An Architecture of a Universal DBMS for Graphics Applications

Andrey Collison Hanspeter Bieri

Research Group on Computational Geometry and Graphics
Institute of Computer Science and Applied Mathematics
University of Berne, Neubrückstrasse 10, 3012 Berne, Switzerland.
E-mail: `collison@iam.unibe.ch, bieri@iam.unibe.ch`

Abstract

This paper presents a new architecture of a database management system intended to be used together with existing graphics software. In the first place, a system with this architecture shall be capable of coping with the large variety of data representations typically found in the graphics domain. Data is stored in its original form, without prior conversions, thus conserving the maximum information content. A special typing model separating semantics from implementation ensures type safeness and at the same time provides the flexibility and extensibility needed to cope with multiple data representations. Secondly, software components enable seamless integration of existing graphics operations offered by various software packages. The components together with a composition mechanism act as the data manipulation language of the architecture. The architecture to be presented has been implemented in a prototype system called GSCOPE.

1. Introduction

The broad field of computer graphics includes in particular the application areas image synthesis, image processing, image analysis, geometric modelling and computational geometry [11]. *Graphics systems* are mainly aimed at the graphics application developer and provide predefined data structures and algorithms for one or more subareas of computer graphics. The last years have seen a dramatic development of graphics systems, from simple libraries to component-oriented frameworks making use of the latest advances in object-oriented programming techniques e.g. *reusable classes* [10], *design patterns* [7], and *software components* [18].

Reuse of functionality has always been a major concern in the development of graphics systems. Therefore modern graphics systems provide good mechanisms to reuse their functionality. Nowadays *reuse of data*, i.e. database facilities enabling the user to efficiently store and retrieve data, is becoming increasingly important. In contrast to reuse of functionality, reuse of data is only poorly supported by current graphics systems. In fact current systems offer in most cases only a file format to save and reload graphics data [2] [16].

Modern graphics systems are designed to be *extensible*. Their architecture enables the programmer to adapt them easily to new requirements by adding new functionality. Basically, there are two strategies to extend computer software and therefore also graphics systems. The first strategy is to implement new functionality in a system. The second strategy is to integrate functionality which has already been implemented in other software packages. In practice, the first strategy is well supported by modern graphics systems, but there is only little support for the second strategy. The problem with the second strategy is that existing functionality is designed to work on specific representations of data, in many cases different from the representations supported by the graphics system. The main hindrance to integration in the computer graphics domain is the large variety of incompatible representations and the lack of a system capable of coping with multiple representations for similar data.

Reuse and extensibility are not disjoint aspects. Extensibility enhances reuse of functionality. More precisely, implementing new operations, which may be used together with existing operations, increases reuse of functionality within a system, while integration of existing operations enables reuse of functionality among systems. Reuse of functionality is also coupled with reuse of data, as additional functionality represents additional potential for reusing data. The main hindrance to reuse data in current graphics systems is the lack of powerful database facilities and the incompatible data representations prohibiting data exchange among graphics software.

In the present paper an architecture of a universal graphics database management system will be presented. This architecture shall be capable of coping with different representation types of various graphics data and serve as an integration platform for existing graphics software, i.e. offering high degrees of extensibility and reuse of functionality, particularly among graphics software that has been integrated from different resources. Further, powerful database facilities will enhance reuse of data. As a consequence, this architecture should be capable of covering a broad range of the graphics domain.

GSCOPE (**G**raphics **S**ystem offering **Co**mponents and **Pe**rsistence) is a prototype system developed at the University of Berne according to this architecture. GSCOPE is mainly a database which provides persistent storage for multiple representations of various types of graphics data. In order to offer a wide range of operations for data manipulation, GSCOPE also serves as an integration environment, providing a uniform view of graphics functionality coming from different origins. The key techniques used to achieve these properties are modern software component technology [18] and a typing model distinguishing between semantics and representation. GSCOPE also provides a flexible and extensible testbed for research in the area of content-based retrieval methods for graphics data.

A brief overview of some graphics systems and their support for reuse and extensibility is given in Section 2. In Section 3 we present some graphics applications that demand a universal graphics DBMS. Requirements for a universal graphics DBMS are formulated. A typing model based on semantic types and implementation types is presented in Section 4. The techniques used to provide the extensibility needed for our universal approach are presented in Section 5. These techniques together with the typing model play an important part in the GSCOPE architecture being presented in Section 6. Some examples and implications are given in Section 7. Finally a number of conclusions are drawn in Section 8.

2. Reuse and Extensibility in Existing Graphics Systems

Graphics systems have evolved impressively during the last years. In an attempt to judge this progress more precisely, the following four criteria shall be applied:

1. Which areas of computer graphics are supported by a given graphics system?

2. In which way and to which extent does the graphics system allow a new application to reuse solutions that already exist?

		SRGP	PHIGS	OpenGL	OpenInventor	BOOGA	ideal system
area supported	geometric modelling		+		++	++	++
	computational geometry				+	+	++
	image synthesis	+	+	++	++	++	++
	image processing			+	+	++	++
	image analysis					+	++
	2D	+		+	+	++	++
	3D		++	++	++	++	++
philosophy of reuse	function library	++	++	++	+	+	
	framework				++	++	+
	components				+	++	++
DB philosophy	run time database		+		++	++	++
	+ file based persistence				++	++	++
	+ database persistence						++
extensibility	implement new functions				++	++	++
	integrate existing functions				+	+	++

Table 1. Comparison of some existing graphics systems to an ‘ideal’ system.

