually generated by

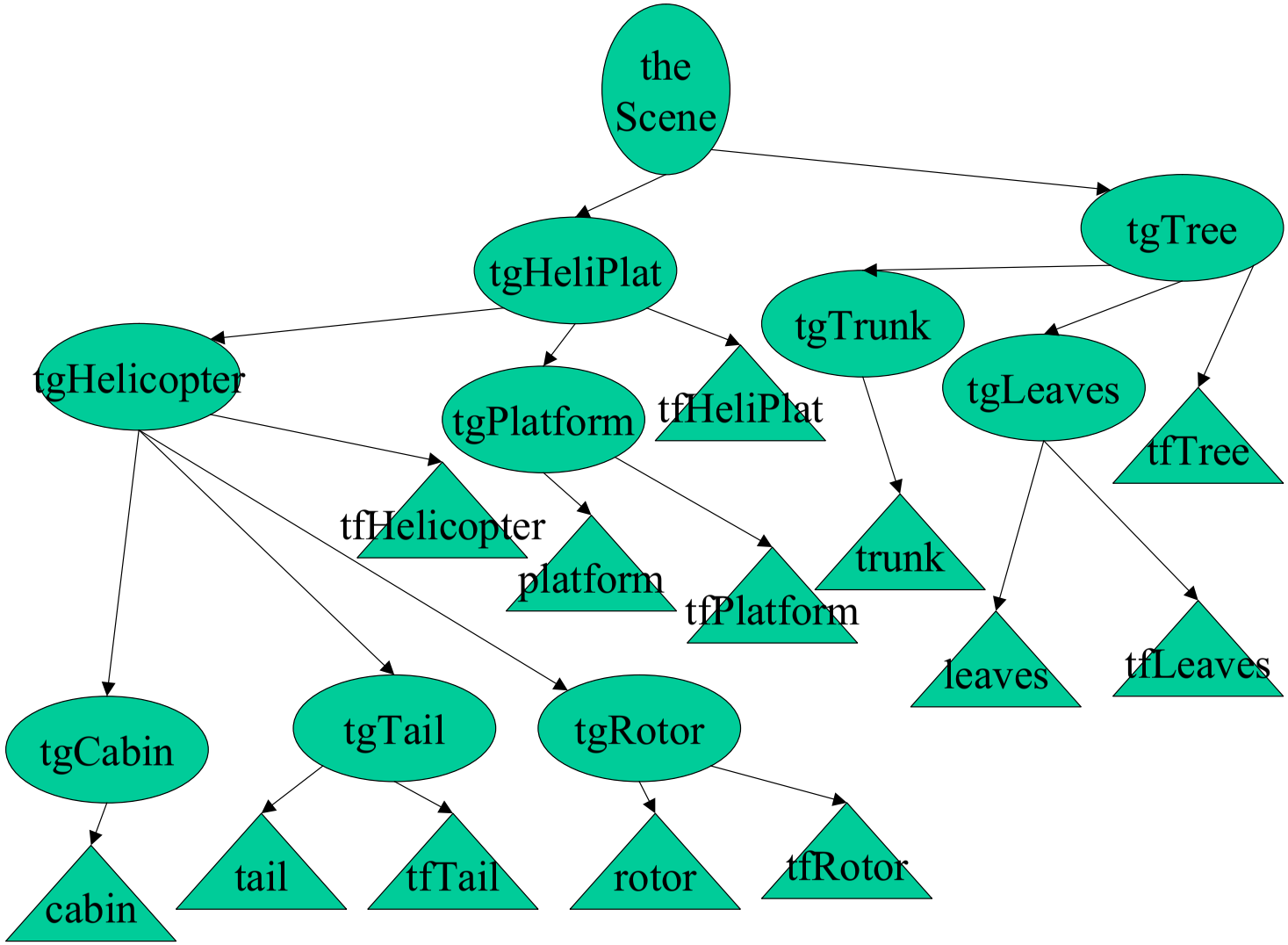
```
BranchGroup theScene =
                    new BranchGroup();
```

- The method `theScene.addChild(...)` can be used to assign nodes – either simple objects or more complex objects in the form of transformation groups – to the root.

