Robot Virtual Assembly Based on Collision Detection in Java3D

Peihua Chen[1,*], Qixin Cao[2], Charles Lo[1], Zhen Zhang[1], and Yang Yang[1]

[1] Research Institute of Robotics, Shanghai Jiao Tong University
cph@sjtu.edu.cn, charleslo77@gmail.com, zzh2000@sjtu.edu.cn, iyangyang186@yahoo.com.cn
[2] The State key Laboratory of Mechanical System and Vibration, Shanghai Jiao Tong University
qxcao@sjtu.edu.cn

Abstract. This paper discusses a virtual assembly system of robots based on collision detection in Java 3D. The development of this system is focused on three major components: Model Transformation, Visual Virtual Assembly, and an XML output format. Every component is presented, as well as a novel and effective algorithm for collision detection is proposed.

Keywords: Virtual Assembly, Collision detection, Service robot, Java 3D, VRML.

1 Introduction

Virtual Assembly (VA) is a key component of virtual manufacturing [1]. Presently, virtual assembly serves mainly as a visualization tool to examine the geometrical representation of the assembly design and provide a 3D view of the assembly process in the field of virtual manufacturing [2-5]. However, in service robots' simulation, we also need to assemble many types of robots in the virtual environment (VE). There are mainly two methods of virtual assembly in these systems, e.g., assembling every single part together via hard-coding using link parameters between each part, and another method is assembling a virtual robot in an XML(eXtensible Modeling Language), both of which are not visual and complicated.

This paper focuses on the visual implementation of making virtual assembly of service robots in Java3D. We choose Java JDK 1.6.0 and Java 3D 1.5.2 as the platform. First we discuss the system architecture and methodology. This is followed by its implementation. After that, an application of this system will be presented and conclusions obtained.

2 System Architecture

2.1 System Architecture

The architecture of this system is presented in Fig.1. It includes three parts: Model Transformation, Visual Virtual Assembly (VVA), and XML Output. Firstly, after we

* Corresponding author.

H. Deng et al. (Eds.): AICI 2009, LNAI 5855, pp. 270–277, 2009.
© Springer-Verlag Berlin Heidelberg 2009

create the CAD models of the robot, we export these models as IGES [7] format to be imported into 3DS Max. In 3DS Max, we transform the Global Coordinate System (GCS) of the IGES model into the coordinate system according to the right-hand rule, and then export the models in VRML format. Then, we load the VRML files into Java 3D. After which, we set the scale factor of the model to 1:1. Then, we set the target model as static, and move the object model towards the target model until the collision detection between them is activated. After which, we attach the object model to the target model, which means appending the object node to the target node. If the virtual assembly is not completed, then import another VRML file into the VE for assembling; if it is finished, then output the assembly result into an XML file.

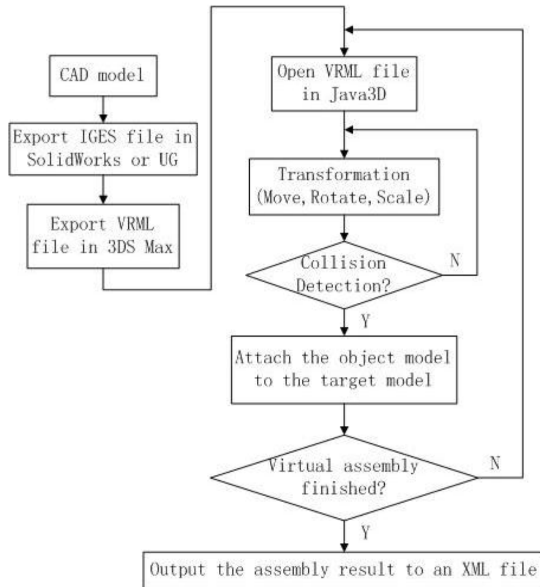


Fig. 1. System Architecture of VA for robots

2.2 Scene Graph Architecture

The use of scene graph architecture in Java 3D can help us organize the graphical objects easily in the virtual 3D world. Scene graphs offer better options for flexible and efficient rendering [8]. Fig.2 illustrates a scene graph architecture of Java 3D for loading the VRML models into the VE for scaling, translation and rotation transformations. In Fig.2, BG stands for BranchGroup Node, and TG, Transform -Group Node.

