Chapter 8

The Task Concept

8.1 Introduction

In Chapter 6 the *instruction* concept was presented as a control building block for emulating abstract actions through the use of hierarchies. In Chapter 7 the *message* concept enabled the capabilities of the *instruction* to be extend across multiple agents. Both types of control have thus provided the conceptual structure to co-ordinate actions. In this chapter we present the *task* concept as the control structure that co-ordinates the actual motions performed by the animated agents.

The *task* entity is a conceptual component aimed at controlling well-defined motional activities such as: walking, the positioning of the body, waving a hand, picking up or putting down of objects, etc. It is analogous to the concept of *operator* in the context of (action) planning studied in AI, which has been discussed in the Chapter 3. A similar analogy has been observed in computer animation. Zeltzer [Zelt82] has called these units of motion *local motor programs* (LMP). In our framework such a partition of motions into units is further explored in the composition of general behaviour.

As has been shown in Figures 5-2 and 5-3, the task is the control layer that behaves as an interface between the Controller and the Basic Animation System by sending animation commands. However, the task entity has an important role in the scheduling process as it deals with the problem of resource allocation. This issue is discussed in Chapter 9.

8.2 The Operation of Task Control Entity

The organisation of a *task* entity is similar to that of an *instruction* in terms of the data and control structure, however, functionally they are different. While an instruction is further specified by a sequence of detailed actions, the task is developed into a sequence of movements. Hence, the task entity is not further developed into any other kind of control structure. Additionally, tasks are parameterised entities which receive data from the parent nodes in the planning.

The handling of the task nodes during plan execution is performed by the Task KS which is shown in Figure 8-1. There are three important features to consider in the operation of a task: resource allocation, coordination of the task process in several states (success, failure, interruption, etc.), and the execution of the task in generating motions. With the exception of the latter, all processing is done before the formation of the following animation frame. These three features are inter-dependent, and it is difficult to establish an order of presentation without referring to the others. The detailed operation of the task entity is described in the following sections. Firstly the general task frame is introduced.

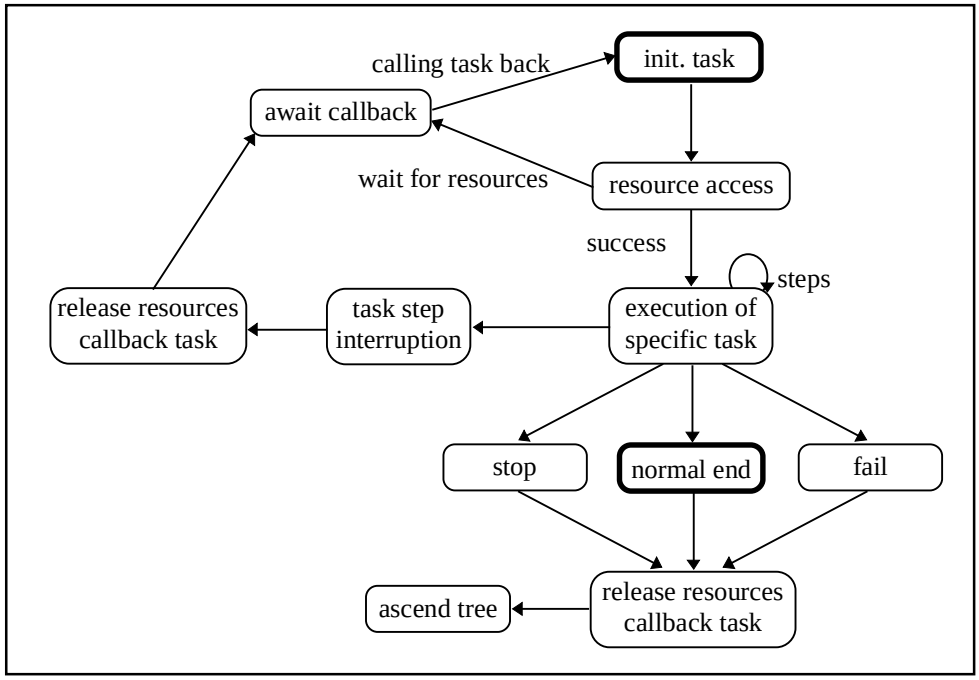

Figure 8-1: Task Controller Scheme.

