An OpenGL Application for Industrial Robots Simulation

C. Marcu, Gh. Lazea, R. Robotin
Technical University of Cluj-Napoca, Romania
{Cosmin.Marcu, Gheorghe.Lazea, Radu.Robotin}@aut.utcluj.ro

***Abstract*- This paper presents a Visual C++ and OpenGL application for 3D simulation of the serial industrial robots. To develop this application we started from the forward kinematics of the robot taken into consideration. The functions implemented in the source code are able to calculate the position and orientation of each robot joint, including the position and orientation of the robot gripper. With the help of the OpenGL functions, the application is able to draw and simulate the 3D kinematic scheme of the robot. In addition, the application has a calculus module where the gripper position can be determined using particular values for the robot joints positions or orientations.**

I. Introduction

The field of computer simulation is over forty years old, but still vibrant and growing. As technology develops faster hardware, old methods of simulation are made faster, and new varieties of simulation emerge through an *extension* process. Extending the core simulation knowledge base involves taking existing simulation concepts and blending them with those outside of the simulation discipline. An extension example is combining two concepts: a system model and an abstract programming *object* (from object oriented design). We can extend system models by designing model components as objects. This extension seems simple enough, however, some controversies arise. Should all physical systems be modeled with objects, or would some better be modeled with equations, for instance? How do models based on equations mesh with object? Sometimes, the interface between the simulation concept and the extensional concept is straightforward, but in most instances, there are many issues to be addressed. The ultimate goal within the simulation community is to walk the narrow line, separating a mathematically and theory-based defined system, on one hand, and the world outside simulation that encourages extension and possible revision of the basic approaches, on the other hand.

Within computer graphics, there has been a noticeable push in the direction of physically based modeling. In actuality, the movement is more due to an increased attention on *modeling*, rather than on physically based modeling. To understand this, consider the way the computer animations have been performed in the past. The first computer animations were drawn on individual sheets of paper or celluloid. When a series of sheets is transferred on film and run at a speed of 24 frames per second [4], an illusion of motion is perceived. From a simulation perspective, the sheets reflect a declarative type of model that is similar to a finite state automaton - one state per sheet. Therefore, early animation models were finite state automata. The next generation of animation programs allowed users to automatically create in between states by specifying certain key states (key frames) and then using geometric smoothing methods such as a Bezier or cubic spline [4]. Now, given this knowledge, we can see how progress has been made in computer animation. Specifically, we can create more cohesive models by basing the animation on more complex model types - such as event modeling, functional modeling or constraint modeling. The term "physically based" normally refers to constraint based, equations, models derived from physical laws; however, the movement called physically based modeling is a step in the direction of using simulation models to replace the older automata type models -inherent in frame based simulation- with more comprehensive system models.

Computer simulation is model designing of an actual or theoretical physical system, executing the model on a digital computer, and analyzing the execution output.

Simulation, like most disciplines, can be generally divided into *methodology* and *applications*. Sometimes, methodology is termed *theory*. The importance of methodology - and not just applications - cannot be overemphasized. The simulation discipline has a core of knowledge which is independent of applications. The simulation methodology can be defined into the sub-fields of model design, model execution and execution analysis. Methodology can apply itself to all sorts of practical real-world applications, but it is a substantial field by itself.

II. Simulator Design Tools

Due to the fact that a simulator requires a theoretical component and a graphical part, the best solution is to use a tool that implements both theoretical and graphical parts. The theoretical part is represented by formulae and motion laws while the graphical one is described by symbols and drawings, strongly connected with the theoretical components. In our case, the tool that accomplished both requirements was *Microsoft Visual C++*.

As the Microsoft Windows and the benefits of the graphical user interface (GUI) became widely accepted, there is an on-growing demand for visual programming on Windows based

1-4244-0361-8/06/$20.00 ©2006 IEEE

platforms. Programming for Windows based platforms is different from old-style batch or transaction-oriented programming. An essential difference between them is that a Windows program processes user input via messages from the operating system, while an MS-DOS program calls the operating system to get user input.

Visual C++ is a textual language which uses a graphical user interface builder to make programming decent interfaces easier on the programmer.

C++ is one of the components of Visual C++. However, its compiler can process both C source code and C++ source code. Furthermore, the new versions compiles Fortran code as well. Visual C++ also includes a large and elaborate collection of software development tools, all used through a windowed interface.

