POLITECNICO DI MILANO
Master course in Computer Engineering
Dipartimento di Elettronica e Informazione

REAL-TIME GRASP POSE GENERATION FOR VIRTUAL HAND

Supervisor: Prof. Marco Gribaudo

Master Thesis by:
Paolo Caputo, serial number: 799137

Academic Year 2014-2015

Abstract

The grasping hand problem consists of designing a non-human hand, which is able to grasp objects autonomously, often trying to duplicate the real human hand and trying to imitate the human brain way of reasoning.

Robotics faces this issue regularly as there's a high need for industrial robots capable of grabbing a huge variety of objects. In the 3D graphics field the problem is slightly different: physics laws can be manipulated or ignored while realism of animations has the greatest importance.

This document describes an experimental way of designing a virtual human-like hand together with an algorithm that allows it to perform realistic finger movements while grasping objects in a 3D environment.

The algorithm has been developed in Unity3D 5.1, while the virtual hand modeling is achieved with Blender 2.7 and MakeHuman software. The aim was to make the algorithm as flexible as possible, so that it worked with different hand structures and models.

A complete 3D scenario in Unity was created in order to test the algorithm under realistic circumstances. This test aims to evaluate the performance of the algorithm in case of eventual application in the video games field. Nonetheless the document explains how robotics might benefit from this experiment as well.

Contents

Chapter 1

Introduction

The issue of designing a non-human hand which is able to grasp different and unknown objects is widely studied under several aspects, as it is a complex problem for which finding a universal solution is impossible. In particular, robotics focuses on physics aspects, while 3D graphics tries mostly to improve the realism of the grasping animation, given that in a 3D environment often physics laws can be manipulated or even ignored.

Grasping hand animation itself includes several problematic aspects that make finding a unique solution extremely hard. Especially concerning human-like hands the main challenge is trying to imitate the human-brain way of thinking and trying to replicate all those tricks that persons unconsciously apply while grasping objects. Although video games represent the most important application field we can rarely observe realistic human-like hand animations even in the most recent products. Mainly because of the limited amount of computational resources, the issue is usually avoided by realizing pre-defined animations and by limiting the complexity of the graspable objects meshes. This works fine for most games but, since the available technology level grows fast, the need for more realistic animations is stronger every year.

The question is slightly different concerning 3D special effects in videos (movies, short films and all kinds of animated 3D scenes): the problem seems less evident because perfect pre-defined animations are achieved

with great efforts and by exploiting high amounts of computational resources. An algorithm that allows to create realistic grasping finger animations with a relatively low computational cost would be helpful in 3D special effects applications without doubts.

The purpose of this thesis is to design an algorithm that allows virtual hands to perform realistic finger movements during the act of grasping small objects. The algorithm is flexible enough to support hand structures having an arbitrary number of fingers and an arbitrary number of phalanges for each finger, although a set of initial operation is still necessary in order to make it works for each hand structure. Given the hand bones structure, the position of the character and an object, the algorithm finds a good grasping position for the character's hand on that object.

A hand positioning system is described, which allows the hand to be connected with the target object and to move around it while always pointing its palm towards it. This is achieved by the introduction of two entities: hook and grasp center. The concept is to link the hand and the object through a hook and apply rotations on the hand with respect to the hook as a pivot point. The hand can also translate along the upward axis that goes perpendicularly through its palm and that always coincide with the hook's one.

Grasp poses are generated by a combination of a generic grasp movement and a target object the fingers collide with. Movements are stored using only an initial pose and a final pose, so that the algorithm can generate an arbitrary number interpolations between them. A classification of grasp movements is shown but the thesis focuses on two among the most important movements: spherical power grasp and thumb-index precision grasp.

The algorithm runs in real-time, showing an intrinsic tendency to be used in video games or other real-time executing applications. It consists of an iterative optimization that in a few instants computes a

correct grasp pose for the hand of a character and a target graspable object. The optimization is based on an evaluation system that allows to compare grasp poses appointing a quality value (real number) to each one of them. As the name suggests a grasp with a greater quality value is preferable with respect to others having lower quality values. The quality is computed with the help of a grasp simulation process that, given the position of the hand and the position of the target object, generates the corresponding grasp as it would be if the hand was in that position. Once the process ends, the quality is calculated as a weighted sum of several arbitrary parameters that can be extracted from the generated grasp pose itself. Particular care is given to the optimization step which is the core of the whole procedure. In one step, a certain amount of grasp poses are compared and the one with maximum quality is chosen to be the base for the next iteration. Depending on the available computational resources, one or more steps can be executed within the time of a frame rendering; the step can be even split among several frames if it afflicts performances too much.

The chosen working environment is Unity 5 while the chosen language is C# (one of the options available in Unity). First of all we created a testing 3D scenario where we developed the algorithm from scratch. After several changes and a long optimization process, the algorithm was applied to a realistic game-like 3D environment that we built with the help of two other applications: Blender 2.7 and MakeHuman. Here it was possible to evaluate performances and to observe the real application result on a basic video game.

Chapter 2

State of the Art

This chapter summarizes the public available research results obtained in the last years concerning the control of 3D virtual grasping hand animations. Some concepts and techniques coming from robotics studies are taken into account as well, given the fact that in this area of interest sometimes robotics procedures are comparable to the graphics ones. Common knowledge and basic methods are discussed first and progressively more specific and experimental techniques follow. In particular we expand the concepts of data-driven animation, physics-based animation, grasp quality function and grasp stability.

2.1 Hand Models

We focus on the 3D representation of human hands mainly because the most faced challenge in this sector is to reproduce closely human body. Plus, human hand has a complex structure (more than virtual robotic hands in general) and an algorithm working fine on it might work very likely also on simpler models.

2.1.1 Human Hand Skeleton

In order to work with animations it's logical to think of manipulating the skeleton of the model (armature in Blender, avatar in Unity) and let the 3D mesh follows it. The skeleton is usually built starting from a root bone and extruding five chains of additional bones (fingers). Bones are set in the same positions as real human bones and although their

number can vary a good standard practice is to assign three bones to the Thumb and four bones to the other fingers [21, 23, 38]. Decreasing the number of bones is doable especially in game engines in order to reduce the cost of animations, although it also reduces realism considerably; as an example in Unity it's possible to animate the hand considering only 2 fingers (Thumb and Middle) with three bones each [34]. The standard nomenclature usually follows the anatomical names of human bones (Figure 2.1). Each link point between two bones is called joint. It's possible to add joints in order to generate more realistic postures but a quantity of 20 joints is usually considered a good approximation that can produce all hand movements without losing too much realism.

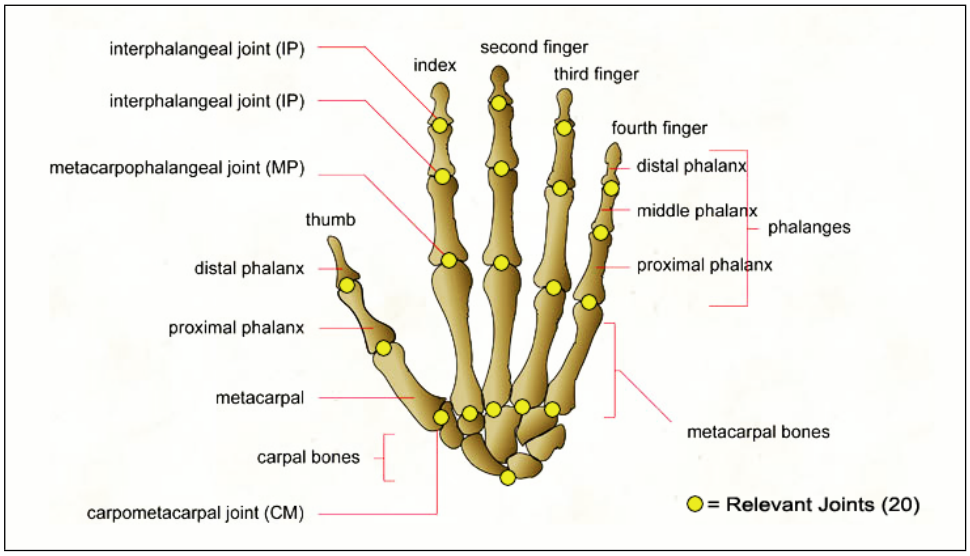

Figure 2.1: Positions and Names of hand relevant joints and bones. (Original image taken from *The Visual Dictionary* http://www.infovisual.info)

2.1.2 Human Hand Constraints

Despite the number of joints the managing effort is lowered if we take into account the anatomical limitations of the real human hand joints. By fact a logical and satisfactory approximation leads to consider two

rotation axes available for the bones and just one for distal phalanges and middle phalanges. Moreover, limitation on rotation angles helps producing realistic postures. A good study on hand constraints was done by John Lin, Ying Wu and Thomas S. Huang [21]. According to them there are three types of constraints:

Type 1: Anatomical joint limits.

Type 2: Angle limits due to tendon links between fingers. Often called dynamic constraints.

Type 3: Angle limits that force a natural posture. These limits are hard to set as they're due to all tendon and muscle links in human body.

A common and necessary approach obtained from hand constraints is to reduce joint degrees of freedom while producing a good grasping posture for the hand: searching only for possible and natural hand postures heavily decreases the computational cost of any grasping algorithm.

2.1.3 Non-Human Hands

Infinite types of virtual hands can be designed but in general all the various structures follow the human hand's one: a root bone and an arbitrary number of fingers (chains of bones) composed by an arbitrary number of phalanges. Usually the number of fingers and the number of phalanges are lower than the relative ones in the human hand structure. Size of phalanges can vary. Some studies show experimental hand designs, mainly based on the purpose of the project; as an example, we mention a hand with soft hemispherical finger tips designed to grab small polyhedral objects [18]. One of the most useful application of virtual non-human hand animation is simulating robotic hands: it may speed the testing process by saving production costs. GraspIt! [24] is a great instance of robotic grasping hand simulator.

2.2 Data-Driven Animations

When we speak of animating the 3D model with pre-defined animations, we mean applying a set of movements already created by some-

one else to it. Those animations can be either build from scratch or recorded with motion capture techniques (in the last years the second option has become the standard one as technologies for capturing human movements are cheaper and more advanced). Even though having ready-for-use animations seems to be certainly helpful for our task, a big disadvantage is represented by the infinitely high number of possible hand grasp configurations. Not only the amount of different graspable object is great, but also each one of them can be grabbed from different positions and by different types of hand. In general the data-driven approach is adopted with the purpose of defining a basic set of grasp postures or movements; after that other techniques must be applied to obtain realistic grasp animations [3, 27, 41].

2.2.1 Grasp Movement Classification

Because of the reason discussed before, lowering the number of possible grasp movements (without considering eventual collisions with grasped objects) is fundamental. An excellent approximation of grasp choices has been presented by Cutkosky and Howe [20]:

Alternative methods for generating correct grasp movements have been presented; they usually require additional devices interacting with the user's hand. We mention the grasp generation with real-time tracking of a camera taking a real hand grasping the object [41] and the Tango device: a ball which measures contact pressures on its surfaces as well as acceleration; with such a device, a grasp movement is calculated based on the user's hand contact with the ball's surface [19].

2.3 Physics Based Animations

In 3D graphics engines physics simulation has reached high levels of realism and performances. In this context the hand is composed by physical bodies and motors that apply forces to them. Although it's true that a 3D physics engine can quite easily manage collisions and anatomical constraints, the real challenge is to design and implement a controller that supervises all the motors of the hand [2, 27, 15]. Predefined animation can't be used in general on a physical body, therefore

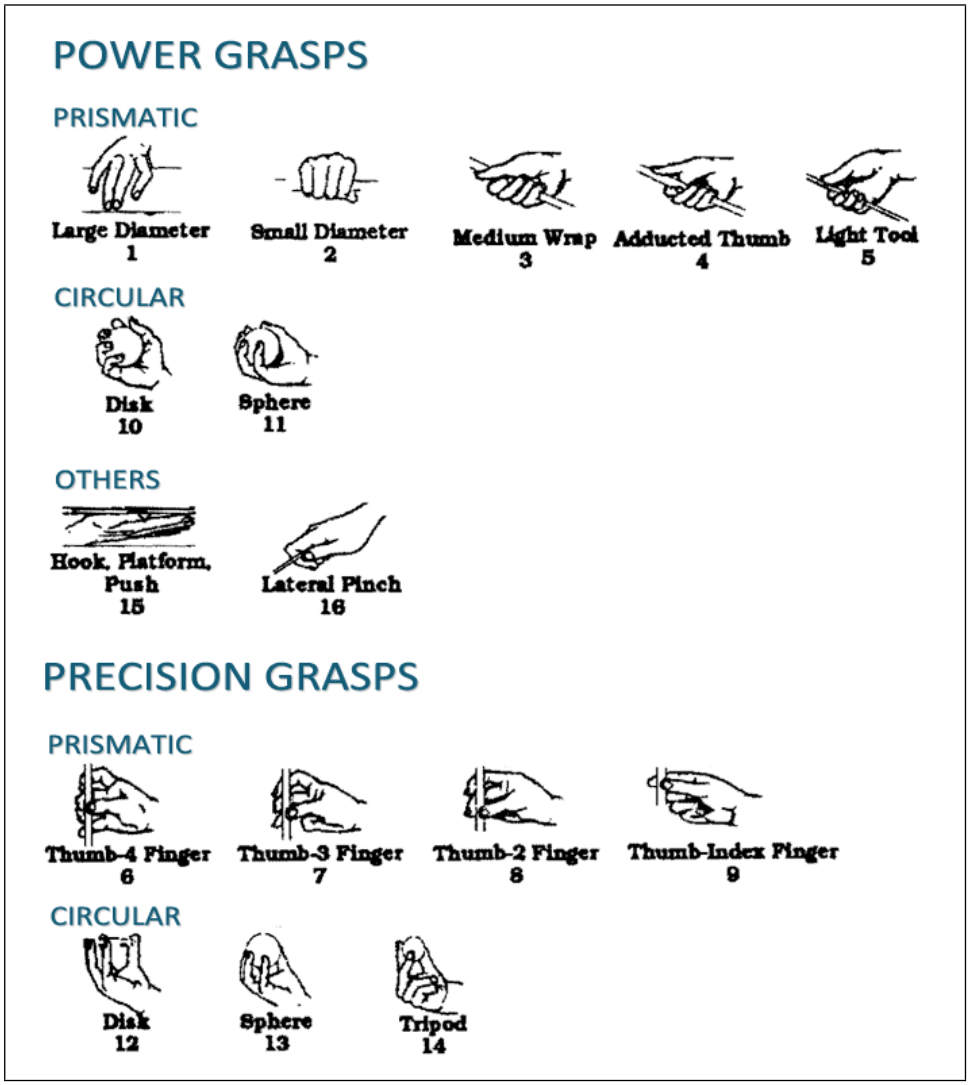

Figure 2.2: Grasp Classification taken from Grasp Choices diagram by Cutkosky and Howe [7].

even producing a simple movement requires particular care. Despite that, especially in the robotic hand simulation field, some prefer to adopt this methods considering that, after all, robots use motors for movements as well.

2.4 Grasp Evaluation

As said before, the number of possible grasp postures is very large and it depends on the hand structure, on the grasped object and on the chosen grasp movement. This makes a strategy to choose among all them necessary. This strategy is the core of the whole grasp algorithm. Searching for a perfect grasp (among all the possible ones) often means being able to evaluate each grasp and to compare it to the others in terms of some arbitrary parameters. Defining the comparison is a complex process because it means describing how human brain decides that a grasp is better than the others. Various methods have been proposed; aside from several exceptions, we can synthetically identify two types of analysis that have been developed more than the others: realism analysis and stability analysis.

2.4.1 Grasp Realism

Evaluating a grasp posture from a kinematic point of view is not a straightforward task. Checking that the posture is possible and natural is mandatory but a harder challenge is to define the analysis procedure that, given a grasp posture and an object, states how realistic and suitable is the posture for that object. A perfect solution for designing such a procedure doesn't exist especially because of the variety of tasks to be performed by the hand that lead to different types of analysis. An example of comparable parameter is the pinch distance (gap between the thumb tip and another finger tip chosen before, usually the index tip or the middle finger tip) [29]. Another possible parameter to check is the empty space between the hand joints and the object after the movement is complete: minimizing this parameter should lead to a correct enveloping grasp [6].

2.4.2 Grasp Stability

If the task is to design a robotic grasping hand simulator then it might be mandatory to verify whether a grasp is stable and secure. Therefore an analysis of the forces applied to the object must be carried out [8, 33, 36]. A physics engine is not strictly needed because it's enough to compute all the forces applied but the object must possess a physical body. If its body is raised and held firmly by the forces then the grasp is stable. Grasp stability check is almost never necessary in 3D graphics applications (video games, special effects applications) because physics can be manipulated or ignored in those cases. The drawback is that some unstable grasps may be considered valid by the evaluation leading to unrealistic effects during the execution.

2.5 Locating Contact Points

Another kind of information we could use while searching for an optimal grasp is to analyze the object and to identify possible contact points for the fingers [5, 16, 39]. In 3D environments the algorithm usually knows the meshes of the objects while in robotics applications shapes could be recognized by cameras [31]. Although locating contact points is a difficult task, it brings a huge advantage: the number of possible grasp postures decreases considerably. Plus, theoretically unrealistic collisions are avoided since collision points are known. Found contact points should satisfy some conditions like their placement dependencies with respect to the hand structure; their positions should also be compatible with a stable grasp. As a consequence it becomes necessary to write an algorithm that is able to compare the different combinations of contact points in terms of grasp realism and stability. In any case, the contact points information must be integrated with other strategies on order to produce good results. One example of such strategies is surface matching.

2.5.1 Surface Matching

Once contact points have been located, it's even possible to perform a surface matching between parts of the object mesh corresponding to

the identified contact points and the surface described by the fingers performing a grasp movement [20, 40]. Maximizing the similarity rate should produce a correct grasp. This solution however is suitable only when searching for enveloping grasps; for instance a picking movement (using only the thumb and the finger) can't be evaluated correctly by this method.

