Slightly revised excerpt from Frisvad, J. R., and Frisvad, R. R. *Real-Time Simulation of Global Illumination Using Direct Radiance Mapping.* Master's Thesis, Department of Informatics and Mathematical Modelling, Technical University of Denmark, Report No. IMM-THESIS-2004-78, 2004.

9.1 Transformation

In this section we will introduce how rotation, scaling, shearing, and translation matrices are constructed and we will introduce the applicabilities of quaternions with respect to rotation.

The basic transformations: Translation, rotation, scaling, and shearing are all so called affine transformations, meaning that parallel lines in the transforming object are preserved. In rotation and translation the shape of the object does not change, such transformations are called rigid body transformations where both length and angles between points in the object are preserved. The following text basically follows chapter 3 in [1].

The transformation of a point is simply the inner product of the matrix and the point. The transformation matrix looks as follows:

$$\boldsymbol{X} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & t_x \\ a_{10} & a_{11} & a_{12} & t_y \\ a_{20} & a_{21} & a_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation, scaling, and shearing are done by altering different values in the 3×3 matrix:

$$\boldsymbol{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

Scaling works by altering values in the diagonal of $\boldsymbol{A}$, where a_{00} scales in the x direction, a_{11} in the y direction, and a_{22} in the z direction. Translation is done by changing t_x, t_y and t_z. Each of the following matrices rotate an entity of ϕ radians about one of the three axes:

$$\boldsymbol{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{R}_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{R}_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 9.1 shows how a shearing affects an object by skewing it to one side.

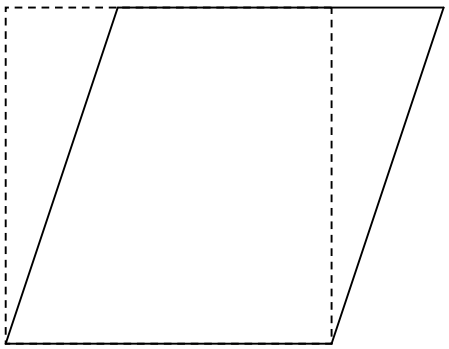

Figure 9.1: Shearing a box.

The shearing matrix is an identity matrix where one of the values that are not in the diagonal of $\boldsymbol{A}$ is non-zero. When one of these values is altered

it corresponds to shearing in one particular direction (positive or negative) on one particular axis.

Several transformations can be gathered in a single transformation matrix. If an object should be rotated a bit according to a transformation matrix $\boldsymbol{R}$ and then translated a bit according to a matrix $\boldsymbol{T}$, the final transformation matrix $\boldsymbol{X}$ is found by matrix multiplication of the translation matrix and the rotation matrix:

$$\boldsymbol{X} = \boldsymbol{T}\boldsymbol{R}$$

In this way more complex transformations can be created still using only one transformation matrix. Notice that $\boldsymbol{R}$ is applied first, but is written last. The order in which the matrices are multiplied has influence on the final outcome. This should be taken into consideration when creating the final transformation matrix.

The inverse transformation matrix is often useful, for example when switching between coordinate systems. The general way for computing the inverse of a matrix is given as explained in [1, p. 728] or any standard text book on linear algebra (eg. [3]). Another very useful way of computing inverses is in the case where $\boldsymbol{A}$ is an orthogonal matrix, which is always the case if the transformation is a concatenation of translations and rotations only. The inverse of an orthogonal matrix is given as $\boldsymbol{M}^{-1} = \boldsymbol{M}^T$, hence the inverse of $\boldsymbol{X}$ if $\boldsymbol{A}$ is orthogonal is given as:

$$\boldsymbol{X}_{\text{ortho}}^{-1} = \begin{pmatrix} a_{00} & a_{10} & a_{20} & -(\boldsymbol{t} \cdot (a_{00}, a_{10}, a_{20})) \\ a_{01} & a_{11} & a_{21} & -(\boldsymbol{t} \cdot (a_{01}, a_{11}, a_{21})) \\ a_{02} & a_{12} & a_{22} & -(\boldsymbol{t} \cdot (a_{02}, a_{12}, a_{22})) \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{9.1}$$

Quaternions

A compact and useful way to represent rotations is by use of a mathematical conception called *quaternions*, introduced to computer graphics in [5]. We will not describe the mathematical background of quaternions here, some references are [1, 2, 4]. Rather we will shortly introduce their usage.

