Chapter 4

The Basic Animation System

4.1 Introduction

In this chapter we present a platform for an animation system that has been developed to provide the facilities essential for generating animation. These facilities are the computational structures which comprise the animated figures and the control modules that manage them. Such a platform functions as a black box in which a variable number of figures are created, managed and visualised. Computer animation is an application that is naturally suited to the object-oriented paradigm [Mahi90]. Every entity, be it an animation figure or a control module, is a computational structure identified as an object with inherent behaviour. These objects communicate with each other through messages as in ASAS [Reyn82], that is, an object activates the behaviour of another by sending messages. The animated objects are the cast of the animation environment and usually are called *synthetic actors*, or simply *actors* [Reyn82, Magn85, Maio90, Magn90]. The concept of actors has its origin in artificial intelligence [Hewi73, Agha86] and fits in well if we consider the animator as being the director of a theatre [Rids87].

Currently in our animation, the cast comprises synthetic human actors and secondary objects such as tables, chairs, counters, and glasses. The human actor is the main animation data type and will be presented in detail later in this chapter. The secondary objects provide the surrounding where the actors will develop their activities, that is, these objects are considered as obstacles, places, landmarks, or as a loads. The environment space is divided into areas which define different places accessed by the actors. The virtual camera is an object that provides different views of the scenario.

4.2 The Human Figure

The human figure is perhaps the most complex object to model and animate. Because of the multiplicity of figures performing in an environment, there is a greater motivation that each of these figures be viewed as an integral entity that combines both the body structure with motion in a single structure [Magn91]. In this section the basic structure of the human synthetic actor is presented.

4.2.1 The Representation of Articulated Bodies as a Chain of Links

The human figure is built around the skeleton which can be regarded as a chain of articulated links. These links are segments that are connected to each other at the extremities by joints. Denavit and Hartenberg [Dena55] formulated a 4x4 matrix representation for a link in a 3-D coordinate system. This matrix represents the position and orientation of a link which Paul calls a *coordinate frame* [Paul81]. In a simple configuration, a chain of links can be regarded as having two ending links: the *proximal* link and the *distal* link [Kore82]. The *proximal* link can be thought as being rigidly affixed to a body external to the chain, serving as a reference or attachment (Figure 4-1). The body is represented in the world coordinate system. The link following the *proximal* link, is based on the coordinate system of the *proximal* link. The same arrangement applies to the other links that follow in succession until the *distal* link, which is the free end of the chain. In this way, because of the dependence of one link on the other, any disturbance along any point of the chain affects the coordinate systems of the remaining links up to the *distal*. Robotic manipulators are typically chains of links where tools such as grips can be attached to the distal links making up a new distal end which is usually called the *end effector*. The final location and orientation of the end effector in an operation is called the *goal*.

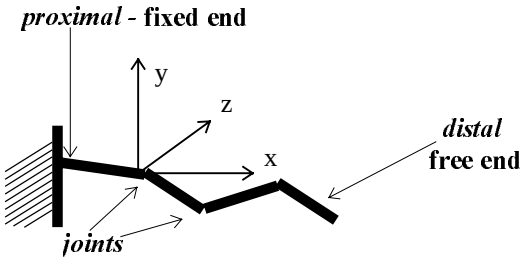


Figure 4-1: Chain of links.

The dependency of one link on another in the chain is therefore represented by a product of matrices,

$$T = A_1 * A_2 * .... * A_N \qquad (1)$$

where the coordinate frame of the link N, A_N is dependent on the coordinate frame A_{N-1}. Alternatively, T represents the coordinate system of the *distal* link given by a sequence of small rotations starting from A_1 to A_N. As Paul explains, six degrees of freedom, three for position and three for orientation, permit the manipulator to reach anywhere within its range of motion. For the sake of simplicity, each degree of freedom (DOF) can be assumed to be represented by an A_i matrix. The matrix T can be written as a six-by-six matrix which is called the Jacobian. Details of the mathematical development are discussed by Paul. Thus, an object or a tool at the tip of the distal end of the manipulator has its position determined by computing the expression,

$$Obj_{World} = T * Obj_{Manipulator} \qquad (2)$$

which is simply an extension of the expression given in (1). The Obj_{World} is basically a matrix with the position and orientation of a specified object in world coordinates and $Obj_{Manipulator}$ is the coordinate of the object at the tip of the manipulator. Such expressions are called *direct* or *forward kinematics*. In fact, what is really useful is the inverse of this expression. That is, the question now becomes, what is the sequence of differential rotations that adjust the configuration of the chain so that the tip of the chain can be in the neighbourhood of the object. So, the problem turns out to be the determination of the inverse of that chain of matrices, which is called the Inverse Jacobian. The inversion of a Jacobian is by no means trivial [Paul81, Nagy87] and much research has been done in the field of inverse kinematics. Inverse kinematics has also been used in computer animation for determining the configuration of an articulated figure [Gira87]. The solution to this problem is in general determined numerically or symbolically. This is particularly difficult in the case of the human body because of the large number of degrees of freedom yielding many possible body configurations.

4.2.2 Representation of the Human Figure

Most animation systems [Zelt82b, Magn85b, Cach86, Badl87b, Miro89] involving human-like bodies have described the articulated body as composed by several chains of linkage organised hierarchically as a tree. Our human figure is composed of two main linkages uniting in the region of the centre of gravity. Figure 4-2a presents a model of a human-like figure while Figure 4-2b is the equivalent tree hierarchy of the figure. The node *body* joins both nodes, *pelvis* and *spine*, behaving as an abstract limb. Although the *body* node is not "physically" represented by a particular limb, it operates as if it were. The change of the figure's direction is affected by the change of direction applied to the *body* node.

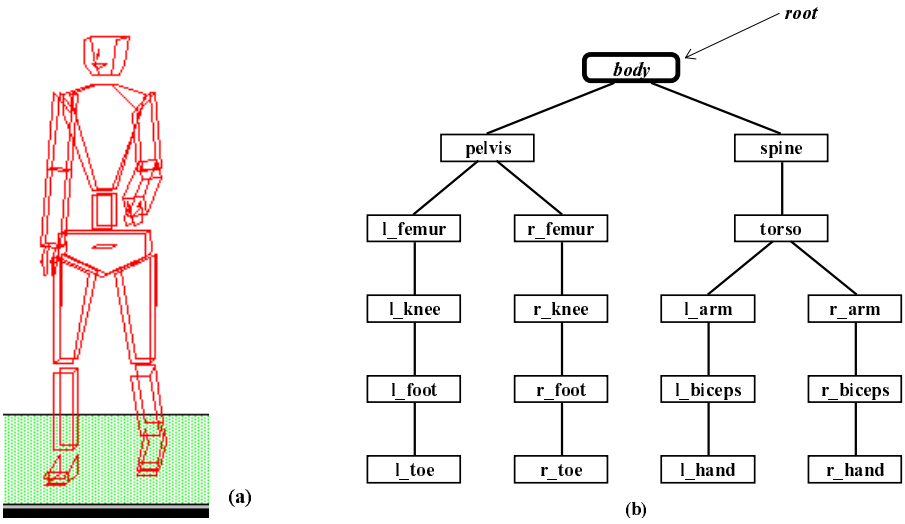


Figure 4-2: The human body as an hierarchical chain.

4.2.3 Motion in Articulated Bodies

The motion in articulated bodies is achieved by bending (rotating) the limbs around the joints. One configuration of the body is thus obtained by a series of bend operations applied to a number of limbs of the body, that is, a single motion is described by a collection of *bend* operators. The transition through a consecutive number of such configurations generates a movement which is the way key-framed animation typically operates. Because one link is dependent on another, the achievement of a new body configuration is made in the form of a tree traversal, starting from the *proximal* limbs and spreading towards the *distal* limbs.

