3.1 Introduction

The first international standard on computer graphics, the Graphics Kernel System (GKS) [ISO 1985] appeared in 1985. It provides a functional interface between the application program and multiple input and output devices. It is a two-dimensional system. This was followed by a related standard for three-dimensional graphics GKS-3D [ISO 1987] in 1987.

Both GKS and GKS-3D standards enable graphical data to be grouped into segments. Segments can be manipulated as single entities. That is, segments can be deleted, made visible or invisible or highlighted. Once a segment is created its contents cannot be modified. That is, existing primitives in the segment cannot be deleted or modified and new primitives cannot be inserted. Transformations can be applied on segments. These operations affect the entire segment, thereby affecting all the primitives within the segment.

A picture may be constructed from a number of disjoint segments. This is because a segment cannot include or contain references to other segments, thus
providing a single-level or flat graphics data structure. Therefore a complex picture is flattened. In contrast to segments, output primitives cannot be addressed or manipulated as individual entities.

Since the structure of a complex picture is hierarchical, not flattened, it is desirable to retain the original hierarchical structure at the graphical level and therefore the segment model of GKS is not sufficient. The Programmer's Hierarchical Interactive Graphics System (PHIGS) standard [ISO 1986] overcomes these problems.

PHIGS is designed to support applications which need highly dynamic displays. The graphical data in PHIGS is organized into structures which consists of a sequence of structure elements. Structures may refer to other structures. This enables handling of hierarchical relationship among graphics data and manipulating geometrically related objects. It provides functions for editing the contents of a structure. But the editing functions are very limited, they only allow structure elements in a structure to be deleted, inserted or replaced. PHIGS does not allow editing the contents of individual elements. For example, it is not possible to edit a point of a polyline or to add a new point to a polyline. To achieve this the entire element must be regenerted.
Both in GKS and PHIGS, entities which are not defined as separate cannot be accessed and manipulated as individual entities. Also, it is not possible to assign names to output primitives and to manipulate them as individual items. That is, both systems distinguish between output primitives and segments/structures. It is not possible to assign attributes to a collection of graphics primitives by name or identifier. For example, if we have to assign colour "red" to a group of primitives, we have to open a segment/structure and then set the attribute colour to "red" and then invalidate the attribute by assigning a colour that is different from "red". We cannot use the identifier for this purpose. Also, it is not possible to assign attributes such as visibility, highlighting, etc. to individual primitives. For example, if an individual primitive has to be made visible or invisible, we have to create a segment/structure consisting of only one primitive.

Further, it is not possible to impose constraints on graphical data. For example, we cannot restrict the character size to a minimum for legibility. Constraints will be discussed in Chapter 4.

The creation and manipulation of individual objects, their geometry and attributes are very important in any graphics systems of today. For this it is necessary to be able to address and access individual objects. Further, it is also important to establish relationship among
graphical objects and define operations on them. Object-oriented paradigm as a basic concept provides these features and so it is becoming very popular.

The emergence of object-oriented languages and systems have added a new dimension in the field of computer graphics. The Smalltalk-80 [Goldberg 1983], [Goldberg 1984] is a typical representation for supporting graphics in object-oriented programming environment. It is a raster graphics system and supports direct pixel manipulation. All graphics primitives are objects and can be called by name and referred by other objects. They can be directly assigned attributes and can be manipulated individually. In addition to graphics primitives, Smalltalk-80 also provides special classes to generate windows, menus and dialog boxes. These features are particularly relevant to the development of user interfaces [Barth 1986], [Myers 1989]. Smalltalk-80 graphics systems do not support hierarchical picture structure and therefore it has to be implemented with available object-oriented programming tools.

Smalltalk-80 operates in 2-D screen coordinates unlike GKS and PHIGS which operate in world coordinates. Also, unlike GKS/PHIGS it does not provide functions for device independent graphics. Smalltalk-80 does not maintain a database of structures as in PHIGS. Unlike Smalltalk-80, GKS and PHIGS use floating point coordinate systems. In general, Smalltalk-80 does not support the
range of functionalities that is expected from a graphics standard. Therefore, it cannot be called complete graphics system and it cannot compete with the graphics standards GKS and PHIGS. They address different needs and applications, and therefore no single graphics package meets all the needs. It is therefore necessary to synthesize between the existing object-oriented graphics and functionalities provided by graphics standards. In this chapter we propose a new scheme to achieve this.

