

# PSE Game Physics

Session (2)

Collisions: Sphere-Sphere, Sphere-Plane  
Interpenetrations

Martin Schreiber, Kristof Unterweger

23.04.2012



# Outline

Some very very basic math

Linear transformations

**Collisions**

Sphere-Sphere

Collision Normal

Collision points & Interpenetration depth

# Outline

## Some very very basic math

### Linear transformations

### Collisions

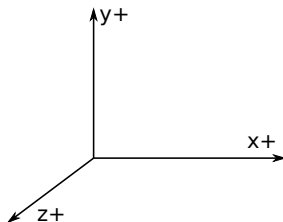
Sphere-Sphere

Collision Normal

Collision points & Interpenetration depth

# Right-handed system

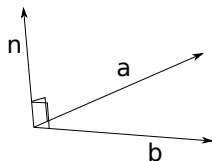
- Our **SBND-Engine** is written in a **Right-handed system** (OpenGL):



**Abbildung:** right-handed coordinate system

- Use your hand to determine the orientation: Thumb (X-axis), forefinger (Y-axis), middlefinger (Z-axis).
- Thus moving an object further away means **decreasing** the z-coordinate!
- **All formulae are given in left handed basis!**

# Cross product

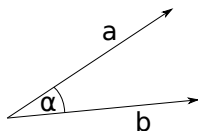


$$\vec{n} = \vec{a} \times \vec{b} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

- We mainly use it to compute a vector  $\vec{n}$  which is perpendicular to two given vectors  $\vec{a}$  and  $\vec{b}$
- E. g. if a plane is given by a single point and 2 vectors, the **plane normal** can be computed
- Remember: This formula is given in a **right-handed coordinate system!**

# Dot product

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^3 a_i b_i$$



**Abbildung:** dot product

- Usage: **Compute the cosine** of the angle created by  $\vec{a}$  and  $\vec{b}$ :

$$\vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cos(\alpha)$$

- Later on, we use it to compute the **projected length** of  $\vec{a}$  on a normal  $\vec{n}$

# Outline

Some very very basic math

## Linear transformations

## Collisions

Sphere-Sphere

Collision Normal

Collision points & Interpenetration depth

# Object properties

Video: NaturalMotion by Euphoria

<http://www.youtube.com/watch?v=Qi5adyccoKI>



# Object properties

- Position
- Orientation
- Velocity
- Mass
- ...

How can we express e. g. Position and Orientation?

# Degrees of Freedom for a rigid body object

”Modifications of Position and Orientation which we need for our physics engine with **rigid bodies**:

- **Translation**: Move object from position  $p(t)$  to  $p(t + \Delta t)$
- **Orientation**: Rotate an object  $q(t)$  to  $q(t + \Delta t)$

Other Modifications not necessary for our physics engine:

- **Shearing** (Meaningless for rigid body objects)
- **Scaling** (Meaningless for rigid body objects)
- **Projections** (Only necessary for visualization, non-affine transformation)

# How to describe the transformations?

## Model- and World-space:

- **Model-Space:** The *space* in which the object is setup without any modifications.
- The object is **moved/rotated** during initialization or by applying a linear or rotational velocity in each timestep.
- Applying this transformation projects the object **from model-space to world-space**.

## Linear transformations:

- Store the *position* of the object in a **vector**  $\vec{p}$ .
- Store the *orientation* of the object in a **quaternion**  $\vec{q}$ .
- Use a  $4 \times 4$  matrix  $M$  which is created based on the position and quaternion to describe the transformation.
- (for most cases, only a  $4 \times 3$  matrix is necessary)

# Translation

- A translation can be expressed by setting specific values in the last column of our 4x4 matrix  $M$ :

$$M_{\text{translate}} = \begin{pmatrix} 1 & & & t_x \\ & 1 & & t_y \\ & & 1 & t_z \\ & & & 1 \end{pmatrix}$$

- To allow a translation simply by applying the matrix  $M$  to a position, we extend the objects position  $\vec{p}$  by a fourth component which is set to 1 to the **homogenous vector**  $\vec{\bar{p}}$ .
- Example:  $\vec{\bar{p}} = (1, 3, -2, 1)^T$ ,  $t_x = (10, 10, 10)^T$

