

CILIA: A Programmable, Task-Level Planning, 2-D Simulation Software for Arrays of Manipulators¹

Murilo G. Coutinho, Peter M. Will, IEEE Members,
USC Information Sciences Institute
Marina del Rey, CA
(310)-822-1511
coutinho@isi.edu

***Abstract:** In this paper we describe the CILIA software package, a programmable, task-level planning, 2-D simulator of arrays of manipulators. CILIA is a design tool that simulates the dynamic interaction of objects placed on the array and its manipulators, by assuming that the manipulators induce a continuous vector force field on the objects. CILIA basic features are described, such as its built-in CAD tool system (that allows the user to draw 2-D objects and force field windows of general complexity), its script editor (used for task-level planning and programming), its client/server architecture, among others. The simulation engine and the impulse-based collision model are also described in depth. In order to generate realistic and accurate simulations, friction between objects being manipulated and the array surface was taken into account in the simulation engine. The CILIA software decouples the design of applications requiring the use of manipulator arrays from the actual devices used to implement them, therefore offering a level of abstraction to the applications designer, who can implement a set of motion plans independent of the array device actually being used. The CILIA software package is available on-line. For more information and download instructions, please, visit the project's site at <http://www.isi.edu/mass>.*

1. Introduction.

In the past few years, the use of arrays of manipulators has been intensively studied as an alternative for automatic parts assembly on both macro and micro scale [6, 7, 8]. The array is usually built from simple manipulators, each capable of moving in one single direction. The individual manipulators can be implemented in a variety of technologies with sizes ranging from a few tens of microns in Micro-Electro-Mechanical Structures (MEMS), to a few centimeters, in extensions of the technology used in baggage carousels in airports, sorting of parcels in post offices, or conveyors in factories.

The common necessary characteristic for the array is to be programmable, that is, the manipulators should be individually addressable or addressable by groups. The individual manipulators are usually arranged in groups of four, each moving along one cardinal direction, and the group is replicated over the entire array [2]. The array, with contiguous

¹ This work is sponsored by DARPA under Fort Huachuca Contract No. DABT-63-92-C-0052.

manipulators acting in unison, can be programmed to have the effect of inducing force fields on objects placed on it. The appropriate choice of force fields can move objects in manners that are suitable for mechanical assembly operations such as translation, rotation, orientation, alignment, sorting, parts feeder, and spatial filtering, among others [3, 4].

In this paper we describe the CILIA software package, a task-level programmable, 2-D simulator of arrays of manipulators. CILIA simulates the dynamic interaction between objects placed on top of the array and its manipulators, using built-in dynamic models based on rigid body and fluid dynamics, and results from practical experiments with Intelligent Motion Surfaces (IMS) [1-5]. It assumes that the manipulators induce a continuous vector force field on the objects being manipulated, that is, the current modeling and analysis does not consider the individual, discrete interaction between each manipulator and the objects. Even though this simplified model may seem to limit the use of the CILIA software, nonetheless, it does give the user enough flexibility to simulate complex systems.

The CILIA software decouples the design of applications requiring the use of arrays of manipulators from the actual devices used to implement them, and it can be used to simulate manipulation and assembly tasks on both macro and micro manipulator array devices.

2. Software Overview.

The CILIA software consists of:

- A Force Field Editor (see Figure 1) and an Object Editor (see Figure 2) to construct force field windows and objects of general polygonal complexity, respectively;
- A built-in library of geometric Boolean operations used by both editors;
- A Script Editor for task-level programming;
- Unlimited Undo/Redo functionality;
- Flexible Save/Load operations: the user can save force fields only, objects only, or a complete configuration including both;
- A Simulation Engine that implements a rigid body dynamics model to simulate the motion plans involving the interaction of objects placed on the array with force field windows defined. The graphical output of the simulator is shown on CILIA's Main Window (see Figure 3);
- An impulse-based collision model;
- User modifiable Simulation Parameters, including: static and dynamic friction coefficients, coefficient of restitution, simulation time step, among others;
- A comprehensive on-line Help tutorial;

The CILIA software comes with a built-in CAD tool that allows the user to draw 2-D objects of general complexity. Force field windows can also be graphically defined using the Force Field Editor. However, objects and fields are limited to polygonal shapes. Polygons

might have an unlimited number of holes, and can be grouped forming a hierarchy of polygon structures.