8.3 The Task Frame

The information required in the operation of a *task* control entity is arranged into a frame structure called *generic task*. Basically it carries slots with the information that is typically used in general motion control and provides storage for internal operations as shown in Figure 8-2. The generic task provides the basis for deriving all the available *tasks* incorporated in the system. For example, a specific task frame, such as the walking activity, is derived from the *generic task* frame and slots related to that specific skill are added to the derived frame. During runtime, instances are created from the tasks frame in order to provide storage for the computed values as well as those values passed down through parameterisation. These instances are subsequently destroyed at the conclusion of their activities. Since tasks are part of the planning process they also point to the corresponding *root* structures which provide them the access to information related to the overall process.

```
frame task ;
  default state      is start_st and              % Initial state.
  default type       is task and
  default ruleset is task_control and             % Single or multiple step motion.
  default param      is nothing and               % Parameters passed.
  default template is nothing and                 % Full list of parameters.
  default parent     is nobody and                % Parent instance.
  default root       is nothing and               % The overall process data structure.
  default start_time  is 0 and                    % Actual start time of the task.
  default final_time  is 0 and                    % Actual conclusion time of the task.
  default resource_allocation is fixed and        % Resource sel. mode:  fixed, selected, or both.
  default resource_list  is nothing and           % List resources used by a task (fixed mode).
  default resource_select is arm and              % Resource used in the selected mode.
  default arm is nothing and                      % Generic parameter: arm.
  default used_resc  is nothing and               % List of all resources currently allocated.
  default category  is primary and                % That is:  primary,secondary, or signal.
  default fork_task is nothing and                % Potential subsequent task.
  default gesture_task is nothing and             % Accompanying gesture task.
  default assoc_task is nothing .                 % List of gestures associated to the task.
```

Figure 8-2: The general task frame.

8.4 Allocation of Resources

A task becomes effective when it can allocate a subset of the existing resources (internal or external to the agent) for its use. The internal resources are the agent's limbs: both

arms, both legs, head, and torso; while the external resources can be glasses, tables, or places at a counter. The specific case of internal resources is of our interest because at least one of the limbs is necessary to achieve motion. There are three modes of resource allocation: *fixed*, *selected*, and *mixed*. In the *fixed* mode a task deals invariably with the same limbs specified in the *resource_list* slot in a particular occasion. For example, the walking task always use both legs for its activity. In the *selected* mode a task knows the kind of limb it is going to use, as indicated in the *resource_select* slot, but the specific resource will become known only at run-time. For example, the *pick_up* task will use the arm specified by the parent instruction which is passed through parameter passing. The *mixed* mode combines both previous modes of resource allocation, that is, one part of the resources is known from the *resource_list* slot and the other part from the *resource_select* slot which has to be determined at run-time. For example, the agent performing the *turn_wave* task always turns the torso but can wave with either arm.

The resources allocated by tasks are recorded in the *resources_used* slot in the agent instance. Conversely, each limb of the agent's instance is characterised by a pair of items: the state (or position) of the limb and the name of the instance of the task that has allocated it. For example, the statement

john's r_leg becomes st(straight, walk_t1)

means that the position of John's right leg has become *straight* and is currently being used by the task instance named *walk_t1*.

The partition of the human figure into six parts or resources permits different motions to be accommodated that may occur simultaneously. While the explicit allocation of resources by tasks avoids the possibility of more than one motion being performed using common parts. Another advantage is that the DOFs of the "physical" body are logically organised into a small number of body parts allowing them to be more easily controlled by tasks. Moreover, most DOFs cannot be operated on in isolation, for example, when the DOFs of a hand are employed to pick up an object obviously the rest of the arm's DOFs are also affected, so it would be overwhelmingly hard to handle each of the arm's DOF individually or ineffectual to deal with the arm as two resources rather than a single one. Therefore, there must be a balance between the number of logical body parts and the degree of control over the DOFs.

8.5 Priority to Access Resources

In a typical dynamic environment there are several tasks that can potentially enter in operation if conditions allow. That is, if within the same agent there is no overlap in the use of resources these tasks could function normally. Nevertheless, overlap in the usage of resources do occur and one way to resolve problem is to include a scheme of priorities for accessing resources. It is obviously desirable to give preference to the most interesting motions. For the time being it suffices to know that motions originated from different sources have different degrees of interest, or priorities, whose rating values have been pre-defined and are assigned to the motion process. Therefore, one motion may have preferred access to resources over others.