However, if the bending joints are always considered from the *proximals* spreading to the rest of the chain, in the performance of certain motions such as walking, the figure will appear to slip without moving forwards. To overcome this problem an additional operator called *fix* is used. Therefore, limbs such as the weight-bearing foot can be attached to the ground while the rest of the body moves relative to it (Figure 4-3). A different scheme has been implemented by Zeltzer [Zelt84]. Zeltzer uses *bend* and *pivot* rotational operators in combination. The *pivot* operator performs exactly the complementary effect of the *bend* operator when rotating around a joint. That is, instead of rotating the remaining chain below the joint as in a *bend* operation (Figure 4-4a), the *pivot* operator rotates the entire chain except the chain extending to the *distal* limb (Figure 4-4b). Thus, a limb that would move under the *bend* operator would remain still under the *pivot* operator while the rest of the body rotates around that joint.

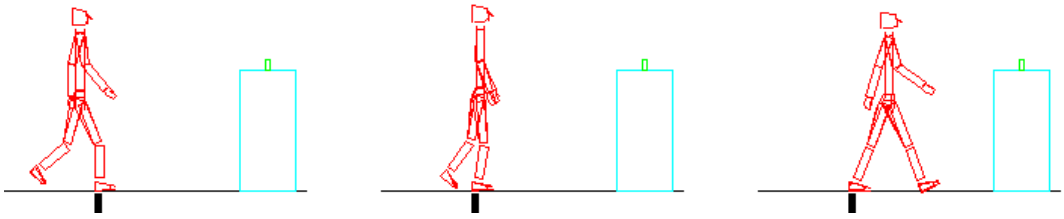
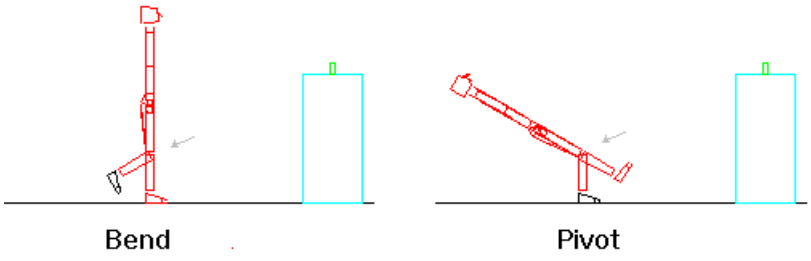


Figure 4-3: The fix operator avoids the "slipping" effect.



(a) (b)

Figure 4-4: Cases of bend and pivot operators applied to the knee.

4.2.4 The Skeleton as Data Type Structure

The object oriented programming approach is a very convenient way to stereotype objects such as the human figure which have both a representational model and associated behaviours. The composition of an animation usually requires a variable number of figures and the specification of figures as high-level types of objects makes the

task of replicating and handling them more convenient. The human figure is thus identified as a class of objects, *Skeleton,* that encapsulates the data structures and procedures. Figure 4-5 presents only a small part of the actual implementation of the skeleton object. Three components are identified: the internal data structures; the initialisation procedures; and the procedures implementing the figure's motions which are also called *skills.* The walking motion is implemented by two procedures, *walk_right_swing* and *walk_left_swing*, which are discussed in section 4.3.8.1.

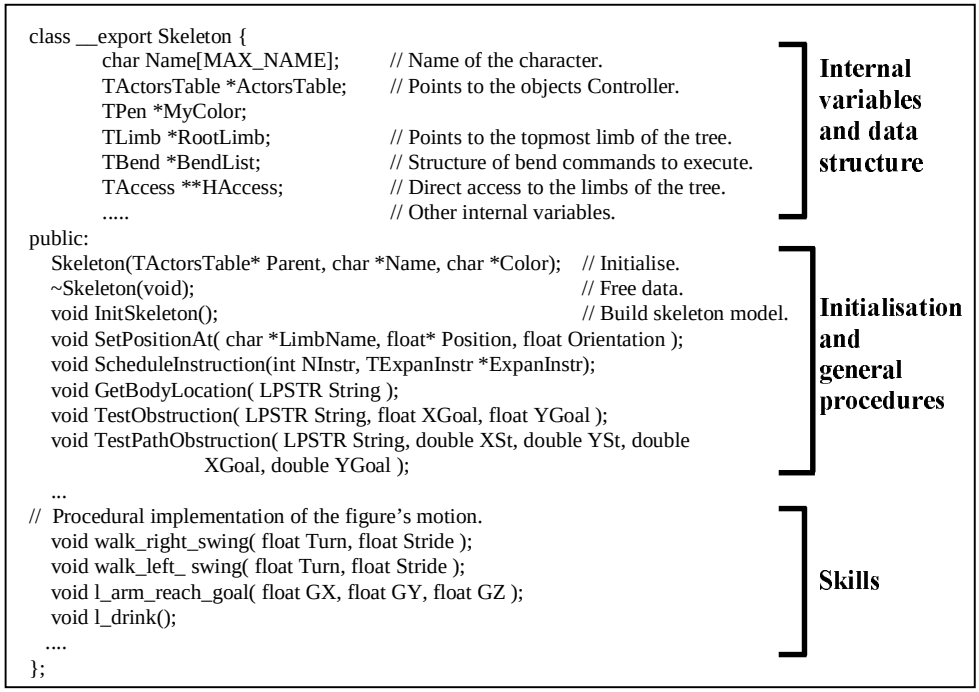


Figure 4-5: The human object type.

4.2.5 The Representation of a Limb Data Type

In our animation system the motions of the human figure are constituted as rotations of rigid limbs around the joints without other kind of movement such as twisting or stretching. Therefore a limb can rotate at its joint around up to three axes in 3-D space corresponding to three DOFs. In the upper part of Figure 4-6, a DOF is shown in the form of a C++ declaration which comprises the internal data followed by procedure declarations. The *ProcessInstruction* sets the new angle to which the DOF is to be rotated and computes the *DegreeStep* by which the DOF will be advanced in the next few animation frames until the *EndTime* frame. The *TickDof* procedure advances a fraction of a bend operation, *DegreeStep*, for the current frame. In the lower part of

Figure 4-6, a limb type structure gathers at most three DOFs which will take part in the composition of the limb's transformation matrix, *Transf*. There are also variables pointing to the next nodes of the chain which are used in the traversal of the figure's tree. In summary, this class representing a DOF is the basic unit of motion, the *bend* operation, in the composition of a "motion structure". This is illustrated in the section 4.3.4.

```
class Type_DOF  {
        float DegreeStep;
        float Angle;
        float AngleMin;
        float AngleMax;
        int   EndTime;
public:
        TDOF() { DegreeStep = Angle = 0.0; EndTime = 0; };
        ~TDOF() {};
        float TickDof(int curr_time);
        void ProcessInstruction( float Degree, int curr_time, int Duration );
};

typedef struct TLimb  {
        char Name[MAX_NAME];
        Type_DOF X, Y, Z;
        TMatrixTransf *Transf;          // Coordinates of the limb.
        FileDescription *JFD;           // Points to the limb model.
        TLimb *Chain;                   // First chain of limbs.
        TLimb *NextChain;               // Next chain in the same level.
        int LimbLength;
} TLimb;
```

Figure 4-6: A limb data structure.

4.2.6 Motion Representation

As discussed previously, the motions of articulated bodies are specified by *bend* and *fix* operations. There are two modes of motion specification: the *interpreted mode* and the *procedural mode*. The interpreted mode is a convenient way to compose new and simple skills into a file and then test them immediately. In the case of complex motions such as the walking motion, a conventional language such as C++ provides a powerful syntax to specify arithmetic expressions for specifying complex equations of motions. The skills written in the form of procedures are thus added to the *Skeleton* class as part of its behaviour (Figure 4-5). Both schemes are indeed composition of motions as keyframes which are convenient in a multiple figure environment.

