BOOGA

A Component-Oriented Framework for Computer Graphics

Stephan Amann Christoph Streit

Hanspeter Bieri

Institut für Informatik und angewandte Mathematik
Universität Bern

December 23, 1996

Abstract

This paper describes the core characteristics of BOOGA, a framework for 2D and 3D computer graphics. BOOGA offers a unified 3-layer model for many areas, including rendering, image processing, computational geometry and scene reconstruction. The framework consists of ready-to-use components to support rapid application development. New components may be seamlessly integrated into the existing system.

1 Introduction

An important trend in the computer industry is the increasing use of computer graphics in many different application areas, from simple still images for a marketing campaign to completely computer generated motion pictures. Object-oriented technology is well suited for the implementation of graphics systems [16, 14, 9], since the mental model of a graphical object maps easily to entities directly supported by this technology, like classes or objects.

BOOGA (**B**erne's **O**bject-**O**riented **G**raphics **A**rchitecture) is an object-oriented graphics framework aimed at a wide range of application areas within the domain of computer graphics, namely *Geometric Modelling*, *Image Synthesis*, *Image Processing*, *Image Analysis*, *Scene Recognition* and *Computational Geometry*. The framework serves as a research platform for computer graphics at the University of Berne, and supports experiments with existing algorithms, and development of new ones within the various application areas. As a research platform, BOOGA has to fulfill somewhat conflicting requirements, some of which are ease of use, flexibility to meet future unexpected needs, high degree of reusability, support of rapid application development (*RAD*). Run-time efficiency, however, is not a central concern. Thus, BOOGA supports the developer by defining a generic architecture for different types of graphics applications, and offers a pool of ready-to-use algorithms for the various areas considered.

In contrast to most existing systems, BOOGA is not limited to a specific application area, but offers a unified view of the entire computer graphics application domain. It provides data abstractions and mechanisms for 2D *and* 3D objects and includes a *component layer* to model high-level operations that manipulate or generate graphical objects of different types. BOOGA currently consists of about 200,000 lines of C++ code distributed over more than 500 classes, and already provides many ready-to-use components that may be plugged together to build applications. A number of small and large applications have been built using BOOGA, for example renderers, editors and browsers.

In the following paragraphs, we first discuss some fundamental issues, divided into a graphical part (Section 2) and an architectural part (Section 3), which are largely orthogonal to each other. The main emphasis is on the architecture of BOOGA (Section 4).

A new component notation (Section 5) is introduced and a few sample applications are presented (Section 6). Finally, some conclusions are drawn (Section 7).

2 Graphical Issues

Graphics applications traditionally deal with a large variety of application areas[1], and numerous systems are available to support the development process. Figure 1 shows a common classification [11]. An important goal when developing BOOGA was to achieve a generic architecture. The classification in Figure 1 served as a good starting point when we looked for an appropriate concept. A system built according to this approach has to handle 2D as well as 3D objects and must deal with conflicting requirements of vector and raster graphic applications. In the following paragraphs, we will have a closer look at important characteristics regarding input and output data within the different application areas.

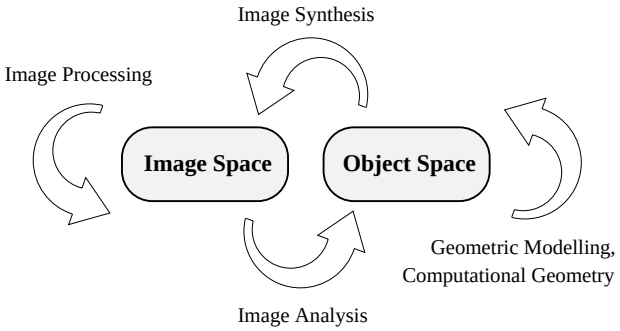


Figure 1: *The different application areas of the computer graphics domain.*

Image Processing
Algorithms within this application area are characterized by a strong emphasis on raster graphics objects, usually represented by bitmaps of different sizes and depths.

Image Analysis
Image analysis applications deal with raster objects as input, preprocessed by image processing operations. The application may result in a description of what is contained in the image, usually represented by a 2D or even 3D scene, depending on the nature of the application. A scene description may contain a large variety of geometric primitives or even application-specific types.