2.6 Reducing Complexity of Graspable Objects

The number of graspable objects is nearly infinite. As we have seen this represent one of the biggest obstacles of grasping animation design. Graspable objects can be composed by different and separate meshes, they can have some preferable parts where to be grabbed, they can require ad-hoc grasping movements and their meshes can be concave or convex. For such reasons it's rational to think of simplifying the object structure when possible. This requires additional efforts but it may lead to a sensible simplification of the grasping algorithm. Surely there are several good approaches to accomplish the task; here after we cite two of the most common ones.

2.6.1 Subdividing Objects in 3D primitives

It's well know that basically all 3D graphics and physics engines perform better and faster working with primitive 3D shapes rather than with complex 3D meshes. A good strategy could be subdividing the 3D mesh of the graspable object obtaining a set of 3D primitive shapes (Figure 2.3). Grasp posture generation on 3D primitives can be heavily optimized allowing to build a fast and precise algorithm [25]. The real challenge is to design an algorithm that performs good subdivisions on all kinds of objects producing acceptable approximations. In fact, if the primitives don't fit the 3D shape perfectly, unrealistic graphics errors like wrong collisions and mesh penetrations may be generated.

2.6.2 Locating Graspable Parts of the Object

Some studies suggest that a better plan is to locate on the object all the sections that possess certain graspable features. In this way only those

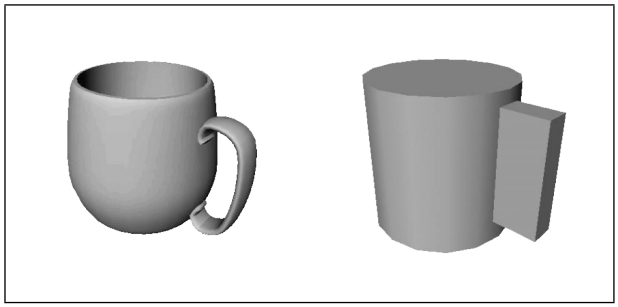

Figure 2.3: Example of 3D primitive subdivision performed on the 3D model of a mug. (Image taken from [25].)

parts are taken into account and the rest of the object can be ignored (not totally, as the whole object must be considered for collisions in any case). An example is the Plumber method proposed by Tolga, Mortara, Patanè, Spagnuolo, Vexo and Thalmann [1]: through intersections of spheres centered on the mesh vertexes it allows to identify tubular sections of arbitrary radius and size which can be marked as graspable components of the object.

Chapter 3

Objectives

The grasping problem is widely studied by researchers coming from several fields, among which robotics is surely the most present. In general the objective of the studies is to make a single non-human hand (robotic or virtual) able to grasp objects of any shape and size, according to its physical possibilities. However, this goal being so complex to achieve, it can be represented as a set of sub-tasks like the grasp stability assurance, the fulfillment of certain hand constraints or the production of realistic finger movements. Depending on the research field and based on the purpose of the project, some sub-tasks might be crucial while others might be ignored.

We decided to face the single-hand grasping problem from a 3D graphics point of view, adjusting the target for all those applications that need to generate grasping hand animations, such as video games or special effect software for movies. Therefore we needed to define our set of sub-tasks which had to be addressed. In broad terms, the goals we chose to accomplish are:

- Realistic choice of grasping pose
- Realistic object-finger collisions
- Algorithm adaptability for different hand structures
- Algorithm adaptability for different object shapes
- Real-time execution

In the following sections we discuss these goals in detail, pointing out the meaning of each term we used. In Section 3.4 we discuss the operation level of the algorithm; that is to say how the algorithm should manipulate the hand and manage its contact with the objects. Finally in Section 3.5 we explain what the algorithm should give as output.

3.1 Virtual Hand

First of all we have to declare what is a virtual hand because it's the main entity of the whole process. With this term we indicate a 3D object composed by a mesh and a skeleton. The mesh is linked to the skeleton and it's moving and blending according to the skeleton movement and rotation (attaching a mesh to a skeleton is the modern standard technique for moving complex 3D objects). The skeleton is the internal manipulable structure; its name is not accidentally related to the human bones, indeed the virtual hand skeleton follows exactly the bone placement of a real human hand, although with some approximation: it's composed by a root bone, which is called wrist, and five chains of bones that corresponds to the fingers.

3.1.1 Hand Variations

Because of the variety of possible applications (just think of humanoid characters in video games), it must be possible to apply the same algorithm to different hand structures. A condition that must be satisfied is that the hand must be prehensile: it must be composed only by fingers that are attached to the same primitive shaped body and that fold in the same direction. The most common alterations are made on:

- Number of fingers (at least two)
- Number of phalanges per finger
- Finger placement
- Hand size and finger size

3.1.2 Grasp Pose and Grasp Movement

The grasp pose is the final result of the act of grasping: it includes position and rotation of wrist and all the fingers when the object is grabbed. In other words it's the output of a grasp generation algorithm and it's computed using all information available: position of the character, hand structure, object structure and grasp movement. The grasp movement instead is just the path that the fingers follow without considering the grasped object. As an example, just think of a open hand that blend its fingers until it becomes a fist. Starting from a grasp movement an infinite number of grasp poses can be generated, especially because of finger-object collisions that modify the final rotation of the fingers.

3.2 Graspable Objects

It's necessary to define the features of a graspable object in order to know when the algorithm can actually be used. In order for an object to be graspable with respect to a single hand at least one part of its mesh must have size along at least one axis smaller than the average length of the fingers. The size of the object should not be an issue as long as the previous condition is satisfied. This description includes all objects that are intuitively graspable with one hand, not taking into account their weight. As said before, because of the nature of possible applications, we don't consider the physics while computing the grasp pose. What is not included in the description is any object that necessarily requires two hand to be grasped (or a bigger hand than the considered one) and any object that can't be grasped by hands. Concerning the object's mesh, the only requirement is that it must be closed: all its faces must be connected and it must not have missing faces that result in holes.

3.2.1 Concave Objects

A convex mesh is a 3D set of faces that, drawing a straight line anywhere in 3D space, is crossed by this line only once. A concave mesh might be crossed several times by a generic 3D straight line. Despite

the fact that, from the perspective of graphics engines, working on concave meshes requires much more computational effort than working on convex ones, the algorithm must be able to manage concave objects as they represent the majority of graspable objects in reality.

3.3 Improving Realism

The algorithm should work on graphic simulations of realistic environments (not necessarily real) so achieving a good level of realism is fundamental. The strategies in this direction are different, but we identified three objectives that, if pursued, lead to realistic enough simulations:

- Correct finger-object collisions
- Fulfillment of human body constraints
- Real-time execution

Surely these objectives are not the only possible ones: correct colliding phalanges blending could be in this list, as well as correct object response to the forces applied by the fingers. Many other detail improvements could lead to a higher level of realism, but the one we chose assure the production of a simulation that is good enough to be used by the target applications. Plus, as real-time execution is needed, it's essential to develop an algorithm that doesn't require a high computational effort, so we want rather to reach only the crucial objectives.

3.3.1 Finger-Object Collisions

One of the biggest problems of using pre-defined animations is that, due to the huge variety of graspable objects, the fingers usually don't adapt their movement to the shape of the object; after all it's impossible to store a grasp animation for each different object and for each possible position the object is grasped from. If, like in this case, fingers don't consider the shape of the object then two unwanted phenomena show up: mesh penetration (hand's mesh penetrates object's mesh) and/or empty space remaining between fingers and object at the end of the

grasping movement. These problems can be frequently observed in modern video games especially because for most games the grasping action is not a fundamental one so developers are inclined to ignore this issue as it's considered not worth of the computational effort that it requires.

3.3.2 Human Body Constraints

Naturalness of a grasp pose has a great importance concerning realism. Clearly the algorithm must not allow the hand to perform unnatural finger rotations. However other constraints are given by tendon and muscle disposition in the whole arm. The position of the elbow may allow different rotations for the wrist, while the character posture (considering all the body) influences the possibilities of the arm. Perhaps building a full-body constraint system would be possible by using a physics engine, but in our case an approximation is necessary. In any case we are determined to solve the grasping problem focusing on the hand pose and on finger positions mostly.

3.3.3 Real-Time Execution

Since target applications set include mostly video games and real-time simulations, it's mandatory to design an algorithm which is able to perform in real-time. This means that it must require a small computational effort as it might be executed together with other computations (at least graphic rendering computations).

3.4 Abstraction Level of Operation

The algorithm manipulates the hand bones and manages their contact with grasped objects. In graphic engines bones usually correspond to simple entities located in 3D space. Therefore these entities have a 3D position field, a 3-axis rotation field and a 3-axis scaling field. The algorithm can operate on those fields by changing the numeric values. Modifying those fields means performing the three fundamental 3D transformations: translating, rotating and scaling. Translation and

rotation are both metric transformation (they preserve all metric properties of the entity) while scaling is an affine transformation (only affine properties are preserved, such as angle between lines). In our case the algorithm doesn't exploit the scaling transformation as it's unnecessary, so we may conclude that it only applies metric transformations to the skeleton parts. The bones compose chains, being connected one with the other. In particular one bone may have a predecessor (all except the wrist) and a successor (all except the tips). When applying a transformation to a bone, it's automatically applied to all its successors. We'll discuss this in detail in Chapter 6. Concerning the contact between the hand and the objects, the algorithm must exploit all those features of the graphic engine that allow to perform collisions detection between entities in 3D environments. Summarizing, the algorithm can:

- Translate entities
- Rotate entities
- Detect collisions between entities

The computation should be performed within the time of a frame rendering, so it must take not more than a few milliseconds. Eventually the algorithm may be able to split the computation among several frames as long as the real-time effect is preserved.

3.5 Output of the Algorithm

As a final result, we obtain a grasp pose: position and rotation of all the bones (wrist and fingers) in 3D space. Each entity will know the target values for its fields. In this way a smooth movement of the character can be performed using simple inverse kinematics techniques. Grasp poses are different depending both on the grasped object and on the position the object is grasped from. The major benefit coming from this algorithm is that grasp poses don't need to be stored anymore, but they're computed just before the grasping action. In Figure 3.1 there are some examples of grasp poses computed by our algorithm.

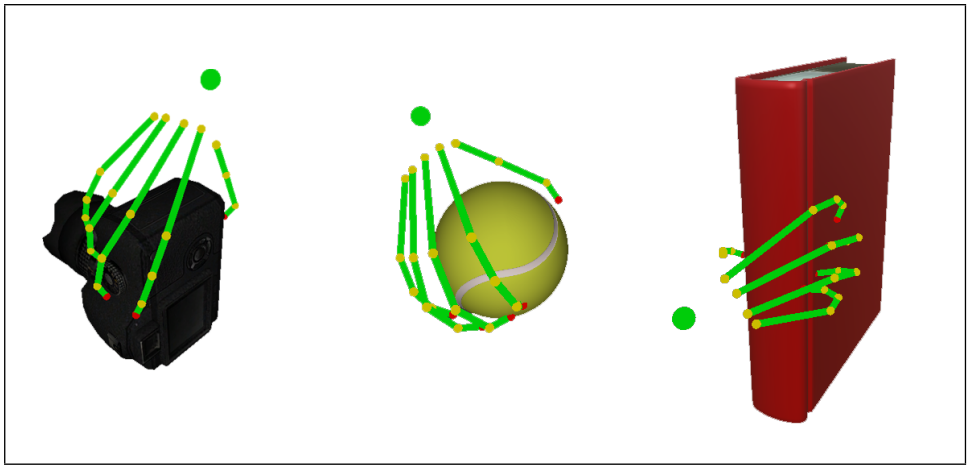

Figure 3.1: Output examples.

Chapter 4

Human Hand Model

In this chapter we analyze in detail all the feature of the hand structure which is the principal entity that the algorithm manipulates. Also we describe the necessary requirements to have a fully compatible hand 3D model and we expand the explanation of hand structure variations already mentioned in section 3.1.1.

4.1 Human Hand

In order to build a 3D model that imitates a human hand, it's mandatory to study the structure of a real hand and its composition. If observed from outside, it shows three different components:

- Wrist
- Palm
- Fingers

Although the wrist may be considered as a component of the arm, we take it into account because we need it to manage the rotation of the whole hand for which the wrist is responsible. The palm and the fingers are the components that experience the contact with the grasped object. Finger movements are much more evident than palm deformations and in some cases an approximation that ignore the latest ones is enough. We will demonstrate that in our case this approximation wouldn't bring any difference in terms of computational effort

so we consider palm deformations being of the same kind as finger movements. Fingers are composed by three phalanges each, with the exception of the thumb that has only two phalanges. For what concern the hand anatomy, the only notions we need are the bones composition and placement and the rotation constraints imposed by bone joints, tendons and muscles. However we don't need any specific information about tendons and muscles themselves.

4.1.1 Human Hand Bones

Following the real bones disposition and classification is useful and convenient. According to the anatomic nomenclature hand bones belongs to three sets:

- Carpal bones
- Metacarpal bones
- Phalanges

In Figure 4.1 the sets are shown together with the placement of each bone. We don't go into the details of carpal bones composition because, as we point out in Section 4.2, we don't consider them as single bones. Metacarpal bones and phalanges are basically of the same kind but, if we consider the whole hand composition, metacarpal bones are situated in the hand palm while phalanges correspond to the relative finger phalanges. A further subdivision labels the phalanges attached to the metacarpal bones as proximal phalanges, while the remaining are called distal phalanges (one for the thumb and two for the other fingers).

4.1.2 Mechanical Constraints: Fingers

From now on we consider a reference on the hand. In Figure 4.2 we show the hand in default configuration that is every component has rotation 0° around all the three axes. The z-axis is the one that goes from the wrist and follows the fingers; the x-axis is the one crossing all the knuckles horizontally (except the thumb one); the y-axis is perpendicular to the hand palm surface and it's obviously orthogonal to the others.

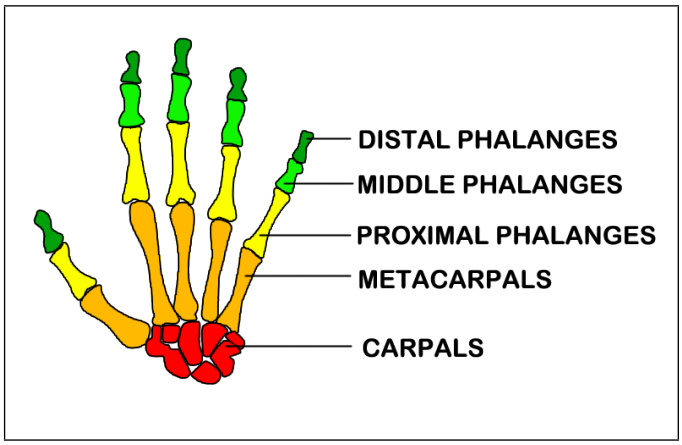

Figure 4.1: Hand bones classification.

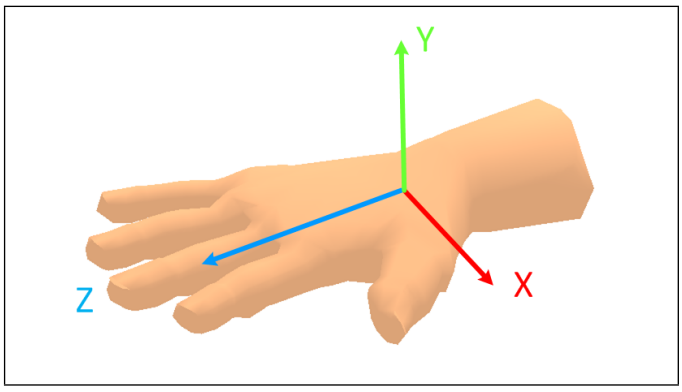

Figure 4.2: Reference axes shown on the hand in default position.

Because of the joints nature and because of the tendon and muscle limitations bones can rotate only within certain angle ranges and only around a limited number of axes. The thumb axes have different orientation. Concerning the fingers we get our information form a study conducted by John Ling, Ying Wu and Thomas S. Huang [21]. They point out an excellent classification of finger constraints as well as their computation in terms of angles (values are necessarily approximated). The subdivision consists of three types of constraints:

4.1.2.1 Type I: Static Constraints

These constraints correspond to the anatomical limitations of the fingers. Distal phalanges can only rotate around x-axis (flexion); Proximal phalanges and the thumb metacarpal can rotate around x-axis (flexion) and around y-axis (abduction/adduction). The metacarpals of the other fingers can only rotate around x-axis (flexion). The following equations describe the angle limitations (in Figure 4.3 angles are referenced):

$$\begin{aligned} 0^\circ \leq \alpha 1 \leq 90^\circ \\ 0^\circ \leq \alpha 2 \leq 110^\circ \\ 0^\circ \leq \alpha 3 \leq 90^\circ \\ -15^\circ \leq \alpha 4 \leq 15^\circ \end{aligned}$$

A common approximation is to consider no abduction/adduction movement for the metacarpals and for the middle finger proximal phalanx:

$$\begin{aligned} \beta_{MIDDLE} = 0^\circ \\ \gamma = 0^\circ \end{aligned}$$

4.1.2.2 Type II: Dynamic Constraints

This group includes all those constraints that link fingers during the motion. They can be further classified in two sets:

- **Intra-Finger Constraints:** Limitations due to the rotations of the other bones in the same finger. For instance the two distal

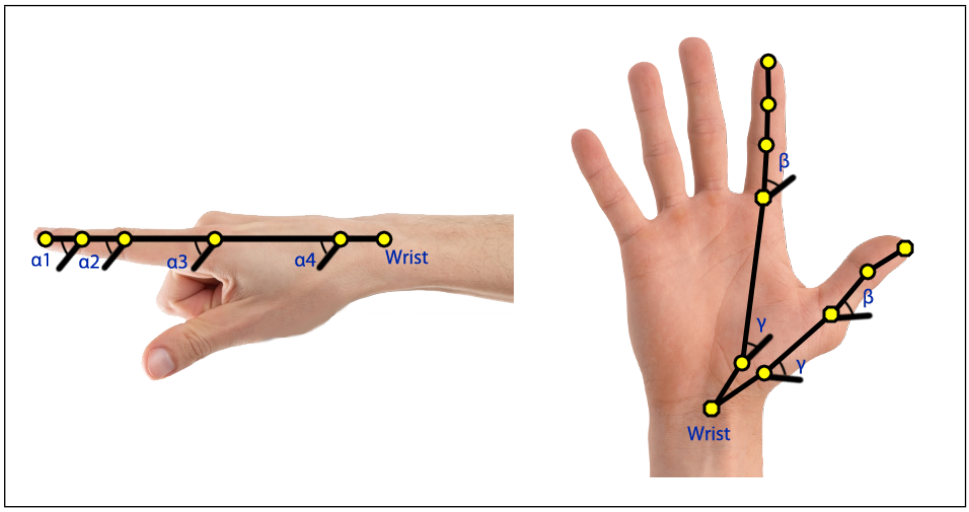

Figure 4.3: On the left, the blending angles are shown; on the right, the abduction-adduction angles.

phalanges of the fingers can only blend together. A commonly used approximation states:

$$\alpha 1 = \frac{2}{3} * \alpha 2$$

- **Inter-Finger Constraints:** These constraints are the ones imposed by the other fingers movements. As an example, think of a flexion movement of the index: the middle finger is forced to follow it, even if not performing a complete flexion. Inter-Finger constraints are not easy to formalize. It's possible to find some studies that show a few equations about them but in our case those equations are not necessary.