As shown in Fig.2, the Simple Universe object provides a foundation for the entire scene graph. A BranchGroup that is contained within another sub-graph may be reparented or detached during run-time if the appropriate capabilities are set. The TransformGroup node specifies a single spatial transformation, via a Transform3D object, that can position, orient, and scale all of its children [9], e.g., moveTG, rotTG, scaleTG, etc., as shown in Fig.2. The moveTG handles the translations of the VRML model, rotTG is for rotations, and scaleTG is for scaling.

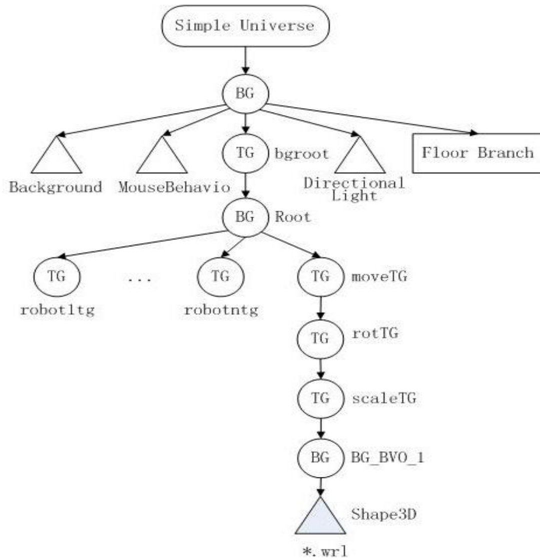


Fig. 2. Scene graph architecture in Java 3D

3 Implementation

3.1 Loading VRML Models

Java 3D provide support for runtime loaders. In this paper, we use the VrmlLoader class in package j3d-vrml97 to load the VRML files. In our project, we rewrite a vrmlload class with the construction method of vrmlload (String filename). After selecting a VRML file, the model will be seen in the 3D Viewer of our project, as shown in the left picture of Fig.3. The coordinate system in 3D Viewer is right-hand, with the orientation semantics such that +Y is horizontal to the right, +X is directly towards the user, and +Z is the local gravitational direction upwards.

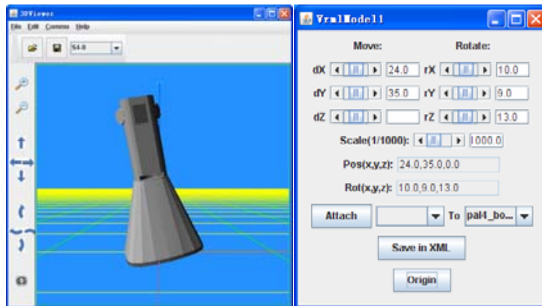


Fig. 3. A repositioned VRML model and its control panel

Fig.3 also shows how a model's position, orientation, and size can be adjusted. Here it shows a robot body that has been moved and rotated. In the control panel of the model, there are several operations. They are: translations of x, y, and z axes; rotations of the model around x, y, and z axes; scaling the model; displaying "Pos(x, y, z)" and "Rot(x, y, z)" at runtime, as shown in Fig.3. In addition, when another model is loaded, a corresponding control panel will be created at the same time.

3.2 Algorithm for Collision Detection

In the VVA system, collision detection plays an important role. Every node in the scene graph contains a Bounding field that stores the geometric extent of the node [10]. Java3D offers two classes "WakeupOn -CollisionEntry" and "WakeupOnCollision -Exit" to carry out collision detection for 3D models. When any objects in Java 3D scene collide with other bounds, "Wakeup -OnCollisionEntry" is triggered and when collision is released, "WakeupOnCollision -Exit" is triggered. So we can use them as triggering conditions and we are able to obtain the control behavior after collision occurs.

