JMONKEY NOTES

CONTENTS

FAST LOOKUPS - A COLLECTION OF ANSWERS TO FAQ

Problem	Solution	Code Location
disable the default camera movement	flyCam.setEnabled(false);	simpleInitApp()
make the annoying monkey disappear (skip start screen)	app.setShowSettings(false)	before call to app.start() in main()
disable the screen statistics	setDisplayStatView(false) setDisplayFps(false)	simpleInitApp()
Change the background color	getViewPort().setBackgroundColor(ColorRGBA color)	simpleInitApp()

Set screen settings, e.g. resolution, vSync and fullscreen	`public static void main(String[] args) {`
	`AppSettings settings = new AppSettings(true);`
	`settings.setResolution(1024, 768);`
	`settings.setFullscreen(true);`
	`settings.setVSync(true);`
	`Main app = new Main();`
	`app.setShowSettings(false);`
	`app.setSettings(settings);`
	`app.start();`
	`}`
		before call to app.start() in main()

Getting a wireframe model of your Geometry (e.g. for debugging)	`mat.getAdditionalRenderState().setWireframe(true);`	At any place, given a material *mat*

GEOMETRIES

Geometries are your "main objects" you are working with. Your player figure, the enemies, the landscape, all these visible elements are Geometries. This is the place where you define how your objects look, in terms of shape, texture/color, and location. Therefore it is not surprising, that geometries integrate

- the mesh data (class Mesh)
- the material (class Material)
- a transformation (internal: of type Matrix4f)

In order to create a Geometry, the following steps are needed:

1. create a Mesh
2. create the Geometry
3. create and set the Material
4. set the Transformation

In order to make the Geometry part of your scene,

5. Add the Geometry to your scene graph.

Since all elements added to a scene graph are "Spatials", the class Geometry inherits from Spatial (Geometry is therefore a sibling of Node), which already is the end of the line of inheritance (except for java.lang.Object, of course).

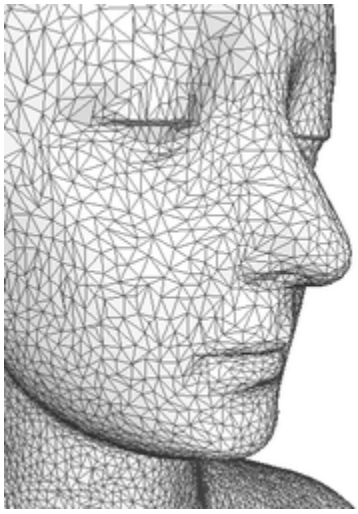
Example:

```
// create Mesh
Box b = new Box(Vector3f.ZERO, 1, 1, 1);
// create Geometry
Geometry geom = new Geometry("Box", b);
// create and set Material
Material mat = new Material(assetManager,  "Common/MatDefs/Misc/Unshaded.j3md");
mat.setColor("Color", ColorRGBA.Blue);
geom.setMaterial(mat);
// set transformation
geom.rotate(45.0f*FastMath.DEG_TO_RAD,0f,0f);
// add to scene graph
```

```
rootNode.attachChild(geom);
```

A VERY QUICK INTRODUCTION TO MESHES, MATERIALS, AND TRANSFORMATIONS

MESHES: DEFINING THE SHAPE



Meshes determine the *shape* of your objects. I assume that the programmers of JMonkey followed the philosophy that shape is the minimal requirement for physical existence. They therefore assigned this property the honorable position to be passed in the Constructor of a Geometry, while the other properties are set (e.g. setMaterial, setLocalTransformation) at a later point in time. Meshes in JMonkey always consist of 3D points, the mesh-*vertices* (yes, it's "vertices", not "vertexes", as you can read frequently in the JMonkey API). The vertices are connected by mesh-*edges* in order to create a *triangulated* surface mesh. This means, that all surface elements of any shape you encounter in JMonkey are triangles!

These meshes are often referred to as wireframe-models, since they do not explicitly describe surface properties; you can look through them.

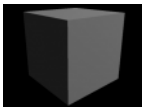
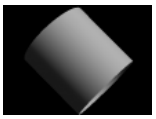
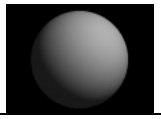
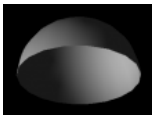
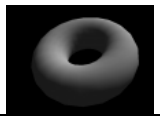
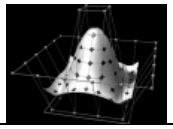
A mesh is mainly defined by two sets of data:

1. The vertices: 3D points (x,y,z), providing the location of each vertex in the Geometry's coordinate system
2. The edges: a set of index pairs (start, end), indicating which vertices to connect.

In JMonkey, a mesh contains even more data for the renderer to connect this data to texture and lighting, but this is what you should, at first, think of a mesh: vertices and edges.

You can create a mesh in one of the following three ways:

- From scratch: you define the data by hand, or you compute it. You might have seen beautiful visualizations of complicated models in Math: this is how they created those. To create a mesh this way is probably the hardest way. There is a tutorial online in the advanced JMonkey3 documentations, look for "custom meshes".
- You can use 3D modeling software: Blender, Sketchup etc. This is the usual way to create meshes that represent natural shapes. You can connect these with 3D scanners (Kinect etc.) to replicate models from the real physical world.
- You can use the (limited) set of built-in JMonkey *Shapes*. These are a good way to start. The following shapes are provided in the package com.jme3.scene.shape

JMonkey Shapes provided in the package com.jme3.scene.shape	
Box – A cube or cuboid. Single-sided Quad faces (outside only). StripBox – A cube or cuboid. Solid filled faces (inside and outside).	
Cylinder – A disk or pillar	
Sphere – A ball or ellipsoid	
Dome – A semi-sphere, e.g. SkyDome. This also allows for cones and pyramids: For a cone, set the Dome's radialSamples>4 and planes=2. For a pyramid, set the Dome's radialSamples=4 and planes=2	  
Torus – A single-holed torus or "donut".	
PQTorus – A parameterized torus. A PQ-Torus looks like a knot, a twisted, spiraled torus.	
Surface – A curved surface (called NURBS) described by knots, weights and control points. (Compare with the 1D spline curve shape.Curve)	

Remark: There are also non 3D shapes (Quad, Line, Curve, and Point).

MATERIALS: DEFINING THE TEXTURE

Materials define the texture properties of each triangle. Texture properties are color distribution and different kinds of reflectivity. The color distribution, i.e. the actual texture, can be defined by either a single color, a color gradient, or, and that's the most advanced, by patching an image onto the surface. But materials are more than that: materials come to life when their reflectivity is defined: is it a shiny material, is it even, bumpy, transparent, opaque etc. These properties can be defined in the Material class as well. Naturally, materials are closely related to the scene lights, in order to create the reflective appearance of your object. The topic of materials and lights can easily confuse, we will handle it in more detail later.

Hint: If you want to see the mesh of your Geometry, you have to set the Geometry's material to "**wireframe**" mode:

mat.getAdditionalRenderState().setWireframe(true);

TRANSFORMATIONS: DEFINING THE POSITION

By now we can create a shiny golden globe. But it still is always at the same position, it is exactly where the mesh-vertices are located. Of course we don't just want to create still lives with non-moving objects, all placed at the origin. What we want to do is to transform our Geometries in terms of shape (scaling) and location (rotation & translation). When we talk about transformations of Geometries, we talk about transformation of meshes, and therefore we talk about transformation of the mesh vertices. Again:

Transforming a Geometry means transforming the mesh-vertices.

A transformation can be anything that affects the vertices. The most important transformations are, however, scaling, rotation and translation. These are the transformations that JMonkey offers in different kinds of methods, and these are the only transformations you will need for any program. Transforming a Geometry using JMonkey transformations affects all mesh-vertices at one time, you can not single out specific vertices. This means, that, no matter what scale, orientation and location you assign to your Geometry, the shape always stays the same (in different sizes at different locations). These transformations are therefore called *rigid* transformations. If you want to change the shape of your mesh (e.g. to create ripples on a lake-surface), you must apply your own, non-rigid transformations. Non rigid transformation is an advanced topic. Let's stick with the classics, scaling, rotation, and translation. These can already sufficiently confuse. So let's have a closer look at them.

TRANSFORMATIONS

In this chapter we will take a closer look at scaling, rotation and translation of Spatials, i.e. Nodes and Geometries. Transformations are the main topic you will deal with when creating a 3D scenery. We will have a look at the mathematical description of transformations, which will make it much easier to describe and model your scene graph structure. We will also look at the JMonkey transformation details.

GEOMETRIES AND THEIR LOCAL TRANSFORMATIONS

The transformations we are interested in are the following affine transformations:

- scaling
- rotation
- translation

Every Geometry comes with its own, *local* coordinate system, in which the local transformations take place. When the Geometry is added to the scene graph, *its coordinate system* is added to the scene graph (with the Mesh-vertices in it). When utilizing a Geometry's transformation methods, e.g. Geometry.rotate(), or Geometry.setLocalTranslation(), this has an effect only to the Mesh inside its local coordinate system. Please meditate a second about this fact. Read the previous sentence again! Hence, when we talk about Geometry transformations, or *local* transfromations, we talk about a transformation relative to the coordinate system the mesh is embedded in. In contrast, a *global* transformation, which we will handle later, will transform the entire local coordinate system relative to a system it is embedded in, which very often is the 'world' or 'global' coordinate system. Although there is no theoretical difference between global and local transformations (in fact, they utilize the same methods inherited from Spatial), it makes sense to distinguish between these two, to unravel the quite often complex transformation sequences.