Image Synthesis
A 2D or 3D scene description is transformed into a raster object by applying hidden surface removal, rasterization and illumination algorithms. Scenes are usually organized as trees or directed acyclic graphs (DAG).

Geometric Modelling and Computational Geometry

Geometric Modelling creates or transforms 2D or 3D scene descriptions. Usually, the scene is organized hierarchically to represent its structure. Special structural objects (aggregates) are used to achieve this.

Computational Geometry applications can also be considered as handling scene descriptions, i.e. collections of objects that are organized according to the application's needs. Most algorithms deal with problems in 2D or 3D, but computational geometry is not limited to these dimensions.

A system supporting these application areas has to deal with a wide range of different data types. Not only are there raster and vector objects to be manipulated, but the system also has to satisfy demands for 2D and 3D data structures.

3 Architectural Issues

In order to make BOOGA a research platform, it is important that new object types and new algorithms may easily and seamlessly be integrated into the existing core. The concepts necessary to achieve this goal are described in Section 4.

Using BOOGA, it is possible to design, implement and test new algorithms and whole applications in different areas of computer graphics in a fast and easy way. It has been proven that new software elements, once implemented conforming to the guidelines of BOOGA, are reusable in future work. These objectives could only be achieved by means of a clean system concept (see Section 4) and a list of architectural guidelines:

Design for local extensiblity

BOOGA has evolved into a large software system. The work of several researchers and students has already been based on the structure of BOOGA or on

[1] We denote the whole variety of computer graphics *application domain*, and the various subdomains *application areas*.

parts of it. Obviously, it is hard to extend a system if its elements are very tightly coupled. Decoupling becomes even more critical when developing a platform for research, since future activities are less predictable than in other environments.

Another critical factor is the number of independent developers. Research is done by individuals or small teams. Although all their work is based on BOOGA, researchers have a strong desire for independence.

This leads to the main guideline for the development of BOOGA:

Extensions in one part of the system must have no influence on other parts.

As an example, new types of geometric objects can be added without changing the existing system. Every geometric object is derived from one of the object classes (see Figure 4) and implements a small number of protocols. All existing components, such as parsers, renderers, and editors, can deal with these new objects because they are based on these protocols.

Provide good defaults for the 95% cases

Several reuse techniques exist [10], *composition* and *inheritance* being two of them. Composition (an example of *black-box reuse*) and inheritance (an example of *white-box reuse*) are the central reuse techniques directly supported by BOOGA. While composition is easier to use, inheritance often provides more flexibility. This is also known as the trade-off between black-box and white-box reuse [8].

If a class is designed for the normal case (the '95% case': [15]), users can build their applications using the black-box reuse approach, which is always easier, less error-prone and faster than white-box reuse.

There is a trade-off between flexibility and complexity: the more flexible a design is, the more complex it becomes, and the harder it is to use. The 95% rule is a way to trick this trade-off in that the user only has to worry about the complexity when he needs the flexibility.

Most BOOGA classes are built to meet the 95% case. In cases where the defaults are not suitable, there are normally several possibilities to extend the functionality or change the behaviour of an existing class.

Provide flexibility where needed

Flexibility is expensive – writing flexible software is hard, understanding it is even harder. Flexibility is nevertheless necessary because inflexible systems will not be used, and will therefore become even more expensive. In general, it is wrong to provide flexibility where *possible*; it is wiser to provide it only where *necessary*.

To discover the right degree of flexibility is a challenging task. There are, however, a few promising movements in the object community to support a developer in finding the right amount of flexibility, e.g. *hot spots* [12] or *design centers* [6].

The hot spots of BOOGA may change over time, e.g. when exploring a new research topic. The initial design was not meant to meet all possible future needs, but to provide flexibility in situations where new requirements were likely to appear. This changeability over time is best covered by use of an iterative development approach, as discussed in the next guideline.