Because of the possibility of coexistence of tasks, the availability of the required resources is verified first in order to avoid conflict. If no conflict occurs, then all the required resources are allocated to the task and it can be started. However, in the case of conflict the following action is performed:

- make a list of tasks that are currently using the resources requested by the new task
- compare the priority of the new task with the highest rated task of the list

a) if the new task has a lower or equal priority and it is a "worthwhile task", then the *callback* slots of the old tasks will include the new task as one of the tasks to be called back at the end of their activities. A task is considered "worthwhile" if it is a non-gesture motion, that is, gesture motions are not essential movements though important. This is discussed in the next chapter.

b) if the new task has a higher priority than those in activity then the state of these old tasks are set to the *interrupt* state, the *callback* slots of the old tasks will include the new task and the *callback* slot of the new task will include the old tasks instances names. The interruption state is discussed in the next sections.

8.6 Calling Processes Back to Activity

In the previous section we have seen that tasks are put in the awaiting condition when the required resources are being used by one or more active tasks. The awaiting tasks

are called back to be re-evaluated when one of the active tasks releases its control over them. Such a protocol is needed because an engaged motion cannot be simply interrupted at any time. Also because of the possibility of more than one task requiring control of common resources, all the waiting tasks are put in one callback list of the running process, so that they are called back when the process releases the resources. An important aspect is that they are called back in decreasing order of priority. The process of calling back instances of control does not apply only to tasks, but also to instructions, and more precisely it is applied to the whole process which is identified by the root instance. The process gains activity and will thus reactivate the instance recorded in the *current_instance* slot in the root instance, which is the point where the process had previously stopped..

The callback scheme, although simple, is useful for dealing with situations with multiple changes of activities. For example, Figure 8-3a illustrates the case of two active tasks, possibly with different start times and durations, being interrupted by a new task of a higher priority, the first active task to conclude its motion calls the new task back for re-evaluation and stays "out of focus", that is, it now waits to be called back by the new task. The new task is thus reconsidered for resource allocation. In the case that there is still a (second) active task using part of the required resources, the callback procedure is re-applied, that is, the new task is included into callback list of the (second) active task, however, if a process is already in the callback list of another, the inclusion is not made. Finally, at the conclusion of the second active motion, the new task is re-evaluated for a second time, now with success. At the conclusion of this (ex-new) task, the suspended tasks are then called back, whose success in the re-evaluation will depend on whether no higher rated task has arrived in the meantime. The Figure 8-3b exemplifies the second case of a new task having less priority, therefore it waits to be called back when the current tasks are completed. If, however, the executing tasks are interrupted by a higher priority task they will enter into operation in the next opportunity and then call the pending task when they terminate.

