A computational platform for simulating reach-to-grasp actions: modelling physics, touch receptors and motor control mechanisms

Timothy Neumegen

a thesis submitted for the degree of

Master of Science

at the University of Otago, Dunedin,

New Zealand.

20/12/2012

Abstract

Reaching to grasp is a fundamental human action. In this thesis, I present a new computational platform for simulating reach-to-grasp actions and exploring how they are learned and performed. There are three innovations in the platform. Firstly, many existing platorms for simulating reach-to-grasp actions hard code a definition of 'a stable grasp'. My aim was to create a platform based on an existing physics engine (the JMonkey game engine), so that the stable grasp is represented using general-purpose definitions of force and friction. Secondly, the platform implements a new model of the soft finger pads of the human hand, and of the finger's tactile mechanoreceptors. Each finger pad is modelled as a lattice of solid objects connected by springs to create a deformable surface. This allows a good simulation of the mechanoreceptors which respond to local finger pad deformations, and to other kinds of tactile stimulus. Finally, the platform supports a novel model of the motor controller responsible for generating reach-to-grasp actions. Most existing computational models assume that the hand's trajectory to the target is precomputed in advance, but there is evidence that this does not happen in the primate reach/grasp neural pathway. The motor controller in my system is a combination of a low-level feedback controller which tries to move the hand and arm into a learned goal state, and a high-level controller which perturbs or moves this goal state to create a virtual target location for the low level controller to reach towards. Combining these controllers allows for complex actions to be learned, without precomputing trajectories.

Acknowledgements

I would like to thank Alistair Knott for guiding me through the twisting path of this thesis. Your creative ideas, enthusiasm, knowledge and patience have made this a success.

To my friends and family who patiently listened to me talk about reach-to-grasp for the last few years, your support has been invaluable.

To Veronica, your curiosity and kindness is inspiring.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reaching to grasp is a fundamental human action. It is involved in almost any task that we perform. When an agent wants to pick something up he is only conscious of selecting a target object. His shoulder, elbow and hand then make the correct movements without him having to consciously think about it. One useful way to study the mechanisms which produce the movements is to simulate them computationally.

To create a simulation of reach-to-grasp actions there are two core issues: what platform the simulation will be based on and how the action will be modelled. In this thesis I cover both of these issues. The options for a simulation platform range from custom environments built to test a specific aspect of motor control to fully featured simulators representing real-life robotic graspers. I have developed a new platform based on a general purpose game physics engine. This overcomes the limitations of the custom environments which sometimes take short cuts in the simulation of physical reality, and also allows customisation to create a representation of human touch sensors in a way that most robotic simulators do not. For the action modelling most existing computational models represent the reach-to-grasp action by precomputing the hand's trajectory onto the target in advance of the motion actually being produced—in other words, the trajectory is planned in detail before it is executed. Computationally, this is a well understood method for producing reach-to-grasp actions, but there is evidence that it is not biologically accurate. I have developed a new model of reaching-to-grasp which works by combining a simple low-level **feedback controller** which reaches towards a goal motor state with a learned high level controller which can shift or 'perturb' this goal state. Combining these perturbations allows for complex actions to be performed without precomputing a trajectory for the arm to travel along. Some of the models use precalculated affordances for an object or calculate whether a grasp has

been performed based on the location of the contacts. My simulation uses the general purpose physics engine to ensure that each grasp is physically plausible. Combining my new platform and model has created an extensible simulation framework. Some of the work reported in this thesis and some extensions to the framework are included in the paper 'Representing reach-to-grasp trajectories using perturbed goal motor states' (Lee-Hand, Neumegen, and Knott, 2012).

1.1 Structure of the thesis

The structure of the thesis is as follows. First in Chapter 2, I will provide background on the biological systems we are simulating, and I will discuss the current platforms and physical simulations of the hand and arm ranging from robot simulators to custom solutions. Next in Chapter 3, I will review the different models for performing a reach to grasp. After outlining some basic principles of motor control I will review some of the relevant findings about the neural mechanisms subserving reaching-to-grasp in humans and monkeys. I will then discuss some recent neurally-insired computational models of reaching-to-grasp, highlighting the problem of precomputed hand trajectories, and my proposed solution. In Chapter 4, I will introduce my platform for simulating reach-to-grasp actions, and the different factors that have influenced its design. I will first discuss the choice of an underlying physics engine, and will then show how the different parts of my implementation connect to form the platform. In Chapter 5 I will focus on the finger implementation, showing a novel model of 'soft fingers' - the soft pads at the fingertips and the mechanoreceptor model to give sensory feedback. In Chapter 6, I will outline my model of the mechanisms involved in controlling a reach-to-grasp action, and of how these mechanisms are learned, giving implementation details and results for target objects of several different types. Finally, in Chapter 7, I will discuss the possible future directions for the research.

Chapter 2

The hand/arm motor system: Physical background and existing platforms

The simulation of a reach-to-grasp requires a representation of the hand and arm. Different models descend to different levels of detail, depending on their particular goals: for instance, some models include detail about the fingertips, while others focus just on the arm and omit details of the hand.

In this chapter I will begin by discussing how the human hand-arm system works to show what I am modelling. Then I will give an overview of the mechanoreceptors which provide the tactile feedback in a grasp action. Next I will review some existing frameworks for simulating reach-to-grasp actions and show why they did not suit my aims. I will show the different options for representing fingers and finally how these lead to the aims of my project.

2.1 The hand-arm-system

We can think of the hand-arm system as having two components. One component is the musculoskeletal structure, which determines the basic degrees of freedom of the system and how they can be controlled. The second component is the soft tissues of the hand/arm, which are what actually come into contact with objects in the world. The physical structure of these tissues is very important in our ablities to manipulate objects. In particular, the soft tissue of the palm and the finger pads have properties which help achieve grasps on objects. An agent does not have direct control over these

but the fine-grained sensory information they return is important in helping the agent control the action. In this section I will describe these two components and the choices for modelling them.

2.1.1 Basic degrees of freedom

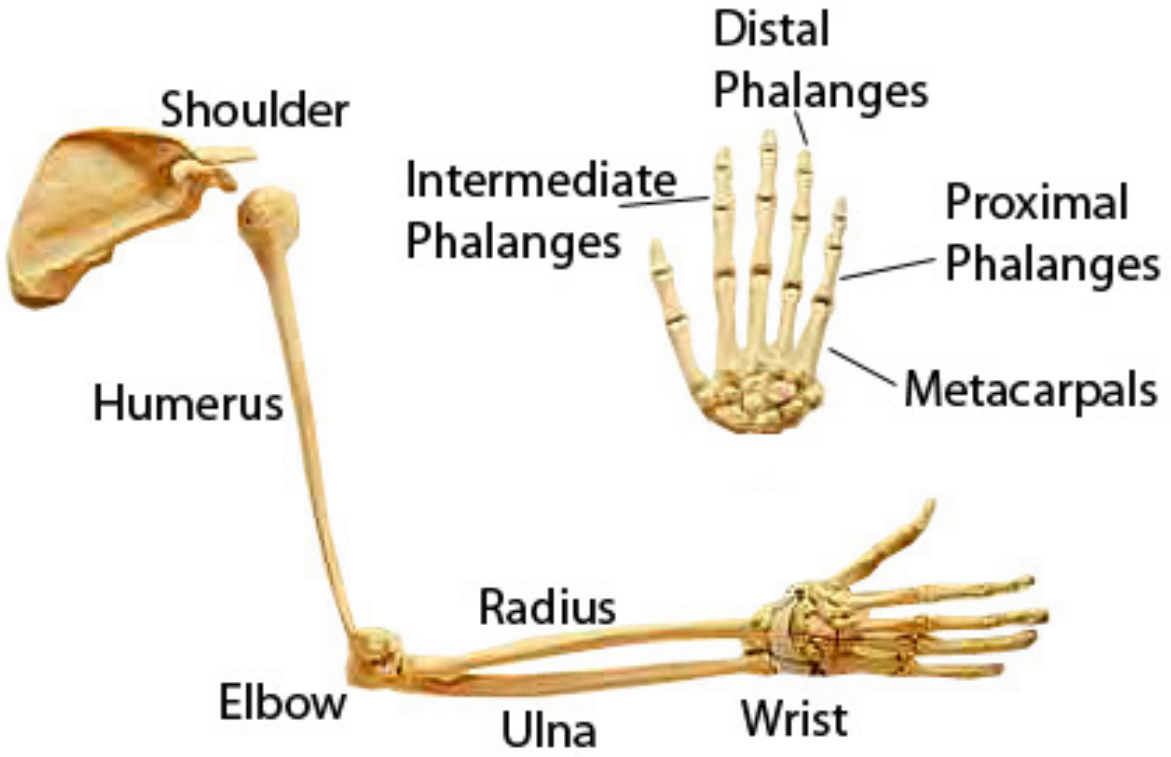

Figure 2.1: The bones in the arm and hand.
Image from Dorling Kindersley http://www.dorlingkindersley-uk.co.uk

There are two parts of your body used in a reach-to-grasp action, the arm and the hand. Figure 2.1 shows the bones and joints of the arm and hand. The arm is made up of two sections, the upper arm and the forearm. The upper arm bone is called the **humerus**. It joins to the rest of the body at the shoulder with a ball joint. The forearm is made up of two bones, the **radius** and the **ulna**. These join to the upper arm at the elbow with a hinge joint which also permits rotation about the axis of the forearm. The arm is then connected to the hand at the wrist by a joint with two degrees of freedom. The hand is made up of four fingers and a thumb. The first bones

of the fingers are the **metacarpals**; these are joined together, creating the palm of the hand. The last three bones are the **phalanges**. The first bones are the **proximal phalanges**, the middle finger bones are the **intermediate phalanges** and the final bones are the **distal phalanges**. These outer bones are separate and can be moved independently using the hinge joints connecting them. The metacarpal of the thumb is not joined to the fingers, giving the thumb a larger degree of freedom and allowing it to apply an opposing force in a grasp. The thumb is made up of two bones: the **proximal phalange** and the **distal phalange**.

2.1.2 Physical and sensory structure of the fingers

The finger pads are an important part of the reach-to-grasp action. From a physical point of view the contact they make with the object will define whether the object can be held in a stable grasp. They also provide feedback in the form of sensory information about the state of the grasp. In this section I will outline what is known about the touch receptors in the skin of the hand, and the kind of information they provide. This review draws on the description in Encyclopedia Britannica online Britannica (2010).

Basic composition of the skin

The skin is made from two different layers: the **epidermis**, which is the outermost layer, and the **dermis**. **Mechanoreceptors** are what we use to sense with our fingers. They are found in both layers of skin. There are different types of sensors for detecting pressure, texture and temperature. Sensors which are deeper in the skin have a wider and more general area on the surface which activates them. This area is called the sensor's receptive field.

Basic architecture of mechanoreceptors

Receptors work by sending a pulse to neurons. The sensors have ion channels which can release positively or negatively charged ions into the cell. When the sensor is deformed the ion channels release positive sodium ions into the cell which depolarises it. When the depolarisation crosses a threshold an action potential is created, which causes the ion channels to fully open creating a depolarisation. This in turn sends a pulse to the neuron. Once this happens ion channels open which release potassium ions into the cell, polarising it back to its original state. The sensor cannot send a new

pulse until this has happened so the speed of the repolarisation defines the maximum rate at which a sensor can send pulses.

Being over the threshold is a Boolean value. The amount a sensor is deformed is not sent in the pulse. Instead the magnitude is encoded by the frequency of the pulses. The pulse rate will be low when the sensor is passive, and will increase when it has a sensation. The speed at which the pulse returns to its original passive rate is called the adaptation speed. Some sensors only send a pulse when there is a change in deformation while others will change the speed of the pulse based on how active they are.

The pulses are carried by nerve fibres. There are different sizes of nerve fibres which carry signals at different speeds. These fibres are grouped into bundles. Each bundle is connected to sensors from a certain area of skin about 2.5cm wide. Different bundle's areas overlap slightly. These bundles join up in the spine and the signals travel up to the thalamus in the brain and finally to the posterior rolandic cortical sensory area. There are also reflex loops which process tactile stimuli in the spinal cord and generate motor responses without involving the brain at all.

The four varieties of mechanoreceptor

There are four main types of mechanoreceptor in the skin: I will discuss each type in turn.

Pacinian Corpuscle sensors are for sensing large changes in your fingers' shape. They are in the dermis layer. If you press your finger onto something so that the finger pad is deformed this is what would detect it. The sensor only sends a neural signal when there is a change. When your finger gets pushed in it sends a signal that this has happened. If it stays pushed in then the sensor will stop sending the signal.

Meissner's Corpuscles sensors are for sensing a light touch. They are just below the epidermis layer. They only send a neural signal when there is a change in stimulation. They are very sensitive but can only tell whether or not something is touching the skin. If you lightly brush your fingers over your hand these are what would sense it.

Merkel's Cells are for sensing the texture and shape of an object. They are in the epidermis layer. They send a low frequency stream of pulses while there is a stimulation and the frequency increases if the stimulation is changing. They only stimulate in a small area of the skin. If you place your fingers on a smooth surface (such as a table) and a rough surface (such as sand paper) there is only a small difference in the sensation

so it is hard to tell between them. If you move your fingers across the surfaces the difference in texture is much more obvious. Merkel's Cells are what sense this difference and are used to read Braille.

Ruffini Endings sense the skin's degree of stretch and a joint's change in angle. They help control finger movements and detect the slippage of objects These are what help you hold on to objects. They are located deep in the dermis layer and receive inputs over a wide area of the surface. When a Ruffini Ending has been activated it fires responses at a constant rate. When a change is occurring it will fire at an increased rate.

2.2 Existing platforms for reach-to-grasp simulation

To perform simulations of a reach-to-grasp there needs to be a system to run in. One option is to implement the physics algorithms from the ground up. There are many calculations which need to be performed such as forces acting on objects, inertia and collisions between objects. If one writes one's own implementation then the algorithms can be customised to suit one's exact problem, and it is possible to implement simplifications or short-cuts. It is possible to build a simulation which runs with simplified physical rules such as no friction or gravity. Simplifying the rules can help a theorist focus the simulation on the problem, but can also give unrealistic results in areas where corners have been cut. If friction is ignored and the simulation sticks the object to the hand once contact has been made then the hand can pick up the object with a grasp which would not work in real life.

Another option for constructing a simulation platform is to use an existing domain specific framework that is specialised for simulating a reach-to-grasp. This lets the theorist concentrate on specific reach-to-grasp research without building the representation in the simulation. There are a number of existing frameworks with a range of abstraction. As a framework becomes more focused on a specific task it becomes harder to modify it to do anything else. Also to make a fundamental change to an existing framework can require an in-depth knowledge of how the framework works when it may be easier to build the functionality from the ground up.

A third option is to use a **general purpose** physics engine. This is a representation which is designed for any physical interactions. Often these are used in game or animation frameworks. One advantage of using general purpose engines is that because

they have a domain-general representation of physics, it is harder to cut corners in the way that is possible when implementing physics algorithms specific for one particular modelling task. For example a stable grasp should be one which requires enough friction to overcome the force of gravity on an object. If your motor controller can achieve a stable grasp in a general-purpose physics engine, this provides some measure of confidence that the controller's results have at least some basis in reality.

In the remainder of this section I will consider the existing domain specific grasp packages.

2.2.1 Mirror neuron system

One example of building from the ground up is a simulation platform by Oztop (2002) which is built to model the mirror neurons system: the system which allows an agent to use its own motor system to represent actions observed by the agent. It is written in Java based on the standard libraries. The aim of this platform is to plan, recognise and display reach-to-grasp actions. The system can either plan a grasp and perform the action using inverse kinematics (See 3.1.1) or observe a grasp action in motion to determine the type of grasp from a known set. The hand and fingers are represented as rigid objects. The system implements many of the goals I wanted for a platform. It displays the finger movements in 3D and detects when a grasp has been made. The focus of the simulation was on the neural network representing the mirror neuron system so various underlying parts of the simulation are hard-coded. The types of grasp were fixed and stored as different movements of joints. One feature of this system was its implementation of **object affordances** (Gibson, 1977). The affordances of an object are the actions which can be done on it: when an agent looks at an object, one of the things he has to do is compute these affordances, so he can select one of the actions which the object permits. In Oztop's system, affordances were hard-coded. I was interested in building a system which learned the motor affordances of objects, so Oztop's system was not suitable as it stood; in addition, because this system was designed for a specific purpose, it appeared rather hard to extend for my project. I wanted to learn the affordances which were hard-coded in Oztop's system and have a realistic representation of the fingers instead of modelling them as solid objects.

2.2.2 iCub simulator

The iCub platform (Zdechovan, 2011) is a humanoid robot simulator for studying embodied cognition. Masters student Luk Zdechovan is using it as a platform to study modeling object grasping with neural networks. The simulator has a virtual robot and has representations of the sensors such as Boolean or pressure sensors for the fingers and the ability to view through the eye cameras. It uses the Open Dynamics Engine physics framework (Smith, 2000). Since this is a simulator for the iCub robot the simulator is very focused. I wanted to model the sensors in a human more accurately that the iCub simulator which is modelling the specific sensors of the robot. I needed a more general simulation.

2.2.3 GraspIt

GraspIt by Miller and Allen (2004) is a simulator for performing and analysing robot grasps. There are three types of objects it can simulate: robot parts which are part of the system being controlled, graspable bodies which can be affected by forces and obstacles which are static and only give collision information. The robot parts can be joined together into a system such as using a hand and an arm. The robot can be controlled by specifying joint angles or using inverse kinematics (See 3.1.1). Collisions between objects are calculated based on the mesh of the objects. When one mesh intersects with another they are moved so they are just touching then the contact is calculated. When a grasp happens a **grasp wrench space** is constructed which approximates force vectors around each contact point and approximating the torque based on the centre of the object. They rate grasps on the minimum amount of force needed to disturb it in its weakest direction and compare this to the other directions. GraspIt uses solid shapes for its objects. Soft contacts are approximated by using a different collision algorithm and friction calculation for sections designated 'soft'. The mesh is not deformed. GraspIt is focused on robot simulation and has a large codebase which would be difficult to adapt to the features I wanted for modelling human hands. It does not have the sensor feedback or soft fingers I wanted for representing human fingers.

2.2.4 OpenGRASP

OpenGRASP by Len, Ulbrich, Diankov, Puche, Przybylski, Morales, Asfour, Moisio, Bohg, and Kuffner (2010) is a grasping simulation built on top of the robot simulator

OpenRAVE. Their aim was to create a grasping simulator which overcame some of the shortcomings of GraspIt by being modular and built on top of existing frameworks such as OpenRAVE and Blender. The models of the graspers are built from solid shapes. The models have two types of sensors: a tactile sensor which calculates the force on a surface area and a joint sensor which calculates the forces applied while grasping. The joints are controlled by specifying the angles, velocities or voltages. Grasps can be calculated offline then performed in the simulator or using vision to detect the grasp points on an object and aligning the model with these. Although the authors had designed OpenGRASP to be extensible with their modular approach the simulation is very focused on representing robots rather than humans. I wanted to control the joints as muscles instead of with joint angles. I also wanted to use soft finger contacts to represent human skin.

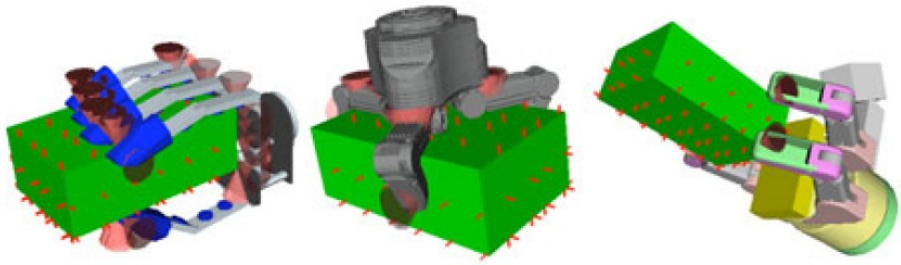

Figure 2.2: Hands in OpenGRASP performing a grasp action (Len *et al.*, 2010)

2.3 Finger representation in reach-to-grasp simulations

To use a hand in a simulation, it needs to have a physical representation. One of the key issues is how to represent the fingers of the hand. Fingers are 'soft' and this softness plays an important role in our ability to grasp. There are two main ways to represent the fingers: using solid geometrical shapes or flexible shapes. The choice between these depends on the focus of the simulation. Solid shapes are easier to build and faster at run time, but less realistic. Soft shapes are more like human fingers but are more difficult to create and computationally complex.

2.3.1 'Hard fingers'

Figure 2.3: Fingers made from solid objects Oztop (2002)

Hard fingers are where we use solid geometrical shapes to represent the fingers as shown in Figure 2.3. The solution uses stretched cubes or blocks to represent the fingers. An advantage of this method is that it gives fast collision detection, as the shapes are in the standard physics engine so no extra code needs to be added. Hard fingers are also easy to create, as the engine already supports them and it gives the basic shape of the hand quickly. The downside is that it is difficult to pick an object up with hard fingers. If your fingers are solid and you are grasping a curved object then each finger is only touching the object at a single point. This means that you have to use a lot of force to grasp the object to pick it up. Another way to achieve this result in a simulation is to increase the friction coefficient of the fingers, but this gives unwanted side-effects. Effectively you are modelling fingers covered in a sticky substance, which interact unnaturally with the objects they touch. With hard fingers it is also easy, especially in a pinch grasp, to squeeze an object away from the fingers if the contact forces are not exactly normal to the object's surface. This is similar to

trying to pick up an object using chopsticks. If the object is curved then it is difficult to get a stable grasp because so little surface area of the chopsticks is touching the object.

2.3.2 Soft fingers

The idea of soft fingers is to replicate the soft pads on human fingers to make picking objects up easier. There are different algorithms for achieving soft fingers. Barbagli, Frisoli, Salisbury, and Bergamasco (2004) try four different methods for emulating soft fingers which treat the fingertip as a sphere. When there is a collision between a finger and an object they work out what the deformation would be and apply the appropriate forces. The model that they found gave the most realistic results was treating the finger as a liquid filled membrane. Another group (de Pascale, Sarcuni, and Prattichizzo, 2005) combine this algorithm with a standard physics engine. The physics is handled by the engine until there is a collision with a finger. They then run a soft finger algorithm in a separate thread overriding the engine's behaviour. This gives the speed and ease of use of a physics engine with the accuracy of the soft finger model. In a human finger only the front of the finger is soft. Ciocarlie, Lackner, and Allen (2007) deal with this by having 'patches' when there is a collision with the finger pad. These patches approximate the area of the finger involved in the collision. They then use the same algorithm as Barbagli *et al.* (2004) to calculate the resulting forces.

2.3.3 Other methods for creating soft objects

Deformable objects is a wider area of research than just soft fingers. There are many solutions to this problem. One of these is to use a mesh of cubes such as the method used in Rivers and James (2007). The authors make a lattice of cubes joined by springs which is used for the collision calculations. This makes the objects flexible and bend when they collide.