Use of an iterative development process

> *"A complex system that works is invariably found to have evolved from a simple system that worked." [4]*

The development of BOOGA is based on a strongly iterative approach. Every new application is likely to add new requirements and therefore results in changes to the system. The *waterfall* approach is not applicable, since it is not possible to foresee future demands:

> *"Good frameworks can be used for things that the designers never dreamed of." [7]*

A flexible system design will always reflect this iterative approach, and not only supports but also enforces the process.

Keep the number of concepts minimal

Every new concept added to a system is to some extent a new barrier for current and future users of the system, making the learning curve steeper and steeper. It is critical for the success of a system to keep the overall number of concepts as small as possible. They appear on all levels of abstraction: architectural and design patterns, language specific idioms, naming conventions, and coding styles. Throughout all these levels it is absolutely necessary to focus on as few different concepts as possible.

4 The structure of BOOGA

Analysing the results of most algorithms in computer graphics, only a few input and output types can be found, as already discussed in Section 2. Obviously these algorithms work with either 2D or 3D data as input and yield the corresponding 2D or 3D data as output. This fact meets the needs of BOOGA as an architecture (Figure 2) that is open enough to allow

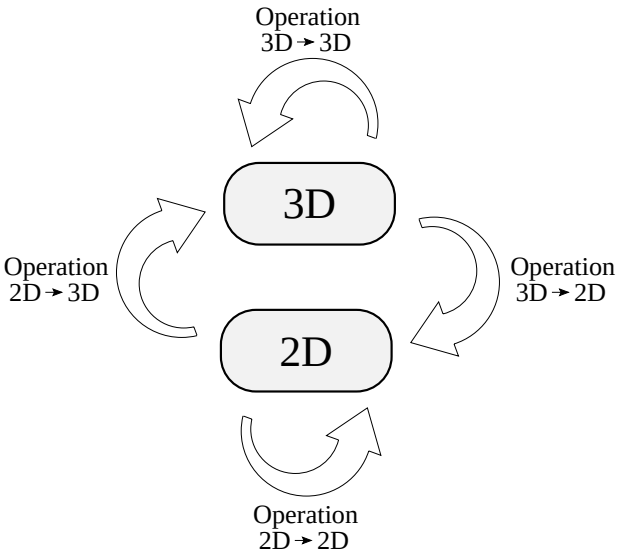


Figure 2: *BOOGA's concept is an abstraction of the classification of Computer Graphics in Figure 1.*

for future integration of all kinds of algorithms and applications, but is also restrictive and simple enough to enforce the development of reusable components. All BOOGA-based applications follow the *dataflow paradigm*, since 2D or 3D data is flowing through interconnected components.

4.1 The 3-Layer Model of BOOGA

Users of a graphics system have many requirements. They may want to build applications using existing components, add new geometric object types, or implement new algorithms. The 3-layer model of BOOGA, consisting of a *library*, *framework* and *component* layer (Figure 3) meets these needs:

Library Layer

Classes for *common tasks*, like string handling, mathematical operations or wrappers to the underlying operating systems, form the library layer.

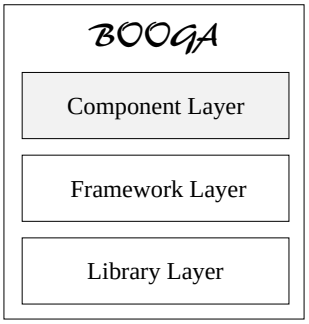


Figure 3: *The 3-layer model of BOOGA: different layers focus on different reuse techniques. White-box reuse is addressed by the library and framework layers, black-box reuse is the concern of the component layer.*

Framework Layer

This layer introduces *geometric objects* for 2D and 3D graphics. They are viewed as data containers supporting a small number of well designed protocols (see Section 4.2). Objects are organized as a directed acyclic graph (DAG) which represents the scene. It may be traversed according to the needs of a specific application.

Working with the framework layer corresponds to the white-box reuse approach.

Component Layer

Components are the basic building blocks of applications and constitute this layer. They accept and produce objects from the framework layer.

The component layer corresponds to a black-box view of the system, making it simple to use.

In the following paragraphs the responsibilities and characteristics of the layers are discussed in detail. The library layer is just a regular class library without any special structural contribution to the overall architecture of BOOGA. It is therefore not discussed further.