3. What are the DB facilities of the graphics system?

4. In which way and to which extent can the graphics system be extended?

Table 1 states these four criteria more precisely and compares an ‘ideal’ system – which is not yet on the market – to five existing graphics systems. SRGP (Simple Raster Graphics Package, 1988) is a popular graphics package for educational purposes [6]. It provides basic 2D raster capabilities and basic interaction handling in the form of a device independent function library. SRGP is an immediate mode library, i.e. the objects are directly displayed without being stored before in the main memory. PHIGS (Programmer’s Hierarchical Interactive Graphics System, 1988) is a 3D graphics system that allows objects to be composed hierarchically [6]. The objects are stored in a display list and may be manipulated and viewed from different angles. However there is no possibility offered to store the objects persistently. OpenGL (1992) is a streamlined 2D and 3D drawing package providing a flexible and efficient interface to specific graphics devices [14]. OpenGL focuses mainly on fast image synthesis. It also provides limited support for basic image processing operations. Open Inventor (1992) is an object-oriented 3D framework offering particularly good support for the development of interactive applications in the area of geometric modelling [16]. It is built on top of OpenGL and offers a library of components providing reusable functionality to communicate with the windowing system. Open Inventor may be extended by integrating new interactive objects. It defines a standard file format for 3D data interchange.

BOOGA (Berne's Object-Oriented Graphics Architecture, 1997) is a new graphics system developed at the University of Berne [2] [1] [17]. It has been designed to cover most aspects of 2D and 3D graphics and is based on a 3-layer architecture consisting of a component, a framework, and a library layer. Applications are built using a composition model and are divided into single processing steps, each of them represented by a component. Using the component layer of BOOGA, the developer can write – or rather compose – applications easily. The framework and the library layer provide the necessary mechanisms to create those components which are not yet available. However, most applications can be implemented by largely reusing existing components. BOOGA includes an extensible language called BSDL (BOOGA Scene Description Language) to describe 2D and 3D scenes. BSDL is used by BOOGA as a file format to persistently store objects and scenes.

BOOGA is closer to the 'ideal' system than the other four systems mentioned above, but still quite remote from it. It mainly lacks the following two possibilities – which are neither supported by the other four systems: true database facilities and comprehensive support for integration of functionality. In current graphics systems there is no support to manage large data collections and to retrieve specific data from such collections. Therefore the potential of *reusing* existing graphics data is clearly restricted. Current graphics systems support integration of functionality only as long as this functionality itself is capable of handling the particular representations of the graphics system. In order to simplify integration of existing functionality, a graphics system should be capable of coping with multiple representations of graphics data types.

3. Need for a Universal Graphics DBMS

As we have discussed in Section 2, some of today's graphics systems cover a wide scope of the graphics domain and offer good reuse of functionality within the system. Substantial improvements of reuse in the graphics domain can be obtained by applying two techniques not yet considered by current systems:

1. Improved *reuse of data* achieved by database facilities.

2. Additional *reuse of functionality*, not within one, but *among* different kinds of existing graphics software.

To achieve these improvements is the main goal of our universal DBMS architecture to be presented in this paper. The following three applications shall give more concrete motivations for the new architecture:

3.1. Application 1 : rendering scenes of different representations

An advertising agency is using computer graphics software to produce images for brochures and posters. In order to achieve the desired modelling and lighting effects, different kinds of rendering software are being used. The different renderers require different input representations for the scenes – a typical situation in the computer graphics domain. Now for the creation of a WWW-site, several images have to be recalculated in a new resolution. The resulting images have to be converted to the JPEG format.

A user will normally have to do the following: provided that he remembers the names of the scenes, he will search for them, using the facilities of the operating system. Then for each scene he will have to select the correct renderer. He might have to check the user manual to find out how the rendering resolution is set for this particular renderer. Then he will start the renderer and wait until the image has been produced. As the image most probably will not be in the JPEG format, the user will have to find appropriate conversion software and apply it to the image. These steps will have to be repeated for each scene.

Certainly more comfortable situations for the user can be imagined. He rather wants to have the scenes organized within a database and simply tell the system to do the following:

> *Recalculate the given scenes to fit resolution $x \times y$. Convert the resulting images to the JPEG format.*

This however requires a database capable of storing different representations of scene data. The database must also offer a possibility to integrate existing software and to use this software for data manipulations. In our example these are the different renderers and the conversion tools. The processing step 'recalculate the given scenes...' is formulated on an abstract level, leaving it to the database system to decide which renderer to select and how to set the resolution parameters. Therefore, a mechanism allowing the user to formulate data manipulation sequences on an abstract level is required. As the data manipulation will be applied to a group of specific scenes, a possibility to define collections of data is needed as well.

3.2. Application 2 : query by information content

A user of a computer-based furniture catalogue is looking for a particular chair he has in mind. As he cannot *remember the name of the chair, he prefers to formulate his query by sketching the object he is looking for.*

This application is an example for the various retrieval methods which should be offered by a powerful graphics database. Often textual query methods, based on keywords associated with the data, are not very suitable to search for visual information. In many cases content-based methods allowing the user to specify shapes or colours are more promising [3][8][13]. Such methods have already been successfully implemented in image databases like QBIC (Query by Image Content) [5]. However these methods have been specifically designed for image data. Research on content-based retrieval methods for other kinds of graphics data, such as 3D objects, is only beginning [9].

> *"The real merit of a visual information retrieval system is its ability to allow enough extensibility and flexibility that it can be tuned to any user application."* [8]

Therefore the next step in this area is to offer a flexible and extensible testbed capable of supporting research for new content-based retrieval methods.

3.3. Application 3 : cooperation of research teams

Two research teams want to cooperate: the first team is working on face recognition and owns a 3D scanner to scan human faces. The scanned face-models are used to test the team's latest face recognition software based on 3D information. The second research team works on modelling and animation of human faces and develops modelling software allowing to model different facial expressions. A fruitful cooperation of both teams should allow the first team to examine the robustness of its recognition software with respect to particular changes in facial expressions which have been modeled using the software of the second team. The second team could compare its face-models to real data. In order to realize the cooperation, a database of human faces has to be established. The database would have to work with the software of both teams.

This application illustrates the need for a flexible system capable of easily integrating various existing software packages. It is an example illustrating how profitable data reuse among different subareas of computer graphics can be.

