Simox: A Simulation and Motion Planning Toolbox for C++

Nikolaus Vahrenkamp, Tamim Asfour and Rüdiger Dillmann

Humanoids and Intelligence Systems Laboratories (HIS)
Institute for Anthropomatics
Karlsruhe Institute of Technology (KIT)

Email: vahrenkamp@kit.edu

Version 1.1

Contents

Chapter 1

Introduction

Simox is a lightweight platform independent C++ toolbox, containing three libraries for 3D simulation of robot systems, sampling based motion planning and grasp planning. The library *Virtual Robot* is used to define complex robot systems, which may cover multiple robots with many degrees of freedom. The robot structure and it's visualization can be easily defined via XML files and environments with obstacles and objects to manipulate are supported. Basic robot simulation components, as Jacobian computations and generic IK-solvers, are offered by the library. The two libraries *Saba* and *Grasp Studio* host algorithms related to motion and grasp planning. State-of-the-art implementations of sampling-based motion planning algorithms (e.g. Rapidly-exploring Random Trees) are served by the *Saba* library, which was designed for efficient planning in high-dimensional configuration spaces. The possibility to exchange the underlying collision detection library allows to customize the planning framework and due to the multi-threading support efficient planning concepts can be realized. *Grasp Studio* offers possibilities to compute the grasp quality for generic end-effector definitions, e.g. a humanoid hand. The implemented 6D wrench-space computations can be used to easily (and quickly) determine the quality of an applied grasp to an object. Furthermore, the implemented planners are able to generate grasp maps for given objects automatically.

Chapter 2

Virtual Robot

In this chapter, an introduction to the simulation library *Virtual Robot* is given. It is shown how to create a simple robot system by defining an XML structure and how to visualize the results. Convenient methods for accessing the robot are presented and several features of the library, as Jacobian calculations or IK-solving, are discussed.

2.1 Robots

2.1.1 Defining a robot

Simox uses a custom XML format for defining robot systems which is easy to understand and mostly self-explanatory. A robot consists of multiple so-called *Robot Nodes* which are linked together and which may contain visualizations or not. In case a *Robot Node* does not contain any visualization it is used as a virtual joint (i.e. such a node can be used as a virtual coordinate system). The *Robot Nodes* hold information about children, Denavit-Hartenberg (DH) Parameters (or simple translation + rotation parameters), name, visualization and collision models. The 3D models are defined via OpenInventor files (see [1]). An simple example of a robot with three degrees of freedom is given below.

Listing 2.1: SimpleRobot.xml

```xml
<?xml version="1.0"?>
<Robot>
    <Type value="SimpleRobot"/>

    <RootJoint>
        <Name value="DemoRobot"/>
        <ChildNode name="Joint1"/>
    </RootJoint>

    <ChildJoint>
        <Name value="Joint1"/>
        <DH>
            <alpha value="90"/>
            <thetaJoint value="1"/>
            <a value="300"/>
        </DH>
        <Visualisation>
            <IVModel file="joint_rot_sphere.iv"/>
        </Visualisation>
        <CollisionChecking>
            <IVModel file="joint_rot_sphere.iv"/>
        </CollisionChecking>
        <ChildNode name="Joint2"/>
    </ChildJoint>
```

```xml
    <ChildJoint>
        <Name value="Joint2"/>
        <DH>
            <thetaJoint value="1"/>
            <a value="300"/>
        </DH>
        <Limits min="-90" max="45"/>
        <Visualisation>
            <IVModel file="joint_rot_sphere.iv"/>
        </Visualisation>
        <CollisionChecking>
            <IVModel file="joint_rot_sphere.iv"/>
        </CollisionChecking>
        <ChildNode name="TCP"/>
    </ChildJoint>

    <ChildJoint>
        <Name value="TCP"/>
    </ChildJoint>
</Robot>
```

Since multiple instances of the robot type are allowed, the robot definition does not include a name, but a type (*DemoRobot*) and the name of the instance is defined when loading (or cloning) the robot. The kinematic structure of a robot can be seen as a tree of connected joints, for that a specific root node defines the start. In the given example, the robot definition consists of four nodes, but only two of them define a movable joint (*Joint1* and *Joint2*). These node definitions include DH parameters (a, d, theta, alpha) where non given arguments are set to zero. The parameter *thetaJoint* indicates that the joint is movable and the theta value describes the flexible part of the joint (other DH parameters can also be flexible, e.g. a joint with the DH-definition $< dJoint\,value = "1"/>$ would result in a translational joint). The allowed movement of node *Joint2* is limited from -90 to 45 degrees. Examples of several robot definitions are given in Fig. 2.1.
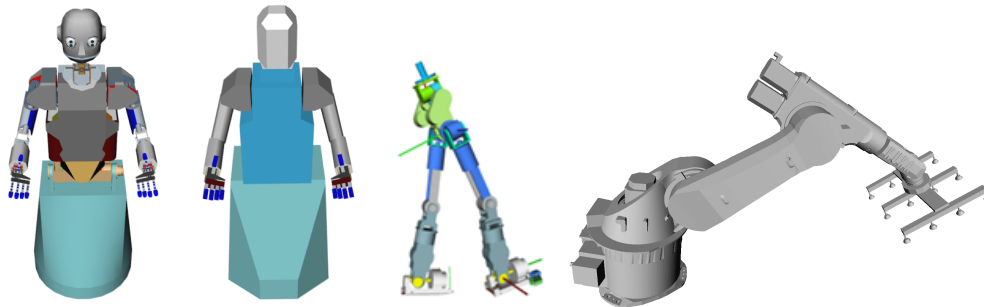
Figure 2.1: Full and reduced model of the humanoid robot ARMAR-III. The robot consists of 94 RobotNodes with 43 degrees of freedom. Furthermore humanoid legs and a $Kuka^{©}$ KR60-3 are depicted.

2.1.2 Accessing a robot

Once a robot is defined and the definition is stored in an XML file, it can be read by the XML parser and in case, the robot definition is valid, an instance of the robot is created as shown in Listing 2.2.

Listing 2.2: Loading a robot

```cpp
std::string sFilename("demoRobot.xml");
std::string sInstanceName("MyRobot");
CRobot *pRobot = CRobot::Load(sFilename, sInstanceName);
```

In Listing 2.3, *pRobot*, an instance of *CRobot*, which is the main class of the *Virtual Robot* library, is constructed (this implies loading of all 3D models). The code snippet shows how to get a visualization of the robot and several ways of setting and retrieving configurations are shown.

Listing 2.3: Accessing a robot

```cpp
// get visualization
SoSeparator *pVisuFullModel = pRobot->GetFullIVModel();
SoSeparator *pVisuColModel = pRobot->GetCollisionIVModel();

// set joint values
std::string sJoint1("Joint1");
std::string sJoint2("Joint2");
pRobot->SetJoint(sJoint1, M_PI / 2.0);
pRobot->SetJoint(sJoint2, -0.1f);
pRobot->ApplyJointValues();

// get joint values
float fJointValue1 = pRobot->GetJointValue(sJoint1);
CRobotNode *pNode2 = pRobot->GetNode(sJoint2);
float fJointValue2 = pNode2->GetJointValue();
float fLimitLo = pNode2->GetJointLimitLo(); // -PI/2
float fLimitHi = pNode2->GetJointLimitHi(); // PI/4
```

You can find an example on how to load, access and display a robot at *Simox/Virtual-Robot/examples/RobotViewer*.

2.1.3 Kinematic Chains

A kinematic chain is a collection of names of previously defined *RobotNodes* which can be used to access multiple joints at once. The definition of the *StartNode* and *TCP* are optional. The start node defines the node in the kinematic structure from which an update of the joint poses has to be done in case the joint values of the kinematic chain change. This can be useful when large robot systems are built, and only a small sub-part of the robot needs to be updated when joint values change. You may think of a humanoid robot and when updating the joint values of the right arm, the first joint, from where the re-calculation of joint poses has to be done is the right shoulder joint. Hence, a complete re-calculation of all joint poses can be avoided when accessing the kinematic chain. The *TCP* node can be used in case Jacobian-based movements are applied (see section 2.4). Note, that a kinematic chain is just a collection of nodes, which means you can also define sets of nodes which do not form a valid kinematic chain.

Listing 2.4: Definition of a kinematic chain

```xml
<Robot>
    ...
    <KinematicChain>
        <Name value="Rightarm"/>
        <StartNode name="Shoulder1 R"/>
        <Node name="Shoulder1 R"/>
        <Node name="Shoulder2 R"/>
        <Node name="Upperarm R"/>
        <Node name="Elbow R"/>
        <Node name="Underarm R"/>
        <Node name="Wrist1 R"/>
        <Node name="Wrist2 R"/>
        <TCP name="TCP R"/>
    </KinematicChain>
</Robot>
```

The kinematic chain can be accessed as shown in Listing 2.5.

Listing 2.5: Accessing a kinematic chain

```
std::string sKinName("Rightarm");
CKinematicChain* pKinChain = pRobot->GetKinematicChain(sKinName);

// set values given as std::vector
pRobot->SetJointValues(vValues, sKinChain);
```
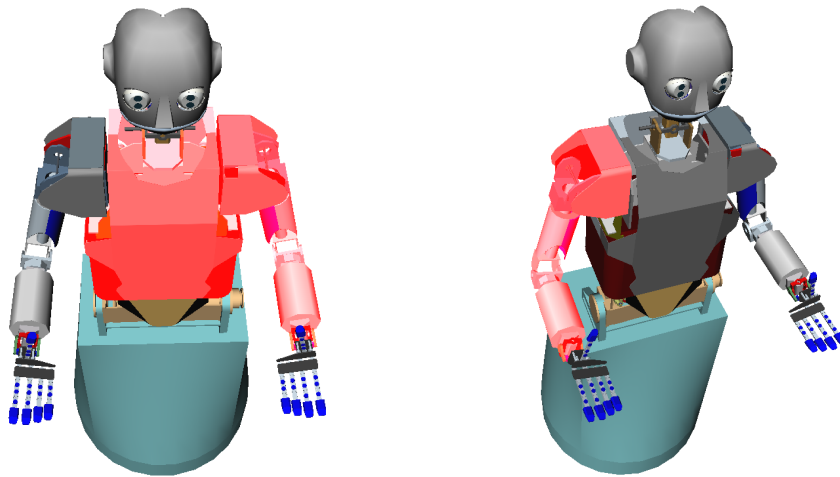

Figure 2.2: Two kinematic chains defined for the humanoid robot ARMAR-III [2].