4.1.2.3 Type III: Naturalness Constraints

The naturalness of a hand pose has nothing to do with anatomical limits. Fulfilling this type of constraint would mean knowing how the brain works while performing any action with the hand. In other words, it would mean knowing exactly which movement a person prefers in order to accomplish every manipulating task. Not only these constraints are

almost impossible to describe with equations, but also they can vary for each character. We simply avoid a formalization of them, entrusting our intuition in generating natural hand poses by the imitation of reality.

4.1.3 Mechanical Constraints: Wrist

Wrist constraints require particular care. If we get along with the previous classification and if we make the same assumption about the reference axes and the angles, we can approximately describe the *Type I* constraints of the wrist as (angle references in Figure 4.4):

$$-55^\circ \leq \theta x \leq 55^\circ$$
$$-45^\circ \leq \theta z \leq 110^\circ$$
$$-30^\circ \leq \theta y \leq 45^\circ$$

We can't say anything about *Type II* constraints as the wrist is considered as a single bone. For what concern the *Type III* constraints the same line of reasoning already introduced can be applied here.

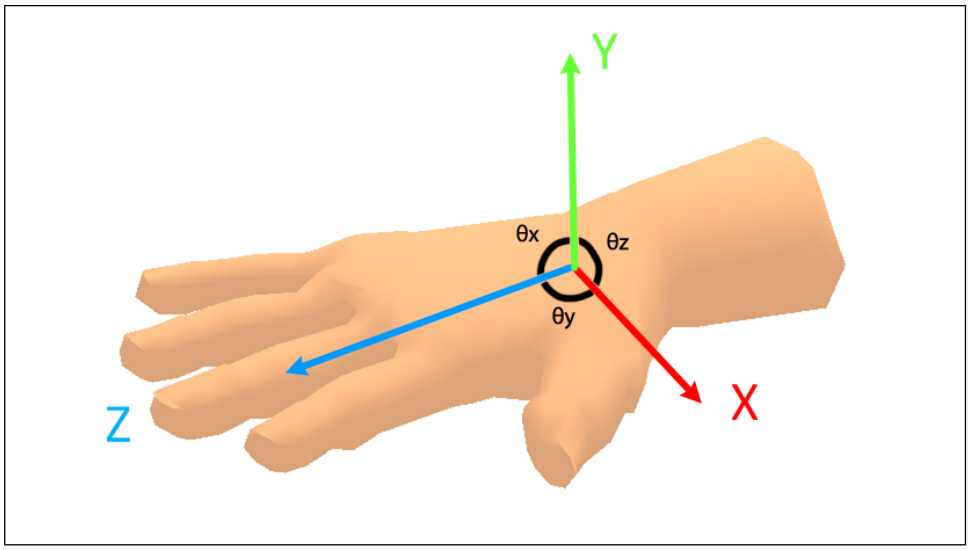

Figure 4.4: Wrist rotations is determined by the main reference axes.

4.2 Structure Approximation

Although there are several possible approximations for the hand skeleton that can be used in 3D graphic engines, we mean to employ the most complete between the common ones. We name each bone and each joint as we often need to distinguish between them. As carpal bones don't move (they move in conjunction with the wrist) they're not considered as bones, but rather an extension of the wrist bone. The wrist itself is considered as a single short bone because this allows us to perform wrist rotations without the need to involve the arm. In Figure 4.5 we show the structure elements and their names.

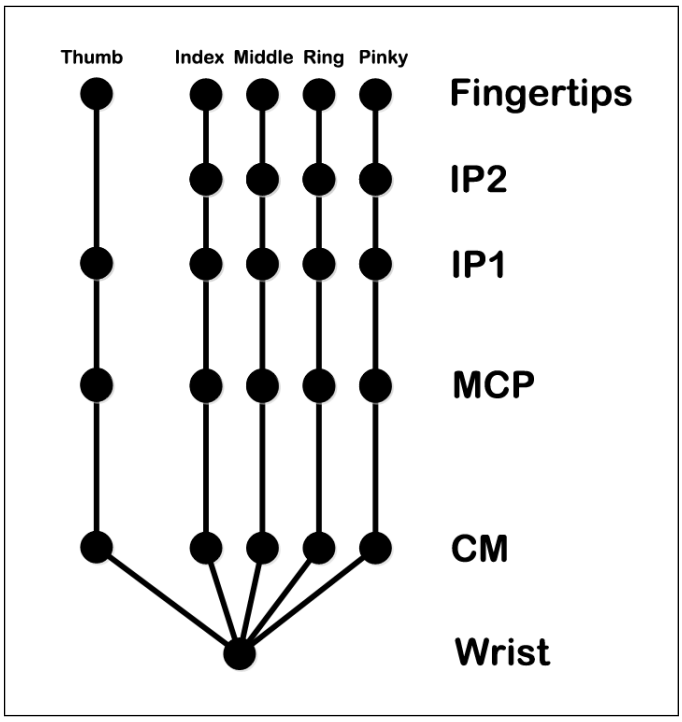

Figure 4.5: The sketch of the hand structure we use. CM: Carpometacarpal joints - MCP: Metacarpophalangeal joints - IP: Interphalangeal joints.

4.3 Structure Variations

Because of the nature of the target applications, it's fundamental to specify a compatibility scheme for the hand structure. Virtual hands can be of any kind but mostly they can be grouped as human, humanoid and robotic. All those categories have some features in common. The algorithm supports any structure belonging to these as long as it satisfies some requirements:

1. The structure has a single root bone and all the other root bones derive directly or indirectly from it.

2. Fingers are composed by chains of single bones. Chain roots aren't necessarily attached to the main root bone but they must be virtually connected in any case. Number of phalanges per finger is arbitrary as well as the number of fingers.

3. All of its grasping actions are intended to grasp a single object or a single part of an object. In other words the grasping action must have only one focus point. For instance, a robotic hand composed by six fingers, of which three are meant to grasp a part of an object and the other three are meant to grasp another part of the same object, would not be compatible with the algorithm.

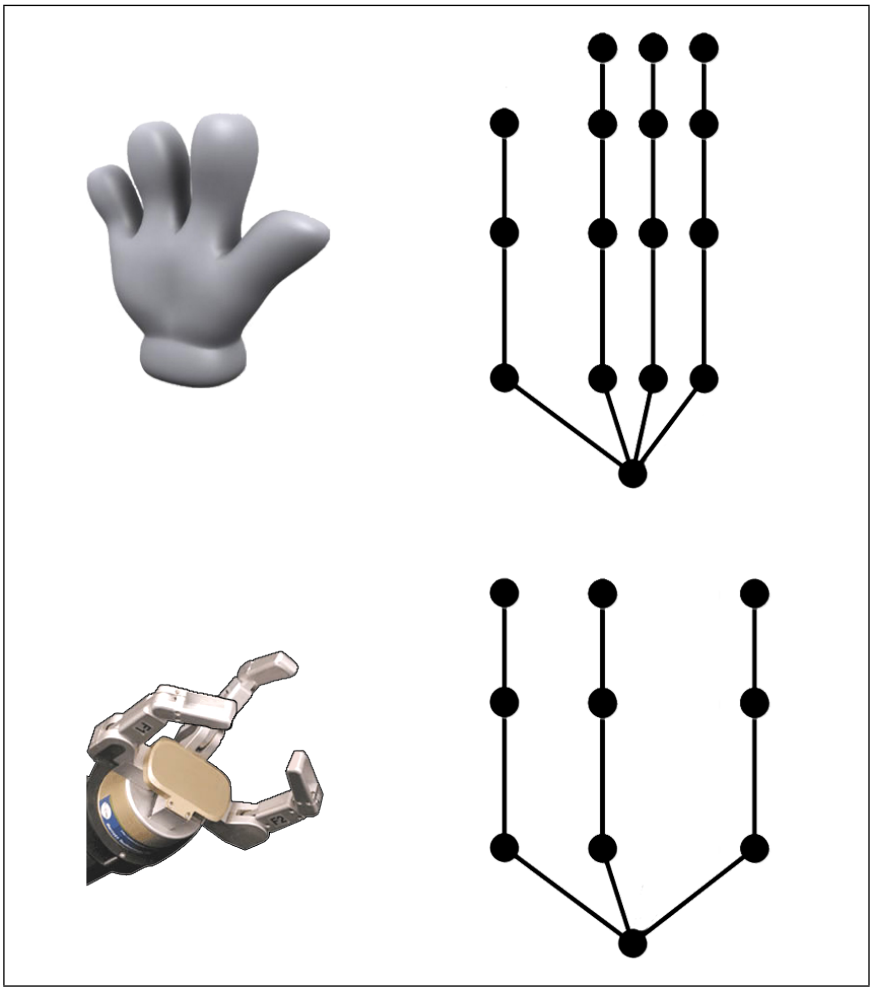

Figure 4.6: A couple of examples of hand variations compatible with the algorithm. On the right, the respective eventual structure.

Chapter 5

Algorithm Design

In this chapter we illustrate the concepts exploited by the algorithm as well as the ideas that led us to develop its core. Imitating the line of reasoning pursued by human brain was our first purpose so, before getting into the design process, we tried to analyze basic real-life grasp movements performed by ourselves and others in order to understand what is the logic behind such a natural and intuitive movement. In the following sections we describe the main phases of the developing process, from the basic picture of a strategy to the final output. In the last section we summarize the concepts and we present an outline of the whole algorithm which will be taken as base for the implementation.

5.1 Iterative Optimization Approach

As said before the algorithm must work within the time of a frame rendering. This is a hard task to accomplish if we want to complete a grasp pose computation all at once. The best solution is without doubts splitting the computation between frames; in this way we obtain much more computational resources depending on the time taken to exploit them. For instance if we split the computation among 30 frames and if the graphic engine renders 30 frames per second then in one second we would obtain 30 times the computational resources that we could exploit in one frame. Considering such availability together with the need for designing simple and time-saving strategies, a good compromise is to think of *iterative optimization*. In broad terms, we

generate one (or more) grasp pose for each frame, we evaluate the goodness of such pose and frame by frame we try to improve the pose by moving the hand around the object and comparing the evaluations. In this way the output is improved after each frame and a suitable one is found provided that the time necessary to render enough frames has passed. It's possible to use this approach without considering the actual frame rendering: simply splitting the computation among several time instants is an option; another one could be exploiting multi-threading as long as the graphic engine allows it. Summing up, we could split the computation in three ways:

- Distribution among frames
- Distribution among time instants
- Multi-threading

The core of an iterative optimization strategy is represented by an evaluation system that allows to compare poses according to some evaluation criteria. The comparison parameters are discussed in Section 5.5. Instead in Section 5.6 we explain when the evaluation is performed and how the result values are managed.

5.2 Hand Positioning

A fundamental issue to face is hand positioning: how to relate hand and grasped object in order to carry out the evaluation mentioned in the previous section? We took care of this by developing a simple system based on 3D space relations. We concentrated on the fact that during the great majority of real-life grasping actions, the hand focuses on a single potential point in the space located for example inside the fist when performing a power enveloping grasp. We call this point *Grasp Center*. In any grasped object we can also identify one or more potential points in the space that would be suitable locations for the grasp center in order to generate a correct grasp pose. We call those points *Hooks*.

Although a target object might host more hooks, for any grasp action we consider only one hook. Local axes of grasp center and hook must be in sync. They are set such that the y-axis corresponds to the y-axis of the hand as shown in Figure 5.1.

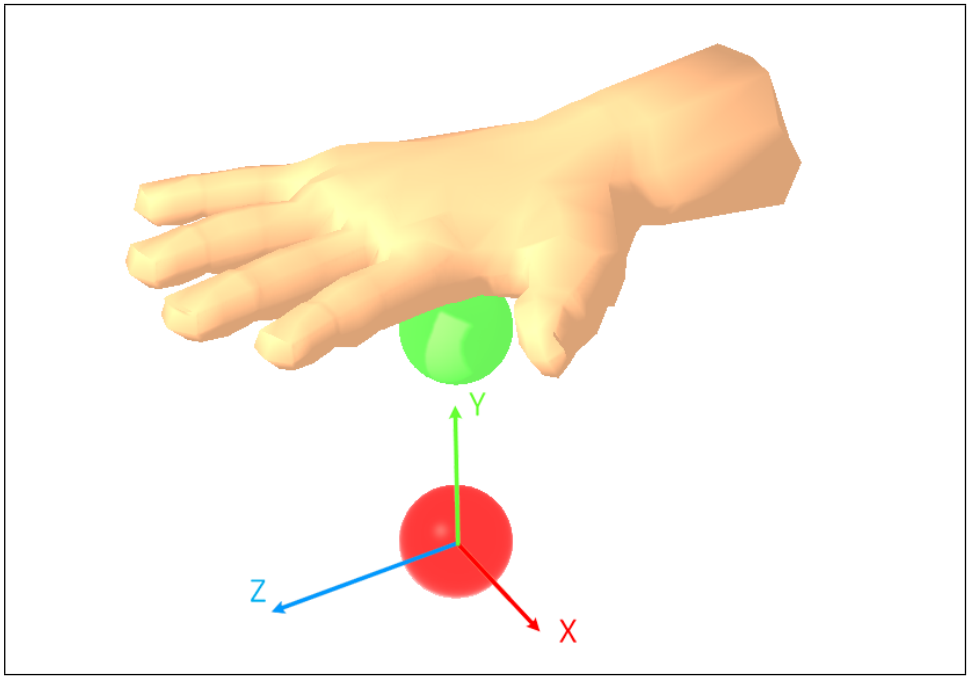

Figure 5.1: The green ball represents the grasp center while the red ball is the hook. Reference axes are the same for hand, grasp center and hook. Rotations are only applied to the hook in order to move the hand around it.

Moreover hook, grasp center and hand must be linked such that all transformations applied to the hook are performed also on the grasp center and all the transformations applied to the grasp center are performed also on the hand. The pivot of all those transformations is the hook. In this way, by rotating the hook, we can move the hand around the object making it always points toward the hook. Examples on rotations applied to the hook are shown in Figure 5.2. In the ideal situation the grasp center and the hook perfectly coincide. Because of hand-object collisions this is not always possible, so the grasp center

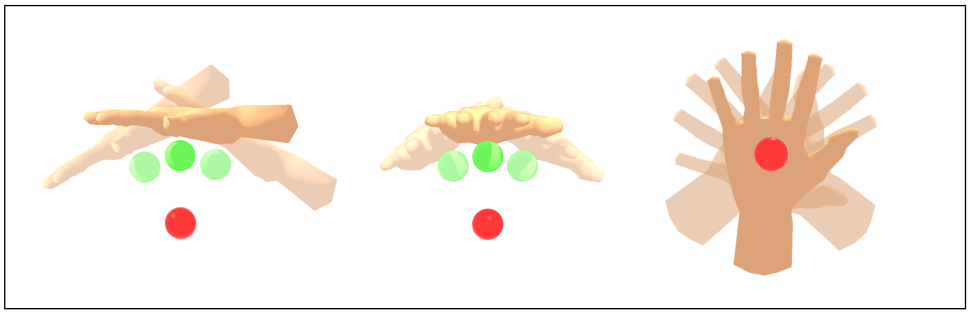

Figure 5.2: Possible rotations of the hook. From left to right, rotations around x, z and y axes are shown.

(and all the hand together with it) must be able to translate along its y-axis just enough to set the hand free from collisions.
In other words, hand position is entirely defined by:

- Rotation of the hook
- Translation of grasp center along its y-axis

5.2.1 Grasp Center

Grasp center has a certain relevance because as we'll describe (mostly in Section 5.5) important parameters depend on its position. Therefore its placement must be carefully thought out. For some grasp movements deciding its location is straightforward: the thumb-index precision grasp clearly needs its grasp center in the central contact point between the thumb tip and the index tip. The spherical power grasp on the other hand requires more attention while determining the grasp center position. There is no mathematical way to state where the grasp center should be in 3D space for each grasp movement but intuition is sufficient in order to place it. For instance we instantiate the grasp center in the middle of the fist for the spherical power grasp. It's enough to remember that the grasp center stands for the grasp's focus point.