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & . & . & t_x \\ . & 1 & . & t_y \\ . & . & 1 & t_z \\ . & . & . & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 + t_x \\ 3 + t_y \\ -2 + t_z \\ 1 \end{pmatrix} = \begin{pmatrix} 11 \\ 13 \\ 8 \\ 1 \end{pmatrix}$$

# Orientation

- One way to express 3D Orientation is to combine 2D rotations.
- E. g. 2D rotation around Z-Axis:

$$M_{rotate_z} = \begin{pmatrix} \cos \alpha & -\sin \alpha & \cdot & \cdot \\ \sin \alpha & \cos \alpha & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{pmatrix}$$

- Combining 3 rotation matrices (Around X, Y and Z axis), we can express all possible rotations.

# Putting everything together

- Now we can express rigid body transformations by **applying all transformations to the object's position** (or at any point on the object's surface):

$$M_{translate} \cdot M_{rotate} \cdot \vec{p}$$

- By setting  $M = M_{translate} \cdot M_{rotate}$  the transformation can be expressed by only a **single matrix-vector expression**:

$$M \cdot \vec{p}$$

- Model matrix  $M$  gives the transformation of an object **from mode-space to world-space**.
- (Model matrix is also called world matrix when programming with DirectX).

# Inverse Transformations

- It's also possible to **transform objects from world-space to the model-space**.
- This becomes handy when **intersection tests** can be easily **handled in the model-space** of an object (e. g. ball-box intersection tests)
- The transformation of an object  $O_1$  to the model-space of another object  $O_2$  can be simply handled by using the inverse of the **model-matrix**:

$$M_2^{-1} \cdot M_1$$

- Hint for a better understanding: **read this formula from right to left**
  - Firstly, object 1 is transformed to the world space
  - By applying the inverse transformation for object 2, transforms object 1 to the model-space of object 2

# How to transform Vectors?

- A **vector** at an arbitrary position **cannot be transformed simply by applying the model-matrix!!!**
- To transform a vector  $\vec{v}$  from one space to another given the point transformation matrix  $M$ , the vector  $\vec{v}$  can be either transformed by
  - Applying  $M$  to the origin point as well as to the vector  $\vec{v}$  and subtracting both values to get the appropriate solution or
  - by applying the **inverse transpose** of  $M$  (see next slide)



## Inverse-transpose of a matrix (1/2)

- Here we give a short sketch (See <http://www.faqs.org/faqs/graphics/algorithms-faq/> for more details)
- We start by **splitting up** the matrix  $M$  into the 3x3 **matrix**  $L$  and the **translation vector**  $\vec{t}$ :

$$M = \left( L \mid \vec{t} \right)$$

- Assuming that  $\vec{p}$  and  $\vec{q}$  lie on an object's surface, the vector connecting both points is defined by  $\vec{v} = \vec{p} - \vec{q}$ .
- Then we can describe the transformation of an arbitrary vector  $\vec{v}$  connecting with the transformation of its starting and end point  $\vec{p}$  and  $\vec{q}$ :

$$\vec{v}' = (L\vec{p} + \vec{t}) - (L\vec{q} + \vec{t}) = L(\vec{p} - \vec{q}) = L(\vec{v})$$

## Inverse-transpose of a matrix (2/2)

- When transforming two vectors, the **angle remains the same** which can be expressed by a **dot-product**.
- In this case, our second vector is represented by a **normal**  $\vec{n}$  on the **surface point**  $\vec{q}$ .
- Since we know, that  $\vec{v}$  can be transformed by applying  $L$ , we can rewrite the equation to

$$\vec{n}' \cdot \vec{v}' = (N\vec{n}) \cdot (L\vec{v})$$

with  $N$  being the matrix we like to figure out.

- By replacing the dot product using the transposed:

$$\vec{n}' \cdot \vec{v}' = (N\vec{n})^T (L\vec{v}) = \vec{n}^T N^T L \vec{v} = \vec{n}'^T \vec{v}'$$

- Since the angle between the 2 vectors  $\vec{n}$  and  $\vec{v}$  also **have to be equal in model-space**, we can assume that  $N^T L = I$ .
- Thus, we get  $N = L^{-T}$  for the matrix **transforming vectors** to the basis given in  $M$ .

