Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

E.H. Blake

Introduction to aspects of object oriented graphics

Computer Science/Department of Interactive Systems Report CS-R9009 March

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Introduction to Aspects of Object Oriented Graphics

E.H. Blake

Centre for Mathematics and Computer Science (CWI),
Department of Interactive Systems, Kruislaan 413,
1098 SJ Amsterdam, The Netherlands.
Email: edwin@cwi.nl

Any attempt to deal with the complexity of interactive computer graphics should have a well founded and appropriate underlying abstraction. This report introduces the object oriented paradigm and critically examines its suitability as a foundation for computer graphics.

It is found to be a good basis for graphics standards and particularly useful in dynamic graphics, that is, animation and interaction. The need for extensions to overcome some shortcomings, such as the lack of a part-whole hierarchy, are discussed. A brief comparison is made with the other major abstraction for computer graphics: the functional approach.

CR Categories and Subject Descriptors:
I.3.3 [Computer Graphics]: Methodology and Techniques — Languages
I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling — Hierarchy
D.1 [Programming Techniques]

Key Words & Phrases: animation, computer graphics, functional, interaction, object oriented, graphics standards.

§1 Introduction.

This paper introduces and critically examines the object oriented paradigm as it applies to computer graphics. Object oriented programming developed from simulation languages and from projects to manage programming complexity and facilitate human-computer interaction. It has an intuitive appeal to regard the parts of a model, interaction, or animation as independent active objects; these actors (concurrent objects) having an internal state and communicating via messages. This initial appeal, elaborated to become object oriented graphics, does seem to stand up to the closer scrutiny.

It could be said that computation for graphics is complex not just in an *algorithmic* sense but also in a *programming* sense, since quite apart from the complexity of the algorithms used another kind of *programming* complexity arises due to the scale of the overall system. This is because these systems are typically very large integrated programs making use of a wide range of techniques. They deal with large and disparate databases. They allow concurrent interaction with a user and between a large number of actors.

A well founded and appropriate underlying abstraction is needed to deal with the complexity of computer graphics. The aim of any abstraction is to provide a context within which problems can easily be solved, not replacing existing techniques but instead providing a firm basis for thinking about them and implementing them.

Report CS-R9009
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Any abstraction in graphics has to face the tradition of *ad hoc* approaches to computer graphics. But this tradition has run into greater and greater difficulties as the complexity of the computer graphics problem has compounded itself [cf. Earnshaw, 1987]. Up to now, what calls there have been for an underlying theory have been for the incorporation of the laws of physics [Greenberg, 1988]. But computer graphics is more than realistic simulation, it is also a branch of computer science: this paper seeks to address that aspect.

We'll now explore the basic ideas of the object oriented abstraction and to see if they are an adequate and suitable foundation for building graphics systems.

1.1 Outline.

In this brief foretaste of the paper certain terms may be unfamiliar: the reader may take this as encouragement to read the body of the paper.

The ideal areas of application for the object oriented paradigm seems to lie in two broad areas: human-computer interaction and three-dimensional modelling and simulation. The connection between the two is that in both cases users/modellers have intuitions which correspond closely to the notion of objects within the object oriented paradigm.

Different internal data representations and algorithms often have to co-exist in graphics systems. Frequently, particularly in computer aided design (CAD) applications, multiple external views are required on the same object. The principles of data abstraction which are embodied by object oriented languages enable this to be done.

Another common requirement is the ability to handle time dependency and concurrency. Graphical objects typically persist over time while changing their internal configuration or state. Actor systems (a flavour of object oriented programming) offer a way of addressing these issues. It can be argued that in most graphical situations this abstraction is preferable to that offered by declarative approaches (i.e., functional or logic programming).

In this paper we shall now review the concepts which seem to belong to the object oriented paradigm (§2). It will become clear that there are several kinds of object oriented programming. The following section (§3) is concerned with the use of object oriented programming in computer animation and interactive graphics and CAD. The issues which arise are common to many graphical applications. This also serves to introduce some of the citations in the bibliography which provide further examples of the use of the concepts in computer graphics. We shall not be so concerned with design methodologies for object oriented graphics systems [but see: Cox, 1986; Meyer, 1988].

Why is object oriented programming useful, what are the limitations, how can these limitations be overcome? What are the alternative abstractions which one might use? These questions, which are topics of current research, are asked in §4. We shall pay particular attention to functional graphics. Finally, in the conclusion (§5) the need for a usable abstraction for graphics is stressed.

§2. The Object Oriented Paradigm.

"Object orientation" has sometimes been used rather loosely in graphics to mean nothing more than using 3-D object models internally instead of dealing only in 2-D images. This is not what is meant by the term here. We adopt the more conventional "definition" [Cardelli & Wegner, 1985; Rensch, 1982; Stefik & Bobrow, 1985] where object orientation is some combination of:

> *Data abstraction* (named interfaces and hidden local state) plus *object types* (or classes) plus *type inheritance* (attributes inherited from superclasses). Processing is done by objects sending and replying to *messages*.

Languages need not conform to all these characteristics to be called "object oriented".

Computer abstractions are expressed in languages. We can best illustrate the various object oriented approaches by looking at languages that have realized them. This has the benefit that it also introduces a concrete way of implementing the ideas.

Object oriented languages descend from Simula [Dahl & Hoare, 1973] and are exemplified by Smalltalk [Goldberg & Robson, 1983]. Some established languages have also been given object oriented features (e.g. C++ [Stroustrup, 1986], Objective-C [Cox, 1984; 1986]). In Hewitt's actor formalism [Agha, 1986; Agha & Hewitt, 1987] greater emphasis is placed on concurrency and message passing. Meyer [1988] discusses general features of object oriented programming as well as the language Eiffel.

The Doré Toolkit [Ardent, 1988; Williams, 1988 — press release] is an object based system for interactive image synthesis. The NeWS windowing system has shown how naturally PostScript supports object oriented programming for user interfaces [Adobe Systems, 1985; NeWS, 1987; Densmore & Rosenthal, 1987].

A comparative survey of some languages can be found in Micallef [1988]. Views on what object oriented programming 'really' is are variously given by Madsen & Møller-Pedersen [1988], Nygaard [1986] and Stroustrup [1988].

