Towards Dynamics Animation on Object-Oriented Animation Database System "MOVE"

Kunihiko Kaneko, Susumu Kuroki,
and Akifumi Makinouchi

Department of Computer Science and Communication Engineering
Kyushu University, 6-10-1 Hakozaki, Higashi-Ku Fukuoka, 812 Japan

ABSTRACT

The authors present why and how to develop three dimensional animation database system MOVE. Three dimensional animation system is an important application of mutilmedia database system, because many animators and engineers want the management facility of animation data.

The authors designed MOVE in object-oriented way. They implemented three dimensional animation database schema on a conventional Object-Oriented Database Management System (OODBMS). The schema contains some basic classes for three dimensional computer graphics such as vector operations, three dimensional object, polygon-meshed curved surface, and local illumination model. They also implemented basic algorithm like the Z-buffer algorithm and scan-line algorithm. Using MOVE, users can combine basic computer graphics objects and can build a scene. They can also extend these classes without knowing the details of MOVE.

And interface between simulation program and computer graphics routines is discussed here from the point of view of kinematic constraint and dynamics simulation.

Keywords : animation database, computer animation, computer graphics, animation data model, object-oriented database, multimedia database system

1 INTRODUCTION

Recently, three-dimensional computer animation became an important research issue and have been worked by many researchers. The application areas of computer animation spread from art and entertainment to CAD, robotics and three-dimensional volumetric simulation. Three dimensional animation is comprehensive enough to show three dimensional mechanical parts, robot manipulators, physical phenomena and so on.

The most important purpose of computer animation systems is to generate a realistic animation automatically. The most attractive way of simulation is to analyze the dynamism of a physical phenomenon by the law of physics and to simulate it automatically by a computer. The animators need not specify the details of motion as long as satisfiable solution is generated by such simulation. At present, many animation systems are being developed based on this idea. For example, hand animation[5], facial animation[20] are produced according to this idea.

Roughly speaking, computer animation involves following three activities: 1) modeling animated entity 2) motion specification and 3) image rendering[14]. In the case of simulation-based animation, animators specify the shape, spatial layout, attributes of the surrounding surfaces (i.e. color, texture, and so on) of three dimensional object, light-sources, and virtual-camera in modeling activity. And they specify how animated entities change their states by implementing simulation routines in motion specification activity. These information about the model and the motion are stored in a computer. And image rendering is performed by a computer automatically. In this activity, the computer calls simulation routines and calculates the states of animated entities at every frame and then generates images from these data.

Thalmann et.al. pointed out that the concept of *state variable* of animated entities is useful to unify various animation algorithms such as keyframe, parametric interpolation, kinematic simulation, dynamics simulation and other simulation-based algorithms[11]. In their idea, the motion of an animated entity is represented by a set of state variables that change by the progress of time. From their points of view, modeling is to determine what kind of state variables the animated entities have and what the initial values of these variables are. Also the motion specification is to specify how the state variables change.

Management of the date of such sets of state variables is one of the main problems of animation systems. So,

animation database system is necessary, because it is important to manage the animation data easily. In animation database, state variables are defined as database schema, and each value of state variables is stored as a database record.

Moreover, animation systems should allow the animators and engineers to access these state variables, because many animators and engineers often want to use the animation data which they previously made. So animation data should be stored in such a way that it is easy for them to use the data repeatedly. In addition, it is desirable that the animation systems provide flexible data definition mechanism. For example, the users want to use more flexible data definition mechanism when they design new three dimensional primitive objects, new illumination model, and simulation routines. The solution is to make these model extensible.

Recently, several computer graphics packages are developed such as MIT X-Window[25], PEX[24], and RenderMan[23]. These systems don't have animation data model, and they don't have the management facility of animation data. And they can't store animation data to secondary storage device. For example, Renderman has *current-transform* and *object-handle* mechanism to manipulate animated entities, but it doesn't have animation data model. PEX has *locator* mechanism to manipulate three dimensional objects, but it doesn't mention about data storage to secondary storage device.

That is why computer animation will become an important application of multi-media database systems. KUNII et.al. developed animation their database system based on relational data model[17][27]. They studied about their animation data model[17], management of geometric and motion data[18], and dynamics-based algorithms[26]. And they expressed the data-independence of their system, but they didn't pay much attention to the extensibility.

Among animation systems, some lack the data management mechanism, and others lack the extensibility. This doesn't indicates that an extensible animation model has not been implemented. So the authors propose new extensible animation data model.