4.2.6.1 The Interpreted Mode

In this mode the definition of a skill starts with the INSTRUCTION keyword as shown in Figure 4-7a. Each skill comprises one or more keyframes which start with the MOVE keyword. The start time of each keyframe is defined relatively to the overall skill and usually the first *move* starts at time zero of the skill. A keyframe comprises a collection of *bend* operations and at most one *fix* operation, and all these operations take place simultaneously. The skills implemented in the interpreted mode are stored in a file and a table of skills is thus created during the initialisation stage of the BAS (see section 4.3).

```
INSTRUCTION <skill_name> <total_duration>
  MOVE <keyframe name> <start_time> <duration>
    BEND <limb_name> <DOF_axis> <target_angle>
    BENDs ...
     ......
    FIX <limb_name>

  MOVE
    BENDs ...
     ......
INSTRUCTION
....
```

(a) Motion specification in interpreted mode

```
INSTRUCTION flex_muscle 10
MOVE  lift_arm   0   6
BEND  r_arm  y  70
BEND  r_arm  x  80
BEND  r_biceps z 100

MOVE  flex_arm   6   4
BEND  r_biceps  z  120
```

(b) Example of an arm motion

Figure 4-7: Representation of skills in the interpreted mode.

4.2.6.2 The Procedural Mode

In the procedural mode the composition of a skill is similar to the interpreted mode, however, some "operational" procedures (*SaveCallBack* and *start*) have to be invoked explicitly following the format presented in Figure 4-8. These procedures help to build an arrangement of *bend* and *fix* operators at runtime. The use of procedures allows the bend operators to consider values from arithmetic expressions coded in the program rather than predefined constants used in the interpreted mode. These expressions depend on external parameters entered in <*param_i*>. The procedure *SaveCallBack* stores the time when the process indicated by *curr_action* will be concluded and acknowledged back to the external caller. The <*keyframe_i name procedure*> are procedures written by the user which must have a call to the procedure *start* followed by sequence of calls to *bend* procedures. These procedures are expanded into a motion structure when <*skill name procedure*> is invoked and each part will be executed at the specified time. Figure 4-8 presents the minimum specification for a motion structure.

```
void Skeleton::<skill name procedure>( <param_1>, ...., <param_n> )
{
        SaveCallBack( curr_time + <duration expression>, curr_action );.
        <keyframe1 name procedure>( <start time expression>, <duration time expression>, <param_list>
);
          ...
        <keyframeN name procedure>( ... )
}
void Skeleton:: <keyframe1 name procedure>( int Start, int Duration, <param_list> )
{
        start( Start );
        bend( <limb_name>, <DOF_axis>, <target angle expression>, <duration expression> );
        bend
         ....
        fix(<limb_name> );
}
```

Figure 4-8: Representation of skills in the procedural mode.

4.3 Components of the Animation System

The basic animation system (BAS) depicted in Figure 4-9 provides the facilities to create, control and visualise animated figures. In fact, apart from the *interface* module, all the components of the BAS are entities which are created during run-time, be it a controller component or a controlled figure. These entities are typically objects in the sense of object oriented programming, that is, they have internal data structures which are their private memory and they have procedures that implement their intrinsic behaviours. The operation of an entity is made by invoking these procedures which are implemented as methods.

The *interface* is a module composed of several procedures which realise the task of decomposing external commands into calls to the internal components of the BAS as well as returning the results of the operation. Two procedures undertake the initialisation and finalisation of an animation session while the others undertake tasks such as the creation of a new object, the scheduling of a motion to an actor, advancing one frame of animation, the enquiry of the direction and location of an actor, testing the proximity of an actor to specified object, etc. Because the external commands are encoded in long strings, the procedures of the *interface* firstly decode the strings into objects names and parameters, and forward them to the components of the BAS or they invoke the objects to execute specified operations. The initialisation operation creates the three components of the BAS: *decor controller*, *cast controller*, and *visualisation component*.

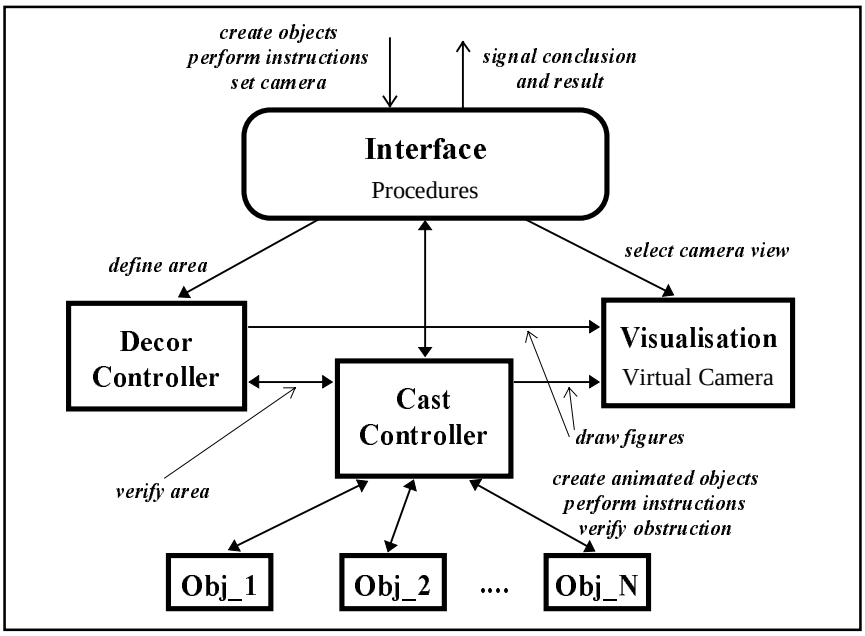


Figure 4-9: Organisation of the animation system.

4.3.1 The Decor Controller

This component creates a table of areas requested for the animation environment. These areas are just coloured rectangles for drawing the layout of a floor for the animation scene. Thus the only action performed by this controller is to add definitions of new areas and call the *virtual camera* to draw them. The purpose of these areas is to diversify the environment of the animation scenario by associating them with different uses or meanings. For example, an area can represent a restaurant, a bar area, an exit area ("door"), etc. In the level of the system controller, which controller is discussed in the following chapters, each area is handled as an entity with characteristics associated to it that can affect the behaviour of the animated figures. In the BAS level, the Decor Controller is simply limited to drawing these areas.

4.3.2 The Cast Controller

The *cast controller* is the component of the BAS that organises in a single unit the control of all animated objects created during the animation. This includes the creation of an instance of an object and its destruction, as well as the access to the object instance when a specific operation invokes its action. The other major purpose of the *cast controller* is to allow a collective operation with the cast. For example, when generating

a new animation frame all animated objects have the ongoing motions advanced by a fraction. In another situation the controller can behave as a "smart table" in which an object can have access to the others of the cast by "broadcasting" a call for observing a specific behaviour. For example, in the walking activity a figure broadcasts a call to others to test themselves if someone happens to be in a given path of that figure. There are two groups of objects under this controller, the *active objects* and the *static objects*.

4.3.2.1 The Active Objects

An active object is basically the *Skeleton* data type which was presented in section 4.2.4. In the initial position, all the limbs of the figure are in straight angles as shown in Figure 4-10. That is, the DOFs have a zero degree as rotation angle and the limbs are placed vertically or horizontally; centred or at either extreme. The right arm and the left femur have their coordinate positions on top while the torso has it at the bottom. The orientation of the figure in the plane *x-z* is zero when the figure's front is aligned with the *x*-axis of the world coordinate system.