2.4 Detailed aims of my project

As I have discussed in Section 2.2, there are many simulations of the human ability to reach and grasp objects, which highlight different aspects. Often these use custom physics solutions which require a large investment in development before the simulation can start. The first aim of my project is to show that you can use an 'off the shelf'

physics engine to achieve similar results. The advantages of using an existing physics engine is that you can get up and running straight away and you are building on top of a robust platform. The underlying physics environment is not tied to the simulation and is only interacted with through its general APIs. This means that the plant is not getting any special information about the world around it.

The second aim of my project is to give a reasonably detailed model of the tactile system of the hand, which plays an important role in grasping. Since the tactile system relies heavily on information about the deformation of the soft tissues of the finger pads, this in turn requires a particular implementation of 'soft fingers'. My model of soft fingers must simulate the way fingers deform around an object when they apply force to it. There are many methods for simulating soft fingers which require a custom physics solution or at least modifications to a standard physics engine. In my project I tried to achieve similar results with a standard physics engine. The platform and hand model I developed will be described in chapter 4. By using linked rigid bodies I was able to simulate the deformation of the fingertips without using a custom physics solution.

The third aim of my project is on modelling the controller which learns to generate successful reach/grasp actions. The controller needs to be able to move the hand on a trajectory towards the target object, and bring the fingers into a suitable grasp position on it. I have implemented a new approach, using 'virtual trajectories' which will be covered in chapter 6. To motivate the new approach, I begin by reviewing existing approaches in Chapter 3.

Chapter 3

Computational models of reaching-to-grasp

In this chapter I will review some of the literature on how motor actions are represented in the brain and models of reach-to-grasp actions. In Section 3.1 I will introduce different types of motor controller and show how they can be used together. In Section 3.2 I will summarise the biological pathways involved in reaching-to-grasp in animals that can perform this action, i.e. humans (and other primates). In Section 3.3 I will show some existing models for simulating motor control in reach to grasp and finally in Section 3.4, I will show how the assumption of precomputed trajectories these existing models are founded on may be wrong.

3.1 Motor control background

3.1.1 Kinematics

When we are simulating motion we need a method for calculating how the objects and joints interact. There are two commonly used methods for simulating arm movements: **forward kinematics** and **inverse kinematics**. Forward kinematics is where the joint angles of the arm are controlled and the hand position is calculated as a result of the movements. An example of this is bending your elbow which causes your hand to move on a curved path. The second method which is a common element in many simulations is **inverse kinematics** (Tolani and Badler, 1996). This is also used in animation and replaying motion capture data. Inverse kinematics is where the end hand position is defined and the joint angles are solved to achieve this position. If you concentrate on

moving your hand in a straight line with the shoulder and elbow following, inverse kinematics is the hand trajectory being set and the joint angles calculated to achieve it. Finding the joint angles to achieve a goal for inverse kinematics is more complex and expensive to calculate than the hand position using forward kinematics (Zhao and Badler, 1994). The reason why inverse kinematics is used is the desired end state of the hand is known but the joint angles to achieve it are not. When moving your hand in a straight line there are many positions for your elbow that will achieve the same hand movement. Constraints are needed to decide between the different trajectories such as avoiding obstacles or minimising the energy of the movement.

3.1.2 Feedforward and feedback control

Motor control is basically performing actions to interact with the world. A model of motor control has two parts: the **motor system** which the controller gives commands to (also sometimes called the **plant**), and the **sensory system** which provides feedback about the motor system state and interactions with the world. The motor controller takes information from the sensory system as input, and provides information to the motor system as output.

In engineering, the problem of motor control is often approached hierarchically. First there is the **planner** which starts with high level goals and turns them into a trajectory to execute. The **controller** then turns this goal trajectory into a sequence of motor commands to deliver to the plant being controlled. For an overview of motor control see Jordan and Wolpert (1999). Many of the papers discussed below follow this engineering structure as a theory of how motor control works in humans.

There are two basic types of motor controller. In a **feedforward controller**, the system performs movements without any information about how the plant is currently affected. Instead, the system uses learned model of the effects brought about by motor commands on the state of the plant. This type of controller is useful where the task is well-known or can be pre-learned. Because the controller does not need to process any unexpected input and the paths are known, the movements can be efficient. The downside is that the controller is not able to adapt to changing circumstances.

A **feedback controller** uses information from the plant to adjust its movements. At its most basic, a feedback controller takes a representation of the system's current motor state, as well as a representation of the goal motor state, and issues a command which will have the effect of reducing the difference between these two states. No learning is required to implement a feedback motor controller: it just needs information

about the current and goal motor states. A feedback controller also has the advantage of being able to react to unexpected circumstances—for instance an unexpected change in the plant's current state. However, there are some disadvantages to feedback control too. Most importantly, there is a lag between the controller giving the commands and receiving feedback about the effects of these commands. This can cause oscillations as the controller tries to correct the previous error. For example, if a joint has not moved far enough the controller can apply more force. The next time step the movement has caused it to overshoot so the controller will apply opposite force to get it back on target. This alternation of applying forces will cause the joint to oscillate around the goal position. One feedback controller which minimises this oscillation is a **PID controller** (Bennett, 2005). A PID controller uses the current state, but also a history of recent events, to calculate the forces required. I will discuss this type of controller in detail when I describe my implementation of a PID controller in Chapter 4.

The task of the control system is managing the transformation between motor and sensory information. When a motor action is performed it produces sensory feedback from its physical interaction with the world. Sensory information has a large delay so an **internal model** can be used to provide faster feedback to make it easier to control rapid movements. Wolpert, Ghahramani, and Jordan (1995) cover two types of internal models: **forward models** and **inverse models**. Forward models can mimic the external interaction of the system, predicting a motor action's effect and sensory feedback. The opposite to forward models are inverse models which predict the motor action required to achieve a specified sensory state. Inverse models can be used directly on a feedforward controller.

A specific example of an inverse model is a **predictive model** which allows an action to be performed without relying on feedback from the sensors. The model predicts what forces are needed to achieve certain sensory results. It does this by taking a copy of the motor commands sent to the muscles and learning the sensory consequences of the actions. This means that the correct forces can be used without relying on feedback to correct them. The inverse model only needs to be an approximation to give good results as feedback from the internal forward model and physical sensory information can be used to correct errors.

Kawato (1990) covers three ways of learning the inverse model. The first is **direct inverse modelling** where it takes the desired trajectory as input and estimates the torque required. This is then compared to the actual torque from the sensory system and the difference is used as an error signal to train the inverse model. The second

is **forward and inverse modelling** which combines a forward model and an inverse model. The forward model learns the system by taking in the motor commands and resulting actions. The inverse model is then given the desired trajectory and predicts the required motor command. This command is given to the forward model which predicts the resulting error. This error is used as the error signal for training the inverse model. The third is **feedback error learning** which uses a **feedback controller** and an inverse model. The feedback controller takes in the trajectory error and turns it into motor error. The inverse model uses this motor error as its error signal and predicts the motor commands required. The combination of the feedback controller's motor error and the inverse model's prediction are then given to the muscles to perform the action. As the inverse model learns, the signal from the feedback controller tends towards zero.

3.2 Neuroscience background

We are trying to build a model which has a basis in the biological motor control system. To do this we need an understanding of the neural pathways involved in reach-to-grasp actions.

3.2.1 The dorsal and ventral visual pathways

The neural pathway involved in motor control basically maps from sensory, and especially visual, areas of the brain, to motor areas. Milner and Goodale (1995) show that there are relatively different independent neural pathways from the visual cortex of the brain to the motor cortex for different patterns of behaviour. This is less obvious in complex animals such as humans where they are not directly linked to motor actions but instead to higher level systems. There are two pathways from the primary visual area V1. One is a 'ventral' pathway running through the lower portion of the brain, specifically the inferior temporal cortex. This is for object recognition, identifying what shape an object is. It focuses on the features of objects and can categorise an object even viewed from different angles and distances. The other is a 'dorsal' pathway running through superior portion of the brain, specifically the posterior parietal cortex, which is focused on the object's location and how to grasp it. Cells in this pathway activate depending on the retinal location and orientation of the object.

The authors focus on a patient with impaired ventral pathway function. The patient could not describe the orientation of a slot but was able to rotate and insert a card into the slot accurately. The patient could also recognise how to grasp an object,

giving similar results to a normal subject. This was compared to another patient with impaired dorsal pathway function. The second patient often grasped the object in unstable ways.

3.2.2 Reach and grasp pathways within the dorsal pathway

Jeannerod (1996) shows that within the dorsal sensorimotor pathways there are two pathways in the brain that are used when picking up an object, the reach and the grasp. The **reach pathway** has the goal of getting the hand to the object. It deals with the object's location in relation to the body. The reach pathway uses the arm joints of the shoulder, elbow and forearm. The **grasp pathway** deals with directly interacting with the object. Its focus is on the type and shape of the object and matching the finger movements to this.

Reach Pathway

The reach pathway's goal is getting the hand to a goal location. To do this it needs to convert the locations of target objects from retinal coordinates to motor commands which bring the hand to the target.

Bullock, Cisek, and Grossberg (1995) review physiological experiments which suggest that the primary motor cortex and parietal cortex are involved with many areas of limb control. Movements can be prepared (primed), and then performed with a later 'go' signal. They identify four functions of the voluntary movement system: continuous trajectory formation; priming, gating and scaling of movement commands; static and inertial load compensation; and proprioception.

Bullock *et al.* (1995) create a model for testing their four functions against experimental data. Their model creates an arm movement vector in the parietal cortex by comparing the perceived current position and the target position. The perceived current position is created with a copy of the commands from the primary motor cortex and error feedback of the actual position. On a goal signal the arm movement vector is projected into the motor cortex.

Kakei, Hoffman, and Strick (2001) performed experiments on the **ventral premotor area** and **primary motor cortex** of a reach action. They trained a monkey to perform wrist movements while holding a handle which could rotate to match the wrist bending horizontally or vertically. In front of the monkey was a screen showing a cursor which matched the monkey's wrist movements and a target. In each trial the

monkey had to match its cursor to the starting target. The target was then moved and the monkey had to move the cursor to the new location. They tested eight different wrist movement directions with three different wrist postures for holding the handle.

Kakei *et al.* found 117 neurons which were task related in that they had a significant change in activity during the action. About half of these coded the direction of the action in all the wrist postures. The directions coded were consistent between the planning and execution stages of the action. Most neurons in the ventral premotor area kept the same direction even when the forearm was rotated 180 degrees. In the primary motor cortex a third of the neurons had a large change in their preferred direction when the wrist position was changed. They suggest that the ventral premotor area neurons are encoding a higher level representation of the action which is then transformed into a motor action. In summary, there is a well-defined neural pathway for generating movements of the wrist and arm from visual inputs.

Grasp Pathway

The grasp pathway turns visual representations of object shapes into hand movements. It needs to learn the actions performed by the fingers as the hand approaches the object. The final state the fingers are in when an agent is holding on to an object must be achieved gradually, while the reach action is underway. The fingers need to open then close in a grasp or pinch action. This is especially true for a curved object such as a cup; the fingers cannot follow the same path if they reach their final position too soon.

Taira, Mine, Georgopoulos, Murata, and Sakata (1990) looked at neurons in the **parietal cortex** in monkeys. Monkeys use their hands to perform a grasp action in a similar way to humans. For the experiment monkeys were trained to manipulate objects which required different types of movement. These were: pulling a knob inside a groove, pulling a lever, pulling a knob and pushing a button. This was called the **object-manipulation** task. They were also trained to look at the object without interacting. The researchers called this **object-fixation**.

The experiment used two lights to instruct the monkey, one in front of its eyes and one on top of the object. The object-manipulation task started when the light in front of the monkey was switched on and the monkey focused on it. When the light changed colour the monkey reached for the object. Once it had grasped the object the second light turned on. When the light on top of the object changed colour the monkey had finished the task. When the monkey was performing the object-fixation task the light shifted from in front of the monkey to the object. The monkey had to stare at the

object until the light changed colour. Some of the tasks were performed in the light and others in the dark.

Taira *et al.* (1990) studied the cells in the posterior parietal cortex while the monkey was performing the actions. They found some cells activated when the monkey performed the manipulation task but not on the fixation task. They called these **Motor dominant** neurons. Others, which they call **visual and motor** neurons, activated more strongly in light than dark when performing an action. Half of these also activated by fixation on the object. A third group only activated during light and activated equally between manipulation and fixation tasks. They called these **visually dominant** neurons. They found that cells usually activated most strongly for a particular type of object. The activity also depended on the orientation of the object. This was shown strongest for the lever. This lead to the conclusion that groups of neurons are activating for a particular type of grasp.

In a similar study Saleh, Takahashi, Amit, and Hatsopoulos (2010) investigated how neurons code the temporal part of the grasp. They trained two monkeys to grasp objects with the right hand. Objects were moved towards the hand simulating the monkey reaching for it. The monkeys pre-shaped their hand and held the object until it was removed. The types of grasp they were trained on were a key grip between the thumb and the hand, a power grasp at two wrist orientations and a power grip with fanning of the fingers. During the grasps they recorded neural data and motion tracking on the fingers.

Saleh *et al.* (2010) found that about half of the neurons they measured had a significantly different activation rate during the movement period. The neurons were encoding the joint angular velocities more often than the joint positions. Their results indicated that the neurons were encoding a grasping action rather than the movement of a single joint.

3.3 Computational models of reaching-to-grasp

Erhan Oztop and Mitsuo Kawato review a number of models for control of grasping (Oztop and Kawato, 2009). They found that there are two choices for representing the arm and hand: treat the arm and hand separately or as a single limb. Arm and hand being independent seems the most likely as it is easier to learn and coordinate simpler controllers than one large limb controller.

3.3.1 A hierarchical neural network model for control and learning of voluntary movement

In (Kawato, Furukawa, and Suzuki, 1987) the authors found that the central nervous system solves three problems in a reach action. First it chooses a trajectory in the visual coordinate space. This must be chosen from all the possible trajectories to complete the task. Second it transforms this trajectory from **visual coordinates** into **body centric coordinates**. This is represented in joint angles or stretch of muscles. Finally it has to generate the motor commands to activate the muscles. Their focus is on the final problem, which they create a **hierarchical neural network model** to solve based on physiological research and previous models.

When a movement is first learned it is performed slowly. This is because it is mainly relying on sensory feedback which has a large delay. The sensor information is measured from the muscles then has to reach the motor cortex before new commands are sent. This feedback control is performed by calculating the planned trajectory with the actual muscle movements. Because of the large delay there is a limit on the speed of the actions for them to be controllable. As a movement is repeated an internal model can be formed. This has smaller delays which means it can achieve better performance. The result is an internal feedback model with the representation of the action and a feedforward model to the muscles.

The model the authors created had both the feedforward and feedback components. Before learning, it relied on feedback and performed tasks slowly. As the internal model was learnt the model could produce different movements and perform tasks faster. The model created an error term which was the difference between the input from the inverse dynamics and the output of the motor system. This was then used as an error term to learn the feedforward function. The inverse dynamics gradually became the dominate controller.

3.3.2 Modeling Parietal-Premotor Interactions in Primate Control of Grasping

Fagg and Arbib (1998) present a model of how the visual system recognises **grasp affordances** which were introduced by Gibson (1977). He found that objects could be interacted with in different ways which he called **affordances**. For example, steps afford stepping with the person's legs and obstacles afford injury. He defines something as having a grasp affordance if it has 'opposite surfaces separated by a distance less

than the span of the hand'. An object having grasp affordances means that there are parameters for motor interaction that are signalled by sensory cues without needing to recognise the object (Explicit object recognition happens in the ventral pathway. People can grasp an object even if they cannot categorise or recognise it.) Fagg and Arbib (1998) used experimental evidence to study neurons discharging when a grasp action is performed. They found that some neurons discharge more strongly when the subject is looking at the object. Other neurons are not affected by visual stimuli and only react to the motor movements. The cells are usually active until the object is released. Most of the neurons only fired for a specific type of grasp such as precision grip, finger prehension or whole hand prehension. Some neurons fire when the subject sees a graspable object while others fire when the subject observes their own or another's grasping action.

The neurological data led the authors to create the **FARS** (Fagg-Arbib-Rizzolatti-Sakata) model. This receives the affordances and grasps for a model from the visual system. The model compares the currently executing grasp to the planned one. The location of the object is not stored in a biologically accurate way as the authors are concentrating on the grasp action. Objects are represented by populations of neurons for each class of object, such as a sphere or a rod. These populations then store the parameters such as the width and height of the specific object of this class.

To perform a grasp, visual information of an object is passed to a model. The affordances of the object are then calculated for the object. From the affordances a grasp is selected. This is based on constraints such as recently executed grasps. The affordance for the grasp is selected and the others are suppressed. The experimenter then triggers the start of the grasp. The model begins by performing the preshape of the hand. Once the fingers are completely open the model starts closing the fingers. The next trigger is when tactile stimulus is detected which stops the closing action and starts the holding action.

3.3.3 Infant learning grasp model

Oztop and Arbib created their **Infant Learning Grasp Model** to try to simulate the way human infants learn to grasp, as revealed in developmental research. Their focus was on the neural networks representing the recognising and storing of grasps. Their 'learning to grasp' model aims to replicate infant grasping behaviour with the neural network design based on monkey neurophysiology data. They aim for a range of grasping behaviours that can adapt quickly without retraining. The system is based

on their previous model (Oztop and Arbib, 2002) which in turn is built on top of the Mirror Neuron System (Oztop, 2002) discussed in chapter 2.

The system uses open loop execution where the grasp is pre-planned before it is executed. This was used because there is evidence suggesting that infants do not use online visual feedback in their grasps. (Clifton, Muir, Ashmead, and Clarkson, 1993) found that infants had a similar result as touching an object in light and dark conditions showing that they were not using the sight of their limb to guide their reaching. (McCarty, Clifton, Ashmead, Lee, and Goubet, 2001) tested infants matching their grip to oriented rods. They found that the infants oriented their hands with similar behaviour in light and dark conditions. Oztop and Arbib's review of the current research shows that infants rely on somatosensory information to improve their reach-to-grasp actions and that visual feedback is built on top of this later. They wanted to see if a grasp could be performed with limited visual processing. They show that infants go through three stages of learning. First they have spontaneous movement. Second they learn to recognise the effects of their actions and third they use what they have learned to plan new actions.

Oztop and Arbib use three controls for their reach-to-grasp. First the hand is moved into position by the arm. The reach moves through a **via-point** on the way to the target object. The arm uses inverse kinematics to plan the trajectory. The goal of the inverse kinematics is the index or middle finger. The second control is the wrist which bends and rotates to line the hand up with the object. This aims to line up one of the **opposition axes** of the hand with one of the opposition axes of the target object. The final part is the fingers curling at a controlled rate along the grasp. This curl motion is set to a fixed rate at the start of a grasp and continues until a contact is made. Once the fingers have a contact the spinal reflex takes over and each finger bends until it touches the object or it reaches its joint limit. They use solid fingers and count each intersection with the object as a point contact. These contacts are used to emulate the somatosensory feedback.

The Infant Learning Grasp Model is not aiming to replicate adult style reaches. The focus is on learning a variety of different grasp actions. Everything is set up at the start of each run of the simulation. The only dynamic action is once contact has been made the fingers bend until they make contact or reach maximum bend. Once the fingers have completed their action the stability of the grasp is calculated. The simulation does not produce forces so a formula is used to calculate the cost of the grasp. Each finger touch gives a contact point which is used in the calculation. Stable,

or near stable grasps, give a positive value. Poor, single contact or missed grasps give a negative value. This value is used as a reward function of how well the action performed and is feedback into the neural network.

The system can take various inputs to set it up. The simplest is simply that there is an object somewhere to grasp. Extra information it can receive is the affordances of the object or its location. Some of the runs have the palm automatically face the object while in others this behaviour had to be learnt. Some runs moved the object each iteration with an oracle telling the simulation of the location.

Results of Oztop and Arbib's simulation indicate that power grasps are the most common initial grasp, which matches experimental infant data. The system learnt different types of grasps in situations which were not suited to a power grasp such as using a precision pinch to pick up a small object from a table. The system learnt to have multiple grasp plans for each object location. They found that palm orientation could be learned and the system did not need to know the affordances to perform a grasp but knowing them did improve the robustness of the grasps. This is similar to the difference between a 5 month old which is grasping using its tactile sensors and a 9 month old which is extracting visual information from the object.

One improvement suggested by Oztop and Arbib is that their model needs a more realistic account of how the arm trajectory is formed. Currently their arm controller is given a trajectory in advance, and simply calculates the sequence of forces required to move the arm along this trajectory. But this assumption of a precomputed trajectory is implausible, as I will describe in the next section. The system they have built on uses a simplified collision model between the hand and the object and ignores gravity. Oztop and Arbib also suggest that a tactile search could be used, but would require a complex model of the fingers which could give mechanoreceptor and indentation feedback.

3.4 Problem: trajectory is always calculated in advance

In all the simulations described in this chapter, the hand's trajectory to the goal is precomputed in advance. But this does not seem to happen in biological reaching-to-grasp. Cisek (2005) argues that neural data does not support a desired trajectory or pre-generated plan. The same populations of neurons in the posterior parietal, premotor and motor cortices are involved in planning a reach and executing it. There seems to be some separation of planning and execution. For example, primary motor cortex

is more involved in execution, but the areas overlap considerably. During both movement planning and movement execution, populations of neurons in the motor pathway appear to represent a general direction rather than a trajectory. Before the movement onset, this direction is often given in a coordinate system centred on the controlled object, rather than in motor coordinates (Sergio and Kalaska, 2002; Cisek, Crammond, and Kalaska, 2003). This means that the initial trajectory of a reach movement is not always in the exact direction of the goal (see Gordon, Ghilardi, and Ghez (1994) for evidence confirming this). Cisek (2005) suggests that movement planning just involves selection of a desired direction in which a controlled object should move, and that the neural computations which determine the effector's actual trajectory mainly occur while the action is underway. A movement starts with the effector moving in the right general direction, with online processes adjusting it correctly based on learned forward models and sensory feedback.

In Section 3.1 I introduced the standard theoretical approach to motor control, in which a planner precomputes a desired trajectory, and a separate motor controller is responsible for generating a sequence of impulses which cause the effector to move along this trajectory. Cisek (2005) argues that studies of neurons in the parietal and premotor cortices do not support this theoretical approach. First, there is substantial overlap between the 'planning' and 'execution' regions of the brain. Second, the 'planning' circuitry in the brain computes a general direction rather than a detailed trajectory. The detailed trajectory of the effector is computed online while the action is performed.

Cisek (2005)'s discussion on whether planning and control are separate in the brain is quite subtle. He considers three possible meanings of 'separate'. In one case, 'separate' means 'implemented in different areas of the brain'. As just discussed, there is not much support for this idea. In another case, 'separate' has a temporal meaning: planning processes precede control processes. As just discussed, there is some basis for this: the nature of the information represented in motor pathways changes before and after movement onset. Finally, 'separate' could mean 'use different coordinate systems'. It is often assumed that planning is done in a coordinate system which abstracts away from low-level motor movements; and indeed, the fact that early representations of intended direction appear to be given in target-centred coordinates seems to support this idea. Further evidence for this kind of separation of planning from motor control comes from the phenomenon of **kinematic invariance**: the observation that a moving hand usually travels along a straight trajectory (Abend, Bizzi, and Morasso, 1982) and has a simple bell-shaped velocity profile in Cartesian space. This has often been used

to support the proposal that trajectories are precomputed. But Cisek (2005) shows that these phenomena can be explained in other ways: if the online motor controller uses a vision-based error signal, and is optimised to minimise this error, then we expect straight Cartesian trajectories, and it has been shown that a controller optimised to minimise variance in the effector's eventual location will have a bell-shaped velocity profile (Harris and Wolpert, 1998).