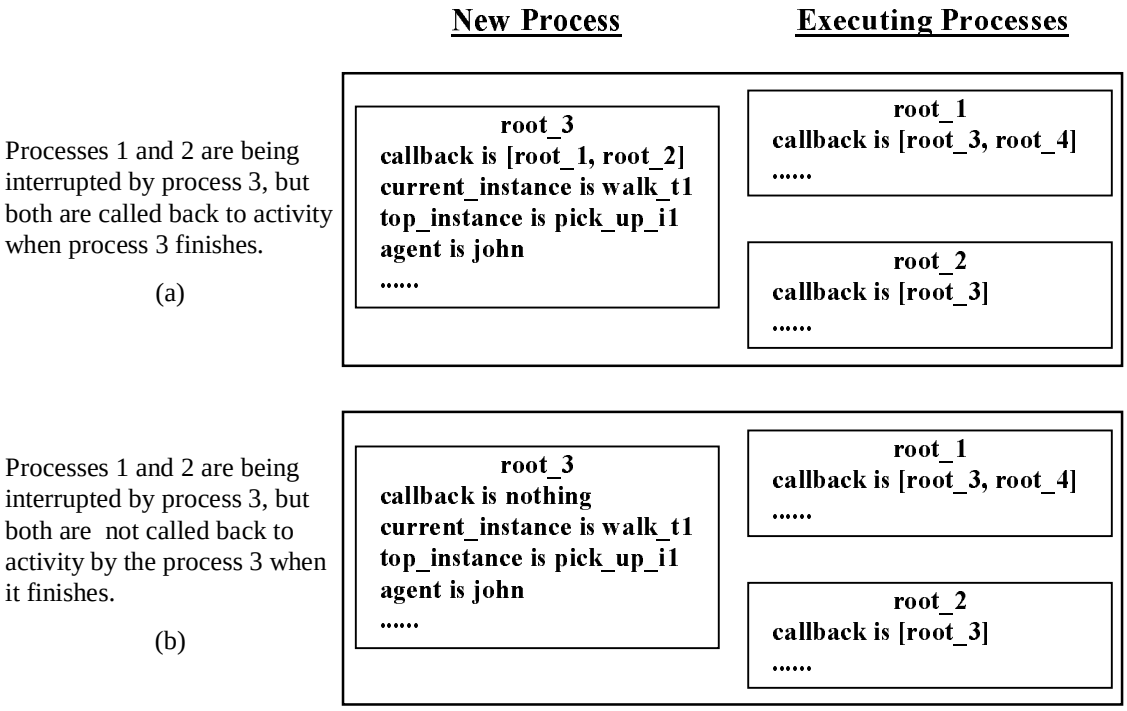

Figure 8-3: Process callback scheme.

8.7 The Task Specific Process

The purpose of a task entity is to provide the schedule of movements to the Basic Animation System (BAS) whose motions are animated at tick rate. This part of the task operation called *task specific* is implemented as a procedure, or in the form of rules, and sends messages to the BAS. The BAS will decompose the messages and call the specified functions which generates the motions. The complexity of organising the schedule of these messages depends on the implemented motion. Most of the motions performed by agents are typically single motions[4] and a high-level ones such as a turn of the head, a gesture with an arm, a movement for picking up a glass, etc. These motions are usually of short duration which can be realised in one uninterrupted step. However, extended tasks like the continuous wave of an arm or the walking motion are movements repeated a variable number of times. In order to avoid monopolising resources for long times and to allow that changes in the environment be taken into consideration at frequent intervals, long movements are divided into fragments called *steps*. An individual

[4] These are the BAS motions discussed in the Chapter 4.

step of a movement is thus executed without interruption and the length of a *step* should therefore be reasonably short. Thus, a possible collision with another character can be avoided between one step and the next by taking an alternative action at the conclusion of the current *step*. Figure 8-4 shows an example of the schedule of a *step* of motion. The contents of the actual message sent to the BAS is contained in the third parameter of the *send* command. Similar to the *Add List* of a typical Strips operator in section 3.3, the <*post operations*> is a list of "updates" which are kept in the blackboard and, when the corresponding motion has finished, the BAS sends a message back to the controller to realise the updates in the database. This operation is illustrated in Figure 8-5. The details of each stage of the motion operation are presented in the following sections.

```
send( <agent>,  <instance>,  <message to BAS>,  <post operations> )
```

```
send( Agent, Instance,
      cmd5( arm_reach_goal, Arm, X, Y, Z ),
      { attr(Agent, Arm, extended),   attr(Object,state,held),      attr(Object,holder,Agent),
        attr(Agent, Hold, Object),    attr(Object,place,Agent),    exclude(Object, Place),
        exec( p_hold_object( Agent, Arm, Object )),    exec( fork_new( Instance, Agent) ) } )
```

Figure 8-4: Message call to BAS.

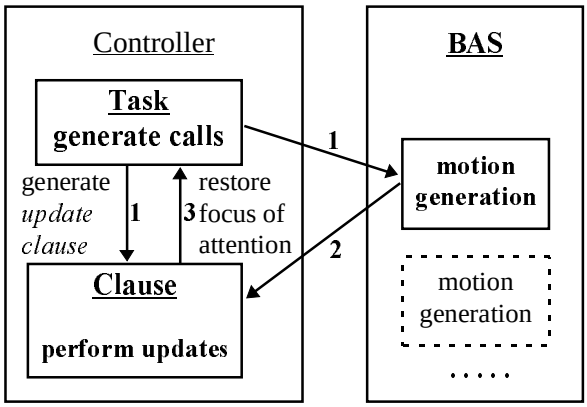

Figure 8-5: Scheme

8.7.1 Data Retrieval and Checking Constraints

Before any motions calls are sent to the BAS, the task procedure retrieves some data and checks if the attributes or relationships are the expected ones. If no problem occurs the messages are composed with the retrieved data and then sent to the BAS. The task temporarily loses the focus of attention of the Task KS until the requested movement of