In our scheme [Bhalla 1991a] all graphical entities are treated as objects, which can be addressed and manipulated independently. Objects are supported as instances of classes, which in turn, are organized into 'is-a' hierarchy. In addition to 'is-a' hierarchy, the scheme also supports part hierarchies. The declarative approach to graphics allows us to represent and manipulate objects at the same level of abstraction at which the user thinks and not at the level of pixels or drawing routines. It allows us to specify the contents of the picture in terms of objects that are to appear on the screen. The contents of the pictures are stored in the database. This enables us to share representation of pictures. Also, our scheme supports multiple output devices. Further, in our scheme graphics data is specified in world coordinates.
We allow constraints to be imposed on individual objects, between two or more objects, on a class of objects, on the relationship between window, viewport and device, and part hierarchies.

3.2 DECLARATIVE APPROACH TO GRAPHICS

In the procedural approach (the most common method) a picture is described by specifying the coordinate values of output primitives. Pictures are typically represented procedurally as a series of calls to graphics drawing routines (or commands), and a series of transformation routines (or commands) which manipulate pixel areas of raster device [Foley 1984]. For example, consider a window that displays a pyramid and a line. The procedural approach for displaying this will in general be as:

WINDOW(0.0,0.0,0.0,100.0,100.0,100.0)
/ Specifies x-min, y-min, z-min and x-max, y-max and z-max coordinates. /

VIEWPORT(0.0,0.0,799.0,479.0)
/ Specifies x-min, y-min, x-max and y-max coordinates of the display surface. /

SETCOLOUR("RED") / Sets the colour to "red" /

POLYLINE( <10.0,10.0,10.0>, <30.0,10.0,10.0>, <30.0,20.0,10.0>, <10.0,20.0,10.0>, <10.0,10.0,10.0> )

POLYLINE( <10.0,20.0,10.0>, <20.0,15.0,40.0> )

POLYLINE( <10.0,10.0,10.0>, <20.0,15.0,40.0> )

POLYLINE( <30.0,10.0,10.0>, <20.0,15.0,40.0> )

POLYLINE( <30.0,20.0,10.0>, <20.0,15.0,40.0> )

POLYLINE( <30.0,20.0,10.0>, <20.0,15.0,40.0> )
This method has some advantages. First, a few types of output primitives are sufficient to produce a large variety of shapes. For example, a primitive 'polyline' can be used to describe a triangle, quadrilateral, hexagon, etc., depending upon the number of points and coordinate values. Further, shapes such as circles, ellipsoids and other smooth-curved shapes can also be described by taking large number of points and short edges. Second, this method does not require a special picture description language [Noma 1989] and can be realized as a set of library routines to be invoked in a high-level programming language such as FORTRAN, Pascal and C.

Because of these advantages, the graphics standard GKS and PHIGS adopt procedural approach as they require the coordinate of output primitives to be specified by the user either directly or indirectly through a set of formulae. Also, in most graphics systems a picture description is converted to this type of description at the final stage of output.

However, in procedural approach, the system assumes that the user has knowledge of difficult mathematical functions such as square roots and trigonometrical functions and knows how to handle formulae including some
As more and more people want to use computer graphics, declarative approach is found to be more useful especially for the non-specialists. In the declarative approach [Tarlton 1989] one specifies what has to appear on the screen and consequently what modifications are required, rather than how the picture is displayed and modifications are performed. The picture contents are described in terms of objects that are to appear on the screen. The user manipulates objects of the picture, rather than manipulating pixel areas. For example, if an object has to be moved from one portion of the screen to another, in the declarative approach one need to assert the new location and not that the object has to be erased and then redrawn. Using declarative approach we represent the above picture as follows:

```plaintext
define pyramid P1
   location=(10.0,10.0,10.0)
   xsize=10.0 ysize=20.0 height=20.0

define line L1
   start=(50.0,10.0,10.0) end=(50.0,30.0,10.0)

define window W1
   location=(0.0,0.0,0.0)
   xsize=100.0 ysize=100.0 zsize=100.0
   contents=(P1,L1)
```

The difference between the two approaches can be appreciated more when we desire to change the picture by modifying some parts of the drawing. To do this, in the procedural approach, the drawing procedure has to be modified, which may be a difficult task. In addition to
this, the overhead of having to redraw everything even when a small change is made may be significant.