StaticSceneExample.java



The scenegraph



The scenegraph

- Leaf nodes are either
  - elementary geometric objects (cabin, tail, rotor, platform, trunk, leaves) or
  - transformations (all nodes beginning with `tf`) used to position the objects correctly.
- All inner nodes are transformation groups (identified by the letters `tg`).

The scenegraph

Transformation groups are used

- to associate transformations to elementary geometric objects in order to transform them or
- to combine transformation groups to more complex objects and to consider and position the combined object as a single object.

In this way, the helicopter can be generated in the origin of the coordinate system, it can then be placed onto the platform and afterwards the platform can be positioned anywhere in the scene together with the helicopter.

Navigation

Adding the code lines

```
OrbitBehavior ob =
            new OrbitBehavior(myCanvas3D);
ob.setSchedulingBounds(
            new BoundingSphere(
                  new Point3d(0.0,0.0,0.0),
                        Double.MAX_VALUE));
simpUniv.getViewingPlatform(
            ).setViewPlatformBehavior(ob);
```

enables navigation through the scene using the mouse.

Animation (Movements)

Instead of a static transformation to position an object, a transformation group can also contain transformations for movements (and also other animations like colour changes).

Interpolators are needed to model the continuous transition of an object or a complete transformation group from an initial position or state to a final position or state.

PositionInterpolator: Linear interpolation between an initial and a final point. (Translation as animation.)

Interpolators

RotationInterpolator: Interpolation of rotating movement.

ScaleInterpolator: Continuous scaling.

ColorInterpolator: Continuous change of a colour.

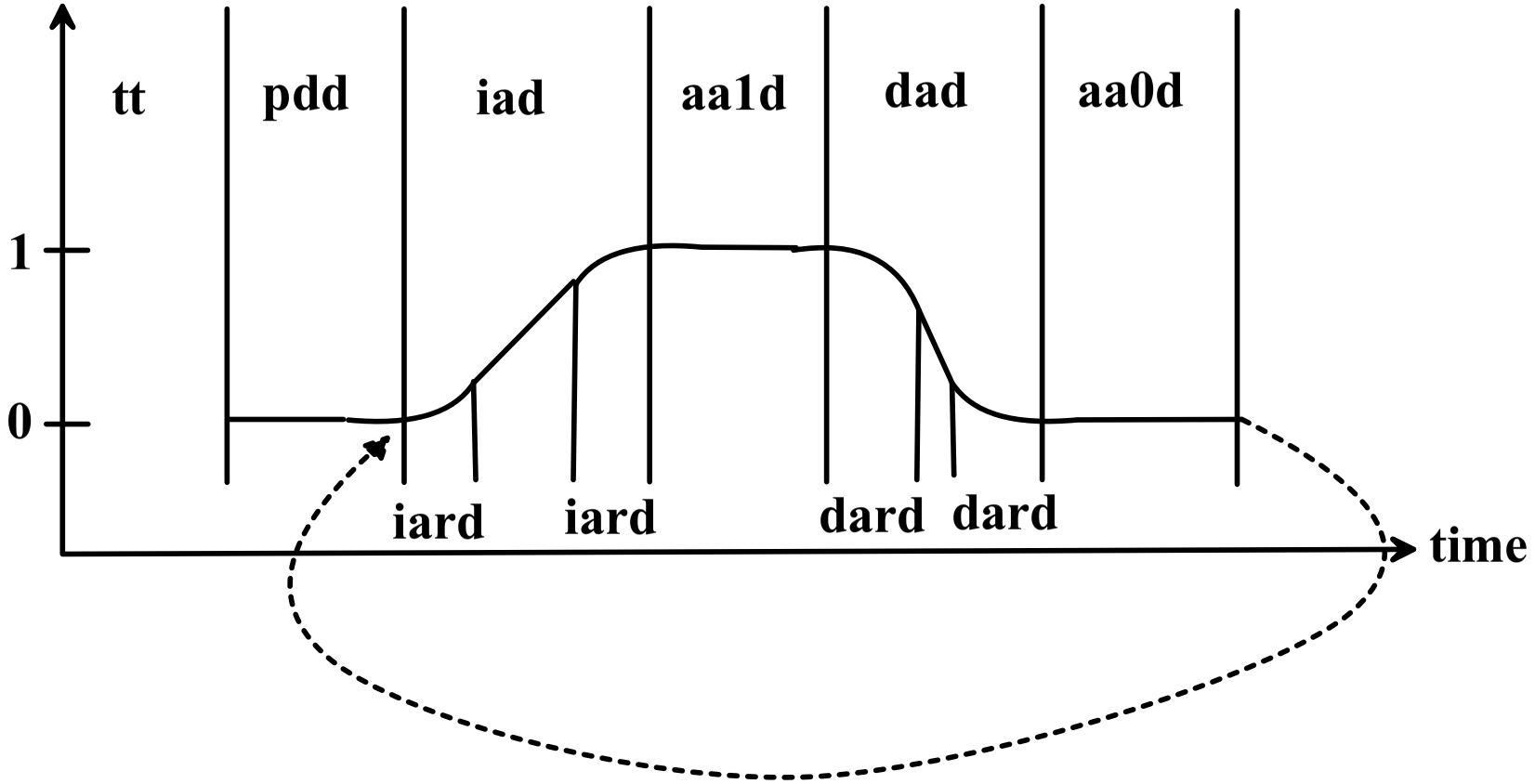
PathInterpolator: Interpolation along various transformations. For example, the `PositionPathInterpolator` enables a movement along a polyline.