A quaternion is represented by a four-tuple $\hat{\boldsymbol{q}} = (\boldsymbol{q}_v, q_w) = (q_x, q_y, q_z, q_w)$, it should, however, not be confused with homogenous coordinates. Operations on quaternions are given as follows [1, p. 45]:

Multiplication	$\hat{\boldsymbol{q}}\hat{\boldsymbol{r}} = (\boldsymbol{q}_v \times \boldsymbol{r}_v + r_w\boldsymbol{q}_v + q_w\boldsymbol{r}_v, q_w r_w - \boldsymbol{q}_v \cdot \boldsymbol{r}_v)$
Addition	$\hat{\boldsymbol{q}} + \hat{\boldsymbol{r}} = (\boldsymbol{q}_v + \boldsymbol{r}_v, q_w + r_w)$
Conjugate	$\hat{\boldsymbol{q}}^* = (-\boldsymbol{q}_v, q_w)$
Norm	$n(\hat{\boldsymbol{q}}) = q_x^2 + q_y^2 + q_z^2 + q_w^2$
Identity	$\hat{\boldsymbol{i}} = (\boldsymbol{0}, 1)$

Suppose we have a point or a vector given in homogenous coordinates $\boldsymbol{p} = (p_x, p_y, p_z, p_w)$. Let the quaternion $\hat{\boldsymbol{p}}$ be given as each component of $\boldsymbol{p}$ inserted in $\hat{\boldsymbol{p}}$. Now, given a unit quaternion $\hat{\boldsymbol{q}} = (\sin\phi\,\boldsymbol{u}_q, \cos\phi)$, where $\boldsymbol{u}_q$ is a vector representing an arbitrary axis, then:

$$\hat{\boldsymbol{p}} \mapsto \hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^{-1} = \frac{\hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^*}{n(\hat{\boldsymbol{q}})} = \hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^* \tag{9.2}$$

rotates $\hat{\boldsymbol{p}}$ (which corresponds to $\boldsymbol{p}$) around the axis $\boldsymbol{u}_q$ by an angle 2ϕ [1]. Note that for a unit quaternion it is the case that $\hat{\boldsymbol{q}}^{-1} = \hat{\boldsymbol{q}}^*$. It can be shown that any rotation can be obtained in this manner, see eg. [4]. This indicates that quaternions are a compact and efficient way of representing rotation in three-dimensional space.

The rotation given above, (9.2), is also called an adjoint map of $\hat{\boldsymbol{p}}$. It can also be shown (eg. [4]) that this adjoint map has a corresponding 3×3 rotation matrix, which is orthogonal. Since we have only made use of unit quaternions in this project, we present the conversion from a unit quaternion $\hat{\boldsymbol{q}}$ to a 3×3 rotation matrix $\boldsymbol{M}^q$ below. For the general formula we refer to [4, 1, 5].

$$\boldsymbol{A}^q = \begin{pmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y - q_wq_z) & 2(q_xq_z + q_wq_y) \\ 2(q_xq_y + q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z - q_wq_x) \\ 2(q_xq_z - q_wq_y) & 2(q_yq_z + q_wq_x) & 1-2(q_x^2+q_y^2) \end{pmatrix} \tag{9.3}$$

Another interesting feature of a quaternion is the ease by which the rotation from one vector to another can be specified. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be two unit vectors denoting a direction in space. A unit rotation axis is then given as $\boldsymbol{u} = (\boldsymbol{s} \times \boldsymbol{t})/\|\boldsymbol{s} \times \boldsymbol{t}\|$. If 2ϕ denotes the angle between $\boldsymbol{s}$ and $\boldsymbol{t}$, then $\boldsymbol{s} \cdot \boldsymbol{t} = \cos(2\phi)$ and $\|\boldsymbol{s} \times \boldsymbol{t}\| = \sin(2\phi)$. "The quaternion that represents the rotation from $\boldsymbol{s}$ to $\boldsymbol{t}$ is then $\hat{\boldsymbol{q}} = (\sin\phi\,\boldsymbol{u}, \cos\phi)$" [1, p. 51].