For example, to draw a pyramid, the user has to specify five points and it is his/her responsibility to ensure that these points give rise to a pyramid. To draw a regular pyramid whose base is parallel to \( z=0 \) plane (i.e., in the \( x-y \) plane), the user has to test that the following conditions are satisfied.

(a) z-component of the vertices \( A, B, C \) and \( D \) of the base are equal.

(b) z-component of the fifth vertex \( E \) is different from z-component of vertices of the base.

(c) \( x_E = (x_A + x_B)/2, \quad y_E = (y_A + y_D)/2, \quad z_E \neq z_A \).

Now, suppose that we want to modify the length of the base of the pyramid in \( x \)-direction and height to twice that of the old pyramid. Then, for the construction of new pyramid, we have to calculate the coordinates of the new vertices \( B', C' \) and \( E' \).

As seen above, in case of modification of height of a pyramid the user has to calculate the new location of the fifth vertex in a procedural approach. Further, if the fifth vertex lies on the base, then two actions can be taken. One is checking before modification and the other is to allow modification and sending error message at the time of displaying. In the declarative approach, we represent a pyramid by specifying the size of the base and
its height. With this approach it is much easier to ensure that height never becomes zero.

3.3 OBJECT-ORIENTED GRAPHICS DATABASE

In our scheme all items relevant to graphics systems are treated as objects. In the object-oriented graphics systems [Jiang 1989], [Wisskirchen 1990] point, line, circle, etc. are supported as objects. In our scheme, we treat window, viewport and device also as objects. We use standard definitions of window and viewport [Foley 1984]. A window is a rectangular region in world coordinate system. A viewport is a rectangular portion on the screen onto which window is mapped. Viewport has position, size, background colour and border properties. This is in contrast to the several screen-oriented text editors and raster graphics systems in which a window is an area on the screen.

In our scheme, each graphical object can be addressed by name/identifier. Attributes can be assigned to objects as part of their own inherit information. Therefore, the values of the attributes can also be inquired directly. Attributes of an object can be geometric such as 'location'; those related to appearance such as 'colour'; or non-graphic such as 'model-number'. 
Objects can be selected and manipulated individually. That is, they can be deleted, modified, transformed, etc. Thus, even primitives can be edited. For example, it is possible to append a new point to a polyline.

An object may be referred by any number of objects. This is very important in computer graphics as it allows us to use predefined graphical objects. For example, it may be necessary to use the same point as the coordinate of two or more output primitives such as line, rectangle.

Objects with same structure (attributes) and behaviour (operations) are regarded as instances of one class. Classes are organized in a class hierarchy called 'is-a' hierarchy. Properties (attributes and methods) are inherited from superclass to subclasses through "is-a" hierarchy. The root of the hierarchy is a class OBJECT as shown in Figure 3.1. Here figure, lines with arrow point from subclass to superclass. The graphical objects that inherit from GRAPHOBJ have spatial attributes such as size, location, orientation, and have visible appearance. Window is a special type of 'GRAPHOBJ' that 'displays' itself by displaying its contents.

Class SHAPE has display attributes such as colour, linestyle, highlight in addition to spatial attributes. It has three subclasses PRIMITIVE, COMPLEX and GROUP. Thus, there exists three types of relations among graphical objects. These are "is-a", "part-of" and
FIG. 3.1: HIERARCHICAL COMPOSITION OF GRAPHICAL OBJECTS.
Primitives such as line, rectangle, etc., are created as objects which are instances of subclasses of class PRIMITIVE. We provide separate classes for each type of primitive. Class hierarchy allows sharing of methods by related primitives. It also helps in setting default values for undefined attributes. Operations such as scale, rotate and translate are inherited by the subclasses through class hierarchy. Operations such as draw, area, perimeter have to be reimplemented. New primitives, for example arrow and grid can be added.