The question arising now is: could the three applications above be implemented using existing graphics systems like BOOGA[1]? As to Application 1, an apparent solution might be to convert the scenes to BOOGA format and render them using a BOOGA renderer. This approach results very likely in quality loss. The problem is that different renderers cannot completely replace each other. Applications 2 and 3 cannot be realized using BOOGA, simply because BOOGA lacks the database facilities needed. An apparent solution might be to extend BOOGA, e.g. by providing data persistence by an OODBMS, and additionally by implementing content-based retrieval methods. The drawback of such a solution would be that it only supports BOOGA applications. To support another graphics system, one would have to largely rewrite the database part. This would not be necessary when using a universal graphics DBMS.

[1]We will use BOOGA as a representative of state-of-the-art graphics systems.

3.4. Specific Requirements on a Universal Graphics DBMS

In order to satisfy the needs of many potential applications, a universal approach to a graphics DBMS has to be found, without limiting the usage of the DBMS to a specific application area. Enough extensibility and flexibility have to be provided in order to meet the needs of various existing and future applications in the graphics domain. The following list states the most important requirements:

1. Provide persistent storing of *multiple representations* for all kinds of data types found in the graphics domain.

2. Provide basic retrieval methods and a *mechanism to plug-in* new content-based retrieval methods.

3. Provide *seamless integration* of existing graphics functionalities *from different sources*.

4. Provide a mechanism to formulate data manipulation operations by *composition of integrated functionality*.

5. Allow the specification of *abstract data manipulation operations* to be concretisized at run time.

6. Provide mechanisms to maintain *collections* of data and *relations* between stored data.

Satisfying the first three requirements implies high degrees of data reuse. Requirement 1 means that the system must be capable of supporting a large basis of existing data. Requirement 2 means that the user must have the facilities needed to efficiently find the relevant data in large data collections. As pure data is useless without functionality, Requirement 3 means that it must be possible to integrate many operations for data manipulation. Satisfying simultaneously Requirements 1, 3 and 4 guarantees high degrees of extensibility. Requirement 4 means that it must be possible to define complex data manipulation sequences. Requirement

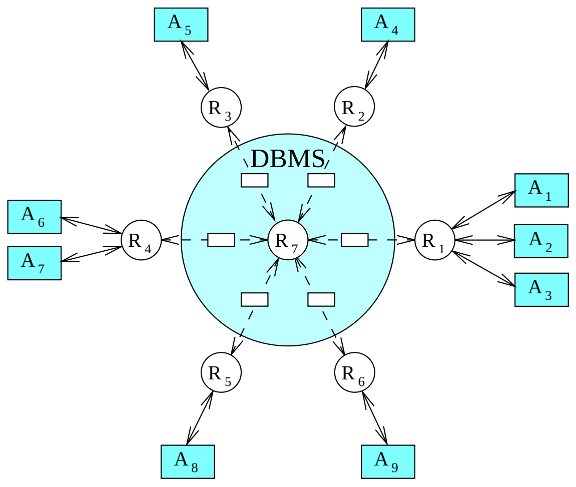


Figure 1. Single persistent representation.

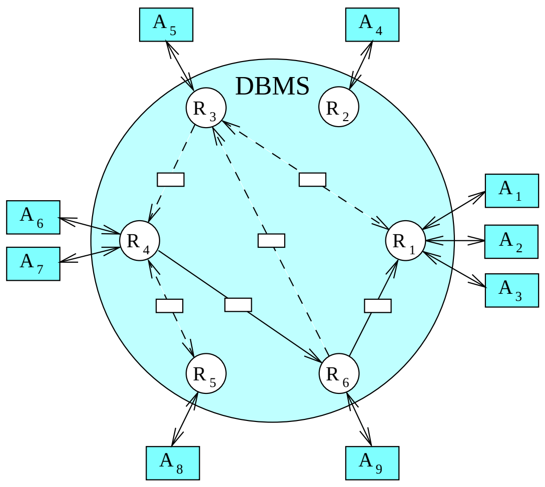


Figure 2. Multiple persistent representations.

5 demands that it must be possible to deal with multiple representations in a uniform way. Finally satisfying Requirement 6 enables applications to hold their own collections of data and allows them to define their own relations between stored data.

It follows that the key issues to be tackled by our approach are the *multiple data representations* to be coped with and the *high degrees of extensibility* which have to be achieved. We will discuss these problems in the following two sections.

4. Coping with Multiple Representations

In order to support data reuse and reuse of functionality among different graphics applications, a database management system must be capable of simultaneously holding data from different sources [15]. The problem is that even for semantically similar data, different applications often use different data representations. From the database point of view, the question is which representation should be used for the storing of persistent data. Basically there are two alternatives, as follows:

1. Use a single representation to store persistent data.
2. Allow coexistence of multiple representations in the database.

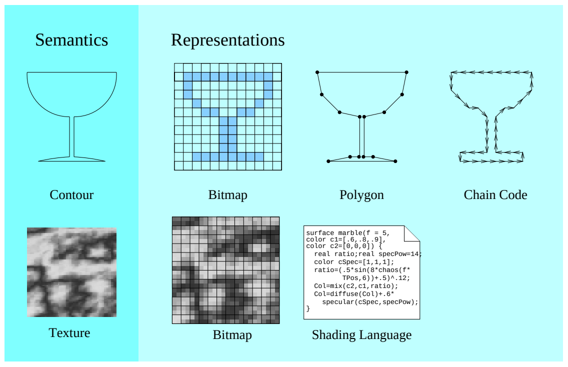
In the following, a representation R_i will be called a *super representation* of R_j if everything which can be expressed in R_j can also be expressed in R_i. Figure 1 shows a DBMS using a single representation, i.e. R_7, for its persistent data. Applications $A_1, A_2, \ldots, A_9$ work with representations $R_1, R_2, \ldots, R_6$. These applications can only be sure of saving and reloading data without loss of information if R_7 is guaranteed to be a super representation of $R_1, R_2, \ldots, R_6$[2]. In practice, there are many cases where this property is hard to fulfill. Fulfilling this property in an open system, with frequent integration of new applications using new representations, is even harder. Hence, the approach of using a single persistent representation is not very practical.