2.1.4 Collision Models

A convenient way of defining a set of collision models is offered by the tag *CollsionModel*. The collections of collision models can be used for easily testing parts of the robot for collisions with obstacles or other parts of the robot (see section 2.3).

Listing 2.6: Collection of collision model definitions

```
<Robot>
    ...

  <CollisionModel>
    <Name value="Left Arm"/>
    <Node name="Upperarm L"/>
    <Node name="Underarm L"/>
    <Node name="Wrist2 L"/>
    <Node name="Hand Palm1 L"/>
    <Node name="Hand Palm2 L"/>
    <Node name="Pinky L J0"/>
    <Node name="Pinky L J1"/>
    <Node name="Ring L J0"/>
    <Node name="Ring L J1"/>
    <Node name="Middle L J0"/>
    <Node name="Middle L J1"/>
    <Node name="Index L J0"/>
    <Node name="Index L J1"/>
    <Node name="Thumb L J0"/>
    <Node name="Thumb L J1"/>
  </CollisionModel>
</Robot>
```

2.1.5 End-Effectors

By defining end-effectors (EEF), more complex robots or manipulators can be realized, allowing to simulate manipulation or grasping actions.

To use an end-effector, at first it has to be specified in the XML file of the robot. Here the EEF definition uses previously defined *RobotNodes* of the robot, so that an end-effector definition is a logical collection of robot joints. The EEF definition includes a name, a base node (which can be seen as the TCP node of the end-effector), a static part (e.g. the palm of a hand) and several fingers. Fingers could also define two ends of a parallel gripper.

Listing 2.7: End-effector definition of a humanoid hand

```xml
<Robot>

    ...

    <EndEffector>
        <Name value="Left Hand"/>
        <BaseNode name="TCP L"/>

        <StaticPart>
            <Node name="Hand Palm1 L"/>
            <Node name="Hand Palm2 L"/>
        </StaticPart>

        <Finger>
            <Name value="Thumb Left"/>
            <Node name="Thumb L0" ColChecking="0"/>
            <Node name="Thumb L1" ColChecking="1"/>
        </Finger>
        <Finger>
            <Name value="Index Left"/>
            <Node name="Index L0" ColChecking="0"/>
            <Node name="Index L1" ColChecking="1"/>
        </Finger>
        <Finger>
            <Name value="Middle Left"/>
            <Node name="Middle L0" ColChecking="0"/>
            <Node name="Middle L1" ColChecking="1"/>
        </Finger>
        <Finger>
            <Name value="Ring Left"/>
            <Node name="Ring L0" ColChecking="0"/>
            <Node name="Ring L1" ColChecking="1"/>
        </Finger>
        <Finger>
            <Name value="Pinky Left"/>
            <Node name="Pinky L0" ColChecking="0"/>
            <Node name="Pinky L1" ColChecking="1"/>
        </Finger>
    </EndEffector>
</Robot>
```

This end-effector can then be accessed in your code as shown below. The EEF can be closed and opened, with and without considering obstacles (see section 2.2). The contact information can be collected, e.g. for measuring the grasp quality (see chapter 4). Fig. 2.3 shows an anthropomorphic hand (see [3]) grasping several objects.

Listing 2.8: Accessing an end-effector

```cpp
std::string sEEFName("Left Hand");
CEndEffector* pEEF = pRobot->GetEndEffector(sEEFName);

pEEF->CloseHand();
pEEF->OpenHand();
pEEF->CloseHand(pObstacle);
pEEF->Highlight();
pEEF->CloseHandContactInfo(vContactInformation, pObstacle);
```
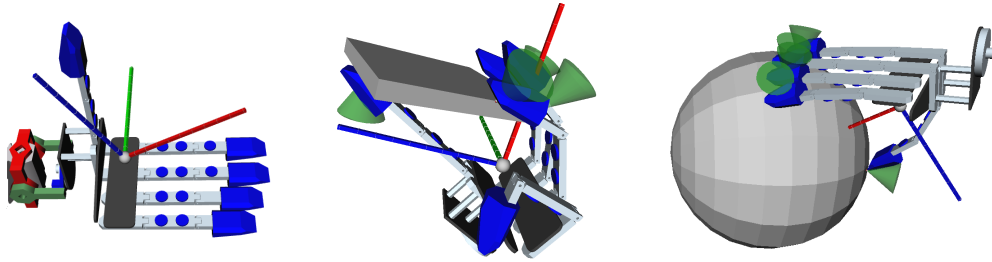
Figure 2.3: The humanoid hand of ARMAR-III [3].

2.1.6 Coordinate Transformations

A robot holds information about poses of all registered joints. This information can be used to transform coordinates between the different coordinate systems of the robot and the world.

Listing 2.9: Coordinate transformations

```
std::string sCoord1("Joint1");
std::string sCoord2("Joint2");
SbMatrix mMat;
mMat.makeIdentity();

// transform pose from local coord system of
// Joint1 to local coord system of Joint2
pRobot->TransformPose(sCoord1,sCoord2,mMat);

// transform pose from world coord system
// to local coord system of Joint1
pRobot->TransformGlobalPose(sCoord1,mMat);

// transform pose from local coord system to
// world coord system
pRobot->TransformToGlobalPose(sCoord2,mJointPose);

// get pose of joint in world coord sytem
CRobotNode *pNode = pRobot->GetNode(sCoord2);
SbMatrix mJointPose = *(pNode->GetPose());
```

In Listing 2.10 the visualization of a coordinate system is enabled and several coordinate systems are shown in Fig. 2.4 .

Listing 2.10: Visualization of coordinate axis

```
<Robot>

    ...

    <ChildJoint>
    ...
        <Visualisation>
            <CoordinateAxis enable="1" scaling="1"/>
        </Visualisation>
    </ChildJoint>
</Robot>
```
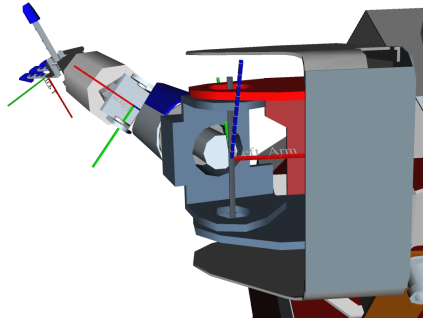
Figure 2.4: Several coordinate systems of the left arm of ARMAR-III.

2.2 Environment, Objects and Scenes

In the previous section it was showed how to define and load a robot and now the surrounding of the robot is discussed. Environments and obstacles are defined as OpenInventor [1] models and it is possible to define two models, one for visualization and the other for collision checking. Together with scene definitions, a complete setup can be realized containing robots, configurations, obstacles and environments.

Listing 2.11: Defining an environment and obstacles

```cpp
std::string sFullFile("EnviornmentHighDef.iv");
std::string sCollisionModelFile("EnvironmentReduced.iv");
CEnvironment *pEnv = CEnvironment::LoadEnvironment(sFullFile);
CEnvironment *pEnv2 = CEnvironment::LoadEnvironment(sFullFile,
    sCollisionModelFile);

// create a standard box
CManipulationObject *pObjBox =
    CManipulationObject::CreateObstacleBox(100,100,100);

// load an object form file
CManipulationObject *pObj1 = new CManipulationObject();
pObj1->LoadIVModel("object.iv");

// move object around
SbMatrix mPose1;
mPose1.setTranslate(SbVec3f(1000,0,0));
pObj1->SetGlobalPose(mPose1);

// add object to environment
pEnv->AddManipulationObject(pObj1);
```
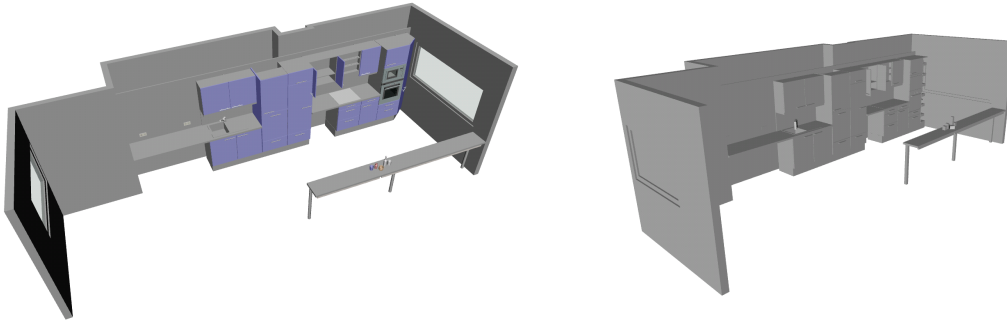
Figure 2.5: Loading environments: Full and a reduced model of a kitchen. This kitchen model was created within the German Collaborative Research Center *Humanoid Robots - Learning and Cooperating Multimodal Robots* (CRC 588) [4].