The scheme also allows collection of graphical objects to be grouped together. The semantics of grouping objects is different from that of complex objects, in which case the parent object "owns" the component objects. An object can be formed by collecting other objects. In this case, a group object refers to its member objects but does not own them. For example, we can form a group from three circles and a triangle. Also, we can form a group from complex objects 'tricycle' and 'house'. To accomplish this classes 'CIRCLES-TRIANGLE' and 'TRICYCLE-HOUSE' are created as subclasses of class GROUP.

Objects may contain other objects as components. These complex objects are created as subclasses of class COMPLEX. This captures the semantics of part-hierarchies. For example, complex object 'robot' and 'locomotive' are
created as instances of class ROBOT and LOCOMOTIVE, which in turn are subclasses of class COMPLEX.

Our scheme works in world coordinates. Also, all objects are stored in the database for long-term storage.

To make an object appear on the screen, the user has to add it to the contents of the window. Removing an object from the window makes it disappear from the screen. Therefore, except calling display routines, we never call draw or erase routines directly. These routines are called internally. In this respect, our scheme departs significantly from other object-oriented graphics systems [Jiang 1989], [Harder 1987], [Noma 1988], [Zhu 1988].

To display a picture the following steps are followed:

1. Define the objects contained in the picture. If the objects are already stored in the database they need not be defined again.

2. Define the window and the viewports where the window has to be mapped. A window is defined by specifying its size and its contents, i.e., objects contained in it. A viewport is defined by specifying its size and the device.

3. A call to DISPLAY window function will display the picture on the specified devices.
The schema of graphics database is shown in Figure 3.2. In this figure, a class is represented by a box containing two horizontal lines. The first section contains the class name. The second section contains a list of instance variables. An arrow points to the domain of the class of the instance variable. A single arrow indicates that the instance variable is single-valued. Similarly, a double arrow indicates that the instance variable is set-valued.

The relationship between WINDOW and VIEWPORT is captured by instance variable "destination" in class WINDOW and instance variable "origin" in class VIEWPORT [Loomis 1987], [Zaho 1988]. A VIEWPORT object is associated with a DEVICE object through the instance variable "device" in VIEWPORT.

A window can be projected onto many viewports of one or more devices as shown in Figure 3.3. If a window is projected on different viewports of the same device, then this has the effect of displaying the picture at different locations of the physical device. If a window is projected onto two or more viewports corresponding to different physical display devices then this has the effect of displaying the same picture on different physical devices.
FIG. 3.2: SCHEMA OF GRAPHICAL DATABASE

USING DECLARATIVE APPROACH
FIG. 3.3: A WINDOW MAPPED TO DIFFERENT VIEWPORTS.
A viewport can have multiple windows projected onto it as shown in Figure 3.4. This allows multiple pictures to be overlapped. For example, a window may contain objects of a picture and another window may contain text (e.g., names of objects). Projecting the two windows on to the same viewport produces the super-imposed effect.

The contents of a picture belong to class GRAPHOBJ. This enables a window to contain another window as shown in Figure 3.5.

Contents of a picture are specified in terms of objects that are to appear on the screen and they are also stored in the database. This enables sharing of representation of a picture whereas in most graphics systems the final drawings (pictures) are shared [Enderle 1984].

3.4 Classes

In our scheme, the following classes have been identified. The class definition includes instance variables, operations and constraints. We note that, unlike Smalltalk-80 our scheme includes constraints in the class definition. Constraints are discussed in Chapter 4.

Class VIEWPORT
origin : set of WINDOW
device : DEVICE
location : POINT
xsize : real
ysize : real
colour : string
bordercol : string
operations
display(v:VIEWPORT) —> action
FIG. 3.4: DIFFERENT WINDOWS MAPPED TO A VIEWPORT OF A DEVICE.
FIG. 3.5: WINDOW CONTAINING WINDOW.
Class DEVICE
dname : string
number : integer
xmaxsize : real
yminsize : real
status : string

Class WINDOW
location : POINT
xsize : real
ysize : real
destination : set of VIEWPORT
contents : set of GRAPHOBJ
operations
display(w:WINDOW) → action

Class SHAPE
position : POINT
size : real
orientation : real
colour : string
linestyle : integer
highlight : boolean
operations
rotate(s:SHAPE) → SHAPE
scale(s:SHAPE) → SHAPE
translate(s:SHAPE) → SHAPE