In this section we will now review some of the central concepts: §2.1 abstract data types, §2.2 classes and inheritance, §2.3 message passing and polymorphism, and §2.4 concurrency. Finally in §2.5 we summarize the features which combine make up object oriented programming. These features can be used to classify programming languages. Two widely available object oriented languages are Smalltalk and C++ and they illustrate two rather different approaches to object oriented programming.

2.1 Abstract Data Types.

In object oriented programming the computing process is factored into objects. Each object is comprised of data elements and procedure elements. Objects are typically instances of some class, or, equivalently, objects belong to a type. The programmer is free to define new types and their associated operations: their protocol.

Procedural abstraction means that procedures can be invoked by naming them without regard to their internals. This forms the basis of structured, modularized, procedure oriented programming.

Data abstraction is used for similar reasons in object oriented programming. The state and implementation of an object is hidden from other objects. Instead an object possesses a protocol of messages which form its only interface with the outside. In order to use an object one need only know the protocol, or equivalently, its class or type [Cardelli & Wegner, 1985; Guttag, 1977; Liskov & Zilles, 1974].

The ability to define new data types (i.e., classes or prototypes — see §2.2) is implicitly present in all object oriented languages. This takes care of one aspect of data abstraction. The other aspect, preventing an object from being accessed except via its protocol, is referred to as the principle of *information hiding*. When this principle is strictly enforced it is known as *encapsulation* [Snyder, 1986; Micallef, 1988].

2.2 Classes and Inheritance.

In most object oriented languages objects belong to classes (these classes being objects in their own right). Objects which are instances of the same class are similar in that they share the same interface and have the same structure. Class and inheritance are based on an analogy with both taxonomy and genetics.

Classes in object oriented languages form a hierarchy. Subclasses are specializations of their superclasses, and they inherit all the characteristics of the superclasses [Cardelli & Wegner, 1985; Danforth & Tomlinson, 1988]. Simple abstract classes characterize the higher levels of the hierarchy while more complex behaviours, in concretely useful classes, are found at lower levels.

The simple hierarchy of inheritance relationships can be extended to a network of relationships. This is referred to as multiple inheritance [Cardelli, 1984; Borning & Ingalls, 1982]. In some systems a class need not inherit all the features of a particular super class. This is even closer to the situation in biology where traits are distributed amongst individuals in a gene pool and can be inherited separately.

Prototypes and delegation are used in some forms of object oriented programming (e.g. actor languages [Agha, 1986]) in preference to the notion of class and inheritance. This means that an object's message protocol includes those messages which can be delegated to *prototypes* or *exemplars* [Borning, 1986b; Lieberman, 1986a & b; LaLonde, Thomas & Pugh, 1986]. For example, if we were modelling horses, the Platonic ideal horse would be a prototype and a particular horse, the nag in the field, would delegate the responses to some of its messages to that ideal prototype.

Although there was some controversy it does seem that the notions of inheritance and delegation are essentially equivalent [Stein, 1987; Lieberman, Stein & Ungar, 1987].

Another 'controversy' concerns the way inheritance can subvert encapsulation. This happens if the inheritance mechanism provides access to the private data of distant superclasses. The very fact that it might be known that an object is defined via inheritance arguably removes one of the benefits of data abstraction: the ability to change the implementation of types without global effects [Snyder, 1986; Micallef, 1988].

Class inheritance (or delegation) express an "IS-A" relation [Brachman, 1983]. Thus an integer IS-A number. Inheritance can also occur in other guises. There is another kind of inheritance down an "IS-PART-OF" hierarchy, this is an inheritance of attributes, e.g. a table's legs inherits its colour. This will be discussed in more detail in §4.1. There is also a kind of 'inheritance' up a part hierarchy. Parts provide abilities to the whole: we see because we have eyes.

2.2.1 Using Classes or Prototypes.

Programming in an object oriented language is a question of designing and implementing classes (or their equivalent in prototypes). A large problem is split into a number of hierarchies of classes.

Here we can possibly make a distinction between classes and prototypes. Classes are used when the system can be mostly designed in advance, when we can categorize most of the types of the objects. If new types appear on the fly (perhaps in CAD or AI applications) then prototypes are more appropriate.

If an object oriented language is to be used in situations where new objects of slightly different types have to be created frequently then the absence of prototypes and delegation can be regarded as a limitation* (cf. §4).

We should also distinguish between inheritance for code reuse and inheritance as a formal relationship between types. The former is an implementation issue and the latter is a behavioural specification. When we are modelling the world, inheritance in the latter sense is a natural consequence of the generalization and specialization of natural forms. Classification of objects

---

* Smalltalk lacks delegation. However it is a flexible system and delegation via message forwarding may be implemented easily. See §3.4 (ThingLab) and Blake & Cook [1987].

(e.g. atoms or animals) into related types is a fundamental scientific method.

2.3 Message Passing and Polymorphism.

Polymorphism can be used in a number of senses; rather perversely the meaning is usually that a single external operation can be applied to a variety of underlying types. For example: conventional typed programming languages allow the parameters of functions to have only one type. If this idea was strictly applied then addition would require a different function for each type of number and generalized routines, like head of an arbitrary list, would be impossible. Polymorphic languages allow for the same functions to accept many different parameter types. This idea can be inverted in a way which is more appropriate to object oriented programming: polymorphic types are types whose functions can be applied to many different types.

The same messages can be sent to a number of different classes and the messages can (mostly) have any type of argument. The messages need not necessarily provide concurrent communication but may behave largely like procedure calls. Unlike procedure calls, messages sending allows polymorphism without the requiring constant checking of parameter types [Ingalls, 1986].

The existence of class hierarchies must entail a certain polymorphism if related but distinct classes are to understand the same messages [*inclusion polymorphism* — Cardelli & Wegner, 1985]. The conceptual power of inheritance hierarchies derives, at least partly, from the way in which they allow automatic but controlled polymorphism for all subclasses. One largely knows the behaviour of an object if one knows the behaviour of its superclass.

Having polymorphic messages makes it easy to extend a language with new types which are on a par with existing types. Smalltalk can be said to exhibit ''true'' polymorphism because all objects are uniformly represented and *can* exhibit uniform behaviour. C++ restricts the use of message lookup, but mostly new types can be incorporated just as elegantly as with Smalltalk.