Alpha values

- When should the interpolation start?
- Should the interpolation only go from state zero to state one or should it also be reversed?
- How long should the transition from state zero to state one take?
- Should the transition between the two states zero and one be carried out with constant speed or should it accelerate slowly in the beginning at state zero until a maximum speed is reached and then slow down again to have a smooth stop in state one?
- Should the interpolation be carried out just once or should it be repeated?

Alpha values

```
Alpha a = new Alpha(lc,id,tt,pdd,iad,
                    iard,aa1d,dad,dard,aa0d);
```



Alpha values

- `lc` specifies the attribute `loopCount`: No. of repetitions (for -1 repition without end).
- `id` defines the attribute `mode`:
  - `id=Alpha.INCREASING_ENABLE`: Interpolation from 0 to 1 only.
  - `id=Alpha.DECREASING_ENABLE`: Interpolation from 1 to 0 only.
  - `id=Alpha.INCREASING_ENABLE+ Alpha.DECREASING_ENABLE`: Alternating between interpolation from 0 to 1 and back to 0.

Alpha values

All following parameters are of type `long` and specify a duration in milliseconds.

- `tt` defines the attribute `triggerTime`: After how many milliseconds after the start of the program `Alpha` should deliver the first values.
- `pdd` defines the attribute `phaseDelayDuration`: `Alpha` remains in the state `0` for `phaseDelayDuration` milliseconds after the `triggerTime` has passed.

Alpha values

- `iad` defines the attribute `increasingAlphaDuration`: Length of the transition time from state 0 to state 1.
- `iard` defines the attribute `increasingAlphaRampDuration`: Duration of the linear acceleration phase until the constant maximum speed is reached.
  Also used for slowing down the movement, before state 1 is reached, so that the object comes to a smooth and not a sudden stop.

Alpha values

- `aa1d` defines the attribute `alphaAtOneDuration`: Duration for state 1.
- `dad` and `dard` determine the attributes `decreasingAlphaDuration` and `decreasingAlphaRampDuration`, respectively. Same meaning as `increasingAlphaDuration` and `increasingAlphaRampDuration`, but for the transition from 1 to 0.
- `aa0d` defines the attribute `alphaAtZeroDuration`: Duration for state 0.

PositionInterpolator

- Definition of the axis on which the positions to be interpolated lie.

  `Transform3D axis = new Transform3D();`

  defines the x-axis.

  If a movement along another axis should take place, the transformation `axis` must be defined in such a way that it maps the x-axis onto the desired axis.
- Definition of the corresponding `Alpha alpha`.

PositionInterpolator

- Definition of the `PositionInterpolator`.

```
PositionInterpolator pi =
  new PositionInterpolator(
          alpha,transformgroup,axis,
          startingPoint,endPoint);
```

  - `transformgroup` is the transformation group to which `pi` is assigned.
  - `startingPoint` and `endPoint` specify the initial and the endpoint on the `axis` for interpolation.