Many researchers have pointed out that object-oriented concepts are useful for animation data modeling[12] [13]. So, the authors are now designing and implementing new three dimensional animation database system MOVE in object-oriented way. They are developing MOVE using an Object-Oriented Database System (OODBMS). The purpose of system development is to examine about animation data model and its extensibility. In MOVE, all the state variables of every animated entities are managed by system. And every data is *object*. And the basic unit of data management is also object.

The authors implemented basic classes for three dimensional computer graphics, such as vector operations, three dimensional object, polygon-meshed curved surface, and local illumination model. These classes are implemented as database schema of the conventional OODBMS ONTOS[19] using programming language C++[2]. These basic concepts are introduced in section 2 as well as system architecture of MOVE.

In section 3, the idea of *scene object* of MOVE is discussed. A scene object is a frame of an animation, and the data of the scene object are used to produce the image. The users of MOVE can get an image easily through constructing scene object from basic objects prepared by MOVE.

In section 4, the design and implementation of some classes, such as *solid object*, *surface object*, *surface-shader object*, *light-source object* and *camera object* are explained. System extensibility of MOVE is also discussed, such as user-defined three dimensional primitive object, and extensible illumination model.

In section 5, the authors discuss about the object-orientation and instancing mechanism of MOVE.

Finally, interface between simulation program and computer graphics routines is also discussed from the point of view of kinematic constraint and dynamics in section 6.

2 ANIMATION DATABASE SYSTEM "MOVE"

2.1 General Facilities of MOVE

Animation database systems should have computer graphics facilities, animation facilities and system extensibility. In this subsection, these facilities are introduced briefly.

1. Scene object
   A scene object is a frame of an animation. The users of MOVE can get an image using a scene object in this way: First, the user constructs a scene object with the basic objects of MOVE. After the user completes the scene object, MOVE generates an image from the scene object.
2. Vector operations
   All the geometric data of MOVE are vectors and points in three dimensional world. Mathematically, these data are represented by vectors in affine space[6]. So, MOVE supports affine vectors and their transformations.
3. Three dimensional object
   MOVE manages the data of three dimensional objects with curved-surface. The authors assume that all the objects are solid objects. Modeling the soft-objects is our future works.
4. Virtual camera
   Virtual-camera mechanism transforms three dimensional objects into two dimensional computer screen. This mechanism models perspective transformation and deformation. It is a good approximation of the camera with a simple lens.
5. Light-source
   The environment of a three dimensional world which

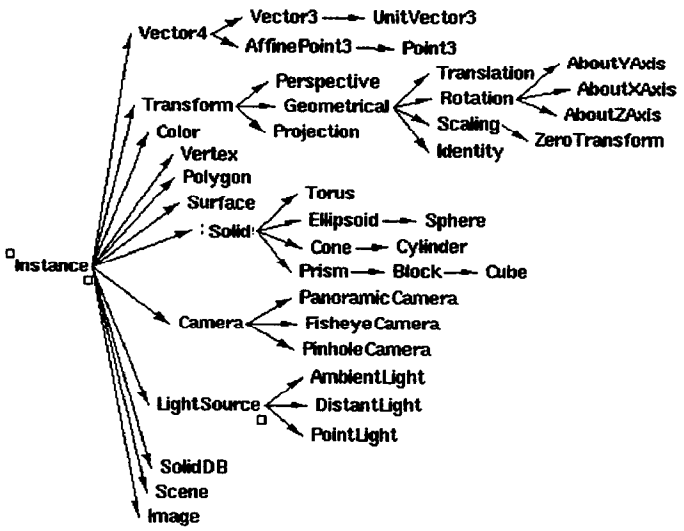

Figure 1: The class hierarchy of MOVE

affects the appearance of animated entities can be represented by a set of light-sources. MOVE has three kinds of light-source models: point light, distant light and ambient light.

6. Local illumination model
   To make the computer-generated images realistic, it is important to simulate the illumination and reflection precisely. The color at one pixel of image is calculated from the information about the geometry of three dimensional object, virtual camera, light sources and surface attributes. The equations of the local illumination model are implemented in MOVE.

7. Extensible model
   The users of MOVE can define the classes of user-defined three dimensional primitive object, and illumination model (i.e. surface attributes, light-source and virtual camera).

8. Animation support
   From the point of simulation-based animation, the interface between a scene objects and their own motion law is necessary. MOVE manages such relationships.

2.2 System Architecture of MOVE

The prototype of MOVE has been implemented on a *SUN SparcStation 2* workstation. The classes of three dimensional computer graphics are developed by SUN C++ ver.2.1. The authors designed classes of MOVE based on object-oriented concepts using Coad-Yordon's *object-oriented analysis*[22]. Figure 1 shows the class hierarchy of MOVE.