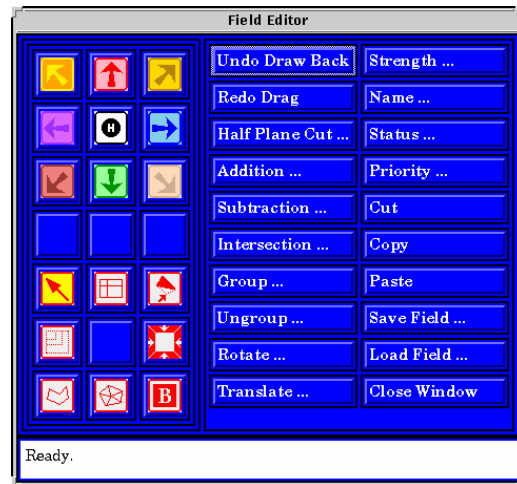


Figure 1: The Force Field Editor with a rich set of functionality to manipulate and edit force field windows of general complexity.

CILIA can be executed in two different modes: standalone or client/server. The standalone mode was implemented for the Sun platform running the Sun OS operating system (actually, there are versions for the Sun OS 4.1.3 and Solaris 2.5). It corresponds to the situation where the entire program is executed on a single machine. The client/server is used over the network. The server program was also implemented for the Sun OS platform, whereas the client program has two versions: a Motif version for the Sun OS platform (machine dependent, used mainly for development and testing purposes) and a Java applet version (machine independent, accessible from within any Java enabled Web browser). The Java applet has the same functionality as the Motif version, but because of the Java language security requirements, some of its functionality have had to be disabled. For instance, save/load capabilities are disabled in the client Java applet. We plan to release upgrades of the client/server version on a timely basis, based on our own research developments and user feedback comments and suggestions. For more information and download instructions, please, visit the project's web page at <http://www.isi.edu/mass>.

3. Implementation Details.

Basically, the client/server version of the CILIA software is written in the C language and runs to about 100,000 lines of code at this stage. The client program, with all GUI components, accounts for 30% of the program's size. As mentioned in the previous section, the client was implemented in two flavors: the Sun OS Motif based version and the Java version (as an applet).



Figure 2: The Object Editor with some sample objects being displayed. The user can define as many objects as desired. Objects are added to the internal database and can be hidden from the user to avoid excessive crowding of the canvas area. Hidden objects have their display status changed to *inactive* using the function “Status ...”. The function “View Objects” shows all objects defined in the database with their respective display status. The user must use this function to select *inactive* objects.

CILIA implements its client/server using the *thin client* model, where the core functionality is kept on the server program. This model assumes the client machine to be simple and with almost no resources (easy maintainability). On the other hand, the model assumes a powerful host running the server program and a fast network connection. This model is most suitable for Intranets. Practical experiments revealed that in this model, the network latency plays a big role on the overall program’s performance. This is due to the fact that in the thin client model the bulk of the computations is performed on the server side. For instance, each GUI event on the client side generates a request that is sent to the server through the network connection. The request is then processed on the server and the updated information is sent back to the client. Add to that the overhead of checking for corrupted messages! It is clear that without a fast network connection, the program’s performance will go to its knee. A way to overcome this problem would be to rewrite the server program in Java, and run CILIA as a standalone Java program on the client machine. We would choose Java as the implementation language due to its portability.

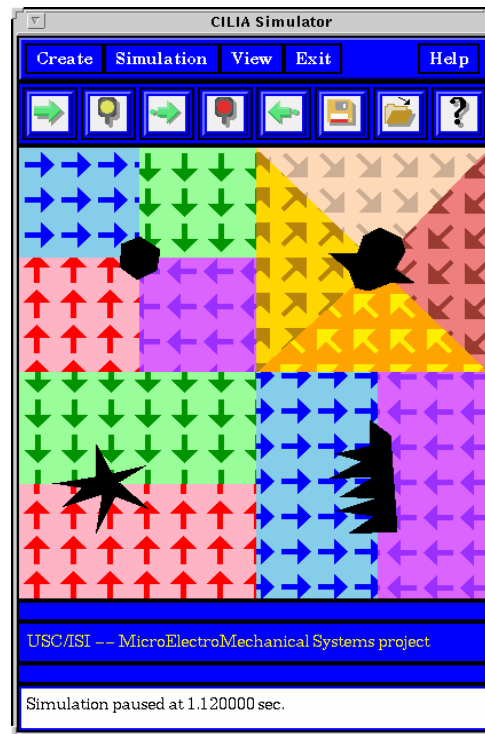


Figure 3: The Main Window with some concurrent simulations paused. In the first (top) two cases the objects are being rotated around the center of the force field windows. In the last (bottom) two cases the objects are being centered along the center line of the force field windows.