A more pragmatic approach is the second one, i.e. to allow coexistence of different representations in the DBMS. The idea is to store the data always in its original form. As no information loss caused by conversions will occur, the database will hold the maximum possible information contents (cf. Figure 2). Adding new applications to work with the DBMS is rather straightforward: new representation types have simply to be registered. Generic check-in and check-out mechanisms are provided by the DBMS. This approach has the advantage that converters must only be written when they are really needed. As the original information is stored in the database, the system can also run with simple incomplete converters which can be replaced by improved versions later on.

4.1. A Type Mechanism for Heterogeneous Data Representations

In a DBMS with coexistence of multiple representations for semantically similar data types, specific operations will only work with specific representations. Type-safe execution of operations has to be assured. A representation can be regarded as the implementation of a semantic data type.

[2]If, in the mean time, data has been loaded and rewritten by an application using another representation, information loss can still occur. This problem can be solved by letting the database hold different versions of data simultaneously.



```
surface marble(f = 5,
color c1=[.6,.8,.9],
color c2=[0,0,0]) {
  real ratio;real specPow=14;
  color cSpec=[1,1,1];
  ratio=(.5*sin(8*chaos(f*
         TPos,6))+.5)^.12;
  Col=mix(c2,c1,ratio);
  Col=diffuse(Col)+.6*
    specular(cSpec,specPow);
}
```

Figure 3. N:M correspondence of semantics and implementations.

Semantics and implementations of data are often, but not always, in a one to one correspondence. E.g. a semantic type called ‘image’ may be represented by different image formats such as PPM, RLE, GIF, or JPEG. Another example is a contour which may be represented by a bitmap, polygon, or chain-code, as it is illustrated in Figure 3. On the other hand, a representation type may be used to represent semantically different things. In Figure 3 a contour and a marble texture are both represented by bitmaps. In such a case it is very important to distinguish the semantics: what is the meaning of the data stored in the bitmap?

Now let us consider the following situation: An operation A produces a 2D contour of a 3D object viewed from a certain viewpoint. The resulting contour is given in bitmap representation. An operation B is capable of extracting features out of contours provided that the contours are given in polygon representation. Obviously some kind of converter C is needed which is capable of converting contours represented by bitmaps to contours represented by polygons. It has to be assured that C is not applied to a bitmap, simply because such an operation would not make sense. This can be achieved by distinguishing semantic types. On the other hand it must also be assured that B cannot be applied to contours not represented by polygons. This can be achieved by distinguishing implementation types. As the example shows, both semantics and implementations of data have to be clearly distinguished.

Integration environments separating semantics and implementation have already been described in the literature [12]. Separation of semantics and implementation leads to a more flexible system design. GSCOPE introduces two types for all kinds of data: the semantic data type describing the meaning of the data, and the representation type identifying the concrete implementation of the data. The input and output parameters of an operation are described by type pairs. In the following the list of type pairs describing the input and output parameters of an operation will be called the *signature* of the operation [19]. Operations may only be applied if the parameters of the operands match the signature of the operation with respect to both, the semantic type and the implementation type. For the example described above, the operations A, B and C can be expressed in the following way:

$$\begin{aligned}
(\mathit{Object3D}, \mathit{VRML}) &\longrightarrow^{A} (\mathit{Contour}, \mathit{PPM})\\
(\mathit{Contour}, \mathit{PPM}) &\longrightarrow^{C} (\mathit{Contour}, \mathit{Polygon})\\
(\mathit{Contour}, \mathit{Polygon}) &\longrightarrow^{B} (\mathit{SOCFeature}, \mathit{ASCII})
\end{aligned}$$

We have also considered a more general approach using semantic type lists instead of a single semantic type. But when implementing GSCOPE we have found that single semantic types already provide sufficient flexibility.

However significantly improved flexibility can be achieved by using *type hierarchies* and operation overloading. Figures 4 and 5 show a possible hierarchy of semantic types and, respectively, of implementation types. An edit operation defined to work on an ASCII implementation of any semantic type might start up an ASCII editor. As the VRML implementation type inherits all operations from ASCII, the edit operation may also be applied to the VRML data, allowing a user to directly edit VRML code. However the operation may also be overloaded for VRML implementation types, e.g. a true VRML editor may be started up.

We can also specify pure semantic operations which work on data having undefined implementations (symbolized by '$*$'). Accordingly, a general raytrace operation may be given as follows:

$$(\mathit{Scene3D}, *) \longrightarrow^{Raytrace} (\mathit{Image}, *)$$

At run time this purely semantic operation will be replaced by a concrete implementation according to the implementation type of the input data. In our example one of the following implementations of a raytracer could result:

$$\begin{aligned}
(\mathit{Scene3D}, \mathit{Ray}) &\longrightarrow^{Rayshade} (\mathit{Image}, \mathit{RLE})\\
(\mathit{Scene3D}, \mathit{POV}) &\longrightarrow^{POVRay} (\mathit{Image}, \mathit{PPM})\\
(\mathit{Scene3D}, \mathit{BSDL3}) &\longrightarrow^{BoogaRay} (\mathit{Image}, \mathit{PPM})
\end{aligned}$$

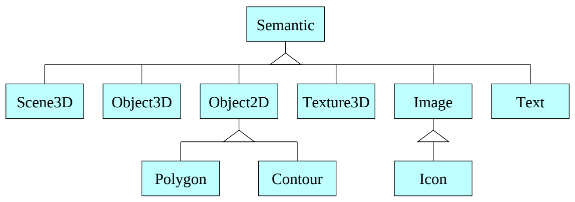


Figure 4. A hierarchy of semantic types.

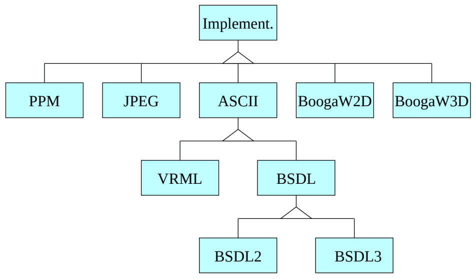


Figure 5. A hierarchy of implementation types.