The class definition consists of approximately 12,000 lines. The database schema specified by these classes definition is managed by the object-oriented database management system ONTOS. The C++ code are compiled and resulting object files are collected as the library.

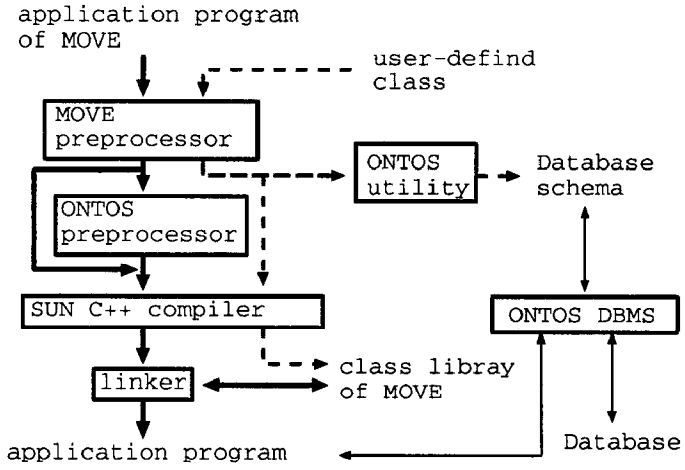

Figure 2: An application development in MOVE

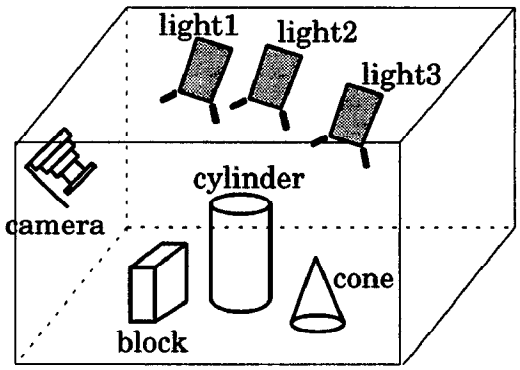

Figure 3: The model of three dimensional world

Users can create animation data in MOVE. Almost all the animation data are stored in database, and the others are on memory. Users can retrieve animation data which are previously made. And users can invoke the image rendering routines specifying what scene object they want to render. The produced image is also stored in database ONTOS.

MOVE maintains the relationships between 1)three dimensional objects and illumination model data, 2) animated entities and simulation routine and 3) animation data and rendering data.

Figure 2 shows the interfaces between MOVE and users. The user can define a new class which make up for the MOVE's weak points. The user can define the class by C++ and combine the class and MOVE classes. And the user can use the new class as well as MOVE classes. Such extensibility is one of the features of MOVE.

3 SCENE OBJECT

In MOVE, a scene object is a frame of an animation. A scene object contains all the model data that are necessary to generate an image. A scene object contains a pointer

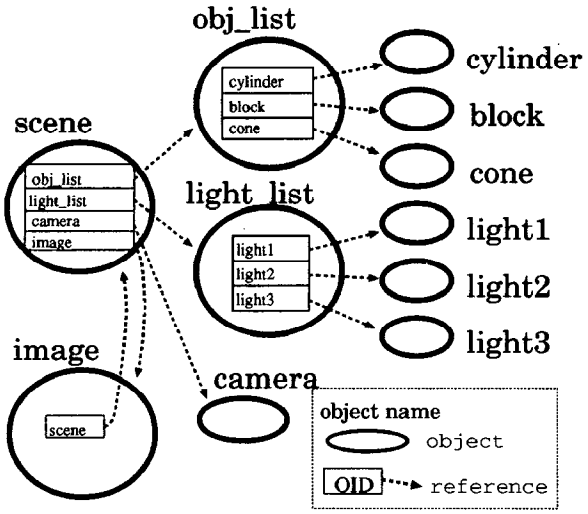

Figure 4: The structure of *scene object*

to the image object, too. The model data is the information about three dimensional world, and it consists of a set of three dimensional object, light-sources and a virtual camera (Figure 3). So, if the user wants to specify a scene object, the user has to specify three dimensional object, light-source objects and a camera object.

The user creates three dimensional objects and light-source objects using instancing mechanism of corresponding classes such as the classes `Cylinder`, `Block`, `Cone` and `PointLightSource`. Then users combine these objects and build a scene object. These basic computer graphics classes are convenient building blocks. The instancing mechanism is described in section 6.