A few trigonometric calculations show that the formula finding a unit quaternion specifying the rotation between two vectors $\boldsymbol{s}$ and $\boldsymbol{t}$ is given as [1]:

$$\hat{\boldsymbol{q}} = (\boldsymbol{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+\boldsymbol{s}\cdot\boldsymbol{t})}}(\boldsymbol{s} \times \boldsymbol{t}), \frac{\sqrt{2(1+\boldsymbol{s}\cdot\boldsymbol{t})}}{2} \right) \tag{9.4}$$

This ends our presentation of transformations. In the following sections we will give some examples of their use.

9.2 Animation and Motion Control

Building on the transformation matrices introduced in the previous section, we can create a series of frames where different transformation matrices are

used in order to give an object the appearance of continuous movement. Such a movement is referred to as animation. Animation can be a simple movement, like a ball bouncing up and down, or it can be more complex, like a person walking or a facial expressions. All movements over time qualify.

Animation is normally done in a modeling application by use of key frames. Since the object is moving over time they need to be redrawn in a slightly different position for each frame. Redrawing each moving object for each frame would be an overwhelming task. With key frames only key transforms of the objects moving are set, after that the application computes the transformations of the frames in-between.

A simple calculation of object positions in-between key frames is by linear interpolation. However to create realistic transformations it is sometimes necessary that the movement is not linear. An example could be a ball jumping up and down. If the ball is to follow the laws of physics, it will move slower around its highest point and faster at set off and just before hitting the ground. To create such effects we can use interpolation curves, which can be found using quaternion curves, see [5, 2]. Movements like this could also be simulated by a physics engine supporting gravity, which is why a good physics engine can save a lot of animation time.

In Blender key frames are controlled in the *action viewer*, where all key frames are placed. By this view key positions of objects can easily be moved or copied to different frames. Blender also supports interpolation curves in the *Ipo view*.

As we have seen in chapter 7, characters are often created from one mesh. If characters are supposed to move body parts only (for example an arm or a leg), just some of the mesh can be moved instead of the entire object. For simulating walking, for instance, parts of the character mesh must be moved in different directions at the same time. For purposes like this we can create a skeleton-like mesh defining bones that can be attached to parts of the mesh. Such a skeleton is called an *armature*. Whenever a bone in the armature moves vertices attached to it will follow. In this way we can modify only parts of the mesh influenced by the bones. An example of an armature can be seen in figure 9.2.

There is difference between animations for a movie and animations for a real-time application such as a game. Except for the fact that movie animations are much more detailed and spectacular, they are also predetermined. This means that the artist can concentrate on particular movements only. In a game, for example, movement of objects most often depends on how the user interacts with the scene. The fact that the exact movement of objects in a real-time application is unknown, means that animations must be split up into smaller pieces, each containing movement we know will be useful in many situations. Moreover animations like these must be able to follow each other in a fluent manner.

Small animation snippets, like those described above, are called cycles.