This paper proposes a novel and effective algorithm to detect collisions. VRML file format uses IndexedFaceSet node to describe shapes of faces and triangle patches to reveal all types of shapes in 3D models. IndexedFaceSet includes a coord field that contains a series of spatial points, which can be used in the coordIndex field to build the surfaces of 3D models. We conserve the points' position, and establish space for collision detection using the following method.

BoundingPolytope is chosen to use more than three half-space [11] to define a convex and closed polygonal space of a VRML model. We make use it to build the collision detection space. The function that defines every half-space α is: $Ax + By + Cz + D \le 0$, where A, B, C, and D are the parameters that specify the plane. The parameters are passed into the x, y, z, and w fields, respectively, of a Vector4d object in the constructor of BoundingPolytope (Vector4d[] planes). Thus we must make out the parameters A, B, C, and D of every triangle plane. The following method is to compute the value of A, B, C, and D. We suppose that one triangle is made up of three points: L, M, and N, shown as in Fig.4. And we obtain vector $\overrightarrow{LM}$ and $\overrightarrow{MN}$ based on a vertex sequence.

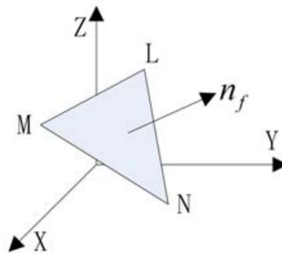


Fig. 4. Triangle plane in the WCS

$$\overrightarrow{LM} = (X_M - X_L)i + (Y_M - Y_L)j + (Z_M - Z_L)k \quad . \tag{1}$$

$$\overrightarrow{MN} = (X_N - X_M)i + (Y_N - Y_M)j + (Z_N - Z_M)k \quad . \tag{2}$$

The normal vector of plane α is the vector product of $\overrightarrow{LM}$ and $\overrightarrow{MN}$. That is

$$\overrightarrow{n_f} = \overrightarrow{LM} \times \overrightarrow{MN} = \begin{bmatrix} i & j & k \\ (X_M - X_L) & (Y_M - Y_L) & (Z_M - Z_L) \\ (X_N - X_M) & (Y_N - Y_M) & (Z_N - Z_M) \end{bmatrix} . \tag{3}$$

Once we obtain normal vector $\overrightarrow{n_f}$, we get values of A, B, C. Then we can obtain the value of constant D if we substitute any point's coordinate into the equation. After transmitting all plane parameters to one group of Vector4d objects, we can use BoundingPolytope to build a geometrical detection space. Then, we invoke setCollisionBounds(Bounds bounds) to set the collision bounds. After setting a collision bound, Java3D can carry out collision detection in response to the setting bounds.

3.3 Visual Virtual Assembly

While assembling the object model to the target model, we have to rotate the object model around the z-axis, y-axis, and x-axis in this order to correspond with the target Local Coordinate System (LCS). Then we need to move along the x, y, and z axes in both the positive and negative directions. In Fig.5, we present how we assemble the model J4 to model J3. Fig.5 (a) shows the initial positions of the two VRML models. Fig.5 (b) presents the state after the second model (J4) is rotated about the x-axis by -90°. We can also see from this picture that we don't need to make any translations