Cisek (2005) finds an example of evidence for a separation between the planning and controller systems is **motor equivalence** where a high level goal can be achieved with a variety of hand-arm configurations. If you are trying to reach a point then there are many possible rotations of your shoulder and elbow that will achieve the same goal location for the hand. Another piece of evidence is **kinematic invariance** where the hand usually travels along a straight trajectory (Abend *et al.*, 1982) and has gradual acceleration and deceleration. He finds that this only provides evidence for his first point of regional separation and not his second and third points. He also shows that the straight trajectories could have other causes such as using a vision-based error signal for correcting motor errors. One example is Todorov and Jordan (2002) who created a feedback system that optimises for task performance. It reproduced many of the features normally seen in a system with a desired trajectory.

3.5 Aims of my project: a reach-to-grasp algorithm which does not precompute trajectories

As reviewed in this chapter, in current computational models reaching-to-grasp, the hand's trajectory is calculated in advance and executed using reverse kinematics. The only time a grasp is calculated is at the end to measure how well the action performed. But, as also reviewed in this chapter, in biological motor control, there is no precomputation of hand trajectories. One of the aims of my project was to develop a model in which the reach and grasp movements happen dynamically, so that there are forces acting on the hand/arm system throughout the whole simulation, and so that the controller can respond in real time to the current motor state, and to tactile feedback. To do this I needed a way to perform a reach-to-grasp action without precomputing a trajectory. I will discuss my model of reach-to-grasp learning in Chapter 6. In the next chapter, I will introduce my environment for simulating the hand and arm.

Chapter 4

A new platform for simulating hand-arm actions

In this chapter I describe my platform for performing reach-to-grasp simulations. In Section 4.1 I will outline the goals I wanted to achieve with this platform. In Section 4.2 I will give an overview of the 3rd party components that make up the platform, show how how the underlying physics algorithm works, and describe how I have modeled muscles in the simulation. In Section 4.3 I will describe how I have represented the arm and hand. In Section 4.4 I will overview the software architecture which the platform uses, and I will give details of the implementation in Sections 4.5 and 4.6. In Section 4.7 I will outline how this architecture is used to create a simulation. In Section 4.8 I will describe the manual user interface to the platform, which allows a human user to control the hand and arm.

4.1 Goal of the platform: to use an 'off the shelf' physics engine

As already discussed, one of the key aims for this project was to build a platform on top on an existing physics engine. An advantage of using an existing underlying platform is that features that would normally be excluded because they were not worth the extra effort were already included. This meant that my platform could use real time 3D graphics along with the physics. The goal of the system is to perform a series of simulations and store the data about how these runs performed. The design of the system is based on extensibility, with the different components built on top of common interfaces to allow new types of components to easily integrate with the system.

4.2 Choosing a physics engine and graphical front end: JMonkey, OpenGL and Bullet

To build a simulation of reaching-to-grasp, there are two separate problems. First, the actual physics of the system needs to be simulated. This includes calculating all the forces, moving the objects based on these forces and taking into account the possibility of collisions between objects. The second problem is to display what the simulation is doing. One of my main research goals is to create a model which learns to control the hand-arm system to generate successful reach-to-grasp actions. There are several off-the-shelf platforms which deal with these problems separately. There are many free physics engines such as ODE (Smith, 2000) and Bullet (Coumans, 2007). To view what is happening the physics engine then needs to be linked to a 3D renderer, such as Open Graphics Library (OpenGL, 1992), to show the simulation on the screen. One area where both these technologies are combined is in game engines (Wiki, 2010). The game engine that I chose was JMonkey. This gives an interface to both the graphics and physics and means I can interact with one object which is used in both the physics calculations and rendered on the screen.

The JMonkey Engine (jme2, 2009) is written in Java and uses Open GL for the graphics rendering and Bullet for the physical simulation. The basic unit of representation in JMonkey is the **node**; a simple rigid body with a defined shape and mass. A node is the object that the user of the engine interacts with. It is rendered on the screen and used by the simulation, so what you see is also what the physics is interacting with. Nodes are joined together with joints, which are not displayed on the screen. These have defined degrees of freedom and pivot points.

The user sets up the nodes and joints, then runs the simulation. While the simulation is running, the user can change the forces acting on joints to interact with the objects. The node's movements are constrained by the connecting joints. If two nodes enter the same space this is a collision and the physics prevents them moving through each other. The nodes and joints are the base which my platform is built on.

4.2.1 Physics in Bullet: Rigid body dynamics

The physics engine does a lot of work behind the scenes. Understanding what it is doing will show why it is needed. The most basic concept that the physics engine is dealing with is objects moving over time. This is called **kinematics** as covered

in Section 3.1.1. The rules that cause the objects to move are called the **dynamics**. These are the forces acting on them, such as gravity and the masses of the objects. The type of objects we are using are called **rigid bodies**. This means that their shape stays the same. This type of physics simulation is called Rigid Body Dynamics, (see for example (Hecker, 1997)). I will state the ideas informally with the formulae in the appendix.

Calculating the new position of an object is based on integration. If the acceleration of an object is known, integrating this will give the velocity. Integrating its velocity will then give its new position. This process happens at each time step for each object. The three values required are its acceleration, which is caused by the forces, its current position, and the acceleration which was calculated in the previous time step.

Calculating the acceleration requires the mass and force. The mass times the velocity is called the momentum. To simplify the equations a centre of mass is used so equations can use single point. To find the acceleration, the force is divided by the mass. This gives the acceleration at the centre of mass. The force value will be the sum of all the force vectors acting on the object.

The type of physics I am using is called rigid body dynamics. This has two types of objects, nodes and joints. Nodes are the parts you can see. These are made up of geometric shapes such as a sphere or a cube. It is also possible to create more complex shapes out of triangles but these are more computationally expensive. Joints are used to connect the nodes together. There are joints with different degrees of freedom, such as a hinge joint which only bends on one axis, and a ball joint which rotates on two.

When the simulation runs, the engine is calculating the forces acting on each node and moving them appropriately. The joints act as a constraint on where the nodes can go. Sometimes these constraints cannot all be simultaneously satisfied, which leads to unexpected results. This is often due to setting up the joints in an unfeasible way or using unrealistic forces. When two objects hit each other, there is a collision. The engine then changes the forces for each object in the collision.

4.2.2 Modeling muscles as springs

In Bullet you use a force or a motor to change the angle of a joint. If a force is applied to a node, it will affect the joints connected to it. This makes it difficult to control the joint as it is hard to apply a constant force to a node's rotational direction. The second method is through a motor. Some joints in Bullet, such as a hinge joint, have a motor function. This lets you specify the goal speed you want the joint to be rotating

at and what acceleration towards this goal you want. This gives accurate control over the joint but does not relate to human joints which have muscles.

One way of representing muscles is to imagine them as springs (see for example (Lindstedt, Reich, Keiml, and LaStayo, 2002)). When you want to change the angle of a joint you change the resting state of the spring. When the spring is not at its resting state it will accelerate towards this position. I used this idea to abstract away the motors in the physics engine. The program interacts with a joint that acts like a spring. The joint then does the conversion into a motor velocity and acceleration which the physics engine uses. This lets the program use joints as springs by saying where it would like the joint to be. The joint then uses the difference between its current position and the goal position to work out the velocity and acceleration to give to the motor at each time step. The system allows various implementations of the spring to be used. The current spring function is where the strength is proportional to the distance from the resting angle.

4.2.3 A Basic PID Controller

A simple controller such as a spring simply uses the difference between its current state and the goal state as the input for the movement. One problem with the springs is that they do not decelerate until they have passed the goal location so they can overshoot. This leads to the joint oscillating around the resting state until it loses enough momentum to settle in position. Another problem is if there is a small error which does not cause enough input to the motor system then the controller will never reach its goal. This behaviour is not what a human arm does as it moves into the correct position and stops. To achieve this behaviour, I replaced the spring model with a PID controller (see for example (Bennett, 2005)). This uses three control components to decide on the force at each time step. The P is the proportional component. This is the difference between the current position and the goal position. The I is the integral component which is the sum of all the differences so far. The D is the derivative component which is the change in position from the last time step. Using only the proportional step gives the same result as a spring. The integral makes sure that the joint will achieve the goal. In a situation where the proportional difference does not give enough force to reach the goal over time these will add up, increasing the integral term and getting the joint to the right position. The derivative term is to slow the acceleration as it approaches the goal to minimise the overshoot and stop the joint oscillating around the goal position. In testing I found that the integral term quickly

became the dominant term in the equation giving unrealistic results. Using just the proportional and derivative terms gives acceptable results so I am using a simple PD controller. The parameters of the PD controller are set so that the hand reaches its target without overshooting. These parameters are shown in Appendix A.2.

4.3 Core plant

The goal of the system is to simulate a reach-to-grasp action so the core of the system is the physical model of the hand and arm. The basic structure of the plant is created in two classes: the reach and the grasp, which represent your arm and your hand. The reach consists of the physics objects responsible for getting the grasp into the right position. There are currently a variety of different configurations representing arms with differing degrees of freedom. The only requirement for the reach is that it has a connection for the grasp at the end. The second part is the grasp which is involved with holding the object. This connects onto the reach and is where the touch sensors are located.

The reason why these two classes are separated rather than being one system is so that different types of components can be mixed and matched. The reach can be a simple arm which only moves at the shoulder or one with many degrees of freedom representing the elbow, the forearm twisting and the wrist. The grasp component can range from a ball which only detects touches or a claw with one movement to a representation of a human hand with deformable fingers. By using these two interfaces new types of reach and grasp components can be easily integrated with the rest of the system.

The implementations of the reach and grasp components are built out of objects called **parts**. These interact directly with the underlying JMonkey physics engine. Parts are all stretched cubes to minimise the number of sides. (Each side is dealt with separately in the physics. A sphere is made from many small triangles so is more expensive to use because the physics engine is comparing triangles rather than spheres.)

Parts are held together with joints. As described above, in the physics engine, joints are associated with motors, which I control by modelling them as springs. The control system changes the resting state of the spring. This then activates the motor with dynamic acceleration depending on the difference between the current state and the desired state. The joints have the greatest acceleration when the resting state has changed and there is a large distance between the current state and the desired

state. As the current state nears the end state the acceleration decreases, like an actual muscle.

4.3.1 My reach-grasp implementation

I implemented two separate physical plants for reaching-to-grasp, for different purposes. One plant features a very simple claw-like grasper, illustrated in Figure 4.1. In this system, the reach is a simple shoulder and wrist twist to achieve different positions for the grasp. The grasp is made from two opposing fingers which can be solid or have soft pads. This plant is used for demonstrating the properties of my model of soft fingers in a very controlled reach/grasp movement.

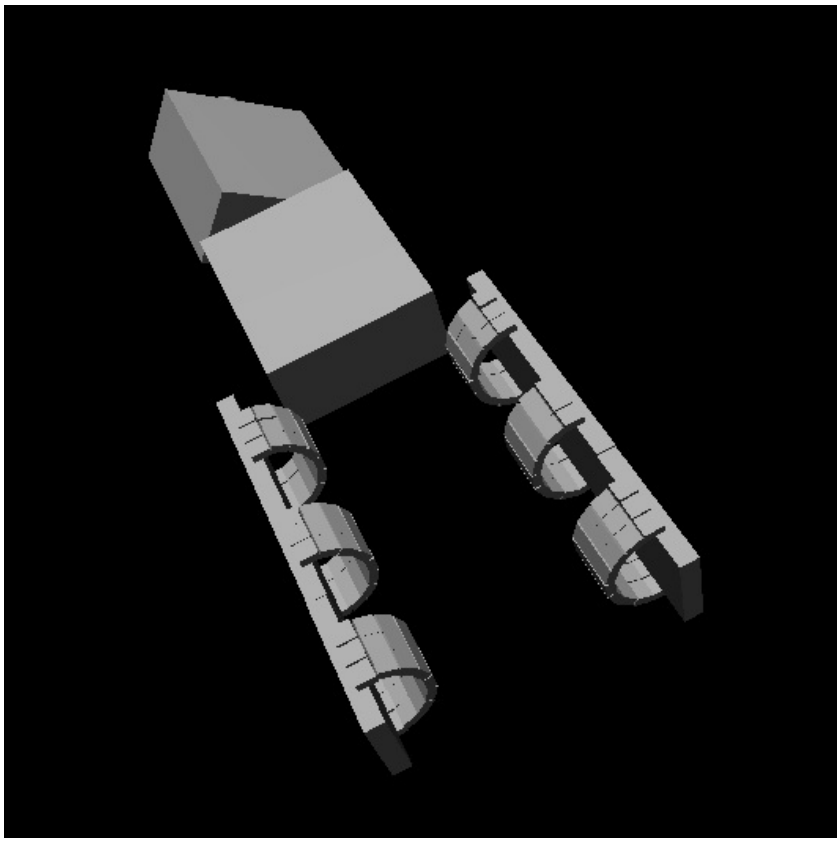

Figure 4.1: A claw with soft finger pads

The second plant is a more human-like reach-to-grasp system; it is illustrated in Figure 4.2. This system is used for all my models of reach-to-grasp learning. In this system, the reach uses four degrees of freedom: the shoulder, the elbow, the forearm twist and bending the wrist. The first two degrees of freedom are to get the grasp into

the correct position. The second two are to get the grasp into the right orientation. The grasp is based on a human hand with four fingers and a thumb. The fingers are all the same length to simplify the power grasp, meaning that for a cylinder, all the fingers can be bent into the same plane. Because the fingers on a human hand are different lengths they have to be bent at different angles to touch the same place on a cylinder. The grasp has two degrees of freedom: bending the fingers and curling the last joint of the fingers. The thumb and fingers work off the same goals. This hand can execute a power grasp with all fingers or a precision pinch using only the forefinger and the thumb.

The mass of each part of the arm and hand is weighted to approximate the relative masses of a real arm. The masses were modified to give the best performance in the implementation as the physics engine had some unexpected behaviour which put a constraint on the possible masses. If a part was too heavy, it stretched the joints on an axis they did not bend on, such as the elbow getting stretched downward. If a part was too light, when it collided with another object the force would be too great compared to its mass and fly away. These masses are shown in Appendix A.1.

4.4 Software architecture

The aim of the system is to build on top of an 'off the shelf' game engine and be modular and extensible. To achieve this, the code is broken into a number of modules shown in Figure 4.3. These different component modules have a single task and are loosely coupled following Object Oriented design principles. The way these different modules interact is through **directors**, which are discussed in the following section.

4.5 Directors

For the system around the core plant I aimed for a modular, event driven architecture. To achieve this I organised the activities into a series of **directors**. Each director is responsible for one aspect of the platform such as the state or the score. They are event driven and are composed dynamically at runtime. Some directors have interfaces associated with them. A class which implements this interface can interact with the director. This allows new classes to be created and work within the existing framework. The interfaces for each director will be detailed in the following sections.

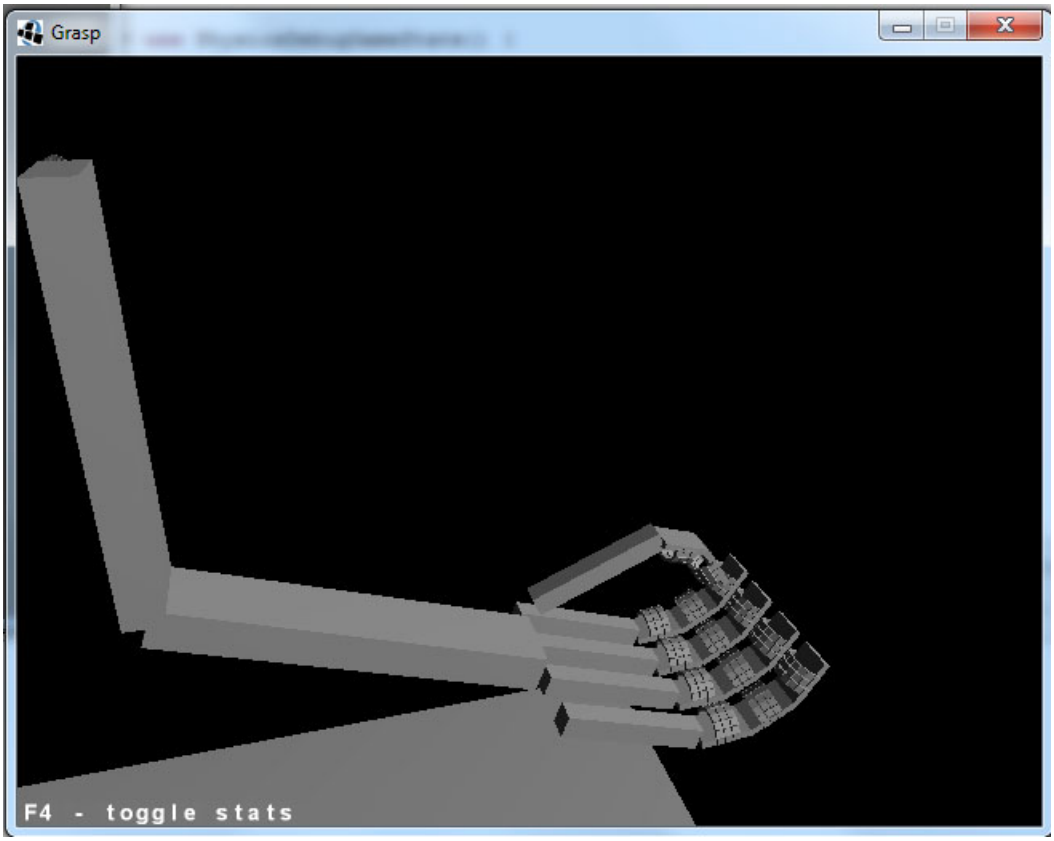

Figure 4.2: Arm and hand with soft finger pads

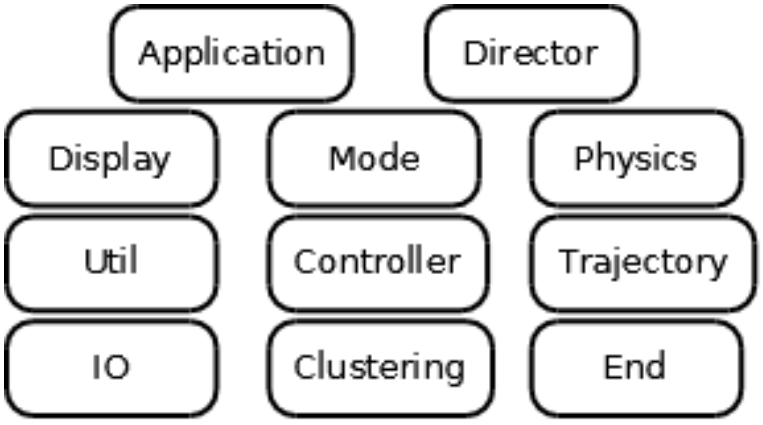

Figure 4.3: The different components which make up the framework

4.5.1 Update director

The underlying JMonkey game engine works on update cycles. Each cycle updates the 3D models and refreshes the screen. It also performs the physics calculations and moves the physics objects. The system interacts with this cycle through the **update director**. The basic design of the system is that a goal location is set, the plant moves towards it, then the goal may change. Currently this is represented by three update points in the update director: Goal, Current and Observer. Components which are involved with setting up a goal location are updated first. Next components involved with the current movement towards the goal are updated. Finally, components which are observing the system are updated last. This ensures that new goal locations are set before the components try to reach them and that movements have happened before the observers report on it.

Each class that needs to be updated implements the 'UpdateCycle' interface. This defines when the class will be updated. It also provides a method to be implemented which defines the behaviour at each update cycle. Classes which need to be updated are dynamically added to the Update Director at the start of each simulation run. This allows new classes to be added to the system and participate in the existing update cycle.

The update cycles happen for each component of the plant. For example, first the controller in charge of the reach updates its goal location. Then the controller in charge of the grasp updates its location. Next the arm moves towards the goal it has been set. The grasp then tries to reach its goal. Finally, the observers record the movement and touches are updated. This system allows components to enter and exit the update cycle dynamically. It also allows new update points to be added if additional steps are required. The method in which these goal and current states are sent out is controlled by the director in the following section. The learning routine for the system requires the arm to make a large number of movements. Each attempt to reach the goal is called an iteration.

4.5.2 State Director

The **state director** is involved with passing the state of the system between the various components. States store three values: which component is involved, what the type is such as goal or current and what the value is. All the values are stored as an abstract 'Joint Proportion' so that different components have a common interface

for passing information. There are two interfaces associated with the state director, the 'state listener' and the 'state creator'. Components which implement the state creator interface can create new state information and send it to the state director. Components can also implement the state listener interface to register with the state director to receive information. They can get information about states for a component with a specified type and can register for multiple types of information. For example, the trajectory displays receive both the current and goal state for the component they are representing. Components can also implement both interfaces to be a listener and a creator of state. The core plant components listen for a new goal state, perform their update, then send out the new current state.

The aim of the state director is to encapsulate each component from the other components interested in its updates. Rather than the components being directly coupled together they all only interact with the state director. Components add themselves as a listener to the director waiting for a certain type of state and do not need to know which component the new state is coming from. This allows different creators, such as a new type of grasp, to be swapped in and the state displays still show the state changing. It also allows new components, such as a different type of display, to be added to the system and interact with the existing framework.

4.5.3 Score director

The **score director** manages components interested in how well the grasp is performing. The score represents how good the grip on the object is. Classes can implement the **score listener** interface to receive new scores. They can also give a new score to the Score Director. This will then be passed on to all the listeners.

The grasp gets the score information directly from the collision detection module of the physics engine. The details of the score depend on the implementation of the grasp. Each grasp type has an associated score which matches the physics of the grasp. The solid fingers can show what sections of the finger are colliding with an object whereas the soft fingers can show the amount of deformation currently happening giving an idea of where the collision is happening and the shape. These specific score types implement the interface 'Score' which abstracts away the details allowing components to deal with one common score type. This is used by some controllers to tell how well the grasp is performing, the score display which shows the haptic information and the trajectories which record how well an interaction performed.