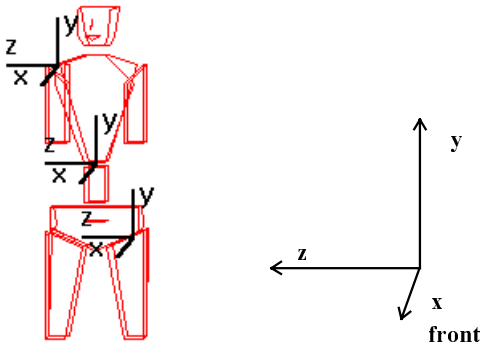


Figure 4-10: Coordinate of the skeleton in resting position.

4.3.2.2 The Static Objects

The static objects are part of the environment decor. They can be considered as resources to be accessed by the active objects or as obstacles to be avoided. These objects may have properties associated with them such as colour, size, location, and orientation. The implemented objects, such as glasses, counters, tables, and chairs, take part in the "bar scenario".

4.3.3 The Visualisation Component

This component is basically a *virtual camera* entity with a choice of two kinds of projections, parallel and perspective. Obviously typical parameters such as centre of projection, reference point, view up, etc. can be set up for the virtual camera [Harr83, Magn86, Mort89]. In order to make the handling of the camera efficient, a number of predefined views can be stored in an external file and then selected during animation.

For each frame of the animation sequence the virtual camera is called to draw the floor and all the cast. All figures are modelled as wired-frame structures and the virtual camera only displays wire frames without hidden line elimination and rendering.

The drawing of non-articulated figures such as a table and a chair is trivial. Articulated figures are fairly complex to draw because the bending of one limb requires that the rest of the chain affected by it must also be updated. Such an update is effected by a traversal across the tree structure of the figure as discussed in section 4.2.3. The traversal starts from the centre of gravity, the *body* node. Each DOF (degree of rotational freedom about x, y, or z) of a limb is checked to determine whether a bend is required. If so, a rotation matrix for this DOF-axis is determined and then applied to the coordinate system of those limbs under its influence. When all DOFs of this limb have been updated then the virtual camera is called to draw it. The process successively deals with each node of the tree.

The determination of the rotation matrix is a laborious computation because such a matrix must describe a rotation about a generic axis. This occurs because each limb of the skeleton is in an arbitrary direction. One approach to derive a rotation matrix about a DOF-axis is to bring this axis into alignment with one of the world coordinate axes, for example the z-axis. The alignment is done by translating the limb coordinate system to the world origin, rotating it about the x- and y-axes to make the coordinate system of the limb match the world coordinate system. After which the specified rotation about the DOF-axis can be applied about the z-axis. To complete the composition of the rotation matrix, the limb must return to the original coordinate system. Thus, the limb coordinate system is rotated about y- and x-axis in the opposite direction by the same amount, and

finally it is translated back to the original coordinate location. Therefore, the rotation is obtained by the following matrix product:

$$M_{DOF} = T\, R_y\, R_x\, Rot_z\, R_y^{-1}\, R_x^{-1}\, T^{-1} \qquad (3)$$

where R_y and R_x are the rotations applied to the y- and x-axis that align them to those of the corresponding world coordinate axes. Rot_z is the intended rotation around the generic axis. T is the translation matrix that brings the coordinated system of the generic axis to the origin of the world coordinate system. Details of the composition of the matrices as well as the mathematical developments are explained by Harrington [Harr83].

4.3.4 The Structure of Executing Motion

Upon a request to execute a motion command, the animation interface sends the command to the actor that will perform it. The motion can be performed if its name is found in the *Motion Library*, that is, in the table of skills (for skills implemented in the interpreted mode) or, as a second alternative, be invoked if it has been implemented as procedures. If the skill is a valid one it is expanded into a list of *moves* structures (Figure 4-11) and instantiated with information about the start time, duration, and the targeted angle. Each *move* of the bending list is scheduled for execution in the skeleton structure when the start time matches the animation "current time". In each limb's DOF, as linear interpolation is used, the amount of rotation for each animation frame is defined by the total angle to rotate divided by the duration. At regular times the *interface* receives requests to advance by one animation frame. The *interface* thus sends a call to the *cast controller* to instruct all the cast to be displayed by the visualisation process discussed in the previous section. The list of *callbacks* in the motion structure (Figure 4-11) stores the numbers of the processes of the external commands, so that when a motion related to a process completes the sender of the motion command (the task control entity which is discussed in Chapter 8) is notified.

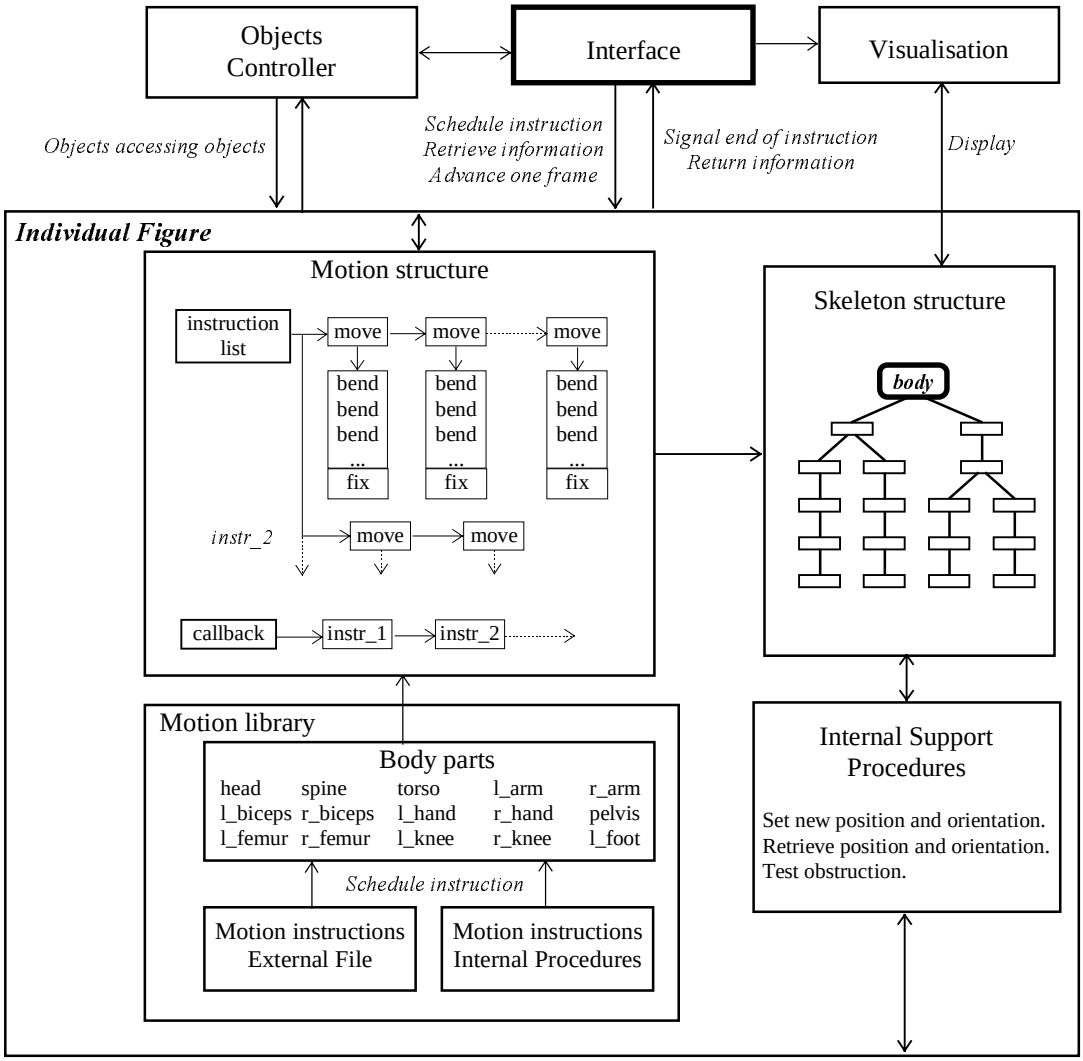


Figure 4-11: Graphical representation of the structure of motion during execution.