PositionInterpolator

- The (viewing) region in which the interpolator should be carried out:

```
BoundingSphere bs =
   new BoundingSphere(
            new Point3d(0.0,0.0,0.0),
            Double.MAX_VALUE);

pi.setSchedulingBounds(bs);
```

- For the transformation group `transformgroup`, changes must be enabled.

```
transformgroup.setCapability(
TransformGroup.ALLOW_TRANSFORM_WRITE);
```

Interpolators

- Finally, `pi` must be assigned to the transformation group `transformgroup`.

```
tg.addChild(pi);
```

A `RotationsInterpolator` is defined in a similar manner.

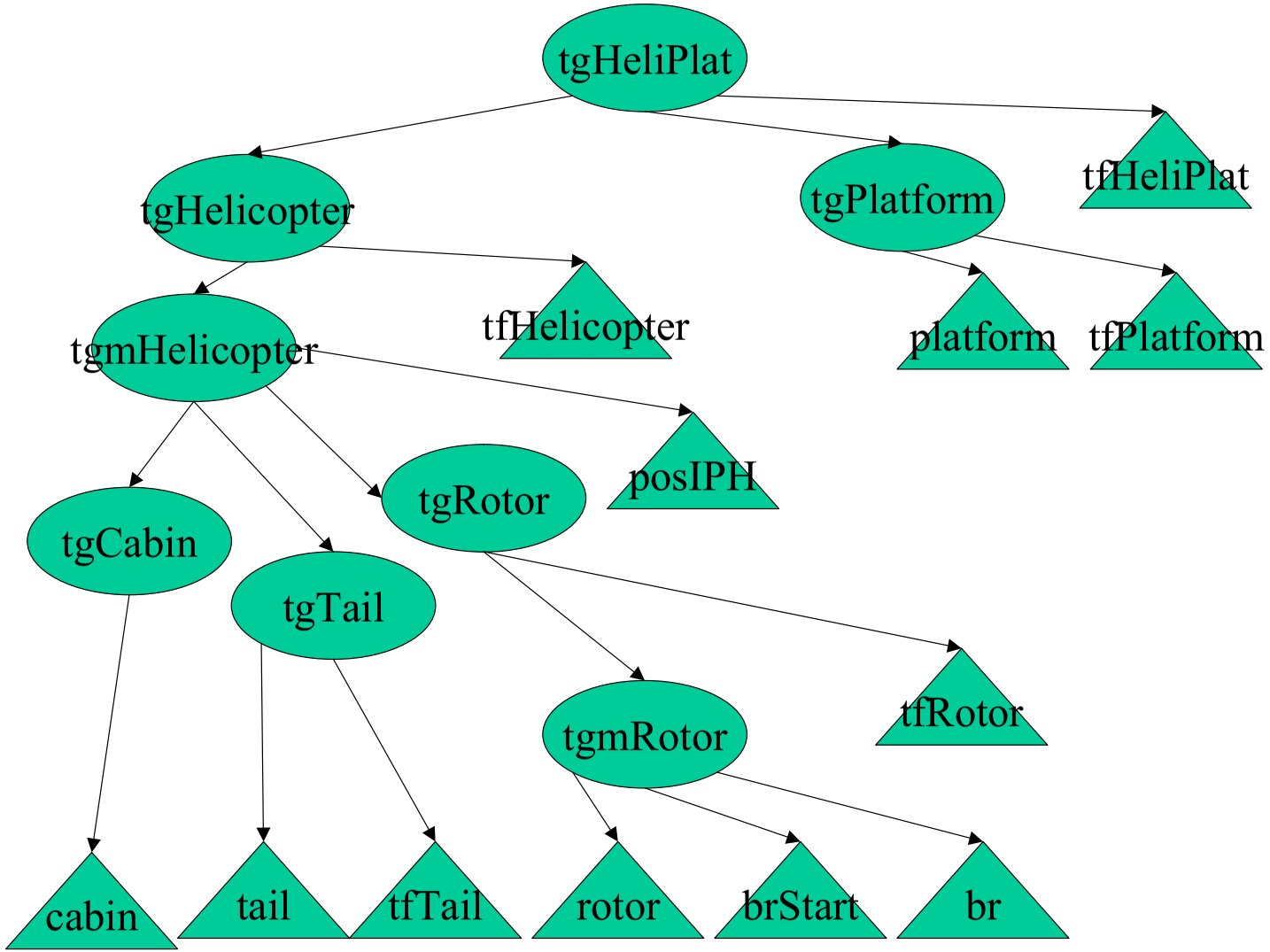
```
RotationInterpolator ri =
  new RotationInterpolator(
        alpha,transformgroup,axis,
        startAngle,endAngle);
```