Class COMPLEX
operations
children(p:COMPLEX) → set of COMPLEX
/ returns immediate components of a part /
allcomp(p:COMPLEX) → set of COMPLEX
/ returns all components of a part /
gshape(p:COMPLEX) → set of PRIMITIVE
/ returns geometric shapes used to construct part p /
copy(p:COMPLEX) → COMPLEX
/ returns an object which is a copy part p /
complevel(p:complex, n: integer) → set of COMPLEX
/ returns all components of a part upto level n
parent(p:COMPLEX) → COMPLEX
/ returns parent of part p /
draw(p:COMPLEX) → action

Class GROUP
operations
members(g:GROUP) → set of SHAPE
copy(g:GROUP) → GROUP
draw(g:GROUP) → action
Class GRID
xstart : POINT
ystart : POINT
xlength : real
ylength : real
xinterval : real
yinterval : real
operations
draw(g:GRID) → action

Class POLYLINE
numpoints : integer
xarray : ARRAY of real
yarray : ARRAY of real
operations
draw(p:POLYLINE) → action

Class POLYGON
fillpattern : integer
fillcolour : integer
operations
draw(p:POLYGON) → action

Class TEXT
location : POINT
height : real
textstring : string
operations
draw(t:TEXT) → action

Class POINT
xcor : real
ycor : real
operations
draw(p:POINT) → action
distance(p1,p2:POINT) → real

Class LINE
tail : POINT
head : POINT
operations
length(l:LINE) → real
draw(l:LINE) → action
Class CIRCLE
centre : POINT
radius : real
operations
draw(c:CIRCLE) → action
circumference(c:CIRCLE) → real
area(c:CIRCLE) → real

Class RECTANGLE
location : POINT
xsize : real
ysize : real
operations
draw(r:RECTANGLE) → action
perimeter(r:RECTANGLE) → real
area(r:RECTANGLE) → real

Class ARROW
location : POINT
xlength : real
ylength : real
direction : real
operations
draw(a:ARROW) → action

Such a representation can easily be extended to 3D scenes. For this we create a class GRAPHOBJ3D as subclass of OBJECT, which has subclasses WINDOW3D and SHAPE3D. Then, we create three subclasses PRIMITIVE3D, COMPLEX3D and GROUP3D of class SHAPE3D (Figure 3.6). For example, class WINDOW3D defines the window. Primitive classes such as POINT3D, LINE3D, etc., are represented similarly. These primitive classes inherit from class SHAPE3D. The classes VIEWPORT and DEVICE remain unchanged.

Class WINDOW3D
location : POINT3D
xsize : real
ysize : real
zsize : real
destination : set of VIEWPORT
contents : set of GRAPHOBJ3D
operations
display(w:WINDOW3D) → action / Isometric projection /
FIG. 3.6: HIERARCHICAL COMPOSITION OF 3-D GRAPHICAL OBJECTS.
Class SHAPE3D
position : POINT3D
size : real
orientation : real
colour : string
linestyle : integer
highlight : boolean
operations
rotate(s:SHAPE3D) → SHAPE3D
scale(s:SHAPE3D) → SHAPE3D
translate(s:SHAPE3D) → SHAPE3D

Class COMPLEX3D
operations
children(p:COMPLEX3D) → set of COMPLEX3D
allcomp(p:COMPLEX3D) → set of COMPLEX3D
gshape(p:COMPLEX3D) → set of PRIMITIVE3D
copy(p:COMPLEX3D) → COMPLEX3D
complevel(p:COMPLEX3D, n: integer) → set of COMPLEX3D
parent(p:COMPLEX3D) → COMPLEX3D
draw(p:COMPLEX3D) → action

Class GROUP3D
operations
members(g:GROUP3D) → set of SHAPE3D
copy(g:GROUP3D) → GROUP3D
draw(g:GROUP3D) → action

Class POLYLNE3D
numpoints : integer
xarray : ARRAY of real
yarray : ARRAY of real
zarray : ARRAY of real
operations
draw(p:POLYLNE3D) → action

Class POLYGON3D
fillpattern : integer
fillcolour : integer
operations
draw(p:POLYGON3D) → action