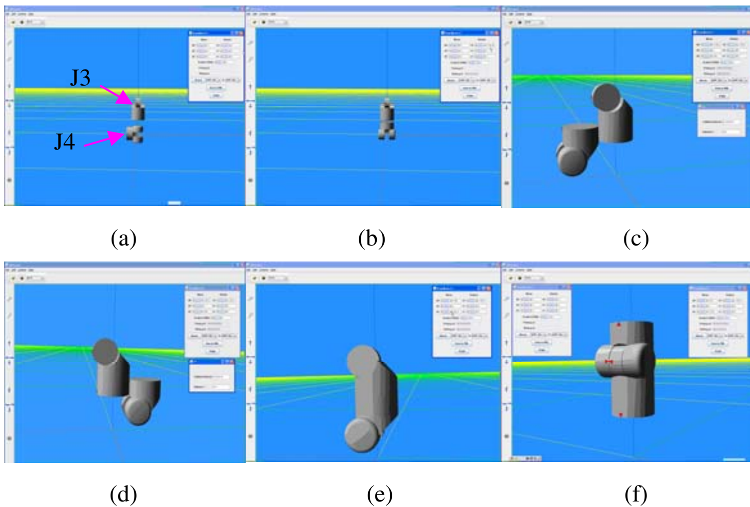


Fig. 5. Example of assembling one VRML model to another

along y-axis anymore, because the second model's y-axis is already corresponding to the first model. From Fig.5(c) and Fig.5 (d), we can obtain the offset values along +x direction and –x direction through the collision detection between the two models. As we have set the threshold value of collision detection at 0.5mm, thus when the distance between the two models is smaller than 0.5, the collision detection will be activated and the actual offset value will be obtained. Here, the offset value in the +x direction is 140.5-0.5=140(mm), and the offset value in the –x direction is -80.5+0.5=-80(mm), as shown in Fig.5 (d). So, in order to make the x-axis of the second model correspond to the first one, the total offset value along the x-axis is: (140 + (- 80)) / 2 = 30(mm), and the result can be seen in Fig.5 (e). Then move along the z-axis, and the operations are similar with the previous method. The final result of this virtual assembly is presented in Fig.5 (f).

After we have moved the second model to the correct position, we attach the second model's node to the first model's node.

3.4 Save the Virtual Assembly Result in XML Files

After we have realized the virtual assembly of the robot, we save the virtual assembly result as XML files, as shown in Fig.1. In this paper, we use JAXB (Java Architecture for XML Binding) [12] to access and process XML data. JAXB makes XML easy to use by compiling an XML schema into one or more Java technology classes. The structure of the XML schema of our project is presented in Fig.6, and the output data could be referred to, in this structure.

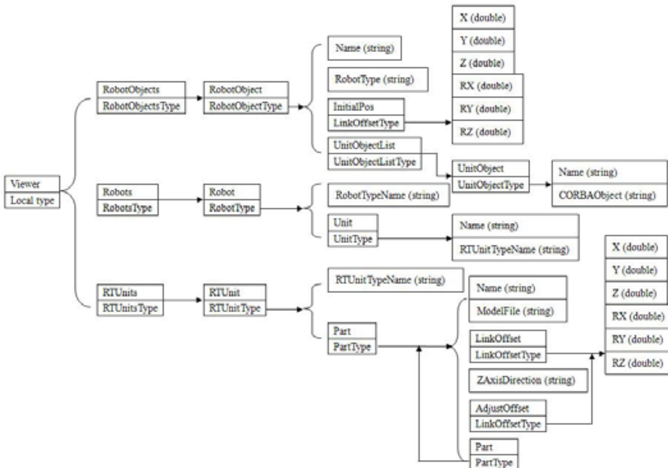


Fig. 6. Structure of the XML schema

4 Application

In this paper, we create the CAD models of the real robot as shown in Fig.7, and then translate them to VRML files according to the right-hand rule.

Fig. 7. A real robot with left arm

Then, we start the visual virtual assembly in Java 3D, as shown in Fig.8. After the virtual assembly is completed, we save the virtual assembly result to an XML file. Then we run the program again, and open the XML file just created, as shown in Fig.9 (a). At the same time, we can control the motion of every joint through the corresponding control panel, as shown in Fig.9 (b).