5.2.2 Hooks

Individuating hooks in a target object is done manually. For most of small objects (a tennis ball, a glass, a rubik cube...) we can consider

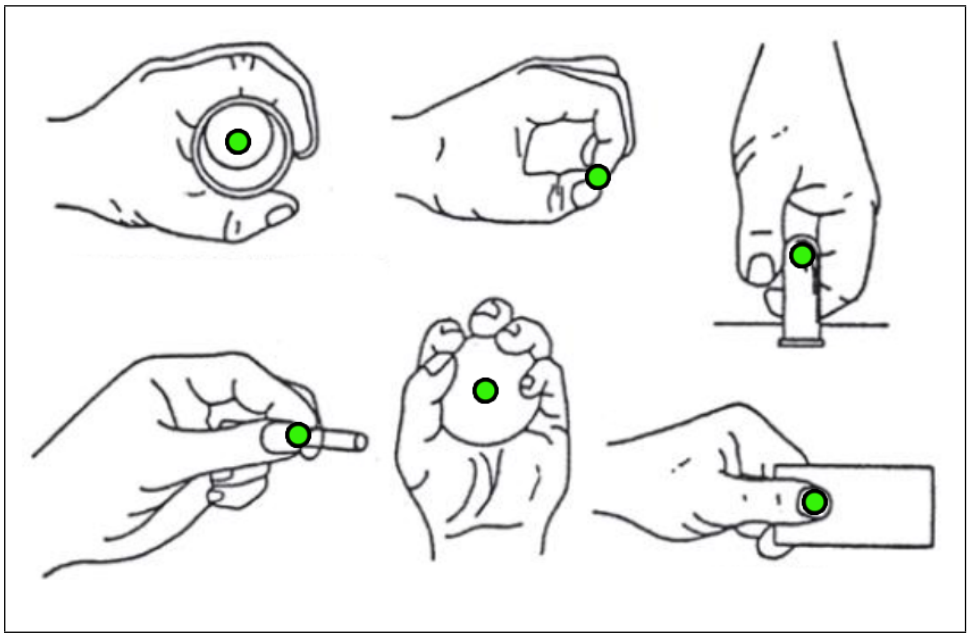

Figure 5.3: Green dots represent the positions of grasp centers. The positioning is quite intuitive but it has to be defined manually.

only one hook, situated in the mass center of the object. It's natural to think that the grasp focuses on the mass center of the object in this case. Things change when we want to locate the hooks of bigger but still graspable objects (a suitcase) or when we deal with particular objects that should be grasped in different ways (a sword can't be grasped by its blade). Identifying possible positions for the hooks in such objects is relatively easy if done manually: it's enough to answer to the question: where would the hand focus its grasp? Examples of hooks are shown in Figure 5.4.

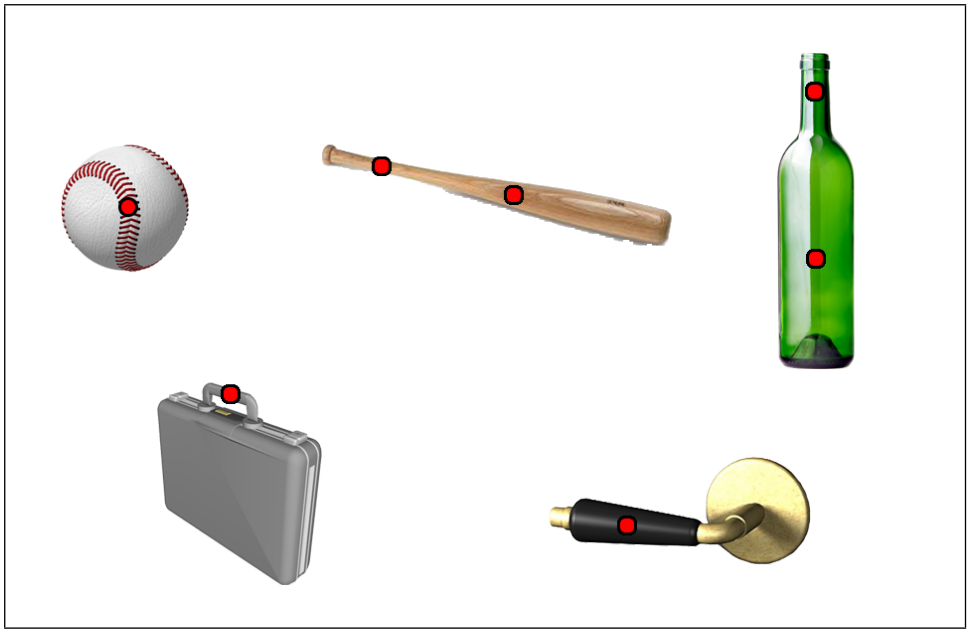

Figure 5.4: A few examples of hook placement. Red dots represents the hooks. There might be several hooks for a single object.

5.3 Grasp Movements

The fact that an infinite number of grasp poses exist is unquestionable. However, in the human brain there isn't a storage of all the possible grasp poses suitable to grasp all the known objects. Even if with some questionable exceptions, human brain computes the grasp pose every

time the person needs to grasp an object. What is really stored in the brain is a set of muscular movements that doesn't depend on the target object. As an example, just think of the grasping actions for a tennis ball and for a rubik cube: the muscular movement of each finger is the same, but resulting grasp poses are different for the two objects (see Section 3.1.2 for a definition of grasp pose). Therefore what we really want to store is a set of grasp movements, without taking into account the shape of the eventual grasped object. Clearly different objects may require different movements for the grasp, but the number of movements is drastically reduced if we only consider the muscular movements of the hand.

In Section 2.2.1 we illustrated a grasp movement classification described by Cutkosky and Howe [20]. Taking that classification as a reference, it's possible for instance to store only information about those sixteen movements and use it as a base to compute the final output, choosing the best movement suitable for the target object. For our purposes we wanted to test the algorithm at least with two possible grasp movements. We chose the most commonly used ones (according to us): the sphere power grasp and the thumb-index precision grasp.

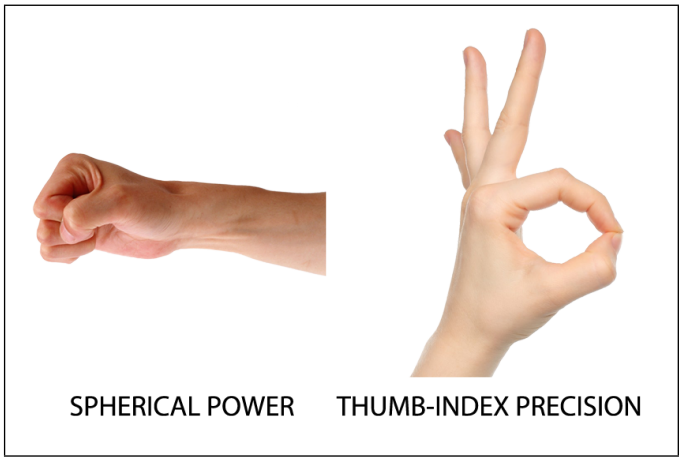

Figure 5.5: Among the grasps described by the classification we chose the spherical power grasp and the thumb-index precision.

5.3.1 Movement Information Storage

Storing information about a grasp movement is a relatively simple task and it can be achieved in several ways of which we describe the most common:

- **Animations:** a grasp movement can be simply stored using animations. They are usually created by a 3D model designer. This solution is not particularly recommended in our case because of two reasons mostly: fingers perform very simple movements (a big effort is not required to generate them at runtime); phalanges need to be stopped when colliding with the grasped object (usually animations shouldn't depend on collisions).

- **Interpolation Storage:** it's possible to store all the various interpolations that compose grasp movements. Although this is a rough method, it allows to easily manage the the phalanges stop in presence of collisions. A drawback is that the number of interpolations for each movement should be decided from the start. Moreover, generating many interpolations for each movement requires maybe a too long time and preferably it's done by a 3D modeler designer.

- **Initial and Final Poses Storage:** if the graphic engine allows it, instead of storing all the interpolations that compose a grasp movement, it's possible to store only the initial one and the final one and let the engine computes the remaining ones. Using this technique it's possible to vary the number of interpolations for each movement at any time. Also, it doesn't require a long time and the objective can be reached without the help of a 3D model designer.

Among those methods, we chose the third one, as we heavily base the algorithm on collision detection and as the number of poses we have to store is relatively small. In Figure 5.6 we show them. The starting pose can be shared by the power grasp and the precision grasp without loss of realism. Notice that in the final poses we don't worry so much about mesh penetrations because likely during all simulations fingers

will be stopped before they reach the last interpolation due to collisions with the target object.

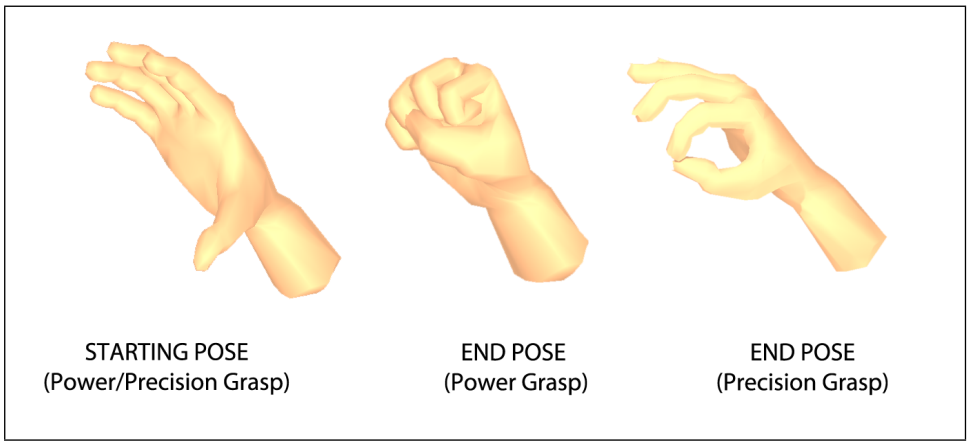

Figure 5.6: The three poses we stored in order to generate two grasp movements. The starting pose is shared between them.

5.4 Grasp Simulation

As we want to evaluate grasp poses, we need a procedure that, based on a start hand position (defined by hook rotation and grasp center translation) and on a grasp movement, generates the grasping pose taking into account the grasped object, computing the final positions and rotations of all the bones.

First of all we need to compute all the interpolations between the initial one and the final one. The number of interpolation should be allowed to vary because it can be influential for the performances but we estimated that a minimum of 10 interpolations is required in order to avoid problems. Indeed if the interpolations are not enough mesh penetration issues can arise because collisions are detected with a too noticeable delay. Once we have all the interpolations we also have the grasp movement and we can start the simulation. Notice that the simulation is never rendered and it's computed in a much shorter

time than the frame render. For this reason it's not possible to exploit the physics engine in order to move the fingers, nor any feature of the graphic engine that performs actions split among frames. The grasp simulation follows the strategy shown in Figure 5.7

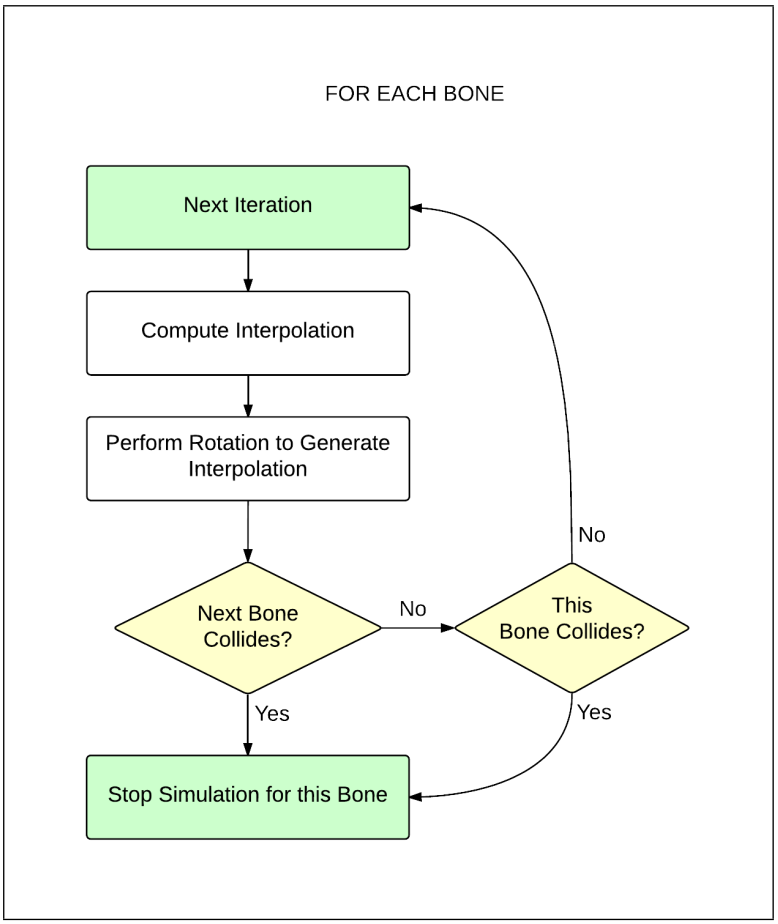

Figure 5.7: Grasp Simulation Flowchart.

Starting from the first one, for each interpolation a collision check is performed on all the bones. Clearly we can't check a whole finger at once because phalanges don't move (rotate) totally in sync: considering a single finger, if a phalanx touches the object all the previous phalanges stop rotating while the further phalanges continue the simulation. The process ends when all the bones stopped (either because of collisions or because they reached the last interpolation). In order to

realize this we need to know the predecessor and the successor of each finger and we need a way to check whether a phalanx is colliding with the target object.

At the end of the simulation we have the grasp pose corresponding to a starting hand position and a grasp movement. The pose consists of the knowledge of positions and rotations of all the bones after performing the movement. Having that knowledge it's possible to evaluate the goodness of the pose and compare it with other poses. In addition, a useful datum we get from this procedure is the interpolation reached by each finger. If this number is equal to the total number of interpolations for each bone of a finger, we know that the finger didn't collide with the object during the entire simulation.

5.5 Quality Value

The essence of an iterative optimization process is the evaluation. If a correct comparison system is designed then the algorithm will always reach a suitable output in a finite number of steps. In order to identify the parameters to use for the comparison we had to take as examples several grasp poses on the same object and try to understand why certain poses were more suitable than others. Again, designing the comparison means in some ways trying to imitate the logical reasoning performed by human brain. We can compare two grasp poses only based on numerical parameters: the simplest way to solve the comparison is defining a value called *Quality* that stands for the goodness of a pose. We arbitrarily decide that a pose is more appropriate than another if its quality value is greater (we could have defined the *Cost* instead, inverting the comparison).

For each grasp pose during the optimization process we compute its quality as a combination (weighted sum in our case) of several parameters that we deduce from the grasp simulation output. When the simulation finishes, the available information we could use to compute the parameters consists of:

- Grasp center's position and rotation
- Hook's position and rotation
- Position and rotation of each bone
- Amount of interpolations reached by each bone
- Character's position and rotation
- Target object's shape (3D mesh)

Based on this information we may compute several parameters that we can combine to build a quality value. At this point parameters are measured by simple math calculations so the actual number of parameters and their nature is easily adjustable. Depending on the type of hand, it might be necessary to modify the combination of parameters, or even the parameters themselves. In the following sections we describe the set of parameters we believe are the most suitable for the simulation of a grasping human hand.

5.5.1 Evaluation Parameters

We illustrate the parameters we found suitable in order to compare the goodness of grasp poses (notice that not all of the parameters will necessarily be part of the quality value):

- **GRASP CENTER-HOOK DISTANCE (gDist)**
  In the ideal case this value is equal to zero as grasp center and hook coincide. If the target object is small enough this happens also in practice; for this reason it's not a parameter that can be used alone otherwise several grasp poses would have the same quality. In any case this parameter is relevant because it says how close the position of the grasp center is to its position in the ideal situation. The value is quite easy to compute, as it's nothing but a distance in 3D space between two known positions.
- **AMOUNT OF INTERPOLATIONS REACHED**
  We obtain this information (for each bone) directly after the grasp simulation process. It's very useful especially because it allows to

check whether a finger touched the target object. Indeed, considering a finger, if all its bones reached all the interpolations of the grasp movement then we can conclude that the finger bones didn't collide during the entire simulation. Moreover this value states the blending level of each finger and, accordingly, the average amount of interpolations reached by all bones represents the state of progress the hand reached for the grasp. However this is not a one-way valuation: for some objects the hand performs better grasps when it reaches a higher progress in the simulation while for other objects the grasp is ideal when the hand reaches a lower progress. Because of this ambiguity we prefer to avoid this use of the value.

- **TIPS-HOOK DISTANCE (tDist)**
  Simply as the name says, this parameter is equal to the sum of the distances between all the finger tips and the hook. Those distances are relevant especially when dealing with precision grasps: it might be necessary to specify that only the tips must touch the object (think of picking a dice placed on a table) instead of letting the whole finger envelopes the object like in the case of a power grasp.

- **FIRST CONTACT PHALANGES-HOOK DISTANCE (fDist)**
  We found out that computing distances between the tips and the hook was not enough to evaluate a grasp because in several situations the hand uses different phalanges to envelope an object. We noticed that for each finger in most cases the phalanx which really performs the grasp (the one that holds the target object) is the first phalanx to come in contact with the target object. For this reason we thought that a suitable parameter to analyze would be the sum of all the distances between the phalanges which first collide with the object and the hook (considering one finger at the time). Considering one finger, the distance is taken from the end joint of the phalanx, as shown in Figure 5.8.

  This parameter is especially used for enveloping grasps like the spherical power grasp. It might be used for precision grasps as

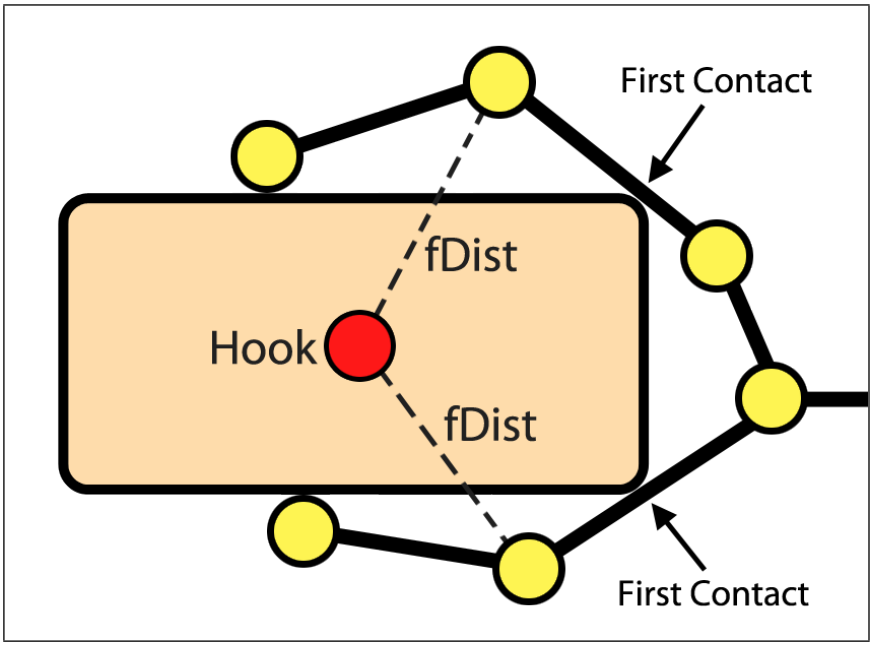

Figure 5.8: A fDist representation: for each finger, considering the first colliding bone, the distance between the end of the bone and the hook is taken. Such distances are then summed up.

well, but if the first colliding phalanges are necessarily finger tips, then some incorrect evaluations can be done. In those cases it's correct to use directly the tips-hook distance parameter.