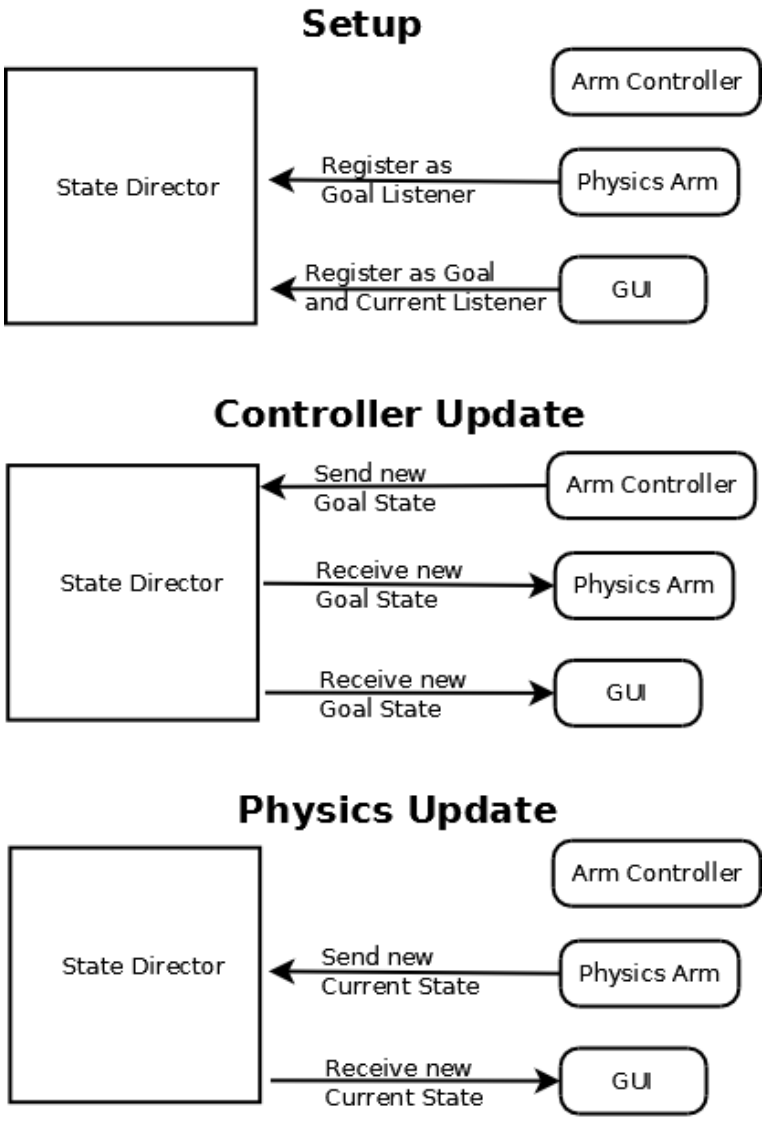

Figure 4.4: The interaction with the state director. In the 'Setup' phase the Physics arm and GUI register as listeners to the state director. In the 'Controller Update' phase the Arm controller generates a new goal state and sends it to the state director. The state director then sends it to all the listeners registered for goal states. In the 'Physics Update' the Physics arm sends a new current state to the state director. This current state is then sent to the listeners registered for current state, in this case just the GUI.

4.5.4 Other Directors

The **display director** manages the various information panels that are created. It interacts with classes extending the 'Grasp Panel' parent. The display director ensures that the panels are created in a grid so that they do not overlap. It also ensures that when a run of the simulation ends the panels associated with that run are properly closed. The panels it is involved with are the haptic displays and joint movements.

The **storage director** is for storing the information about a run. It holds items during the run so they can receive updates then gives them to longer term storage after the run has completed. The director interacts with objects which have implemented the 'Storable' interface. One component that implements this interface is the 'Trajectories'. These store information about a specific component during an iteration. They receive the goal and current states, and the scores at each time step to have an accurate picture about the state of the system. These can be used to recreate a run iteration if necessary, or serve as a basis for a new run.

The 'deletion director' handles resetting the system between iterations. Some components are deleted while others, such as the directors, are kept but have their data cleared. Classes implement the 'Reset' interface then add themselves to the director. The interface has a method which the class implements to free all of its resources. This is especially important with classes interacting directly with the physics engine. Some of the JMonkey resources are low level interacting outside the JVM such as the OpenGL graphics. These need to be freed to prevent a memory leak. When an iteration ends, the reset director calls the reset method on each of its components clearing everything ready for the next run.

4.6 Components

The framework is made up of various components, each performing a specific role. The directors allow the various components to interact without being coupled together. I will now cover the components which make up the system.

4.6.1 Application

The 'application component' is the start of the user's interaction with the system. The screen presented to the user shown in Figure 4.5 allows the user to choose what type of finger surface they will use and what task the simulation will perform. This can be

the full learning task which will learn to grasp an object (which will be described in Section 6.5) or various more specialised demonstration tasks which allow the user to see a specific feature of the framework.

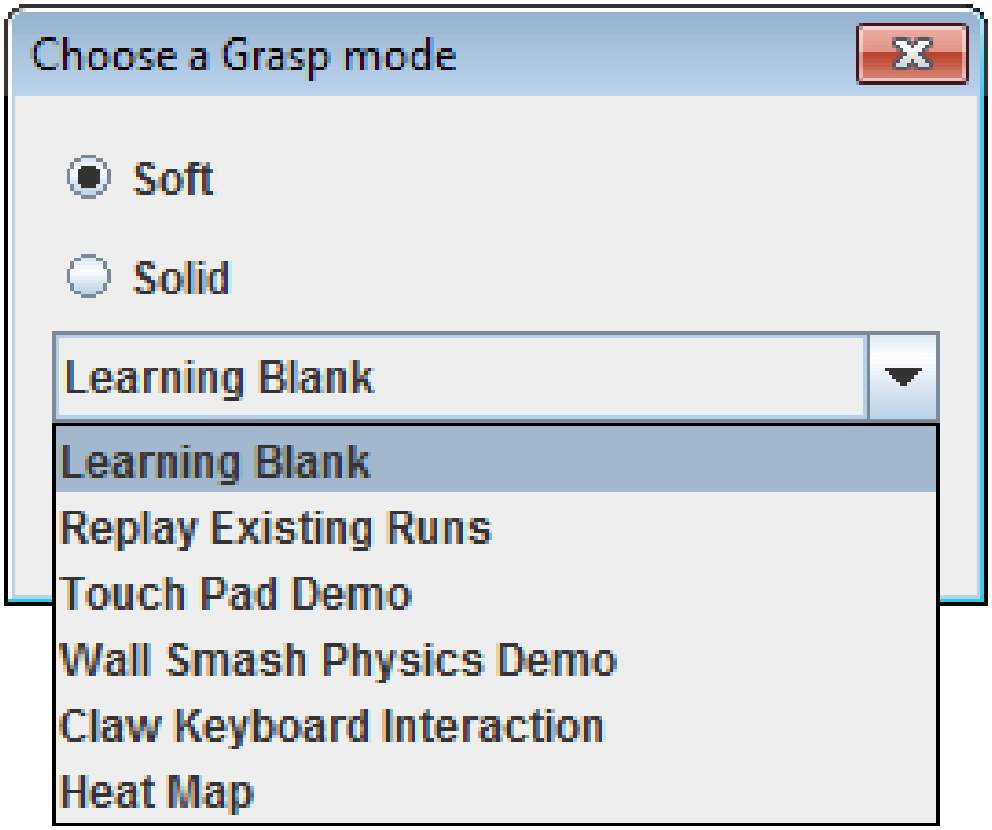

Figure 4.5: The application screen which allows the user to define what finger surface to use and what mode the simulation will run in.

4.6.2 Physics

The physics component holds all the classes interacting with the underlying physics described in Section 4.3. These are arranged in a tree structure with abstract 'reach' and 'grasp' classes at the top which interact with the directors working down to the specific implementations of different types of graspers and arms. The bottom layer is the interface between the framework code and the underlying JBullet physics, where the actual physics nodes are created and the collisions are recorded. All of the interaction with the lower level components of JMonkey are encapsulated in this component.

4.6.3 Modes

The modes component holds the different types of learning algorithms. Modes are a high level representation of what type of action is being performed such as exploration or improving on a previous iteration. One mode is chosen at the start of each iteration and sets the behaviour during the iteration. It chooses which 'controllers' to use to implement the behaviour. This is covered in the following section.

4.6.4 Controller

The controllers control the behaviour of the physics components during the simulation. They create new goal states for the components to reach and can receive information about the current state of the system. The different types of controllers are covered in more detail in 6.4.3.

4.6.5 Trajectory

The trajectory component is for storing information about a run. It receives information about when a new goal state is created and when the current state changes. It also records the score. These are then used as a basis to improve upon. These trajectories are stored in memory while the simulation is running.

4.6.6 Display

The display component holds all the graphical user interface objects. These display various information to the user such as the current goal and current states for each component represented in an abstract joint space, the touches for a finger pad and the representation of stored trajectories. These use common themes to achieve a consistent look and feel, and receive their information from the directors.

4.6.7 Util, IO and End

The final components are utility components to help the system. The Util component contains various common methods and constant values used throughout the system. The IO component deals with taking items stored in memory and writing them out to disk. It also handles reading the items back in when the application is started. This is all encapsulated into one place rather than the components doing it themselves so that different IO routines can be introduced. If the system was extended to use a database

instead of writing directly to disk, only this component would need to be upgraded. The End component stores the information about when to finish a simulation cycle. Depending on the stage of the learning the run may end once a certain score threshold has been reached or too many update cycles have happened. These end conditions are stored at the start of each run and the end component records the various parameters until an end condition is met. This then ends the simulation run and begins a new one.

4.7 Using the platform

In the following I will show how to set up the simulation. For access to the code and installation instructions visit http://graspproject.wikispaces.com/Installation .

The two types of JMonkey objects that the simulation interacts with directly are the physics space and the root node. The physics space is where all the objects handled by the physics engine are stored. These represent objects which will be affected by gravity and collide with other objects. The root node is the top of the rendering tree. It defines what objects are going to be displayed on the screen. The simulation does the actual interaction with these objects at a low level ensuring that the physics objects and rendered objects are the same so that what you see on the screen is what is happening in the physics. Basically these two objects are passed downward to the lowest level simulation object of 'Part' which performs the translation between the simulation parameters and JMonkey.

Creating a new system looks like this:

```
Reach reach = new ArmFactory().createReach(pSpace, rootNode);
Grasp grasp = new HandSoftPowerGraspFactory().createGrasp(reach, pSpace, rootNode);

creatExploreController(ComponentType.ARM);
creatExploreController(ComponentType.WRIST);
creatPerturbationController(ComponentType.FINGERS);

createDisplay(ComponentType.ARM);
createDisplay(ComponentType.WRIST);
createDisplay(ComponentType.FINGERS);

createTrajectories();
```

```
setupEnd();
```

First the factories create the reach and the grasp. The factory first creates the reach component which will create the physics objects. The factory then adds the components to the update director and the state director.

```
final Reach reach = new Arm(pSpace, rootNode);
UpdateDirector.addUpdater(reach, ComponentType.ARM);
UpdateDirector.addUpdater(reach, ComponentType.WRIST);

StateDirector.addListener(ComponentType.ARM, reach,
        StateType.GOAL);
StateDirector.addListener(ComponentType.WRIST, reach,
        StateType.GOAL);
```

```
final Grasp hand =
        new Hand(reach.getGrasperConnection(), rootNode,
                pSpace, SurfaceType.SOFT, false);
UpdateDirector.addUpdater(hand, ComponentType.FINGERS);
StateDirector.addListener(ComponentType.FINGERS, hand,
        StateType.GOAL);
```

The controllers are then created for each component. In this case they are all using StepExplorationControllers. The controller is created then added to the update director.

```
Controller controller =
        new StepExplorationController(componentType);

UpdateDirector.addUpdater(controller, componentType);
```

If the controller required the current position of the reach that would go in here also.

```
StateDirector.addListener(ComponentType.FINGERS,
        controller, StateType.CURRENT);
```

The trajectory set keeps track of the current state of the simulation to be stored. It does not directly interact but listens to the various directors for changes.

```
TrajectorySet trajectorySet =
        new TrajectorySet(ComponentType.ARM,
                ComponentType.WRIST, ComponentType.FINGERS);
StorageDirector.addStorable(trajectorySet);

StateDirector.addListener(ComponentType.ARM, trajectorySet,
        StateType.CURRENT);
StateDirector.addListener(ComponentType.ARM, trajectorySet,
        StateType.GOAL);

StateDirector.addListener(ComponentType.WRIST, trajectorySet,
        StateType.CURRENT);
StateDirector.addListener(ComponentType.WRIST, trajectorySet,
        StateType.GOAL);

StateDirector.addListener(ComponentType.FINGERS,
        trajectorySet, StateType.CURRENT);
StateDirector.addListener(ComponentType.FINGERS,
        trajectorySet, StateType.GOAL);
ScoreDirector.addListener(trajectorySet);
UpdateDirector.addUpdater(trajectorySet,
        ComponentType.OBSERVER);
```

Finally, the end condition waits for a trigger to end the current run of the simulation.

```
EndCondition endCondition = new EndCondition(EndCondition.SHORT\_UPDATES, EndCondit
        ScoreDirector.addListener(endCondition);
         UpdateDirector.addUpdater(endCondition, ComponentType.OBSERVER);
```

4.8 Manual interaction with the model: A graphical user interface

My initial experiments with the model used a mode where the joints are directly controlled by the user. The user interface consists of a series of 2D panes which each represent a two-dimensional joint space. Each axis of a joint has an axis in the pane.

For example, in the shoulder/elbow controller pane, movement along the X axis represents the rotating of the shoulder and movement in the Y axis represents the bending of the elbow. Three stages of this bending can be see in Figure 4.6. Any position of these two joints is represented as a coordinate in the pane. To change the position of the joints, the user selects a point in the pane with a mouse click. This point represents the goal motor state. Once the goal is chosen, the joints will rotate to the new position through the physics of the PD controller. A trail from the current state to the goal state is drawn as the joints move through their rotation to show the trajectory, in joint space, that they have taken. A demonstration of the manual interaction can be found at http://graspproject.wikispaces.com/Movies 'Manual Interaction'.

4.9 Summary

In this chapter I have shown how I interact with the underlying physics engine, the architecture of the platform and the purpose of the different components and how to interact with the platform.

In the next chapter I will describe some of the main innovative features of the architecture: The models of the soft fingers and the mechanoreceptors.

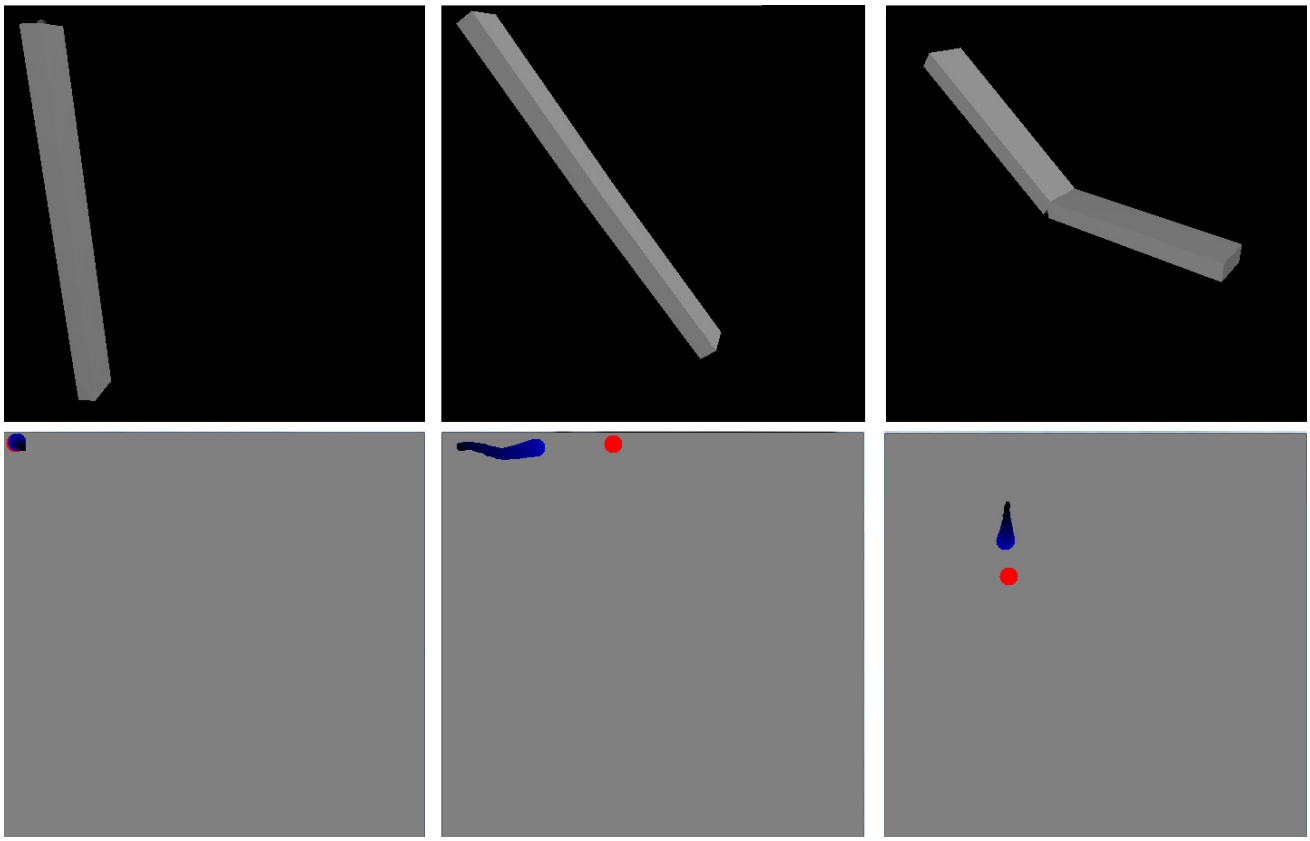

Figure 4.6: This shows three stages of a shoulder/elbow movement. The top row shows a graphical representation of the upper and lower arm. The bottom row shows the 2D controller pane for the shoulder and elbow in joint space. The red circle is the goal state and the blue cone is the trajectory of the current state. In the left hand pane the goal state and current state are the left top corner. This is where the system is at rest. The middle pane shows the goal state moved to a new position in the X axis. This represents a rotation of the shoulder. The blue trajectory shows the system moving towards the new goal state. The right pane shows a the goal state moved to a new position in the Y axis. This represents a rotation of the elbow.

Chapter 5

A model of soft fingers and mechanoreceptors within the new platform

In Section 5.1 I will show my novel implementation of 'soft fingers' and demonstrate them against solid fingers. In Section 5.2 I will show my novel implementation of mechanoreceptors in the skin, based on the soft fingers.

5.1 Soft fingers

One of the aims of this project is to attempt to represent the tactile system of the hand reasonably accurately. I wanted an accurate model of the different types of mechanoreceptor to show how they are used in making a grasp. In this section I will show my model for representing soft touches.

5.1.1 A new proposal for modelling soft fingers: modelling the skin as an articulated rigid body

My idea was to adapt a method for modelling deformation used in computer graphics. Rivers and James's (2007) approach to deformation is to take an arbitrary shape and approximate it with a lattice of cubes as shown in Figure 5.1. They then apply deformations to the lattice by interacting with the vertices of the cubes. These deformations are then applied to the more complex original mesh.

In my implementation I create a lattice of small cubes to represent the finger pads.

Because of the underlying physics engine I cannot directly change the vertices so the lattice is implemented by linking the cubes with springs which allows it to bend. I call the cubes in the lattice **tiles**.

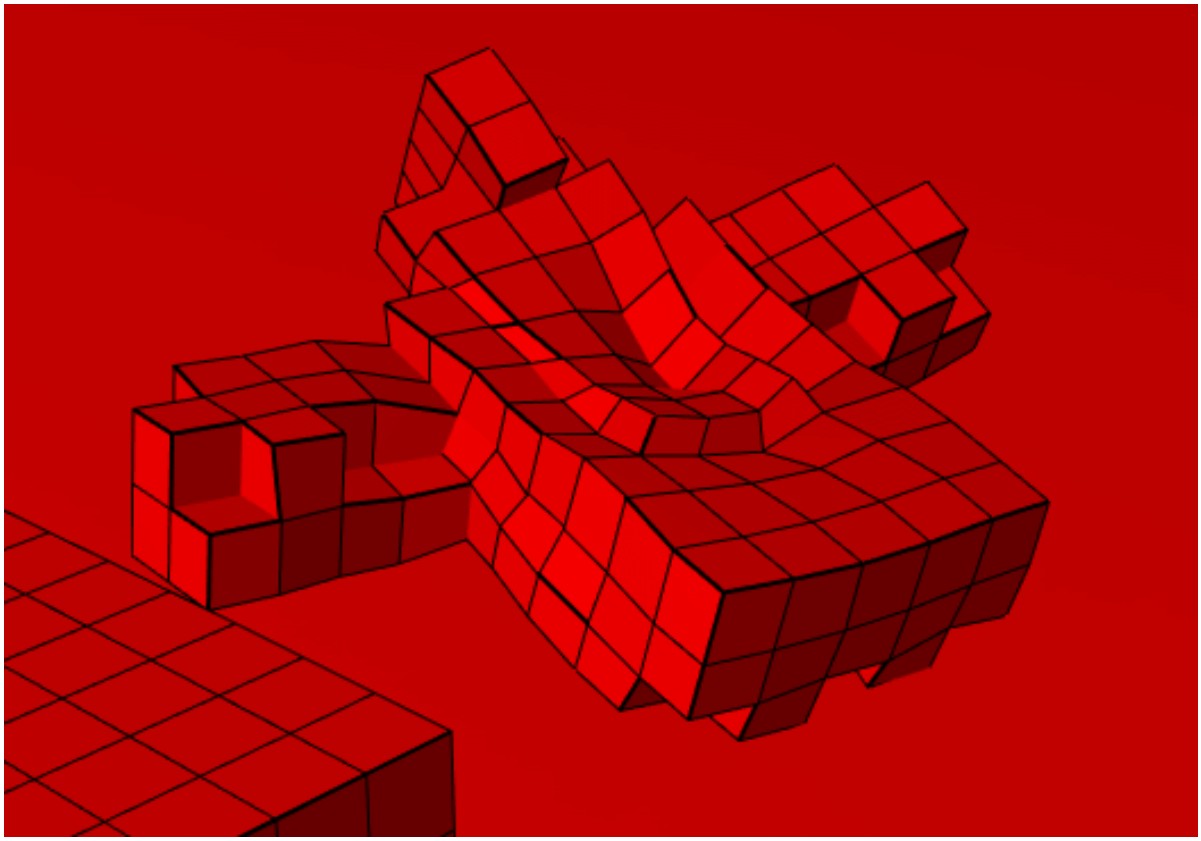

Figure 5.1: A lattice of cubes to approximate deformations in the FastLSM 3D Demo Rivers and James (2007)

Tiles have a collision mask so they do not collide with their neighbours, providing a continuous surface for the finger. The underlying physics engine handles all of the collisions while giving the benefits of soft fingers. This implementation also allows the user to directly see how the finger is being deformed. My implementation can scale to any size such as Figure 5.2 but for my soft fingers I have used nine tiles across and three high. Each tile is joined to its neighbours resulting in up to four connections.