Purely semantic operations allow the user to program on a semantic level without having to bother about implementations. It is also possible to define operations working on data having a specific implementation type regardless of the semantic type. The following operation converts BSDL3 data to Rayshade data:

$$(*, \mathit{BSDL3}) \quad \longrightarrow^{BSDL3ToRay} \quad (*, \mathit{Ray})$$

This operation could be applied to semantic types like Scene3D, Object3D, Texture3D or others. Of course an operation can also be restricted to work only on a specific branch of the semantic type hierarchy. Operations that may be applied to any kind of data can also be defined. For instance the following operation gets the name of any kind of data:

$$(*, *) \quad \longrightarrow^{GetName} \quad (\text{Text}, \mathit{ASCII})$$

Using the type mechanism as described in this section means satisfying Requirements 1 and 5 of Section 3. It has been implemented in the GSCOPE system.

5. Achieving Extensibility

An extensible system is capable of being adapted to new needs by acquisition of additional functionality. New functionality can be acquired by implementing it in the system or by importing it from other systems. With both approaches the system has to provide an interface to plug-in the additional functionality. Many systems offer specialized interfaces allowing to integrate specific functionality. This is the case e.g. with many raytracers which provide a predefined interface to primitive objects. These raytracers may be extended to support additional primitives which obey the predefined interface. In object-oriented programming such extensibility is usually realized by means of inheritance. However far better extensibility can be achieved if a system supports an interface to plug-in general functionality. As general functionality will have various input and output parameters, a descriptive interface to functionality is required. In order to easily deal with general functionality, a user needs a uniform view of this functionality i.e. a uniform way to access various kinds of functionality. This can be offered by so-called *software components* [18].

The term *software component* has different meanings depending on the people using it. In this paper software components are considered black-boxes offering a service. The implementation of a component is hidden from the user. Access to component functionality is provided in a uniform way by means of a service interface describing the arguments and the resulting data. A configuration interface allows the components behavior to be adapted to different needs. A composition mechanism also allows components to be plugged together to form new components or applications. Well designed components act as building blocks when constructing software and can increase reuse of functionality dramatically. The functionality of a component can be obtained in two ways: by implementing new functionality within the system, or by importing functionality from other systems. In most cases the first alternative results in an expensive implementation of the functionality in a specific programming language. Functionality to be integrated can have different forms, ranging from executable programs to source code written in a particular language. With a component approach one has to define special components which translate the specific interface of the functionality to the component interface. Such components are often called *wrapper components*. If the functionality is given as source code, specialized wrapper components will have to be written which provide access to it. If the functionality already supports a component mechanism, generic wrapper components can be defined, translating from one component model to the other. *Command line executables* may be regarded as components. The input and output files correspond to the service interface, and the command line options correspond to the configuration interface. Therefore, the same technique can be applied.

Functionality to be integrated will usually only work with specific representations of data. Therefore integrating new functionality often means that new representations have to be integrated. An open system has to provide facilities to add new data types. These data representations can vary from file-based representations to complex data representations held in main memory. File-based representations may be easily integrated by generic data wrappers. Such wrappers encapsulate data of a specific data type, thus allowing the data to be passed from one component to the other. For memory-based representations specialized wrappers have to be built in order to provide access to the data.

File-based representations may be made persistent by generic check-in and check-out mechanisms. Making memory representations persistent is rather difficult. In most cases it is better to convert a main memory representation to a file-based representation. Converters will also be needed to translate different representations of data into each other in order to apply specific functionality. The wrapper components, the data wrappers, and the converters provide together the basis of an open and extensible system. They allow to integrate a large variety of functionality and representations which may be used transparently. Introducing a component-based integration facility will fulfill Requirements 3 and 4 of Section 3. The mechanisms described in this section have been implemented in the GSCOPE system.

6. The Architecture of GSCOPE

After having presented the necessary preparations in Section 4.1 and 5, we now can look at the architecture of GSCOPE. An overview of the architecture of GSCOPE is given in Figure 6. A component concept as described in Section 5 has been chosen to enable seamless integration of existing software. Components define a uniform view of functionality and therefore simplify access to that functionality. In Figure 6 this uniform interface is indicated by a software bus which allows new functionality to be easily plugged-in. The functionality and the data types that are available have to be registered somewhere. In GSCOPE this is the duty of the so-called *component-oriented database management system* (CODBMS) layer. This layer is also responsible for providing basic database operations and facilities which help to guarantee database consistency. The type mechanism implemented in this layer corresponds to that presented in Section 4. The CODBMS layer has been implemented on top of an ordinary object-oriented database management system (OODBMS) corresponding to the ODMG-93 standard [4]. It provides access to the actual data repository.

Figure 7 – in UML notation – gives an overview of the relations between GSCOPE's data, data types, and components. According to Section 4.1, any data has a semantic type and an implementation type. Therefore the input and output data of components have to be specified as type pairs describing the semantics and implementations of each input and output parameter. The components as implemented in GSCOPE correspond to those described in Section 5. The configuration interface has been implemented as a list of parameters. Before a component is executed, it configures itself according to the parameter values set in the configuration interface. In the following we will have a closer look at the CODBMS layer and then show how this layer has been extended to arrive at the actual GSCOPE system.

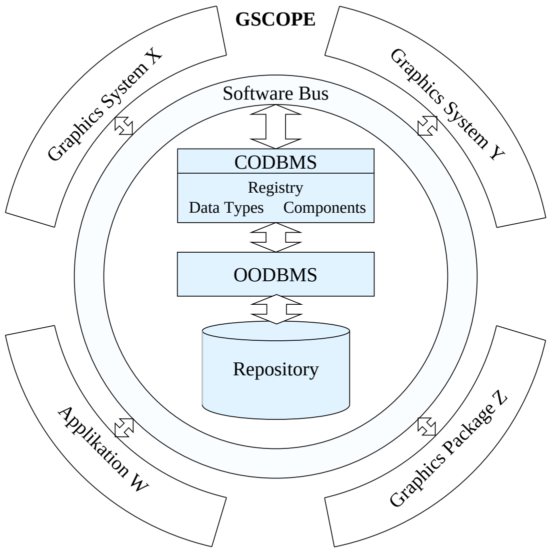


Figure 6. GSCOPE overview.