Message passing affects how we view computation. In object oriented languages computation is a process whereby the abilities of self contained objects are invoked by the exchange of messages. In other languages computation is variously regarded as a process of deduction, or of transitions between states, or of the transformation of input data to output data.

2.4 Concurrency.

Real events happen at the same time. Interacting with, or simulating, such an environment requires concurrent execution of the objects representing elements of this environment, at least in principle. Luckily ''time'' when applied to computer animation often means discrete steps synchronized every twentieth of a simulated second. Thus one could simply service all objects sequentially in each time slot. However explicit support for concurrency can be useful.

In real-time applications processing speed becomes critical. To avoid the 'von Neumann bottle-neck' we have to use parallel hardware. Concurrency thus addresses both these issues:

A. Natural decomposition of complex problems.

B. Exploitation of parallel hardware for performance.

A basic abstraction for concurrent object oriented systems is provided by the actor/message passing formalism [Agha & Hewitt, 1987]. Various concurrent object oriented languages have also been reported [e.g. Yonezawa et al., 1986]. A recent ACM SIGPLAN workshop was devoted to the topic of object based concurrent programming. Apart from the many more specific papers the proceedings contain a useful survey and (somewhat partisan) view of the basic issues in concurrent programming by Agha [1989].

2.5 Summary of Object Oriented Features.

To summarize the constellation of features which comprise object oriented programming, we have [Wegner, 1987]:

- Objects.
- Message Passing.
- Classes or Prototypes.
- Inheritance or Delegation.
- Information Hiding.
- Strong Typing or Run-Time Type Checking.
- Concurrency.

The questions to be asked about these features for graphics are:

- Are they desirable?
- How important are they for graphics?
- Are they sufficient for building graphical systems or are extensions needed?
- Are the chosen features consistent?
- Are they orthogonal?

Languages can be classified according to these features. For example one might call all languages with objects "Object Based": e.g. Ada & Modula. Languages with objects and classes can be called "Class Based": e.g. CLU. "Object Oriented" under this scheme would be reserved for languages which also supported inheritance: e.g. Smalltalk. Some object oriented languages do not support information hiding: e.g. Simula & Flavors.

Actor languages do not have a particular mechanism for inheritance. They do support delegation. Thus one can have actors for an object, the prototype of that object, and even the class of that object.

§3. Object Oriented Graphics: Structuring Complex Programs and Data.

Complexity in computer graphics arises from many sources: the representation of geometric detail, motion, interactions between actors, the user interface, and the complexity inherent in large systems. We might want to deal with complex natural scenes. Dynamic graphics vastly compounds the problem since it adds changes and interrelations over time.

Object oriented graphics standards (§3.1) are a way of dealing with one aspect of complexity.

Computer animation, the combination of simulation and 3-D graphics, gives a good example of object oriented programming in graphics. We first abstract some basic requirements of an animation system (§3.2). We can then give a review of work in this area (§3.3). ThingLab (§3.4) was a significant early application developed in Smalltalk, it provided a basic implementation for a number of important features: constraints, prototypes, part-hierarchies and multiple views.

In the end (§3.5) it will become apparent that the basic theoretical requirements for object oriented animation correspond rather closely to those of interactive computer graphics and CAD. The only real difference is that a more dynamic classification system is needed.

3.1 Object Oriented Graphics Standards.

There has been some work on incorporating conventional graphics standards into object oriented programming, for example, GKS [Lubinski & Hutzel, 1984] and PHIGS [Wißkirchen, 1986].

The Doré toolkit, introduced in §2, is one of the prime examples of a new direction in graphics standards. Future standards need to provide extensible and flexible interfaces to all the specialized facilities modern computing platforms provide. These include specialized hardware and many different parallel architectures. Data abstraction to hide implementation details and local subclassing to extend standards in a consistent way, go a long way to meet these requirements.

3.1 Animation Basics.

Animation systems often represent natural scenes and are found in aircraft simulators, in video games and the production of special cinematic effects. Robotics also has a lot in common with animation. There are two aspects to animation: producing computer representations which allow movement and controlling that movement. We shall be concerned with providing an underlying mechanism which will allow movement to be handled elegantly.

Computer animators note the need for abstraction as a way of dealing with their rather difficult problem. Zeltzer [1985] describes different kinds of abstraction useful for character animation (the word "motion" can be added before "abstraction" in every case).

- Structural abstraction describes the kinematic properties of the figure, i.e. the hierarchy of jointed limbs and their possible motions.
- Procedural abstraction describes the movements in terms of the desired results rather than the particular kinematic structure.
- Functional abstraction describes movements in terms of parameterized skills or basic movements. For example: walking or grasping, which can be fast or slow.

At the lowest level a figure can be modelled as a tree structure of joints and parts. The parts are embedded in a generalization lattice of attributes, this lattice being supplied by some sort of multiple class inheritance hierarchy.

Each part of the hierarchical representation of an object has its own changing local coordinate system. In animation and rendering these coordinate systems have to be related to one another and to the world coordinate system. Animated figures and robots are governed by constraints on their allowed movements.

The complex modelled environment of an animated object has to be structured in some way which allows rapid testing for the proximity of objects. The description of objects in terms of a hierarchy of parts which also reflect levels of detail should go a long way towards meeting this need.

A general abstraction is needed which incorporates the notion of "an object made up of active parts which are related via constraints". This has profound implications for the way such objects have to be modelled (see §3.4).

3.2 Actors and Animation.

Actors are met in object oriented animation systems. This kind of actor is different from, but related to, the strict definition of actors in Hewitt's formalism (see §2 & §2.4). Early examples are DIRECTOR [Kahn, 1976] and ASAS [Reynolds, 1982]. A succession of animation systems based to a greater or lesser extent on the actor formalism have been developed by Magnenat-Thalmann & Thalmann [1985].

Message passing actors have proved to be very appropriate for modelling 3-D animation. Object oriented animation was influenced by Logo [Kay, 1977]. Logo is not object oriented. In Kay's terms Logo is a data-procedure language, whereas in Smalltalk the data and procedures are replaced by the single idea of "activities" which belong to families. New families are created by combining and enriching properties which are inherited as traits. This message-activity system is inherently parallel.

A similar strong influence of Logo is apparent in Kahn's [Kahn, 1976; Magnenat-Thalmann & Thalmann, 1985] Director language. Like Kay he emphasizes that a computer language should reflect both the structure of its applications and the intuitions of its users. For animation this means that each entity should be a "little person" who communicates with others by means of messages. An animation as a whole is then produced by a number of parallel cooperating processes.