Class POINT3D
xcor : real
ycor : real
zcor : real
operations
draw(p:POINT3D) → real
distance(p1,p2:POINT3D) → real
Class LINE3D
  tail : POINT3D
  head : POINT3D
  operations
  length(l:LINE3D) ⦿ real
  draw(l:LINE3D) ⦿ action

Class SPHERE
  centre : POINT3D
  radius : real
  operations
  draw(s:SPHERE) ⦿ action
  surface-area(s:SPHERE) ⦿ real
  volume(s:SPHERE) ⦿ real

Class CUBOID
  location : POINT3D
  xsize : real
  ysize : real
  zsize : real
  operations
  draw(c:CUBOID) ⦿ action
  surface-area(c:CUBOID) ⦿ real
  volume(c:CUBOID) ⦿ real

Class ARROW3D
  location : POINT3D
  xlength : real
  ylength : real
  zlength : real
  direction : real
  operations
  draw(a:ARROW) ⦿ action

Class PYRAMID
  location : POINT3D
  xlength : real
  ylength : real
  height : real
  operations
  draw(p:PYRAMID) ⦿ action
  surface-area(p:PYRAMID) ⦿ real
  volume(p:PYRAMID) ⦿ real
3.5 Assignment of Attributes

In our scheme, we impose rules while assigning attributes to objects. First, we note that an attribute can be assigned to a primitive object. Second, an attribute can be assigned to a group which we call 'group attribute'. Third, an attribute can be attached to a part which we call 'part attribute'. These rules are stated below:

1. Attributes assigned to a group have higher priority than attributes assigned to primitive objects.

2. Attributes assigned to a part have higher priority than attributes assigned to primitive objects used for constructing parts.

3. Attributes assigned to components have higher priority than attributes assigned to the parent. This rule ensures that more specific the object is higher is the priority attached to its components.

4. Group attributes have low priority compared to part attributes. For example, if a group contains a tricycle and two rectangles, then the group attribute can change the colour of the rectangles, but not of tricycle.
3.6 Modelling of Part Hierarchies

Class hierarchy and part hierarchies are two very important concepts in computer graphics. In the last section, we discussed class hierarchy which expresses a behaviour relation. In this section we discuss part hierarchy which expresses composition relation.

The basic idea behind modelling part hierarchies is the construction of complex objects. Building complex object is accomplished by suitable arrangement of primitive shapes which in turn is used to construct yet higher level objects. We use lines, rectangles, polygons, ellipses, etc, as primitive shapes to model part hierarchies. Also, for 3D graphics, shapes such as cylinders, parallelepipeds, spheres, pyramids, etc., are used as primitive shapes.

We know that a complex object is an object with a hierarchy of component objects. For example, object 'tricycle' consists of components 'front', 'body' and 'back'. Similarly, object 'front' consists of components 'wheel', 'handle', and so on (Figure 3.7). In this case object 'tricycle' is clearly the root object.

Since any object has to be an instance of a class, the classes corresponding to the root of a complex object and its components form a hierarchy of classes. This hierarchy of classes is called complex object schema [Banerjee 1987]. The complex object schema for 'tricycle'
FIG. 3.7: COMPLEX OBJECT 'TRICYCLE'.
is shown in Figure 3.8. Here, all classes except the root class are component classes. We define the leaves of this hierarchy as the primitive component classes and instances of these classes are called primitive components. Also, in this figure, lines with arrow originate from component classes and point to the parent classes. For example, object 'tricycle' belongs to class TRICYCLE. Similarly, objects 'front', 'body', 'back' belong to classes FRONTPART, BODY and BACKPART respectively. We allow components of two or more objects to belong to same class, but we do not permit one component to be owned by two parents. Thus, a part hierarchy is viewed as a tree, whereas classes to which complex object and its components belong to will form a directed acyclic graph. This enables the manipulation of components individually. For example, the wheels of a tricycle belong to class WHEEL, but they are three distinct objects. Knowledge about the similarities in shape of the wheels is maintained by the constraints and these are discussed in Chapter 4.

Primitive components have geometric description which is a primitive shape. Two or more primitive components (belonging to same or different part hierarchies) may share the same primitive shape. In this case a primitive component 'refers' [Banerjee 1987] to a primitive shape but does not 'own' it. For example, the wheel hubs may refer to same sphere object. A change in the value of an attribute of a primitive shape object is automatically
FIG. 3.8: COMPLEX OBJECT SCHEMA FOR 'TRICYCLE'.

propogated to all higher-level objects that use it.