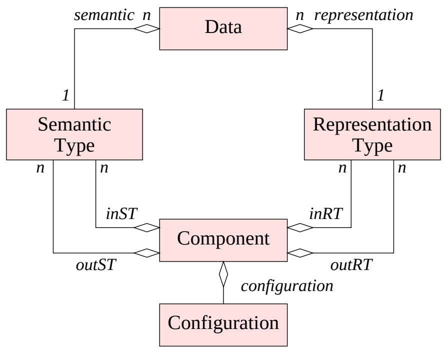


Figure 7. The relations between data and components.

6.1. The CODBMS Layer

The CODBMS layer is the heart of the GSCOPE system. It is responsible for performing data manipulation by components, as follows:

1. The data is checked-out of the repository and locks are set.
2. The component manipulates existing data or possibly creates new data.
3. After the execution of the component the manipulated data is checked-in into the repository and locks are released.

Data manipulation performed by a component has to transform the database from a consistent state to another consistent state. That is, each component defines a transaction. All the component's actions are performed within this transaction which is either committed or aborted as a whole [4].

The components together with configuration and composition mechanisms form the DML (data manipulation language) of the CODBMS. The CODBMS layer provides a set of predefined components for primitive database operations. Such operations include setting or removing relations between data, navigating through relations, selecting elements of a collection, assigning values to attributes, and generic check-in and check-out mechanisms applicable to most data representations. These predefined components provide an interface to the different managers of the CODBMS layer.

Figure 8 gives a more detailed view of the CODBMS layer. The data type manager implements the concept of separated semantic and representation types as presented in Section 4 and Figure 7. Additionally, the data model allows inheritance of semantic types, representation types, and components. Inheritance of semantic types and representation types works as described in Section 4. The information on registered components, configurations, semantic types, representation types and hierarchies is stored in the repository.

This CODBMS layer represents a universal architecture which could potentially be used in other domains than graphics as well. Domain-specific layers are built as extensions of this layer, i.e. by adding domain-specific data types and domain-specific components. Next we will illustrate how a graphics domain layer can be built on top of the CODBMS layer.

6.2. The Graphics Domain Layer

The graphics domain layer is built on top of the general CODBMS layer by means of integrating existing graphics software. After having made the decision which graphics software should be integrated, the data types and operations offered have to be identified. Carefully analyzing the data semantics helps specifying the semantic data types and the corresponding implementation types. Then operations can be integrated by means of wrapper components as it is discussed in Section 5.

In order to verify our concept, we have built a small graphics domain layer which integrates the following software: the graphics framework BOOGA, the graphics package *ImageMagick*, and the raytracing software *Rayshade*. The integration of the BOOGA framework was straightforward. As BOOGA already supports the component concept, it was only necessary to supply wrapper components in order to translate the BOOGA component interface to the GSCOPE component interface. Additional wrappers had to be written in order to encapsulate internal representation types of the BOOGA framework. These wrappers together enable GSCOPE to transparently use all of the BOOGA components. Integrating command line-based software was even simpler. Unix commands may be regarded as components. The input and output files correspond to the service interface, and the command line options correspond to the configuration interface. A generic wrapper for Unix commands provides access to *Rayshade* and to the different command line applications offered by the *ImageMagick* software package. Using this wrapper, integration of the functionality offered by Unix commands can be performed at run time, without need to recompile or restart GSCOPE.

7. Application of Gscope

Now some of the possibilities of GSCOPE will be demonstrated by showing how the three applications discussed in Section 3 can be implemented.

7.1. Application 1

Figure 9 indicates an implementation of Application 1. The input is a collection of 3D scenes given in different forms of representation. The output is a collection of rendered images, all given in the JPEG format. The following components can be used to build Application 1:

- The `ForEach` component iterates through a collection of data and sends each data element to the subsequent command.
- `DBExport` is a generic component used to check-out data from the database.
- `Raytrace` is a pure semantic component. According to the input representation, this component will call the corresponding implementation of a raytracer.

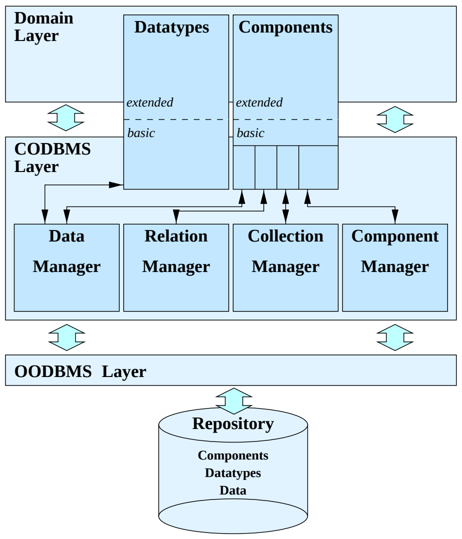


Figure 8. CODBMS layer and domain layer extension.

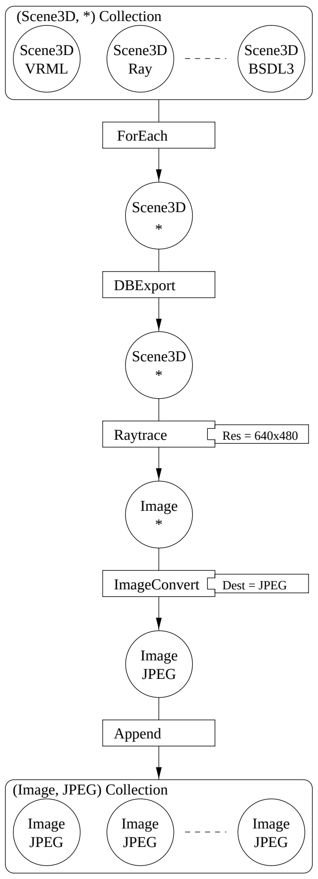


Figure 9. Implementation of Application 1.