2.2.1 Grasping Information

The class *CManipulationObject* can be used for defining obstacles (see above), or additional information can be stored as feasible grasps related to an end-effector. Therefore the classes *CFeasibleGrasp* and *CFeasibleGraspCollection* are used. The grasping information can be generated using the tools offered by *GraspStudio*, see chapter 4 for details. The grasping information can be read from and stored to XML files:

Listing 2.12: Definition of a ManipulationObject

```
<ManipulationObject>
    <Name value="cup"/>
    <InventorFilename value="cup.iv"/>

    <FeasibleGraspCollection>
        <Name value="GraspCollectionRight"/>
        <ManipulationObject value="cup"/>
        <Robot value="Armar3"/>
        <EndEffector value="Right Hand"/>
        <FeasibleGrasp>
            <Name value="Grasp Right 0"/>
            <Pose>
                <Row1 m1="1.0" m2="0" m3="0" m4="0"/>
                <Row2 m1="0" m2="0" m3="1.0" m4="0"/>
                <Row3 m1="0" m2="-1.0" m3="0" m4="0"/>
                <Row4 m1="-7" m2="33" m3="-2" m4="1"/>
            </Pose>
        </FeasibleGrasp>
    </FeasibleGraspCollection>
</ManipulationObject>
```

The grasp information can be accessed via the appropriate methods of *CManipulationObject*. Furthermore, the grasping poses can be used to put an object in the hand of the robot.

Listing 2.13: Accessing the grasping information

```
std::string sObjectFile("CupObject.xml");
CManipulationObject *pObj = CManipulationObject::Load(sObjectFile);

// get grasp information
std::string sEEF("Right Hand");
CFeasibleGraspCollection* pGraspCollection = pObj->GetFeasibleGrasps(sEEF);

std::string sEEF("Grasp Right 0");
CFeasibleGrasp *pGrasp = pGraspCollection->GetGraspConfig(sGraspname);
```

```cpp
// get grasping pose of EEF in world coord system when applying the grasp
SbMatrix mPose = pObj->GetGlobalGraspingPose(pGrasp);

// get visualization of grasping pose
SoSeparator* pGraspVisu = pObj->GetGraspVisu(pRobot, pGrasp);

// set object to grasping position
std::string sLeftHand("Left Hand");
CEndEffector *pEEF = pRobot->GetEndEffector(sLeftHand);
CRobotNode* pBaseNode = pEEF->GetBaseNode();
SbMatrix mEefPose = *(pBaseNode->GetPose());
SbMatrix mObjectPose = *(pGrasp->GetObjectPoseInHandFrame());
mObjectPose.multRight(mEefPose);
// now, mObjectPose is the global pose of the object when pGrasp is applied
pObj->SetGlobalPose(mEefPose);

// close the hand to grasp the object
pEEF->CloseHand(pObj->GetCollisionModel());

// attach object to robot's end-effector (at current pose)
// and add object to collision model of left arm
std::string sLeft("Left Arm");
CRobotCollisionModelCollection *pColModel = pRobot->GetCollisionModel(sLeft);
std::string sNodeName = pBaseNode->GetName();
pRobot->AttachObject(sNodeName, pObj, pColModel);
```
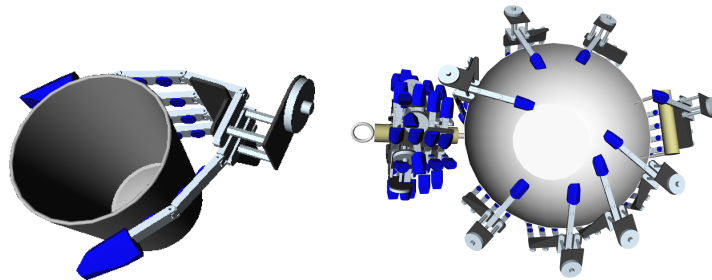

Figure 2.6: Grasping definitions for two objects and the right hand of the humanoid robot ARMAR-III.

2.2.2 Scenes

Scenes offer a convenient way of defining and storing a situation.

Listing 2.14: A scene definition in XML

```xml
<Scene>
    <Robot filename = "armar3.xml" name="ArmarIII" configuration="init">
        <Configuration name="init">
            <item name="X_Platform" value="1700.0"/>
            <item name="Y_Platform" value="-1500.0"/>
            <item name="Yaw_Platform" value="0.0"/>
            <item name="Elbow L" value="0.0"/>
            <item name="Shoulder1 L" value="0.0"/>
            <item name="Underarm L" value="0.0"/>
            <item name="Elbow R" value="0.0"/>
            <item name="Shoulder1 R" value="0.0"/>
            <item name="Shoulder2 R" value="0.0"/>
            <item name="Underarm R" value="0.0"/>
        </Configuration>
        <Configuration name="Pose1">
            <item name="Elbow L" value="1.57"/>
            <item name="Shoulder1 L" value="-0.2"/>
```

```xml
                <item name="Underarm L" value="0.43"/>
                <item name="Elbow R" value="0.1"/>
            </Configuration>
    </Robot>

    <ManipulationObject filename="l.iv" newName="lamp" configuration="StartPose">
        <Configuration name="StartPose">
            <item name="x" value="4830.0"/>
            <item name="y" value="390.0"/>
            <item name="z" value="0.0"/>
            <item name="roll" value="0.0"/>
            <item name="pitch" value="0.0"/>
            <item name="yaw" value="0.0"/>
        </Configuration>
    </ManipulationObject>
    <ManipulationObject filename = "Table.xml" configuration="init">
        <Configuration name="init">
            <item name="x" value="3000.0"/>
            <item name="y" value="1800.0"/>
            <item name="z" value="840.0"/>
            <item name="roll" value="0.0"/>
            <item name="pitch" value="0.0"/>
            <item name="yaw" value="0.7"/>
        </Configuration>
    </ManipulationObject>
    <Environment filename = "kitchen.iv">
    </Environment>
</Scene>
```

Listing 2.15: Accessing the scene information

```cpp
std::string  sFile("scene.xml");
CScene *pScene = CSceneIO::Read(sFile);

// get visualisation
SoSeparator* pSep = pScene->GetSceneSep();

// get robot
std::string  sRobot("ArmarIII");
CRobot* pRobot = pScene->getRobot(sRobot);

// get environment
CEnvironment* pRobot = pScene->GetEnvironment();

// get manipulation objects
std::vector<CManipulationObject*> vObjects;
pScene->GetAllManipulationObjects(vObjects);
```

2.3   Collision Detection

Simox offers an interface for performing collision checks and distance calculations on 3D models. Three classes directly interfere with the underlying collision checker (*CCollisionChecker, CCollisionModel* and *CCollisionModelCollection*), which offers the possibility to exchange the used collision detection library in case another implementation is preferred. Currently collision detection and distance calculations are done by the PQP library which uses OOBB and swept sphere volumes for efficient and accurate collision detection and distance computation (see [5]).

There is a global instance of the collision checker (accessible with *CCollisionChecker::GetGlobalCollisionChecker()*) and in case you are not doing concurrent collision detection in separate threads, this instance is all you need. In multi-threading applications you have to create an instance of *CCollisionChecker* for each thread (see next section for details).

Collisions and distances between 3D models can be determined by calling the methods *Check-Collision()* and *CalculateDistance()*. The arguments for calling these methods can be of type *CCollisionModel* or *CCollisionModelCollection*. By defining collections of collision models, logical sets of collision models can be addressed in a convenient way (e.g. an arm of a robot) and collisions are only reported between the collections (i.e. collisions of models which belong to the same collection are ignored).

Listing 2.16: Collision detection and distance calculation

```cpp
// get global collision checker instance
CCollisionChecker *pColChecker = CCollisionChecker::GetGlobalCollisionChecker();

// get collision model of one joint
std::string sNode("Joint1");
CRobotNode *pNode = pRobot->GetNode(sNode);
CCollisionModel *pColModelJoint = pNode->GetCollisionModel();

// get collision model of a manipulation object
CCollisionModel *pColModelObject = pManipulationObject->GetCollisionModel();

bool bCollision = pColChecker->CheckCollision(pColModelJoint, pColModelObject);
float fD = pColChecker->CalculateDistance(pColModelJoint, pColModelObject);

// get a set of collision models
std::string sLeft("Left Arm");
CRobotCollisionModelCollection *pColRob = pRobot->GetCollisionModel(sLeft);

CCollisionModelCollection* pColModelEnv = pEnvironment->GetCollisionModel();
bool bCollision2 = pColChecker->CheckCollision(pColRob, pColModelEnv);
```
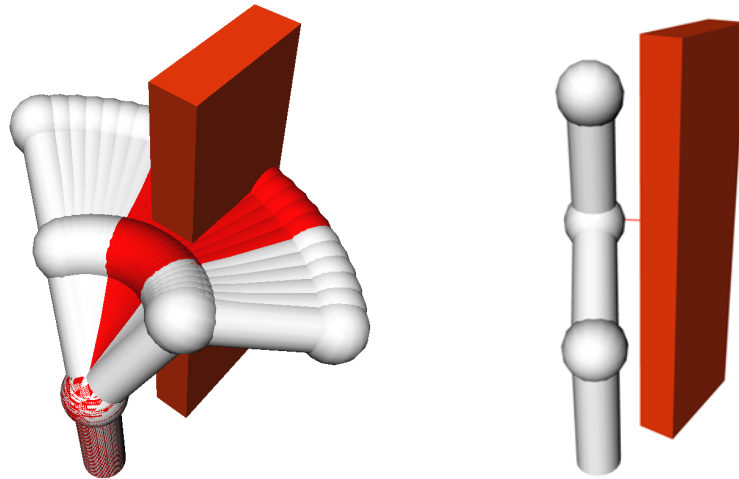

Figure 2.7: Collision detection performed on several configurations of a simple three DoF robot (left). The distance calculation methods offer the possibility to retrieve the points with shortest distance between two models (right).

2.3.1 Multi-Threading

When you plan to implement a multi-threaded application where the routines of the collision checker are called in parallel, you will need multiple instances of the collision checker and multiple instances of the robot(s) and object(s) you operate on. Each robot or object is linked to an instance of the collision checker which can be specified on construction (when no instance is set on construction, the global collision checker is used automatically). An easy way of creating multiple instances of objects and robot is offered by the *Clone()* methods which takes a pointer to a new collision checker as parameter and creates a complete copy of the robot or object that is linked to the given instance of the collision checker.