5.1.2 Finger pads

My model of finger pads aims to get the advantages of soft fingers while still using the rigid body dynamics physics. The key effect is shown in Figure 5.3, where it can be

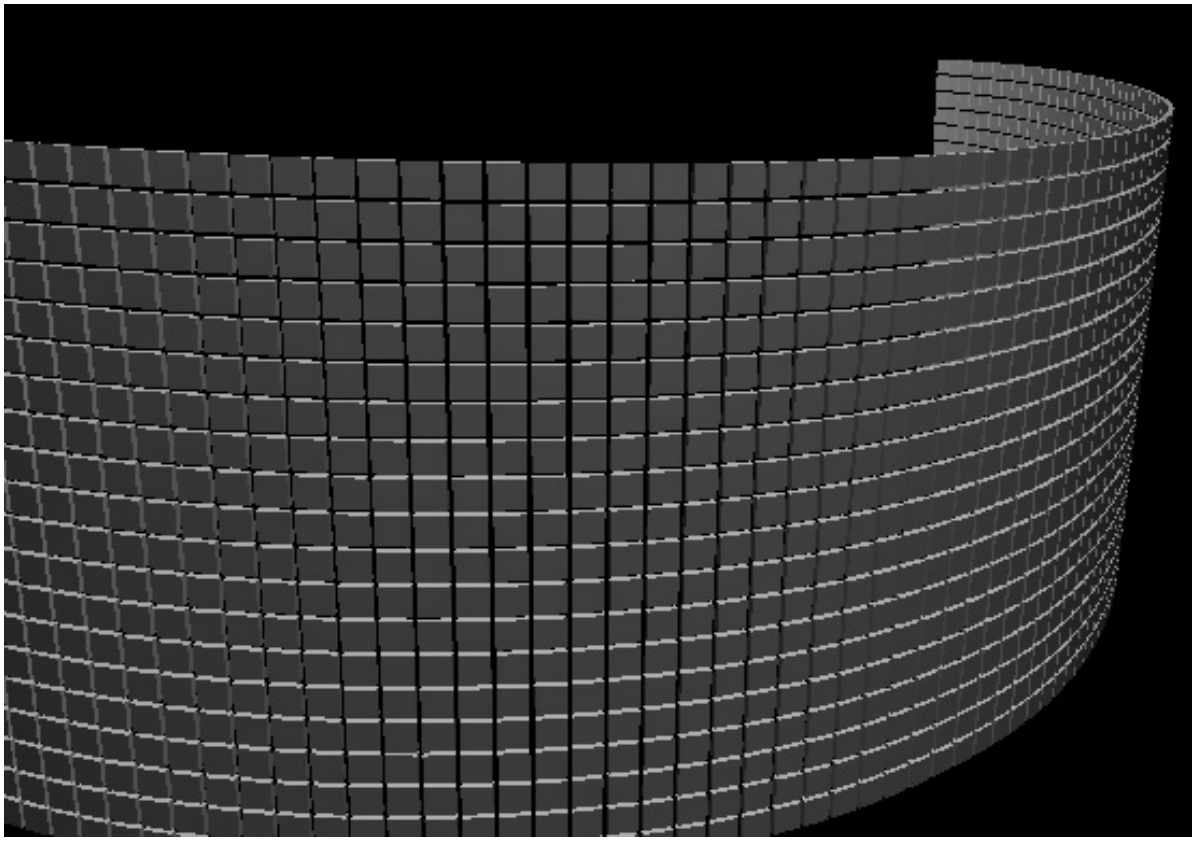

Figure 5.2: A mesh of 100 by 20 tiles

seen that the pad deforms when making contact with an object. As the tiles touch an object the springs bend, and the curve of the finger pad is deformed to match, or approximate, the shape of the object. The nodes on the finger can intersect or go through each other to give a continuous surface. The springs in the fingers are weak enough to bend around an object. When the object is removed they will shift back into their original shape. In Section 5.1.3 I will demonstrate the difference between these soft fingers and hard fingers made from a single node.

Figure 5.3: Finger pad bending during forceful contact with a flat surface. Note that the finger pad becomes (approximately) flat.

5.1.3 Demonstration of hard vs soft fingers

To explore the properties of hard and soft fingers, I constructed a demonstration in which the simple claw grasper attempts to hold a sphere with either solid or soft fingers. The simulation has three stages and runs for 20 seconds. During the first six seconds, the open claw is moved into a prespecified resting position in relation to the target. The

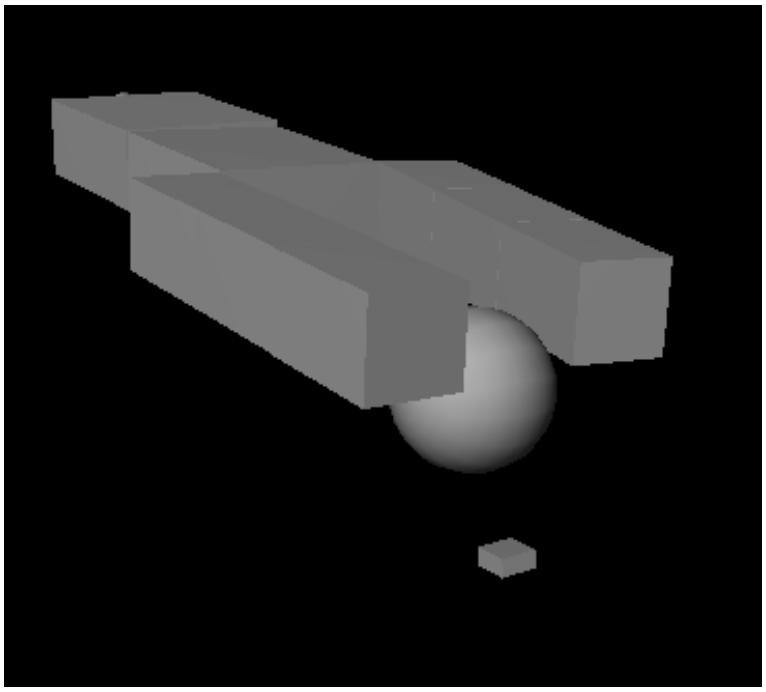

Figure 5.4: Solid fingers unable to maintain a grasp on the sphere. The small cube is the platform which the sphere started on, showing how far it has been lifted before it was dropped.

position is systematically varied to explore the effect of grasps in different positions. The grasp then receives a new goal location for the fingers to move to the centre of the claw causing them to squeeze. This runs for four seconds to allow a grasp to be made. In the final ten seconds the arm moves slowly upwards and a timer checks how long a grasp is maintained. In this last stage, we are testing for a stable grasp: it is important that the hand can move while holding the target when it is dislodged from its pedestal. If the sphere is held for the full ten seconds then it is rated as a stable grasp. Otherwise the score is the amount of time the sphere was held before it was dropped.

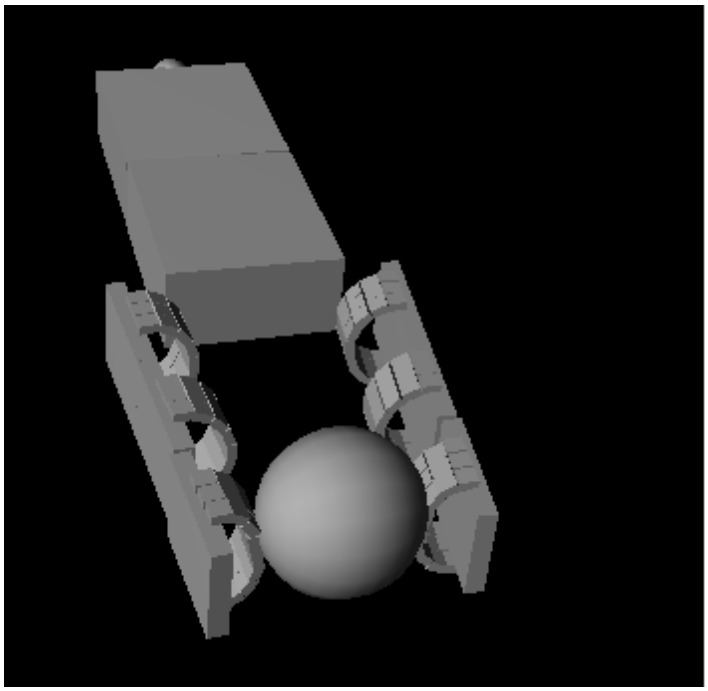

Figure 5.5: The soft fingers holding the sphere by bending the finger pads around its surface to maintain a grasp.

The simulation is run multiple times with the specified resting position changing each iteration. The starting position starts below and to the left of the sphere and moves through a grid of positions, finishing in a resting position above and to the right of the sphere. This is to find all possible starting locations that result in a grip on the sphere. The runs are then graphed as shown in Figure 5.7. The left image shows the runs for the solid fingers and the right image the runs for the soft fingers. Each square is one run with the x and y positions on the grid matching the x and y positions of that particular run. The green of each square shows how long a grasp was maintained

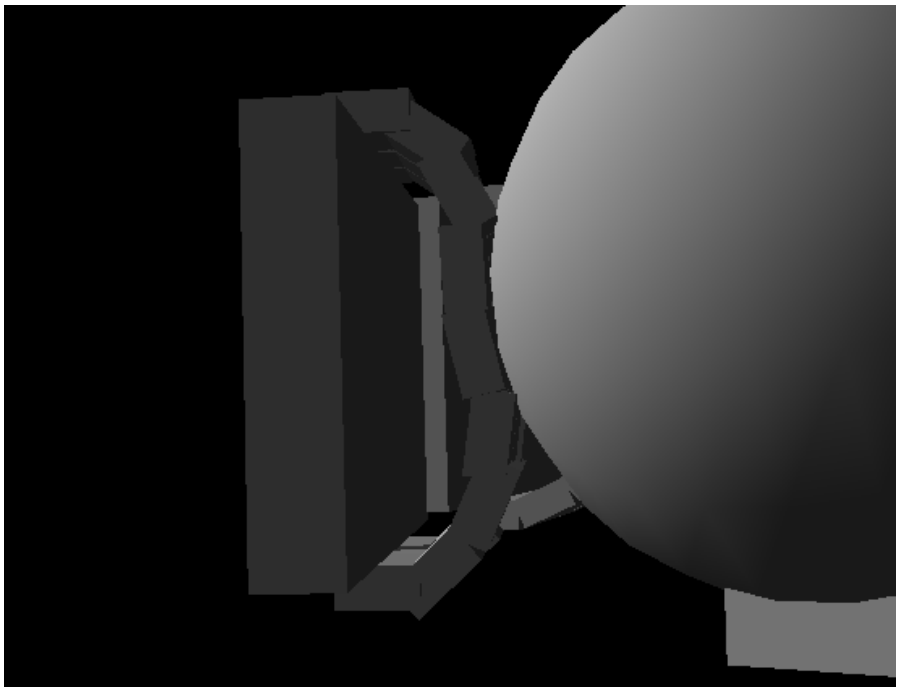

Figure 5.6: A close up of the finger pads bending around the sphere.

as shown by the gradient in the lower left.

The solid fingers are unable to maintain a grasp on the sphere. Even when they are squeezing tightly as the arm moves up, the sphere slips out as it is only maintaining contact at a single point each side. The soft fingers are able to wrap around the curve of the sphere holding it in place, as seen in Figure 5.5 and close up in Figure 5.6. Figure 5.7 shows a solid green bar on the right hand image where the finger pads were able to hold the sphere. Outside of this the finger pads do not get a grip near the centre of the sphere so it is pushed away. The solid fingers on the left show a wider area of touch but everything is at a lower score as no positions are able to maintain a grasp.

The tests were repeated with a cube instead of a sphere. The cube was slightly smaller than the gap between the fingers. There was only one position where the solid fingers could achieve a stable grasp, as shown by the single bright spot in Figure 5.8. The soft fingers achieved grasps on a wider range of locations with the cube than the sphere.

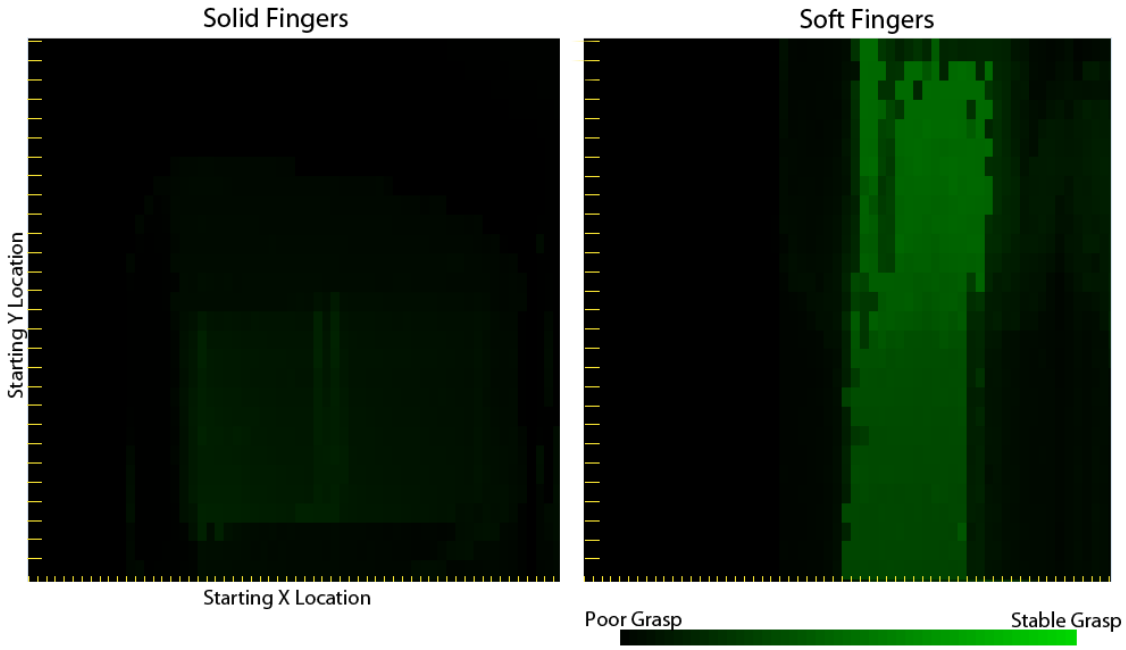

Figure 5.7: The finger pad comparison graph showing 1800 attempts at picking up a sphere in different positions for solid and soft fingers. The Y axis shows the initial Y position of the grasp. The X axis shows the initial X position of the grasp. The green shows the score of the grasp with black being a poor grasp and bright green being a stable grasp as shown by the gradient on the lower right. The soft fingers achieve a higher quality grasp than the solid fingers shown by the bright green.

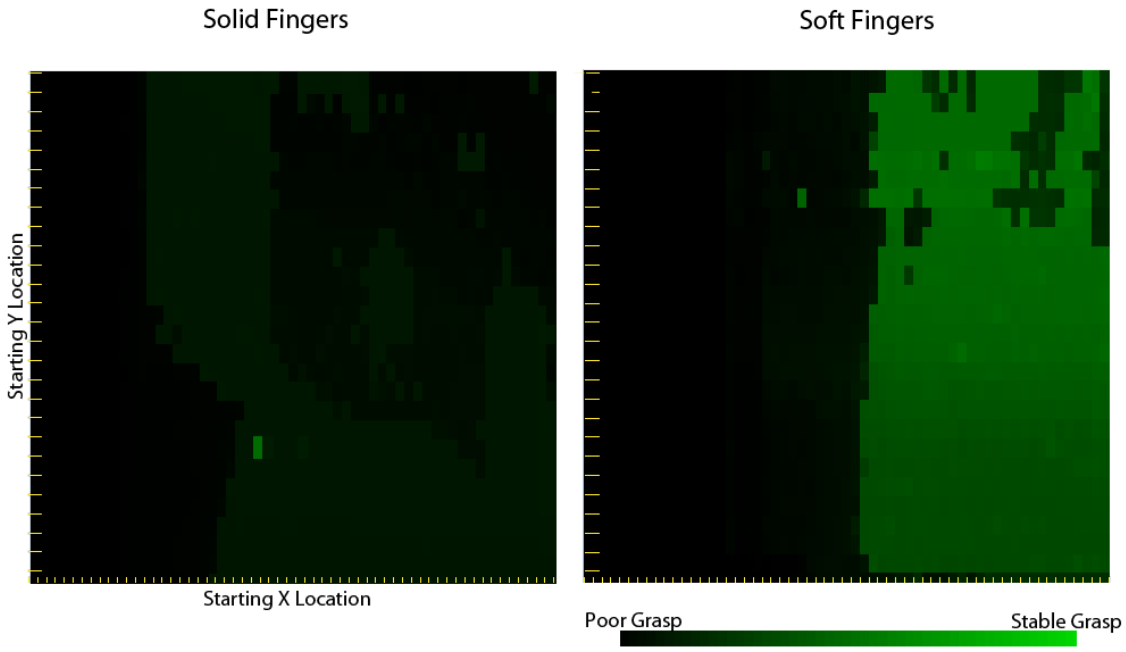

Figure 5.8: The finger pad comparison graph showing 1800 attempts at picking up a cube in different positions for solid and soft fingers. The solid fingers achieve one stable grasp when they are in the optimal position shown by the bright green. The soft fingers achieve a stable grasp in a wide range of positions.

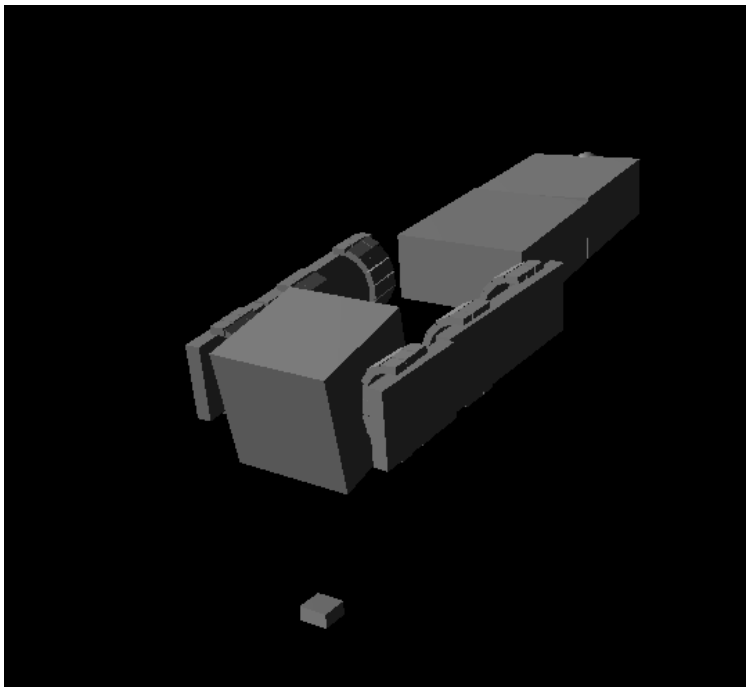

Figure 5.9: The soft fingers holding the the cube with the finger pads flattening against its surface.

5.2 A model of the mechanoreceptors in the hand

One of the benefits of modelling finger pads as lattices of small objects is that it allows for a fine-grained simulation of tactile sensations. This means the system can simulate the mechanoreceptors on the finger pads. The physics engine detects contacts on each individual tile and provides information about the angles between joined tiles. This allows the finger pads to have an accurate simulation of the information delivered by mechanoreceptors in each area of the pad, which is helpful in modelling a reach-to-grasp action.

As described in Section 2.1.2, there are four types of mechanoreceptor in the skin. I will show how my model simulates each of these different sensors. Pacinian Corpuscles sense large deformations in the shape of the finger because they are deep in skin. In my simulation, the deformation can be read directly from the amount of deformation of the finger pads. The further the pad is from its resting position the more deformation in that finger pad. The pads are joined to their neighbours vertically and horizontally. These four connections are used to work out how far the pad is from its resting state.

Meissner's Corpuscles sense light touch and Merkel's Cells sense the texture and

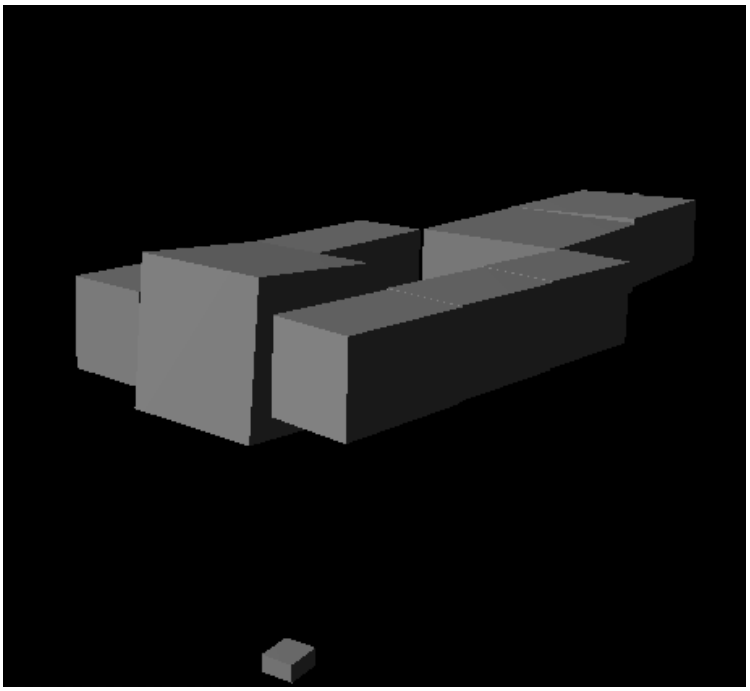

Figure 5.10: The solid fingers holding the cube. Only the far face of the cube is against the finger. The closest finger is only touching the cube with its edge.

shape of an object. In my simulation, when an object has a collision with a finger pad the sensor stores the combined friction of the nodes. The recent collisions are displayed on the UI which shows whether there is constant friction or slippage based on the colours.

Ruffini Endings sense joint changes and skin stretch. The Ruffini Endings in the joints are represented by the system knowing the joint angle and changes. It would be possible to estimate the skin stretch based on the stretch of the joints between the finger pad tiles. If the tiles are pulled away from each other in a direction that the joint does not pivot around then it stretches. However, this information is not currently used by the simulation.

5.2.1 Demonstration of the haptic interface

In order to demonstrate the haptic interface I developed a graphical interface which allows the user to see tactile stimuli on the finger pads as they occur in real time while the hand is in movement. In this section I will use this interface to demonstrate the kind of tactile information which the hand can register.

On a light touch the Meissner's Corpuscles react. This is represented in the graphical interface by the red circles as shown in Figure 5.11. The information about these sensors is shown in a circle. Each circle represents a tile which is currently touching an object. Each collision value from the touch is displayed as a ring. The centre dot of the circle is the value of the most recent collision. The rings around it are progressively older collisions. Each time step the collision moves outwards until it reaches the edge of the circle and is removed. Having a circle of a solid colour means that there is a constant touch and friction. If there are different colours of rings then the friction is changing suggesting that there is slippage, texture or the pad is unable to rest against the object.

The left hand circles are a solid colour showing that there is a stable touch. The right hand circles have rings showing there is a variation in the friction between the object and the finger pad. This suggests that the pad is slipping against the object or cannot find a stable resting position against it.