- `ImageConvert` is capable of converting the image from most image formats to the output format as specified in the configuration.

- The `Append` component appends data to an existing collection of data.

The desired image resolution is set in the configuration section of the `Raytrace` component. Large parts of the command sequence in Figure 9 have been defined on a pure semantic level, i.e. the application is not restricted to specific input representations.

7.2. Application 2

Content-based retrieval is based on features describing the data and not on the data itself. There are two main steps needed to implement content-based retrieval methods. In the first step, features have to be identified in the data, and an index structure has to be built. The second step is the actual query step. A retriever gets a query feature and has to find the data possessing feature properties similar to those of the query feature.

We will now look at an example showing 3D object retrieval based on 2D shape retrieval. The idea is to search for a 3D object by comparing the contours of its projections orthogonal to the object coordinate axes with a query contour given by the user. Figure 10 illustrates the feature extraction part. The application starts with the so-called *extent* of 3D objects, i.e. a specific collection of all 3D objects. GSCOPE stores an extent for each data type. For each 3D object, shape-based features are extracted and used to build an index on 3D objects. The following components are important in this application:

- `SceneConvert` converts different 3D object representations to BOOGA3 representation[3], normally without any substantial information loss.

- `OrthoScenes` reads an Object3D in BOOGA3 representation and produces three scenes of the object which are placed in front of a white background. These scenes represent 'orthogonal views' corresponding to projections along the three axes of the object coordinate system.

- The `Contour` component identifies the boundary pixels of a connected region which is given as a PPM bitmap.

- The `SignOfCurv` component calculates the *sign of curvature* feature of a contour.

[3]BOOGA3 representation is the representation of a BSDL3 file loaded in main memory.

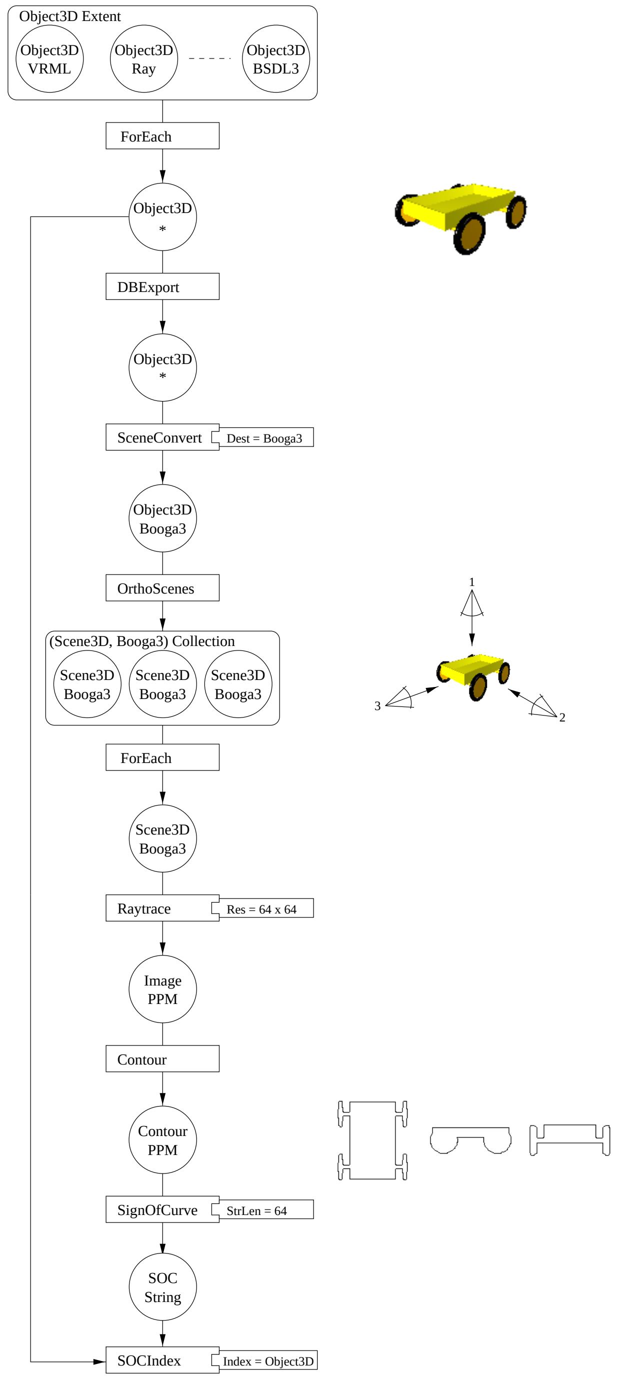


Figure 10. Feature extraction.

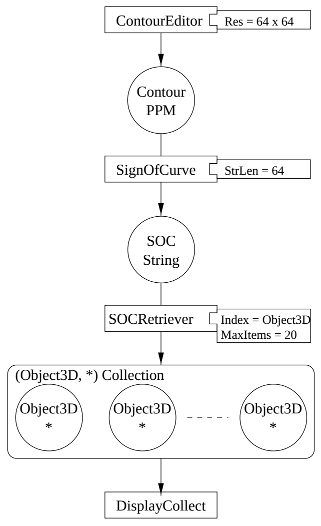


Figure 11. Shape retrieval.

- `SOCIndex` generates a feature index entry of a sign of curvature.

Figure 11 shows the retrieval part. The user defines a query shape, and the application returns the ‘most similar’ 3D objects.

- `ContourEditor` allows the user to draw a query contour.
- `SOCRetriever` finds in the index, specified by a configuration parameter, the data having the most similar *sign of curvature* features.
- `DisplayCollect` displays data collections to the user. Normally this data will be displayed as an icon together with a name.