the specified *step* is completed. The state of a task is pre-set to the *end* state, so that in the case of a task be made up of a single *step* it is handled by the finalisation procedure as it regain the focus of attention of the Task KS. Otherwise, the task will be handled by the designated procedure associated with the state set in the task which is one of the states specific to that task. This will be made clear in the next section. If, however, a constraint has failed a test, then the state of the task is set to the *fail* state or any other state customised for that task which handles the problem.

8.7.2 Associated Actions

The associated actions are motions represented by tasks, or instructions, which are launched (or scheduled) in the *task specific* procedure along with the main motion. Despite this they are indeed independent processes which complement the main motion. These motions are typically of little importance and remain as "potentially active" motions which are enabled only when the required resources are free. These motions are typically *gestures,* the purpose of which is to accompany the main motions. This will be made clear in the next chapter. For example, the action of swinging of the arms is complementary to the walking motion, but the arms are not essential to the locomotion activity. The action is thus stopped if any other task requires its resources. As this kind of action is associated with a main motion, it is also deactivated if the correspondent main motion is deactivated.

8.7.3 Database Update

As a result of the performance of each *step* of a movement, the animation database is updated with new facts about the objects. These updates serve as conditions to guide subsequent actions or as the constraints to be tested by other tasks. The informal examples are: in a plan selection, "*if the agent's position is seated then select a plan that includes the standing up action*", "*if the counter's front_left is engaged then check other sides*", and "*if the counter's object_list is not empty then collect an object*"; in the walking task, "*if the left_leg is forward, then the next step will be to swing the right_leg forward*".

The operation of updating is performed according to the clauses used: ***attr***, ***include***, or ***exclude***, as illustrated in Figure 8-4. For example, assignments (***attr***) update the

attributes: the positional state of the limbs (legs becomes *straight*), the posture of an agent (*sitting*), the location of the object (*the tall_glass is on_the_table*), etc. Also ***include*** and ***exclude*** update objects in a list, for example, *bar_counter* has [*tall_glass, glass2*]), etc.

The database update also provides the opportunity to effect more elaborate "updates" in the form of a call to procedures (***exec*** clauses*)*. That is, facts are not only used to update database they are also used as clause to call procedures specific to the effected *step*. As shown in Figure 8-4, the specific operation

exec(p_hold_object(Agent, Arm, Object))

whose procedure, *p_hold_object,* tells the BAS that the *Object* becomes "attached" to the *Agent's Arm*, so that the Object coordinate system becomes relative to the arm rather than to the world. The other procedure operation in the update list is a general one

exec(fork_new(Instance, Agent))

fork_new schedules an instruction which acts as a "backup" action. That is, it is scheduled to be a potential execution in the case that no proper continuation is scheduled to run after the performance of the main motion task. For example, if a plan prescribes the following sequence of actions "*picking up a glass* and *walking to the bar counter*", if no action operates on the arm which has just held a glass, except the swinging arms of the walking activity, then the glass would simply swing along with the arm during the walking. For such cases, the action of "hold the object" would be more appropriate than swinging the arm, specially if the glass contains liquid. This example is discussed in more detail in the next chapter.