As an example for local and global transformations, imagine a car, consisting of multiple parts represented by simple Geometries (e.g. box for the chassis, cylinder for the wheels). We want to move the wheels relative to the chassis, but we also want to move the car, as a whole, relative to a street. We would therefore use local transformations to place the parts relative to each other to form a car. Then, in order to move the car as a whole, we would use global translations, which moves the entire coordinate system, taking all the car's parts along, relative to the street's coordinate system.

Let's look at local transformations first. This means, we are, for now, dealing with a single coordinate system, with a mesh embedded in it. Also, we are only interested here in scaling, rotation and translation. As we all know from linear algebra, the order of transformations is very important: it makes a huge difference, if we rotate first and translate then, or vice versa. Imagine, you get the commands "turn left (i.e. rotate) 90 degrees, then walk forward (i.e. translate) 5 steps". This ends up in quite a different pose (=position and heading) than the reverse command "walk forward 5 steps, then turn left 90 degrees). More mathematically spoken: if we want to transform a Geometry, we have to transform its n mesh-vertices. These can be represented by n 3D points v_1, v_2, …, v_n with $v_i=(x_i, y_i, z_i)$. Linear algebra (and the chapter about rotations below) teaches us, that a rotation can be performed by multiplication with a rotation matrix R: we yield a rotated version v_r of vertex v_i by $v_r=Rv_i$. The same holds for scaling, with a diagonal scaling matrix.

Using the *local* transform commands for JMonkey Geometries, however, the order does not matter, or better: the order is predefined. The reason is, that when these commands are executed, they only *set* the transformation parameters, yet they do not *perform* the actual transformation. What happens in JMonkey is, that the transformation related variables in the Geometry are updated, nothing else (JMonkey stores the transformation internally in a 4x4 matrix). The actual transformation takes place whenever the scene graph is rendered, which is at a later point. That point is not under the control of the (non-advanced) JMonkey user. However, we can just trust the JMonkey system,

that, at some time which usually is only split seconds away (in reality, it is *time-per-frame, tpf*, seconds away), the required transformation will be performed.

At rendering time, the local transformation is always performed in the same order:

1. **Scaling**
2. **Rotation (always around the origin, since we didn't translate yet)**
3. **Translation**

Hence, again, the order of *setting the local transformation parameters* does not matter. Setting the parameters is what happens when you use the transformation commands.

Note: all permutations of the following lines lead to the same result!

```
geom.setLocalTranslation(0,0,-10f);
geom.rotate(45.0f * FastMath.DEG_TO_RAD, 0f, 0f);
geom.setLocalScale(2f,2f,2f);
```

This seemingly leaves the programmer with the situation that he/she cannot translate first. So what to do when we need to translate first?

What does matter in terms of transformation order is the hierarchical order of Nodes and Geometries in the scene graph, along with their local transformations, leading to nested coordinate systems.

NESTED COORDINATE SYSTEMS AND THEIR RELATIVE TRANSFORMATIONS: GLOBAL TRANSFORMATIONS

Local transformations in Geometries are very limited, because of two facts:

1. The order is pre-defined (Scaling Rotation Translation)
2. The origin is unmovable, i.e. the coordinate system stays at its place, and does not change orientation.

For very simple sceneries this is sufficient. But imagine our everyday environment: a house on Earth, Earth itself, the moon, and the sun, not even mentioning the other planets flying around. If we try to model that system with only one, static coordinate system, it soon becomes hard. Let’s look at a different approach, multiple, nested coordinate systems. In order to do so, we just take the view point of each of the objects of interest, look how they are

transformed relative to a base system, and model each transformation in its own coordinate system. This means, we shift our view from working on mesh vertices directly to grouping those vertices into one coordinate system. We then perform one transformation on this coordinate system, and attach the result to yet another coordinate system, and so on, until we reach the static, global level, the unmovable coordinate system – the universe.

Let's start the example. We want to model a system containing:

- The sun. In our system, the sun does not move relative to the universe. It is static.
- The earth. It spins around its own axis, is translated relative to the sun, and rotates around the sun.
- A house on Earth. It is translated relative to the Earth's center, but follows all moves of the Earth.
- The moon. It spins around itself, is translated towards the center of the Earth, rotates around the Earth, but is independent of the Earth's spin.

This model is, of course, simplified. But it surely has some complexity to it, looking at all the transformations. Please sit down for a second and try to model this scenery with transformations in a single coordinate system! You will soon be frustrated, the complexity is already too high. Of course it is possible to model all transformations using a single coordinate system – after all, that's exactly what JMonkey has to do: it needs to find the single (very complex) transformation for each vertex, to place it into the right location in the one, global coordinate system that eventually matters: the view-system. What we will do, is to break down that procedure. The nested coordinate systems are the tool on the way to build a single, complex transformation. JMonkey (and all other game engines) go the same way: the construct of nested coordinate system is, you guessed it, the scene graph. Each of its nodes represents a single coordinate system, which is transformed relative to its parent. The single final transformation of a mesh-vertex, which resides in some node (Geometry) of the scene graph, results from subsequent transformations, defined by the unique path from the node, up the scene graph, to the root. Hence, when we build our model of nested coordinate systems, we model the scene graph.

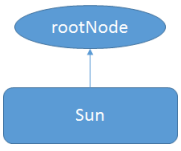
Step 1: determine all necessary single transformations for every Geometry

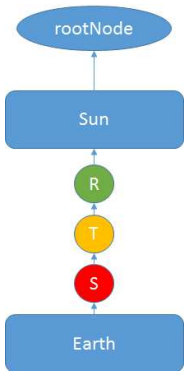
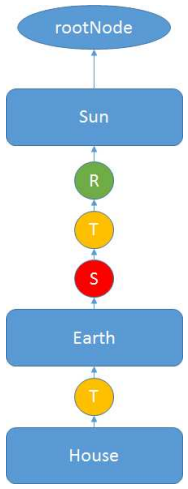
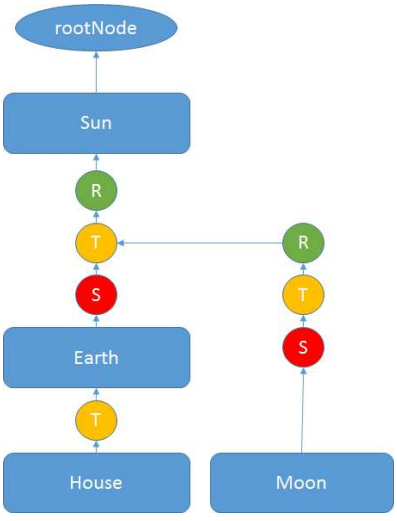
This step is the secret to success. You take the viewpoint of the mesh in the Geometry, and write down all transformations that occur, relative to the universe (i.e. the root-node).

It is of highest importance to get the order of transformations right!

The rule of thumb is: Spinning (rotation around your own axis) first, then translation, then rotation.

In our model:

The sun: when we take the sun's viewpoint, we see that it's the center of our universe, it does not change. That's simple. No transformations.	

The Earth: when we take the Earth's viewpoint, we observe the following (watch the order): a) It spins around itself b) It is translated relative to the sun c) It rotates around the sun d) It follows all other movements of the sun (which are nonexistent, at this time)	
A house on Earth: taking its viewpoint, we observe: a) It is translated relative to the Earth's center b) It follows all Earth movements, incl. the spinning	
The moon: from the moon's viewpoint, we see: a) It is spinning around itself b) It is translated relative to the Earth's center c) It rotates around the Earth d) It does NOT follow the Earth's spin, but all other movements of the Earth	

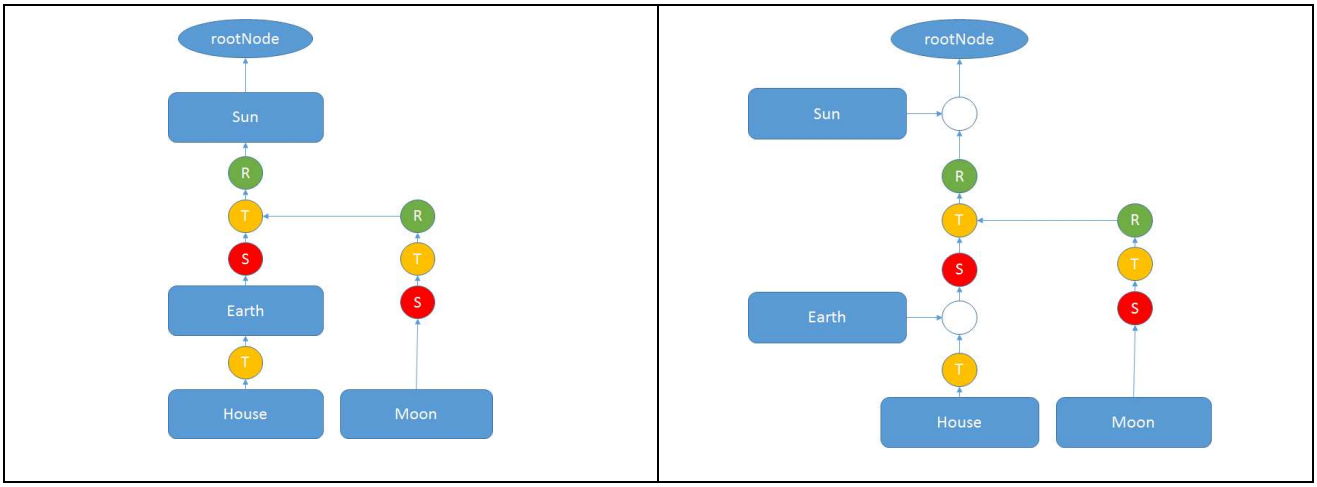
Note again: "S" stands for spinning, i.e. a rotation, not scaling!

From the written model (left column), it is straightforward to get to the diagram:

1. Define the element you want to hook into the diagram. Sun, Earth, Moon, House.
2. For each element, go backwards through its transformation description. This tells you where to hook into the diagram, and which transformations to insert. Stop, when you end at another element, or you are at the root.
3. Every transformation is a node in the diagram.