Interpolators

```
ScaleInterpolator si =
   new ScaleInterpolator(
         alpha,transformgroup,axis,
         minScale,maxScale);
```

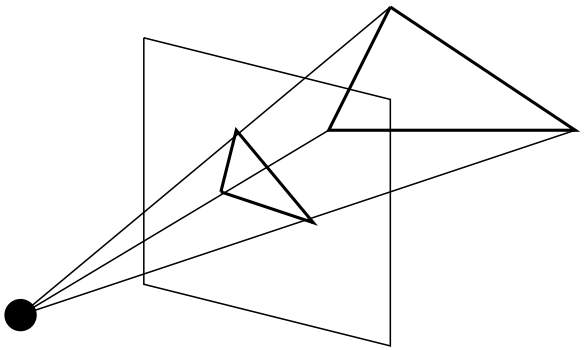
defines a `ScaleInterpolator` carrying out a continuous scaling starting with the factor `minScale` and ending at the factor `maxScale`.

Excerpt from the scenegraph

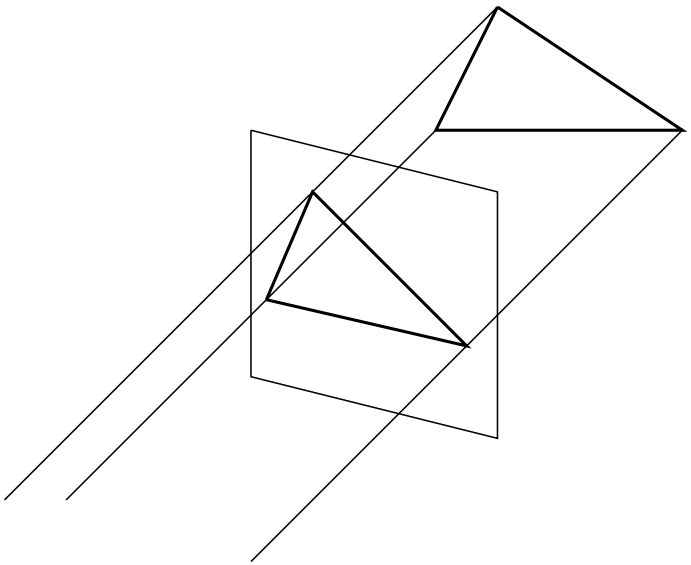


Projection

The projection of an object onto a projection plane is obtained by connecting the points of the object with the **centre of projection** and computing the intersection points of these lines, called **projectors**, with the projection plane. The centre of projection might be at infinity.



perspective projection



parallel projection

Projection

For a **parallel projection** the centre of projection is at infinity and the projectors are parallel. If a concrete centre of projection is given, the projection is called **perspective projection**.

The centre of projection represents the position of the viewer.

Java3D uses perspective projection by default.

Switching to parallel projection:

```
simpUniv.getViewer().getView().
setProjectionPolicy(
View.PARALLEL_PROJECTION);
```

(see `ViewParallelProjection.java`)