Kahn's animation system is a practical approximation to this ideal. There is a Universe which holds the actors. Each actor remembers its own actions and the Universe (the scheduler) merely sends a 'tick' message to them. At each tick an actor performs its actions and interactions for that time increment. The display messages are sent to a screen actor and these messages can also be remembered to make a movie.

Two problems addressed in §4, part-whole relations and different ways of regarding an object, are mentioned as future research goals by the previous authors. Kahn mentions the need for better primitives for dealing with composite objects and for constructing objects out of parts. Kay advocates the development of an "observer language" by which he means a language which allows objects to be regarded from different viewpoints with respect to what they are said to be composed of.

Both Kay and Kahn produced rather simple two-dimensional images. In ASAS [Reynolds, 1982] we get much more realistic three-dimensional graphics.

Reynolds [1987] described a notable new system, and many other object oriented approaches to animation have appeared [e.g. Uchiki et al., 1983; Blake 1987; Breen et al., 1987; Fiume et al., 1987]. An object oriented language for video game design has been developed [Larrabee & Mitchell, 1984], while a physical simulation system based on actors has been described by Haumann & Parent [1988].

3.3 ThingLab

ThingLab [Borning, 1979; 1981; 1986a; Borning et al., 1987] is a system for simulating physical objects (e.g., geometric shapes, bridges, electrical circuits, documents, calculators). In ThingLab, objects consist of parts. Multiple class inheritance hierarchies, and *part-whole hierarchies* are used to describe the objects and their interrelations. Parts are referred to symbolically by means of *paths* that name the nodes to be visited in proceeding down the part hierarchy.

In ThingLab the superclasses of an object are a part of the object: an object contains an instance of its superclass (in the type theoretic sense) as sub-part (in the sense of part versus whole objects). The class which describes such a superclass part is a subclass of the normal part description class. Apart from this notion of multiple superclasses ThingLab also employs prototypes to provide initialized instances of objects.

The major contribution of ThingLab is a system for representing and satisfying constraints which exist between the parts. However, when it comes to providing a tool for modelling complex objects confusion can arise. In particular we should draw a clearer distinction between class hierarchies and part hierarchies. When classes are available it is also possible to dispense with prototypes, not that classes are necessarily preferable to prototypes.

3.4 Interactive Object Oriented Graphics.

Object oriented approaches are fundamentally procedural. The messages invoke procedures with side-effects. These side-effects are contained to a greater or lesser extent by the data encapsulation. However it is relatively easy to ensure that certain relationships are maintained between objects. Constraint based approaches are very useful in making user interfaces [Borning & Duisberg, 1986].

In object oriented implementations, particularly those based on prototypes, it is easy to make incremental changes to the behaviour of objects. This is useful in CAD applications. However since the operations on the object, or its type, can be altered very freely, it can be difficult to reason about relations between objects and to transform relations [Beynon, 1988].

Object oriented methods have been used for user interfaces since the earliest days [Kay, 1977]. Characteristic features of such interfaces were: *what you see is what you get*, *visibility* (no obscure modes or key combinations) and a *physical metaphor* [e.g., Lipkie et al., 1982]. This combination of features enables the display to reflect the user's model of the application directly. Ideally this gives the user the feeling that he is manipulating aspects of the application without the mediation of a computer [Anson, 1982]. Object oriented approaches are well suited to building direct manipulation interfaces [Shneiderman, 1983; Hartson, 1989].

Smalltalk provides the Model-View-Controller method of building user interfaces [Krasner & Schmidt, 1989]. The model (or application) is created independently of the user interface. The model has to provide methods for interrogating its state and only has to broadcast a general message to all dependent objects when this state changes. The view and controller present the model to the user and translate user actions into operations on the model. Depending on the user, there may be multiple views on the same model.

CAD systems have also used the object oriented paradigm [e.g., Girczyc & Ly, 1987]. Myers [1989] gives a compact survey of user interface tools which contrasts object oriented approaches with the alternatives.

In user interface design and CAD there is also a strong requirement for a system which allows one to express the constraints which govern the behaviour of parts. The Animus system [Duisberg, 1986] extended the constraint methods of ThingLab to allow constraints involving time. This removed the essentially static nature of ThingLab constraints and allowed the description of evolving structures and animations.

A system to build graphical interfaces based on icons and windows is described by Barth [1987] and a Unix* specific one by Budd [1989]. The use of a part hierarchy in user interfaces is discussed by Wißkirchen [1988].

§4. Limitations and Extensions.

From §3 it can be seen that there are two salient features of an animated figure which we must capture: (a) it is composed of parts which depend on each other, and (b), these parts can move subject to various constraints. These same requirements arose again in the discussion of interactive graphics [see also Tomiyama, 1989]. Here we first consider how object oriented languages ought to be extended if we want to model the part-hierarchy of real physical objects and still retain all the conceptual and programming advantages of object oriented programming (§4.1). The provision of constraints between objects is discussed in the next section (§4.2). Section §4.3 mentions possible extensions to allow multiple views of the same object. Section §4.4 introduces encapsulators which are useful for dealing with incomplete objects. Finally (§4.5)

---

* Unix is a trademark of AT&T.

we briefly review logic and functional programming as alternative abstractions to object oriented graphics.

In §2.2.1 we first saw how necessary prototypes and delegation can be if numerous changes in the type of an object has to be made. The lack of such features in a system intended for interactive modelling may also be regarded as a limitation.

4.1 Representing Physical Objects: The part hierarchy.

Things are often described in terms of parts and wholes; the way the division into parts is made depends on the purpose of the analysis. A part is a part by virtue of its being included in a larger whole. A part can become a whole in itself, which can then be split into further parts. In this way we build up a hierarchy of parts and wholes, which we have called the *part hierarchy*.

We distinguish between a mere *collection*, or additive whole, or heap, (e.g., a bag of marbles, a pile of electronic components) and a more *structured whole* (e.g., an animal, a wired-up electronic circuit). To the former we apply set theory, to the latter a part hierarchy. (See Smith [1982] for a formal description of part-whole relations).