4.3.5 Goal-directed Motion

As discussed in section 4.2.1, the problem of finding a configuration for a chain of articulated links in which an end component, the *distal* link, reaches a point in the space is an inverse kinematics problem. The solution of the problem is thus to find a set of rotation angles q_i that satisfies a system of kinematic equations. Korein and Badler [Kore82] survey several alternative solutions. Their approach is to solve the goal achievement as a reach hierarchy as shown in Figure 4-12a. The solution is focused in one DOF at time from *proximal* to *distal* links. It considers two links (parts) in its analysis: the link which has the DOF being solved and the subsequent chain ending to the distal link. The latter part is treated as a single *segment* which is identified by this

nomenclature in this chapter. W_i is the workspace covered by link i. The combination of both links reaches any point in the grey area (Figure 4-12a).

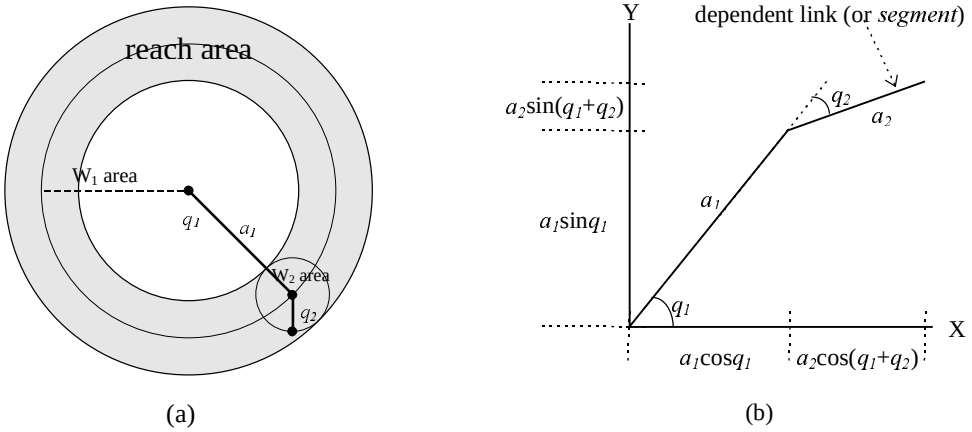


Figure 4-12: Korein's approach to reaching a point goal.

In this simple example of Figure 4-12a, a chain of two unconstrained links has two rotational DOFs on the plane which space of reach is confined in the greyed cylindrical area. The solution is thus to determine q_1 and q_2 from the trigonometric equations (Figure 4-12b). Their solution is described by the algorithm in Figure 4-13 [Kore82], however, details of mathematical considerations are discussed in their paper. Later on, Badler et al [Badl87b] improved this algorithm by using multiple constraints. Each DOF is constrained to specified joint angle limits and the reach is "proportionally" spread throughout the chain.

Let the chain be C1 and its workspace, W1. Let the subchain with just the most proximal joint and link deleted be C2 with workspace W2, and so on.

If goal **p** is not W_1, then
    it is not reachable: give up.
Otherwise:
    for i := 1 to number of joints in C:
      adjust q_i only as much as is
      necessary so that the next
      workspace W_{i+1} includes the
      goal **p**.

Figure 4-13: Korein's algorithm for the reach approach.

4.3.6 Goal-directed Arm Reach

Korein's method [Kore82] for reaching a goal has been employed in the implementation of the *pick up* action in the current system. The chain of limbs of an arm considered in this action are: *arm*, *biceps*, and *hand* [2]. As the human arms are capable of very complex motions, much more than any robotic manipulator, some assumptions have to be established in order to draw a strategy for solving the figure's motion. Firstly it is assumed that the figure can only reach objects within the figure's front half-space, more specifically, a half-sphere with its centre in the joints formed by *shoulder* and *arm* at either sides of the figure. The *arm* is a limb joining the *shoulder* with three DOFs (x, y, z), the *biceps* has only one DOF (z-axis), and the *hand* two DOFs (x, z). The following steps are strategies for reducing the complexity of the reach problem. That is, each DOF is solved one at a time always observing that the goal is kept within the reach. Working from the *proximal* limb towards the *distal* limb, that is, from the *arm* to the *hand*, and looking to the problem as comprised by "two links" as discussed in the previous section (Figure 4-12).

i) **observe if the object is within reach.** If the target object is not within the range of a fully stretched arm, i.e. the workspace of an arm, then the figure must first approach the object, either by walking, leaning forward, or turning. The figure adjusts its direction towards the object if the object is not in the semi-space of the figure's front.

ii) **find the x-axis of the arm.** This is not necessarily a problem to be solved given the degree of redundancy of the arm. If this DOF remains where it is, the goal can be reached with the arm in an "open wing" fashion. However, if the "wing" formed by the *arm* and the *torso* is opened too wide a simple heuristic could be used to close it to a visually acceptable angle (Figure 4-14a).

iii) **align the arm with the world coordinates.** This is similar to the problem of rotating about an arbitrary axis discussed in section 4.3.3. The purpose is to prepare for the two next steps by making the y-axis of *arm* coincide with the y-axis of the world

---

[2] In order to avoid confusion, names of rigid limbs are written in italics.

coordinate system. Obviously not only the whole arm (*arm*/*biceps*/*hand*) should be rotated as a rigid block, but also the distance and orientation of the object should be kept relative to the *arm*'s *new* coordinate system, that is, the same rotation applied to the entire arm is applied to the target object. In Figure 4-14b both the whole arm and the object have been rotated and from the view from above the extension of the *arm* does not appear because it is perpendicular to the paper. A small circle indicating the *arm* is shown in Figure 4-14b.

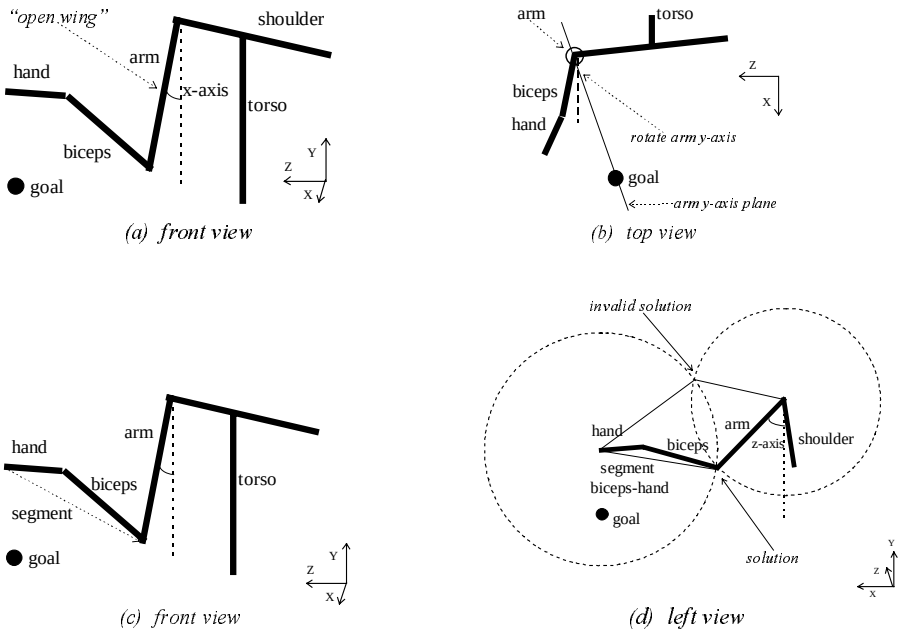


Figure 4-14: Configurations of an arm.