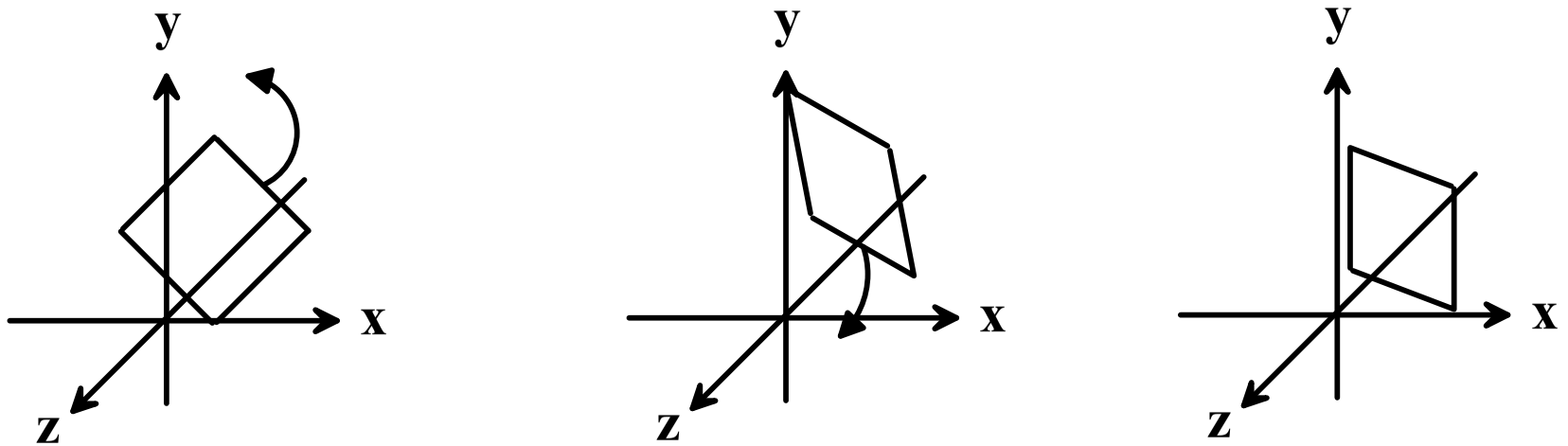
Parallel projection in hom. coord.

Example: Projection plane parallel to the (x, y)-plane with $z = z_0$ in homogeneous coordinates:

$$\begin{pmatrix} x \\ y \\ z_0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Parallel projection in hom. coord.

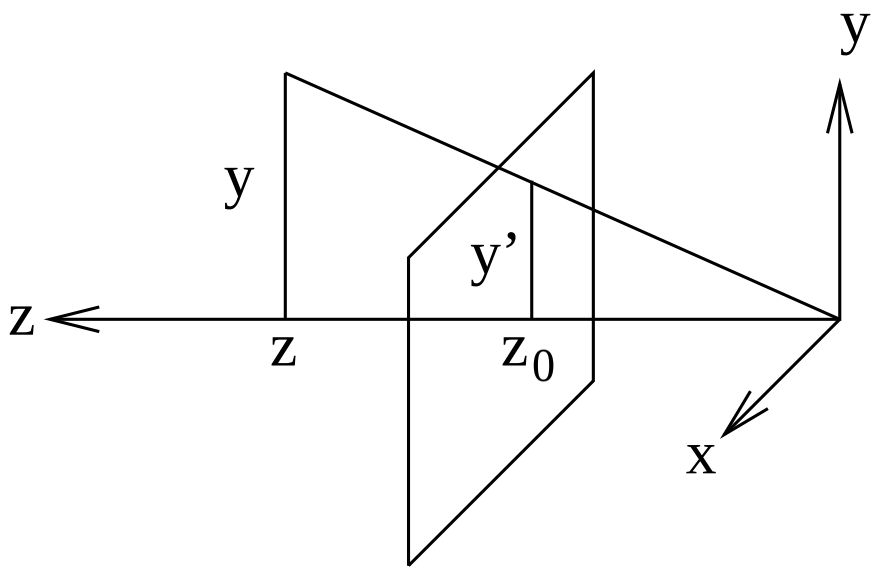
Applying a suitable geometric transformation, the virtual world coordinate system can also be changed such that the projection plane is parallel to the x/y-plane.



Persp. projection in hom. coord.

Example: Perspective projection with centre of projection at the origin of the coordinate system and projection plane parallel to the (x, y)-plane at $z = z_0$.

Applying an intercept theorem yields



Persp. projection in hom. coord.

$$\frac{x'}{x} = \frac{z_0}{z} \quad \text{and} \quad \frac{y'}{y} = \frac{z_0}{z}$$

i.e.

$$x' = \frac{z_0}{z} \cdot x \quad \text{and} \quad y' = \frac{z_0}{z} \cdot y$$

$$\begin{pmatrix} x \\ y \\ z \\ \frac{z}{z_0} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{z_0} & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

where the result is in **nonnormalised** homogeneous coordinates.

Persp. projection in hom. coord.