3.1. Basic Functionality.

CILIA comes with a system library that implements several complex geometric manipulation tasks. On both standalone and client/server version, CILIA includes both Force/Velocity Field editor and the Object editor. The user is able to draw, group, ungroup, cut, copy, paste, intersect, add, subtract, half plane cut, rotate, translate, change force field strength, classify the fields according to their priority levels, change priority levels, change the display status (making the force field windows visible or invisible to the user), save, load and edit, among the default set of editing functionality that is provided (see Figures 1 and 2). All geometric Boolean operations implemented can handle general polygons with an unlimited number of holes and vertices (as long as there is RAM memory available!). If the system runs out of memory, CILIA frees the memory allocated for the Undo/Redo structures and tries to perform the memory allocation operation again. If it fails again, CILIA outputs an out-of-memory error message and exit program execution.

CILIA provides other more advanced features, such as unlimited undo/redo for both Field and Object editors, functions to change the simulation sampling time (used by the simulation engine), functions to change both static and dynamic friction coefficients between the array

surface and the objects being manipulated, and functions to change the density of the objects in order to allow the design of objects with the same size and shape but made of different materials. Multiple parts can be tested at the same time, and force field windows can be dynamically created, modified or destroyed in order to achieve complex tasks, such as moving a part among several moving obstacles. The latter will be a very common situation in a real assembly line, where we want to optimally minimize the array's power consumption and at the same time maximize the use of the array's surface by as many objects as possible.

3.2. The Simulation Engine.

As mentioned earlier, the simulation engine assumes the manipulators induce a continuous vector force field on the objects being manipulated. Therefore, it computes the net force acting on each object placed on the array by intersecting them with each force field window. The area of the intersecting polygons is then multiplied by the associated force field strength density, giving the total force contribution of each force field window, to the object's dynamics. The total force acting on the center of mass (CM) of each intersecting polygon is translated to the object's CM as a force and torque pair. The net force and torque are obtained after subtracting the friction force. The intersection computation is speeded up by first checking if the bounding boxes of the polygons defining the object and each force field window intersect. If so, we proceed and compute their intersection (if any). In this case, we improved the polygon intersection performance even more by doing bounding box checks before intersecting the polygon's edges. Edges with non-intersecting bounding boxes are ignored during the computation.

3.3. Modeling the friction.

The friction between the array surface and the objects being manipulated is computed using an hybrid friction model, based on both the object's weight and instantaneous velocity. Practical experiments indicate that objects moving on force fields behave like parts moving on fluids, with the friction force being proportional to the velocity of the part [12]. In this case the friction force is computed as:

$$f = -Kv \quad (\text{eq. 1})$$

where v is the object's velocity and K is an arbitrary constant of proportionality (we managed to produce realistic results using $K = 5$.) However, for small velocities, the friction force is proportional to the weight of the part. This can be noticed if we place an object on the array and try to move it using a small force. It is clear that the force will move or not the object, depending on its weight and static friction coefficient, and not on its velocity (which is zero in this particular example.) In this case, the friction force is computed as:

$$f = m Mg \quad (\text{eq. 2})$$

where μ is the friction coefficient, M is the object's mass and g is the gravity acceleration. The friction coefficient used can be either static or dynamic, depending on whether the object is moving or not.

The simulation engine computes what would be the friction force on both cases, that is, the friction proportional to the weight and the friction proportional to the velocity, and then chooses the maximum value between them. Doing so, we don't need to assign a constant threshold to switch between friction force models, and we can realistic simulate the friction force acting on object's of different weights moving at different speeds. The most difficult part was to compute the reduction of the angular momentum due to the friction force. We called it the "friction torque". For instance, an object with pure rotation (no translation) should have its angular velocity being reduced until it comes to a full stop. The derivation of an approximate solution for this problem will be published elsewhere.