- **WRIST-CHARACTER DISTANCE (wDist)**
  It's logical to think that the character would choose among the various suitable grasps the one which requires less effort to perform. A way to reduce the physical effort is reducing the distance between the actual position of the hand and its target position. Considering the grasp pose, an estimation of the physical effort could be represented by the distance between the wrist (in the simulated grasp) and the character (more correctly its shoulder): minimizing this distance would mean also reducing the physical effort needed by the character to perform that grasp action. We're aware of the fact that this is not the most correct approximation that describes how the human brain automatically thinks of reducing physical efforts, but we discovered it to be valid enough for our purposes.

5.5.2 Quality as a Weighted Sum

As mentioned before, we decide to define the quality value of a grasp pose as a weighted sum of sub-values, taken from the set of parameters just described. The combination might be slightly different for the various grasp movements and for different hand structures. In the case of human hand and spherical power grasp movement, we choose the following combination of parameters:

$$Quality_{(Power)} = -k_1 * gDist - k_2 * fDist - k_3 * wDist \qquad (5.1)$$

Terms are rightly negative because the distances must be minimized in order to get a suitable quality value. The k-terms are weights and their values may be set arbitrarily. The quality value for the thumb-index precision grasp is:

$$Quality_{(Precision)} = -w_1 * gDist - w_2 * tDist^* - w_3 * wDist \qquad (5.2)$$

where the w-terms are the weights and $tDist^*$ is the sum of the distances between the finger tips and the hook considering only the thumb

and the index. Weight values are very influential in determining the final quality. A series of tests is required in order to find the proper weights: in 3D space the order of distances may change due to different units of measure and scaling factors. If the weights are wrongly regulated some problems on the evaluation may arise: if one of the terms prevails on the others it will be in practice the only one considered by the algorithm. All the terms should belong to the same order. In Chapter 6 we illustrate the values we found out to be suitable for our scenario both in case of spherical power grasp and in case of thumb-index precision grasp.

5.6 Optimization Step

Once the evaluation system has been set, we only need a procedure that chooses and evaluates the possible grasp poses. We already have a positioning system that allows us to move the hand around the object while the palm is always pointing towards it (see Section 5.2). In this system, moving the hand is done by changing the rotation angles of the hook. The optimization step follows the schema in Figure 5.9: starting from a position, each step consists of seven grasp simulations and seven evaluations accordingly. The current pose is evaluated, then the hook is rotated in the six possible directions (positive and negative rotation around x-axis, y-axis and z-axis) and for each of the six poses generated another evaluation is performed. The poses obtained after the simulations are associated with a quality value; before the next step the hand will move to the pose with greater quality. If the current pose has the maximum quality value then the optimization ends and the current pose is the output of the algorithm.

This method is a slight variation of *Depth First Search* applied on an endless tree. Indeed hand poses can be represented as nodes of a tree. Actually, if we choose an integer angle variation for each step, poses aren't infinite but considering that the search can go over the same pose more than once we can say that the poses tree is endless. As the hand moves, lower nodes of the tree are reached. Clearly, computing all the

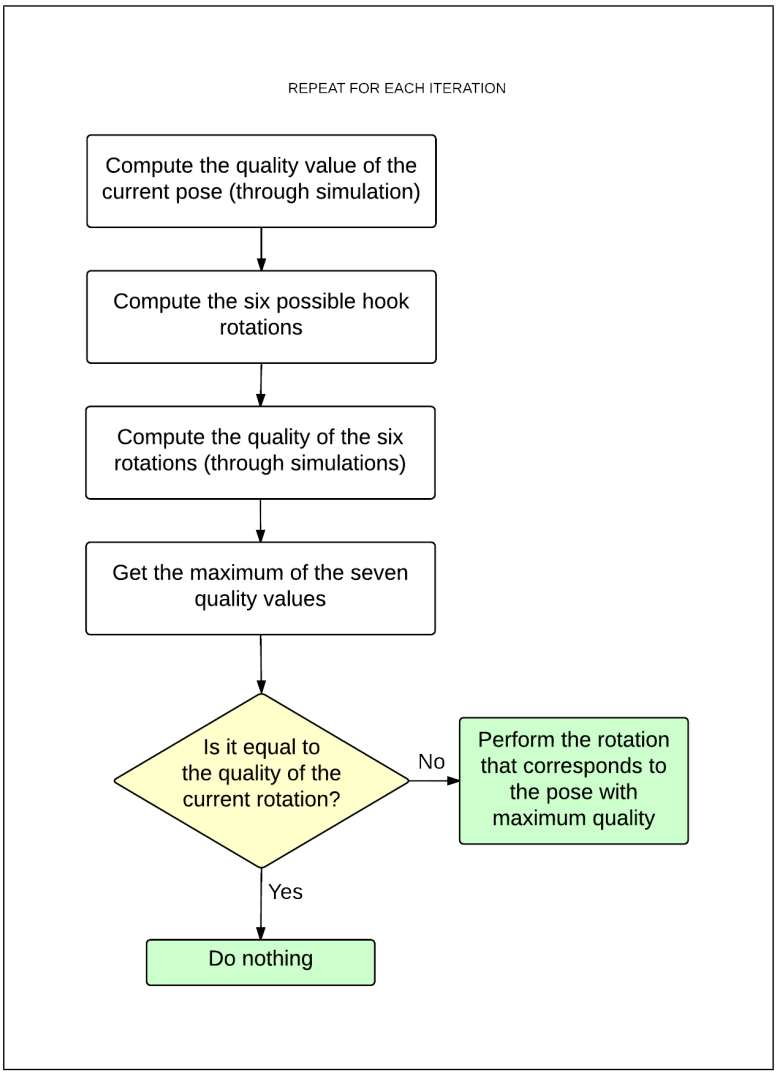

Figure 5.9: Optimization step flowchart. It's preferable that one iteration is executed for each frame rendering, but this is not always the case.

values of the entire ramification of the tree is not necessary: knowing the values of the ramifications directly starting from the current node is enough. The search ends when the value contained in the current node is greater than all the values contained in its direct ramifications. One of the known problem of a depth first search is that the found solution is not necessarily optimal if the tree has no constraints on the values of its nodes. We illustrate two ways of working around this issue in Section 5.6.2. An example of search tree is shown in Figure 5.10: the values contained in the nodes are purely symbolic; they represent the quality value computed for the grasp pose corresponding to the node. A transition from one node to another represents the rotation (either positive or negative) around one of the three axes.

Finishing the whole computation in one time instant or frame rendering time slot might be too expensive in terms of computational effort. The first logical solution to this issue is performing a small number of steps for each time instant, splitting the computation. This is easy to achieve and the only limit is that at least one step must be completed in one time slot. If the computation of a single optimization step in one time instant affects too much application performances, it's possible to perform a similar type of optimization steps, even if this will reduce the efficiency: instead of performing seven simulations, it's possible to perform only three of them for each frame (time instant). If this is the case, the set of evaluated poses includes the current one and the ones generated from a rotation in one direction (positive and negative). We call this step variation a *unidirectional step*. It's enough to switch the direction at every step in order to be sure that all the possible grasps are taken into account. Using this kind of step reduces the efficiency because the search is performed in one direction only (it changes at every step, but the search power is lower anyway).

A noticeable fact is that, if the computation is split among several frames or time instants, parameters may change due to the transformations happening in the 3D environment. For instance, the *wDist* parameter is related to the position of the character which can move

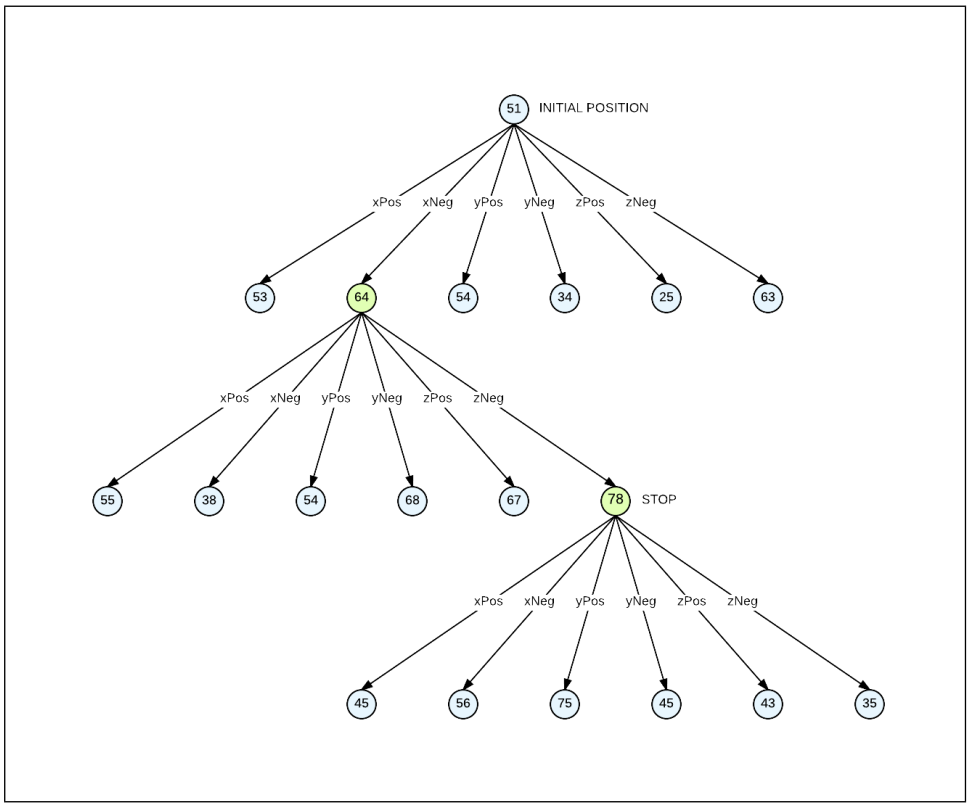

Figure 5.10: Iterative optimization as a depth first search. Values inside the nodes represents the quality values of the computed positions. Notice that the tree is endless because of the variable nature of the quality value. In other words, after each iteration branches of the tree may change.

around during the execution. This produces a even greater number of possible tree configurations but it doesn't affect much the performances because each transformation in the environment requires several time instants or frames to be performed: this allows the algorithm to exploit enough computational resources to tolerate small changes in the tree.

5.6.1 Starting Hand Position

An iterative optimization process must start from an initial state in which the first optimization step takes place. In our case, the initial state consists of a hand position (described by the hook rotation). In other words, we needed to define a system that chooses the initial hand position based on the position of the character and on the target object. We analyzed the way a person approaches an object when trying to grasp it and we found out that in the majority of cases the hand advances to the object from the top, with the palm pointing towards it. Therefore we designed a function that, based on the character's shoulder position and on the hook's position (belonging to the target object), computes the hook rotation necessary to obtain a position of the hand with the fingers pointing in the direction shoulder-hook and with the palm directed to the hook itself. The iterative optimization begins by evaluating this position and the six positions around it, according to the optimization step system described before. A few examples of initial hand positions are shown in Figure 5.11.

5.6.2 Local Maxima Problem

We already pointed out that an iterative optimization process performed on the kind of tree we have doesn't necessarily lead to an optimal solution. As we explain in detail in Chapter 6 a relevant problem arises: quality values almost never follow a continuous trend. In other words, the optimization may get stuck in a position with a quality value that is suitable if we consider the very close range of rotations around it but that is far from being the best solution among all rotations; this value represents a local maximum in the virtual function built from the quality values, varying the rotation angles of the hook. This is mostly due to irregularity of the shape of target objects which

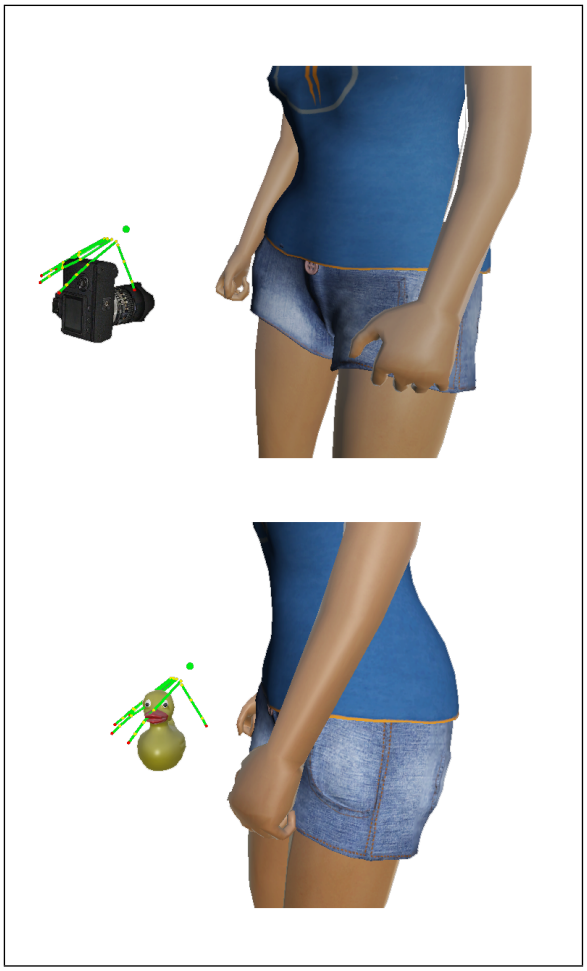

Figure 5.11: The initial position depends on the character location with respect to the target object. At the beginning of the process, the clone hand is set to assume the starting pose of the chosen grasp movement. In our case the starting pose is the same for all the movements we generate.

are approached from different directions, each time producing a different quality function. Indeed, only a perfectly spherical object would produce a monotone quality function having only one maximum that represents the best solution. We developed two conceptually different solution to the local maxima problem: one more logical and the other more coherent with resources availability.

5.6.2.1 Logical Solution: Initial Approximation

The first solution we found is quite straightforward: we try to guess the range of rotations where the optimal solution could be located, we place the hand into that range and finally we apply the iterative optimization in order to get the hand to the optimal position. The initial guess must take into account the entire range of possible positions and it must estimate the quality without performing any simulation (otherwise it would be too expensive and it wouldn't be different from the optimization step). Our intention is to subdivide the range of possible positions in several sample hook rotations that will be analyzed. This is easily done by taking only some of the possible angle values for each rotation axis, equally distributed between 0° and 360°. Quality estimation is performed for each combination of sample rotations around x-axis, y-axis and z-axis. The number of combinations might be great so the estimation method must be quite fast. We thought of a simple enough solution that gives a correct estimation of the range where to perform the optimization: for each combination of sample rotations the hand is translated along y-axis until it's free from any collisions; after that, the distance between grasp center and hook ($gDist$) is taken as estimation of the quality. This strategy works fine in the great majority of situations. Notice that the simulation is not performed, so the hand bones movement progress is at the initial interpolation of the grasp movement and the collision check is made on this hand configuration. It's possible to reduce the possible combinations of hook rotations by eliminating from the comparison all those rotations that corresponds to unnatural hand positions (the ones which don't fulfill hand constraints). This solution is conceptually correct but, because of the great number of positions that have to be analyzed, it's not

particularly suitable if the algorithm runs in real-time.

5.6.2.2 Practical Solution: Random Angle Search

One more suitable technique that allows us to work around the problem of local maxima and still to perform in real-time is the random angle search. While executing the optimization step, instead of choosing a fixed basic rotation angle (positive and negative) to rotate the hook around its three axes, we let the system chooses a random angle (between 0° and 180°). In this way, even if it takes a longer time, we're able to explore the total range of possible hook rotations. The random angle should let the hand tries positions that are far from each other. Bigger angles allow to try out different parts of the whole combination range while smaller angles will work as expected in order to find the best combination in the short range. Although a random search could seem something very far from the imitation of human brain line of reasoning, it turned out to be quite effective because of the nature of the problem. Together with the time-saving effect, we obtain another benefit from this method: a random number is not necessarily an integer, so we can analyze an infinite number of possible hand positions; search power doesn't change, but search range increases.

5.7 Algorithm Outline

Finally we summarize what we explained in this chapter and we show the whole algorithm's flowchart (Figure 5.12). We can identify two main phases:

1. The hook is chosen and the connection between the hook and the hand is set. Based on the position of the character, a starting rotation for the hook is computed and applied. In case of initial approximation (see Section 5.6.2.1) the initial rotation is computed by checking the estimated quality between a set of possible rotations equally distributed in the whole rotation range and the one with the greatest estimated value is chosen as initial rotation.