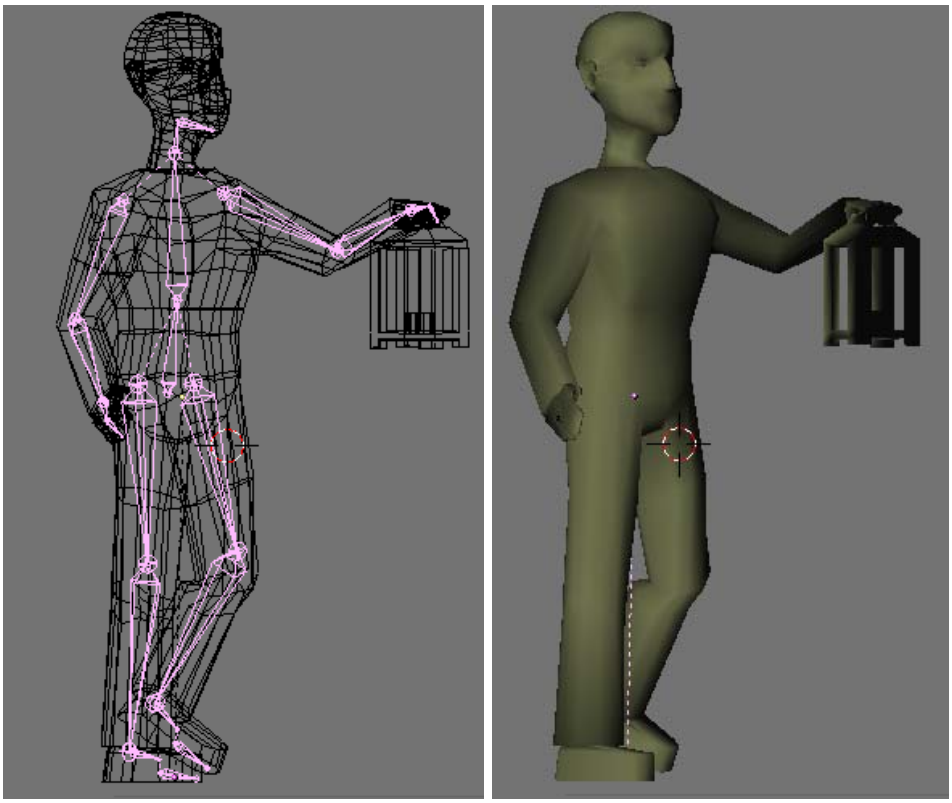

Figure 9.2: Armature of the male character in the cave scene. Notice how the mesh follows the bones.

In a game with a character controlled by a user, cycles such as a walk cycle or a hit cycle could be useful. In the example of a walk cycle the animation snippet must be able to 're-cycle', meaning that if several walk cycles follow one another, they will appear as if the character just kept walking.

To have a moving light source in our cave scene the character shown in figure 9.2 is supposed to walk about with a lantern, a walk cycle was created for this purpose and an idle cycle. Unfortunately there has not been time to export these animations to our own application. In order to show that a scene is truly dynamic, we must also be able to alter it dynamically. In real-time applications this usually happens by moving things around in the scene using some sort of input device (mouse, keyboard, etc.). Interactive control is the subject of the next section.

9.3 Interactive Control

In many real-time 3D applications a simple virtual track ball is connected to a mouse input device and used for camera navigation. This is also the

case for the application implemented during this project. The track ball we use was originally distributed during the DTU "Computer Graphics" course (02561). During this project we have modified the track ball from time to time and we have particularly adapted it to work for navigation of a chosen object (according to the current camera view) as well as it works for camera navigation. In the following we will first describe how the track ball works for interactive *camera* control, and second we will describe our expansion of it to include interactive *object* control.

The track ball is initialized by a center $\boldsymbol{L}$ around which the camera should rotate, and a distance z_{eye} which specifies how far away along the z-axis the eye point, or camera, should be placed. Most often the center is defined as the center of the scene in which the track ball is placed or the center of a particular object in the scene.

Internally the track ball has a translation vector $\boldsymbol{t} = (t_x, t_y, t_z)$ which is applied in view space, meaning that altering (t_x, t_y) results in a pan motion of the camera, while altering t_z results in a zoom motion. Pan motion is described by mouse motion when the right mouse button is down. Zoom is described by the mouse moving forwards or backwards when the middle mouse button is down.