Visual C++ contains a large collection of libraries made to ease programmer's work but most of them are used to work with windows, controls and basic C/C++ components. One of its disadvantages is the missing of high performance graphical functions. The built-in graphical functions are very simple and not designed to obtain high quality graphics.

Because a simulator requires a graphical interface too, even if it's not necessarily to work at high quality, we had to link other tools with the VC++ interface. Other tools/addons that works well with VC++ are *DirectX* and *OpenGL*.

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions [4]. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

Any visual computing application requiring maximum performance-from 3D animation to CAD to visual simulation-can exploit high-quality, high-performance OpenGL capabilities. These capabilities allow developers in diverse markets such as broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to produce and display incredibly compelling 2D and 3D graphics.

OpenGL routines simplify the development of graphics software - from rendering a simple geometric point, line, or filled polygon to the creation of the most complex lighted and texture-mapped NURBS curved surface [4]. OpenGL gives software developers access to geometric and image primitives, display lists, modeling transformations, lighting and texturing, anti-aliasing, blending, and many other features [4].

Every conforming OpenGL implementation includes the full complement of OpenGL functions. The well-specified OpenGL standard has language bindings for C, C++, Fortran, Ada, and Java. All licensed OpenGL implementations come from a single specification and language binding document, and are required to pass a set of conformance tests. Applications utilizing OpenGL functions are easily portable across a wide array of platforms for maximized programmer productivity and shorter time-to-market.

OpenGL is supported on all UNIX workstations, and shipped standard with every Windows 95/98/2000/NT and MacOS PC. It runs on every major operating system including Mac OS, OS/2, UNIX, Windows 95/98, Windows 2000, Windows NT, Linux, OPENStep, and BeOS; it also works with every major windowing system, including Win32, MacOS, Presentation Manager, and X-Window System. OpenGL is callable from Ada, C, C++, Fortran, Python, Perl and Java and offers complete independence from network protocols and topologies.

Talking about the application where OpenGL is often used, the following types can be mentioned:

a). *Windows Applications*
- 3D animation / modeling / rendering and creative content creation;
- CAD/CAM /Digital Prototyping;
- Developer Toolkits & Libraries, Game Engines, High Level 3D APIs;
- Games;
- VRML Authoring & Viewing;
- Utilities: Screensavers, Format Converters, Benchmarks;
- Simulation & Visualization;
- Scientific, Data Analysis & Geographic Mapping.

b). *Linux Applications* – all categories;

c). *Mac Applications* – all categories.

As a conclusion, Visual C++ can be considered an open-source development tool which can implement and build user defined components and libraries. It is to be remembered that OpenGL is not an interface but a set of libraries which can be implemented and used in the programming environments mentioned. OpenGL has nothing but powerful functions and libraries that can be called, used and modified in programming environments. Some applications which implemented OpenGL in their engines and are commonly used in 3D modeling are: SolidWorks, Unigraphics, Catia, I-DEAS, VariCAD, Graphite.

III. Simulator Main Window

Each type of MFC dialog windows [1] has a default form. Depending on the user needs, extra controls can be easily added. In this case we have as initial data the components of nominal geometry matrix (3.1) [3].

$$M_{\substack{vn \\ (n+1)\times 6}}^{(0)} = \left[\begin{bmatrix} \bar{p}_i^{(0)T} & \bar{k}_i^{(0)T} \end{bmatrix} i = 1 \to n+1 \right]^T \quad (3.1)$$

The matrix contains the index of each joint (1 to 7), type of joint (rotation or translation), position of each joint and the motion axis. The position of a joint can be established related to {0} coordinate system (robot base) or related to the position of the previous joint.

Table I presents an example of the nominal geometry matrix $M_{vn}^{(0)}$ [3].

TABLE I

i 1…7	Joints (R;T)	$\left\{\overline{p}_i^{(0)}\right\}^T$			$@\left\{\overline{p}_{Ai}^{(0)}\right\}^T$		
1	R	0	0	l_0	0	0	1
2	T	0	l_0	0	0	1	0
…	…	…	…	…	…	…	…
7	-	0	0	l_0	0	0	1

These are, theoretically, the data needed to generate the robotic structure. $\left\{\overline{p}_i^{(0)}\right\}^T$ represents the position of each joint on X, Y, and Z. In this case, the position is relative to the previous joint. $@\left\{\overline{p}_{Ai}^{(0)}\right\}^T$ represents the motion axis of each joint [3].