The JMonkey related interpretation of this diagram:

The diagram is very closely related to the scene graph. The blue boxes are the Geometries, circles are Nodes. The only little catch is, that in JMonkey, you cannot attach anything to a Geometry, only Nodes have the method “attachChild”. This means, you need to make a little change in the diagram: Geometries which have something attached to them must get their own node. The figure below shows the difference: Earth and Sun got their own Node. However, that’s more a technical detail. Perform this little step when you are about to implement the scene graph. You might also want to simplify the graph then, which we will handle a bit later.



Note again: "S" stands for spinning, i.e. a rotation, not scaling!

Talking Math

It is useful to look at the sequence of transformations independently of JMonkey terms, to get a deeper understanding of the underlying processes. Let’s talk coordinate systems.

When a Mesh is created, it is defined by vertex locations in its local coordinate system. That’s the blue boxes.

With every node in the diagram, we introduce a new (parent) coordinate system, in which the child coordinate system is embedded. Example: The earth mesh-vertices are defined in the local coordinate system “Earth”. This coordinate system is then embedded in a parent system (the red “spin” parent node), and transformed relative to this parent system, here: spinning (rotation around the origin). Then, the spinning coordinate system is embedded in yet another coordinate system, its parent, the yellow translation system “T”. The spinning coordinate system is then translated relative to the translation coordinate system, which is then embedded into its parent system. And so on.

Important: the transformation in one coordinate system affects all children recursively! As we know, order of transformation is important. The order is bottom up, from the Geometries (blue boxes), up to the root. If we write down the sequence for the "House", we get:

To each mesh-vertex in the House Geometry, apply T_E, then S_E, T_S, R_S. With the (somewhat confusing, since order-reversing) math operator "apply after", symbol "o", we get the transformed vertex v_t from a vertex v:

$$v_t = (R_S \text{ o } T_S \text{ o } S_E \text{ o } T_E)\, v$$

Translations can be expressed by adding a translation-vector. rotations (S, R) can be represented by matrices (please remember that "S" stands for spinning here, not scaling). Hence we get:

$$v_t = (R_S \text{ o } T_S \text{ o } S_E \text{ o } T_E)\, v = R_S(S_E(v + T_E) + T_S) = R_S S_E\, v + R_S S_E T_E + R_S T_S$$
$$= (R_S S_E)\, v + (R_S S_E T_E + R_S T_S)$$

Look at this equation! It is the solution to the frustrating problem to model the transformation in a single coordinate system. The right side shows a rotation $R = (R_S S_E)$, and a translation $T = (R_S S_E T_E + R_S T_S)$. These two transformations are applied to each vertex v in the house-mesh. Given the scene graph, JMonkey builds these transformations for us, and performs the transformation. That's the core of what the transformation system in JMonkey does.

If you understood the last paragraph, you could actually write a big chunk of the JMonkey game engine yourself! It is the important core part of recursively traversing the scene graph to create the transformation for every Geometry.

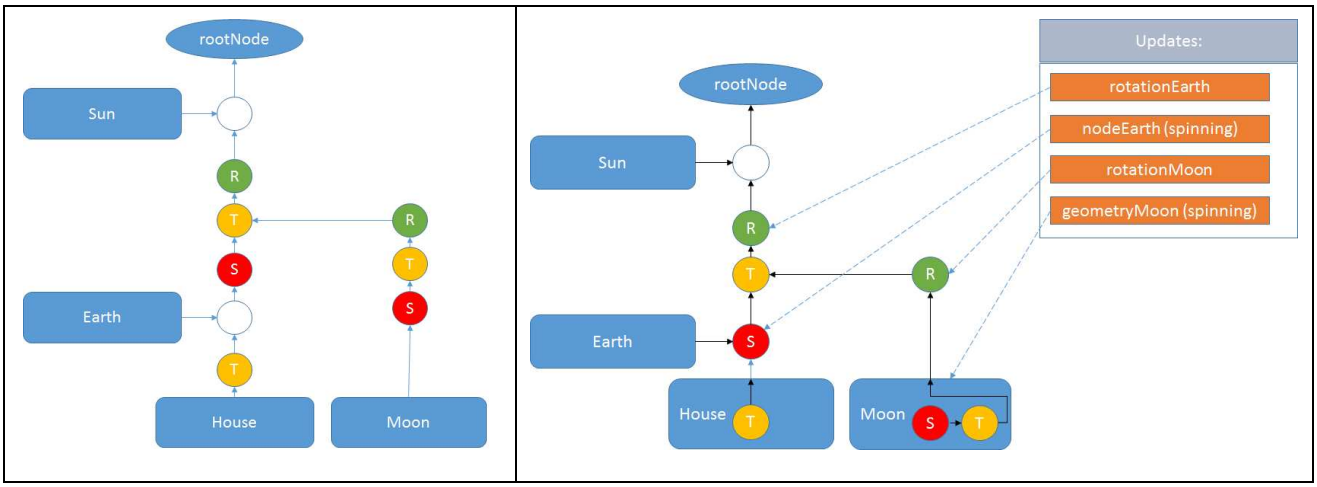
Simplifying the Scene Graph

You might have noticed that the diagram, or scene graph, that we created contains some unnecessary nodes. Remember that Geometries and Nodes are Spatials that can perform (scaling and) rotation and translation (in this pre-defined order). In certain cases, you can pull transformations in there, erasing some of the extra Nodes. This is a step you should only do at the very end of your design, when you are sure that your scene graph model is final!

In JMonkey, each spatial can perform one set of transformations, namely scaling, rotation, translation. Related to the scene graph structure, the spatial-inherited classes Node and Geometry differ in the way that only Nodes can have child-nodes. Nothing can be attached to a Spatial-object, or a Geometry-object. Hence, whenever we have a Node that has a child, it cannot be deleted/simplified.

But, whenever we have a sequence of spinning (i.e. rotation around the origin) and translation, *and no other branch hooks into these nodes*, we can pull them into one (parent) node..

This leads to the following, simplified diagram (left original, right simplified):



By pulling transformations into the Nodes and Geometries (if nothing is attached), we yield 9 Spatials (4 Geometries, 5 Nodes), instead of 13 Spatials (4 Geometries, 9 Nodes). To bring life into the system, the four rotation nodes (see figure) need to be updated.

Please observe that we could pull the spinning and translation into the Moon Geometry, while we needed to keep the Earth's translation separate, because the Moon's rotation node hooks into it. Simplifying too early in the design process can be critical, situations like adding an extra moon etc. happen often!

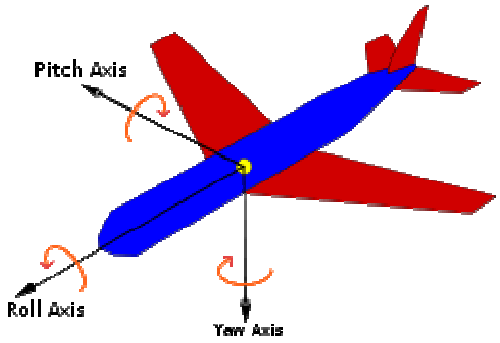
In terms of speed performance, I would assume both models to be very similar: JMonkey has to recursively collect all single translations anyway. The expensive processing part is the generation of the transformation sequence, hence the minimal extra work of collecting the sequences from extra nodes is most likely unimportant. In terms of memory usage, the simplified version wins, of course. But seriously, memory usage for a Node? All in all, you may or may not simplify the graph. It might be a good practice to pull the spin transformations into the Geometries, and leave the rest extra.

(There's a Demo related to this chapter: Examples_4350_SceneGraph_SunAndMoon)

DETAILS ABOUT SCALING, TRANSLATION AND ROTATION

ROTATION USING EULER ANGLES

Euler angles describe 3D rotations around the coordinate axes x, y and z. These rotations are often referred to as pitch, roll and yaw (unfortunately with different and often inconsistent mapping between {pitch, roll, yaw} and {x, y, z}).



There are two very important facts about Euler angles:

1. Euler rotation consists of 3 single rotations. The result of Euler rotation depends on the order of the rotation sequence!

2. Euler rotation suffers from the gimbal-lock problem: it might be that an Euler rotation aligns two coordinate axes. A subsequent Euler rotation then has only 2, not 3 degrees of freedom! This is a very important problem if subsequent rotations are applied (i.e. chasing camera etc.)

The following will discuss these facts in more detail.

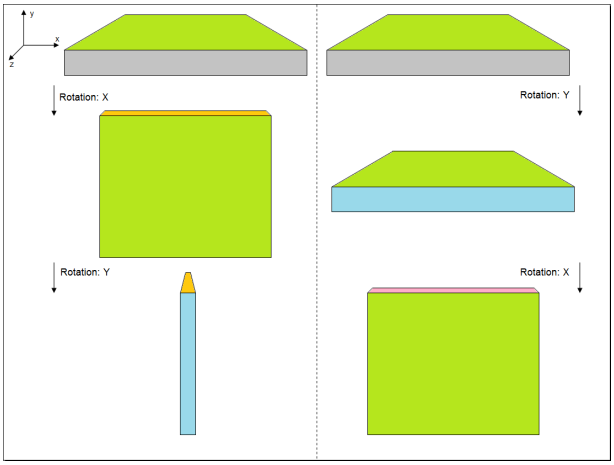
The Order of Euler Rotation in JMonkey

The method to rotate a spatial (i.e. Geometry or Node) using Euler angles is

```
Spatial.rotate(x, y, z)
```

This is a relative rotation, and, in contrast to scaling and translation, there is no absolute rotation method (e.g. setLocalRotation) using separate Euler angles. However, there are methods to set the local rotation using a rotation matrix or quaternions. There is no real reason for this, yet it can be seen as a hint, that in many cases when rotation is involved, it is preferable to use quaternions.