Ideally information should be stored in the part hierarchy at its corresponding logical level. Information about the whole is not to be stored in the parts, while information about the parts, independent of the whole, remains with the parts. Ideally the whole knows the parts but the parts do not know of the whole.

Part-whole analysis is crucial in engineering and technology (assemblies and subassemblies). Parts are also met in those branches of computation where physical objects are represented, for example, *model-based computer vision* and *computer graphics*. For example, in Foley & van Dam [1982] there is reference to the object hierarchy. The graphics standard PHIGS (Programmer's Hierarchical Interactive Graphics System) [1986] organizes objects in a structure hierarchy. Both *structure hierarchy* and *object hierarchy*,, are synonyms for part hierarchy.

PHIGS also has the concept of *inheritance* on a part hierarchy where attributes of the whole are inherited by the parts. E.g., the legs of the table could inherit the colour of the whole. The requirement is not quite as general as it at first appears. This ''inheritance'' is only used when the structure is traversed. The object based graphics toolkit Doré (see §2) uses groups in a similar manner.

Frame-based representation [Fikes & Kehler, 1985] has similarities to the object oriented approach. Frames describe parts and attributes by means of *slots*. Composite objects are provided in Loops [Stefik & Bobrow, 1985], but Smalltalk and most other object oriented languages fail to provide the facility to describe objects in terms of their parts. Or more accurately, when we want to model objects consisting of parts we are confronted with a dilemma: either sacrifice the data encapsulation properties of the language or utterly flatten the part-whole hierarchy [Blake & Cook, 1987].

4.2 Constraints.

Constraints specify relations between objects which must be maintained. When a change occurs in the system it has to adjust all affected objects such that all the constraints remain satisfied. There are therefore two aspects to constraints:

Descriptions.

> Constraints specify the relations which obtain between objects, particularly between parts and sub-parts. This is the declarative aspect of a constraint, it is a rule.

Methods.

> In order to satisfy the constraints methods of constraint satisfaction have to be given. This is the procedural aspect of constraints.

Once the methods for satisfying constraints have been defined constraint based programming acquires a declarative feel, that is, the programs become largely static specifications of the relations which have to obtain between objects [but see, Duisberg, 1986]. It must be borne in mind that implementing effective constraint satisfaction techniques is a difficult task.

The basis for object oriented constraint satisfaction was laid by ThingLab (§3.4). It is argued that the locality of reference afforded by data encapsulation is vital in reducing the scope of an alteration to the system. This reduces the number of objects which have to be involved in satisfying a constraint. Furthermore the generality of methods allowed in object oriented programming allows many constraint satisfaction techniques to be employed (including, for example, those found in logic programming). There are systems which combine logic and object oriented programming (see §4.5.1).

Constraints based programming has not been seen as an integral part of object oriented programming. However it must be viewed as an integral part of interactive computer graphics and of computer animation. Constraint satisfaction techniques often have a pleasing generality, however domain specific knowledge will also be required. Thus the technique cannot be provided independently of the intended application.

4.3 Multiple Views.

In object oriented systems with data encapsulation a whole can pretend to have parts which are not actually stored as such [Borning 1979, ''virtual parts'']. Since access to the parts is only via messages the responses to these messages can be generated on the fly, rather than stored. A rectangle can be stored in terms of a top-left and bottom-right corner, but it can equally well pretend to have a centre which can be read or modified.

This allows one to have multiple views of the same object. A complex number can be accessed as a real and imaginary number or as a radius and angle, without regard to the 'true' underlying representation.

It is apparent that doing this would have been impossible if we had sacrificed data encapsulation. These different names (like ''phantom'', or ''imaginary'', or ''virtual'') for such parts refer purely to an implementation issue: to the outside of the object the distinction does not exist.

4.4 Encapsulators.

An encapsulator provides a transparent, often temporary, interface to an object. The object is surrounded by an encapsulator so that all messages to the object and all replies from the object are intercepted by the encapsulator. This allows pre- and post-processing of messages while the encapsulated object remains externally identical to the enclosed object [Pascoe, 1986]. Encapsulators are another step in the direction of control over side-effects which we have seen developing out of the original idea of data abstraction, progressing through information hiding and encapsulation (§2.1).

Encapsulators can be used to implement *Futures* to manage concurrent synchronization and *Delays* to provide lazy evaluation [Halstead, 1985]*. A Future is an encapsulator which encloses the result of a child process which has not yet completed execution, messages to this as yet non-existent object are held until execution is completed. In the meantime a Future can be passed around as if *it is* the result. Lazy evaluation is used where objects are created only if there is a demand for them, this can provide (conceptually) infinite data structures.

---

* Encapsulators, Futures and Delays are easily implemented in Smalltalk. The essential idea is that an encapsulator understands no messages and thus all messages can be intercepted by the ''not understood'' method.

In graphics and particularly CAD incomplete objects are often created. Encapsulators seem to be an ideal way of managing such objects.

4.5 Alternatives to Object Oriented Graphics: Declarative Languages.

The complexity of graphical computation can also be ameliorated by declarative methods. The principal benefit of declarative languages is that they shift the burden of deciding how a thing has to be done from the programmer to the architecture. Both *functional* and *logic* languages are declarative. The need for declarative programming has already been explored in §4.2 on constraints.

Pure declarative languages employ no side-effects whatsoever and assignment of values to variables is impossible. In object oriented terms one could say that data is always completely encapsulated. In fact, declarative languages lack a notion of 'state of computation'. This is in direct opposition to the idea of self contained objects which persist over time while their internal configuration changes. This seems to lead to conceptual difficulties when we consider interactive graphics and computer animation.

4.5.1 Logic Programming.

The main tenet of logic programming [Kowalski, 1979] is that an algorithm consists of logic and control. The logic, that is, properties of the problem and its solution, are supplied by the programmer. The machine is responsible for the control, that is, how the solution is computed. This ideal is not yet achieved. Computers generally implement a subset of first order predicate calculus (e.g. Prolog).

Logic programming can be used for graphics and CAD systems [Swinson, 1983]. There are also CAD systems which combine logic programming with object oriented programming [Arbab, 1989]. This is part of a broader effort to unify object oriented and logic programming [Goguen & Mesguer, 1987; Newton & Watkins, 1988]. This can produce an impure hybrid language which provides the benefits of both. It is an open question whether logic is the best way to express the relations which exist between graphical objects.