3.4. Numerically solving the Equations of Motion.

The CILIA's simulator engine implements the simulation dynamics using a motion model based on rigid body and fluid dynamics [11]. Once we have the net force and torque acting on the object (explained in section 3.2), we numerically solve the differential equations of motion, given by:

$$F = Ma, \quad a = \frac{dv}{dt}, \quad v = \frac{ds}{dt} \tag{eq. 3}$$

$$T = I\alpha, \quad \alpha = \frac{dw}{dt}, \quad w = \frac{dq}{dt}$$

where F , T , a , v , α and w are the net force, net torque, linear acceleration, linear velocity, angular acceleration and angular velocity, respectively. Initially, we did some simulation experiments using Euler's method to numerically solve the above differential equations. Although Euler's method was fast, we managed to generate some unstable simulations, proving it was not suitable for our needs. The main reason is that Euler's method uses a constant integration time step, and the numerical errors accumulated on complex simulations can induce unstable simulations. In order to overcome this problem, we decided to use the more time consuming, but robust numerical method of Kutta-Merson (also known as 5th order Runge-Kutta.) The method consists of computing intermediate values, besides computing the initial and final values when solving the differential equations for each integration time step. The 5th order Runge-Kutta method is summarized as follows. Given a differential equation $\dot{y} = f(y, t)$, its solution y_n, t_n at time step n , and the integration time step $t_{n+1} - t_n = h$, the solution at time step $(n+1)$ is computed using five intermediate steps, as follows:

$$\begin{aligned}
(y_n)_1 &= y_n + \frac{1}{3}h f(y_n, t_n) \\
(y_n)_2 &= y_n + \frac{1}{6}h f(y_n, t_n) + \frac{1}{6}h f((y_n)_1, t_n + \frac{1}{3}h) \\
(y_n)_3 &= y_n + \frac{1}{8}h f(y_n, t_n) + \frac{3}{8}h f((y_n)_2, t_n + \frac{1}{3}h) \\
(y_n)_4 &= y_n + \frac{1}{6}h f(y_n, t_n) - \frac{3}{2}h f((y_n)_2, t_n + \frac{1}{3}h) + 2h f((y_n)_3, t_n + \frac{1}{2}h) \\
(y_n)_5 &= y_n + \frac{1}{6}h f(y_n, t_n) + \frac{3}{2}h f((y_n)_3, t_n + \frac{1}{2}h) + \frac{1}{6}h f((y_n)_4, t_n + h)
\end{aligned} \tag{eq. 4}$$

The solution at time step $(n+1)$ is finally given by:

$$y_{n+1} = (y_n)_5 \tag{eq. 5}$$

The numerical error on each time step is given by:

$$\frac{1}{5}|(y_n)_4 - (y_n)_5| \tag{eq. 6}$$

If this error is greater than a pre-defined threshold, the integration time is divided by 2, and the computations are repeated for the corresponding animation time step. Also, if the numerical error is smaller than another (smaller) threshold, the integration time step is doubled to speed up the computations on the next animation time step. Note, however, that the maximum possible integration time step is limited by the animation time step value. Dynamically adjusting the integration time step, we are able to adjust the performance of the simulator with respect to the complexity of the system being simulated. Therefore, with the Kutta-Merson method, we managed to use a variable integration time step, keeping the animation time step constant².

3.5. The Impulse-based Collision model.

The simulation of the dynamic interaction among rigid bodies being moved by the array takes into account various physical characteristics such as elasticity, friction, mass and moment of inertia. Collisions among objects moving on the surface are modeled using impulse-based techniques [9, 10]. Consider two rigid bodies colliding during an animation time step (see Figure 4.)

² The integration time is used to numerically solve the differential equations, whereas the animation time (also referred to as sampling time) is the time elapsed between each screen update of the simulation.

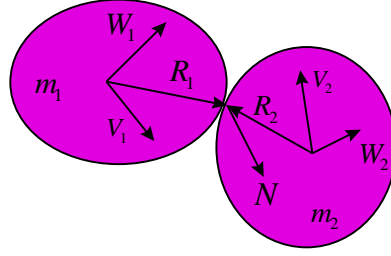


Figure 4: Framework analysis of two rigid bodies colliding.

where N is the normal to the tangent surface of contact between the bodies, and m_i, W_i, V_i, R_i are the mass, angular velocity, linear velocity and distance from the object's CM to the collision point, respectively, for object $i \in \{1,2\}$. On impact, the objects act on each other with impulse P . The conservation of linear and angular moments give:

$$m_1(V_1^* - V_1) = -P, \quad I_1(W_1^* - W_1) = R_1 \times (-P), \quad (\text{eq. 7})$$

$$m_2(V_2^* - V_2) = +P, \quad I_2(W_2^* - W_2) = R_2 \times (+P)$$

where the subscripts stand for each of the bodies and the asterisk denotes the quantities after the impact. The (empirical) generalized Newton's law gives another relation:

$$\frac{(V_1^* + W_1^* \times R_1) \cdot N - (V_2^* + W_2^* \times R_2) \cdot N}{(V_1 + W_1 \times R_1) \cdot N - (V_2 + W_2 \times R_2) \cdot N} = -\epsilon \quad (\text{eq. 8})$$

where ϵ is the coefficient of restitution. Coulomb's law states the relation between the normal and tangential forces at the moment of the impact:

$$|F_t| \leq \mu F_n \quad (\text{eq. 9})$$

where the equality holds if the two bodies are moving tangentially relative to each other at the point of contact. Initially, we assume the two bodies do not slip at the point of contact. We have:

$$[(V_1^* + W_1^* \times R_1) - (V_2^* + W_2^* \times R_2)]_t = 0 \quad (\text{eq.10})$$

In the 2-D case, the above equations give us 8 independent equations for 8 unknowns (P and the linear and angular velocities of each object after the impact). After solving the equations, we have to make sure the no slip assumption is valid, by checking if:

$$|N \times (P \times N)| < \mu |P \cdot N| \quad (\text{eq.11})$$

If the above no slip condition is not satisfied, the two bodies are sliding at the collision point, and in this case we need to re-compute the solution using $P_t = 0$ (no impulse along the tangent direction of the collision). In the case multiple collisions are detected during an animation time step, the simulation engine goes back in time to just before the first (most recent) collision and applies the collision model computation to it. Having updated the new velocities after the collision, it proceeds simulating the system until it completes the current animation time step. Doing so, we can accurately deal with the occurrence of several collisions during an animation time step, taking into account the effects of the most recent collision on the development of the system dynamics. Remember that the display will only be updated by the end of the animation time step and not after each collision is computed!

3.6. Task-Level Programming.

CILIA gives the user the flexibility of performing task-level programming using scripts. Each script consists of an ordered sequence of frames and their associated execution times. A frame is a static configuration of force field windows. The execution time defines the time interval in which the frame will be used to drive the objects placed on the array. In other words, the force field windows defined in the frame will be used to compute the total force/torque acting on the objects placed on the array. After the frame's time interval has elapsed, CILIA changes to the next available frame. This allows the user to implement manipulation strategies by programming sequences of force field windows that will interact with the objects for a limited amount of time. Figure 5 shows the Script Editor window with five frames already defined. The user can update, add, delete, save or load frames, and change the frame's execution time interval.

4. Simulation Example.

In this section we present a simulation example of sorting parts using CILIA's Script Editor. We assume parts of different shapes are being placed on the array surface at a pre-defined position and we want to select one of them at a given orientation, and reject the others. For the sake of simplicity, let's consider the simple case where we are sorting rectangular parts out of a set composed of rectangular and triangular parts only. Figure 6 shows an example of a script that sorts rectangular parts with a given orientation (due to space limitations, we are showing only the main canvas part of CILIA's Main Window for each script frame). The basic idea is to use a hole to sort the rectangular part at the desired orientation. All parts that do not completely fit inside the hole are moved away by the surrounding force fields.

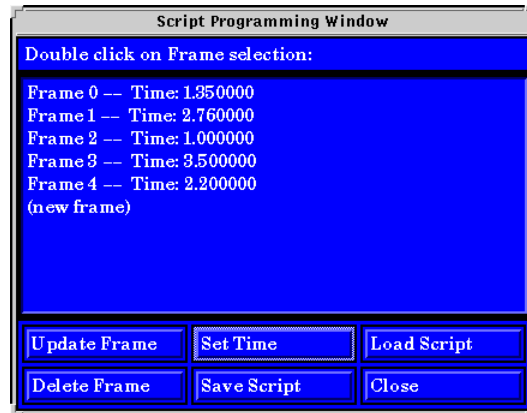


Figure 5: Script Editor window with five sample frames already defined. The user can add new frames, by first drawing on the Main Window canvas the desired combination of force field windows and then double-clicking on “(new frame)”. The label of the push button “Update Frame” is changed to “Add New Frame” whenever the user selects the “(new frame)” option.

5. Conclusion.

In this paper we described the CILIA software package, a programmable, task-level planning, 2-D simulator for arrays of manipulators. Several implementation and technical details were discussed.