As mentioned before, a *full* Euler rotation consists of a sequence of 3 rotations, which we will call *partial* rotations. The result of the full Euler rotation depends on the order of its partial rotations: an object rotated around x (e.g. by 90 degrees) first, then around y (e.g. by 90 degrees) will receive a different pose than the same original object rotated y first, then x:



The question is therefore, in which order does JMonkey perform the partial Euler rotations? Although the rotation angles are passed into the method rotate(x,y,z) in order x,y,z, the order JMonkey rotates in order: X, Z, Y. Again:

JMonkey performs the Euler rotation in order of axes: X, Z, Y

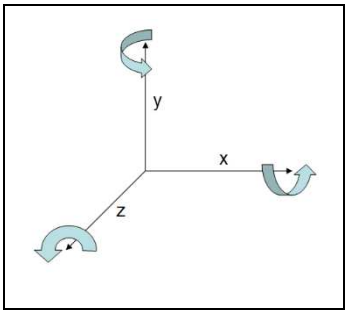
If we want to talk in terms of pitch, roll and yaw, and we want to use the terms in compliance with aeronautics as in the figure above, then we need to imagine an airplane flying towards us (i.e. along the Z-axis, wings spanned along the X-axis). An airplane aligned like that would be rotated by JMonkey in order pitch, roll, and yaw. Again:

If given an airplane directed towards us, JMonkey performs the Euler rotation in order

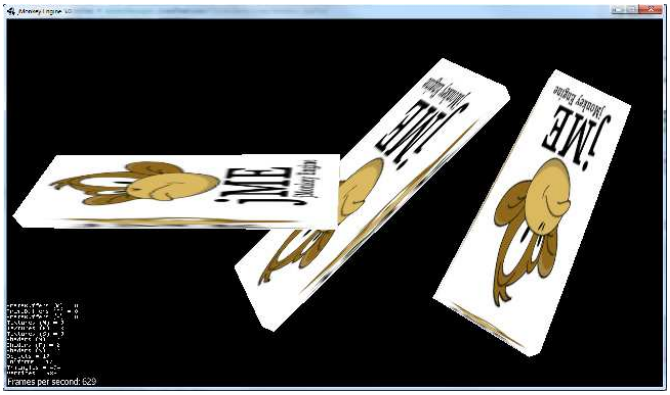
pitch (=x), roll (=z), yaw (=y)

Direction of Rotation

A positive rotation angle defines a counter-clockwise rotation when looking *against* the respective axis-direction (the axis-arrow points towards you):



The following program rotates 3 flat boxes around (x), then (x,z), then (x,z,y) respectively. Please note that it illustrates the order of partial Euler rotations in JMonkey (x – z – y):



```
public void simpleInitApp() {
        Box cubeMesh = new Box(3.5f, 0.2f, 1f);
        Material cubeMat = ... // (removed for readability)
        for (int i = 0; i < geometries.length; i++) {
            geometries[i] = new Geometry("My Textured Box", cubeMesh);
            geometries[i].setMaterial(cubeMat);
        }

        // geometry 0 goes to the left
        geometries[0].setLocalTranslation(-4.0f, 0f, 0f);
        rootNode.attachChild(geometries[0]);

        // geometry 1 stays at the origin
        rootNode.attachChild(geometries[1]);

        // geometry 2 goes to the right
        geometries[2].setLocalTranslation(4.0f, 0f, 0f);
        rootNode.attachChild(geometries[2]);

        // rotation
        float angle = 45f*FastMath.DEG_TO_RAD;
        geometries[0].rotate(angle,0f,0f);        // ROTATION ONLY IN x
        geometries[1].rotate(angle,0f,angle);     // ROTATION IN X AND Z
        geometries[2].rotate(angle,angle,angle);  // ROTATION AROUND X,Y,Z
    }
```

Rotation Angles and Rotation Matrix, 2D Case

Let's start with a simple 2D (not 3D) case: a rotation in 2D (not 3D) is defined by one rotation angle a only. Vector algebra tells us, that all linear transformations (and rotation is a linear transformation) can be expressed by matrix multiplications: given a vector $v = \begin{pmatrix} x \\ y \end{pmatrix}$, a positive (counter-clockwise) rotation is described by left-multiplication with the rotation matrix

$$R = \begin{pmatrix} \cos(a) & -\sin(a) \\ \sin(a) & \cos(a) \end{pmatrix}$$

which results in the rotated vector $v_r = \begin{pmatrix} x_r \\ y_r \end{pmatrix}$

$$v_r = Rv$$

With

$$x_r = xcos(a) - ysin(a)$$

$$y_r = xsin(a) + ycos(a)$$

In case you wonder where this matrix comes from, it's very easy to derive yourself (in case you did not wonder, just skip this paragraph). Let us have a look at the arbitrary vector $v = \begin{pmatrix} x \\ y \end{pmatrix}$. In order to look at its components separately, we can represent it by:

$$v = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ y \end{pmatrix}$$

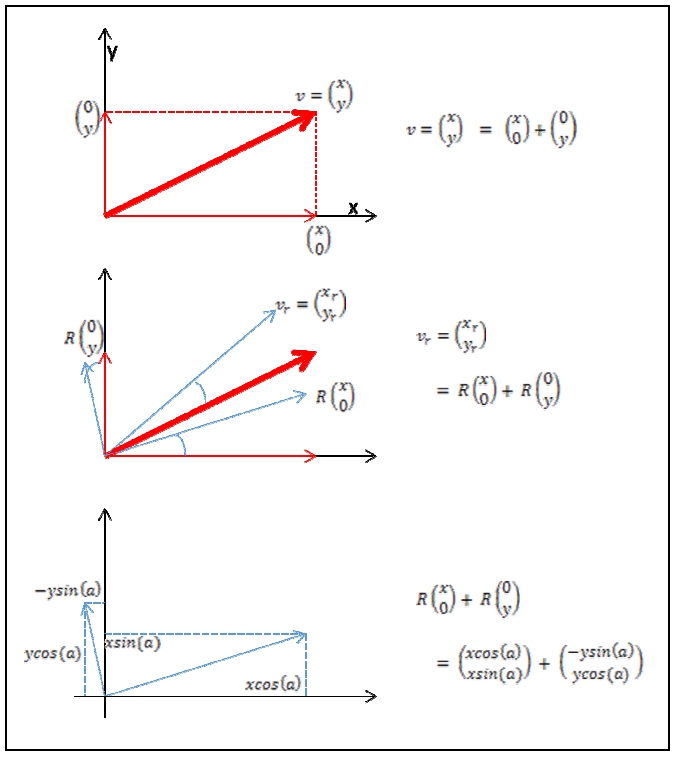
If we represent the rotation by R, we want to compute the rotated vector

$$v_r = \begin{pmatrix} x_r \\ y_r \end{pmatrix} = R\begin{pmatrix} x \\ y \end{pmatrix} = R\left(\begin{pmatrix} x \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ y \end{pmatrix}\right)$$

$$= R\begin{pmatrix} x \\ 0 \end{pmatrix} + R\begin{pmatrix} 0 \\ y \end{pmatrix}$$

Looking at the figure, we can easily see how to compute $R\begin{pmatrix} x \\ 0 \end{pmatrix}$ and $R\begin{pmatrix} 0 \\ y \end{pmatrix}$:

$$R\begin{pmatrix} x \\ 0 \end{pmatrix} = \begin{pmatrix} xcos(a) \\ xsin(a) \end{pmatrix}, \qquad R\begin{pmatrix} 0 \\ y \end{pmatrix} = \begin{pmatrix} -ysin(a) \\ ycos(a) \end{pmatrix}$$



hence,

$$v_r = \begin{pmatrix} x_r \\ y_r \end{pmatrix} = R\begin{pmatrix} x \\ 0 \end{pmatrix} + R\begin{pmatrix} 0 \\ y \end{pmatrix} = \begin{pmatrix} xcos(a) \\ xsin(a) \end{pmatrix} + \begin{pmatrix} -ysin(a) \\ ycos(a) \end{pmatrix} = \begin{pmatrix} xcos(a) - ysin(a) \\ xsin(a) + ycos(a) \end{pmatrix}$$

which is exactly what we stated above.

From 2D Rotation to 3D Rotation

To make the transition to 3D, just imagine the 2D rotation to be around the axis sticking out of the 2D plane. This would be the rotation around z in JMonkey. In 3D, we need to create a 3x3 rotation matrix, which is multiplied with a 3D vector (x,y,z). *Note that when rotating around z, the z coordinate of the vector does not change*. Hence, if we augment our 2D rotation matrix by one row and column in the following way, we gain the correct rotation R_z:

$$\mathrm{R_z} = \begin{pmatrix} cos(a) & -sin(a) & 0 \\ sin(a) & cos(a) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotating our 3D vector $v = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ yields:

$$v_r = \mathrm{R_z}\, v = \begin{pmatrix} xcos(a) - ysin(a) \\ xsin(a) + ycos(a) \\ z \end{pmatrix}$$

Similarly, the rotation matrices for the remaining axes are as follows:

$$\mathrm{R_x} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(a) & sin(a) \\ 0 & -sin(a) & cos(a) \end{pmatrix}$$

$$\mathrm{R_y} = \begin{pmatrix} cos(a) & 0 & sin(a) \\ 0 & 1 & 0 \\ -sin(a) & 0 & cos(a) \end{pmatrix}$$

$$\mathrm{R_z} = \begin{pmatrix} cos(a) & -sin(a) & 0 \\ sin(a) & cos(a) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(Please observe the difference in the "-sin" in R_y !)

