Institut für Informatik — TU München
Scientific **C**omputing in **C**omputer **S**cience
O. Meister, R. Wittmann

SS 2015

Bachelor Lab Scientific Computing (Game Physics) Worksheet 3: Springs, Ropes, Linear momentum & Quaternions

Assignment 1: Springs

File: *iPhysicsSoftConstraintSpring.cpp*,
Method: *cPhysicsSoftConstraintSpring::updateAcceleration*
Springs belong to the so called *weak constraints* connecting 2 rigid body objects. Depending on the length of the spring, a force is applied to both attached objects at the point of attachment. To stay simple, you only have to apply the force acting on the middle of the sphere. Use Hook's law

$$\mathbf{F} = -k \cdot \mathbf{d}$$

with $\mathbf{F}$ being the applied force, k as the spring coefficient and $\mathbf{d}$ as the distance between both objects to update the acceleration accumulator for both objects. Don't forget to insert a call to apply the soft constraint for all objects (*updateSoftConstraints*) in the method *simulationTimestep()*.

Hint 1: Use the mass of the object to compute the acceleration.
Hint 2: The same force is applied to both objects (actio = reactio). Test your simulation by looking at scene 7.

Assignment 2: Linear momentum

File: *cPhysicsCollisionImpulse.hpp*,
Method: *cPhysicsCollisionImpulse::applyCollisionImpulse*
After detecting a collision, an impulse has to be applied to the object. Since we don't know rotations yet, we stick to the simple linear case, applying the impulse to the center of mass. Further information about how to apply the impulse to each object is given in the presentation slides.

Additionally you will have to insert a call to *applyCollisionImpulse()* in the method *simulationTimestep()*.

Assignment 3: Ropes

File: *iPhysicsHardConstraintRope.cpp*,
Method: *cPhysicsHardConstraintRope::updateHardConstraintsCollisions*
Ropes belong to the *hard constraints*. Similar to an impulse, they can be implemented by applying an impulse created by a negative interpenetration. Therefore, they are implemented by creating collision data. Open the corresponding file and fill in the necessary code to create an appropriate collision information.

Similar to the previous assignment, you also need to execute the handling of the hard constraints for each timestep. Insert a call to *getHardConstraint-Collisions()* in the method *simulationTimestep()*. This method computes more hard constraints which are used by the upcoming methods to resolve the interpenetrations and to apply impulses. Test your simulation by looking at scene 8.

Assignment 4: Rotations with Quaternions

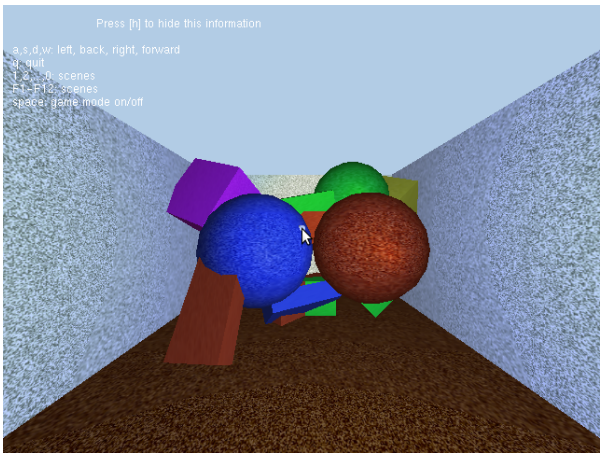
As explained during the lab session, Quaternions are useful to avoid *Gimbal locks*. Since all rotations within the engine are implemented using Quaternions now it's up to you to implement this class.

File: *CQuaternion.hpp*,
Method: *CQuaternion::setRotation*
Implement the method to setup the quaternion scalars to fit to the given rotational parameters.

File: *CQuaternion.hpp*,
Method: *CQuaternion::getRotationMatrix*
Implement the method *getRotationMatrix* which computes a matrix based on the quaterion scalars.

File: *CQuaternion.hpp*,
Method: *CQuaternion::operators**
Implement the method *multiplication operators* which multiply 2 quaternions.

To test your implementation, the first scene should look like this:



Good luck,

Roland & Oliver