Classes to which complex objects belong to are created as subclasses of class COMPLEX which is a subclass of class SHAPE as shown in Figure 3.1.

Component objects must be positioned at the right place in the parent object. In order to fit properly in the parent object, they may have to be re-sized and re-oriented. Therefore, for every component of an object, there exists an attribute in the parent object which gives the transformation to be applied to the component at the time of displaying the part. The value of this attribute belongs to class TMATRIX. This transformation is relative to the parent object. It is to be distinguished from the transformation associated with component itself (local transformation) and the transformation associated with the root of the part hierarchy (global transformation).

As mentioned before, a component can be owned by only one parent. A component may however be referred by any number of objects. For example, a picture may contain only the three wheels of a tricycle. In this case, the picture 'refers' to the wheel object but does not 'own' them.

As stated earlier, a component object depends on its parent and it cannot exist in isolation. Further, it is always integrated in exactly one part hierarchy. Thus, if a parent object is deleted then the component object is
also deleted, and in this case all references to it are also removed.

Class TRICYCLE
model : STRING
manufacturer : STRING
tfront : TMATRIX
front : FRONTPART
tback : TMATRIX
back : BACKPART
tbody : TMATRIX
body : BODY

Class FRONTPART
thandle : TMATRIX
handle : HANDLE
twheel : TMATRIX
wheel : WHEEL

Class WHEEL
trim : TMATRIX
rim : RIM
thub : TMATRIX
hub : HUB

Class RIM
g几个人ometry : CYLINDER
tgeometry : TMATRIX

Class HUB
g几个人ometry : SPHERE
tgeometry : TMATRIX

3.7 Comparison with PHIGS

Handling multi-level part hierarchies in computer graphics possess two problems. First, the system should maintain the individuality of parts. As in real life, we know that a wheel cannot belong to two cycles. Second, the system should be able to support geometrically equivalent parts. That is, the system must have
'knowledge' of these equivalent parts.

To share a picture in PHIGS, it has to be defined as a structure. For example, a picture A can be defined as a structure with calls to 'execute table' and 'execute chair' structures. In this case objects in the picture can only be 'viewed' on the screen. More precisely, a structure hierarchy can be traversed only when the structure is posted to a workstation. For example, if it is desired to access pictures that contain a 'table', they have to be posted to a device. But in our scheme, it is also possible to achieve this without having to 'see' the picture. Hence, our scheme allows sharing of representation in 'true sense'.

As mentioned earlier, PHIGS supports multi-level data structures which allow the implementation of part hierarchies. These hierarchies are organized as network of structures forming a directed acyclic graph, whose information becomes visible on the display medium only after the traversal process. The individual components in the display medium cannot be addressed because the result of a posting operation cannot be accessed from within an application program. Hence, in PHIGS a part has 'no knowledge' about its components. Thus, there is no one-to-one correspondence between structures/components and visible objects on the screen.
As a special case, when structure is not referred multiple times, the directed graph degenerates into a tree. With this representation of part hierarchy we can make one-to-one correspondance between structure and visible objects on the screen. To accomplish this it is essential that the same information be stored in many structures. In this case, knowledge about similarities in the geometrical description is lost.

For example, to create a tricycle in PHIGS we will have to define a structure wheel and execute it three times with different transformations. A change in the wheel structure will reflect all the wheels. Suppose we decide to increase the size of the hub in the front wheel, then we will have to copy wheel and hub structures into new structures. But then the knowledge that the rims of the wheels are geometrically of same size is lost. Also, representation of structures do not correspond to visible objects on the screen. It executes wheel structure three times, but it cannot distinguish them as front, left and right hind wheels. In our scheme, the components can be manipulated individually and they can share the same geometric description. We impose constraints to ensure similar geometric description on two parts. These are discussed in Chapter 4.
Our approach synthesize between these two fundamental requirements. In our scheme, a part belongs to one and only one parent. Thus, there is a one-to-one correspondance between parts and visible objects on the screen. We therefore, view a complex object as a tree. However, two or more parts (belonging to same or different part hierarchies) can share the same primitive shape. Using constraints we ensure that they always refer to the same shape.