Using the x/y-plane as the projection plane and therefore shifting the centre of projection to the position $-z_0$ on the z-axis, results in

$$x' = \frac{z_0}{z_0 + z} \cdot x = \frac{x}{1 + \frac{z}{z_0}} \quad \text{and} \quad y' = \frac{z_0}{z_0 + z} \cdot y = \frac{y}{1 + \frac{z}{z_0}}.$$

In matrix form:

$$\begin{pmatrix} x \\ y \\ 0 \\ 1 + \frac{z}{z_0} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{z_0} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

Persp. projection in hom. coord.

perspective projection as a composition of a geometric transformation and a parallel projection:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{z_0} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{z_0} & 1 \end{pmatrix}$$

This transformation (the right matrix) does not change the points on the plane $z = 0$.

Persp. projection in hom. coord.

Given an arbitrary perspective projection, its centre can first be placed in the origin of the coordinate system.

Analogous to parallel projection, suitable rotations map the projection plane to a plane parallel to the x/y-plane.

A translation by the vector $(0, 0, z_0)$ yields a perspective projection with its centre of projection at $(0, 0, -z_0)$, which corresponds to another transformation plus a final parallel projection to the x/y-plane.

Persp. projection in hom. coord.

Therefore, any perspective projection can be considered as a suitable transformation $\widetilde{T}$ followed by a parallel projection to the x/y-plane.

It is sufficient to consider the parallel projection to the x/y-plane in order to understand any other parallel or perspective projection.

Other projections can always be considered as a transformation plus a final parallel projection to the x/y-plane.

Persp. projection in hom. coord.

Changing the position or the direction of view of the viewer corresponds to applying a suitable transformation before the above mentioned transformations and the parallel projection.

Persp. projection in hom. coord.

Properties of the transformation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{z_0} & 1 \end{pmatrix}.$$

Consider the point $\left(0, 0, \frac{1}{w}\right)$.

In homogeneous coordinates: $(0, 0, 1, w)$.

Persp. projection in hom. coord.

$$\begin{pmatrix} 0 \\ 0 \\ 1 \\ \frac{1}{z_0} + w \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{z_0} & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ w \end{pmatrix}$$

In cartesian coordinates:

$$\left(0, 0, \frac{1}{w}\right) \mapsto \left(0, 0, \frac{z_0}{1 + z_0 \cdot w}\right)$$

$w \to 0 :$ $\qquad (0, 0, \infty) \mapsto (0, 0, z_0).$

Persp. projection in hom. coord.

Transformation of the lines passing through the point $\left(0, 0, \frac{1}{w}\right)$:

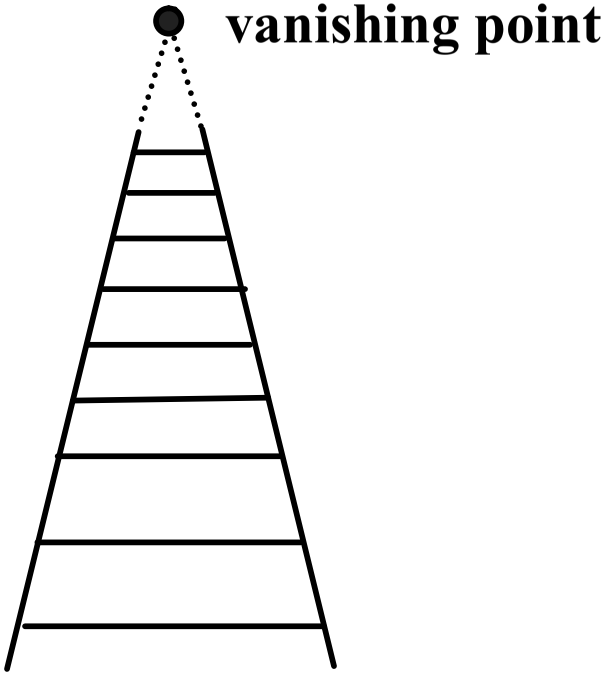
The transformed lines meet in the point $\left(0, 0, \frac{z_0}{1+z_0 \cdot w}\right)$.

$w \to 0$: The lines passing through the point $\left(0, 0, \frac{1}{w}\right)$ are transformed into lines parallel to the z-axis.

The transformed lines pass through the point $(0, 0, z_0)$.

This point is called **vanishing point**.

Persp. projection in hom. coord.



Persp. projection in hom. coord.

When the projection plane intersects with two or three axes of the coordinate system, such perspective projections are called **two-point** and **three-point perspective projection**, respectively.