4.5.2 Functional Programming.

Functional programming also provides an abstraction for computer graphics [Arya, 1986; Burton & Kollias, 1989; Henderson, 1982; Salmon & Slater 1987]. A few points of contrast and similarity with object oriented graphics will be given.

Functional programming derives its power from giving functions first class status [see e.g. Hughes, 1989]. Functions can be combined and manipulated just like any other object. Data structures are defined by means of constructor functions which make abstract data objects. Access is only via the operations defined on the data objects. The usefulness of data abstraction has already been mentioned. Polymorphism is also emphasized and functional languages are well suited to programming concurrency [Peyton Jones, 1989].

Pure functional programs are static objects. The meaning of an expression does not change as computation proceeds. Real objects persist while their configurations and attributes change over time. Animation, as the mimicking of three-dimensional physical objects, depends on a notion of *state*. We have seen that this meshes rather well with the concept of actors and objects in object oriented programming. In functional graphics the emphasis is shifted to dealing with a sequence of *different* objects related by a sequence of transformations. This model of computation is found in key frame animation, which is mainly used for two-dimensional pictures. Slater also discusses the fact that difficulties arise when using functional languages for programming interaction and when using attributes [Salmon & Slater, 1987, pp. 290-291].

This does not prevent functional programming from being used in practice in time dependent situations. Generally it is possible to "abstract away" the notion of time, and replace it with some idea of sequences over infinite lists [Arya, 1986]. Lazy evaluation (cf §4.4) is an elegant way of coping with such infinite structures [Jones & Sinclair, 1989]. To discover the relation between the figures in the list one refers to the functions which constructed them. *But* the conceptual advantages of functional programs remain limited precisely because such programs describe a dynamically changing world as a (conceptually) frozen system of infinite sequences.

Functional programming is evolving, and the final conclusions regarding functional *versus* object oriented approaches to computer animation cannot yet be drawn. The extent to which a (possibly impure [e.g., Halstead, 1985]) functional approach can be elaborated for three-dimensional animation needs further investigation. On the other hand, object oriented animation already seems well suited to modelling changing objects executing concurrently.

§5. Conclusion.

We have tried to review some of the issues in object oriented graphics. Other general discussions of (aspects of) object oriented graphics include: Barth [1986 — user-interfaces], Magnenat-Thalmann & Thalmann [1985 — computer animation], Myers [1989 — a general discussion of user-interface tools], Tomiyama [1989 — CAD] and Wißkirchen & Rome [1988 — tutorial].

It was apparent that object oriented programming has many aspects whose importance varies with the intended application. Thus we saw that with CAD, where objects change incrementally, delegation rather than inheritance might be preferable.

In §4 we mentioned where further research is needed in order to adapt object oriented programming for interactive graphics and animation. The need for a part hierarchy as an integral part of any language for representing physical objects has been identified. It is also apparent that the complexity of specifying interactions and computer animated sequences means that a constraint based declarative approach to programming is also needed. This will allow a user to specify *what* has to happen on the assumption that the basic details of *how* something has to be done has been solved previously. The brief mention of alternate views on objects does not indicate the unimportance of the topic for research, on the contrary.

It does seem that on the whole the object oriented approach offers more promise as a basic abstraction for computer graphics than its alternatives (functional or logic programming).

§5.1 A Note for Some Practitioners.

In a flexible approach to introducing object oriented methods to graphics one should beware of being too easy going. It is all too easy, but lazy, to regard the discipline of information hiding and type hierarchies (for example) as restrictive, instead of regarding them as elegant ways of building complex systems. When problems arise they should be surmounted by means of extensions to the object oriented method which fit in with the general thrust of the approach. Examples of such extensions are the part hierarchies and encapsulators.

Acknowledgements.

I would like to thank my colleagues Kees Blom, Ravic Pieters Kwiers and Jan Rogier for their helpful comments on the first draft of this paper.

Bibliography.

Adobe Systems (1985) *PostScript Language Reference Manual.* Addison-Wesley, Reading, MA.

Agha, G. (1986) *SIGPLAN Notices* **21**, 10 58-67
"An overview of actor languages."

Agha, G. (1989) *SIGPLAN Notices* **24**, 4 60-65
"Foundational issues in concurrent programming."
Proc. ACM SIGPLAN Workshop on Object-Based Concurrent Programming. San Diego, Sept. 26-27, 1988.

Agha, G. & Hewitt, C. (1987) **in:** Shriver, B. & Wegner, P. **(ed)** *Research Directions in Object-Oriented Programming.* The MIT Press, Cambridge, Massachusetts. 49-74.
"Actors: a conceptual foundation for concurrent object-oriented programming."

Anson, E. (1982) *SIGGRAPH'82: Computer Graphics* **16**, 3 107-114
"The device model of interaction."

Arbab, F. (1989) **in:** Akman, V., ten Hagen, P.J.W. & Veerkamp, P.J. **(ed)** *Intelligent CAD Systems II: Implementation Issues.* EurographicSeminars Series. Springer-Verlag, Berlin. 32-57.
"Examples of Geometric Reasoning in OAR."

Ardent Computer Corporation (1988) *Doré Programmers' Guide.*

Arya, K. (1986) *Computer Graphics Forum* **5**, 4 297-311
"A functional approach to animation."

Barth, P.S. (1986) *ACM Trans.Graphics* **5**, 2 142-172
"An object-oriented approach to graphical interfaces."

Beynon, M. (1988) **in:** Earnshaw, R.A. **(ed)** *Proc.NATO ASI: Theoretical Foundations of Computer Graphics and CAD.* 1083-1097.
"Definitive principles for interactive graphics."

Blake, E.H. (1987) *Eurographics'87.* Elsevier, Amsterdam. 295-307
"A metric for computing adaptive detail in animated scenes using object-oriented programming."

Blake, E.H. & Cook, S. (1987) **in:** Bézivin, J., Hullot, J.-M. Cointe, P. & Lieberman, H. **(ed)** *Lecture Notes in Computer Science, no.276. ECOOP'87: European Conference on Object-Oriented Programming.* Springer-Verlag, Berlin. 41-50
"On including part hierarchies in object-oriented languages, with an implementation in Smalltalk."
Paris, France, June 15-17, 1987

Borning, A.H. (1979) *Xerox Palo Alto Research Center report SSL-79-3*
"ThingLab: A constraint-oriented simulation laboratory."
a revised version of: Stanford University PhD. thesis, Stanford Computer Science Department Report STAN-CS-79-746

Borning, A.H. (1981) *ACM Trans.Programming Languages and Systems* **3**, 4 353-387
"The programming language aspects of ThingLab, a constraint-oriented simulation laboratory."