The basis of the view space coordinate system is given by the current rotation of the camera. This rotation is specified by a quaternion $\hat{\boldsymbol{q}}_{\text{rot}}$. Moving the mouse while the left button is down will each frame provide a new mouse position to the track ball. This mouse position is projected to the sphere representing the virtual track ball. The sphere (or ball) is located at the center of the scene in view space, that is, the position the camera will always be pointing at. The new position found on the sphere will specify a new viewing direction $\boldsymbol{v}_2$. Suppose the previous viewing direction was stored as $\boldsymbol{v}_1$, then the quaternion $\hat{\boldsymbol{q}}_{\text{inc}}$ specifying the rotation from $\boldsymbol{v}_1$ to $\boldsymbol{v}_2$ is given by (9.4).

As described in section 5.1, eye point $\boldsymbol{E}$, 'look-at' point $\boldsymbol{L}$, and up vector $\boldsymbol{v}_{\text{up}}$ are sufficient to describe the camera orientation. If we choose the default orthonormal basis for the camera orientation $(\boldsymbol{e}_x, \boldsymbol{e}_y, \boldsymbol{e}_z)$, the camera eye point, 'look-at' point, and up vector are given as:

$$\begin{aligned}
\hat{\boldsymbol{v}}_{\text{up}} &= \hat{\boldsymbol{q}}_{\text{rot}} \hat{\boldsymbol{e}}_y \hat{\boldsymbol{q}}^*_{\text{rot}} \\
\hat{\boldsymbol{L}} &= t_y \hat{\boldsymbol{v}}_{\text{up}} + t_x \hat{\boldsymbol{q}}_{\text{rot}} \hat{\boldsymbol{e}}_x \hat{\boldsymbol{q}}^*_{\text{rot}} \\
\hat{\boldsymbol{E}} &= \hat{\boldsymbol{q}}_{\text{rot}} ((z_{\text{eye}} + t_z) \hat{\boldsymbol{e}}_z) \hat{\boldsymbol{q}}^*_{\text{rot}} + \hat{\boldsymbol{L}}
\end{aligned}$$

where each resulting quaternion corresponds to a vector or a point in homogenous coordinates.

Now, all we need to do in order to rotate the camera incrementally according to the mouse motion, is to calculate $\hat{\boldsymbol{q}}_{\text{rot}} := \hat{\boldsymbol{q}}_{\text{rot}} \hat{\boldsymbol{q}}_{\text{inc}}$ for each frame. We can even let the camera spin according to a previous motion after the

left mouse button has been released by storing the incremental quaternion $\hat{\boldsymbol{q}}_{\text{inc}}$. The spin stops when $\hat{\boldsymbol{q}}_{\text{inc}}$ is reset to quaternion identity.

The extension for this track ball is to freeze the camera when the user picks an object (eg. by pressing 'p' when the mouse is located over an object), and then let the mouse control the selected object instead of the camera. What we want to specify with the mouse is the modeling transform of the selected object.

In order to move an object intuitively with the mouse, the motion should be controlled in view space, since this is the space where the user works. The task is now to find the modeling transform in view space.

When the track ball is frozen we store the old view transformation matrix specified by $\boldsymbol{E}$, $\boldsymbol{L}$, and $\boldsymbol{v}_{\text{up}}$ (how to find the matrix from these three is described in section 5.1). If we let $\boldsymbol{M}_{\text{view}}$ denote the view transform, then a transformation $\boldsymbol{X}$ carried out in view space is given in world space as:

$$\boldsymbol{X}_{\text{world}} = \boldsymbol{M}_{\text{view}}^{-1} \boldsymbol{X}_{\text{view}} \boldsymbol{M}_{\text{view}} \tag{9.5}$$

Luckily the view space transformation consist of translation and rotation of the camera only, therefore we can find $\boldsymbol{M}_{\text{view}}^{-1}$ using (9.1).