Starting from this, in the simulator's main window were introduced edit boxes for each component of the matrix and some additional controls like buttons, check boxes, group boxes and combo boxes [1].

As initial data, the user must know the number of joints for the structure, the position of each joint related to the previous one (or to the base), the type of joint (rotation or translation; not applicable for gripper) and the motion axis. Figure 1 presents the main window of the simulator after all needed controls and edit boxes were added.

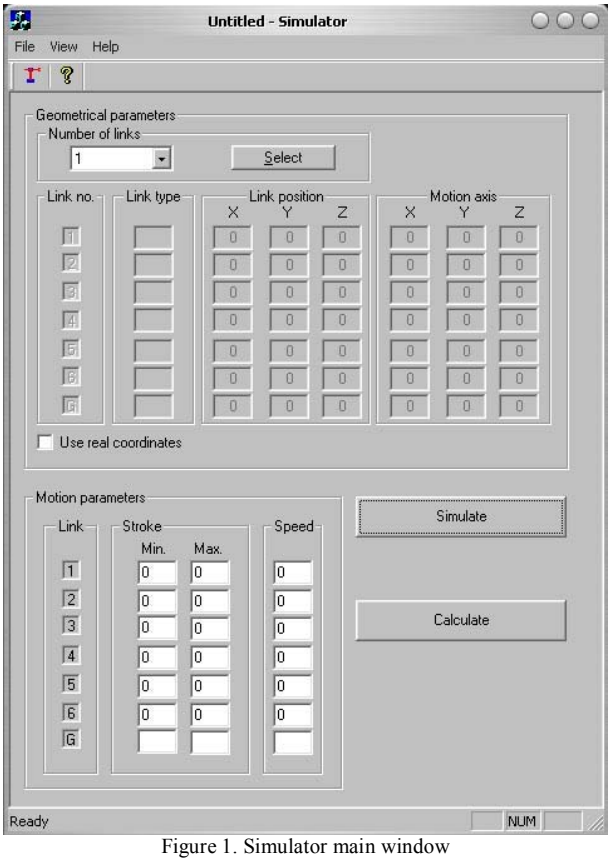

Figure 1. Simulator main window

Under "Number of links" group the user can select the number of links and joints for the structure that needs to be generated. By default, all edit boxes are disabled (can't be edited) and they are enabled only when "Assign button" is pushed. The maximum number of joins that can be selected is 6. This is because most of the serial robots configurations have maximum 6 degrees of freedom. In some cases, when additional joints are used they can reach 7 DOF.

The "i" group represents the index of each joint and it is disabled by default. In the "Link" group there can be selected the type of joint, rotation or translation, by adding "R" or "T". The edit boxes from "Position" group are accepting only integer values and all are by default 0. In the "Axis" group the values that can be entered are "0" or "1". The axis that corresponds to the value "1" in the "Axis" group is the motion axis of the corresponding joint.

In the source program there are implemented functions for warning and restricting the user to insert wrong values. For example in the "Link" group only "R" or "T" can be selected. If other key is used a warning message will be shown. The warning window is presented in Figure 2.

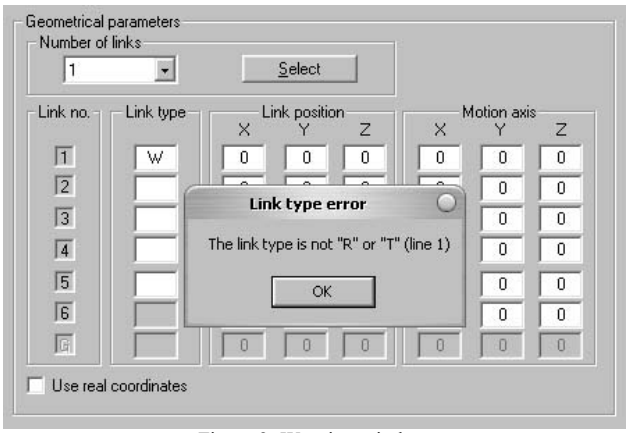

Figure 2. Warning window

There are some restrictions for position and axis too. As it was mentioned above, position edit boxes are accepting only integer values and a warning window will open if other types are used. If for motion axis are used other values than "0" and/or "1" the warning window will show up. For axis, there are considered as correct values only arrays having three elements, one of them being "1" and the other two being "0". This vectors correspond to X (1,0,0), Y(0,1,0) and Z (0,0,1) as motion axis.