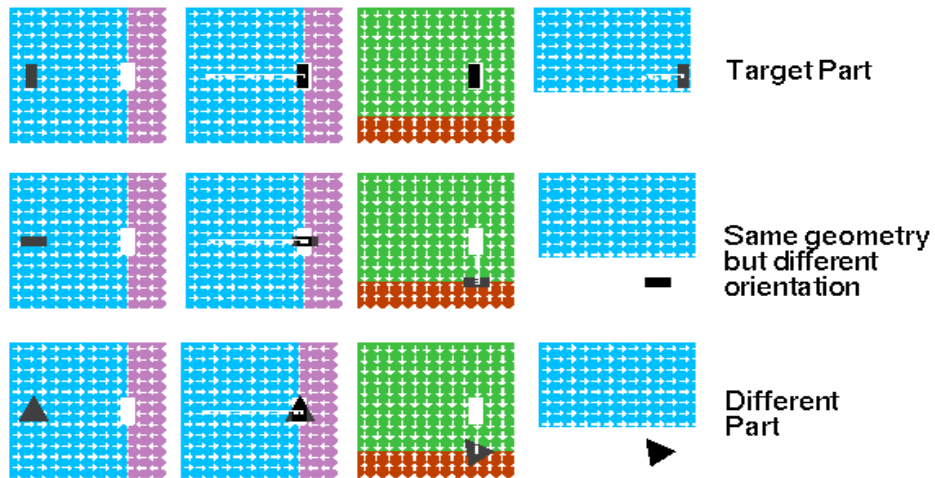


Figure 6: Simulation example of sorting parts using scripts. We use a hole slightly bigger than the part to be sorted to prevent it from being pushed away by the neighbors force fields.

Future research directions include the modeling of the discrete interaction between each manipulator and the objects, and the study of sensorless manipulation strategies using the software as a test bed. Also, we have plans to port the entire program to the Java language in order to overcome the network latency problem discussed on section 3. For more information, please, visit the project's web page at <http://www.isi.edu/mass>.

6. References.

- [1] M. Coutinho and P. Will, "*Using Dynamic Force Fields to Manipulate Parts on an Intelligent Motion Surface*", to appear at the IEEE International Symposium on Automation and Task Planning, Marina del Rey, CA, August 1997.
- [2] M. Coutinho, P. Will and S. Viswanathan, "*The Intelligent Motion Surface: a hardware/software tool for the assembly of meso-scale devices*", IEEE ICRA, New Mexico, April 1997.
- [3] P. Will and W. Liu, "*Parts Manipulation on a MEMS Intelligent Motion Surface*", ISI Research Report - ISI/RR-94-391, Marina del Rey, CA, May 1994.
- [4] W. Liu and P. Will, "*Parts Manipulation on an Intelligent Motion Surface*", Human Robot Interaction and Cooperative Robots, Vol. 3, Proceedings of the IEEE/RSJ IROS, pp. 399-404, Pittsburgh, PA, August 1995.
- [5] W. Liu, P. Will, and M. Pottenger, "*Modeling and Simulation of an Intelligent Motion Surface in MEMS*", Simulation and Design of Microsystems and Microstructures, Proc. of the First International Conf. on Simulation and Design of Microsystems and Microstructures, pp. 311-320, Southampton, England, September 1995.
- [6] C. Liu, T. Tsao, P. Will, Y. Tai, and W. Liu, "*A micro-machined magnetic actuator array for microrobotics assembly systems*", in Transducers - Digest International Conf. on Solid-State Sensors and Actuators, Stockholm, Sweden, 1995.
- [7] N. Takeshima and H. Fujita, "*Design and Control of Systems with Microactuator Arrays*", Proc. IEEE Work. in Advanced Motion Control, pp.219-232, Yokohama, Japan, March 1990.
- [8] K. Pister, R. Fearing and R. Howe, "*A planar levitated electrostatic actuator system*", Proc. IEEE Microelectromechanical Systems Conf., Napa Valley, CA, pp. 67-71, Feb. 1990.
- [9] J. K. Hahn, "*Realistic Animation of Rigid Bodies*", Computer Graphics, pp.299-308, vol.22, number 4, August 1988.
- [10] Brian V. Mirtich, "*Impulse-based Dynamic Simulation of Rigid Body Systems*", Ph.D. thesis, Univ. of California at Berkeley, Fall 1996.
- [11] Norman I. Badler et.al, "*Making them Move: Mechanics, Control, and Animation of Articulated Figures*", Morgan Kaufmann Publishers, 1991.
- [12] W.V. Tipping, "*An Introduction to Mechanical Assembly*", Business Books Ltd., London, 1969.