Listing 2.17: Using multiple collision checkers

```cpp
// create two instances of collision checker
CCollisionChecker *pColCheckerA = new CCollisionChecker();
CCollisionChecker *pColCheckerB = new CCollisionChecker();

// clone robot
std::string sNewNameA("Robot A");
std::string sNewNameB("Robot B");
CRobot *pRobotA = pRobot->Clone(pColCheckerA, sNewNameA);
CRobot *pRobotB = pRobot->Clone(pColCheckerB, sNewNameB);

// clone object
std::string sObjA("Object A");
std::string sObjB("Object B");
CManipulationObject* pObjA = pManipulationObject->Clone(pColCheckerA, sObjA);
CManipulationObject* pObjB = pManipulationObject->Clone(pColCheckerB, sObjB);

...

// thread A
CRobotCollisionModelCollection *pColA = pRobotA->GetCollisionModel(sLeft);
bool bColA = pColCheckerA->CheckCollision(pColA, pObjA);

// thread B
CRobotCollisionModelCollection *pColB = pRobotB->GetCollisionModel(sLeft);
bool bColB = pColCheckerB->CheckCollision(pColB, pObjB);
```

2.4 Jacobian and it's Pseudoniverse

The Jacobain calculations are using Newmat's datatypes as *Matrix* and *ColumnVector*. The library *newmat* is provided in the *ExternalDependencies* directory of Simox. Please note that newmat starts counting with 1, which could be confusing for C++ programmers. In the following example you can see how the Jacobians and the Pseudoinverse Jacobians are calculated and how a Cartesian Delta is used to compute joint deltas via the Pseudoinverse Jacobian.

Listing 2.18: Jacobians

```
// calculate the Jacobuian for the given kinematic chain
Matrix mJac1 = pRobot->GetJacobian(pKinematicChain);

// calculate the jacobian in a given coordinate system
std::string sCoordSystem("Right Shoulder");
Matrix mJac2 = pRobot->GetJacobian(pKinematicChain, sCoordSystem);

Matrix mInvJac = pRobot->GetInverseJacobian(pKinematicChain);

// do a inverse jacobain calculation
ColumnVector cvStoreJointDelta(m_nDimension);

// the Cartesian Delta (x,y,z followed by Roll Pitch Yaw angles)
ColumnVector cvDeltaInGlobalCoordSystem(6);
cvDeltaInGlobalCoordSystem << 10.0f << 0 << 0 << 0 << 0 << 0; // 10mm in x
cvStoreJointDelta = mInvJac*cvDeltaInGlobalCoordSystem;
for (unsigned int i=0;i<m_nDimension;i++)
{
    pJointValue[i] += cvStoreJointDelta(i+1); // ! note the +1
}
// apply values
pRobot->SetJointValues(pJointValues, pKinematicChain);
```

2.5 IK-Solver

The probabilistic IK solver uses the Pseudoinverse Jacobian to iteratively reduce the error from the current pose of the TCP to the target pose. If that fails, a randomly chosen pose is used as starting point for further iterations. After a specified amount of unsuccessful tries the IK solver returns a failure.

Listing 2.19: An example how to use the probabilistic IK solver

```
std::string sKinChainHipArm("TorsoLeftArm");
CKinematicChain *pKinChainTorsoArm = pRobot->GetKinematicChain(sKinChainHipArm);
std::string sTCP("TCP L");
CRobotNode *pTCP = pRobot->GetNode(sTCP);
CProbabilisticIKSolver *pIKSolver =
    new CProbabilisticIKSolver(pRobot, pKinChainTorsoArm, pTCP);

float pTargetPose[6];
float pStoreResultConfiguration[10];

// transform target values
SbMatrix pGraspPose;
pGraspPose = pObj->GetGlobalGraspingPose();
MathTools::SbMatrix2PosRPY(pGraspPose, pTargetPose);
bool bRes = pIKSolver->solve(pTargetPose, pStoreResultConfiguration);
if (bRes)
    pRobot->SetConfiguration(pStoreResultConfiguration, sKinChainHipArm);
```

An example for using the probabilistic IK solver can be found in the directory *Simox/VirtualRobot/examples/IK-Demo*. Here the generic Jacobian calculations and IK solvers are used to move around several kinematic chains of the robot.
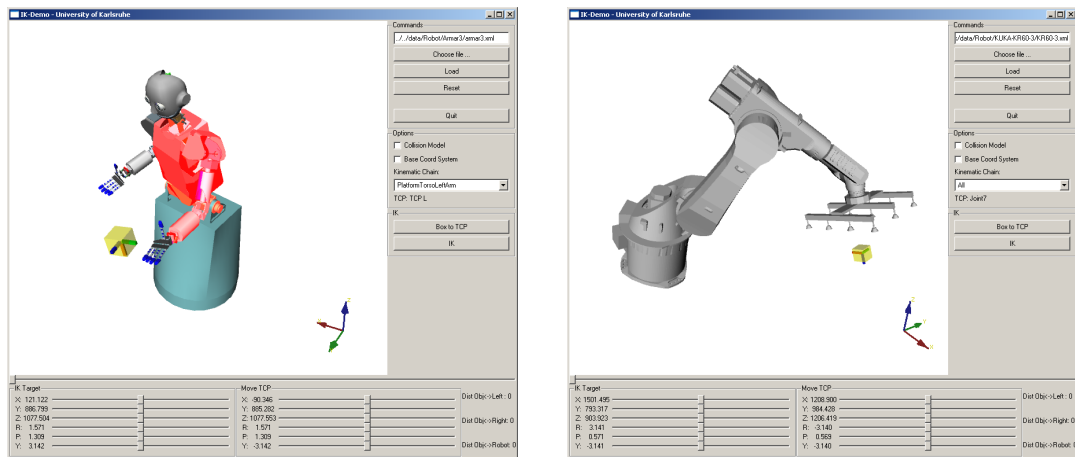
Figure 2.8: The example IK-Demo shows how to access the Jacobians and the generic IK solver.

2.5.1 Reachability Distribution

The reachability distributions (called *ReachabilitySpaces*) approximate the reachability of 6D poses (position and orientation) for a given kinematic chain, e.g. a manipulator or an arm. This data can be used to quickly decide whether a 6D pose is reachable or not which can be useful when searching a reachable position for an object. Instead of calling the IK-solver for lots of samples with low chance for finding an IK-solution, the reachability spaces can be used for fast checking and discarding lots of configurations.

The reachability spaces are generated in an offline step by randomly sampling a large number of configurations for the given kinematic chain and the corresponding poses of the TCP are determined by calculating the forward kinematics. These poses are used to fill the reachability distribution and finally the data can be stored to binary files for using it later on. The examples depicted in Fig. 2.9 have been generated by sampling more than 400 million configurations which took three days on a standard Linux PC. The reachability spaces can be used to speed up the probabilistic IK solver as shown in Listing 2.20.

Listing 2.20: Accessing the reachability distributions

```cpp
// load rachability distribution
CReachabilitySpace *pReachSpace = new CReachabilitySpace();
pReachSpace->Load(sFilename, pRobot);

// visualize
SoSeparator *pSep =
    pReachSpace->CreateVisualisation(CReachabilitySpace::eRed);

// get entry of reachability distribution
float pPose_PosRPY[6] = {100.0f, 0, 0, M_PI, 0, 0};
int nValue = pReachSpace->GetEntry(pPose_PosRPY);
if (nValue==0)
    cout << "This pose seems not to be reachable..." << endl;

// create an instance of the IK solver, with reachability information
CProbabilisticIKSolver *pIKSolver =
    new CProbabilisticIKSolver(pRobot, pKinChainTorsoArm, pTCP, pReachSpace);
```
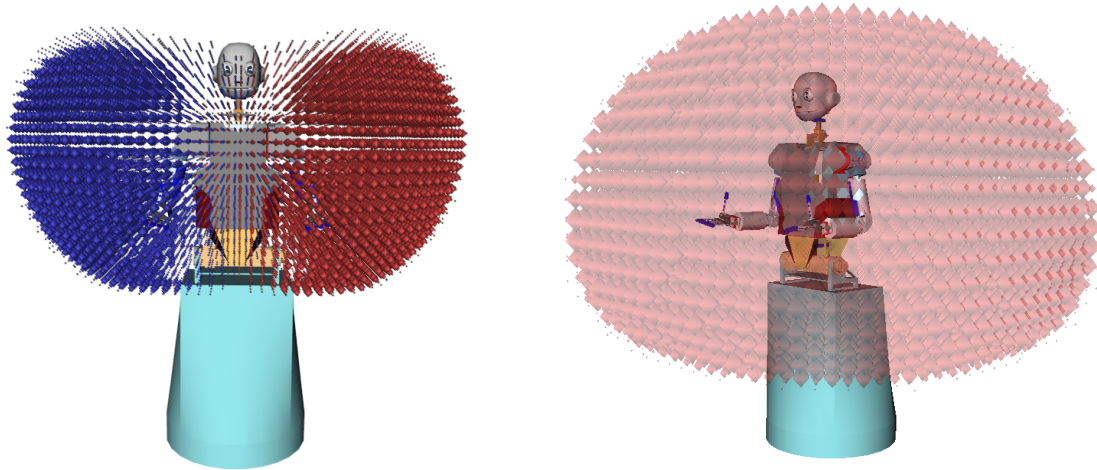
Figure 2.9: Approximated reachability distributions for two kinematic chains of ARMAR-III. The left image shows the reachability of the arms where the size of the balls is proportional to the reachability value in the corresponding cell. The right image shows the reachability distribution of the 13 DoF kinematic chain covering platform rotation, hip and left arm. Note that the visualizations are three-dimensional projections of the six-dimensional reachability distributions.

2.6 Helper Methods

Virtual Robot provides some helper methods for convenience. The functions provided by *MathTools* can be used to convert representations of orientations including Quaternions, Roll-Pitch-Yaw angles and homogeneous 4x4 matrices. The class *CIVTools* offer possibilities for converting OpenInventor models to triangulate data which can be easily accesed for further use.