8.8 Task Control States

The Task KS monitors the operation of the task control entities by applying the procedures of their current states which can be *start*, *end*, *stop*, *fail,* and *interrupt*. In addition to these, there is the possibility of merging them with states related to the control specific to the task. This occurs when a task is implemented to deal with multiple *step* motions (e.g., the walking task) and, as such, intermediate control states

might be necessary. By default, all tasks are set to the *start* state through which they attempt to become effective by gaining control of the resources. Once the tasks have the control of the resources, they schedule motions to the BAS and they become momentarily "out of focus" of the attention of the Task KS as long as the motion is performed. When these motions have been completed in the BAS, the tasks are returned back to "focus" and, as most of the tasks are a single motion, they are normally processed by the finalisation procedure which is identified by the *end* state. In the case of multiple *step* motion, before being processed by the finalisation procedure, the tasks are processed by procedures associated to the states set in the instance which are likely those specific to that task. In normal conditions, the tasks go to the *end* state where they release the resources, then they call the pending tasks or instructions back, if any, for re-evaluation, and finally the tasks are finished and the control of the processes are passed back to the parent instances.

```
ruleset task_control contains task_start, task_end, task_stop, task_fail, task_interrupt .

rule task_start
if task T is on focus and T's state is start  then  do task_initialisation on T.

rule task_end
if task T is on focus and T's state is end  then  do task_finalisation on T.

rule task_stop
if task T is on focus and T's state is stop  then  do stop_process on T.

rule task_fail
if task T is on focus and T's state is fail  then  do handle_failure on T.

rule task_interrupt
if task T is on focus and T's state is interrupt then  do suspend_task on T.
```

Figure 8-6: The Task KS.

In the case that the control of the task is set to the *stop* state or to the *fail* state, the corresponding procedures are similar to the *end* state, but with some differences. In the *stop* state the whole process is simply terminated. In the *fail* state, the control is handed back to the parent instance to deal with the failing condition.

The *interrupt* state is reached by tasks of multiple stage motions that have temporarily given up their control over the resources to other tasks. Their states are re-set to the

start state and they become "out of focus". Thus, when they are eventually called back they will be re-evaluated for resource allocation.

8.9 Example

The *drink_t* is an example of a simple task whose procedure is described in Figure 8-7. This procedure is executed in the *start* state as a function call. In the procedure the initial information can be retrieved from the task instance and the agent instance. The constraints are checked, in this case, to determine if the glass is empty. Should a problem occur, the state of the task is set to the appropriate error handling state, which in this case is the *fail* state, and control exits from the procedure *drink_t*. In the normal case of one or more motions commands ***send*** can be composed and scheduled to run in the BAS. The *drink_t* task is shown schematically in Figure 8-8a where the control of the task is composed exclusively by the procedures associated to states of the Task Controller. The *walk* task has additional states which are merged with the task control as shown in Figure 8-8b.

```
relation drink_t( Agent, Instance ) if
  lookup( drink_hand , Instance, Hand ) and
  lookup( drink, Instance, Glass ) and
  lookup( volume, Glass, Volume ) and

  if  Volume is 0                                    // The glass is empty.
  then
     Instance`s state becomes state_failed and       // Go to fail handling state.
     ! and fail
  else
     p_compute( New_Vol, Volume - 1 ) and

     send( Agent, Instance, cmd2(drink, Hand),              // Motion command sent to BAS.
           {attr(Glass, volume, New_Vol), attr( Glass, condition, used )} )     // Update info on return.
 end if .
```

Figure 8-7: The drink task procedure.

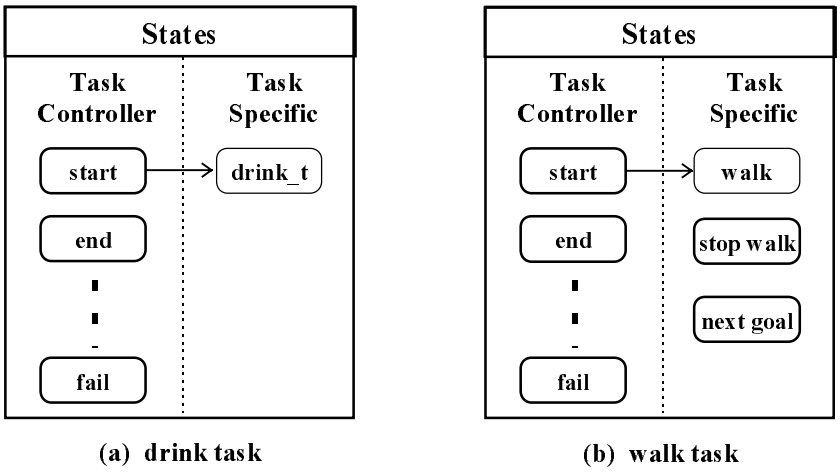

Figure 8-8: Blending states for task control.