A scene object has a hierarchical structure(Figure 4). A scene object has four pointers as follows, They are: 1) a pointer to a list of three dimensional object, 2) a pointer to a list of light object, 3) a pointer to a camera object, and 4) a pointer to an image object. Here, a list of a three dimensional object is itself an object. And the object has a list of pointers to some three dimensional objects. A list of light object is also an object. And the object has a list of pointers to light objects, too. And the image object has a back pointer to the scene object.

When the user creates a scene object, MOVE initializes these data automatically: 1) List object of three dimensional object, list object of light-sources, camera object and image object are created. 2) The pointers of the scene object are set. 3) The contents of two list objects remain to be empty. 4) The contents of image object is also empty. Now, list object of three dimensional object and list object of light-sources are ready to use. The user can add some corresponding objects to the list objects.

A scene object has two methods. One is the method `render()` and the another is the method `instancing()`. The method `render()` is to render an image, and the method `instancing()` is to create a copy of an object. When the user has defined an object and invokes the

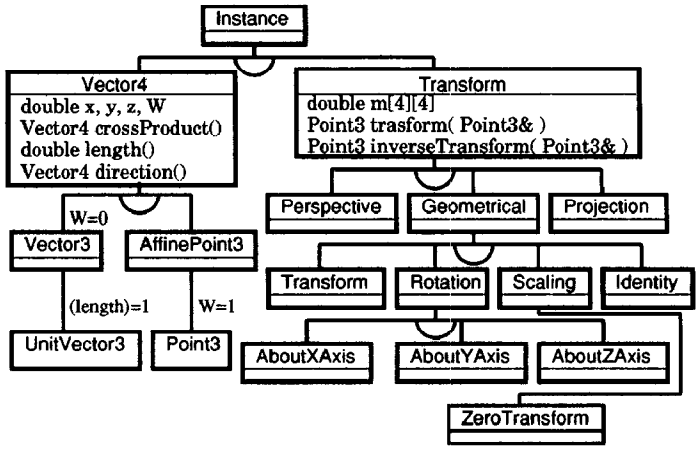

Figure 5: The class design of classes for vector and matrix operations

method `render()` of the object, the rendering routines are called and the rendering routines store the final image as an image object. And the user updates the scene object and invoke the method `render()`, the contents of image data is substituted by a different image data. When the user wants to avoid such substitution, the user invokes the method `instancing()` to make a copy of the object.

4 CLASS DESIGN DETAILS

4.1 Vector Operations

class	operations	mathematical expression
Vector3 (x,y,z,0)	addition	$v_1 + v_2$
	subtraction	$v_1 - v_2$
	multiplication	$k \cdot v, v \cdot k$
	division	k/v
	inner product	$v_1 \cdot v_2$
	outer product	$v_1 \times v_2$
Point3 (x,y,z,1)	addition	$p + v, v + p$
	subtraction	$p - v$
Transform (4×4matrix)	transformation of a point	$M \cdot a$
	inverted transformation of a point	$M^{-1} \cdot a$
	composed transformation	$M_1 \cdot M_2$

v, v_1, v_2	: vector	k	: floating point number
p	: point	a	: vector in the affine space
M, M_1, M_2	: transformation		

Table 1 : Vector operations

MOVE has classes for representation and transformation of a vector. In a three dimensional world, a vector and a point are quite another, because they have different operators (see Table 5). So, a vector belongs to the class `Vector3` and a point belongs to the class `Point3`.

The vector transformation and the point transformation can be treated similarly using the homogeneous

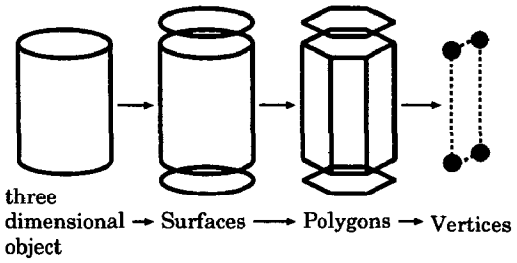

Figure 6: Polygonal approximation of three dimensional object

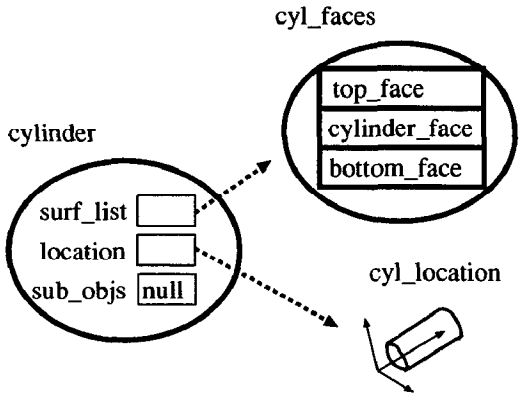

Figure 7: Data structure of three dimensional object in MOVE

co-ordinate system[6]. The class `Vector4` represents a vector in the affine space. The classes `Vector3` and `Point3` are derived from the class `Vector4`. The methods `transform()` and `inverseTransform()` are defined as methods of the class `Transform`. These methods take as an argument an object sometimes in class `Vector4` and sometimes in the derived classes of the class `Vector4`. This is why the class `Transform` represents the both of `Vector3` transformation and `Point3` transformation.