2. The iterative optimization takes place, starting from the initial

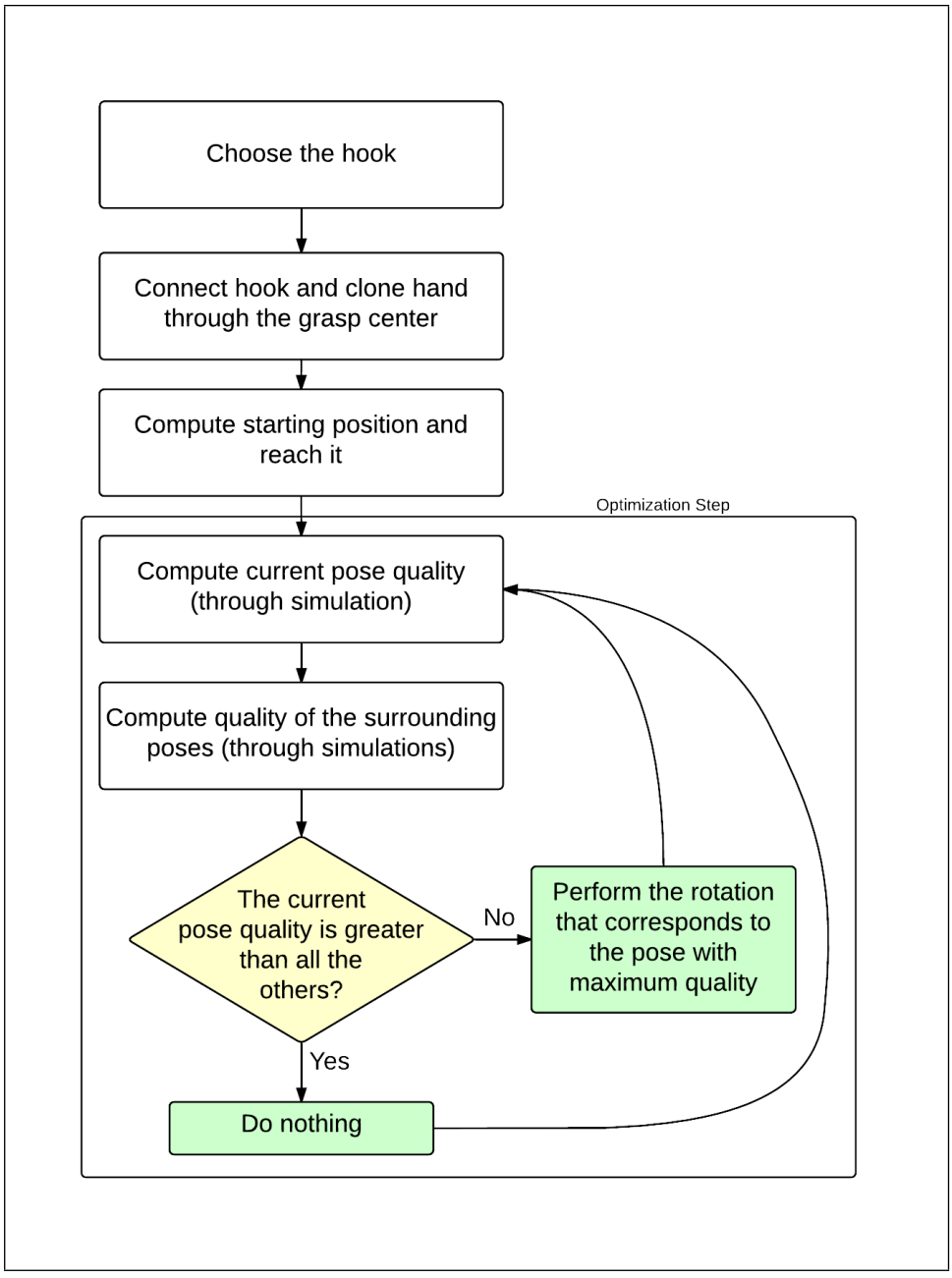

Figure 5.12: Algorithm Flowchart.

hook rotation computed in phase 1. Each optimization step consists of a simulation stage in which a few grasps simulation are performed and an evaluation stage in which the output poses computed during the first stage are analyzed and compared. The iteration ends when, of all the poses obtained from the simulations, the one with the highest quality value corresponds to the actual position of the hand. As explained in Section 5.6 optimization one can split the optimization steps in several ways (for clarity we only use the term *time instant* but the computation can also be split among frames):

- All the iterations in one time instant
- One iteration in one time instant
- One unidirectional iteration in one time instant

Chapter 6

Implementation and Performances

We decided to implement and test the algorithm on Unity 5 Personal Edition, one of the most popular between the currently available free game engines. Code is written in C# language, supported by Unity. The 3D models we used are in part taken from the web and in part realized with Blender 2.7 while we generated the human character's model and skeleton using MakeHuman software. After a brief description of the Unity features we exploited, we illustrate the 3D scenario which is the test-field of the algorithm and the developing of the algorithm itself, pointing out the practical issues that were not addressed in the design process. Finally, we show the results of tests from which we deduce the performance level of the algorithm.

6.1 Unity 3D Environment

In Unity the *object* is the basic entity. An empty object consists of a point in 3D space with coordinates x, y and z called *pivot*. A *scene* is the 3D space where objects are placed and where they interact with each other. Every object has a *parent* field (it's null by default); this allows the generation of hierarchies of objects. Unity engine is object-oriented in the sense that objects behave independently from each other and they interact only when they're supposed to. Developers can attach several *components* to an object. Components define the features of

objects; a list of the components we exploited most is:

- **TRANSFORM:**
  Every object has a transform component by default. It determines its actual position, rotation and scale in the 3D space of the scene. Values can be modified both manually by the developer and by the physics engine (for instance if a force is applied on the object). The transform component is used to apply basic transformations to objects (translation, rotation, scaling) with respect to their pivot point. If a transformation is applied on an object with children, it's automatically applied also on all the children (all the objects under it in the hierarchy) with respect to the pivot of the parent object.

- **BEHAVIOR SCRIPT:**
  One or more user-written scripts can be attached to an object, defining its behavior. With these scripts the developer can manage interactions between objects, perform computations and access other components at runtime.

- **RIGIDBODY:**
  This component defines the physics-related properties of an object. Although we don't use the physics engine to compute grasp poses, we need this component in order to allow the system to check collisions. Plus, when target objects are not grabbed by the character, they're subject to gravity, so they need to be taken into account by physics engine.

- **COLLIDER:** It's the container that allows collision detection with other objects (other colliders). Usually the shape of a collider is much simpler than the shape of the object it's attached to, due to the fact that for the physics engine checking collisions between primitive 3D shapes (or simpler shapes anyway) is much less expensive in terms of computational effort than checking complex shape collisions, but it's still possible to generate a mesh collider based on an object's mesh.

- **ANIMATOR:** The animator component manages the pre-defined

animations of an object. Animating an object requires a controller which chooses the animations and decides when they should be activated and an avatar which is the skeleton (armature) of an object (see sub-section 6.1.1).

- **MESH FILTER:** It defines the mesh of an object. Primitive shape meshes can be created in Unity but usually object meshes are realized with a 3D modeling software and then imported.
- **MESH RENDERER:** If the object has a mesh filter component the mesh rendering takes care of rendering that mesh attaching all the graphic features like textures, visual effects and others.

6.1.1 The Avatar

The skeleton of an object is called *avatar* in Unity. Not every object has an avatar, only those which are animated with pre-defined animations have one. When it's the case, the mesh of the object distorts and blends according to the movements of the avatar (weights of mesh vertexes are set by the 3D model designer). As we need to act on hand bones, we need to know how to define an avatar and how to spare a part of it (the hand of the character) from the animation system. We focus on the avatar of a human hand. The default configuration of hand bones in Unity doesn't correspond to the one we wanted to use: metacarpal bones are not taken into account when animating the human hand. This is not a big problem as it's possible to import more complex avatars in any case and as our algorithm doesn't deal with with pre-defined animations.

Rotation and translation values determined by pre-defined animations overwrite the current ones before each frame rendering. In order to superimpose different values it's enough to modify them between the moment in which the animator changes them and the actual frame rendering.

6.1.2 Coroutines

In Unity, computation is frame-based: some functions are called during each frame, like the *Update()* function while some others are called only

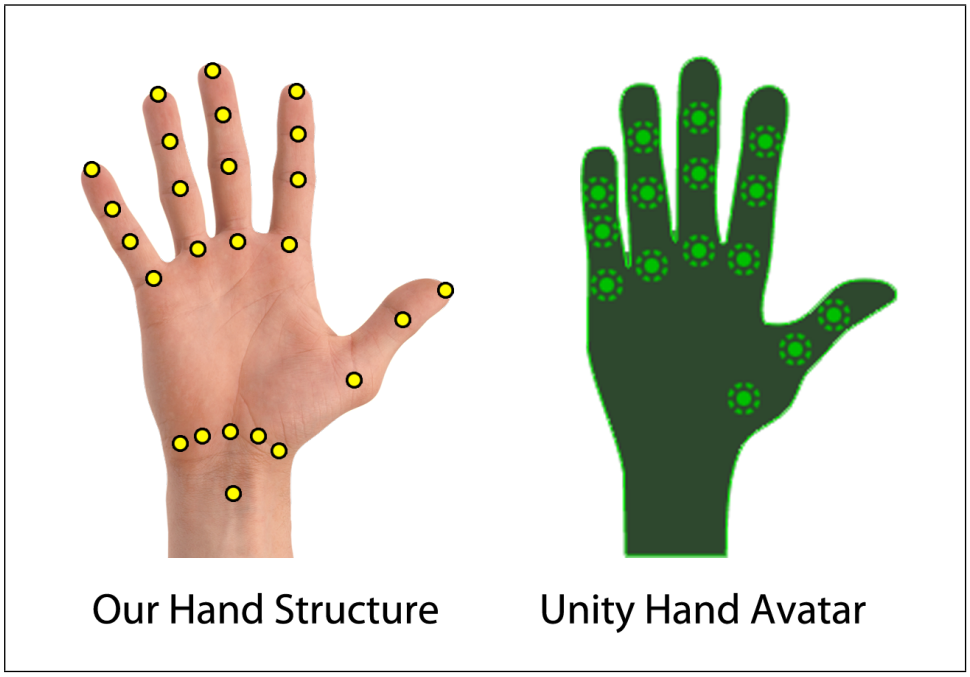

Figure 6.1: The hand avatar in Unity is less complex than our skeleton model: CM joints are not included and this means that rotations of metacarpal bones are always null. Moreover Unity doesn't include fingertips in the avatar (the correct choice for any animation system) so we need to add them manually if we deal with models designed to be imported in Unity. Wrist joint is present in Unity avatar but it's not shown on the hand as it's considered to be part of the arm.

in certain moments or when other functions need them, but always within the time of a frame rendering. If a function needs a few seconds to complete its computation the game stops for a few seconds because the next frame is rendered after all functions finish their computations. Our algorithm needs to perform several computations which would take more than a frame rendering interval. The immediate solution would be to use a multi-threading technique in order to move the computation in a parallel time line. However this is not possible, as Unity game engine is not thread-safe: we can't act on object in the scene if we're out of the main thread. Indeed we need to manipulate the transform component of the bones. Unity provides a solution which we found out to be really convenient in order to split computation in time: *Coroutines*. These are particular functions that can stop the computation in a certain point, defined by the developer. In the next frame slot, the computation starts from the point where it stopped previously. This kind of functions is particularly useful in order to split code cycles among several frames (code cycles are exactly the structures we need in our algorithm).

6.1.3 Quaternions

Rotations of an object can be stored in a vector of three float values (rotation around x-axis, y-axis an z-axis) which intuitively allows to get an immediate sense of the rotation. However Quaternions can store rotations as well. Quaternions are extremely efficient when working with interpolations, so we use them when we need to interpolate between the starting pose and the final pose of a grasp movement (see Section 6.4).

6.2 Scenario

We created a 3D scenario for our test. It consists of a large living room and some furniture with some graspable objects placed on. A human character is able to walk around the room, to grasp the objects and to drop them. Physics engine manages graspable objects bodies only when they're not in the character's hand, while it never considers the character. This scenario might represent a small portion of a video

game in which the algorithm could be used.

Figure 6.2: The 3D scenario created with Unity. It's a living room filled with both graspable and ungraspable objects.

6.2.1 Human Character

We used MakeHuman software to build a realistic human character and its skeleton. In MakeHuman the shape of the character is customizable as well as its clothes. The skeleton can be chosen from a list of 6 types. The one which fits our purposes has the same structure Maya3D software uses for humanoid characters. Although the number of bones is higher than the one supported by Unity avatar system and the bone placement slightly differs from the Unity default one, the hand structure turns out to be exactly the one we planned to use. When imported in Unity, the skeleton is turned in a series of empty objects placed where the bones start. They're linked by parent-child relations: the first spine bone is the parent of all the other bones; from that, chains of bones are naturally set to be children and children of children.

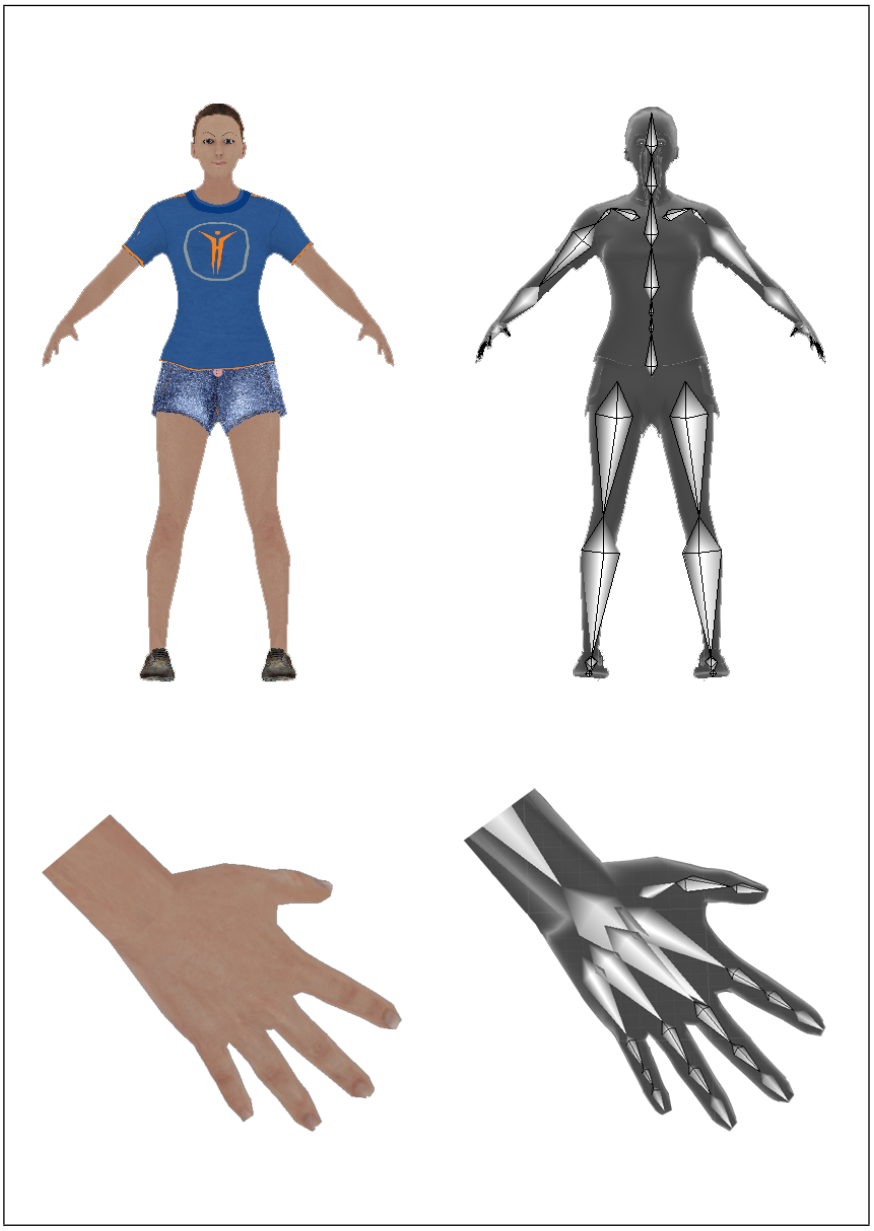

Figure 6.3: Character is made with MakeHuman and imported in Unity after some adjustments done with Blender. One of the skeletons provided in MakeHuman includes a hand structure that corresponds to the one we decided to use.

6.2.2 Graspable Objects

For the algorithm tests, several small objects with different shapes have been placed in the scenario. Each object needs one or more hooks in order for the algorithm to work with them. We decided that the developer should define the hooks as empty child objects only when the graspable object has more than one hook or when its hook does not coincide with its pivot point. In case hooks are not defined, the algorithm automatically takes as hook the pivot point of the target object. For instance if the pivot point of a tennis ball is its mass center position, then the tennis ball needs no hook definition, while a suitcase (which usually doesn't have its pivot point on its handle) needs it if we want the character to try grasping its handle.

6.3 Algorithm Setup

The algorithm works directly on the objects components, it's not pure computation, so it needs to know what objects it can manipulate and at the same time the character needs information about the algorithm's output in order to move the hand in the right grasp position. A series of setup operations will allow the algorithm to work on the defined models. Before the description of setup operations, we need to illustrate the main entities in game together with a brief explanation of their behaviors.

6.3.1 Main Entities and Behaviors

- **CHARACTER:**
  The human character is a container (a parent) for most of the important objects in game: its children are the character mesh, its avatar (composed by several objects corresponding to the bones) and, as we'll describe, also the grasp controller and all grasp centers. Its behavior (script) defines the character walking and blending movement through the animator component and the inverse kinematic system that allows the hand to reach a target pose.

- **RIGHT HAND (WRIST):**

It's the empty object corresponding to the right wrist bone and it's the hand skeleton root. The algorithm output defines the position for all the children object of the wrist and for the wrist itself. The rest of the body is managed by the character animator and by the inverse kinematic system. The wrist should always be attached to the arm. The behavior of the wrist and all its children defines the pose of the whole hand (rotations of all the bones).

- **GRASP CONTROLLER:**
  The grasp controller is an empty object that manages all the phases of the algorithm. It decides when the optimization steps are executed and when the character performs the actual grasp movement.

- **HOOK CENTER:**
  It's an empty object that is placed in the current hook. While the algorithm is working, the current grasp center is made a child of the hook center. The grasp controller manages hook center's translation and rotation. The hook center also represents the pivot point of the hand rotation.

- **GRASP CENTERS:**
  There is one grasp center for each grasp movement. Grasp centers are wrist's children but when the algorithm computes the optimization instead the wrist is a child of the grasp center (see Chapter 5) that corresponds to the chosen grasp movement. The grasp controller manages the translation and the rotation of the grasp center involved in the process.

- **GRASPABLE OBJECTS:**
  Graspable objects possess a mesh and a collider. The collider may have a primitive shape but, unless the complexity of the object is very low, it has the same shape as the object's mesh (mesh collider). Even if the object has a concave mesh (majority of cases) the mesh collider is set to have a convex shape because the physics engine can't easily manage concave meshes collisions but when the algorithm starts a grasp simulation the collider is set to have the natural concave mesh of the object.

6.3.2 Setup Operations

- **CLONING HAND SKELETON:**
  The algorithm needs to manipulate the hand for the grasp simulation and evaluation of the optimization steps but this should happen among each frame and it's not possible to operate with the original hand as it must remain attached to the character. The straightforward solution to this issue is to duplicate the hand skeleton starting from the wrist: in this way an invisible clone hand is used to search for a pose while the original one remains on the character's arm. The algorithm core will only work on the clone hand.