The *full* Euler rotation R_E on a vector v (in JMonkey order x, z, y) can be described by the multiplication sequence

$$R_E\, v = (R_y\, (R_z\, (R_x\, v)))$$

$$R_E v = (R_y(R_z(R_x R_x))) = (R_y R_z R_x)v \quad \Leftrightarrow \quad R_E = (R_y R_z R_x)$$

The full Euler rotation can be described by a single matrix R_E, obtained by multiplication of the 3 partial rotation matrices (multiplied in the correct order). It is terrifyingly cumbersome to write the full, multiplied rotation matrix. Luckily we never have to create this by hand. JMonkey provides the class TransformationMatrix, which contains the method "setEulerRot(x,y,z)" (the internet will also show you the full rotation formula, if you want to see it).

Note: in most cases, it is useful to utilize quaternions instead of Euler angles. You can use quaternions without any deeper knowledge of their math for now. JMonkey offers a conversion from Euler angles to quaternions in one of the Quaternion-class constructors.

Working with Euler Angles: Examples

Euler angles are convenient when the angles are simple to derive. This is usually when the rotation axis is equal to one of the coordinate axes, or at least easily transformable, e.g. only involving 2 axes. Simulation of a "tilt swivel" unit for cameras, for example, is a typical Euler angle application --- in fact the device's name already points towards Euler angles. The interesting problems are those, where a vector is given, and Euler angles, or the Euler rotation matrix has to be determined. In these cases, we need operators that can translate vector properties to angular properties, hence in these examples operators like dot products, cross products, inverse tangents etc. come into play.

Example 1: determining Euler angles explicitly

Let us assume we have a camera on a tilt-swivel (rotation around z, rotation around y) unit. The current view direction is (0,0,1), along the z-axis. We now want to rotate it such that it points into the direction of a vector $v = (x, y, z)$. Let's assume $|v| = 1$, i.e. it is normalized. The task is to find the rotation angles (α, β) for rotation around x (tilt) and rotation around y (swivel) respectively. The order of rotation is tilt first, then swivel. To find the angles, we are looking at the inverse problem: rotate the vector v such that it becomes $(0,0,1)$. We then invert the resulting Euler angles, and the order of rotation. Hence, for our inverse problem, we look at swiveling first.

Finding the rotation angle β for rotation around y (swivel):The angle is described by the angle between the x-axis and the projection of v onto the x-z plane. Hence it can be computed with the help of atan2: $\beta = -atan2(x, z)$. Please observe the sign, and the order of coordinates! We apply the rotation R_y^β around y to our vector v, and obtain a new vector $\hat{v} = R_y^\beta v$, which lies in the y,z-plane.

Finding the rotation angle α for rotation around x (tilt): The angle is described by the angle between $\hat{v}$ and the z-axis. This is a 2D problem since $\hat{v}$ is already in the y,z-plane. Again, atan2 comes to the rescue: $\alpha = atan2(\hat{y}, \hat{z})$. Rotating $\hat{v}$ around z by α using a rotation R_x^αwould gain a vector (0,0,1), i.e. the z unit-vector. We don't need to perform this rotation, since we were only interested in the angles.

Finally, since $R_x^\alpha R_y^\beta v = (0,0,1)$, the solution to our problem of rotating our view-direction (0,0,1) towards v is:

$$R_x^\alpha R_y^\beta v = (0,0,1) \quad \Leftrightarrow \quad R_y^{-\beta} R_x^{-\alpha} (0,0,1) = v$$

The wanted rotation matrix is $R_y^{-\beta} R_x^{-\alpha}$ (please observe the inversion in angles and rotation order).

Example 2: determining the rotation matrix without explicit computation of angles

This is a very interesting case, that only makes use of the cross product, and a very simple corollary from linear algebra. The task is the same as in the previous example: determine the rotation matrix that rotates the unit z vector onto a given (normalized) vector v . This time we will solve the task without explicit computation of the angles. We see the problem as a transform of coordinate systems: we want to rotate to rotate our given reference coordinate system, such that the z-axis points towards v. Let us create a coordinate system, in which v is one of the coordinate axis. This is simple, we only need to determine a system of 3 mutually perpendicular vectors (v, w, u), one of them being v. There is no unique solution to this version of the task, hence let's single out a natural solution: the axis of rotation, which directly rotates $z = (0,0,1)$ onto v, should be part of the coordinate system. too (this leads to a very

natural rotation system). This rotation axis is given by a vector that is perpendicular to both, the unit z vector, and v, i.e. it is the normal to the v,z-plane. We can compute it using the cross product:

$$w = z \times v$$

Of our wanted new coordinate system (v, w, u), we already determined 2 vectors, v, w. The remaining vector must be perpendicular to both, hence the cross product comes into play again:

$$u = w \times v$$

Normalizing these vectors (v, w, u) gives us a coordinate system we were looking for. Now comes the little trick from linear algebra: we want to rotate our reference coordinate system $(x, y, z) = \left(\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}\right)$ such that

$$R\begin{pmatrix}1\\0\\0\end{pmatrix} = u, \qquad R\begin{pmatrix}0\\1\\0\end{pmatrix} = w, \qquad R\begin{pmatrix}0\\0\\1\end{pmatrix} = v$$

(the first two can swapped, important is the mapping of z to v).

Linear algebra teaches us a simple fact, that with a each matrix represented transformation*, the columns of the transformation matrix are the images of the basis vectors* (meditate a second about that). For example:

$$\begin{pmatrix}a\ d\ g\\b\ e\ h\\c\ f\ i\end{pmatrix}\begin{pmatrix}1\\0\\0\end{pmatrix} = \begin{pmatrix}a\\b\\c\end{pmatrix}, \quad \begin{pmatrix}a\ d\ g\\b\ e\ h\\c\ f\ i\end{pmatrix}\begin{pmatrix}0\\1\\0\end{pmatrix} = \begin{pmatrix}d\\e\\f\end{pmatrix}, \quad \begin{pmatrix}a\ d\ g\\b\ e\ h\\c\ f\ i\end{pmatrix}\begin{pmatrix}0\\0\\1\end{pmatrix} = \begin{pmatrix}g\\h\\i\end{pmatrix}$$

We want to transform our basis vectors to u, w, v. Hence the transformation matrix contains u, w, v in its columns:

$$R = \begin{pmatrix}u_x\ w_x\ v_x\\u_y w_y v_y\\u_z w_z v_z\end{pmatrix}$$

Naturally, this matrix transforms $z = \begin{pmatrix}0\\0\\1\end{pmatrix}$ to $R\begin{pmatrix}0\\0\\1\end{pmatrix} = v$, which was our task. The operations to compute orthogonal vectors etc. were necessary to make this matrix a rotation matrix: rotations are represented by orthogonal matrices, which are matrices where rows and columns are orthonormal vectors, i.e. orthogonal and of unit length. Please observe that we obtained this matrix without computing any angles!

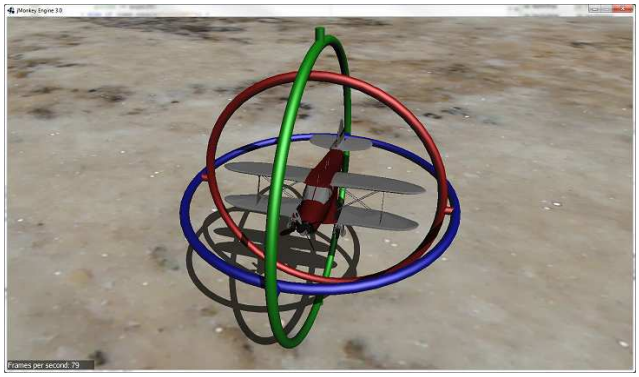
Rotation Matrix in JMonkey

There is no method in JMonkey to create a rotation matrix directly from Euler angles. Instead, given Euler angles, the rotation matrix can be created using the seemingly circuitous way via quaternions. In order to obtain the rotation matrix, use:

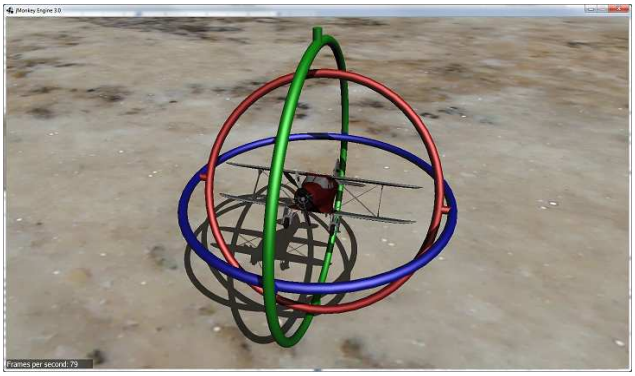
```
Quaternion q = new Quaternion();
Matrix3f rot = q.fromAngles(α,β,γ).toRotationMatrix();
```

SEQUENCES OF ROTATIONS USING EULER ANGLES: GIMBAL LOCK PROBLEM

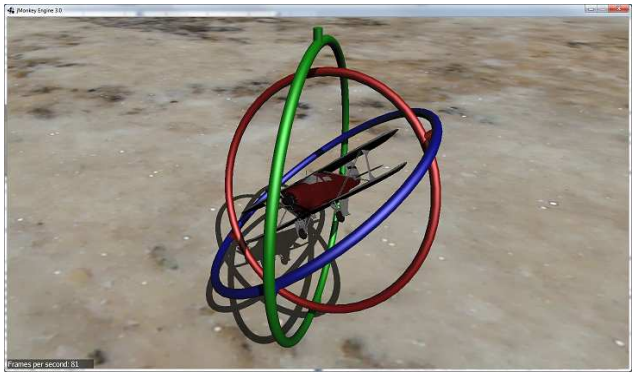
Of course it is possible, to describe sequences of rotations using Euler angles. Sequences of rotations could for example occur when controlling a chase camera. From the original state, a sequence of relative rotations is applied. The sequence then is represented by the multiplication of Euler rotation matrices. While there is seemingly no problem, certain cases can cause huge trouble, namely the cases of "gimbal lock". To explain the gimbal lock problem, let us first look at a mechanical device, the gimbal, that can visualize the result of Euler rotations very nicely.