Borning, A.H. (1986a) *IEEE/ACM Fall Joint Computer Conf.* 36-40
"Classes versus prototypes in object-oriented languages."
Dallas, Texas, Nov 1986.

Borning, A.H. (1986b) *ACM CHI'86 Proceedings*. 137-143
"Defining constraints graphically."

Borning, A.H. & Duisberg, R.A. (1986) *ACM Trans.Graphics* **5**, 4 345-374
"Constraint-based tools for building user interfaces."

Borning, A.H., Duisberg, R.A., Freeman-Benson, B., Kramer, A. & Woolf, M. (1987) *OOPSLA'87: SIGPLAN Notices* **22**, 12 48-60
"Constraint hierarchies."

Borning, A.H. & Ingalls, D.H.H. (1982) *Proc.Nat.Conf.Artificial Intelligence* 234-237
"Multiple inheritance in Smalltalk-80."
Pittsburgh, PA.

Brachman, R.J. (1983) *Computer* **16**, 10 30-36
"What IS-A is and isn't: an analysis of taxonomic links in semantic networks."

Breen, D.E., Getto, P.H., Apodaca, A.A., Schmidt, D.G. & Sarachan, B.D. (1987) *Eurographics'87*. Elsevier, Amsterdam. 275-282
"The Clockworks: An object-oriented computer animation system."

Budd, T.A. (1989) *Software Practice & Experience* **19**, 1 35-51
"The design of an object-oriented command interpreter."

Burton, F.W. & Kollias, Y.G. (1989) *IEEE Software* **6**, 1 90-97
"Functional programming with quadtrees."

Cardelli, L. (1984) **in:** Kahn, MacQueen & Plotkin **(ed)** *Lecture Notes in Computer Science, no 173. Semantics of Data Types*. Springer Verlag. 51-67
"A semantics of multiple inheritance."

Cardelli, L. & Wegner, P. (1985) *Computing Surveys* **17**, 4 471-522
"On understanding types, data abstraction, and polymorphism."

Cox, B.J. (1984) *IEEE Software* **1**, 1 50-61
"Message/Object programming: an evolutionary change in programming technology."

Cox, B.J. (1986) *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley. , Reading, Massachusetts.

Dahl, O.-J. & Hoare, C.A.R. (1973) **in:** Dahl, O.-J., Dijkstra, E.W. & Hoare, C.A.R. **(ed)** *Structured Programming*. Academic Press, London. 175-220
"Hierarchical program structures."

Danforth, S. & Tomlinson, C. (1988) *ACM Comp.Surveys* **20**, 1 29-72.
"Type theories and object-oriented programming."

Densmore, O.M. & Rosenthal, D.S.H. (1987) *Computer Graphics Forum* **6**, 3 171-180
"A user-interface toolkit in object-oriented PostScript."

Duisberg, R.A. (1986) *ACM CHI'86 Proceedings* 131-136
"Animated graphical interfaces using temporal constraints."

Earnshaw, R.A., Bresenham, J.E., Dobkin, D.P., Forrest, A.R. & Guibas, L.J. (1987) *SIGGRAPH'87: Computer Graphics* **21**, 4 345
"Panel: 'Pretty pictures aren't so pretty anymore: A call for better theoretical foundations.'."

Fikes, R. & Kehler, T. (1985) *Comm.ACM* **28** 904-920
"The role of frame-based representation in reasoning."

Fiume, E., Tsichritzis, D. & Dami, L. (1987) *Eurographics'87*. Elsevier, Amsterdam. 283-294
"A temporal scripting language for object-oriented animation."

Foley, J.D. & van Dam, A. (1982) *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Massachusetts.

Girczyc, E.F. & Tai Ly (1987) *24th ACM/IEEE Design Automation Conference*. 757-763.
"STEM: an IC design environment based on the Smalltalk Model-View- Controller construct."

Goguen, J.A. & Mesguer, J. (1987) **in:** Shriver, B. & Wegner, P. **(ed)** *Research Directions in Object-Oriented Programming*. The MIT Press, Cambridge, Massachusetts. 417-477
"Unifying functional, object-oriented and relational programming with logical semantics."

Goldberg, A. & Robson, D. (1983) *Smalltalk-80: the language and its implementation*. Addison-Wesley, Reading, Massachusetts.

Greenberg, D.P. (1988) *Comm.ACM* **31** 123-129,151
"Coons award lecture."

Guttag, J. (1977) *Comm.ACM* **20**, 6 396-202
"Abstract data types and the development of data structures."

Halstead, R.H. (1985) *ACM Trans.Programming Languages and Systems* **7**, 4 501-538
"Multilisp: A language for concurrent symbolic computation."

Hartson, R. (1989) *IEEE Software* **6**, 1 62-70
"User-interface management control and communication."

Haumann, D.R. & Parent, R.E. (1988) *The Visual Computer* **4** 332-347
"The behavioral test-bed: obtaining complex behavior from simple rules."

Henderson, P. (1982) *Symposium on Lisp & Functional Programming*. ACM. 179-187
"Functional geometry."

Hughes, J. (1989) *The Computer Journal* **32**, 2 98-107
"Why functional programming matters."

Ingalls, D.H.H. (1986) *OOPSLA'86: SIGPLAN Notices* **21**, 11 347-349
"A simple technique for handling multiple polymorphism."

Jones, S.B. & Sinclair, A.F. (1989) *The Computer Journal* **32**, 2 162-174
"Functional programming and operating systems."

Kahn, K.M. (1976) *User-oriented design of interactive graphics systems (ACM/SIGGRAPH workshop)*. 37-43
"An actor-based computer animation language."

Kay, A.C. (1977) *Scientific American* **237**, September 230-244
"Microelectronics and the personal computer."

Kowalski, R. (1979) *Comm.ACM* **22** 424-436.
"Algorithm = logic + control."

Krasner, G.E. & Pope, S.T. (1988) *J.Object-Oriented Prog.* **1**, 3 26-48
"A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80."