Listing 2.21: Helper methods for converting coordinates

```cpp
float pPosEulerZXZ[6];
float pPosQuat[7];
float pPosRPY[6];
SbMatrix mMat;
float pQuat1[4];
float pQuat2[4];
float pQuat3[4];
float fAngle;

// a selection of available convertion methods
MathTools::SbMatrix2PosEulerZXZ (mMat, pPosEulerZXZ);
MathTools::PosQuat2SbMatrix (pPosQuat, mMat);
MathTools::PosRPY2PosQuat (pPosRPY, pPosQuat);

// quaternion calculations
MathTools::QuatMulQuat (pQuat1, pQuat2, pQuat3);
MathTools::DeltaQuat (pQuat1, pQuat2, pQuat3);
MathTools::QuatAngle (pQuat1, fAngle);
```

Listing 2.22: Helper methods for OpenInventor

```cpp
CIVTools::Model3d StoreModel;

// the high res model where the distance between all points is <= 10
SoSeparator* pHighResIVModel = CIVTools::RefineModel(pIVModel, 10.0f);

// convert IV model to internal data structure
CIVTools::ConvertModel(pIVModel, StoreModel);

// convert internal data structure to IV model
SoSeparator *pIVModel2 = CIVTools::ConvertModel(StoreModel);

// print out vertices
std::vector<CIVTools::Vec3d>::iterator itVertices =
    StoreModel.m_Vertices.begin();
while (itVertices!=StoreModel.m_Vertices.end())
{
    cout <<
        (*itVertices).x << "," <<
        (*itVertices).y << "," <<
        (*itVertices).z << endl;
    itVertices++;
}

// print out facets
std::vector<CIVTools::Face3d>::iterator itFaces =
    StoreModel.mFaces.begin();
while (itFaces!=StoreModel.mFaces.end())
{
    cout << "Vertice IDs: " <<
        (*itFaces).m_nId1 << "," <<
        (*itFaces).m_nId2 << "," <<
        (*itFaces).m_nId3 << endl;
    cout << "Normal: " <<
    (*itFaces).m_Normal.x << "," <<
    (*itFaces).m_Normal.y << "," <<
    (*itFaces).m_Normal.z << endl;
    itFaces++;
}
```

Chapter 3

Saba

The *Sampling-Based Motion Planning Library (Saba)* can be used for planning collision-free motions for robots that are defined by the *Virtual Robot* library. The library provides several generic algorithms for efficient planning, most of them are based on techniques related to Rapidly-exploring Random Trees (RRT) (see [6]).

3.1 Configuration Spaces

The configuration space (C-Space) covers all configurations of the system, which can be a complete robot or a part of it, defined by a kinematic chain. The main class, used by most planning approaches, is *CSpaceSampled* providing functionality for sampling, validating and collision checking in C-Spaces.

3.1.1 Managing Collision Detection of Multiple Objects

Since a C-Space must provide functionality to determine whether a configuration is valid or not, methods for collision detection have to be realized. To conveniently define different setups for collision detection (e.g. multiple obstacles, self-collisions or parts of robots), the class *CCollisionCheckingManagement* is used. The collision checking management (CCM) holds information about which *CCollisionModels* or *CCollisionModelCollections* should be tested against collisions. In Listing 3.1 a CCM is defined and several collision models are added. After defining all components, a situation with a collision can easily be checked by calling the method *CheckCollision()* . In this example any mutual collision between the environment, the manipulation object, the left arm and the right arm of the robot will be reported (note that no collisions within the left or right arm are considered).

3.1.2 Defining a C-Space

By using the CCM definition, a CSpace can be constructed easily (see Listing 3.2). The robot for which the C-Space is used, the CCM, and the string defining the kinematic chain of all joints that span the C-Space have to be specified on construction. The kinematic chain implicitly defines the dimensionality of the C-Space (which equal to the number of joints of

the kinematic chain) and the extends of the dimensions (which are retrieved from the limits of the joints). In Listing 3.2 an instance of *CSpaceSampled* is generated and the collision status of a single configuration and of a path segment is computed. Checking the collision status of path segments is realized by an efficient divide and conquer algorithm.

Listing 3.1: Defining a CCM

```
CColCheckManagement *pCCM = new CColCheckManagement();
pCCM->SetEnvironment(pEnv);

CCollisionModel *pColModel = pManipulationObject->GetCollisionModel();
pCCM->AddCollisionModel(pColModel);

std::string sLeft("Left Arm");
CRobotCollisionModelCollection *pColModelR1 = pRobot->GetCollisionModel(sLeft);
pCCM->AddCollisionModel(pColModelR1);
std::string sRight("Right Arm");
CRobotCollisionModelCollection *pColModelR2 = pRobot->GetCollisionModel(sRight);
pCCM->AddCollisionModel(pColModelR2);

// check if there is a colliding situation
bool bCollision = pCCM->CheckCollision();
...
// in case multi-threaded setups are used, all objects accessed by one thread
// have to be registered to the same instance of the collision checker
CColCheckManagement *pCCM_MT = new CColCheckManagement(pColChecker);
```

Listing 3.2: Defining a C-Space

```
CSpaceSampled *pCSpace = new CSpaceSampled(pRobot, pCCM, sKinematicChain);

// the sampling size is used when new paths are added
pCSpace->SetSamplingSize(0.1f);

// the collision checking sampling size is used
// to determine whether a path is collision free or not
pCSpace->SetColCheckSamplingSize(0.05f);

// Now a single configuration or a path between
// two configurations (straight line) can be checked
// for collisions (pConfig has to be a float array of valid size)
bool bValid1 = pCSpace->IsConfigValid(pConfig);
bool bValid2 = pCSpace->CheckPath(pConfig1, pConfig2);
```

3.1.3 Exact Collision Detection

In the class *CSpaceSampled* the collision status of a path segment is determined by generating intermediate samples and performing discrete collision checks. If none of the samples is in collision, it is assumed that the path is collision-free. By setting the sampling parameter to an adequate value, most situations can be handled by this discrete collision detection (DCD) algorithm. In case, the exact collision status of a path is requested, continuous collision detection (CDC) routines can be used. Therefore, the *free bubble* concept of [7] is integrated in *Saba*. A *free bubble* of a configuration is a sphere in C-Space, for which a guarantee can be given, that all covered configurations are collision-free. The radius of the sphere can be calculated from the obstacle distance in workspace. By sampling a path segment, so that the

free bubbles of the samples overlap, a guarantee can be given, that the path is completely collision-free. The class *CSpaceFreeBubbles* implements an optimized version of this *free bubble* check and since *CSpaceFreeBubbles* is derived from *CSpaceSampled* it can be used alternatively with all planners that rely on *CSpaceSampled*. In Listing 3.3 an example is given.

Listing 3.3: Exact collision detection

```
CSpaceFreeBubbles *pCSpace =
    new CSpaceFreeBubbles(pRobot, pCCM, sKinematicChain);

bool bValid_Guaranteed = pCSpace->CheckPath(pConfig1, pConfig2);
```

3.2 RRT-based Planning of Collision-Free Motions

For planning collision-free motions several RRT-based planners are provided by *Saba*. The use of uni-directional planners as *CRrtExtend* and *CRrtConnect* are shown in Listing 3.4. The bi-directional planner *CRrtBiPlanner* needs two instances of *CSpaceSampled*, one for each search tree. In Listing 3.5 it is showed how to use a bi-directional planner and how to access the results. It is shown, how a solution path can be interpolated and how the search trees, that were generated during planning, can be accessed.

Listing 3.4: A uni-directional planner

```
// using the EXTEND method for planning
CRrtExtendPlanner *pPlanner = new CRrtExtendPlanner(pCSpace);

pPlanner->setStart(pStartConfig);
pPlanner->setGoal(pGoalConfig);
bool bRes = pPlanner->Plan();

// using the CONNECT method for planning
CRrtConnectPlanner *pPlanner2 = new CRrtConnectPlanner(pCSpace);

// setup the probability of connecting the target to the tree
// -> with this probability the planning goal is used as new target
//    for connecting instead of using a random configuration
pPlanner2->SetProbabilityGoalDirection(0.05f);
pPlanner2->SetStart(pStartConfig);
pPlanner2->SetGoal(pGoalConfig);

// perform the motion planning
bool bRes2 = pPlanner2->Plan();
```

Listing 3.5: Bi-directional planning

```cpp
CSpaceSampled *pCSpace1 = new CSpaceSampled(pRobot, pCCM, sKinematicChain);
CSpaceSampled *pCSpace2 = new CSpaceSampled(pRobot, pCCM, sKinematicChain);
CRrtBiPlanner *pPlanner = new CRrtBiPlanner(pCSpace1, pCSpace2,
    CRrtBiPlanner::RRT_CONNECT, CRrtBiPlanner::RRT_CONNECT);

pPlanner->SetStart(pStartConfig);
pPlanner->SetGoal(pGoalConfig);
bool bRes = pPlanner->Plan();
if (bRes)
{
    // access the solution
    CRrtSolution *pSolution = pPlanner->GetSolution();
    // get intermediate configuration
    float pConfig[nDimensions];
    pSolution->Interpolate(0.5f, pConfig);
    // and set robot to this config
    pRobot->SetConfiguration(pConfig, sKinematicChain);
    // access the planning trees
    CSpaceTree *pRrt1 = pPlanner->GetPlanningTree();
    CSpaceTree *pRrt2 = pPlanner->GetPlanningTree2();
}
```

3.2.1 Optimizing Planned Trajectories

The results of RRT-based planners usually define a collision-free motion that brings the robot from the start to the goal configuration, but in general the results are not optimal. To optimize planned motions the class *CShortcutOptimizer* can be used as shown in Listing 3.6.