- **DEFINING BONES CONNECTIONS:**
  Even if a phalanx only has at most one parent and one child, we need to define the relations in the behavior scripts so that each bone knows exactly which are its child and its parent. The only exception is the wrist: it has no parent (the forearm is not considered) and five children which are the finger roots. This knowledge is important especially in the collision detection process.

- **STORING GRASP MOVEMENTS:**
  Grasp movements are stored using only two poses each (the initial one and the final one). We decided to implement two movements: spherical power grasp and thumb-index precision grasp. We set the hand in the pose we wanted to store and we saved the rotation values of all the bones as variables. Each bone of the clone hand knows its rotation for each pose. We stored three poses in this way (the starting pose is shared by both movements):

  - Starting pose
  - Spherical power grasp end pose
  - Thumb-index precision grasp end pose

6.4 Grasp Simulation

In Section 5.4 we described the simulation process: a certain number of interpolation between the two poses (initial and final) is computed and for each interpolation collisions between bones and the target object are checked. If a bone collides it doesn't advance in the simulation together with all its parents in the bone chain. In order to compute interpolation we use the *Quaternion.Lerp(...)* function provided by Unity.

> ***rotOut = Quaternion.Lerp(rot1, rot2, f)***
> This function takes two rotations as input (Quaternion class) and a float number f which has a value included between 0 and 1. The function's output is a rotation *rotOut* which is the interpolation between rot1 and rot2 at f percentage. If f is equal to 0 the output is *rot1* while if f is equal to 1 the output is *rot2*.

We show the conceptual code of the grasp simulation:

Algorithm 1 Grasp Simulation A

```
1:  procedure SimulateMovement
2:      mov = GRASP MOVEMENT
3:      n = N_INTERPOLATIONS;
4:      i = 1;
5:      while i <= n do
6:          f = i/n;
7:          for all bones do
8:              if bone.isColliding = false then
9:                  bone.rot = Quaterion.Lerp(mov.startRot, mov.endRot, f);
10:                 if bone.checkCollision() then
11:                     bone.SetIsCollidingTrue();
12:         i = i + 1;
```

We define an integer number $N_{INTERPOLATIONS}$ and we set up a cycle that repeats $N_{INTERPOLATIONS}$ times. For each interpolation reached

an f value is computed as:

$$f = \frac{N_{REACHED}}{N_{INTERPOLATIONS}}$$

and the rotation of each bone is updated using the *Quaternion.Lerp(...) function* (each bone knows its initial and final rotations). After the update, the collision check is performed on each bone and if a bone collides with the target object a boolean *isColliding* is set to be true for it and for all its parents. Clearly, the interpolation update is performed only on the bones with *isColliding* set to false. At the end of the cycle we obtain that all the bones are stable and a grasp pose is generated.

Algorithm 2 Grasp Simulation B

```
1: procedure SetIsCollidingTrue
2:     isColliding = true;
3:     if parent != null then
4:         parent.SetIsCollidingTrue();
```

6.4.1 Collision Check

We can't use a mesh collider with the same shape of the hand because we need to know which bone is colliding exactly on each interpolation so we check the collisions for each bone singularly. It's much convenient to use primitive shaped collider because it improves the performances. We found two ways to do that: using sphere colliders and using capsule colliders. Collision is checked between one bone and any other object. We know the starting point of the bone and we can get the starting point of the next bone in the chain. We could distribute a few sphere colliders among the two points in order to cover all the bone. Instead, we choose to use a capsule collider as Unity provide the useful function *checkCapsule(point1, point2, radius)* which checks whether the capsule with radius *radius* and with centers of the end hemispheres in *point1* and *point2* collides with anything in the scene. If it collides the functions returns true, false otherwise. Notice that capsules of connected

bones penetrates each other but the check is performed singularly, so this is not an issue. In order to use this system we need to add an empty object for each finger that represents the tip of the finger itself; it must be a child of the last phalanx. In this way even for the last bone of a finger we can correctly get the ending point of the capsule collider.

6.5 Real-Time Grasp Search

Having a good grasp simulation system allows us to set up the grasp search as an iterative optimization performed in real-time. In this section we illustrate in detail all the core points of the algorithm we implemented. We also discuss the problems that arise in practice and that the previous chapter didn't examine in depth. First of all, we describe a function we implemented that will be mentioned in the next subsections: the *TranslateToFree()* function.

6.5.1 *TranslateToFree()* Function

In order to fully understand the purpose of the function, it's necessary to have a clear notion of our hand positioning system described in Section 5.2: the hook center is the main pivot point to which all the rotations are applied. The grasp center follows those rotations around the same pivot point and so does the hand. In this way the hand is able to move around the target object while always focusing on it. If we don't consider collisions (as we did so far) the grasp center coincides with the hook center but in this case it may happen that the hand mesh penetrates the target object's one. The *TranslateToFree()* functions solves this penetration (only if needed) by translating the grasp center and the hand together with it along the local y-axis of the hook center (the same as the grasp center). This must be done by steps: for each step a collision check is performed (see previous section) and if any bone collides the hand is translated away from the object along the y-axis. The magnitude of the translation may be set by the developer. A too small magnitude would afflict performances because of the great amount of collision checks to be performed while a too big magnitude

would let the hand translate too much away from the object, generating a wrong pose.

Algorithm 3 Translate to Free

```
1: procedure TranslateToFree
2:     vStep = MAGNITUDE;
3:     isFree = false;
4:     if hand.collisionCeck() = false then
5:         isFree = true;
6:     while isFree = false do
7:         hand.TranslateY(vStep);
8:         if hand.collisionCheck() = false then
9:             isFree = true;
```

6.5.2 Initial Phase

Before the actual optimization starts, we need to perform a few operations. We can also compute a starting pose for the hand without afflicting too much the performances. This is not strictly necessary but it helps to find a good output pose in a shorter time. Before applying the optimization steps, the algorithm needs to:

1. Get the hooks of the target object and take the closest.
2. Move the hook center such that it coincides with the closest hook.
3. Make the clone hand a child of the grasp center that corresponds to the chosen movement.
4. Move the grasp center in the same position as the hook center.
5. Rotate the hook center such that its axes coincide with the grasp center's ones.
6. Make the grasp center a child of the hook center.

Those operations set up the hand positioning system: applying simple rotations on the hook center we're able to move the clone hand around

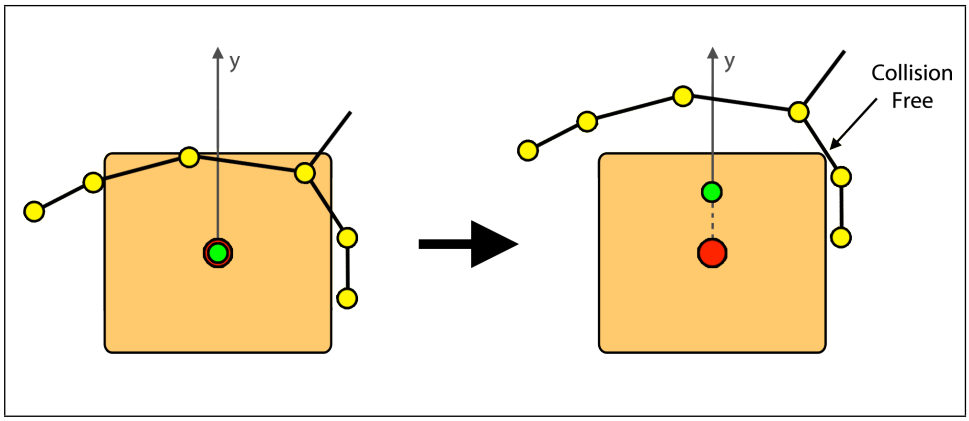

Figure 6.4: After choosing and performing a hook rotation, it may happen that the hand collides with the object already in the starting pose. The *TranslateToFree()* function translates the grasp center (green dot) along the y-axis of the hook (red dot) until the hand is free from any collision. Notice that the hand translates together with the grasp center so we don't need to perform any transformation on the hand.

the target object while making its palm always points towards it.

In Section 5.6.1 we illustrated a good example of system to compute an initial pose from where to start the optimization. Provided that the z-axis of the grasp center corresponds to the one of the hand's finger direction (excluding the thumb) we can easily compute the pose starting from the hook center position and the character right shoulder position. We compute the direction:

$$d = hookCenterPos - shoulderPos$$

and then we rotate the hook center such that its z-axis points towards *d*. This is easily achieved with the *Transform.LookAt(Direction d)* function provided by Unity. Notice that if the set up operation are performed correctly the z-axis of the hook center should always correspond to the z-axis of the grasp center.

6.5.3 Optimization Step with Random Angle

Once the hand positioning system is set up and the clone hand has an initial pose (it might be the one we computed in the previous subsection) we can finally start to write the iterative optimization described in the previous chapter. We don't use the logical solution illustrated in Section 5.6.2.1, instead we adopt the practical solution of Section 5.6.2.2 mostly because it's the one that allows a real-time execution. The computation takes place inside a particular function defined by Unity in each behavior script: the *Update()* function.

Update()
This function is called between each frame rendering. It's possible to allow certain actions to be executed before or after this function in the same frame interval. For instance, animations are computed by the animator component after the *Update()* function.

The strategy is to execute a step for every frame rendering. We need to use a coroutine (see Section 6.1.2) that suspends the execution just after a step is completed and that resumes it after the next frame is

rendered. For each iteration, before the actual optimization step we need to make the target object ready to be working on: it must be free from external forces and its collider must not be convex, but concave. Unity defines as *kinematic* a body that is not affected by external forces. After the step is completed, the hook center's rotation represents the current state of the optimization. For the reasons described in Section 5.6.2.2 we compute a random angle between 0°and 180°for each iteration and we pass it as a parameter to the *MakeStep()* function which will be defined later. If we suppose that every step can be the last one, we have to set the correct pose for the hand relative to the hook center's rotation. We do this with a combination of the functions *TranslateToFree()* and *hand.SimulateMovement()* seen before. At the end of each iteration the clone hand assumes a correct grasp pose and after every additional iteration the pose improves. The coroutine never ends the computation unless the developer decides so. A conceptual iteration is shown in Algorithm 4.

Algorithm 4 Iteration

```
1:  procedure ITERATION
2:      angle = RANDOM(0, 360);
3:
4:      object.isKinematic = true;
5:      object.convexCollider = false;
6:      makeStep(angle);
7:      object.isKinematic = false;
8:      object.convexCollider = true;
9:
10:     TranslateToFree();
11:     hand.SimulateMovement();
12:     suspend until the next frame
```

Algorithm 5 shows the actual optimization step. Having an angle value we produce the grasp poses that will be compared based on their quality. Hook center can be rotated around three axes (x, y, z) so, if we consider both a positive and a negative rotation, with one angle value

we can generate up to six grasp poses in addition to the current one. This leads to a comparison between seven grasp poses. The quality of a pose is computed by the function *QualityCheck()* that will be analyzed in detail later. We set up a list of quality values and for the comparison we simply take the maximum between those values. If the maximum corresponds to the one computed for the current grasp pose, the hand doesn't move (the hook center doesn't rotate), otherwise the hand will move to the position corresponding to the grasp pose that had the greater quality among all the others.

The *QualityCheck()* function is what decides how the comparison between poses is defined, so it's up to the developer determining which parameters are to be used and how great their weights should be. After a series of tests we found a good combination of parameters that allows a good comparison between poses. The function structure shown in Algorithm 6 is quite straightforward: the values are computed exploiting some methods of the hand and then they're returned in a weighted sum. Values names and meanings are defined in Section 5.5. This may change depending on the type of application and on the chosen grasp movement. Algorithm 6 is just an example of such a function (the one we used to evaluate spherical power grasps). Notice that weights are negative because they're multiplied by distances that should be minimized; the quality increases when any of those distances decreases. In this case all resulting quality values will be negative, but this is not relevant for the comparison.

For clarity we also show how the *hand.ComputeFDist()* function works (Algorithm 7). For each bone chain the first colliding bone is obtained choosing the last (the further in the chain) between the ones that reached the smallest number of interpolations. A distance is computed between the end point of this bone and the hook center. Done that for all the fingers, the distances are returned in a sum.

Algorithm 5 Optimization Step

```
1:  procedure MakeStep(angle)
2:      qualities = new List(value, rotation);
3:      qualities.add(QualityCheck(), current);
4:
5:      hookCenter.rotate(x, +angle);
6:      qualities.add(QualityCheck(), xpos);
7:      hookCenter.rotate(x, -angle);
8:
9:      hookCenter.rotate(x, -angle);
10:     qualities.add(QualityCheck(), xneg);
11:     hookCenter.rotate(x, +angle);
12:
13:     hookCenter.rotate(y, +angle);
14:     qualities.add(QualityCheck(), ypos);
15:     hookCenter.rotate(y, -angle);
16:
17:     hookCenter.rotate(y, -angle);
18:     qualities.add(QualityCheck(), yneg);
19:     hookCenter.rotate(y, +angle);
20:
21:     hookCenter.rotate(z, +angle);
22:     qualities.add(QualityCheck(), zpos);
23:     hookCenter.rotate(z, -angle);
24:
25:     hookCenter.rotate(z, -angle);
26:     qualities.add(QualityCheck(), zneg);
27:     hookCenter.rotate(z, +angle);
28:
29:     max = qualities.Max();
30:     if (max != qualities(current)) then
31:         hookCenter.rotate(rotation corresponding to max);
```

Algorithm 6 Evaluation

```
1: procedure QualityCheck
2:     gDist = Distance(graspCenter, hookCenter);
3:     wDist = Distance(wrist, character);
4:     fDist = hand.ComputeFDist();
5:     k1 = -5;
6:     k2 = -1;
7:     k3 = -1;
8:
9:     quality = k1 * gDist + k2 * wDist + k3 * fDist;
10:    return quality;
```

Algorithm 7 Computing FDist

```
1: procedure ComputeFDist
2:     fDistances = new List();
3:     for all fingers do
4:         pos = new Position();
5:         for all bones do
6:             if bone.N_REACHED <= nextbone.N_REACHED then
7:                 pos = bone.endPoint;
8:         fDist = Distance(hookCenter, pos);
9:         fDistances.add(fDist);
10:    fDistTotal = fDistances.Sum();
11:    return fDistTotal;
```

6.5.4 Fulfilling Hand Constraints

For what concern finger constraints we shouldn't carry out any check because we already provided to determine manually the starting pose and the final pose of the grasp movements. The interpolation poses never present incorrect finger positions or rotations. Instead we might want to check whether the wrist position and rotation actually represent a possible hand pose which is reachable by the character. Although this issue should be addressed by future pertinent projects (see Section 7.1) we want to add a simple check that approximates the real human wrist constraints, when the character position is given. The algorithm deletes from the comparison all those poses that aren't included in a range of possible poses. First of all we need to define that range: we state that, starting from the basic pose described in Section 5.6.1 the wrist can rotate at maximum:

- 145°clockwise around z-axis
- 70°counter-clockwise around z-axis
- 45°clockwise around x-axis
- 45°counter-clockwise around x-axis
- 30°clockwise around y-axis
- 30°counter-clockwise around y-axis

We also state that the hook center rotation range can be approximately the same as the wrist one. So the check becomes quite straightforward: in the *QualityCheck()* function, before the evaluation we check whether the hook center actual rotation is included in the range of the possible ones. If it isn't, the quality of that pose is set to be equal to *-Inf.* In this way that pose is never chosen by the algorithm unless all the poses analyzed in one step correspond to incorrect hook center rotations (an extremely rare event, inconsequential if we consider the amount of steps performed per second). Notice that the starting position from which the wrist constraint are computed changes together with the character position, so the range must be updated after each frame rendering.

6.5.5 Unidirectional Step

As explained in Section 5.6 in order to speed performances we might want to use the *unidirectional step*. This will lighten the computation per frame but it will reduce the search power. In many situations this strategy could be successful; for instance if the character takes a long enough time to grasp an object, the computation finds a good output without problems even using the unidirectional step, because a sufficient number of iterations occurred. Algorithm 8 shows this alternative, but it's necessary to modify the iteration code as well: the direction passed as a parameter to the unidirectional step must change every time; in our tests we just let the directions alternate regularly in the natural order x, y, z.

Algorithm 8 Unidirectional Step

```
1:  procedure MakeStepUNI(direction, angle)
2:      qualities = new List(value, rotation);
3:      qualities.add(QualityCheck(), current);
4:
5:      hookCenter.rotate(direction, +angle);
6:      qualities.add(QualityCheck(), pos);
7:      hookCenter.rotate(direction, -angle);
8:
9:      hookCenter.rotate(direction, -angle);
10:     qualities.add(QualityCheck(), neg);
11:     hookCenter.rotate(direction, +angle);
12:
13:     max = qualities.Max();
14:     if (max != qualities(current)) then
15:         hookCenter.rotate(rotation corresponding to max);
```

6.5.6 Character Movement During Optimization

One problem that arises from the combination of iterative optimization and real time execution is the feeble validity of the starting position. In fact the starting position depends on the position of the character

and the character can move during the optimization. It's important that the starting position updates regularly because it is used both as a landmark for the positions computed at each optimization step and as a reference that defines the limits of the wrist rotation range. However we can't update it at every iteration or at every frame, because this wouldn't let the optimization proceeds, generating an output that always corresponds to the initial position for the current frame. We found a simple and effective solution that solves this issue. Let's think of our 3D environment as if it was a 2D environment with only x and z axes. When the initial position is computed, we define a line a that goes from the hook to the character and we store that x-z direction. After every frame, we compute the same line using the new position of the character and we call it b. While a doesn't change, b varies and if the convex angle between a and b is greater than 10° then we update the starting position and the direction of a that becomes equal to b. At this point the iterative optimization automatically starts again from the new starting position. This strategy doesn't allow to perform correct grasps while the character moves but, if we consider the time the character needs to point towards a target object and to prepare the grasp, in the case of steady character most likely the optimization has enough time to produce a correct output.