iv) **determine the *segment* defined by elbow to the tip of the hand.** If no constraint is imposed on the x- and z-axes of the *hand* then the position of the *hand* can either be maintained as its with the x/z angles or they can be assigned to zeroes in which situation the length of the segment formed by *biceps* and *hand* is exactly the sum of the length of both limbs. If either *hand*'s axes (x or z) are currently non-zero then the equivalent *segment* has to be determined. This *segment* is a composed vector of the sub-chain (*biceps* plus *hand*) which is shown in Figure 4-14c.

v) **align the *arm-segment* plane with the goal.** This determines the amount of angle the *arm* must turn about the y-axis such that the tip of the hand will be on the plane formed by the *arm* and the object (Figure 4-14b).

vi) **find the *arm* z-axis rotation.** The aim is to determine the amount of rotation about the z-axis an *arm* should do in order to make the tip of the *segment* (tip of the *hand* component) reach the goal position. Obviously the determination of the z-axis of the *arm* requires the determination of the z-axis of the *segment* (i.e., elbow) in relation to the *arm* (step vii). One way to do this is to draw two circles with the *arm* (centred at the base) and with the *segment* (centred at the finger) as shown in Figure 4-14d. If they intercept in two points, the "lower" intersection point is chosen, as the elbow has its movement constrained to one side. The trivial case occurs when both circles intercept in one point only.

vii) **find the biceps z-axis.** After the previous step the distance between the elbow and the tip of the hand is exactly the length of the segment, thus the biceps z-axis is computed easily. However, if the hand z-axis is non-zero then this contribution should be taken into account by solving the triangle segment-biceps-hand.

4.3.7 Holding and Releasing an Object

Once the hand of a humanoid has reached an object, for example a glass, the action of *picking up a glass* becomes effective if the humanoid can carry it along. Because the glass is a passive object which is now in control of the arm, visually it must accompany the *hand* in any movement. Any object visualised on the screen has a geometrical model whose position and orientation is represented in terms of an homogeneous matrix relatively to the world coordinates. Thus, $Glass_W$ and $Hand_W$ are matrices representing the models of glass and *hand* respectively, while $Glass_H$ is the matrix representation of the glass relatively to the hand. Expressions (4) and (5) are equivalent but they have different uses. They establish that at any moment the glass coordinate system relative to the world, $Glass_W$, has an equivalent coordinate system relatively to the hand, $Glass_H$.

The holding of the glass occurs when the *hand* has reached the vicinity of the glass. Thus, $Glass_H$ is computed using expression (5). Obviously, the visualisation of the glass uses the $Glass_W$ matrix which is obtained through equation (4). As the position and orientation of the *hand* may change, $Glass_W$ is always computed in every animation frame with updated $Hand_W$.

$$Glass_W = Hand_W * Glass_H \qquad (4)$$

$$Hand_W^{-1} * Glass_W = Glass_H \qquad (5)$$

The release of the glass is simple. $Glass_W$ is computed from expression (4) only once and in any subsequent visualisation of the glass $Glass_W$ is used unchanged. So that the glass will appear motionless.

4.3.8 The Locomotion of the Human Figure

The walking activity is probably the most complex motion of human behaviour. The complexities may not be apparent from simple observation. The problem has been the subject of studies in the medical field [Inma82, Lamo71, Murr64, Saun53]. These studies indicate that the walking gait of a normal person varies with age, height, sex, speed, stride, etc. It is necessary to employ suitable equipment and photograph records to determine the magnitudes, directions, and rates of change of translations and rotation of the body elements. Saunders et al. [Saun53] have identified the following determinants in human walking: pelvic rotation, pelvic tilt, knee flexion in the stance phase, foot and knee mechanisms, and lateral displacement of the pelvis. One feature that has been observed, as shown in Figure 4-15 [Saun53], is that the locus of the top the figures in a sequence of movements approximates to a sinusoid.

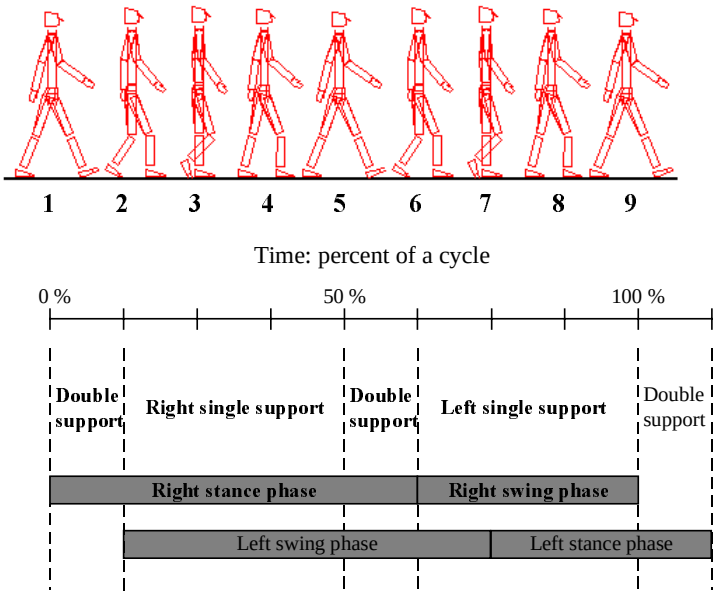


Figure 4-15: Phases of the walking in time and distance.

In computer animation a number of methods to control the walking action have been devised such as keyframes, adaptive walking control [Zelt82], direct and inverse kinematics methods [Boul92], dynamics [Brud89], etc. Inman et al. [Inma82] discuss the process of walking and present the general pattern of walking as shown in Figure 4-15. The number of key postures considered along a cycle of movement may vary from one implementation to another. A common feature of these animation systems is the use of a finite state machine scheme to control the different stages of biped walking.

4.3.8.1 The Walking Motion

In the present work, the walking motion is implemented procedurally using the forward kinematics method. The sequence of key stances is similar to that presented by Inman [Inma82]. Basically there are three keyframes for each of the left and right swing phases which make up a complete walking cycle. In the first keyframe or stage a leg is lifted first in preparation for a step. In the second stage both legs are stretched, with the lifted leg going forward with the heel striking the ground and the weight bearing leg remaining at the back. The third and last stage is the conclusion of the step. The body leans forward transferring most of the weight to the front leg, the body straightens up while the back leg retracts. Figure 4-16 exemplifies a procedure implementing the phase of the right leg swing of the walking motion. The three stages making up the motion are written as procedure calls. Details of implementations have been discussed in section 4.2.6.2.

```
void Skeleton::walk_right_swing( Fraction_forward, Fraction_sidewards,  Turn )
{
        SaveCallBack( curr_time + 9, action_process_number );
        walk_right_stage_1(  0,  3, Turn );
        walk_ right_stage_2( 3,  3, Fraction_forward, Fraction_sidewards );
        walk_ right_stage_3( 6, 3 );
}
```

Figure 4-16: Procedure implementing right leg swing of the walking motion.

A similar procedure is implemented for the swing of the left leg. The procedural implementation of the walking motion allows the length of the stride to be adjusted to a fraction of a full step forward. In addition it is possible to specify a sideways component to the step. Each stage is performed in three frames starting respectively at the relative

times 0, 3, 6 and so on. The stages 1, 2, 3 of Figure 4-17 and Figure 4-18 are performed by the procedure given above. In both cases the left leg is the support leg, and the right leg is swung forward. These three stages of walking are described in detail in the following section.