Listing 3.6: Optimizing a trajectory in C-Space

```cpp
CShortcutOptimizer Optimizer(pSolution, pCSpace1);
int nSize1 = pSolution->GetPathSize();
CRrtSolution *pSolutionOpti;
pSolutionOpti = new CRrtSolution(Optimizer.Optimize(500));
int nSize2 = nSize1 - pSolutionOpti->GetPathSize();
std::cout << "Kicked " << nSize2 << " nodes" << std::endl;
```
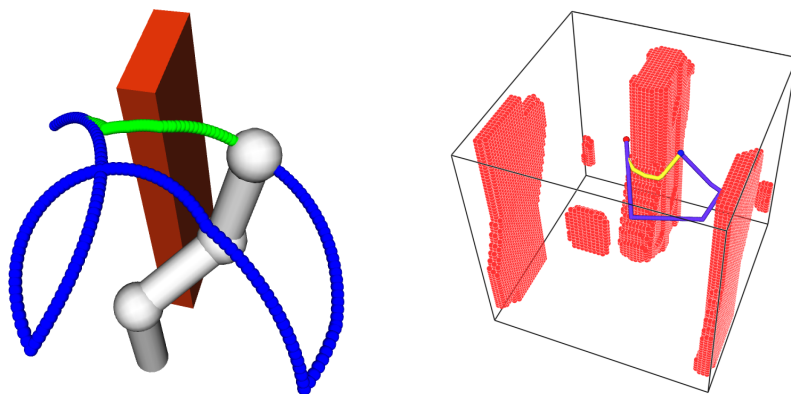

Figure 3.1: A planned path and it's optimized version in work- and C-Space.

3.2.2 Visualizing the Results

The results of a planner can be visualized as workspace movements of a joint or as a direct visualization of the C-Space (in the latter case the C-Space has to be three-dimensional).

Listing 3.7: Visualizing the results as workspace movements

```cpp
std::string sTCP("EndPoint");
CRrtWSpaceVisualization* pWSpaceVisu =
    new CRrtWSpaceVisualization(m_pRobot, sTCP);
// enabling high quality rendering slows down the visualization
// in case large search trees should be visualized
pWSpaceVisu->SetHighQualityRendering(true);
// add two search trees and two solutions to display
pWSpaceVisu->AddTree(sKinematicChain, pRrt1, pSolution, pSolutionOpti);
pWSpaceVisu->AddTree(sKinematicChain, pRrt2);
// create visualization
pWSpaceVisu->BuildVisualizations(bShowTree, bShowSolution);
// get the visu
SoSeparator *pSep = pWSpaceVisu->GetTreeVisualisation();
```

Listing 3.8: Visualizing a three-dimensional C-Space

```cpp
CRrtVisualization RrtVisu;
RrtVisu.setRrt(pRrt1, pRrt2, pStartConfig, pGoalConfig, pSolution);
// first try to load the sampled C-Space (much faster than sampling)
if (!RrtVisu.LoadSampledCSpace(sFilenameCSpaceSampling.c_str()))
{
    // if loading fails, create samplings and save them
    std::cout << "Could not load data, sampling cspace..." << std::endl;
    RrtVisu.CreateCollisionSamplingPoints(0.1f, pRobot, sKinChain);
    RrtVisu.SaveSampledCSpace(sFilenameCSpaceSampling.c_str());
}
SoSeparator *pCSpaceVisuSep = new SoSeparator();
RrtVisu.CreateCSpaceVis(pCSpaceVisuSep);
```
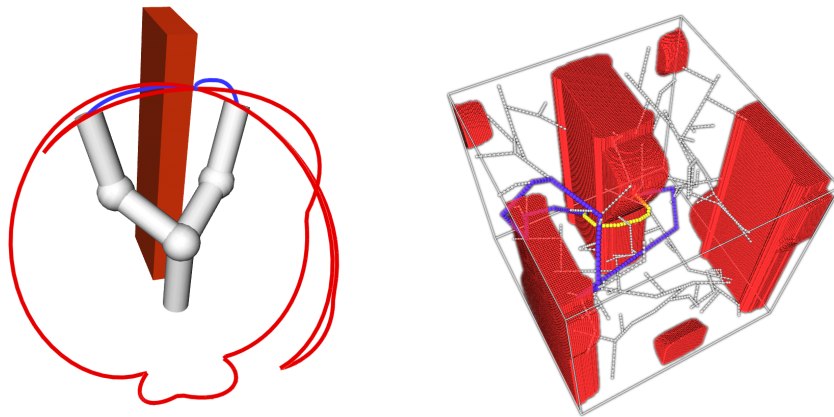

Figure 3.2: Visualization of a planning result in work- and C-Space.

3.2.3   Multi-Threading

It is possible to start a planner or an optimizer in a thread, e.g. for decoupling the planning
from the main loop of a graphical user interface. Therefore the two classes *CPlanningThread*
and *CPostprocessingThread* are provided by *Saba* (see Listing 3.9).

Listing 3.9: Threaded planning

```cpp
CPlanningThread *pPlanningThread = new CPlanningThread();
pPlanningThread->SetPlanner(pPlanner);
pPlanningThread->start();
while (pPlanningThread->IsRunning())
{
    // do whatever you want
}
// now the planner has finished
CRrtSolution *pSol = pPlanner->GetSolution();
if (pSol)
    cout << "Planning was successful..." << endl;
```

Furthermore, Simox offers an multi-threaded implementation of the RRT-Planner. This
planner starts n threads doing collision checks of paths. Therefore multiple clones of the
scene are generated and managed by the planner. An example of an application can be found
at *Simox/SaBa/examples/Kuka* or *Simox/SaBa/examples/MultiThreadedPlanning*.

Listing 3.10: Multi-threaded planning

```cpp
std::vector<std::string> vColModels;
vColModels.push_back(sColModel);
// setup planner with 4 threads
CRrtBiMTPlanner *pPlannerMT =
    new CRrtBiMTPlanner(pCSpace1, pCSpace2, vColModels, 4);
pPlannerMT->SetStart(pStartConfig);
pPlannerMT->SetGoal(pGoalConfig);
pPlannerMT->Plan();
CRrtSolution *pSolution = pPlannerMT->GetSolution();
```
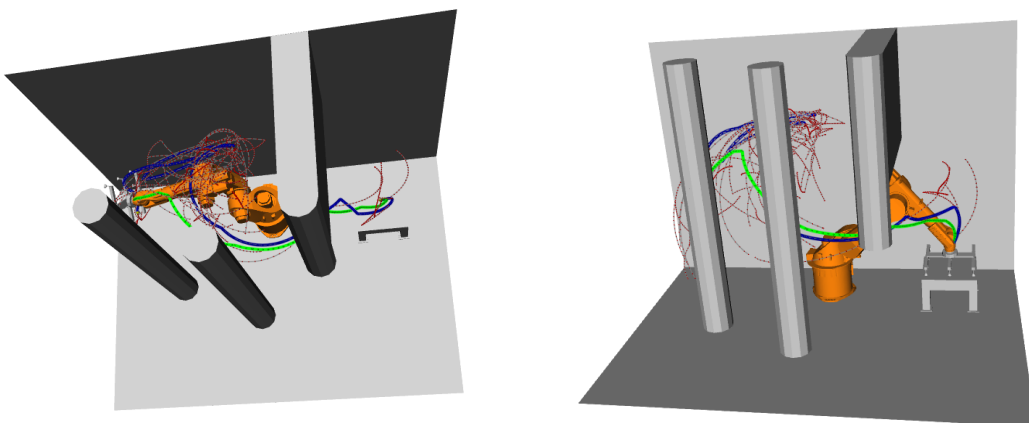

Figure 3.3: Multiple threads are used for checking the collision status of paths.

3.2.4 Planning Grasping Motions

Planning grasping motions can be done with the following two approaches: *GraspJacobian-Planner* and *IK-RRT*. The first one is a general planning algorithm which can be used without the need of any robot-specific implementations, whereas the second approach is based on an efficient probabilistic IK-solver which might be robot-specific (see e.g. [8] for details). Since the *IK-RRT* approach realizes a bi-directional RRT-based search it is faster than the *Grasp-JacobianPlanner* for most setups, but the performance strongly depends on the efficiency of the used IK-solver. An example on how to use the *IK-RRT* approach for planning grasping motions with the humanoid robot ARMAR-III [4] is given in *Simox/Saba/examples/IK-RRTDemo*.

In Listing 3.11 a planner setup is presented. The planner is initialized with instances of *CSpaceSampled* (implicitly defining the kinematic chain used for planning), *CRobot* (the robot we want to use), *CManipulationObject* (the target object to grasp), *CEndEffector* (the end-effector used for grasping and *CFeasibleGraspCollection* (the potential grasping configurations defining how to grasp the object). When the planner succeeds, it implicitly selects a feasible grasp, an IK-solution together with a collision-free trajectory that moves the EEF to the grasping pose.