6.6 Performances

The algorithm only performs simple and common operations on entities: mainly translation, rotation and collision detection. Moreover, the pure computation part shows no complex procedures: it exploits the built-in functions for number, vector and quaternion manipulation. The amount of simple actions and procedures may afflict performances because everything needs to be executed within short time intervals. For this reason it is worth analyzing the order of the number of executed operations and procedures. Certain constants can be set by the developer while others may depend on the hand structure:

- N: Number of interpolations for a grasp simulation
- nb: Number of hand bones

- P: Number of poses evaluated in a step (7 in the normal step, 3 in the unidirectional step)

- v_{AVG}: Average number of vertical translations performed in the *TranslateToFree()* function

Considering one single step, P poses are evaluated; for each pose, the *TranslateToFree()* function performs 1 rotation and computes nb capsule collision detections for each vertical translation. The value v_{AVG} depends on the size of the target object and on the extent of a single vertical translation (defined by the developer), but in general it's close to 10 and never greater than 100 (otherwise it would be better to re-set the extent of vertical translations). Moreover, one simulation is performed for the pose: it consists of N times nb capsule collision detections and the same amount of rotations (at worse). About pure computation, each one of the P evaluations requires the execution of a series (3 in our case) of built-in functions that operate on numbers, vectors and quaterinions. This last part doesn't affect performances because the computation time is much inferior to the frame rendering time. The number of performed basic transformations such as translations and rotations is not big enough for those transformations to interest performances. Collision detection is what really matters in terms of computational time. Summarizing what we said before, the average number of capsule collision detections performed in one step is:

$$cd_{AVG} = P * nb * (v + N)$$

In order to give an idea of the amount of capsule collision detections, we use the values of our test case for the numbers:

- $P = 7$ (normal step)

- $nb = 24$

- $v_{AVG} = 10$

- $N = 20$

Using such values:

$$cd_{AVG} = 7 * 24 * (10 + 20) = 5670$$

In the case of unidirectional step the number changes considerably:

$$cd_{AVG}(uni) = 3 * 24 * (10 + 20) = 2430$$

With a smaller number of interpolations or vertical translation (N can be set by the developer while a value for v_{AVG} can be induced by changing the vertical translation extent) it's possible to decrease cd_{AVG} even more but too small values for N and v_{AVG} would reduce the correctness of the output. For instance if N is not big enough the fingers might not find the correct interpolation that allows them to touch the target object and they could result as detached or penetrating. The number of collision detections that an application can tolerate before lowering its performances depends on the power of the machine where it is executed and on the time interval between iterations.

Computation time of the algorithm bsically depends on cd_{AVG}. We show the results of some tests in which we tried different values, still keeping an acceptable level of realism. In particular we used the values 7 and 10 for v_{AVG} and the values 10 and 20 for N; combinations includes both tests using the normal optimization step ($P = 7$) and the unidirectional step ($P = 3$). T_{AVG} is the average time that an iteration takes to compute. Tests were performed on a Intel® Core™2 Quad Processor Q6600 (2.4 GHz), RAM: 4GB. The following table contains the values of T_{AVG} corresponding to the different combinations of parameters.

Combination	P	v_{AVG}	N	T_{AVG} (ms)
a	7	7	10	11
b	7	7	20	17
c	7	10	10	13
d	7	10	20	19.5
e	3	7	10	5.5
f	3	7	20	9.5
g	3	10	10	6.5
h	3	10	20	9.5

Tests were executed at 30 FPS so the available time for computation between two consecutive frames was $T_{FRAME} = 1$ [s] / 30 = 33.3 [ms]. Obviously the algorithm must not use all the available time because it's not the only process executed between frames. We considered T_{AVG} to be acceptable when $T_{AVG} < 1/2 * T_{FRAME}$. Indeed combinations b and d resulted in a slight reduction of FPS (about 28 instead of 30). If the computation requires more than half of the rendering time for any combination of parameters (just think of our tests executed at 60 FPS instead of 30) it's possible to split even more the single unidirectional step among frames using coroutines in Unity. Given that only a few applications (mostly recent video games) execute at fixed 60 FPS, we can consider the values of T_{AVG} obtained from the tests appropriate.

Chapter 7

Future Works and Conclusions

This document shows a novel approach to the grasping hand problem based on computer graphics methods. As all experimental projects, it's far from being a complete and flawless procedure and it represents a starting point from where several future researches may take place. We show some of the lines of development we intended our project to follow and we give some examples of applications for which the algorithm could be suitable.

7.1 Algorithm Improvements

We focused our efforts on developing the main phases of the algorithm using a different approach from the ones presented by other studies. There are many aspects that, if analyzed in details, could lead to sensible improvements of the procedure both in terms of performances and complexity. The following aspects are the most relevant ones according to our understanding of the whole problem:

- **MORE GRASP MOVEMENTS:**
  The very first feature that could be improved is the number of stored grasp movements. For our tests we only stored the spherical power grasp movement and the thumb-index precision grasp movement but if we follow the classification illustrated in Section 2.2.1 we could store fourteen other movements and we could chose

different types of grasps for different target objects, improving the realism. Plus, it's possible to think of another way to store the movements as we proceeded by saving all the rotation values for all the fingers one by one while there may be another faster and more reliable method. It would be even possible to change the interpolation method used by the algorithm and to try integrating pre-defined animations in order to generate movements. We couldn't achieve that because of the limitations presented by the Unity humanoid avatar (the one available in the Free edition) which doesn't include a complex enough hand structure. Certainly, those pre-defined animations must be quite flexible, such that it should be possible to stop the animation for each bone singularly.

- **AUTOMATIC HOOK POSITIONING:**
  In our design process we decide to set the hooks manually for each object, unless the hook was individual and it coincided with the pivot point of the object. Instead, we could have designed a part of the algorithm that decides automatically where to place the hooks. This could be done by searching for all points in the object structure which allow a stable grasp, like tubular elongated parts or parts that would fit in a eventual power grasp.

- **AUTOMATIC CHOICE OF GRASP MOVEMENT:**
  As the 3D shape of any target object is known from the start, one might think of developing a system that, based on that 3D shape, chooses automatically the grasp movement between the available ones. An example could be the analysis of the size of the object: if it's small enough the character could choose a precision type grasp. The study of the object structure is open to a huge variety of approaches: it might be subdivided it in 3D primitives or some graspable portions of it can be located in other ways. It's even possible to try out several grasp movements and see which one fits better by defining an evaluation system between grasps with different movements.

- **IMPROVING NATURALNESS OF POSTURE:**
  The weakest aspect of the algorithm is maybe the wrist constraint definition. A much detailed system could be set up in order to assure the naturalness of the whole grasp posture. We focused our efforts on studying finger movements and grasp evaluation, but the full-body posture might be a relevant factor for improving the realism.

- **MULTI-HAND GRASPS:**
  A natural evolution of the algorithm would be achieving the generation of multi-hand grasps. The approach could be the same, with the difference that the grasp movements would be much more complex because of the infinite possibilities; a study of new evaluation parameters would be required as well.

7.2 Possible Applications

While designing the algorithm we had in mind an application field in particular which is the video games field. In fact we exploited a game engine for the implemntation and we focused on real-time execution. It is easy to immagine how the algorithm could be used in such a field: even in most modern video games ususally this problem is ignored and characters don't really grasp objects but they're animated with predefined animations. Our algorithm would make any grasping action in modern video games more realistic. Of coruse it would also require different set up operations and several ad-hoc optimizations for the specific video game. Remaining in the computer graphics-oriented applications it would be possible to integrate the algorithm in 3D modeling softwares allowing to generate automatically grasp animations or simple grasp poses. This could be exploited both for 3D model production and for animated 3D scene generation. Robotics might benefit from this algorithm as well: many researchers are still facing the robotic grasping hand problem and a way to direct their studies could be a computer graphics-based approach. Most likely a grasp stability evaluation system would be needed in addition to the quality check. Plus,

the 3D model of the target object must be known completely; this could be achieved by use of sensors or by image analysis techniques (3D reconstruction from images).

7.3 Conclusions

With this work we wanted to deal with the grasping virtual hand problem following a computer graphics-based strategy. We proposed a real-time executed algorithm that generates correct single hand grasp poses with respect to a character and a target object through continuous optimization. We focused on virtual human hands but the algorithm supports a large variety of hand structures.

First of all we studied the hand skeleton and we analyzed the standard way of representing and manipulating hands in 3D interactive environments. Virtual bones and joints are located and named according to the real human hand anatomy; bone rotations are needed to move fingers while the hand's 3D mesh follows them. We also analyzed and modeled some of the anatomical limits of the joints, understanding how to generate natural grasp poses. After that, we defined a hand positioning system that allows the hand to move around the target object while always pointing towards it. Hook and grasp center were introduced in order to easily put that system to use. Hook, hand and grasp center are connected in such a way that every transformation applied on the hook is also applied on the others and every transformation applied on the grasp center is also applied on the hand.

At this point we began designing the core of the algorithm: starting from an initial position, it should have iteratively analyzed the closer positions and evaluated them in order to decide where to move the hand for the next iteration. After a finite number of iteration the algorithm should have moved the hand in a position that corresponded to a correct grasp pose. In order to develop a grasp evaluation system we introduced the possibility to simulate the grasp for a given position and compute a quality value for the produced grasp pose by check-

ing a series of relevant parameters. Comparison between poses was then reduced to a comparison between quality values (real numbers). Grasp simulation is achieved following this strategy: a grasp movement was subdivided in an arbitrary number of interpolations and for each interpolation a collision check was performed on each bone; if a bone collided with the target object then it stopped moving together with all the previous bones in the same chain. The quality value of a grasp pose was computed as a weighted sum of arbitrary parameters; we chose to use three parameters with negative weight: the distance between grasp center and hook, the distance between the end points of the first colliding bones of each finger and the distance between the wrist and the shoulder of the character.

We designed the optimization step such that it can be split in several ways among the frames. It consists of a quality evaluation (through simulations) of the current hand position and of the six closer positions (in terms of positive and negative hook rotations along the three axes); at the end of the step, the hand moves to the position with maximum quality. With high amount of computational resources it's possible to execute an entire step or more steps in one single frame. However, in order to provide a solution that is less effective but requires less computational effort, we illustrated the *unidirectional step* which only analyzes three positions at the time.

The proposed algorithm shows an experimental method that can be considered as a starting point for future researches which are meant to use computer graphics-based strategies in order to approach the grasping hand problem. The algorithm works nicely with a standard human hand model and with a discrete variety of graspable objects and its performances are decent. We believe that with further work and several optimizations and extensions this approach could be used in the process of developing complete and complex applications such as video games or 3D graphics software.

Bibliography

[1] Abaci, T., Mortara, M., Patane, G., Spagnulo, M., Vexo, F., and Thalmann, D. Bridging geometry and semantics for object manipulation and grasping. In *Proceedings of Workshop towards Semantic Virtual Environments"(SVE 2005) workshop* (2005), no. VRLAB-CONF-2005-021.

[2] Abe, Y., and Popovic, J. Interactive animation of dynamic manipulation. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2006), Eurographics Association, pp. 195–204.

[3] Aydin, Y., and Nakajima, M. Database guided computer animation of human grasping using forward and inverse kinematics. vol. 23, Elsevier, pp. 145–154.

[4] Chronister, J. Blender basics 4th edition.

[5] Ciocarlie, M., Goldfeder, C., and Allen, P. Dimensionality reduction for hand-independent dexterous robotic grasping. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on* (2007), IEEE, pp. 3270–3275.

[6] Ciocarlie, M. T., and Allen, P. K. Hand posture subspaces for dexterous robotic grasping. *The International Journal of Robotics Research 28*, 7 (2009), 851–867.

[7] Cutkosky, M. R., and Howe, R. D. Human grasp choice and robotic grasp analysis. In *Dextrous robot hands*. Springer, 1990, pp. 5–31.

[8] Datta, R., and Deb, K. Optimizing and deciphering design principles of robot gripper configurations using an evolutionary multi-objective optimization method.

[9] Douville, B., Badler, N. I., and Levison, L. Task-level object grasping for simulated agents.

[10] ElKoura, G., and Singh, K. Handrix: animating the human hand. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2003), Eurographics Association, pp. 110–119.

[11] Fu, J. L., and Pollard, N. S. On the importance of asymmetries in grasp quality metrics for tendon driven hands. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on* (2006), IEEE, pp. 1068–1075.

[12] Giang, T., Mooney, R., Peters, C., and O Sullivan, C. Real-time character animation techniques.

[13] Goldstone, W. Unity3d tutorial. http://learnunity3d.com/.

[14] Gribaudo, M. Lecture slides. Computer Graphics course, Politecnico di Milano.

[15] Han, L., Li, Z., Trinkle, J. C., Qin, Z., and Jiang, S. The planning and control of robot dextrous manipulation. In *Robotics and Automation, 2000. Proceedings. ICRA 00. IEEE International Conference on* (2000), vol. 1, IEEE, pp. 263–269.

[16] Hasegawa, T., Murakami, K., and Matsuoka, T. Grasp planning for precision manipulation by multifingered robotic hand. In *Systems, Man, and Cybernetics, 1999. IEEE SMC 99 Conference Proceedings. 1999 IEEE International Conference on* (1999), vol. 6, IEEE, pp. 762–767.

[17] Kallmann, M. Scalable solutions for interactive virtual humans that can manipulate objects. In *AIIDE* (2005), pp. 69–75.

[18] Kawamura, A., Tahara, K., Kurazume, R., and Hasegawa, T. Dynamic grasping of an arbitrary polyhedral object. *Robotica 31*, 04 (2013), 511–523.

[19] Kry, P. G., and Pai, D. K. Grasp recognition and manipulation with the tango. In *Experimental Robotics* (2008), Springer, pp. 551–559.

[20] Li, Y., Saut, J.-P., Cortes, J., Simeon, T., and Sidobre, D. Finding enveloping grasps by matching continuous surfaces. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on* (2011), IEEE, pp. 2825–2830.

[21] Lin, J., Wu, Y., and Huang, T. S. Modeling the constraints of human hand motion. In *Human Motion, 2000. Proceedings. Workshop on* (2000), IEEE, pp. 121–126.

[22] Liu, C. K. Dextrous manipulation from a grasping pose. In *ACM Transactions on Graphics (TOG)* (2009), vol. 28, ACM, p. 59.

[23] Miler, J. Finger alphabet by computer animation. In *Central European Seminar on Computer Graphics* (2002), vol. 2002, pp. 1–9.

[24] Miller, A., Allen, P., Santos, V., and Valero-Cuevas, F. From robotic hands to human hands: a visualization and simulation engine for grasping research. *Industrial Robot: An International Journal 32*, 1 (2005), 55–63.

[25] Miller, A. T., Knoop, S., Christensen, H. I., and Allen, P. K. Automatic grasp planning using shape primitives. In *Robotics and Automation, 2003. Proceedings. ICRA 03. IEEE International Conference on* (2003), vol. 2, IEEE, pp. 1824–1829.

[26] Pollard, N. S. The grasping problem: Toward task-level programming for an articulated hand. *MIT Artifical Intelligence Laboratory* (1990).

[27] Pollard, N. S., and Zordan, V. B. Physically based grasping control from example. In *Proceedings of the 2005 ACM*

SIGGRAPH/Eurographics symposium on Computer animation (2005), ACM, pp. 311–318.

[28] Reed, M., Zhou, W., and Wegner, D. Automated grasp modeling in the human motion simulation framework. In *Proceedings of the SAE Digital Human Modeling for Design and Engineering Conference* (2011).

[29] Rijpkema, H., and Girard, M. Computer animation of knowledge-based human grasping. In *ACM Siggraph Computer Graphics* (1991), vol. 25, ACM, pp. 339–348.

[30] Rougeron, M., Le Garrec, J., Micaelli, A., and Ouezdou, F. B. A control approach for real time human grasp simulation with deformable fingertips. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on* (2006), IEEE, pp. 4634–4640.

[31] Saxena, A., Driemeyer, J., and Ng, A. Y. Robotic grasping of novel objects using vision. *The International Journal of Robotics Research 27*, 2 (2008), 157–173.

[32] Suarez-Ruiz, F., Galiana, I., Tenzer, Y., Jentoft, L. P., Howe, R. D., and Ferre, M. Grasp mapping between a 3-finger haptic device and a robotic hand. In *Haptics: Neuroscience, Devices, Modeling, and Applications*. Springer, 2014, pp. 275–283.

[33] Tsuji, T., Harada, K., and Kaneko, K. Easy and fast evaluation of grasp stability by using ellipsoidal approximation of friction cone. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on* (2009), IEEE, pp. 1830–1837.

[34] UnityTechnologies. Unity3d 4 documentation and manual. http://docs.unity3d.com/Manual/index.html.

[35] UnityTechnologies. Unity3d 4 scripting reference. http://docs.unity3d.com/ScriptReference/index.html.

[36] Villani, L., Ficuciello, F., Lippiello, V., Palli, G., Ruggiero, F., and Siciliano, B. Grasping and control of multi-fingered hands. In *Advanced Bimanual Manipulation*. Springer, 2012, pp. 219–266.

[37] Weber, M., Heumer, G., Amor, H. B., and Jung, B. An animation system for imitation of object grasping in virtual reality. In *Advances in Artificial Reality and Tele-Existence*. Springer, 2006, pp. 65–76.

[38] Welman, C. *Inverse kinematics and geometric constraints for articulated figure manipulation*. PhD thesis, Simon Fraser University, 1993.

[39] Ye, Y., and Liu, C. K. Synthesis of detailed hand manipulations using contact sampling. *ACM Transactions on Graphics (TOG) 31*, 4 (2012), 41.

[40] Ying, L., Fu, J. L., and Pollard, N. S. Data-driven grasp synthesis using shape matching and task-based pruning. *Visualization and Computer Graphics, IEEE Transactions on 13*, 4 (2007), 732–747.

[41] Zhao, W., Zhang, J., Min, J., and Chai, J. Robust realtime physics-based motion control for human grasping. *ACM Transactions on Graphics (TOG) 32*, 6 (2013), 207.

[42] Zhu, Y., Ramakrishnan, A. S., Hamann, B., and Neff, M. A system for automatic animation of piano performances. *Computer Animation and Virtual Worlds 24*, 5 (2013), 445–457.

[43] Zoltan, R., Csaba, A., van der Vegte Wilhelm, F., Irme, H., et al. A study of real time simulation of grasping in user-product interaction. *Guidelines for a Decision Support Method Adapted to NPD Processes* (2007).