In detail, there twelve kinds of transformation in MOVE (Table 6). The derived classes may have more efficient transformation algorithm than the parent class. For example, the inverted transformation of a vector often needs the inverse matrix. It is time-consuming to calculate the inverse matrix. Besides, if the transformation is orthogonal (i.e. rotation or translation), the inverse matrix is expressed by transposed matrix. It is not time-consuming to calculate the transposed matrix at all. That is why the authors implemented twelve transformation classes.

4.2 The Model of Three Dimensional Object

Three dimensional objects with curved-surfaces are approximated by polygonal facets. For example, a cylinder could be approximated by an octahedron (Figure 6).

There are several researches about the object-oriented representation of three dimensional geometric objects[3][16]. They focused on data representation and query language, but they paid few attention to the hierarchical structure, instancing and user-defined primitive object. That is why the authors designed the class `Solid`.

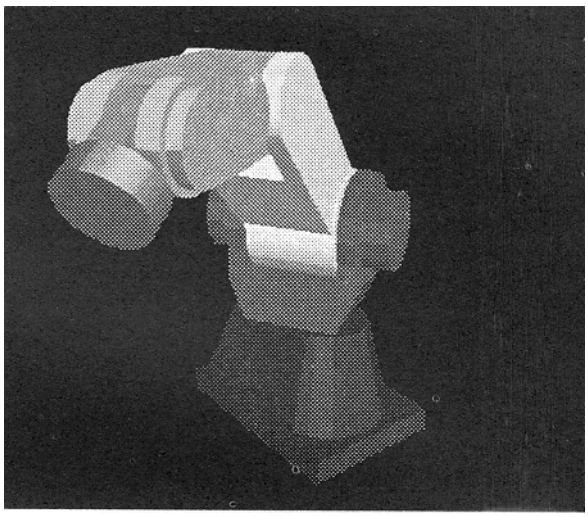

Figure 8: The rendered image of a robot manipulator

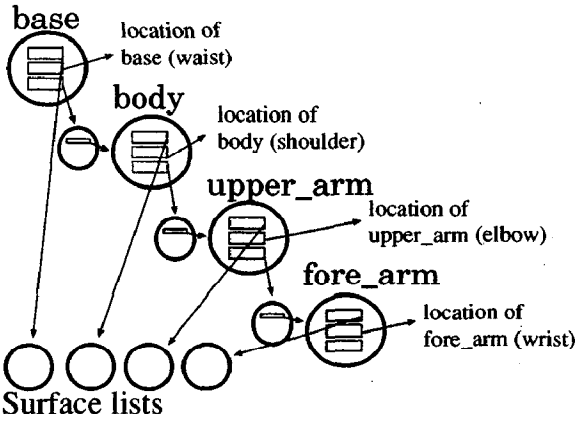

Figure 9: The data structure of the robot manipulator

The data structure are shown in Figure 7. An object `cylinder` contains all the geometric information about itself. An object `cylinder` is defined by a pointer `surf_list` to a list of a surfaces, a pointer `location` to matrix which specifies the cylinder's co-ordinate system, and a pointer `sub_objs` to a list of pointers (in this case, `null`).

MOVE can represent a segmented figure of rigid bodies using `Solid` object. A robot manipulator is a typical example of the segmented figure of rigid bodies (Figure 8). The manipulator is made up of four parts. They are `base`, `body`, `upper_arm` and `fore_arm`. If the `body` moves, `upper_arm` and `fore_arm` also move according to the motion of `body` (Figure 9). But the movement of `base` is not affected by `body`. The location of each part is automatically calculated according to the spatial relationships among the neighbor parts.