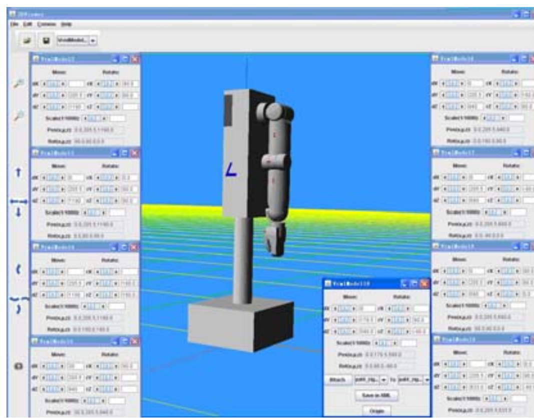
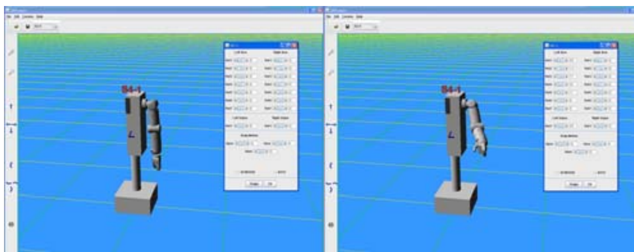


Fig. 8. Virtual assembly of the robot



(a) Open the XML file in Jave 3D

(b) Control the motions of the robot

Fig. 9. Open the XML file stored the VA data and Control the motions of the robot

5 Conclusions

From the application in section 5, Virtual Assembly of robots is carried out based on Collision Detection, and it proves to work well. Virtual Assembly of robots based on Collision Detection in Java3D allows for visual feedback, and makes the process much easier for the end-users. Through the virtual assembly, users can obtain the assembly result file in XML format that complies with a special XML schema. Then the users can easily load the new assembled robot via the XML file, and operate every joint of the robot, e.g., developing a program for workspace and kinematic analysis.

Acknowledgments. This work was supported in part by the National High Technology Research and Development Program of China under grant 2007AA041703-1, 2006AA040203, 2008AA042602 and 2007AA041602; the authors gratefully acknowledge the support from YASKAWA Electric Cooperation for supporting the collaborative research funds and address our thanks to Mr. Ikuo agamatsu and Mr. Masaru Adachi at YASKAWA for their cooperation.

References

1. Jayaram, S., Connacher, H., Lyons, K.: Virtual Assembly Using Virtual Reality Techniques. CAD 29(8), 575–584 (1997)
2. Choi, A.C.K., Chan, D.S.K., Yuen, A.M.F.: Application of Virtual Assembly Tools for Improving Product Design. Int. J. Adv. Manuf. Technol. 19, 377–383 (2002)
3. Jayaram, S., et al.: VADE: a Virtual Assembly Design Environment. IEEE Computer Graphics and Applications (1999)
4. Jiang-sheng, L., Ying-xue, Y., Pahlovy, S.A., Jian-guang, L.: A novel data decomposition and information translation method from CAD system to virtual assembly application. Int. J. Adv. Manuf. Technol. 28, 395–402 (2006)
5. Choi, A.C.K., Chan, D.S.K., Yuen, A.M.F.: Application of Virtual Assembly Tools for Improving Product Design. Int. J. Adv. Manuf. Technol. 19, 377–383 (2002)
6. Ko, C.C., Cheng, C.D.: Interactive Web-Based Virtual Reality with Java 3D, ch. 1. IGI Global (July 9, 2008)
7. http://ts.nist.gov/standards/iges//
8. Sowizral, H., Rushforth, K., Deering, M.: The Java 3D API specification, 2nd edn. Addison-Wesley, Reading (2001)
9. Sun Microsystems. The Java 3DTM API Specification, Version 1.2 (March 2000)
10. Selman, D.: Java 3D Programming. Manning Publications (2002)
11. http://en.wikipedia.org/wiki/Half-space
12. Ort, E., Mehta, B.: Java Architecture for XML Binding (JAXB). Sun Microsystems (2003)