IV. The OpenGL Window

OpenGL offers the possibility to generate high quality graphics by using its built-in primitive functions. The process of generating a 3D object is not as easy as it looks like and requires a high volume of code. However, the basic structure of an OpenGL source program can be simple. Its tasks are to initialize certain states that control OpenGL rendering and to specify objects to be rendered.

Rendering is the process by which a computer creates images from models [4]. These *models*, or objects, are constructed from geometric primitives - points, lines, and polygons - that are specified by their vertices.

The final rendered image consists of pixels drawn on the screen; a pixel - short for picture element - is the smallest visible element the display hardware can put on the screen [4].

Information about the pixels (for instance, what color they're supposed to be) is organized in system memory into bitplanes [4]. A bitplane is an area of memory that holds one bit of information for every pixel on the screen; the bit might indicate how red a particular pixel is supposed to be, for example. The bitplanes are themselves organized into a framebuffer [4], which holds all the information that the graphics display needs to control the brightness of the pixels on the screen.

The OpenGL window is being initialized in a different way comparing with a normal windows based dialog. The initialization source code can vary from one to hundreds of lines depending on the needed image quality. Working with OpenGL functions and constants is difficult and requires exact knowledge about the objects that need to be drawn. Each object, no matter how complicated it is, is being drawn using primitive shapes. For example, to draw a polygon in a XYZ coordinate system, the following code sequence is needed [4]:

```
glBegin(GL_POLYGON);
glNormal3d(0.0,0.0,-1.0);
glVertex3d( x,  y, z);
glVertex3d( x, -y, z);
glVertex3d(-x, -y, z);
glVertex3d(-x,  y, z);
glEnd();
```

This means that, function of what constant we chose (in this case GL_POLIGON [4]), OpenGL will draw the specific object in a different way.

To create a 3D object, each side must be taken into consideration and created independently. For example, to create a cube each side must be dawn separately using GL_POLYGON or GL_QUADS constants [4].

Having the $M_{vn}^{(0)}$ matrix [3], the type of joint, its position relative to the previous joint and motion axis, the kinematic structure can be represented in a XYZ coordinating system. Starting from the classical 2D representation of the links and joints, the translation and rotation joints will have another view (symbol) in a 3D frame. For translation the symbol will be a cube and for rotation a cylinder. The difference between classical 2D representation and 3D representation made by simulator is that the user can remark easily how the links and joints are placed in the structure.

Because the kinematic representation of the structure uses only symbols to provide a schematic view of the mechanical structure, the distances between joints could influence the correct view. That is why there were established fixed distances between the joints and fixed dimensions for links.

A kinematic diagram helps in establishing the right position of the joints. For example a “RRT” configuration can have different geometrical solutions depending on the joint position on a certain axis.

In the next example it was taken into study a 5R robot configuration. Tables II and III present the matrices of nominal geometry and links, $M_{vn}^{(0)}$ and $M_{C}^{(0)}$ [3].

TABLE II

Joint i=1..6	$\{\bar{p}_i^{(0)}\}^T$			$@\{\bar{p}_{Ai}^{(0)}\}^T$		
	[mm]			[mm]		
1	0	0	20	0	0	21
2	0	0	200	1	0	200
3	0	180	200	1	180	200
4	0	360	200	1	360	200
5	0	380	200	0	381	200
6	0	450	200	0	451	200

TABLE III

i=1..6	**1**	**2**	**3**	**4**	**5**
$C_i = \{ R; T \}$	R	R	R	R	R
$\bar{u} = \{\bar{x}_0; \bar{y}_0; \bar{z}_0\}$	$\bar{z}_0$	$\bar{x}_0$	$\bar{x}_0$	$\bar{x}_0$	$\bar{y}_0$

The data presented in the Table II are the coordinates of each joint relative to the structure base and the motion axis. To simplify the matrix we can reduce the two matrices to one in which the coordinates of each joint are given relative to previous joint.

Table IV presents the simplified $M_{vn}^{(0)}$ matrix.

TABLE IV

Joint i=1..6	$\{\bar{p}_i^{(0)}\}^T$			$@\{\bar{p}_{Ai}^{(0)}\}^T$		
1	0	0	20	0	0	1
2	0	0	200	1	0	0
3	0	180	0	1	0	0
4	0	80	0	1	0	0
5	0	20	0	0	1	0
6	0	70	0	0	1	0

Using the program (simulator), to generate the kinematic diagram of this robot structure, we need to assign 5 joints. A line for gripper coordinates will be automatically added. Introducing the data from the nominal geometry matrix in the edit box fields, the simulator will generate the kinematic diagram (for this type of structure), as the one presented in the Figure 3.