Users can combine three dimensional objects to build a structured object. MOVE has seven primitive three dimensional classes - `Block`, `Cube`, `Cone`, `Cylinder`, `Torus`, `Ellipsoid` and `Sphere` (Figure 10). Users can define a

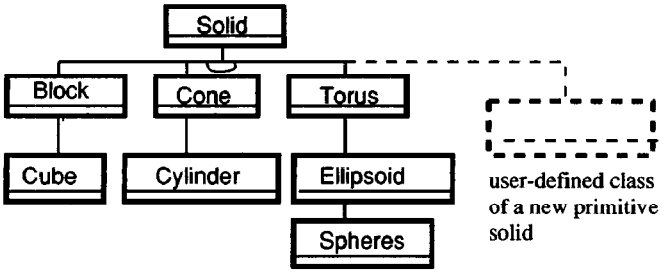

Figure 10: The class hierarchy for representing a solid

new class of three dimensional primitive object: 1) The user defines new class. 2) The user defines the constructor, which is a special method. The constructor is always invoked when a instance is created. 3) The user specifies the shape of the object in constructor. For example, the user specifies the shape of a cylinder as follows:

```
SampleSolid::SampleSolid( int resolution )
{
  ...
 for (i = 0; i <= resolution ; ++i)
   Surface* s->push( x[i],y[i],z[i] );
 this->addSurface(s);
}
```

Finally, the user can define a class of a primitive three dimensional object without knowing the details of the class `Solid` such as geometric modeling and rendering process.

4.3 Illumination Model

Kajiya has pointed out that the illumination process can be formulated as an integral equation[8]. A typical local reflection model is as follows:

$$i(x,x') = \int r(x,x',x'')l(x',x'')dx''$$

Here, $i(x,x')$ is the intensity of light which is incident from the point x' and reaches the point x. The function $r(x,x',x'')$ is the surface bidirectional reflectance function and the function $l(x',x'')$ is the incoming light intensity distribution.

The functions $r(x,x',x'')$ and $l(x',x'')$ is implemented separately.

The routine for the term $r(x,x',x'')$ is called *surface shader*. And the routine for term $l(x',x'')$ is called *light-source shader*.

Light-source shader and surface shader are also objects in MOVE. It is impossible to solve above integral equation and to calculates the solution precisely from two objects, light-source shader and surface shader. So, the authors made one assumption about light shader. The light shaders are approximated by four coefficients, ambient, intensity, direction and color. All the information about light-sources are obtained through these three methods. The assumption is too restrictive to model natural light-source, but the illumination model based on this assumption is good approximation of natural illumination process.

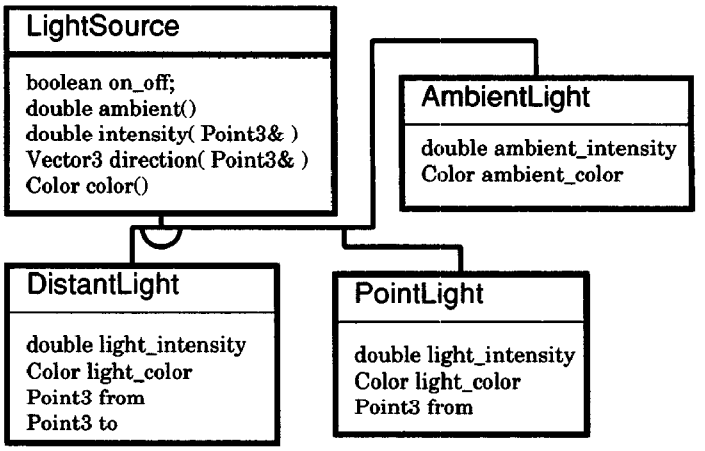

Figure 11: The design of light-source classes

4.3.1 LightSource object

Figure 11 shows the design of the class `LightSource`. The class `LightSource` is a super class for other light-source classes.

- ambient light
  An ambient light is uniformly incident and is reflected equally in all directions by the surface.
- distant light
  All the rays of a distant light come from the same direction. The sun is an example of distant light-source.
- point light
  The rays of a point light-source come from a single point.

The implementation of four methods `ambient()`, `intensity()`, `direction()` and `color()` represent the character of each light-source. For example, in the case of a point light, the intensity of light is:

$$\| l(x',x'') \| = \frac{I}{4\pi \| x' - x'' \|^2}$$

This equation can be implemented as follows:

```
double PointLight::intensity( Point3& p )
{
    return light.intensity /
           4 * pi * ( p-from ) * ( p-from );
};
```

surface material	shading function $r(x,x',x'') =$	what is in DB
constant	K_a	K_a
matte	$K_a + K_d\theta$	K_a, K_d
metal	$K_a + K_s \cos^n \alpha$	K_a, K_s
plastic	$K_a + K_d\theta + K_s \cos^n \alpha$	K_a, K_d, K_s
texture map	$f(x')$	$f(x')$

Table 2 : Five primitive `SurfaceShader` object

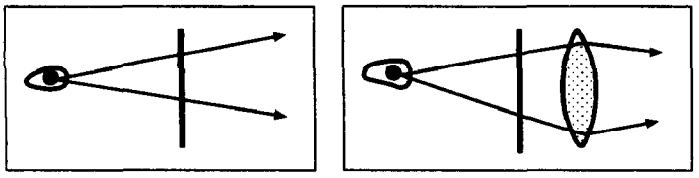

Figure 12: The eyepoint and viewplane camera model. a) without a lens. b) with a simple lens.