Listing 3.11: The GraspJacobainPlanner

```
CGraspJacobianPlanner* pPlanner =
    new CGraspJacobianPlanner(pCSpace, pRobot, pGraspObject, pEEF, pGrasps);

pPlanner->SetStart(pConfig);
pPlanner->Plan();

// retrieve results
CFeasibleGrasp* pGrasp = pPlanner->GetSolutionGrasp();
CRrtSolution *pSolution = pPlanner->GetSolution();
```
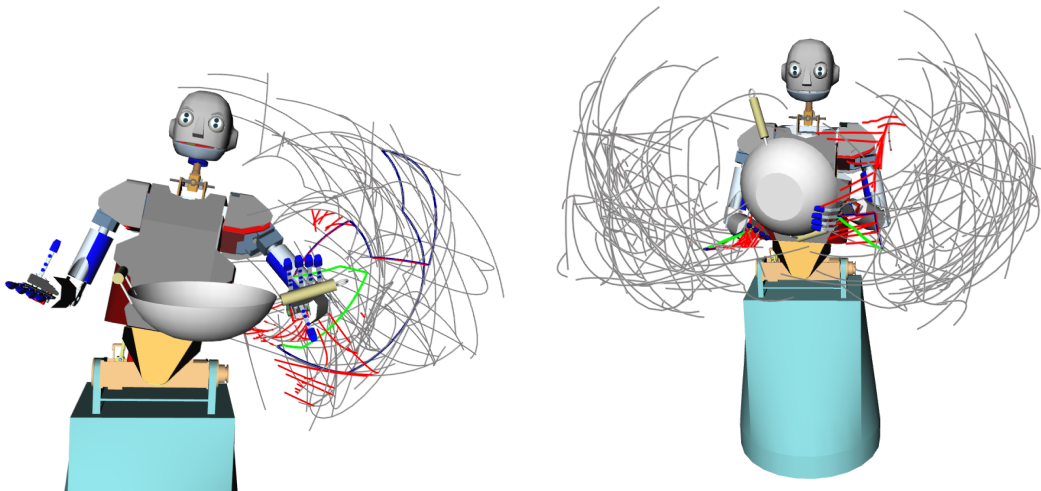

Figure 3.4: Two exemplary results of the *GraspJacobianPlanner*. The red lines are generated by Jacobian-based approach movements during planning

3.3 Robot-Specific Libraries

Since several implementations depend on the robot's kinematic structure (e.g. a bimanual planner needs a robot with at least two arms, or analytic IK-solvers operate on a specific kinematic setup), *Simox* provides a way of defining robot specific code. E.g. the *RobotLibArmarIII* holds implementations as IK-solvers for arms and head, which are related to the humanoid robot ARMAR-III (see [2, 4]).

Chapter 4

Grasp Studio

The *Grasp Studio* library contains methods and tools used for measuring grasp qualities. Therefore an interface to the *qhull* library is provided to build convex hulls in 3D or 6D. The methods for measuring grasp qualities rely on 3D force space or 6D grasp wrench space computations. Grasping setups of simple end-effectors, multi-finger hands, multi-hand and multi-robot grasps can be evaluated by the *Grasp Studio* library. Grasp planners are implemented for building object specific grasp maps for a given end-effector.

4.1 Measuring the Quality of a Grasp

Grasp Studio can be used to measure the quality of a grasping configuration. Therefore, the class *CGraspQualityMeasure* and its derived implementations can be used. To evaluate a grasping configuration, contact information between object and end-effector is needed, which can be acquired as shown in section 2.1.5.

4.1.1 Friction Cones

A common approach in grasp scoring is to approximate the exerted forces by friction cones which can be derived from a contact point in 3D and a contact normal. Therefore the class *CContactConeGenerator* can be used. Some visualizations of resulting friction cones are given in Fig. 4.1.
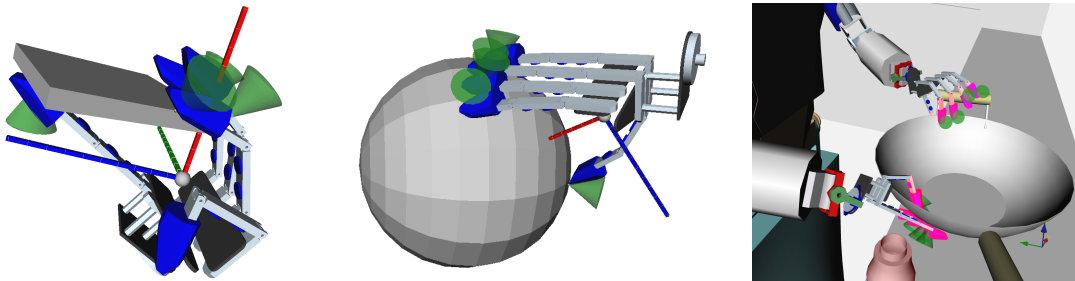

Figure 4.1: The friction cones are used to visualize the applied forces at the contacts.

4.1.2 Convex Hulls

Since for state-of-the-art algorithms for computing grasp qualities rely on convex hull computations, *Grasp Studio* offers an interface to *qhull* [9], an efficient and robust library for computing convex hulls. The open-source implementation of *qhull* is provided with the source-code of *Simox*. An example on how to use the interface to *qhull* for 3D and 6D points is given in Listing 4.1 and in Listing 4.2.
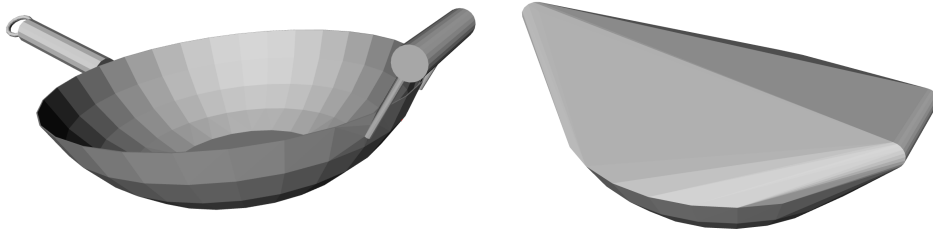

Figure 4.2: A 3D model and it's convex hull.

Listing 4.1: Creating a convex hull from 3D points

```cpp
// create random points
std::vector<GraspStudio::Vec3D> vPoints;
for (int i=0;i<100;i++) {
    GraspStudio::Vec3D point;
    point.x = rand()%1000; point.y = rand()%1000; point.z = rand()%1000;
    vPoints.push_back(point);
}
GraspStudio::ConvexHull3D convexHull, convexHull2;
// create convex hull of a point set
CConvexHullGenerator::CreateConvexHull(vPoints, convexHull);
// create a convex hull from a 3D model
CConvexHullGenerator::CreateConvexHull(pIVModel, convexHull2);
// create a visualization
SoSeparator *pSep = new SoSeparator();
CConvexHullGenerator::CreateIVModel(convexHull2, pSep);
```

Listing 4.2: Creating a convex hull from 6D contact data

```cpp
// create random points
std::vector<GraspStudio::ContactPoint> vPoints;
for (int i=0;i<100;i++) {
    GraspStudio::ContactPoint point;
    point.x = rand()%1000; point.y = rand()%1000; point.z = rand()%1000;
    point.nx = rand()%1000; point.ny = rand()%1000; point.nz = rand()%1000;
    vPoints.push_back(point);
}
// create convex hull of a point set
GraspStudio::ConvexHull6D convexHull6D;
CConvexHullGenerator::CreateConvexHull(vPoints, convexHull6D);
// create a visualization (force/torque space)
SoSeparator *pSep = new SoSeparator();
CConvexHullGenerator::CreateIVModel(convexHull6D, pSep, true /* false */);
```

4.1.3 Grasp Force Space

The class *CGraspQualityMeasureForceSpace* can be used for quick and efficient computations of grasp qualities. Therefore the *ObjectForceSpace* (OFS) is calculated once per object and for a given grasp setup the *GraspForceSpace* (GFS) is set in relation to it.

To build the OFS, the object's surface is sampled and a grasping contact is assumed at each surface point. Then the convex hull of the approximated friction cones on all these contact points defines the OFS. To score a setup of contacts, the *GraspForceSpace* (GFS) is built by approximating friction cones at all contact positions and computing the convex hull of them. By determining the largest scaling factor, that encloses the GFS in the OFS, a grasp quality is computed.

A single handed and a bimanual grasping setup together with a visualization of the force spaces is shown in Fig. 4.3. In [10] an integrated grasp and motion planning approach is described that utilizes the grasp force space for quick online grasp quality computations.
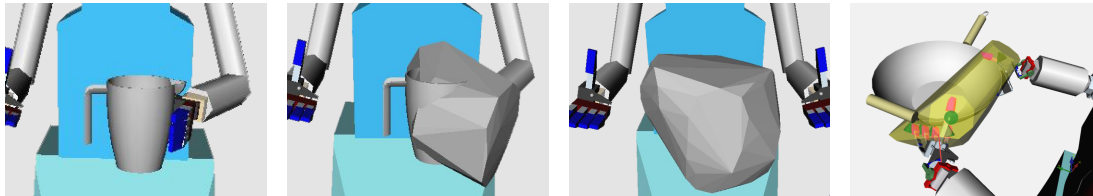
Figure 4.3: From left to right: (a) The grasp setup, (b) A visualization of the GFS, (c) The precomputed OFS, (d) The GFS of a bimanual grasping configuration.

Listing 4.3: Computing a grasp quality score

```cpp
// setup the grasp quality measurement with an object
CGraspQualityMeasureForceSpace* pGQMForce =
    new CGraspQualityMeasureForceSpace();
pGQMForce->SetObjectProperties((SoNode*)pObjSep);
pGQMForce->CalculateOWS();

// compute the grasp score
pGQMForce->SetContactPoints(vContactPoints);
float fScore = pGQMForce->GetGraspQuality();

// retrieve visualizations
SoSeparator *pSep1 = pGQMForce->GetVisualizationGWS();
SoSeparator* pSep2 = pGQMForce->GetVisualizationOWS();
```

4.1.4 Grasp Wrench Space

The *6D Wrench Space* of a grasping setup can be computed with *CGraspQualityMeasureWrenchSpace*. Therefore the wrenches of the contacts are generated and the convex hull of this 6D data is computed (see [11, 12, 13] for further information about Grasp Wrench Spaces). By determining the maximal size of an enclosing sphere, the quality of the grasp is computed. To retrieve an absolute quality value, the object specific *Object Wrench Space* is approximated by sampling the objects surface and assuming contacts at these points.