Deformation of the fingertip is felt by the Pacinian Corpuscles. This is represented in the graphical interface by the green bars between the red circles in Figure 5.12. As the finger is deformed the joints between the skin cubes are bent from their original shape. The intensity of the green bars shows the extent of the deformation. The vertical joints are showing less deformation than the horizontal ones as the pads are deformed

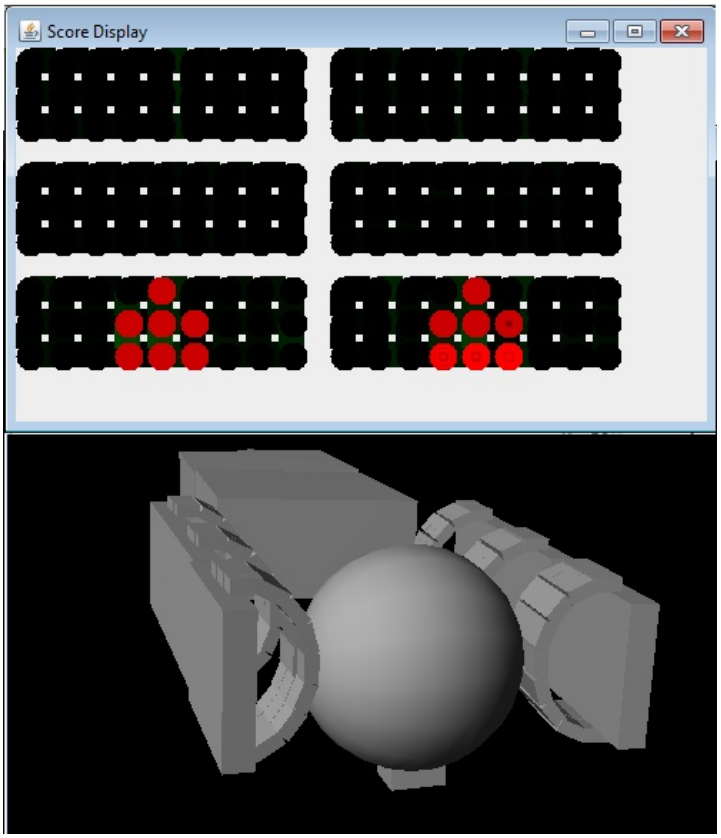

Figure 5.11: Finger lightly touching an object. The left circles are a solid colour showing a constant touch. The right circles have rings showing that there is a variation in touch.

horizontally to a much greater extent. The bright green lines show that certain joints between the tiles are experiencing the greatest change from their resting position.

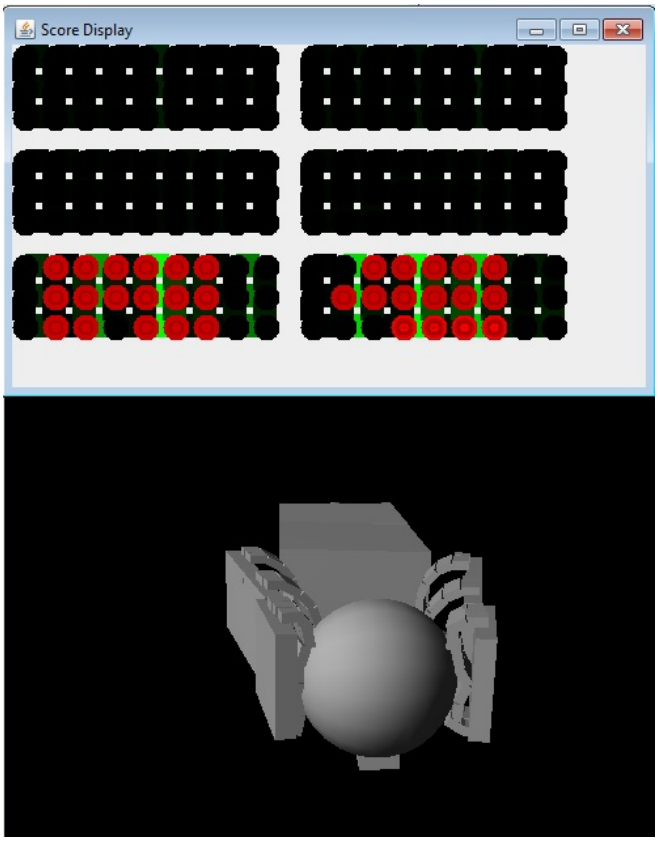

Figure 5.12: Finger pads bending while touching a spherical object with moderate force. The green bars represent the deformation of joints between tiles on the pads. Note that the finger pads deform to match the curvature of the sphere.

Ruffini Endings sense joint changes and skin stretch. One way these are represented in the model is by the system knowing the current angles the joints have. As already mentioned, skin stretch would be quite easy to model, but is not currently represented on the interface.

5.3 Summary

In this chapter I have described two novel aspects of my hand/arm simulation: one is a novel model of soft finger pads as lattices of linked joints, and another is a model of the mechanoreceptors in finger pads. Clearly these two models are linked: it is because

I model the structure of finger pads at a fine level of detail that the model allows an equally detailed model of the tactile receptors which are present in the finger pads.

In the next chapter I turn to a model of the control mechanisms that allow the hand/arm system to execute reach-to-grasp actions.

Chapter 6

Learning reach-to-grasp movements using perturbations

In this chapter I describe an architecture for learning simple reach-to-grasp actions using the hand/arm simulation described in previous chapters. In Section 6.1 I describe my proposal for how reach-to-grasp actions can be learned without precomputing complete trajectories. In Section 6.2 I provide an overview of the methodological considerations I will use in developing the learning architecture. In Section 6.3 I describe the two stages of the learning algorithm. In Section 6.4 I overview how the framework is used to create the actions. The results of my simulations are shown in Section 6.5 . Finally in Section 7.2 I discuss the implications and future steps for the system.

6.1 Using perturbed goal locations to generate reach-to-grasp trajectories

On of my goals is to perform a reach-to-grasp action without a precomputed trajectory. I will review this goal in Section 6.1.1 and show how I perform actions in Section 6.1.2.

6.1.1 Recap: generating reach-to-grasp actions without precomputing trajectories

In a reach-to-grasp action the system is trying to reach a **goal location** which will result in a grasp of the object. Many existing models covered in Section 3.3 precompute the trajectory the arm will make, meaning that the arm's path is plotted before movement starts. One of the goals of this thesis is to perform an action without requir-

ing a precomputed trajectory as I mentioned in Section 3.5. As discussed in Section 3.4, there is good evidence that the brain does not precompute reach-to-grasp hand trajectories, but instead computes them on the fly, as the actions are under way. Another goal of my thesis is to propose a method which allows trajectories resulting in reach-to-grasp actions to be learned, without being precomputed in every detail. The solution for this problem is the topic of my next section.

6.1.2 Using perturbed goal locations to define hand trajectories

I wanted my system to react dynamically to sensory input. One of the limitations of calculating a trajectory in advance is that the path is set before the simulation starts. To achieve the dynamic trajectory I used a simple low-level feedback controller which tries to move the plant into a goal state. This uses proprioceptive information about information about the current state of the arm and interacts with the joints to move towards the goal state. This simple controller achieves the goal of moving towards a goal location without precomputing the path but it is always heading directly towards the goal (at least as directly as the hand's dynamics allow). This simple behaviour means that the system can not perform the complex movements seen with precomputed trajectories.

To achieve more sophisticated movements a second high level controller is added. This interacts with the simple controller by **perturbing** or moving the goal state used by the low-level feedback controller, to create a virtual target location somewhat removed from the actual target location. When the goal state is changed the simple controller will start trying to achieve the new location. Each perturbation can be active for a set amount of time then shifted. These shifts can be joined together to create complex movement patterns. Now the agent can learn a good set of perturbations to apply, which result in the hand achieving a stable grasp. In models where a trajectory is computed in advance, there is a related approach to trajectory learning. This is where the system learns **via points** for the hand to move through on the path to the goal. (Oztop, Bradley, and Arbib, 2004). If the goal is above the current position it will head upwards. If this then perturbs to below the current position then the arm will head downwards without passing through the previous goal location. The goal state can even be something that the system cannot actually perform. An example of this is to set the goal location for the fingers to be inside an object. In attempting to reach the

location the fingers will squeeze the object providing a grip action. The perturbations can represent more information than a via-point by their relative positioning. If the perturbed goal location is far away from the current location then it will generate a large force in the low level controller causing the system to move quickly. The goal locations can therefore be used to achieve a desired trajectory and a desired velocity profile.

I wanted to see if these perturbations could represent the complex movements achieved by models which precompute their trajectories. The low level controller is implemented as a PID controller, as covered in Section 4.2.3. This takes feedback from the sensors about the current state of the arm and combines it with past states and predicted future states to send motor signals to achieve the goal location. This is then updated online and the motor signals change as the arm is moving, as new sensory information is received and as the goal location changes.

The higher level controller perturbs the goal location to achieve different actions. The low level controller is always reaching towards the goal location so when the goal moves the motor signals change so as to reach towards the new target. Activating a sequence of perturbations of the goal location should allow a wide range of different trajectories to be produced. The arm does not actually have to reach any of these perturbed goal locations. For example, to move the arm around an obstacle the goal location just needs to perturb to one side so that the low level controller is reaching beside the obstacle; the controller can then perturb to the actual goal location once the arm is clear. The high level controller can perform perturbations at any time, either ones that have been previously learned or ones created dynamically. This allows it to respond to sensory feedback such as perturbing the goal location inside the goal object when an opposition touch is detected. This will cause the low level controller to make the fingers grip the object. A precomputed trajectory can be represented as a series of learned perturbations. (But note that we are not precomputing the *detail* of the trajectory, just a small number of points: the detailed trajectory is produced when the hand actually generates a movement.)

One important question relates to the coordinate system in which perturbations to the goal motor state are expressed. We want the perturbations learned for a target object in one location to be useful in interacting with a target at other locations: to help achieve this, perturbations are calculated in object-centred motor coordinates, so that learning transfers, at least partially, to target objects at new locations.

6.2 Methodology

Before describing the algorithm for learning perturbations, I will begin by making a few methodological points about the scope of the algorithm, and of the experiments which test it. In Section 6.2.1 I define the level of abstraction I am aiming for with this model and in Section 6.2.2 a simplification which I make in the experiments I report here. In these two sections I will also describe ongoing research with my platform which has extended the system past these limitations. In Section 6.2.3 I describe the **joy of grasping** which is the basis for the reinforcement learning system which I implement, and in Section 6.3 I overview how infants develop their reach-to-grasp actions and show the developmental stages the learning goes through.

6.2.1 Level of abstraction

The system I describe in this thesis is not based on a neurally plausible model. The aim is to test whether perturbed goal locations with online feedback can achieve reach-to-grasp actions at the level of 'computational theory' rather than 'implementation' (Marr, 1982). However, the model of perturbations I introduce here has recently been reimplemented as a neural network within my simulator, and the results are encouraging (Lee-Hand *et al.*, 2012).

6.2.2 Exploring trajectory learning without modelling the role of vision

My simulation does not include vision: for a given training run, the target is always presented at the same location, and is always the same shape. The system's learning involves exploring the reachable space to find a reach action which attains a stable grasp on the object. This is similar to infants' first explorations which are not guided by vision. Clifton *et al.* (1993) found that infants at 16 weeks of age did not rely on visual feedback when performing a reach movement. They tested infants reaching towards an object in different lighting conditions. In the dark conditions the object glowed or made a sound. The infants' early reaches had similar results at grasping in light and dark conditions showing that the infants did not need visual guidance of their hand. The researchers concluded that the infants were guided by the feedback from interacting with the object. However, infants soon start to use vision to guide their reach-to-grasp actions. My simulation does not model development of the visually-

guided component of a reach-to-grasp. It only uses tactile feedback, and proprioceptive information about current joint angles. This means it is necessary for the goal object to remain stationary over a number of simulation runs for the system to learn where the object is and to develop a sequence of perturbations which result in a reach-to-grasp action. However, the learning model I present here has already been extended to include visual information about target location; the results are quite successful and are reported in Lee-Hand *et al.* (2012).

6.2.3 A reinforcement learning approach

I have set up the simulation to learn reach-to-grasp actions through reinforcement learning. This is similar to the way that infants learn. Oztop *et al.* (2004) introduce the concept **the joy of grasping** as the reward for touch sensations in early infant development to motivate infants to explore with their hands. When an infant touches an object s/he get an intrinsic reward from the tactile sensation. This encourages infants to manipulate objects with their hands to maximise these sensations, leading towards reach-to-grasp behaviour. The system similarly gets a reward for each touch on the goal object. The larger the area of the hand that is touching, the higher the reward. If there is a combination of finger and thumb touch then this counts as opposition touch and so scores a high reward because it is close to a grasp. The highest reward comes from the hand performing a stable grasp on the goal object.

6.3 Overview of the learning algorithm

I chose to implement reach-to-grasp learning in two developmental stages following the progression made by infants. In (Berthier, Clifton, McCall, and Robin, 1999) the authors describe infants as having two problems learning to reach: transporting the hand to the target object and conforming the hand to the object to perform the grasp. There are many redundant degrees of freedom in the arm and hand creating a large search space.

Berthier *et al.* (1999) suggest that if infants rely on their upper arm and torso then learning to reach would be much simpler due to the reduction in functional degrees of freedom. To test this they followed nine infants at the onset of reaching. The experiment involved the infants reaching towards a toy held in front of them and to the side to encourage the infants to use their right arm. The infants' movements were tracked with a video and a motion analysis system with markers on their arm. The

experiment was performed in three levels of illumination: fully lit, darkness with a glowing toy and darkness with only a rattle sound for the toy. They defined a reach by the infant as a movement of the hand towards the goal object when the infant had attention towards the goal object. The infants were held on their caregivers' lap during the reach so they could use any combination of torso, shoulder and arm movements to move their hand to the goal object.

The results of Berthier *et al.*'s experiment showed that infants mainly used shoulder and torso rotation to perform their reaches and that most infants did not use their elbow. Half of the infants started each of their reaches from the same location in space, with some infants bringing their hand backwards to this location to begin the action. There were multiple accelerations during the reaches but individual infants showed speed peaks around particular values. As the infants became more experienced their hand path showed smooth acceleration and deceleration of the hand.

Berthier *et al.* (1999) show that the results of their experiment supported their suggestion that infants primarily use their upper arm and torso during early reaching. By stiffening their lower arm the infants can have smoother and more predictable hand movements because the intersegmental forces are dampened. By limiting the degrees of freedom the effects of particular motor commands are reliable because no extra joints are involved. It also allows infants to separate learning into two phases. First they learn how to use the upper arm and torso to perform a reach, bringing the hand to the target. The hand in the early phases provides tactile conformation with the goal object. Then the second phase can build on top of this learning to pre-orientate the hand and utilise the full degrees of freedom in the arm. These phases can be seen as a natural constraint as the infant matures, as the brain and spinal systems develop, the infant has increased control over their arm.

It also takes an infant time to transform its reach-to-grasp actions which achieve the goal into an adult style action which moves the hand in a straight line towards the goal (Konczak and Dichgans, 1997; Bhat and Galloway, 2007; Berthier *et al.*, 1999). The infant's movements are not very smooth and have a wide variation in the trajectory path and speed. I am aiming for this level of infant grasp rather than the optimised adult style straight line grasp.

The system I developed goes through two main stages in learning how to perform a reach-to-grasp action. During the first stage, it learns simply to reach to the goal object and achieve a touch of some kind on it. Because the system only has touch sensors it has to explore the space to find reach actions which come into contact with the object.

These touches are hardwired to give positive feedback (the 'joy of grasping' idea) which leads to more reach actions in this area. The actions are simple reaches towards a goal location just using the feedback controller without perturbations. During the second stage, the system learns to achieve a stable grasp on the object, rather than just a touch. This involves orienting the grasp and closing the fingers around the object. The tactile rewards which guide learning are now more specific types of touch: an **opposition touch** where there is a touch on fingers and the thumb generates a reward, and a **stable grasp** generates a larger reward.

6.3.1 Stage 1: Reaching

The first stage involves learning to achieve a simple reach-to-touch, without any attempt to grasp. Because there is no visuomotor feedback this is done through exploratory reaches. The target object is put in central location and the system performs a reach. If there is a touch with the object then the reach receives a reward. After each reach the target object is reset to its original location in case it was knocked over in a previous run. Each reach is recorded in the system's memory, where it is associated with the touch-based reward it achieved. This reward is called the **score** of the reach. Initially, reaches are made to randomly chosen goal states. But as training progresses, and the system's memory begins to build up a record of high-scoring actions, there is a bias in the random selection towards high scoring motor states. This changeover is covered in Section 6.4. The two degrees of freedom controlled during this stage are the shoulder and the elbow. The hand and wrist are not moved during these reaches so that they can be compared evenly and to reduce the search space. The previous runs are reused and built upon. Once enough reaches have been performed and the hand is reliably reaching and touching the target object, the system switches into the second stage of learning: grasping.

6.3.2 Stage 2: Grasping

The second stage is when the system knows the location of the target object and is learning to grasp it. Starting from a high scoring reach run the wrist and hand are manipulated to try to grasp the object. This involves more degrees of freedom than the reach stage. The wrist has two degrees of freedom for aligning the hand to the object. The hand also has two degrees of freedom, opening and closing the fingers and curling the fingers. The thumb performs the same actions as the fingers to limit the complexity.

The fingers can perform a power grasp using all the fingers or a precision pinch using a single finger and the thumb. Each reach is again recorded in the system's memory. This time, a reach is represented in memory as a data structure comprising not only a goal motor state and a score, but also a sequence of perturbations of the goal state. As in Stage 1, goal states are selected at random, with a bias towards high-scoring states. Crucially, in Stage 2, perturbations of goal states are also selected at random. Perturbations can be to the goal motor state of the shoulder or elbow. Detail about how these perturbations are selected is given in Section 6.4. As training progresses, and a record of high-scoring perturbations is built up in memory, the random selection of perturbations is again biased towards high-scoring perturbations. Another change in Stage 2 is the type of tactile stimulus which generates a high score. While in Stage 1 any touch generates a reward, in Stage 2, the hand must achieve an 'opposition touch', where the fingers and thumb simultaneously make contact with the target object, in order to get a high score. The highest reward score is when a 'stable grasp' is achieved, which is a special form of opposition touch where there is no slippage detected at the opposing points making contact.

6.4 Implementation of the learning algorithm

I will start in Section 6.4.1 with an overview of the different components of the system and how they connect together. Next I will cover these in more detail with Section 6.4.2 showing how the data is stored and Section 6.4.3 describing how the plant is controlled. In Section 6.4.4 I outline my implementation of the reward algorithm mentioned in Section 6.2.3. In Section 6.4.5 I show how an existing reach is ranked and selected to be reused. Finally, in Section 6.4.6 I describe how the reinforcement learning takes place over multiple runs.

6.4.1 Overview of the system

The system can be run in two types of task: one is demonstrations which show a specific feature of the framework and often have direct user interaction, and the other is autonomous learning. In each case, there is a notion of a single **movement trial**. In a movement trial, the hand/arm is created in a fixed position along with the target object. In some of the demonstrations the user can manually choose goal motor states for the hand/arm in real time through the graphical user interface(GUI), and the hand/arm will attempt to reach these states. In other demonstrations the user can

directly interact at a low level through the keyboard, or the system will run a predefined motion. In autonomous learning, there is a sequence of separate movement trials. Each trial is recorded in a **trajectory** and stored in the system's memory. The selection of motor goals during a movement trial is implemented by **controllers**. The different types of controllers will be discussed in Section 6.4.3. At each iteration of a movement trial there are several parts of the simulation which can change. New motor goals can be given to the physics engine. The physics engine can in turn generate updates to the current motor state and new information can be displayed on the GUI. These real-time interactions are handled by **directors**, which were introduced in Section 4.5. The following sections will provide more detail on how the movement trails are stored in Section 6.4.2 and controlled in Section 6.4.3, then how they are ranked in Section 6.4.4.

6.4.2 Storing actions in memory

A 'trajectory' records an action. It stores the current state of the component and the score at each time step. It also records the various goal states that the component was aiming for. By replaying these goal states at the same time steps the plant will perform the same action even though it is dynamically working towards the goal. The current and goal states are stored in the trajectory as a 'data point'. This records the joint proportions and the time step. A new data point is stored each time the state is updated resulting in a large number of current state values and a goal state data point for each time it is perturbed. The trajectories are combined into 'run groups' which store the trajectory for each component involved in the iteration. If the iteration only involved one component such as the arm, then the run group will only have one trajectory. The arm may have had no perturbations so its trajectory would only have one goal data point, along with the current data points and score for each time step. More complex combinations of components will have multiple trajectories each storing the perturbations of goal states and the states performed during the iteration.

The 'memory manager' looks after the storage of the run groups in between runs of the simulation. A new run group is added after each iteration. The memory manager has a limited size so the lowest scoring run group is removed once the space has been filled resulting in a beam search. For the next iteration a run group is selected by a random choice weighted on the score. This is then used for the basis of the next run in the simulation.

6.4.3 Controllers

The 'controllers' create the goal states which direct how the sytem will move. There are various types which are used at various points during the simulation. All of them send new goal states to the State Director. Some receive information from the current state of the system or the current score to make their decision about the new goal state. Some are based on previous movements while others are completely online.

The 'exploration controllers' are for generating new movements and are completely generated online. They do not use any previously generated movements. They are used for exploring the space to get a touch, and trying hand positions to get a grasp. The different types of exploration controllers are 'point', 'step' and 'flow'. The point controller generates a single goal point which is used for an interaction of the simulation. This is used when first exploring to find the simplest movement that will achieve a touch with the goal object. The step controller creates a new goal position after a certain amount of time has passed. This allows for more complex movements with a number of steps such as opening and closing the fingers during a grasp. The flow controller is for creating smooth actions. The goal locations are updated rapidly with small shifts. This produces smooth movements such as gently closing the fingers but creates a complex trajectory.

The 'perturbation controllers' are for improving existing movements. They take an existing movement and perturb or shift the goal points. This altered movement is then run which may produce an improved score. These controllers are used to fine tune the movements once the basic goal of a touch or opposition touch has been achieved. The 'increaser' controller adds in a new goal point between two existing ones in the movement and then perturbs to introduce a new direction into the movement. Conversely the 'reducer' controller removes a goal point to reduce the complexity of a movement and to remove redundant steps. If the new movement is higher scoring than the original then it is more likely to be reused. This new movement will then be perturbed further leading to improved grasps. The 'replay controllers' are similar to the perturbation controllers except they just replay the goal states without any alterations. This is used for reviewing iterations of the simulation. Although it is replaying a previously recorded movement it is doing so in the live simulation with the low-level controllers trying to reach the goal points. The trajectories the perturbations are based on are selected using the method covered in Section 6.4.5.

The 'intuitive controllers' represent low level behaviours that can be used as a basis for other controllers. Currently the only implementation of this controller is

used to perform a grip action. The controller reads in touch information and once the opposition touch on the fingers and thumb has reached a threshold it takes over from the high level controller. It sets the goal state for the fingers to squeeze, representing the palmar grasp reflex of an infant. Before this threshold is reached the higher level controller continues as normal. Any controller can use this controller as its base, allowing it to take over and perform the grip action. The difference using this controller makes is shown in Section 6.5.3.