A more realistic impression of the interaction between the Task layer and the BAS is given in Figure 8-9. Each *send* command gives raise to a motion structure in the BAS as discussed in section 4.3.4.

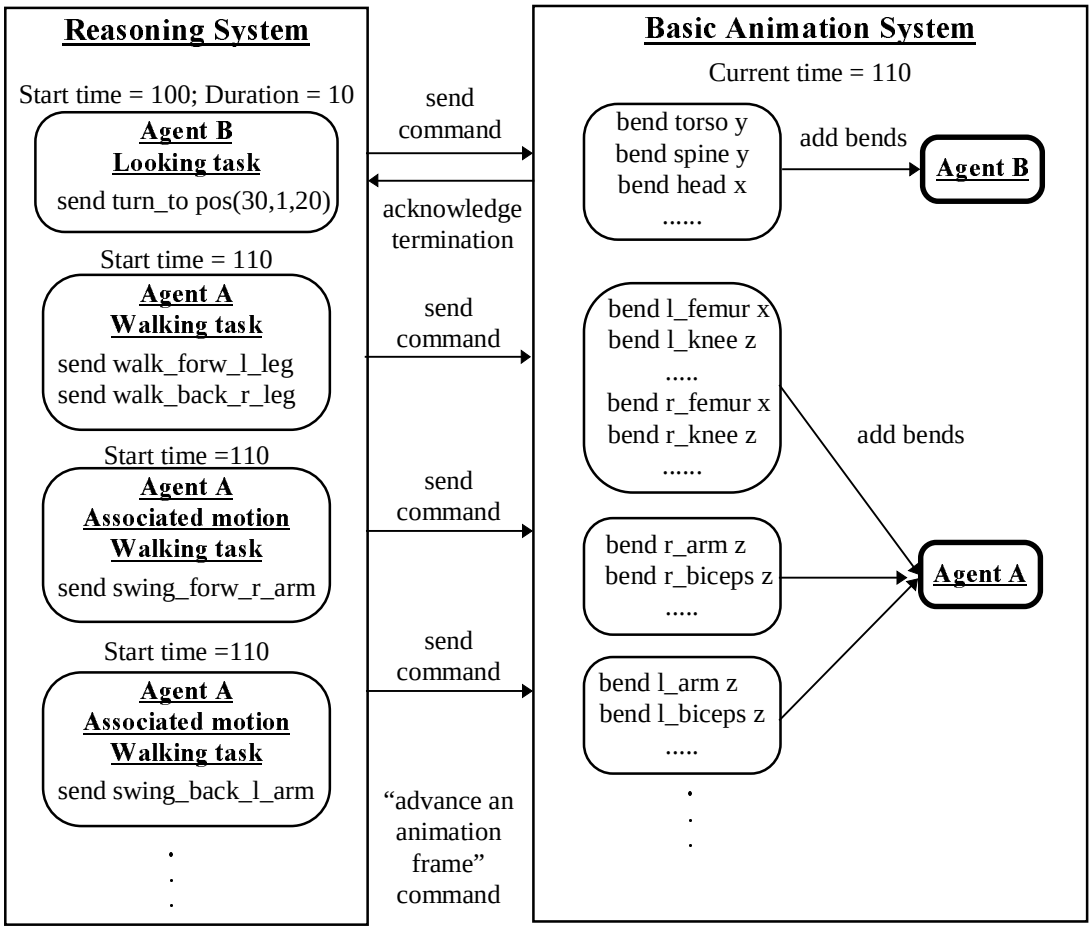

Figure 8-9: Scheduling motion commands to the animation system.

8.10 Summary

Task entities control the actual motions performed by agents and motions are achieved if the required resources are available. Since many potentially active tasks may require common resources, the concept of explicit allocation of resources and the selective execution of tasks using priority testing are effective in co-ordinating contending actions. Active tasks give up the control over resources in favour of higher priority tasks, but this is not always done in the same animation frame, so the scheme of calling "waiting tasks" back to activity has been devised.

The regular updating of the animation database is essential at the completion of each step of movement, so that subsequent actions can yield an adequate continuity of movements. Another important feature is that potential actions parallel or subsequent to the main motions can conveniently complement the main motion.