Listing 4.4: Computing the grasp wrench space

```
// setup the grasp quality measurement with an object
CGraspQualityMeasureWrenchSpace *pGQMWrench =
    new CGraspQualityMeasureWrenchSpace();
m_pGraspQualityMeasureWrench->SetObjectProperties((SoNode*)pObjSep);

// this is optional
// if the OWS was already computed, you can set the results directly
// in order to avoid it's computation
pGQMWrench->PreCalculatedOWS(fPreCalculatedMinDist, fPreCalculatedVolume);


// score the grasping configuration
pGQMWrench->SetContactPoints(contactPoints);
float fScore = pGQMWrench->GetGraspQuality();

// retrieve visualization
SoSeparator *pSep1 = pGQMWrench->GetVisualizationGWS();
SoSeparator* pSep2 = pGQMWrench->GetVisualizationOWS();
```

4.2   Generating Grasp Hypotheses

In case graps maps should be generated by grasp planning algorithms, a component is needed to compute approach movements resulting in grasp hypotheses. *Grasp Studio* offers an interface class (*CApproachMovementGenerator*) that can be used to implement own approach movement generators. One approach strategy is implemented by the class *CApproachMovementSurfaceNormal*, which samples positions on the object's surface and by aligning the *Approach Direction* of the end-effector with the surface normal, an approach direction is constructed. There is still one free DoF (the rotation around the normal) which is set to a random value. This approach movement is used to move the EEF toward the object until a collision is detected. Then, the EEF is moved back until a collision-free pose is reached and the hand can be closed to retrieve the contact information.

The class *CApproachMovementSurfaceNormal* creates an internal instance of *CRobot* that consists of the joints *Move X, Move Y, Move Z, Rotate EulerZXZ 1, Rotate EulerZXZ 2* and *Rotate EulerZXZ 3* followed by the EEF definition. This robot can be used to move the EEF (it's cloned version) around in order to generate approach movements.

In Listing 4.5 an end-effector together with the name of the joint defining the *Grasp Center Point* (GCP) and the start joint from where the cloning should begin are passed to the constructor of *CApproachMovementSurfaceNormal*. The GCP is used as the basis coordinate system for moving the robot around. Then, the internally created robot and end-effector can be accessed and set to potential grasp hypothesis which are generated randomly. By closing the hand, all contact information for the specific grasp can be retrieved for further calculations.

Listing 4.5: Creating grasping hypotheses

```
// create approach movement generator
CEndEffector *pEEF = pRobot->GetEndEffector(sNameEEF);
std::string sStartName("Hand L");
std::string sGCPName("GCP Left Hand");
```

```
CApproachMovementSurfaceNormal *pApproachGeneration =
    new CApproachMovementSurfaceNormal(pManipObj, pEEF, sGCP, sStart);

// retrieve internal clones of EEF-robot and EEF
CRobot *pEEFRobot = pApproachGeneration->GetEEFRobotClone();
CEndEffector *pEEFClone = pEEFRobot->GetEndEffector(sNameEEF);

// set EEF to a potential grasping position
pApproachGeneration->SetEEFToRandomApproachPose();

// close hand and store contact information
pEEFClone->CloseHandContactInfo(vContacts, pManipObj->GetCollisionModel());
```
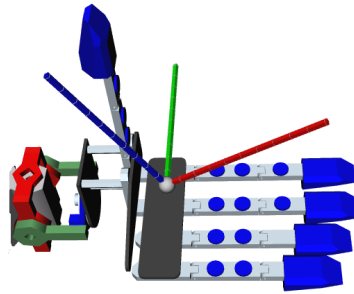

Figure 4.4: The Grasp Center Point (GCP) of a humanoid hand. The GCP is defined as a regular joint of the robot's XML definition. The x-axis (red) of the joint's coordinate system defines the approach direction of the end-effector.

4.3 Grasp Planner: Building Grasp Maps

The interface *CGraspPlanner* is used as a base class for all grasp planners which are used to generate feasible grasps for a specific EEF and object combination. The class *CGraspPlanner-General* can be used to build grasp maps using an instance of *CApprachMovementGenerator* for generating grasp hypothesis and an instance of *CGraspQualityMeasure* to score them.
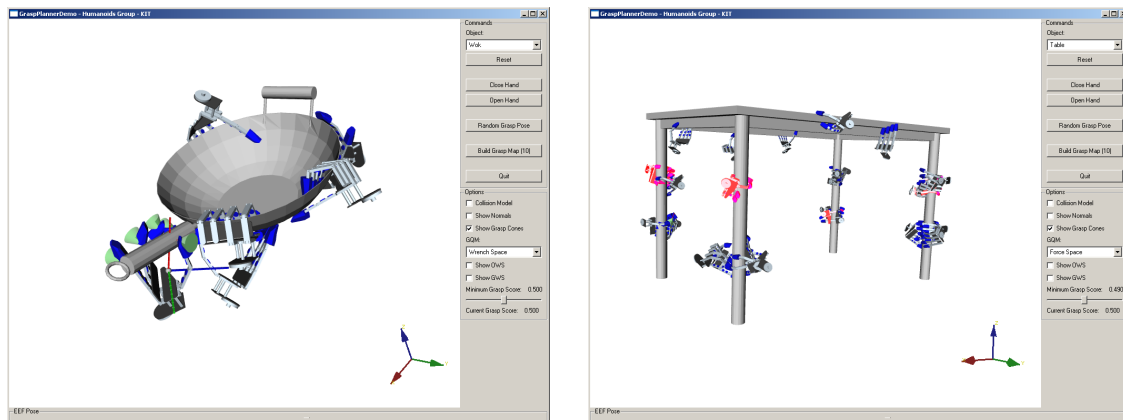
Figure 4.5: The tool *Grasp Planner* can be used to build grasp maps.

4.4 Grasp Studio: The Grasp Editor

The tool *Grasp Studio*, located at *Simox/GraspStudio/examples/GraspStudio*, can be used to create, visualize and edit grasping configurations for *ManipulationObjects*. Therefore robots and manipulation objects can be loaded and several options allow to create and manipulate the grasping information that is stored with the object.
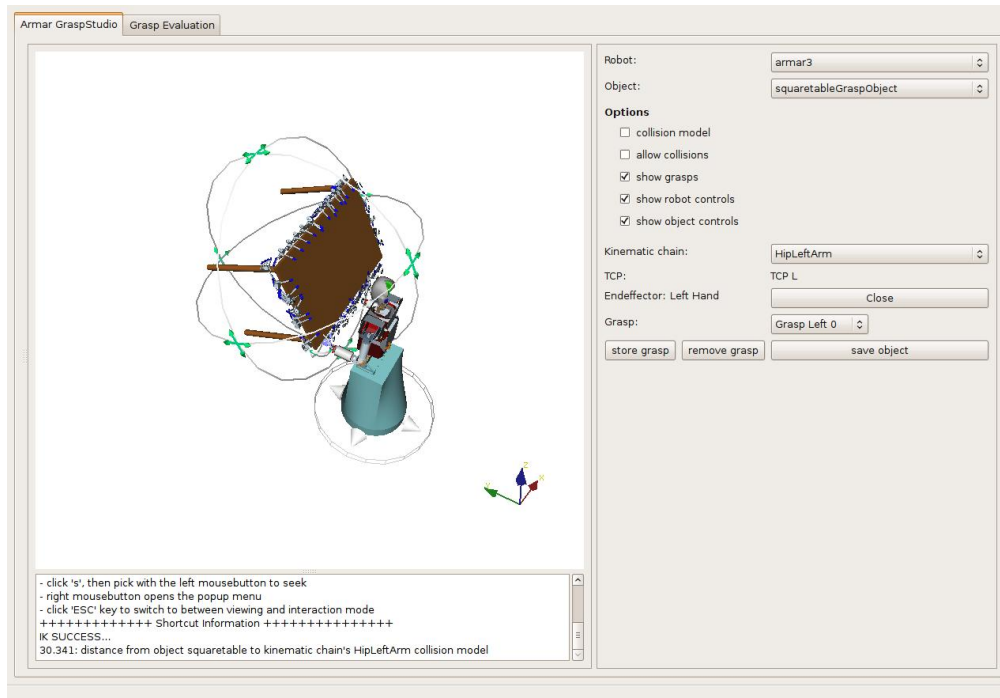

Figure 4.6: The tool *Grasp Studio* is used to manipulate object-related grasping information.

Acknowledgements

Bibliography

[1] J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993.

[2] T. Asfour, P. Azad, N. Vahrenkamp, K. Regenstein, A. Bierbaum, K. Welke, J. Schroeder, and R. Dillmann, "Toward humanoid manipulation in human-centred environments," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 54 – 65, 2008.

[3] I. Gaiser, S. Schulz, A. Kargov, H. Klosek, A. Bierbaum, C. Pylatiuk, R. Oberle, T. Werner, T. Asfour, G. Bretthauer, and R. Dillmann, "A new anthropomorphic robotic hand," in *Humanoid Robots, 2008. Humanoids 2008. 8th IEEE-RAS International Conference on*, dec. 2008, pp. 418 –422.

[4] SFB 588. The german collaborative research center on humanoid robots. [Online]. Available: http://www.sfb588.uni-karlsruhe.de

[5] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha, "Fast distance queries with rectangular swept sphere volumes," in *IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco (CA), 2000.

[6] J. Kuffner and S. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco (CA), 2000, pp. 995–1001.

[7] S. Quinlan, "Real-time modification of collision-free paths," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1995.

[8] N. Vahrenkamp, D. Berenson, T. Asfour, J.Kuffner, and R. Dillmann, "Humanoid motion planning for dual-arm manipulation and re-grasping tasks," in *IEEE International Conference on Intelligent Robots and Systems (IROS)*, October 2009.

[9] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, vol. 22, no. 4, pp. 469–483, 1996.

[10] N. Vahrenkamp, M. Do, T. Asfour, and R. Dillmann, "Integrated grasp and motion planning," in *IEEE International Conference on Robotics and Automation (ICRA)*, Anchorage (AK), 2010.

[11] D. Kirkpatrick, B. Mishra, and C. Yap, "Quantitative steinitz's theorems with applications to multifingered grasping," in *Proc.of the 20th ACM Symp. on Theory of Computing*, 1990, pp. 341–351.

[12] N. Pollard, "Parallel methods for synthesizing whole-hand grasps from generalized prototypes," *Technical Report AI-TR 1464, MIT, Artificial Intelligence Laboratory*, 1994.

[13] A. T. Miller, "Graspit!: a versatile simulator for robotic grasping." Ph.D. dissertation, Department of Computer Science, Columbia University, 2001.