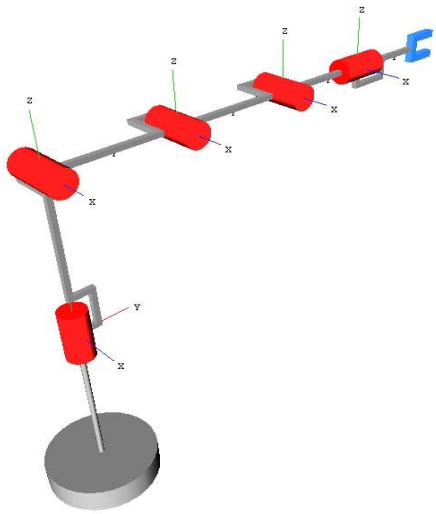

Figure 3. Kinematic diagram for a 5R robot structure

The application has the possibility to animate any structure represented in the OpenGL window. In the bottom of the main window, the user can insert the motion parameters. Depending on the robot taken into consideration, the user can set the stroke and the speed for each joint. By default, all the motion parameters are set to zero. If none of these parameters are changed, the application will draw the robotic structure in "zero" position within the OpenGL window, without moving any joint (see Figure 3). If the motion parameters are set, each joint will move simultaneously and continuously (until the right mouse button is clicked), from the minimum position to the maximum position, with the corresponding speed. Figure 4 presents a robot structure in a particular position.

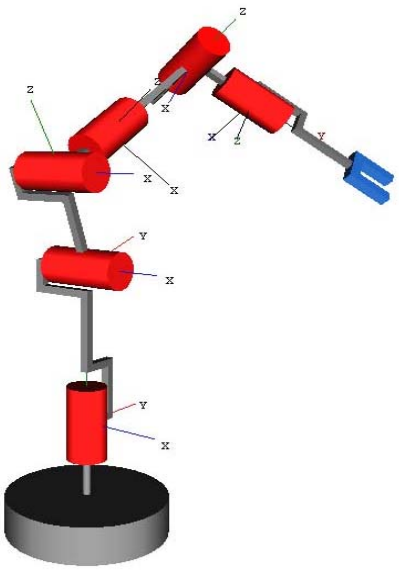

Figure 4. A 6R robot in a particular position

V. Gripper Position Determination

Talking about numerical simulation in case of robots, computers are having a well deserved place. Due to the complexity of the algorithms that are used in robots analysis, first aid comes especially form the simulation programs. Nowadays, as computers are able to solve a high amount of calculus, they are commonly used in numerical simulation.

In robotic numerical simulation an important place is taken by the used algorithm. Function of the needed result, the code made for simulation has different degrees of complexity. The main idea is to transform the theoretical algorithms into simple and generalized ones. The algorithm complexity varies function of the modeling type. When a robot structure is studied as a static model, the simulation is simple comparing with the kinematic and dynamic simulation, where, much more terms have to be taken into consideration (velocity, acceleration, mass, forces, etc.).

Studying different robotic structures, it can be easily observed that each of them has different static, kinematic and dynamic properties. Each type of joint and motion is important when a calculus algorithm is applied.

The most time consuming work, when a robotic structure is studied, is the matrices calculus. Theoretically, many combinations of structures can be analyzed, but, in practice most of them can be replaced with simple ones. When studying a high number of different structures, the best way to ease the calculus is to generalize calculus algorithms and to implement them in robot numerical simulators. Programming languages have an important role in this case.

In this application, for numerical simulation, the homogeneous transformation equations [3] were implemented. Because the first step was the kinematic diagram design, the equivalent step in numerical simulation is the geometrical modeling.

The main window of the simulator, presented in Figure 1, contains a shortcut button to the numerical simulation window. The button is disabled (grayed) by default to avoid the calculus of a non-existent structure. The first step when a user is analyzing a structure with this simulator is to assign correct values for the joints. The second step is to generate (with the help of "Simulate" button) the kinematic diagram of the structure taken into study. After this, the application automatically enables the shortcut button to numerical simulation called "Calculate".

Figure 5 presents the numerical simulation window of the application (a dialog based one [1]).