The 'user controller' takes input directly from the user through keypresses or mouse interaction and transforms it into goal states for the components. The aim of this controller is to give the user an option for direct control over the simulation and to show that corner cases such as direct interaction can be handled by interacting with the existing framework. The two main types are the 'interaction' controller and the 'keyboard controller'. The interaction controller allows the user to take the place of the exploration controller and choose the goal states with mouse clicks on the joint display. This creates a new goal state for the system to achieve. The simulation behaves in the same way as if the goals were generated by any other controller. This allows the user to directly see what effect a goal state in the joint display will have on the simulation. The keyboard controller interacts at a lower level. Instead of integrating into the simulation framework it directly accesses the physics engine. This can be used for fine control over joints of moving the plant. The aim of this controller is to allow the user to directly interact with the physics engine rather than through abstract goal states.

6.4.4 Tactile reward

The tactile reward system is based on the 'joy of grasping' described in Oztop *et al.* (2004) which is where an infant gets a reward from performing actions leading to a grasp. The simplest (and smallest) reward is for achieving a touch with the fingers. This lets the system know that it has found an object and that the arm is in approximately the right place. The reward is proportional to the amount of contact the fingers are making. An intermediate level of reward is for an opposition touch, where the fingers and thumb are both touching the object at the same time. This means that the system is close to a grasp and the orientation is correct. Again, an increased amount of contact gives a higher reward but opposition touches give much higher rewards than a simple contact. The final reward is when a stable grasp has been achieved meaning the system has reached its goal. A stable grasp gives the highest reward.

The design of tactile system is based off the mechanoreceptors covered in Section

2.1.2. The aim of the design is to give fine-grained detail about the position and strength of touches. The tactile system is tied directly to the underlying physics engine. When two nodes in the physics engine intersect this is called a collision. The tactile system receives a notification of each collision detected. From this it looks at the nodes involved to see if they are part of the tactile system. If both nodes are involved then this is counted as a 'self touch' and does not give a reward. If only one node is involved then it receives more information about the touch. The 'impulse' is the strength of the collision and the 'friction' is the movement in lateral directions to the collision. These values are combined and assigned to the component involved. This will be a section for a solid finger or a tile for a soft finger. At each update cycle these values are stored as the hand score. With the soft finger, the bends of the finger pads are also recorded at this step. The hand score represents all the interactions for the last cycle. The base score is the total amount of touch achieved by the hand. The score values for each individual component are summed. This is compared to the theoretical maximum touch of every component registering a score. The next step is to calculate the opposition score. This is the minimum out of the finger and thumb scores representing the touch on opposite sides of the object. The reward for opposition score is scaled so that a perfect opposition touch is worth twice as much as all the individual fingers achieving a perfect touch. The third step is to calculate the slide. This is done by only looking at the friction component of the collisions. If the friction keeps the same value over multiple cycles then it is assumed that the grasp is holding. Once this reaches a threshold it is declared a stable grasp and given the highest reward value. When scores are compared first it looks at the stable value. If either have this then they are rated the highest. Otherwise the base score is combined with the scaled opposition score giving a total score to rank by.

6.4.5 Reach rank and selection

When an action is performed it is broken into three separate components, the arm and elbow movement, the wrist bend and rotation and the finger movements. Each of these parts is either created on the fly using an 'exploration controller' or an alteration of an existing movement with a 'perturbation controller'. When a movement is performed the goal states it used are stored along with the reward score calculated, as shown in Section 6.4.4, for each component. When a perturbation controller is used it requires a previous trajectory. The trajectory is chosen using a weighted random selection. A list is created which stores the possible trajectories. Trajectories can appear multiple times

so that the proportion of the list they appear in is equal to their relative score. For example with one high scoring trajectory and one low scoring trajectory the majority of the selection list will be made up of the high scoring ones. A random trajectory from this list is then selected and used for the iteration of simulation. This means that high scoring trajectories are more likely to be selected. There is also a maximum number of trajectories which are kept called the 'beam'. When a new trajectory is added to the beam causing an overflow, the lowest scoring trajectory is removed. This causes the average score to increase over time.

6.4.6 Reinforcement learning regime

The two tasks that the system tries to learn are how to reach to touch and how to perform a grasp. In each stage it first tries to achieve the task and then improve upon it. The learning regime uses the exploration and perturbation controllers from Section 6.4.3 combined with the reward from Section 6.4.4. To perform the touch it first performs exploration reaches. Each of these reaches is saved as an arm component trajectory storing the goals for the shoulder and elbow. Once a touch has been achieved it introduces perturbation reaches. These take an existing reach trajectory and alter it as described in Section 6.4.3. The proportion of exploring to perturbing is relative to the number of scoring reaches in the beam. As the number of reaches that achieve a touch increases the regime spends more time improving instead of exploring. Once the lowest scoring trajectory in the beam has reached a threshold score the system switches to the next stage of learning (learning grasps).

The reaches only give goal points for the shoulder and elbow, with the hand and wrist being fixed. Once the regime switches to grasp mode it introduces exploratory wrist and finger movements. The arm movements are mainly based on trajectories achieved during the reach phase with occasional explorations in case the touches have given a suboptimal arm basis for grasping. The hand exploration phase continues until an opposition grasp is achieved. Following the same pattern as for the reach, perturbation controllers are introduced for the wrist and hand and the exploration is decreased in proportion to the number of opposition touches. More complex controllers are introduced attempting to add or remove the goal steps and choose components from different trajectories as well as perturbing the high scoring runs. These continue with the lowest scoring runs dropping out as new ones are performed until a stable grasp is achieved.

6.5 Results

In this section I give an overview of the results of the simulations in Section 6.5.3 and show the detail for simulating grasping a cylinder. The reach-to-grasp simulations try to achieve a stable grasp on a target object. In this section I describe the results of learning experiments with different types of target object, focusing on the most successful, a tall cylinder. Then in Section 6.5.4 I will cover the other shapes of a sphere, a cube and a wide short cylinder. The objects are placed in different locations for each simulation. The simulation starts with no knowledge of where the object is. It starts by attempting to touch the object, and then learns to get an opposition touch on the thumb and fingers and finally achieve a stable grasp.

6.5.1 Heat map of touches

A tool which I have created for visualising the results in a 'heat map'. This represents the highest scoring position of a trajectory in joint space on a map. The X and Y position represents the two angles for the components joints, such as the shoulder and elbow rotation for the arm. The size of the point represents the score, the larger the point the higher the reward. Figure 6.1 shows three heat maps for the arm and hand over a series of iterations. At first there is a wide area of small points the arm learns to reach-to-touch. After more learning has occurred, large points showing the first grasps appear. Finally only one point remains in each heat map as the highest scoring position has filled the beam.

6.5.2 Initial simulations with no grip reflex

The first series of simulations I ran had a problem achieving the last step. They would peak at getting an opposition touch, only making small improvements over time thereafter. They would spend thousands of iterations slowly improving the beam but never progressing to a stable grasp. The introduction of the low level grip controller described in Section 6.4.3 had a dramatic effect. The simulation was able to quickly progress to a stable grasp once it had achieved an opposition touch. In the rest of this section I will focus on the results of these latter runs. Demonstration movies for these reach-to-grasp actions can be found at http://graspproject.wikispaces.com/Movies.

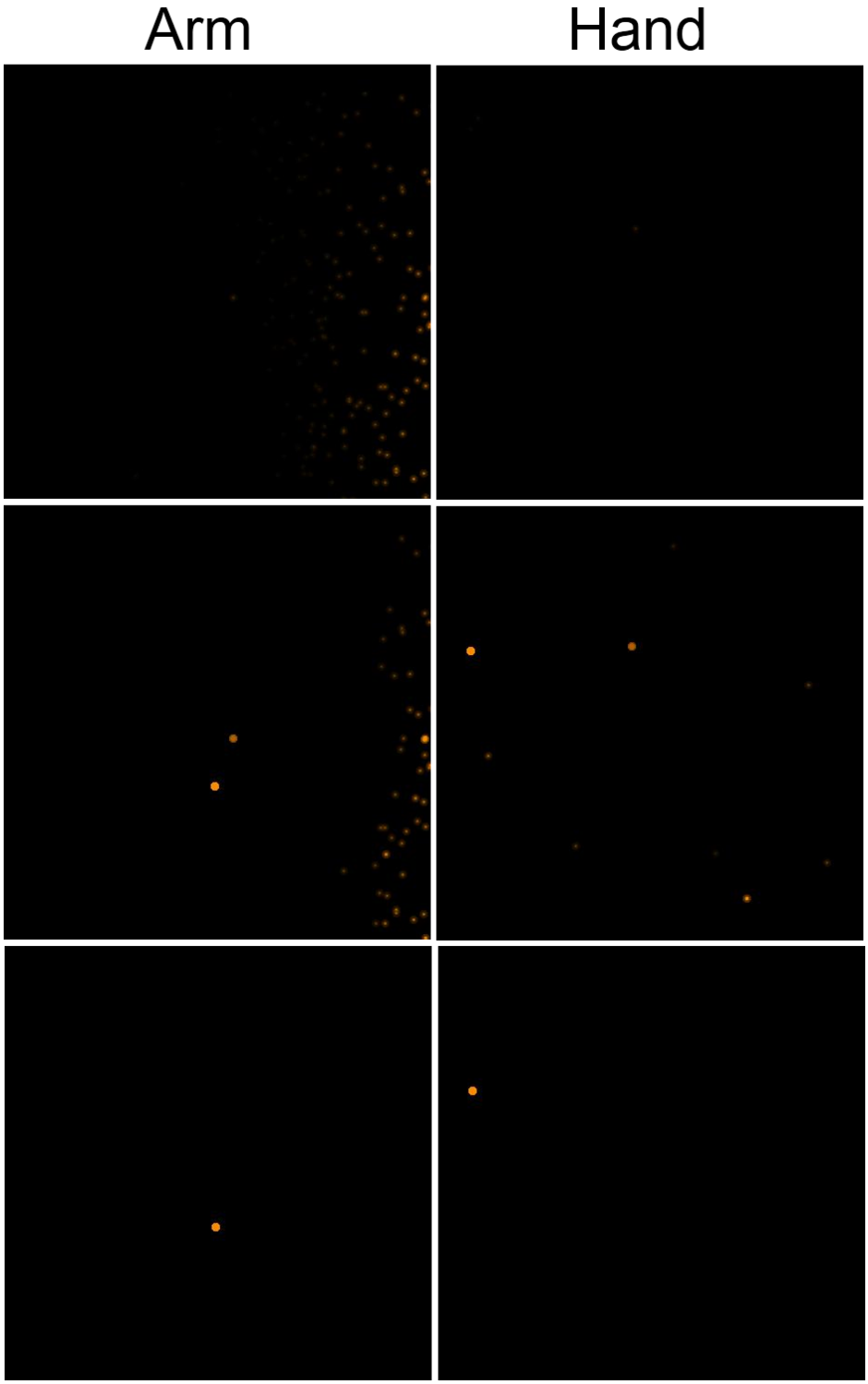

Figure 6.1: The heat map shows the highest scoring position achieved for a component. The position of the dot represents the position in joint space for the two joints of the component. The larger the size of the circle the higher the score. The left column shows the arm with the X axis representing the shoulder angle and the Y axis representing the elbow angle. The right column shows the hand with the X axis representing the finger bend and the Y axis representing the finger curl. The top row is after touches have been achieved, the second row after some grasps and the third row finished with a stable grasp.

6.5.3 Reach-to-grasp

The easiest type of object for the system to grasp is a cylinder. A successful grasp of this object is shown in Figure 6.2. This is because the object has a wide base which is resistant to being knocked over from a touch and the long shape allows grasps at different heights. The graph in Figure 6.3 shows the maximum, average and minimum scores of the beam, averaged over four attempts at grasping a cylinder in different positions. These positions are shown in Appendix A.3. Each jump shows an improvement in the grasp.

The graph shows the two phases of learning, with the switch over at around 400 iterations. The simulation starts off trying to touch the goal object. Many of the initial attempts to achieve a touch miss so the beam fills up with a minimum score of 0. The average of the beam increases as the beam fills with successful touches around 200-400 iterations. The maximum score plateaus from 100 to 400 iterations as it has achieved the best touch for the first phase of learning. By 350-400 iterations the beam is filled completely with successful touches and the simulation switches to the second phase of learning of trying to grasp the object. The large jump in scores is from the greater reward for grasping an object compared with achieving a touch in the first phase. Each jump in the maximum score line is an improvement on a previous grasp or a new type of grasp from a touch arm position. The minimum score slowly increases as more attempts are perturbations of a high scoring grasp and has a jump in score when a final low scoring attempt is removed. The average line shows a steady increase as the beam as a whole improves. Eventually the entire beam fills with small variations on the best scoring grasp giving a flat line.

I will now go into detail of a single simulation. The main steps of the maximum score can be seen at http://graspproject.wikispaces.com/Movies 'Single simulation on a Tube'. The beam scores for this simulation are shown in Figure 6.4. The simulation starts off achieving touches with the shoulder to the right and a variety of elbow positions shown in the first arm heat map. The hand has not achieved any grasps so the first hand heat map is blank. The high scoring touch of 304.222 at iteration 75 knocks the cylinder over but achieves a long touch giving a high score. After 372 iterations the beam is completely full of successful touches so the simulation switches to grasp mode. This can be seen as the sharp rise in the minimum score in Figure 6.4. At iteration 576 a stable grasp is achieved giving a new high score of 369.867. The second row of the heat map in Figure 6.1shows some higher scoring values for original arm touches where the arm position has been used to achieve a grasp. The lower

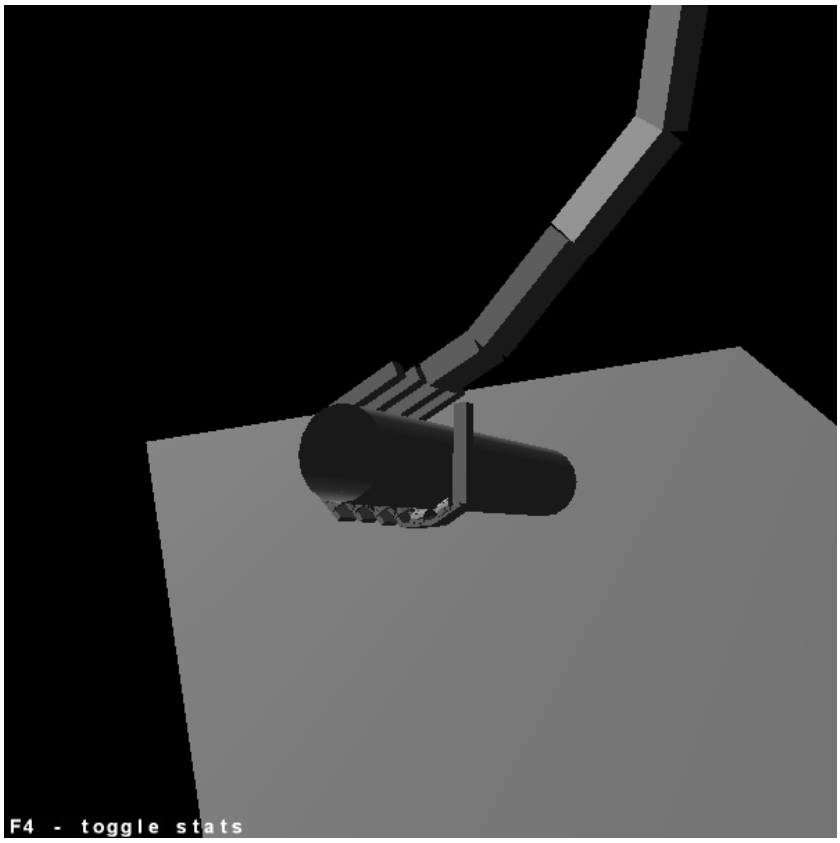

Figure 6.2: Successful grasp of a cylinder.

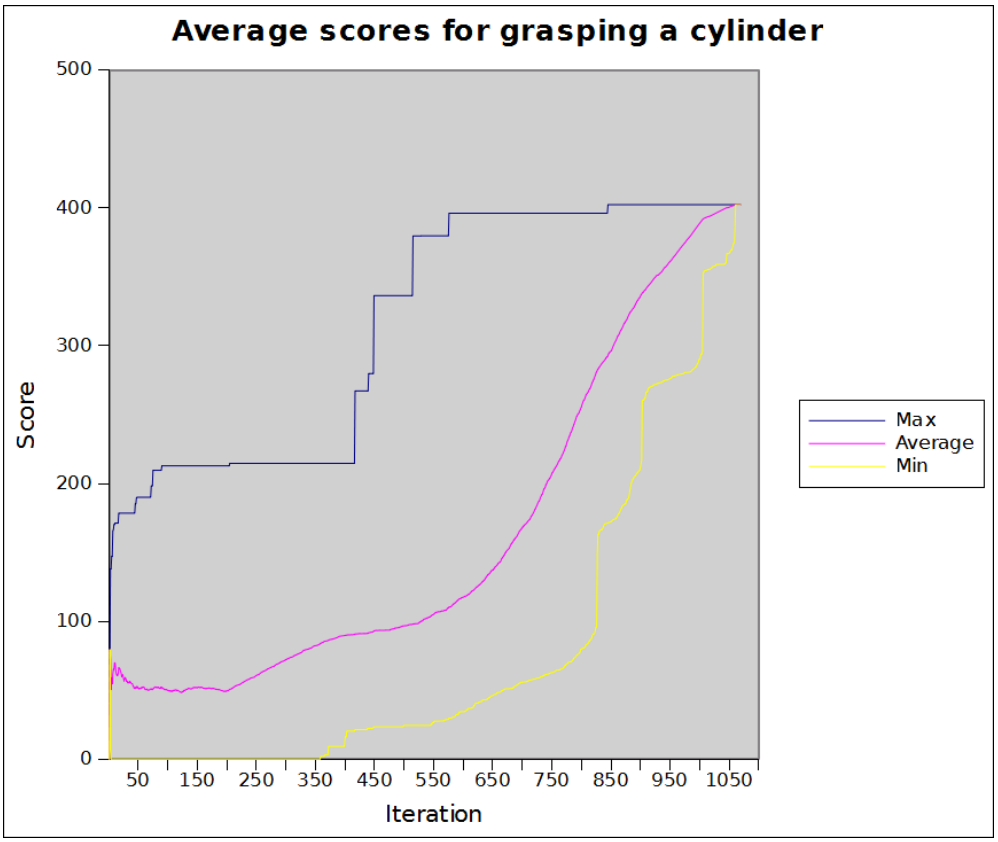

Figure 6.3: The maximum, average and minimum scores for grasping a cylinder averaged over three object positions.

scoring positions have been removed from the beam. It also shows two new positions where the arm has been perturbed to optimise for a grasp rather than a touch. The Hand column shows a variety of finger positions which have resulted in a grasp. As the simulation continues the current stable grasp remains the highest scoring as the beam fills up with perturbations on this. At iteration 1061 the minimum and maximum scores of the beam are equal and the heat map shows just the highest scoring arm and hand position.

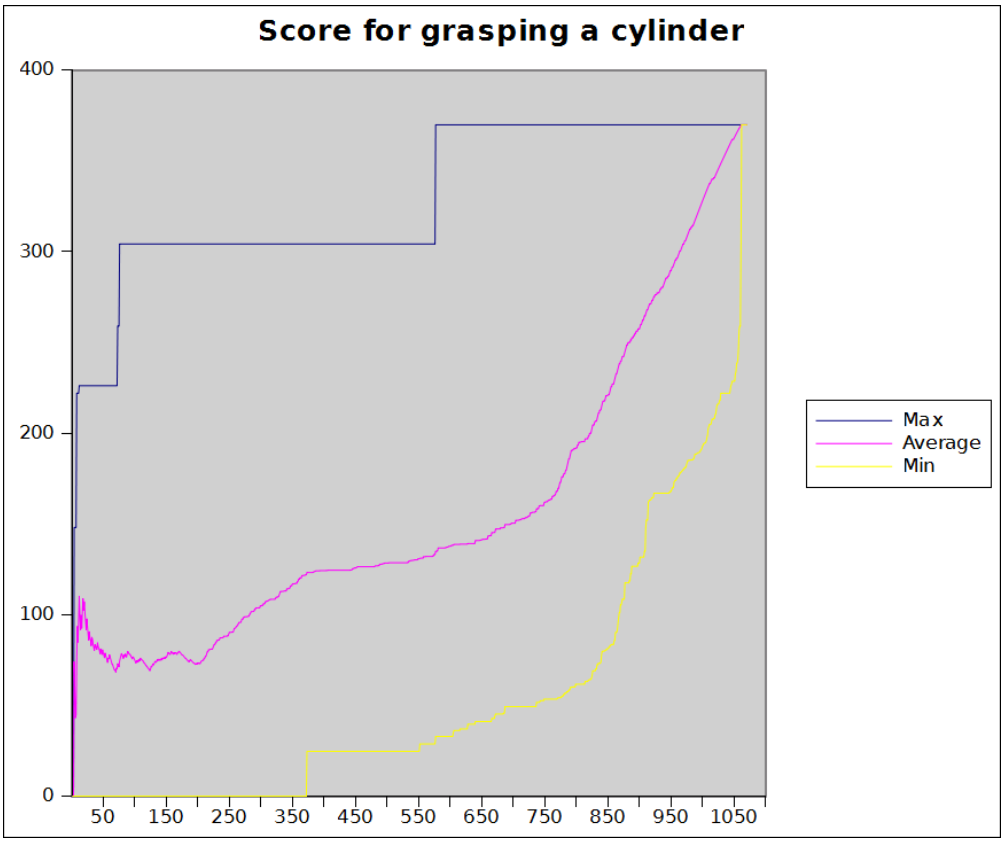

Figure 6.4: The maximum, average and minimum scores for grasping a cylinder for a single simulation.

6.5.4 Trails of other shapes

The system can handle many different shapes. These were not explored as fully as the cylinder shape. The most successful other shape was the Sphere, as shown in Figure 6.5. The graph in Figure 6.6 shows various high scoring touches are achieved until iteration 82 where a grasp is performed. This is improved upon until iteration 449 achieves the highest scoring stable grasp. The sphere object is much more sensitive to

the starting position, with it being much harder to grasp. Because it is sitting in space on a small pedestal it is easy for it to get knocked off giving low scoring touches which do not result in a good arm position for a grasp.