Other retrieval methods based on different feature types can be integrated into GSCOPE by providing four components, i.e. for feature extraction, feature indexing, feature editing and feature retrieval.

7.3. Application 3

An implementation of Application 3 shall only be sketched. As for the ‘face recognition team’, the database will have to store a collection of faces, each face associated with a collection of scanned face-models showing different expressions of the same face. A semantic type `Face` together with an implementation type `FaceRep1` will be needed. For the data stored by the ‘face modelling team’, an additional implementation type `FaceRep2` will be needed. As `FaceRep1` and `FaceRep2` most probably are file-based representations, persistent storage of these representations can be achieved by generic methods offered by GSCOPE. There has to be implemented at least one converter component, allowing to convert from `FaceRep1` to `FaceRep2`. Converting in the opposite direction is not necessary if the features used by the recognition software can also be extracted from `FaceRep2`. The software of the two research groups can easily be integrated utilizing generic wrappers of GSCOPE. Using GSCOPE, only little effort must be made to provide a database for such a cooperation. Both teams continue to use the software they are familiar with, but they have additional possibilities at hand.

By showing how to implement the three applications we have demonstrated that GSCOPE is capable of supporting a wide range of applications in the computer graphics domain. The implementation of Application 1 indicates that GSCOPE satisfies Requirements 1, 3, 4, 5, and 6 for a universal DBMS. The implementation of Application 2 indicates that Requirement 2 is satisfied too. As these results hold much more generally, it may be expected that GSCOPE fulfills all the requirements stated in Section 3 for a universal graphics DBMS.

8. Conclusions

Reuse of data and functionality within and in particular among graphics systems is still restricted, mainly due to the large number of different data representations. Even though VRML and JPEG have become standard representations for Internet applications, a significant tendency to universal representations cannot be detected. Specialized graphics applications will probably always rely on their own data representations. Nevertheless there is an increasing demand for reuse of existing graphics data and functionality. Considering this, we believe that the GSCOPE architecture offers a promising approach to graphics data reuse, mainly because it is designed to simultaneously store multiple data representations of various data types. As the data is stored in its original form, initial information loss can be avoided. As the typing model separates semantics from implementation, it guarantees type safeness and at the same time provides high degrees of flexibility. In many cases the user can program on a semantic level, without having to bother about concrete implementations. A high degree of extensibility is achieved by providing software components. They allow to seamlessly integrate various kinds of graphics functionality which ranges from command line applications to graphics frameworks. Moreover, composition of components allows existing graphics functionality to be reused in a powerful way. As indicated in this paper, first experiences have already shown that different kinds of graphics software can easily be integrated by means of wrapper components. The GSCOPE system is also capable of serving as a testbed for experimenting with visual information retrieval methods. Its concepts are not limited to the computer graphics domain. However the architecture of GSCOPE still needs to be tested in larger scale environments.

There are many interesting directions for further research e.g. the development of new visual retrieval methods for various graphics data, the development of visual programming interfaces, and the improvement of component composition methods. To further improve the reuse and integration possibilities of GSCOPE, a scripting language should be developed and included, and also additional support for the integration of interactive components having different graphical user interfaces. We believe that a system like GSCOPE will become increasingly advantageous if it can be easily made available to a large number of users. Therefore some of our current research focuses on making parts of the GSCOPE system accessible over the Internet within a web browser.

References

[1] S. Amann. Komponentenorientierte Entwicklung von Grafikapplikationen mit BOOGA. Inauguraldissertation der Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern, Nov. 1997.

[2] S. Amann, C. Streit, and H. Bieri. BOOGA: A component-oriented framework for computer graphics. In *GraphiCon'97 Moscow*, pages 193–200, May 1997.

[3] A. Brink, S. Marcus, and V. S. Subrahmanian. Heterogeneous multimedia reasoning. *IEEE Computer*, 28(9):33–39, Sept. 1995.

[4] R. G. G. Cattel, editor. *The Object Database Standard ODMG-93.* Morgan Kaufmann, 1994.

[5] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, 28(9):23–38, Sept. 1995.

[6] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 2nd edition, 1990.

[7] E. Gamma, R. Helm, R. Johnsen, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[8] A. Gupta and R. Jain. Visual information retrieval. *Communications of the ACM*, 40(5):71–79, May 1997.

[9] R. Jain, H. Li, and D. A. White. 3d information management: Tele-manufacturing and shape retrieval. Visual Computing Laboratory, University of California, San Diego `http://vision.ucsd.edu/paper/manu/-manu.html`, Sept. 1995.

[10] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming (JOOP)*, June/July, 1988.

[11] A. Meier. *Methoden der grafischen und geometrischen Datenverarbeitung.* Teubner, 1986.

[12] T. D. Meijler. *User-Level Integration of Data and Operation Resources by Means of a Self-Descriptive Data Model.* PhD thesis, Erasmus University Rotterdam, Sept. 1993.

[13] V. E. Ogle and M. Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer*, 28(9):40–48, Sept. 1995.

[14] OpenGL Architecture Review Board. *OpenGL Reference Manual. The Official Reference Document for OpenGL, Release 1*. Addison-Wesley, 1992.

[15] H. Pahle and C. Hübel. Ansätze einer adaptierbaren Datenorganisation in einer integrierten Entwurfsumgebung. In F. L. Krause, D. Ruland, and H. Jansen, editors, *CAD '92*. Springer Verlag, 1992.

[16] P. Strauss and R. Carey. An object-oriented 3d graphics toolkit. In *Computer Graphics*, volume 26(2) of *SIGGRAPH Proceedings*, pages 341–349. ACM Press, July 1992.

[17] C. Streit. BOOGA: Ein Komponentenframework für Grafikanwendungen. Inauguraldissertation der Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern, May 1997.

[18] A. Weinand. Components: Another end to the software crisis. In *Components User Conference (CUC)*, 1996.

[19] M. Weiss. *Algorithms, Data Structures, and Problem Solving with C++*. Addison-Wesley, 1996.