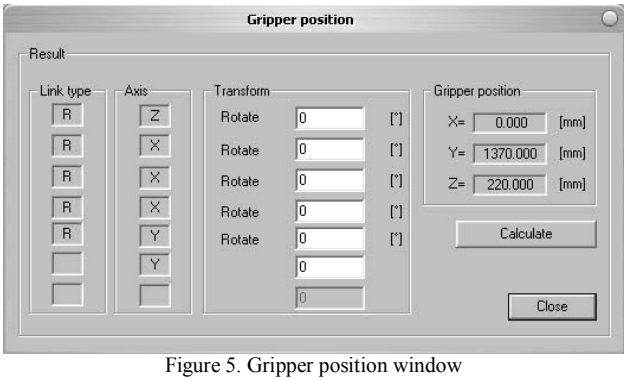

Figure 5. Gripper position window

The numerical simulation window contains the following four main groups:

- "Link Type" group – in this group there are listed the joint types (R for rotation and T for translation) for each joint index. The joint type is taken automatically by the application from the main window.

- "Axis" group – represents the axis of the coordinate system on which the link is performing the movement. As in the "Link Type" case, the values are taken automatically from the main window.

- "Transform" group – in this group there are listed the operations that the user can make with the joints: Translate or Rotate.

- "Gripper position" group – contains three fields in which it is listed the actual position of the gripper on each axis related to the robot base.

Because the algorithm calculates, in first step, the gripper position considering the mechanical structure in "zero" position, an extra control button called "Calculate" was implemented, in order to determine the gripper position after transformations in joints positions.

The algorithm implemented in the source code of the gripper position window actually determines the homogeneous transformation matrix (5.1) [3] for the robot taken into consideration. The homogeneous transformation matrix contains the position and the orientation of an analyzed joint (or gripper), relative to the robot base or previous joint.

$$ {}^{0}_{i}[T] = \prod_{j=1}^{i} {}^{j-1}_{j}[T], \tag{5.1} $$

where ${}^{j-1}_{j}[T]$ is the transformation matrix between the joints "j" and "j-1". Function of the joint type (rotation or translation), the ${}^{j-1}_{j}[T]$ matrix can have the following forms:

$$ {}^{i-1}_{i}[T] = \begin{bmatrix} {}^{i-1}_{i}[R] & | & {}^{i-1}\bar{r}_i \\ ---- & | & --- \\ 0 \quad 0 \quad 0 & | & 1 \end{bmatrix} \tag{5.2} $$

where ${}^{i-1}_{i}[R]$ is the rotation matrix and has a particular form, function of the rotation axis. The rotation matrix around "X" axis with angle θ will be:

$$ R(x,\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \tag{5.3} $$

around "Y" axis:

$$ R(y,\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \tag{5.4} $$

and around "Z" axis:

$$ R(z,\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.5} $$

For translation, the rotation matrix will be the 3x3 unity matrix.

The program automatically selects the proper matrix, function of the joint types set by user in the main window. Having (5.2)-(5.5) implemented in the program source as functions, the program calculates the ${}^{0}_{i}[T]$ matrix and displays the result in the gripper position window.

VI. Conclusions

The use of modern programming environments, like Visual C++, the algorithms generalization and their implementation in the simulator source code as functions, made possible the design of an application with a simple and easy to use interface.

Comparing with other professional simulators made in the field of robotics, this application confide oneself in the graphical part to kinematic diagrams representation, offering different possibilities of viewing the structure.

The numerical simulation part includes the calculus algorithm for determining the gripper position. The user has the possibility to modify the joints configuration and recalculate the gripper position with the new given parameters.

Because of the way it was designed, this application allows coders to implement in the source code new functions, to increase the graphical quality, and new generalized algorithms for numerical simulation.

Analyzing the current stage of the simulator, the main trends are to implement the kinematics and dynamics algorithms. Also, another trend could be structures comparison and choosing the best variant. The design of a version for parallel robots represents another future objective.

References

[1] Davis, C., *Teach yourself Visual C++ in 21 days*, Sams Publishing, 1998

[2] Megahed, S., *Principes of Robot Modelling and Simulation*, John Wiley &Sons, 1993

[3] Negrean, I., *Robotics – Kinematic and Dynamic Modelling*, Ed. Didactica si Pedagogica, Bucharest, 1998

[4] Wright, R., *OpenGL superbible*, Waite Group Press, 2000