# Overview

- So far we are able to **mathematically describe transformations from model- to world-space**.
- Use homogenous coordinates  $(x_1, x_2, x_3, 1)$  for **translations** and position.
- Use homogenous coordinates  $(x_1, x_2, x_3, 0)$  for **vectors**.
- Use **transposed inverse matrices  $M^{-T}$**  to transform vectors.
- Use **inverse  $M^{-1}$**  of transformation matrix to **transform "backward"**.

# Overview

- So far we are able to **mathematically describe transformations from model- to world-space**.
- Use homogenous coordinates  $(x_1, x_2, x_3, 1)$  for **translations** and position.
- Use homogenous coordinates  $(x_1, x_2, x_3, 0)$  for **vectors**.
- Use **transposed inverse matrices  $M^{-T}$**  to transform vectors.
- Use **inverse  $M^{-1}$**  of transformation matrix to **transform "backward"**.
- Seems to be easy, isn't it? **Not for rotations!**

# Outline

Some very very basic math

Linear transformations

## Collisions

Sphere-Sphere

Collision Normal

Collision points & Interpenetration depth

# Our schedule for collision tests

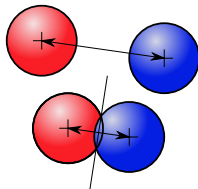
- Collision tests:

	Plane	Sphere	Box
Plane	o	-	-
Sphere	a	X	-
Box	a	X	X

- o: meaningless
- a: not in this session, but available via additional slides.
- -: Symmetric
- Resolving interpenetration

# Collision: Sphere-Sphere

- Really easy to detect this collision:  
Test whether the **distance of the 2 center points is less than the sum of both radii**
- But we need more collision data in order to handle collisions...  
(next slide)



# Intersection data

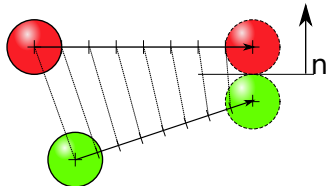
Which intersection data has to be generated generated?

- Collision normal
- Collision points
- Interpenetration depth



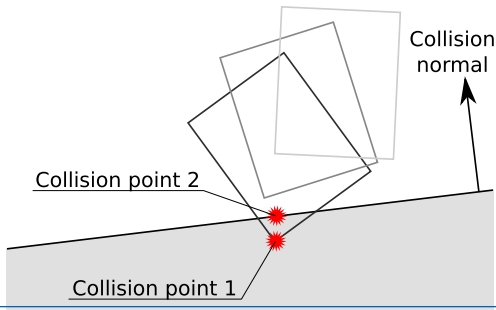
# Collision Normal

- We need to know in which **direction** we have to **apply some force** due to the collision.
- For spheres we assume to determine the collision normal when the 2 spheres are very close.
- Thus the normal from the first sphere aiming to the other one is given by computing the **collision normal** based upon **both sphere's centers**.



# Collision points & Interpenetration depth

- Since an interpenetration can occur during a timestep (not only at the very end), the collision point for every object has to be approximated.
- This collision point gets important when **applying an impulse** to an object which results in a angular momentum.
- The distance of those collision points also gives us the **interpenetration depth**.



## Resolving Interpenetration (1/2)

- If a **binary-collision for 2 objects was detected**, this interpenetration has to be solved somehow
- The first assumption is, that the interpenetration can be solved by moving the objects **parallel to the collision normal**
- Question: **How far** should every object be moved?

## Resolving Interpenetration (2/2)

- We start by considering **2 extreme cases** assuming that resolving the interpenetration depends on the rigid-body mass
  - O1 has  $\infty$  mass, O2 has mass 1  $\Rightarrow$  Only O2 is moved
  - O2 has  $\infty$  mass, O1 has mass 1  $\Rightarrow$  Only O1 is moved
- Linearly interpolating inbetween these 2 extreme cases, the fraction  $d_1$  of the interpenetration distance for O1 can be computed by:

$$d_1 = \frac{m_2}{m_1 + m_2} = \frac{m_1^{-1}}{m_1^{-1} + m_2^{-1}}$$

with  $m_i^{-1}$  being the inverse mass of the corresponding objects.