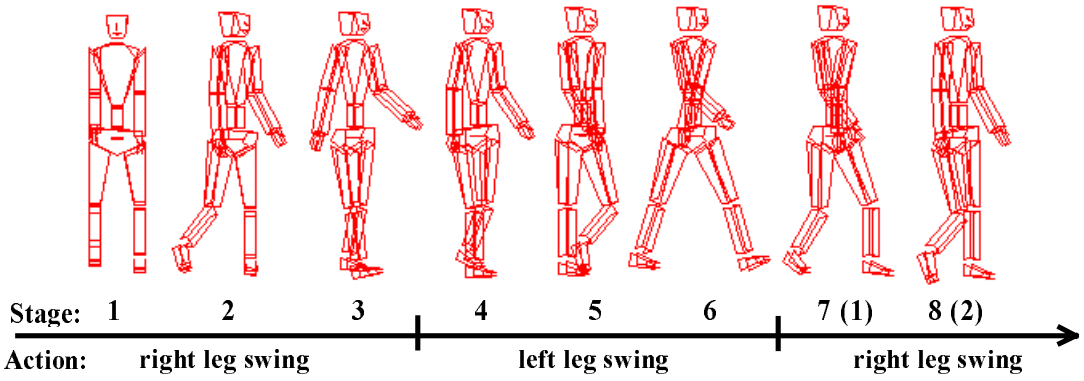


Figure 4-17: Turn 50 degrees to the left, swinging right leg.

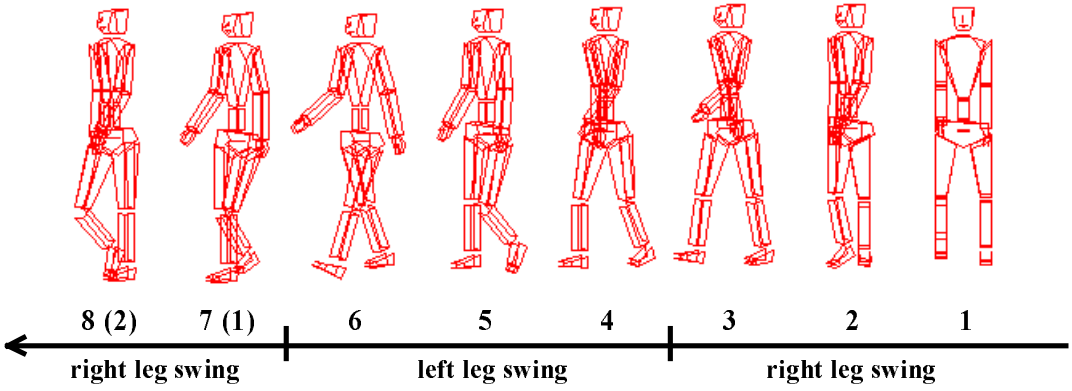


Figure 4-18: Turn 50 degrees to the right, swinging right leg.

First Stage: Leg Lifting and Change of Direction

If a change of direction is required then it is performed in the first stage of a walking phase. The turn basically involves the rotations of the supporting leg and the body of the figure by the same amount but in the opposite direction. The stages (1) and (2) in Figure 4-17 show a perspective view of the right leg swinging forward and the body turned in the left direction. An excerpt from this procedure is shown in Figure 4-19 which is called by the procedure shown in Figure 4-16. Because the motion is implemented in a conventional programming language (C++), code such as "limiting the turning to the maximum of 70 degrees" as well as internal variables (e.g., *body* data structure) can be

included in the procedure in addition to the basic motion structure discussed in section 4.2.6.2.

```
void Skeleton:: walk_right_stage_1(int Start, int Duration, float Turn )
{
        start( Start );
        if (Turn > 70)  Turn = 70;                      // Extra statements to
        else if (Turn < -70)  Turn = -70;               // constrain the turn
        fix( l_foot );                  // Fix the supporting leg.
        bend( body, Y_axis, body->Y.Angle+Turn, Duration );
        bend( l_femur, Y_axis,  0, Duration );
        bend( r_femur, Y_axis, -Turn, Duration );
        ....
}
```

Figure 4-19: Turning the direction of walk.

Second Stage: Step Forward

The length of the stride is controlled in the second stage. A simple analogy is to regard the legs as being like a pair of dividers. Indeed, this is one aspect of the determinants discussed by Inman. Referring to Figure 4-20 the angle between the legs corresponding to the maximum stride could be 44 degrees. However, if the stride is shorter then the angle will be correspondingly less. Therefore, the bending of each leg is limited to $0 < fraction\_of\_stride < 1$ at the femurs' z-DOF. The value of *fraction_of_stride* defines the length of the step.

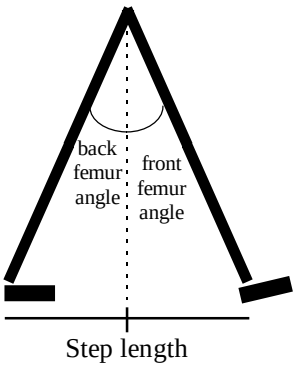


Figure 4-20: Walking stride as a compass gait.

Second Stage: Step Sidewards

A sideways step is similar to the forward step with the sideways motion being performed in the second stage.

Third Stage: Straightening Up

In this stage the movement is consolidated by transferring the weight from one leg to another.

4.3.9 Gestures

There are some motions that are peripheral to the main activity An example is swinging the arms during walking. Although it is a natural component to walking it is not essential and may not even be possible if, for example, the arms are being used to carry something.

4.4 Path Planning

Frequently, in carrying out an action, the animated figure has to move through a room populated with objects and other actors. In order to avoid collisions, a route to reach an intended place should be planned first which the figure can follow. This is the path planning problem which has been studied in robotics [Loza79, Broo83]. There are two classes of trajectory planning in robotics: two- and three-dimensional planning. The kind of path planning typically employed in computer animation is the two-dimensional type.

Two main motion planning approaches that are employed in robotics are the *potential field* and the *configuration space*. The *potential field* uses mechanisms of attraction or repulsion to guide the motion of a robot [Arki87, Wilh90]. In this approach a model of the perceived world is built up as a map of a potential field. Then the robot makes use of a repertoire of sensing strategies to identify the nature of the field to guide its motion through the area (Figure 4-21).

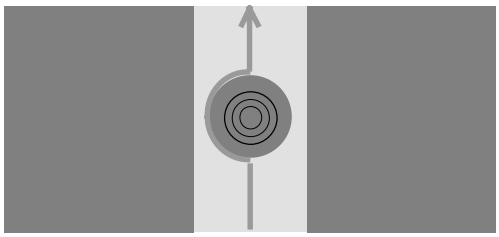


Figure 4-21: Potential field used to guide a robot.

The *configuration space* is most generally used in computer animation [Cavu93, Cher89, Chin92]. Basically the method relies on making a distinction between the occupied and

free areas. Objects are enclosed in bounding boxes which are a simple representation of the area they occupy. Areas occupied by the bounding boxes cannot form part of the path of the robot. An example of *configuration space* is depicted in Figure 4-22 by Cavusoglu [Cavu93]. There the occupied cells are marked as "occupied" making them unavailable and remaining cells are marked with their distance to the destination. Each animated figure will have such a grid appropriate to its destination, and the path followed is obtained by choosing the cells at each stage with the minimum distance to the destination. In a dynamic environment with several actors in motion, grids must be re-computed to reflect the changes of the environment.