4.3.2 SurfaceShader object

The class `SurfaceShader` represents a material of surface. Five typical sub classes of `SurfaceShader`, constant, matte, metal, plastic and texture map are appeared in Table 2. The parameters are stored in the database (in the case of texture map shader, sampled values of $f(x')$ are stored).

Shading classes are defined corresponding to each kind of shading functions. The user can define a new class of surface shader by specifying new shading function.

4.4 Camera object

There are several camera models proposed[4]. Our camera model consists of the eyepoint and viewing transformation A `Camera` object can change the type of the lens (Figure 12).

5 OBJECT INSTANCING

When the user create a new object, instancing mechanism is always used. MOVE has two kinds of instancing mechanisms. One is a class constructor and the other is a virtual constructor.

The class constructor is an ordinal constructor in C++. The *virtual constructor* allows the user to construct a new scene object from other objects already stored in the database. The class constructor requires the name of the class which is to be constructed. So, the user can't use the class constructor to create a new object from other objects in the database, because the user can't know the name of the class until run-time. But, the virtual constructor is implemented as object's method `instancing()`. The users don't have to know the name of class.

In the *virtual constructor* method, the same size of memory as the original object are allocated and the contents are copied. The pointer to the table of method (i.e. *vtbl*) are also copyed.

6 Towards Dynamics Animation

The following procedure generates a series of a hundred of images (i.e. animation). It can be written simply using instancing mechanism.

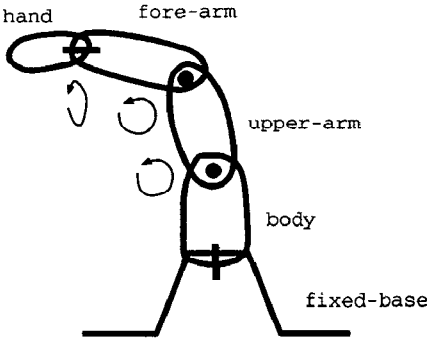

Figure 13: The kinematic structure of the robot

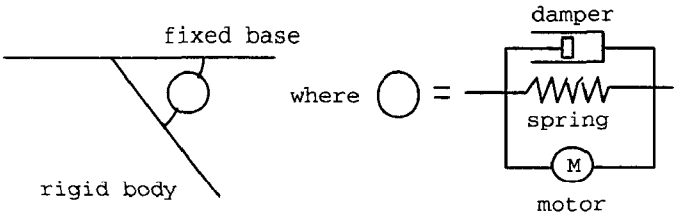

Figure 14: Dynamics elements

```
initialize scene[100] and current_scene;
for ( i = 0; i < 100; ++i )
  scene[i] = current_scene.instancing();
update current_scene;
```

The "*update current scene*" involves kinematic constraints(Figure 13) and Dynamics elements(Figure 14). Arnaldi et al. discussed about the importance of dynamics in animation[1].

Our basic idea is to extend the `Solid` class and to define new classes for kinematic and dynamics simulation. The manipulator has 4 degrees of freedom, then the manipulator can by represented by four variables. The new class maintains the relationships between the four variables and the location of each segment. The relationships between a motor and each segment are also maintained. These are all in object-oriented way.

7 CONCLUSION

In the area of simulation-based computer animation, animation systems manage all the state variables and physical laws of animated entities. To manage animation data easily, animation database is necessary.

The authors designed new animation data model using object-oriented concepts. They authors implemented new animation database system and tested our animation model.

Several interface classes have been defined. Programmers of MOVE can define a new class using programming language C++. If the definition matches the interface

class, the users can use the new class as well as pre-defined class.

The authors discussed about the extensibility of MOVE in defining classes of three dimensional primitive objects and illumination models. The next step of MOVE is kinematic constraints and dynamics animation using MOVE's extensibility.

ACKNOWLEDGMENT

The authors would like to thank Yusuke Kondo and Katuhiko Kikkawa Yusuke Kondo converted the classes in C++ into the database classes. Katuhiko Kikkawa tested the rendering routines in the MOVE. This work was partially supported by the Japanese Ministry of Education, Science and Culture under Grant-in-Aid for Scientific Research(B) (Grant-No. 03858007).