LaLonde, W.R., Thomas, D.A. & Pugh, J.R. (1986) *OOPSLA'86: SIGPLAN Notices* **21**, 11 322-330
"An exemplar based Smalltalk."

Larrabee, T. & Mitchell, C.L. (1984) *IEEE Software* **1**, 4 28-36
"Gambit: A prototyping approach to video game design."

Lieberman, H. (1986a) **in:** Bezivin, J. & Cointe P. **(ed)** *3eme Journées d'Etudes Langages Orientés Objet*. 79-89
"Delegation and inheritance: Two mechanisms for sharing knowledge in object-oriented systems."

Lieberman, H. (1986b) *OOPSLA'86: SIGPLAN Notices* **21**, 11 214-223
"Using prototypical objects to implement shared behavior in object-oriented systems."

Lieberman, H., Stein, L.A. & Ungar, D. (1987) *OOPSLA'87 Addendum to the Proceedings*. 43-44.
"Of types and prototypes: the treaty of Orlando."

Lipkie, D.E., Evans, S.R., Newlin, J.K. & Weissman, R.L. (1982) *SIGGRAPH'82: Computer Graphics* **16**, 3 115-124
"Star graphics: an object-oriented implementation."

Liskov, B. & Zilles, S. (1974) *SIGPLAN Notices* **9**, 4 50-59
"Programming with abstract data types."

Lubinski, T. & Hutzel, I. (1984) *Computer Graphics World* , July 69-75
"An object-oriented graphical kernel system."

Madsen, O.L. & Møller-Pedersen, B. (1988) **in:** Gjessing, S. & Nygaard, K. **(ed)** *Lecture Notes in Computer Science, no.322. ECOOP'88*. Springer Verlag, Berlin. 1-20.
"What object-oriented programming may be — and what it does not have to be."

Magnenat-Thalmann, N. & Thalmann, D. (1985) *Computer Animation: Theory and Practice*. Springer-Verlag, Tokyo.

Meyer, B. (1988) *Object-oriented software construction*. Prentice-Hall, London.

Micallef, J. (1988) *J.Object-Oriented Prog*. **1**, 1 12-36
"Encapsulation, reusability and extensibility in object-oriented programming languages."

Myers, B.A. (1989) *IEEE Software* **6**, 1 15-23
"User-interface tools: Introduction and survey."

NeWS (1987) *Part No: 800-1498-05*. Sun Microsystems, Inc, Mountain View, CA.
"NeWS Technical Overview."

Newton, M. & Watkins, J. (1988) *J.Object-Oriented Prog*. **1**, 4 7-10
"The combination of logic and objects for knowledge representation."

Nygaard, K. (1986) *SIGPLAN Notices* **21**, 10 128-132
"Basic concepts in object-oriented programming."

Pascoe, G.A. (1986) *OOPSLA'86: SIGPLAN Notices* **21**, 11 341-346
"Encapsulators: A new software paradigm in Smalltalk-80."

Peyton Jones, S.L. (1989) *The Computer Journal* **32**, 2 175-186
"Parallel implementations of functional programming languages."

PHIGS (1986) *ISO PHIGS revised working draft*
"Programmer's Hierarchical Interactive Graphics System."

Rensch, T. (1982) *SIGPLAN Notices* **17** 51-57
"Object oriented programming."

Reynolds, C.W. (1982) *SIGGRAPH'82: Computer Graphics* **16**, 3 289-296
"Computer animation with scripts and actors."

Reynolds, C.W. (1987) *SIGGRAPH'87: Computer Graphics* **21**, 4 25-34
"Flocks, herds and schools: A distributed behavioral model."

Salmon, R. & Slater, M. (1987) *Computer Graphics: Systems & Concepts*. Addison-Wesley, Wokingham, England.

Shneiderman, B. (1983) *Computer* **16**, 8 57-69
"Direct manipulation: a step beyond programming languages."

Smith, B. (1982) *Parts and Moments. Studies in Logic and Formal Ontology*. Philosophia Verlag, München.

Snyder, A. (1986) *OOPSLA'86: SIGPLAN Notices* **21**, 11 38-45
"Encapsulation and inheritance in object-oriented programming languages."

Stefik, M. & Bobrow, D.G. (1985) *The AI Magazine* **VI**, 4 40-62
"Object-oriented programming: Themes and variations."

Stein, L.A. (1987) *OOPSLA'87: SIGPLAN Notices* **22**, 12 138-146
"Delegation is inheritance."

Stroustrup, B. (1986) *SIGPLAN Notices* **21**, 10 7-18
"An Overview of C++."

Stroustrup, B. (1988) *IEEE Software* **5**, 3 10-20
"What is object-oriented programming."

Swinson, P.S.G. (1983) *Computer-Aided Design* **15** 335-343
"Prolog: a prelude to a new generation of CAAD."

Tomiyama, T. (1989) **in:** Akman, V., ten Hagen, P.J.W. & Veerkamp, P.J. **(ed)** *Intelligent CAD Systems II: Implementation Issues*. EurographicSeminars Series. Springer-Verlag, Berlin. 3-16.
"Object oriented programming paradigm for intelligent CAD systems."

Uchiki, T., Ohashi, T. & Tokoro, M. (1983) *Computers & Graphics* **7**, 3&4 285-293
"Collision detection in motion simulation."

Wegner, P. (1987) *OOPSLA'87: SIGPLAN Notices* **22**, 12 168-182
"Dimensions of object-based language design."

Williams, T. (1988) *Computer Design* , February 30-31
"Graphics library aids high-level interactive visualization."

Wißkirchen, P. (1986) *Computers & Graphics* **10**, 2 183-187
"Towards object-oriented graphics standards."

Wißkirchen, P. (1988) *German National Research Center for Computer Science Internal Report GMD-F3.MMK*.
"Editing Groups and Parts in a Multi-Level Graphics System."

Wißkirchen, P. & Rome, E.L. (1988) *SIGGRAPH'88 Course Notes*.
"Object-Oriented Graphics."

Yonezawa, A., Briot, J.-P. & Shibayama, E. (1986) *OOPSLA'86: SIGPLAN Notices* **21**, 11 258-268
"Object-oriented concurrent programming in ABCL/1."

Zeltzer, D. (1985) *The Visual Computer* **1** 249-259
"Towards an integrated view of 3-D computer animation."