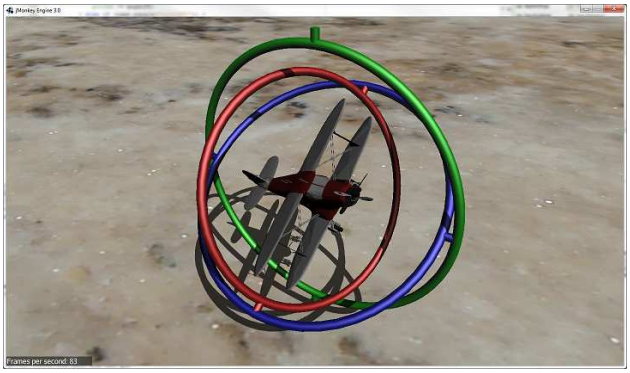
Initial Pose



Rotation around X (pitch)



Rotation around Z (roll) after pitch

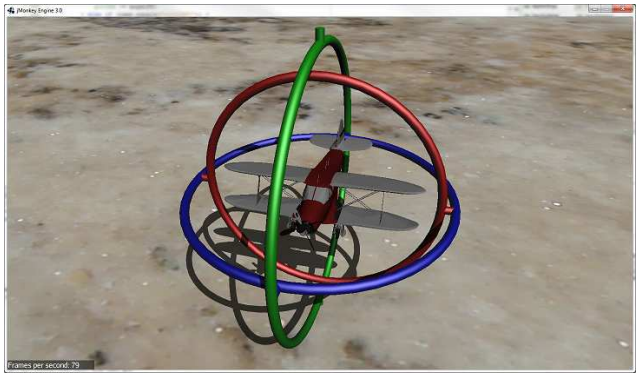


Rotation around Y (yaw) after roll after pitch

A gimbal is a mechanical device, which contains 3 nested rings, connected by hinges that allow each ring to rotate relative to its parent ring in one axis. Initialized in a pose such that the hinges first align with the 3 coordinate axes, applying a rotation to each of the rings performs exactly an Euler rotation. It is important to observe that the rotation of the outer rings (green, blue) do affect the positioning of the rotational axes of their respective inner rings: green does affect blue and red, blue affects red only, red does not affect any of the other rings. This means, that the rotation is performed in order red, blue, green. Looking at the initial position image (top left), these are the axes X, Z and Y (pitch, roll yaw). This gimbal follows exactly the order of Euler rotation in JMonkey.

In case we start from the initial pose, we can determine Euler angles for any possible heading of the airplane. Hence, starting from the initial pose does not cause any problem. In other words: a single Euler rotation can obtain all possible headings.

But let's see what happens if we have a sequence of two Euler rotations. In one special case, this can lead to a catastrophic limitation of possibilities: the gimbal-lock situation.



Initial Pose

90°roll, aligns the red and green axes: gimbal lock!

rotation around green axis

rotation around red axis after roll: same heading of airplane than rotation around green axis

Let's assume the first rotation in the sequence is a roll of 90 degrees. This case unfortunately aligns the hinge of the red ring, with those of the green ring. This means, rotation around red or rotation around green lead to the same result: we just lost the ability to rotate around X (it just results in a rotation around Y). We lost an entire degree of freedom! This situation is called the gimbal lock.

A gimbal lock can only occur in a sequence of Euler rotations, and only if the middle rotation aligns the outer and inner rotation axes. This is the only case, but it's a catastrophic one.

Gimbal lock problems can typically affect chase cameras: initially set to look in the y direction, they are updated with a sequence of rotations, depending on the actions of the object they chase. If one of the rotations results close to a gimbal lock position, then there is no way to smoothly get to a follow up position: you first have to get out of the lock, i.e. a seemingly unnecessary rotation around Y has to be performed first. This often results in weird, sling-like motions of the camera view.

The solution to the gimbal lock problem are quaternions.

ROTATION USING QUATERNIONS

Rotation with Euler angles can be quite problematic, independently of the problems with Gimbal lock. Imagine you want to rotate your Geometry around an axis that is not one of the coordinate axes! In 3D scenarios, this is the normal case --- not always does your player-figure align with the coordinate system, not always is the camera view along a coordinate axis: these are the cases where you need to be able to rotate around arbitrary axes. If you want to handle this case using Euler rotations, you have to break down the rotation into the 3 partial, axis aligned rotations. Determining the angles of the partial rotations is, of course, possible, but it requires some Geometry knowledge, and it is slow and inconvenient: for a single Euler rotation around a known axis, you first need to compute 3 angles, then you need to build the partial rotation matrices, and perform the partial rotations (or, with some longer code, you create the combined Euler rotation matrix).

Quaternions are a different representation of rotations in 3D space. They were first introduced 1843 by the English mathematician Sir William R. Hamilton; they are heavily used in 3D mechanics, Physics, and game programming. Their main advantage over Euler angles is:

- Concatenating rotations is computationally faster and numerically more stable.
- Extracting the angle and axis of rotation is simpler.
- Interpolation is more straightforward
- The representation (by 4 scalar values) is compact, compared to a 9-valued rotation matrix.

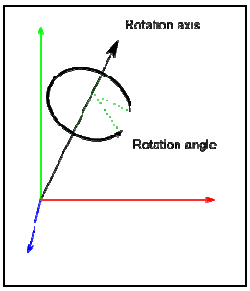
The first point is interesting for the JMonkey engine itself: to compute the resulting transformation of a Geometry in the scene graph, a sequence of single transformations (containing rotations) has to be computed. Quaternions are the faster tool here.

The second point is interesting for the programmer: many problems can be modeled and solved more intuitively using Quaternions instead of Euler angles.

The third point is also interesting for the programmer, especially in game programming cases: knowing the start and end state of a dynamic rotation, Quaternions can easily be utilized to achieve a visually pleasing interpolation. This is not true for Euler angles (especially in gimbal lock situations).

With Quaternions, you can dive into heavy Mathematics, if you want to. We will not do this here. We will introduce the practical side of Quaternions and how they are supported in JMonkey. However, it is always helpful to have a little mathematical intuition about the tools we use, therefore, after introducing the JMonkey Quaternion basics, we will dive a little deeper into the Math world.

Rotations Represented by Quaternions



Quaternions represent rotations in a very natural way, by means of a rotation axis, and the rotation angle.

The rotation axis is a 3-dimensional vector, the rotation angle is a scalar value. Therefore, Quaternions are vectors in a 4D space: q = (a,x,y,z). It would be too simple if the quaternion would just be the angle and the vector, wouldn't it? In fact, it isn't. But it can easily be derived if the angle and axis is known:

With a unit rotation-axis (i.e. length = 1) defined by (x,y,z), and a rotation angle given by α, **a Quaternion q is defined** by

$$q = \left(\cos\alpha\,, x\sin\left(\frac{\alpha}{2}\right),\ y\sin\left(\frac{\alpha}{2}\right),\ z\sin\left(\frac{\alpha}{2}\right)\right)$$

Remark: Quaternions have a vector length |q| = 1. This may come in handy for debugging: if your Quaternions do not have a length of 1, something went horribly wrong.

Now, this is just the representation of the rotation. To perform the actual rotation, we also need a procedure to execute a rotation to a 3D vector. The rotation is executed by performing a *Hamiltonian multiplication with the Quaternion and its conjugate Quaternion*. You see, immediately, there is math involved. Luckily, no details are needed here, since JMonkey provides all these multiplications. However, just to mention it, the actual rotation on a 3D vector v is executed by the Hamiltonian multiplication $v_r = qvq^*$ (more about this later). Again:

With q denoting a quaternion**, a vector v is rotated** by the Hamiltonian multiplication with q and q*:

$$\mathbf{v_r = qvq^*}$$

Remark: the multiplication with q and q* has more terms than a single rotation-matrix multiplication. In this case, if the matrix is known, using matrix multiplication (i.e. Euler rotation) is faster!

That already leads us pretty far. We can now rotate around an arbitrary axis. But it gets better. If we want to perform subsequent rotations around different arbitrary axes, it is as simple as it can be:

The Quaternion describing the resulting rotation can just be computed using the mysterious Hamiltonian multiplication again (this time without the conjugate q*).

With q_1 and q_2 being two Quaternions representing two rotations, **the combined single rotation q_{all}** that describes the sequence of rotating using q_1, then rotating using q_2, is defined by the Hamiltonian multiplication

$$\mathbf{q_{all} = q_2 q_1}$$

Remark: the Hamiltonian multiplication has less terms than the multiplication of rotation-matrices. This is, why Quaternions are preferred for the computation of sequences of rotations.

Important: Hamiltonian multiplication of two Quaternions yields a Quaternion.

What we just achieved is impressive: imagine the problem that you have a fly-camera in an arbitrary position. Assume you want to pan left and right (i.e. rotate around the camera's vertical axis), after rotating the camera around another

axis (e.g. the main axis of a spaceship). This is a very complex task! However, Quaternions make it really simple, just requiring the following steps:

- Create the Quaternion q_1 that describes the rotation around the spaceship axis. This axis is known, the angle is known, hence we can just build q_1 directly
- Do the same using the camera axis and a panning angle
- Multiply the two Quaternions
- Multiply the resulting Quaternions with the vectors to be rotated. Done.

To use Quaternions, we only have to solve the mystery of the Hamiltonian multiplication. We won't really solve it here, we will just mention the JMonkey methods to perform it.