REFERENCES

[1] Arnaldi B, Dumont G, Hégron G, Magnenat-Thalmann N, Thalmann D, "*Animation control with dynamics* In:*State-of-the-art in computer animation*", Springer, pp.113-124, 1989.

[2] Bjarne Stroustrup, "*the C++ Programming Language 2nd edition*", Addison-Wesley, 1991.

[3] Eric Grant, Phil Ambum, and Turner Whitted, "*Exploiting Classes in Modeling and Display Software*", IEEE CG & A, November, pp.13-20, 1986.

[4] Geoff Wyvill and Craig McNaughton, "*Optical Models.* In:*CG International '90*", Springer-Verlag, 1990:

[5] Hans Rijpkema and Michael Girard, "*Computer Animation of Knowledge-Based Human Grasping*", ACM Computer Graphics, vol.25, no.4, pp.339-348, July, 1991.

[6] James Foley, Andries van Dam, Steven Feiner, John Hughes, "*Computer Graphics Principles and Practice 2nd edition*", Addison-Wesley, 1990.

[7] James K. Hahn, "*Realistic Animation of Rigid Bodies*", ACM Computer Graphics, vol.22, no.4, pp.299-308, August, 1988.

[8] Kajiya, James T., "*The Rendering Equation*", ACM Computer Graphics 20(4), pp. 143-149, August, 1986.

[9] W. Kim and F. H. Lochovsky, "*Object-Oriented Concepts, Databases, and Applications*", ACM Press, 1988.

[10] Kunihiko Kaneko, Susumu Kuroki, and Akifumi Makinouchi, "*Design of 3D CG Data Model of "MOVE" Animation Database System*", Proc. the 2nd Far-East Workshop on Future Database System, Kyoko, 1992.

[11] D Thalmann, "*Motion Control: From Keyframe to Task-Level Animation* In:*State-of-the-art in computer animation*", Springer, pp.3-18, 1989.

[12] Mangnenat-Thalmann N, Thalmann D, "*The Use of High-Level 3D Graphical Types in the MIRA Animation System*", IEEE CG & A, pp.9-16, 1983.

[13] Magnenat-Thalmann N, Thalmann D, "*CINEMIRA: a 3D computer animation language based on actor and camera data types*", Technical Report, University of Montreal, 1984.

[14] N.Magnenat Thalmann, D.Thalmann, "*Computer Animation. Theory and Practice. Second Revised Edition*", Spring-Verlag, 1990.

[15] Mark Green and Hanqiu Sun, "*A Language and System for Procedural Modeling and Motion*", IEEE CG & A, November, pp.52-64,1988.

[16] Mohammed Mahieddine and Jean Claude Lafon, "*An Object-Oriented Approach for Modeling Animated Entities* In:*Computer Animation'90* ", Springer-Verlag, pp.177-187, 1990.

[17] Myeong W. Lee and Tosiyasu L.Kunii, "*Design Methodology for Computer Animation Database System*", Proc. DASFAA, pp.73-79,1989.

[18] Myeong W. Lee and Tosiyasu L.Kunii, "*Animation Platform: A Data Management System for Modeling Moving Objects* In:*Computer Animation'91*", Springer-Verlag, pp.169-186, 1991.

[19] Ontologic Inc., "*Ontos Object Database version 2.0 Developer's Guide*", Ontologic Inc., Burlingtion Mass, Feb., 1991.

[20] Parke FI, "Parameterized Models for Facial Animation", IEEE Computer Graphics and Applications, vol. 2, no. 9, pp. 61-68, 1982.

[21] Perlin, k., "*An Image Synthesizer*", ACM Computer Graphics, 19(3), pp.287-291, 1985.

[22] Peter Coad, Edward Yourdon, "*Object-Oriented Analysis 2nd edition*", Prentice Hall, 1991.

[23] Pixar Corporation, "*The RenderMan Interface Version 3.0*", Pixar Corporation, San Rafael CA, May, 1988.

[24] R.J.Rost, "*PEX Introduction and Overview, PEX Version 3.20*", MIT X Consortium, 1988.

[25] Scheifler, R.W., J. Gettys, and R. Newman, "*X Window System*", 1988.

[26] Toshiyasu L.Kunii and Linig Sun, "*Dynamic Analysis-Based Human Animation* In:*CG International '90*", Springer-Verlag, pp.3-15, 1990.

[27] Tsukasa Noma and Tosiyasu L. Kunii, "*ANIMENGINE: An Engineering Animation System*", IEEE CG & A, October, pp.24-33, 1985.