Setting the center of the track ball to the center of the object in world space $\boldsymbol{C}_{\text{world}}$, we can specify the translation of the object in view space $\boldsymbol{T}_{\text{view}}$ according to pan and zoom of the track ball:

$$\boldsymbol{T}_{\text{view}} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where pan becomes moving the object parallel to the view plane and zoom becomes moving the object along the direction from the current camera position to the object center. This results in a quite intuitive translation of the object according to mouse movement. The next step is to rotate the object.

In order to rotate the object intuitively in view space we must first move it to the origin of the view space coordinate system. Having the center of the object in world space coordinates $\boldsymbol{C}_{\text{world}}$, we can transform it to view space coordinates as follows:

$$\boldsymbol{C}_{\text{view}} = \boldsymbol{M}_{\text{view}} \boldsymbol{C}_{\text{world}}$$

The translation is then simple because the origin of the view space coordinate system is now in $(0, 0, 0)$ relative to $\boldsymbol{C}_{\text{view}}$. Translation of the object to the origin of view space is:

$$\boldsymbol{T}_{\text{view},-\boldsymbol{C}} = \begin{pmatrix} 1 & 0 & 0 & -\boldsymbol{C}_{\text{view},x} \\ 0 & 1 & 0 & -\boldsymbol{C}_{\text{view},y} \\ 0 & 0 & 1 & -\boldsymbol{C}_{\text{view},z} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and translation back to the previous object position is:

$$\boldsymbol{T}_{\text{view},\boldsymbol{C}} = \begin{pmatrix} 1 & 0 & 0 & \boldsymbol{C}_{\text{view},x} \\ 0 & 1 & 0 & \boldsymbol{C}_{\text{view},y} \\ 0 & 0 & 1 & \boldsymbol{C}_{\text{view},z} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since we are looking at the object along the z-axis in view space, the local coordinate system around which we want to rotate the object will have a z-axis pointing in the opposite direction. If the object was positioned exactly in the 'look-at' point, the basis of the *object* coordinate system would be exactly opposite the basis of the *view space* coordinate system. Therefore, when we rotate the track ball, a reasonable approximation to a rotation of the selected object instead of the camera is given in view space as the conversion of $\hat{\boldsymbol{q}}_{\text{rot}}^{-1} = \hat{\boldsymbol{q}}_{\text{rot}}^{*}$ to a rotation matrix $\boldsymbol{M}^{q_{\text{rot}}^{*}}$ according to (9.3).

The rotation should as mentioned be performed at the origin of view space. The final transformation of the object in view space is, therefore, given as:

$$\boldsymbol{X}_{\text{view}} = \boldsymbol{T}_{\text{view}}\boldsymbol{T}_{\text{view},\boldsymbol{C}}\boldsymbol{M}^{q_{\text{rot}}^{*}}\boldsymbol{T}_{\text{view},-\boldsymbol{C}} \tag{9.6}$$

Inserting (9.6) in (9.5) results in the transform of the object in world space. If the object had no modeling transform to begin with. We can let $\boldsymbol{X}_{\text{world}}$ specify the new modeling transform of the object. For this project we will always assume that the object has no other modeling transform when this track ball motion control is applied.

This chapter has introductorily shown the impact of interaction and animation on real-time graphics. In the following chapter we will give a brief tutorial on Blender modeling, since one part of this project has been to show the work flow from modeling to rendering.

Bibliography

[1] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering.* A K Peters, Natick, Massachusetts, second edition, 2002.

[2] Erik B. Dam, Martin Koch, and Martin Lillholm. Quaternions, interpolation and animation. Technical Report DIKU-TR-98/5, Department of Computer Science, University of Copenhagen, July 1998.

[3] Jens Eising. *Lineær algebra.* Institut for Matematik, Danmarks Tekniske Universitet, 1997.

[4] Jens Gravesen. *Differential Geometry and Design of Shape and Motion.* Department of Mathematics, Technical University of Denmark, November 2002. Lecture notes for 01243.

[5] Ken Shoemake. Animating rotation with quaternion curves. *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 245–254, July 1985.