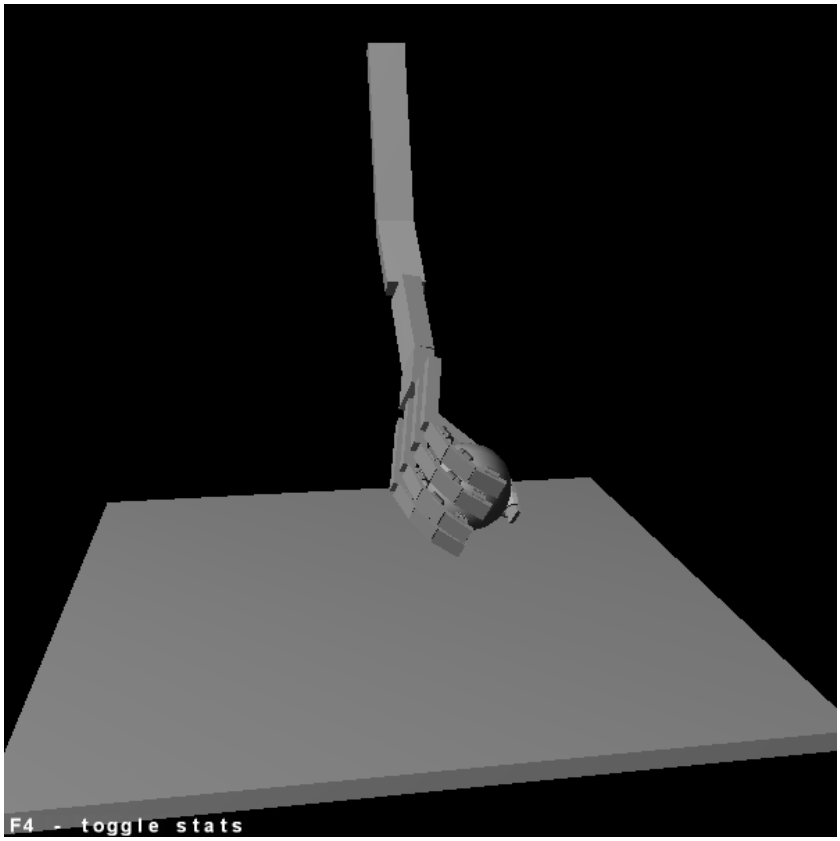

Figure 6.5: A successful grasp on a sphere.

The cube and wide cylinder shapes were less successful. The cube was too small to be properly grasped between the fingers and thumb so on occasions where a grasp did occur it was in an unusual position, as shown in Figure 6.7. The aim of the wide cylinder was to achieve a pinch grasp on the object. Like the sphere this was also very easy to knock off, as shown in Figure 6.8, and did not result in any successful grasps. The system allows new types of goal object to be defined and is not restricted to single shapes, as shown in Figure 7.1. The goal object is read in at run time so the same learning algorithms can be applied to different types of goal object.

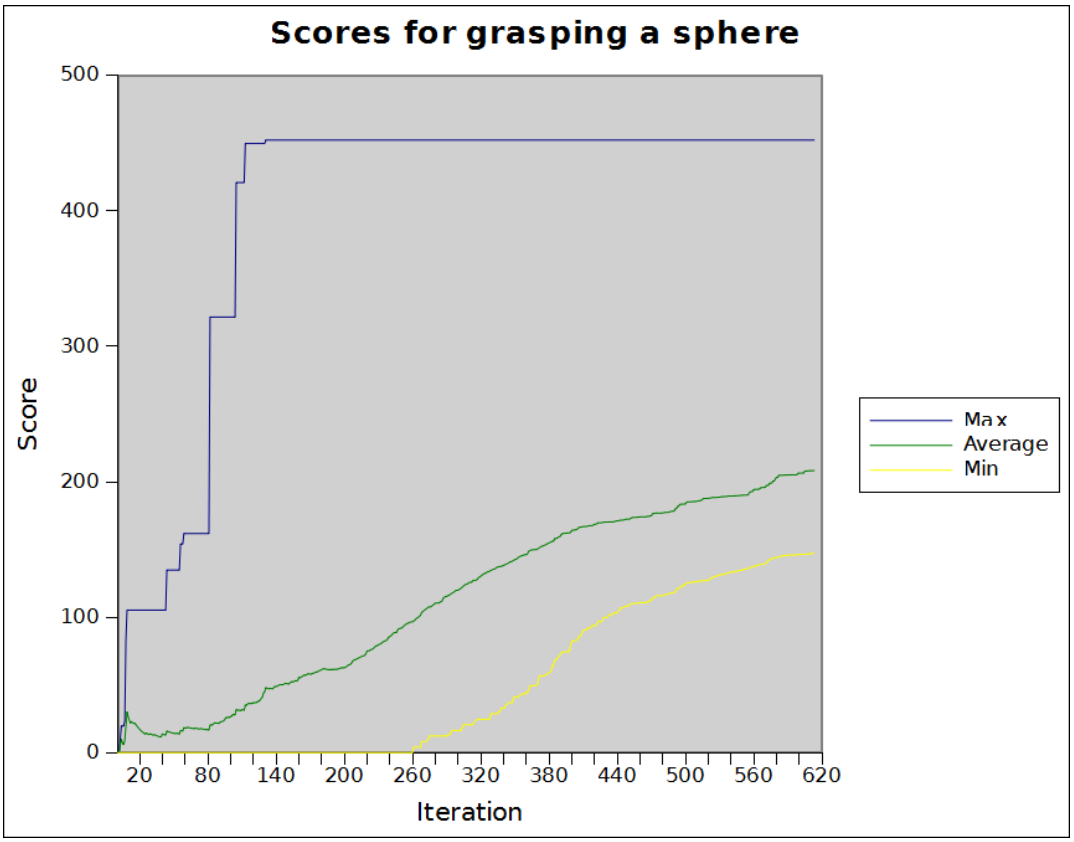

Figure 6.6: The maximum, average and minimum scores for grasping a sphere for a single simulation.

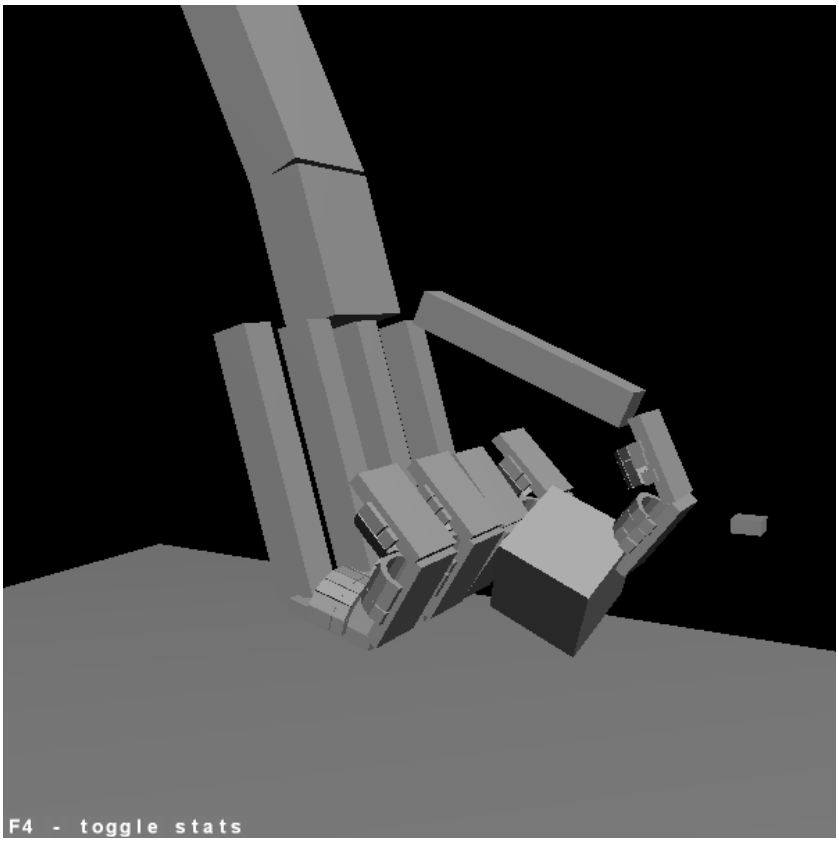

Figure 6.7: A grasp of a cube. Because the goal object is not touching the sensors it will be a low scoring grasp.

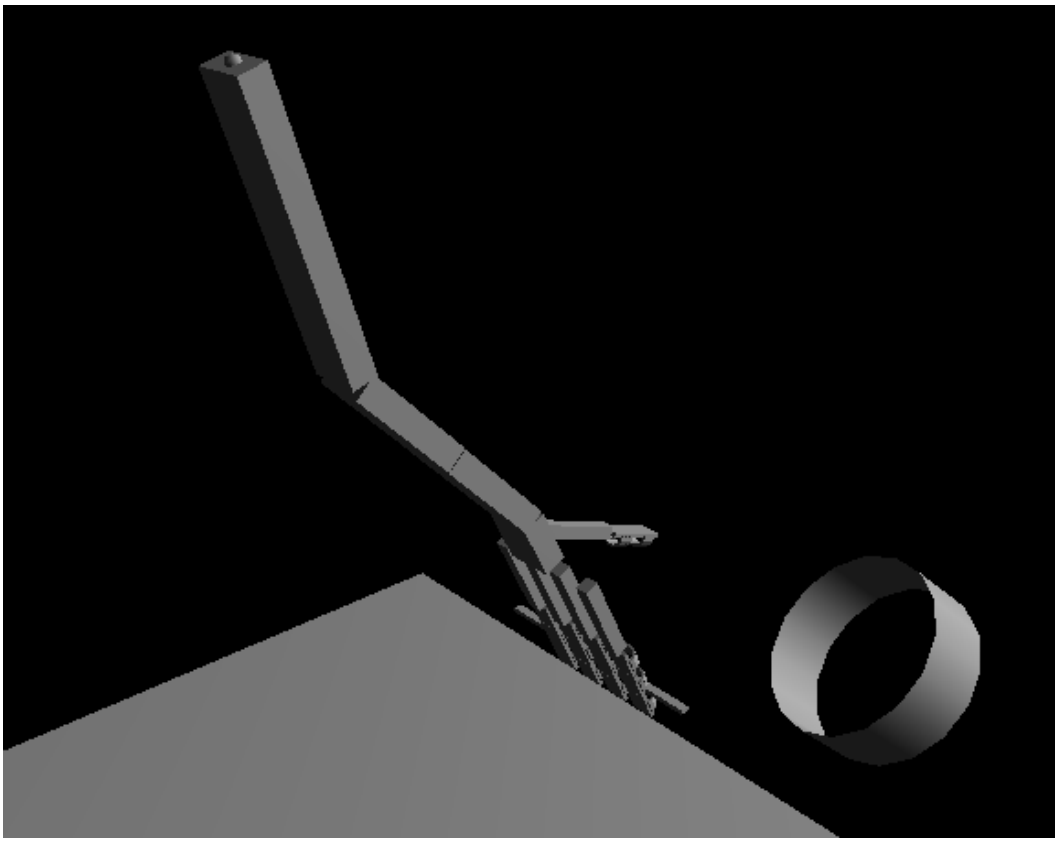

Figure 6.8: A grasp attempt on the wide cylinder.

Chapter 7

Summary and further work

7.1 Summary

In this thesis I have shown a new platform for simulating reach to grasp actions. It leverages an existing physics engine to provide general-purpose definitions of force and friction to ensure the results have a realistic basis. The implementation on top of this physics engine is modular and has some new features. The soft fingers provide real time deformations and mechanoreceptor feedback while still using the standard physics engine. The learning algorithm does not precompute the trajectories of the hand and arm. Instead it uses a combination of a low-level feedback controller which moves the hand towards a goal location and a high-level controller which perturbs the goal state to create complex actions. The combination of these controllers and the touch feedback from the soft fingers has resulted in the system learning to perform a stable reach-to-grasp action on different objects.

7.2 Further work

The system has been designed to be modular and allow different types of experiments to be added. Using the general purpose physics engine means that the platform can support new types of interactions. New components for the plant can be swapped in and the director model can be extended with different types of systems.

7.2.1 Vision

The simulation has already been extended to add a vision component: this extension is discussed in Lee-Hand *et al.* (2012). The extended system simulates the visual field of the grasping agent, and learns a function mapping a representation of the 2D retinal location and distance of the target object to a 3D goal motor state. It starts with 'motor babbling' exploratory movements until it touches an object then learns to orient its palm in relation to the target object. Once it has achieved a certain level of competence it learns to grasp objects in the surrounding area. At the end of each iteration the joint angles and vision inputs are stored with the score of how good the grasp was. The learning is performed using a neural network described in Section 7.2.2.

7.2.2 Neural network simulation of the perturbation learner

The system I described in this thesis did not use a neurally plausible learning model. One of the goals in Lee-Hand *et al.* (2012) was to create a neurally plausible model of how a reach-to-grasp action could be encoded and learned. The platform was extended to use a neural network to learn the different parts of the action. The training data for the neural network is generated by exploratory motor babbling movements. The input for the neural network is the retinal information described in Section 7.2.1. The input units encode the x and y coordinate of the target object from the retina display and a z coordinate representing distance from the eye read directly from the physics engine. The outputs are two angles for the ball shoulder joint and one for the elbow. Between each iteration of the simulation the neural network is trained on the previous data. By the end of the training the goal arm state for a stable grasp can be computed from the visual input, for a range of target locations.

7.2.3 Improving finger pads

Different types of graspers are also simple to add. One extension to the current grasper could be to improve the finger pads. Currently they are a band on each finger section which are the only areas which give touch information. In the human hand the skin provides touch information at any point. Two areas which could use touch information are the fingertips and the palm. Currently if the fingertips touch an object this is not registered as a touch as it does not touch the pads. Also the palm is involved in a power grasp and touch information here is especially important for creating a more accurate palmar grasp reflex.

7.2.4 Different actions

Another avenue for further work is looking at representing different types of actions using perturbed goal states. The simulation will currently occasionally perform an action which results in throwing the object rather than grasping it. A third area to investigate is what happens once the simulation has grasped the object. Can it then learn to use it to interact with the world? Also how will it adjust the dynamics of the plant as a new object becomes attached. Lee-Hand (2012) has created a model for causative actions, extending his neural network. This uses an input layer as an episode recognition buffer to record when the model perceives some event. This allows the system to learn the perturbation to perform an action and the arm movements which cause the event.

7.2.5 Clustering

The algorithm described in Section 6.4.6 can get into a stale state where it is focused on a single run. This is where the perturbations on a high scoring trajectory fill the entire beam and it is unable to perform a different type of action. To prevent this from happening, one option could be a clustering algorithm. This would group different perturbations so that one type of grasp would not dominate the beam. This would allow one object with different affordances to be grasped in different ways. An example of an object with multiple affordances, with the grasps achieved manually, is shown in Figure 7.1.

7.2.6 Conclusion

The new platform provides a stable and extendable basis for running reach-to-grasp simulations. As the future work currently underway shows, it is already proving useful and forms the basis for an active research programme. Hopefully it will allow more people to concentrate on their reach-to-grasp research.

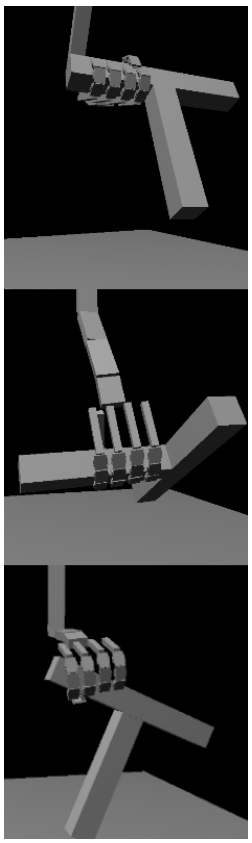

Figure 7.1: The three affordances of the more complex object. Grasping from below (in the top panel), grasping from the side (in the middle) and grasping over (at bottom panel).

References

Abend, W., Bizzi, E., and Morasso, P. (1982). Human arm trajectory formation. *Brain*, *105*(Pt 2), 331–348.

Barbagli, F., Frisoli, A., Salisbury, K., and Bergamasco, M. (2004). Simulating Human Fingers: A Soft Finger Proxy Model and Algorithm. *Haptic Interfaces for Virtual Environment and Teleoperator Systems, International Symposium on*, *0*, 9–17.

Bennett, S. (2005). Development of the PID controller. *Control Systems Magazine*, *13*, 58,62,64–5.

Berthier, N. E., Clifton, R. K., McCall, D. D., and Robin, D. J. (1999). Proximodistal structure of early reaching in human infants. *Experimental Brain Research*, *127*(3), 259–269.

Bhat, A. and Galloway, J. (2007). Toy-oriented changes in early arm movements III: Constraints on joint kinematics. *Infant Behavior and Development*, *30*(3), 515 – 522.

Britannica (2010). Encyclopedia Britannica. www.britannica.com.

Bullock, D., Cisek, P. E., and Grossberg, S. (1995). Cortical Networks For Control Of Voluntary Arm Movements Under Variable Force Conditions. *Cerebral Cortex*, *8*, 48–62.

Ciocarlie, M., Lackner, C., and Allen, P. (2007). Soft Finger Model with Adaptive Contact Geometry for Grasping and Manipulation Tasks. In *WHC '07: Proceedings of the Second Joint EuroHaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, Washington, DC, USA, 219–224. IEEE Computer Society.

Cisek, P. (2005). Neural representations of motor plans, desired trajectories, and controlled objects. *Cognitive Processing*, *6*, 15–24.

Cisek, P., Crammond, D., and Kalaska, J. (2003). Neural activity in primary motor and dorsal premotor cortex in reaching tasks with the contralateral versus ipsilateral arm. *Journal of Neurophysiology*, *89*(2), 922 – 942.

Clifton, R. K., Muir, D. W., Ashmead, D. H., and Clarkson, M. G. (1993). Is Visually Guided Reaching in Early Infancy a Myth? *Child Development*, *64*(4), 1099–1110.

Coumans, E. (2007). Bullet Physics Library. bulletphysics.org.

de Pascale, M., Sarcuni, G., and Prattichizzo, D. (2005). Real-Time Soft-Finger Grasping of Physically Based Quasi-rigid Objects. *World Haptics Conference*, *0*, 545–546.

Fagg, A. H. and Arbib, M. A. (1998). Modeling parietal-premotor interactions in primate control of grasping. *Neural Networks*, *11*(7), 1277–1303.

Gibson, J. (1977). *The theory of affordances*, 127–144. Hillsdale, NJ: Erlbaum.

Gordon, J., Ghilardi, M. F., and Ghez, C. (1994). Accuracy of planar reaching movements. *Experimental Brain Research*, *99*, 97–111.

Harris, C. and Wolpert, D. (1998). Signal-dependent noise determines motor planning. *Nature*, *394*(6695).

Hecker, C. (1997). Rigid Body Dynamics. chrishecker.com /Rigid_Body_Dynamics.

Jeannerod, M. (1996). *Reaching and grasping. parallel specication of visuomotor channels*, 405–460. Academic Press.

jme2 (2009). jMonkeyEngine. www.jmonkeyengine.com.

Jordan, M. and Wolpert, D. (1999). *Computational motor control*, 601–620. MIT Press.

Kakei, S., Hoffman, D. S., and Strick, P. L. (2001). Direction of action is represented in the ventral premotor cortex. *Nat. Neurosci.*, *4*, 1020–1025.

Kawato, M. (1990). Feedback-error-learning neural network for supervised learning. In R. Eckmiller (Ed.), *Advanced neural computers*, 365–372. Amsterdam: North-Holland.

Kawato, M., Furukawa, K., and Suzuki, R. (1987). A hierarchical neural-network model for control and learning of voluntary movement. *Biological Cybernetics*, *57*, 169–185.

Konczak, J. and Dichgans, J. (1997). The development toward stereotypic arm kinematics during reaching in the rst 3 years of life. *Experimental Brain Research*, *177*, 346–354.

Lee-Hand, J. (2012). Actions that are represented by their effects. Masters thesis, University of Otago, In progress.

Lee-Hand, J., Neumegen, T., and Knott, A. (2012). Representing reach-to-grasp trajectories using perturbed goal motor states. *Proceedings of the Pacific Rim Conference on Artificial Intelligence (PRICAI)*. in press.

Len, B., Ulbrich, S., Diankov, R., Puche, G., Przybylski, M., Morales, A., Asfour, T., Moisio, S., Bohg, J., and Kuffner, J. (2010). OpenGRASP: A Toolkit for Robot Grasping Simulation. In *SIMPAR'10*, 109–120.

Lindstedt, S. L., Reich, T. E., Keiml, P., and LaStayo, P. C. (2002). Do muscles function as adaptable locomotor springs? *The Journal of Experimental Biology*, *205*, 2211–2216.

Marr, D. (1982). *Vision*. Freeman.

McCarty, M. E., Clifton, R. K., Ashmead, D. H., Lee, P., and Goubet, N. (2001). How Infants Use Vision for Grasping Objects. *Child Development*, *72*, 973–987.

Miller, A. and Allen, P. (2004). Graspit! A versatile simulator for robotic grasping. *Robotics and Automation Magazine, IEEE*, *11*, 110–122.

Milner, R. and Goodale, M. (1995). *The visual brain in action*. Oxford University Press, Oxford.

OpenGL (1992). Open Graphics Library. www.opengl.org.

Oztop, E. (2002). *Modeling the Mirror: Grasp Learning and Action Recognition*. Ph. D. thesis, University of Southern California.

Oztop, E. and Arbib, M. (2002). Schema Design and Implementation of the Grasp-Related Mirror Neuron System. *Biological Cybernetics*, *87*, 116–140.

Oztop, E., Bradley, N., and Arbib, M. (2004). Infant grasp learning: a computational model. *Experimental Brain Research*, *158*, 480–503.

Oztop, E. and Kawato, M. (2009). *Models for the control of grasping*, 110–124. Cambridge University Press.

Rivers, A. R. and James, D. L. (2007). FastLSM: fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph.*, *26*(3), 82.

Saleh, M., Takahashi, K., Amit, Y., and Hatsopoulos, N. G. (2010). Encoding of coordinated grasp trajectories in primary motor cortex. *Journal of Neuroscience*, *30*, 17079–17090.

Sergio, L. E. and Kalaska, J. F. (2002). Systematic Changes in Motor Cortex Cell Activity With Arm Posture During Directional Isometric Force Generation. *Journal of Neurophysiology*, *89*, 212–228.

Smith, R. L. (2000). Open Dynamics Engine. www.ode.org.

Taira, M., Mine, S., Georgopoulos, A., Murata, A., and Sakata, H. (1990). Parietal cortex neurons of the monkey related to the visual guidance of hand movement. *Experimental rain Research*, *83*, 29–36.

Todorov, E. and Jordan, M. I. (2002). Optimal feedback control as a theory of motor coordination. *Nature Neuroscience*, *5*, 1226–1235.

Tolani, D. and Badler, N. I. (1996). Real-Time Inverse Kinematics of the Human Arm. *Presence*, *5*, 393–401.

Wiki, T. G. P. (2010). The Game Programming Wiki Game Engines. gpwiki.org / index.php /Game_Engines.

Wolpert, D. M., Ghahramani, Z., and Jordan, M. I. (1995). An Internal Model for Sensorimotor Integration. *Science*, *269*, 1880–1882.

Zdechovan, L. (2011). Modelling of objects grasping with neural networks in the iCub robotic simulator. masterthesis.zdechovan.com /thesis-goal.

Zhao, J. and Badler, N. I. (1994). Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Trans. Graph.*, *13*, 313–336.

Appendix A

Appendices

A.1 Physics values used in the simulation

Link	Mass	Length
Upper Arm Section	0.04f	5
Lower Arm Sections	0.04f	2.5
Palm Bones	0.02	2
Finger Bones	0.02	0.7
Finger Pads	0.005	0.1f

A.2 PID Parameters

Link	KP	KI	KD
Arm Joints	0.03f	1.0	5.5
Finger Joints	1	1	1

A.3 Goal Locations

Object	X	Y	Z
Tube1	14.0	-2.5	-1.0
Tube2	12.0	-2.5	-1.0
Tube3	10.0	-2.5	-1.0
Tube4	9	-2.5	-1.0