Quaternions in JMonkey

Quaternions are used in Spatials to set the local rotation of Spatials (absolute rotation), or for relative rotation:

- Spatial.setLocalRotation(Quaternion q)
- Spatial.rotate(Quaternion q)

Quaternions have their dedicated class "Quaternion". To create a Quaternion, different ways are possible (remember, Quaternions describe rotations by means of axis and angle, therefore all methods somehow deal with these parameters). Interestingly, Quaternions in JMonkey do not own a constructor with parameters angle and axes. However, there is a method "Quaternion.fromAngleAxes", which does the trick. Many of the methods in the Quaternion class deal with conversion between Euler angles, rotation matrices and Quaternions. This reflects the practice in game programming: both models are used, and frequent conversions are normal. The following list shows some important methods/constructors in the Quaternion class. Please refer to the API for more details.

Constructor Summary

`Quaternion()` Constructor instantiates a new `Quaternion` object initializing all values to zero, except w which is initialized to 1.
`Quaternion(float[] angles)` Constructor instantiates a new `Quaternion` object from a collection of rotation angles.
`Quaternion(float x, float y, float z, float w)` Constructor instantiates a new `Quaternion` object from the given list of parameters.
`Quaternion(Quaternion q)` Constructor instantiates a new `Quaternion` object from an existing quaternion, creating a copy.
`Quaternion(Quaternion q1, Quaternion q2, float interp)` Constructor instantiates a new `Quaternion` object from an interpolation between two other quaternions.

Quaternion	**fromAngleAxis**(float angle, Vector3f axis) fromAngleAxis sets this quaternion to the values specified by an angle and an axis of rotation.
Quaternion	**fromAngleNormalAxis**(float angle, Vector3f axis) fromAngleNormalAxis sets this quaternion to the values specified by an angle and a normalized axis of rotation.
void	**fromAngles**(float[] angles) fromAngles builds a quaternion from the Euler rotation angles (y,r,p).
Quaternion	**fromAngles**(float yaw, float roll, float pitch) fromAngles builds a Quaternion from the Euler rotation angles (y,r,p).
Quaternion	**fromAxes**(Vector3f[] axis) fromAxes creates a Quaternion that represents the coordinate system defined by three axes.
Quaternion	**fromAxes**(Vector3f xAxis, Vector3f yAxis, Vector3f zAxis) fromAxes creates a Quaternion that represents the coordinate system defined by three axes.
Quaternion	**fromRotationMatrix**(float m00, float m01, float m02, float m10, float m11, float m12, float m20, float m21, float m22)
Quaternion	**fromRotationMatrix**(Matrix3f matrix) fromRotationMatrix generates a quaternion from a supplied matrix.

- If you need to transform your own vectors with Quaternions, you need to perform the multiplication defined in Quaternion: **Vector3d mult(Vector3d).** This performs the multiplication qpq*, not only qp!
- If you have a sequence of rotations, you need to multiply quaternions: **Quaternion mult(Quaternion q2**). This performs the multiplication qq2 (with this=q). Please note the difference between these two “mult” methods!

Example Application

Problem: You have a directional vector d, describing e.g. the direction of a laser beam in a game. You want to rotate the laser such that it points towards a certain point p. Find the rotation.

Assumption: we want to rotate around the origin (this assumption makes things a bit simpler, otherwise we would have to add some translations into the mix).

Solution: we will use Quaternions to solve the problem. Quaternions describe rotations by means of axis and angle, hence we need to determine the rotation axis and angle.

Axis: we find the axis by cross-multiplication of d and p: d x p yields a vector that is orthogonal to the plane P that is spanned by d and p (and which goes through the origin, this is where our assumption is helpful). This orthogonal vector is exactly our rotation axis. Rotating d around this axis always yields a point in the plane P!

Angle: the rotation angle is the angle between d and p, hence we can utilize the vector dot product d o p to compute the angle. However, d o p gives us the cosine of the angle, and getting the angle using FastMath.acos is ambiguous! The solution is: we do not only have the cosine, but also the sine of the angle, the cross product above gave us the sine, too (the length of the vector d x p, when d and p are normalized, is the sine of the angle between d and p). Having sine and cosine, we can use FastMath.atan2(sin,cos) to disambiguate the problem and compute the angle directly.

Given angle and axis, we can create the Quaternion, and (Hamilton-) multiply the vector d, to point in the direction of p.

DETERMINING PARAMETERS FOR TRANSFORMATIONS

Let's have a look back at the chapter about Euler Transformation. We defined angles, and utilized sine and cosine to generate the rotation matrix, then we performed a matrix-vector multiplication to yield transformed vectors which constitute vertices of our scenes. We can see, that our scene representation utilizes linear algebra, while our spatial understanding of the scene is a geometric one. We therefore often need translating operations between these two branches of mathematics. This chapter will shed some light on a few very basic connections between elements of linear algebra and their geometric interpretation. As simple as these tools are (we are not marching into the deep woods of mathematics), as much they empower you to analyze geometric properties of scenes, and to manipulate geometric objects in non-trivial ways.

ELEMENTARY VECTOR/MATRIX MATH

For readability, the example vectors are 3 dimensional, but it is straightforward to extend them to n-dimensions.

- The vector that translates a point $a = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$ into a point $b = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$ is $t = b - a = \begin{pmatrix} b_x - a_x \\ b_y - a_y \\ b_z - a_z \end{pmatrix}$
- Length $|a|$ of a vector a is: $|a| = \sqrt[2]{\sum_i a_i^2} = \sqrt[2]{(a_x^2 + a_y^2 + a_z^2)}$
- To scale a vector a by a scalar factor s, you multiply each component: $sa = \begin{pmatrix} sa_x \\ sa_y \\ sa_z \end{pmatrix}$
- To normalize a vector, i.e. scale it to length 1, you need to divide it by its length: $\hat{a} = \frac{a}{|a|} = a\frac{1}{|a|}$
- The distance between two points a, b is the length of the translation vector $|b - a|$
- The normalized directional vector from point a towards point b is $\frac{b-a}{|b-a|}$
- To scale a vector in a non-isometric way (i.e. different scales on different axes), you can multiply by a diagonal scaling matrix $S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}$: $Sa = \begin{pmatrix} s_x a_x \\ s_y a_y \\ s_z a_z \end{pmatrix}$
- A 2D (this only works in 2D) vector $a = \begin{pmatrix} a_x \\ a_y \end{pmatrix}$ can be rotated by $+90$ degrees using: $\bar{a} = \begin{pmatrix} -a_y \\ a_x \end{pmatrix}$

ANGLES

This chapter will mainly handle rotations and angles. Translations and scaling are too intuitive to dedicate an extra section to it. Which scaling was performed to transform a vector a = (2,7) to become b=(6,28)? Times 3 in x, times 4 in y. Which translation was performed to transform a vector a = (2,7) to become b=(6,28)? Plus 4 in x, plus 21 in y. But, which rotation was performed on a vector a = (3,2) to become b = (1.07, 3.44)? If you immediately could answer that the rotation angle was 39 degrees, then you are a genius and you don't need to continue to read.

So far, we utilized predefined, known angles and the rotation operation, to position geometric objects in a scene. However, whenever we find ourselves dealing with geometric environments, be it for games, simulations, or just visual representations, frequently our problems related to rotations are of a more analytical nature: the objects and their poses in a scene are known, yet we need to determine specific ways for transformations between them. For example, scenes in dynamic environments, like games, develop in complex ways: hordes of animals might roam a fractal landscape, and you might have to determine the camera angles to spot them, or to adjust their movement to each other's relative poses.

Since we describe our scene purely in terms of vector spaces, we are, as we mentioned in the introductory note, in the realm of linear algebra. Our spatial understanding of these scenes however is a geometric understanding, we want to determine angles (in our intuitive understanding of the term). Hence we constantly deal with different variations of the same core question:

Given two vectors of same length, what is the rotation to transform one into the other?

An example for a variation of this question would be: in a rollercoaster game the tracks are made of up of a sequence of straight parts of tracks. The direction of each part is a vector. When the cart proceeds from one track-part to the next, is it describing a left curve or a right curve? The question here is, if the angle is positive or negative.

An example for a combination of different transformations, including rotation, scaling and translation, is the following problem: in JMonkey, when we create a cylinder, its axis is always along the z-axis, its center is the origin. If we want to place the cylinder between two arbitrary points, e.g. to visualize a tube that connects two points in space, we have to scale, rotate, and translate this cylinder. While, again, the first two transformations are trivial (think for a second), the last one has some complexity to it. We will solve this problem at the end of this chapter.

The main operations that help us mapping vectors to angles between them are the dot and the cross product, as well as the $atan2$ function.

Computing the Cosine: Dot Product a o b

Given two vectors $v = (v_1, v_2, \dots, v_n)$ and $u = (u_1, u_2, \dots, u_n)$ of the same vector space $\mathbb{R}^n$, the dot product is defined as

$$v \circ u = (v_1u_1 + v_2u_2 + \cdots + v_nu_n) = \sum_{i=1}^{n} v_iu_i$$

There is a surprisingly simple connection between this sum and the angle $\alpha = \prec (v, u)$ between the vectors v, u:

$$v \circ u = \sum_{i=1}^{n} v_iu_i = |v||u| \cos(\alpha)$$

with $|v|, |u|$denoting the length of v, u respectively.

Again in words: if we perform the dot product between two vectors, it results in a scalar value, that describes the product of length(v), length(u), and the cosine of the angle between v and u. The lengths have to do with scaling, which is not our priority of interest right now. In order to get the angle, which we are interested in, we divide each vector by its length before we perform the dot product, i.e. we normalize the vectors.