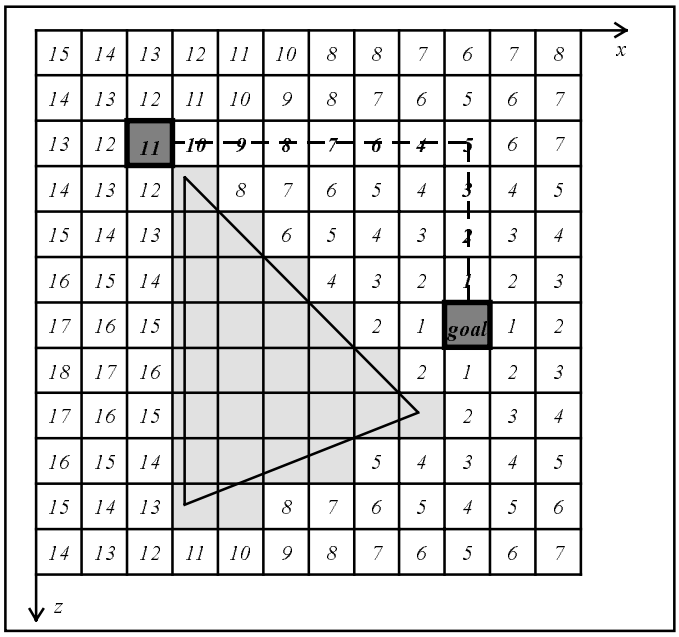
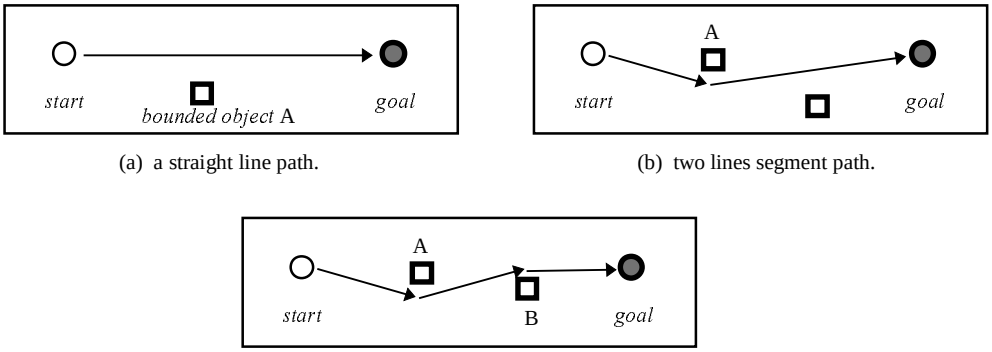


Figure 4-22: Marking the visible cells with distance to the target.

4.4.1 The Segment Sub-division Approach

Although both the above approaches are commonly used in computer animation and robotics, we use the simplest collision avoidance algorithm which has been acknowledged by Lozano-Peres et al. [Loza79] as being a "generate and test" paradigm. It is suitable for a fairly cluttered environment, with many obstacles but not too overcrowded. This algorithm is preferred over the *configuration space* approach because the environment is changing continuously and the determination of the configuration space of the environment for each figure's path can be costly. The method described here combines the use of bounding areas (or volumes) with recursive

subdivision of the path. The simplest path is a straight line from the current position of the actor to the destination. If an object is detected in this path then the algorithm divides the planning problem into two sub-problems, that is, one path that goes from the starting position to a point in the neighbourhood of the obstructing object and another path from this point to the destination. The division is repeated recursively for each segment of the path that intercepts an object. Eventually a list of segments describing a usable path is obtained (Figure 4-23).



(a) a straight line path.

(b) two lines segment path.

(c) three lines segment path.

Figure 4-23: Path determination with obstacles.

In the planning of collision-free paths, the test of potential collisions is greatly simplified as the shapes of objects are approximated to larger bounding circles or rectangles. Small objects such as a chair, a table or an actor are individually bounded by circles; while an object such as a counter is bounded by a larger rectangle. In the case of a small object (Figure 4-24), the obstruction of an actor's path, that goes from point A to B, is avoided by determining an intermediate waypoint M tangent to the circle. This is determined by drawing a line *normal* to the segment AB going through the centre of the obstructing circle. As two points are found, the point closest to the segment is selected as the new waypoint. Obviously other selection criteria might be used depending on the situation.

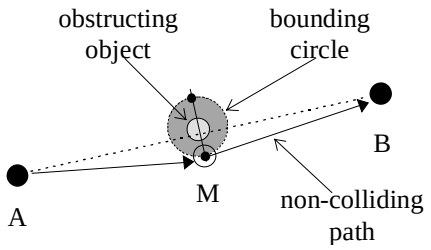


Figure 4-24: Determination of alternative point.

In the case of the rectangle boundary, a similar process is used to determine a waypoint M for the path AB that avoids interception with the object's bounded area. In order to simplify the analysis, the path segment AB and the bounding rectangle are both rotated by an angle that aligns the path segment AB with the origin of the coordinate system (horizontally) as shown in Figure 4-25a. The waypoint M is the one of the two extreme points that lie on the side of the rectangle first crossed by the segment AB. The point lying on the upper or on the lower parallels, tangent to the bounded area, or whichever point is closer to the path segment, is chosen as the new waypoint. In the case of Figure 4-25b further determination of a waypoint N for the sub-path MB is required as it also crosses the bounded area.

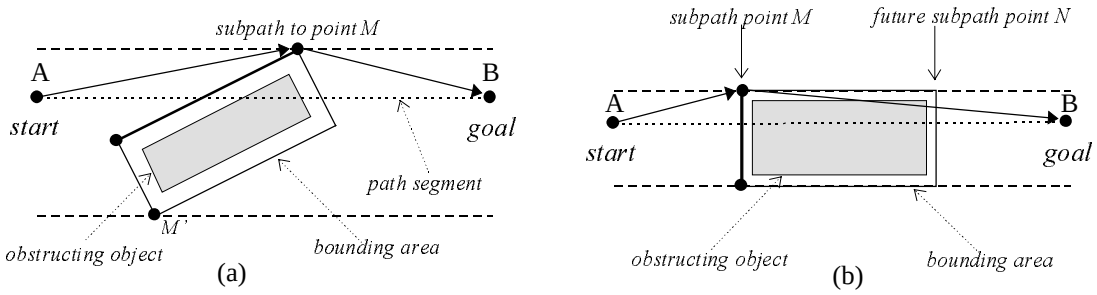


Figure 4-25: Determination of alternative point for a rectangle.

The Prolog implementation of the path planning is:

```
clause 1   find_path( Actor,  pos(Xa, Za),  pos(Xb, Zb),  List ) :-
               test_path_block( Actor,  pos(Xa, Za),  pos(Xb, Zb),  pos(Xc, Zc) ),
               find_path( Actor,  pos(Xa, Za),  pos(Xc, Zc),  Phase1 ),
               find_path( Actor,  pos(Xc, Zc),  pos(Xb, Zb),  Phase2 ),
               append( Phase1,  Phase2,  List ) .

clause 2   find_path( Agent,  pos(Xa, Za),  pos(Xb, Zb),  [pos(Xb, Zb)] ) :-  !.
```

The program *find_path* accepts as input: the actor involved in the planning; the current position of the actor, *pos(Xa,Za)*; and the destination position of the actor, *pos(Xb,Zb)*. As a result a list of points (*List*) specifying the path to the destination position is created, that is, *List* is a solution of path composed by the path until the obstacle (*Phase1*) and after the obstacle (*Phase2*), the obstacle is located at *pos(Xc,Zc)*. The planning procedure takes the following actions:

- The *clause 1* is attempted first. Determine if the path [*pos(Xa,Za)* - *pos(Xb,Zb)*] intercepts any object by evaluating *test_path_block.* One of the following two situations might occur:

* If there is an interception at a point *c* then the segment is split into two at that point, recursively calling path planning for the first segment (*a-c*) and then calling path planning for the second segment (*c-b*). The route to reach the destination *b* is the sequence of waypoints obtained.

* If no interception point was found then abandon *clause 1* and execute *clause 2*. That is, the only waypoint is the destination *b*.

4.5 Conclusions

The basic elements of an animation system have been presented in this chapter. That is, the skeleton and objects data types, and the animation control components have been described. At runtime, new instances of humanoids are created and managed by the Cast controller. The motion of these agents can be described within files as structures of motion command or within a program as procedures. Some examples of motion have been presented, together with a simple and convenient algorithm for finding paths through obstacles.