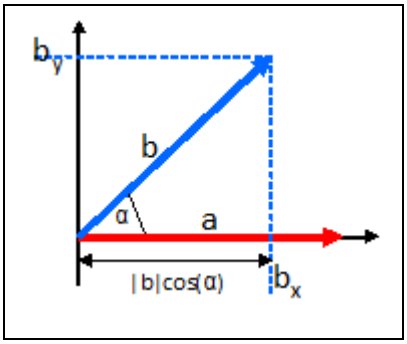
Let $\hat{v} = v/|v|$ and $\hat{u} = u/|u|$ be normalized vectors.

The dot product between two normalized vectors $\hat{v} \circ \hat{u}$ yields the cosine of the angle $\alpha = \prec (v, u)$:

$$\hat{v} \circ \hat{u} = 1 * 1 * \cos(\alpha) = \cos(\alpha)$$

Let us bring a little intuitive understanding into the relation between cosine and dot product. This will help you to not forget about this important relation!

Without loss of generality, we will place this example into 2D space. Again: The dot product between two normalized vectors $\hat{v} \circ \hat{u}$ yields the cosine of the angle between them. Let's therefore assume we have two vectors $\hat{v}$, $\hat{u}$ of length one. If we rotate both vectors by the same amount, the angle between them does not change. We can therefore rotate both vectors such that vector $\hat{v}$ lies on the x-axis, i.e. $\hat{v} = (1,0)$. In this configuration, the relation between the cosine and the multiplication rule is directly visible, look at the figure:

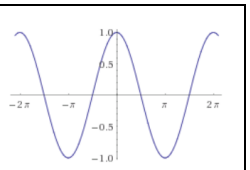


$$\hat{v}_x\hat{u}_x + \hat{v}_y\hat{u}_y = 1\hat{u}_x + 0\hat{u}_y = \hat{u}_x = |\hat{u}|\cos(\alpha) = \cos(\alpha)$$

Note: this transformation only makes use of the $\hat{v}_x\hat{u}_x$ part of the dot product, since $\hat{v}_y = 0$. To realize the need of the $\hat{v}_y\hat{u}_x$ part of the dot product, rotate the vector system such that $\hat{v}$ lies on the y-axis, and you will see.

The acos - Ambiguity Problem

Given the cosine of an angle, we get the angle using the inverse cosine (arccos, acos) function. However, there is a slight problem: computing the angle from a given cosine is ambiguous. The cosine is a symmetric function ($\cos(\alpha) = \cos(-\alpha)$ (see figure), hence, when given a cosine value, there are two angles it could originate from, α and $-\alpha$.



f(x)=cos(x)

By definition, the inverse cosine function always returns the positive angle:

arccos(α) is in [0...180] ([0...π]).

What is the effect on our problem to determine the angle between vectors, using the dot product? In all cases, where the angle is not in [0..180], the computation goes plain wrong:

- All negative angles are computed the wrong way, they are returned as their positive counterparts (the 'flipped' version)
- All angles in [180...360] are computed as 360-α.

In short: the inverse cosine of the dot product returns the **positive inner angle** between two vectors.

How can this be remedied? There are multiple ways. Assume we want to compute the angle between two normalized vectors a, b such that $R_\alpha a = b$

- utilize the atan2 function (see below), which also needs the sine of the angle (see cross product, below)
- sanity check: compute $\alpha = \text{acos}(a \circ b)$. Rotate and check: $R_\alpha a = b$? if not, use -α. The problem in 3D is of course, that you need to determine the rotation axis! This usually involves the cross product, which automatically, as a side product, gives us the sine of the angle, hence we can just go back to solution 1: use atan2. Frequently, even in 3D, the rotation axis is known, or even one of the coordinate axes. We then deal with a 2D problem, which makes the sanity check feasible.

Simple rule: if you have a 2D problem, sanity check might be the way to go, in 3D use atan2 with the help of the cross product. One of the examples below will use this solution.

Two important properties that follow directly from the cosine related property of the dot product:

1. If two vectors are perpendicular (orthogonal, 90 degrees angle), the dot product is zero.

   $a \circ b$=0 $\Leftrightarrow$ a, b are perpendicular

2. In 2D vector spaces, the sine of the angle between normalized vectors a, b can be computed by the dot product between a and a vector $\bar{b}$, which is perpendicular to b: $a \circ \bar{b}$ = sin(a).

Computing the Sine: The Cross Product $\boldsymbol{a \times b}$

Other than the dot product, which is defined in vector spaces of any dimension, *the cross product is only defined in 3-dimensional vector spaces*. The cross product between two vectors a,b does not result in a scalar, but in a third vector, which is perpendicular to both, a and b. It this therefore perpendicular to the plane that is spanned by a and b

(in case a,b are not collinear. In the collinear case, the cross product results in the null vector). The definition of the cross product directly points out its relation to the sine:

$$a \times b = \vec{n}|a||b| \sin \alpha$$

with $\vec{n}$ denoting a vector of length 1, perpendicular to a and to b.

Observe the similarities and differences to the dot product:

- similar to the dot product, the cross product between *normalized* vectors contains information about the angle (sin(a) for cross product, cos(a) for dot product)
- different from the dot product, the cross product yields a vector, not a scalar value. If a, b are normalized, then the resulting vector has length $sin(a)$.

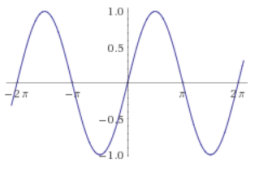
How to compute the cross product:

$$a \times b = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

Analog to the dot-product case, let's add a little bit of intuition, to sink the geometry deeper into memory. Without loss of generality, assume two normalized vectors a, b, rotated in a way such that a is lying on the x-axis (i.e., $a = (1,0,0)$), and b is in the xy-plane (i.e., $b = (x, y, 0)$). Rotating a system of any two vectors into such a configuration does not change the angle between them. Looking at the computation scheme, you see that the components to compute x and y of the result-vector both contain multiplication with a and b's z-component, which is zero. Hence, the result-vector, if not zero, must be on the z-axis, and therewith perpendicular to a and b! Looking closer at the z-component of the result-vector reveals that it actually computes the dot (again: dot!) product between two vectors a* and b*: a* is a, omitting its z-component, b* is b, omitting the z-component, and rotating it by 90 degrees (which makes it perpendicular to b in the xy plane). Remembering, that the dot product between a and $\bar{b}$ (perpendicular to b) yields the sine of the angle between a and b, we can directly see that the cross product creates the vector (0,0,sin(a)).

The asin-Ambiguity problem

Again, comparable to the cosine/inverse cosine case, computing the angle from the sine is ambiguous, since the sine function is anti-symmetric (sin(a) = -sin(-a)). Hence when given the sine, the inverse cosine will return an angle between [-90 ... 90].



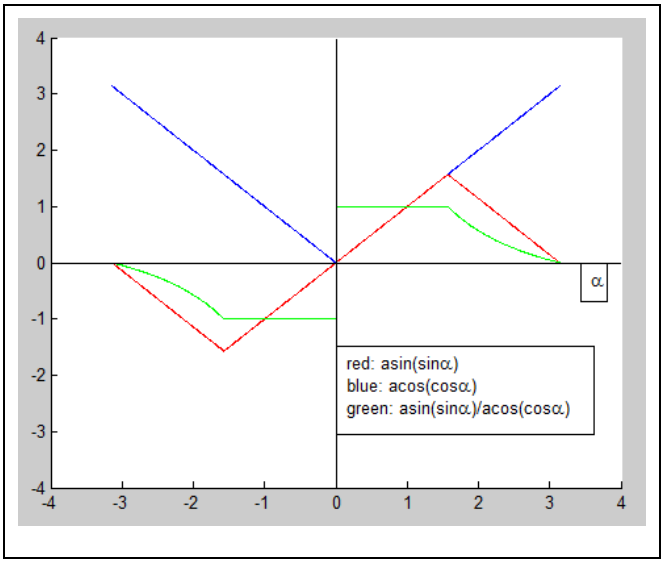
f(x) = sin(x)

arcsin(a) is in [-90 ...90] (-π/2 ... π /2)

For our problem, this means that angles out of that range are wrongly computed. The remedy: compute the cosine (dot product), and use the atan2 function.

Disambiguation: the Atan2 function

The figure to the right shows the result of an angle α (x-axis) and acos(cosα) as well as asin(sinα) (y-axis), respectively. This means, the input is an angle, and the output is the angle that results from plugging this angle into the cosine, and then into the inverse cosine function (same with sine, inverse sine). If there would be no ambiguity problem, we would expect the resulting angle to be identical to the input angle. The ambiguity however creates a different output. Looking at the blue and red curve separately, we see that we cannot disambiguate the angles. However, if we divide the two values (green plot), we see that we can distinguish between the four quadrants, and return a disambiguated angle.



This is exactly what the atan2 function does for us. In many computer languages (incl. JAVA Math package, JMonkey FastMath package), the atan2 function returns the angle between a 2D vector v and the x-axis in a non-ambiguous way, i.e.

atan2(v) is in [-180 ... 180]

The underlying function is of course the tangent function (tan = sin/cos), with its inverse (atan) and a disambiguation depending on the quadrant --- exactly how we just saw in the figure.

Please observe that v is a 2D vector, no a scalar value. The usual parameter order for computer languages is atan2(y,x). Example: to compute the angle of vector v = (15,9) with the x-axis, use atan2(9,15), which returns 30.9 degrees (0.54 rad). How does this help us in a 3D setting? It's as simple as it's tricky: for an angle between two 3D vectors a,b, compute the (ambiguous) cosine and sine values, and use these as input for atan2!

For two normalized 3D vectors $\hat{a}, \hat{b}$, the non-ambiguous angle α is computed by

$$\alpha = atan2(\sin\alpha, \cos\alpha) = atan2(|\hat{a} \times \hat{b}|, \hat{a} \circ \hat{b})$$