4.2 Framework Layer

The framework layer consists of the geometric objects derived from the base class `BOOGAObject`. Unlike some other graphical systems [1], `BOOGAObjects` contain no functionality to display themselves in a specific way. There are several different ways to render geometric objects in 2D and 3D, and drawing is not the only operation that can be applied to objects. Conversions to and from different formats, including ASCII representations, are other examples.

Hierarchy of the Framework Layer

BOOGA explicitly supports 2D and 3D graphics. This architectural decision is reflected in the object hierarchy. Figure 4 shows a simplified representation of the `BOOGAObject` class hierarchy.

Classes representing concrete objects, like spheres, circles or boxes, are derived from the abstract class `Primitive3D` or `Primitive2D`. Structural objects (implemented using the Composite Pattern [5]) inherit from `Aggregate3D` or `Aggregate2D`, respectively. Shared (`Shared3D` or `Shared2D`) objects are multiple references to a single representation and are implemented using the *letter-envelope* idiom [3].

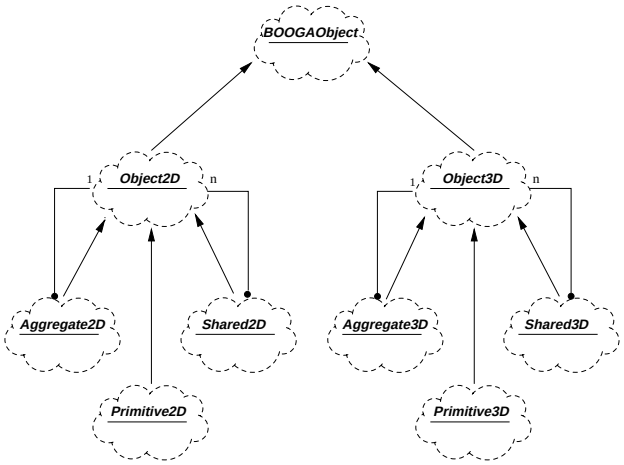


Figure 4: *Simplified hierarchy of the principal data-related classes of the framework layer.*

Concrete examples for aggregates and primitives are: Lists, Grids, Layouters (`Aggregate2D` and `Aggregate3D`); Circle, Line, Polygon, Text, Raster-image (`Primitive2D`); Sphere, Box, Cone, Cylinder, Text, Nurbs (`Primitive3D`).

The separation into 2D and 3D data is one way to design a hierarchy of abstractions. In the following paragraphs, questions concerning this separation are discussed to clarify the reasons for this decision.

Why not separate raster and vector graphics?

A common approach consists in separating raster and vector algorithms. This approach is well known in 2D, where certain applications are dedicated to raster graphics and others are specialized to vector graphics. This model, however, does not offer the possibility to integrate 3D easily. Moreover some applications integrate raster and vector graphics for 2D as well as for 3D graphics.

Why is 2D not a special case of 3D?

Some systems [2] view 2D just as a special case of 3D, where all the z-values are considered to be zero. This allows design of very general algorithms that may handle 2D and 3D cases uniformly. However, 2D graphics is *not* just a special case of 3D graphics. The number of algorithms common to 2D and 3D is surprisingly small. Therefore, modelling 2D and 3D data types separately within the same framework allows the development of more efficient algorithms.

Does explicit support for 2D and 3D neglect reusability?

The separation between 2D and 3D objects offers flexibility for the operations dealing with them, which implies a two-fold hierarchy providing classes for both dimensions. Since most of the base functionality is the same for 2D and 3D, this would result in a duplication of code which, of course, is not acceptable for several reasons. A solution to this problem is the use of templates together with related idioms. Intensive use of these mechanisms makes code duplication for the mirrored hierarchies obsolete.

Mechanisms of the Framework Layer

To allow the integration of new object types, BOOGA's data hierarchy defines a special protocol (i.e. a set of methods). Every object offers at least the following two methods:

`decompose` :
: Not all elements in the system know one another. Some operations may only work with triangles and cannot handle more complex objects such as spheres. In this case, the method `decompose` of the unknown object may be called, which returns an alternative representation. In this example, it might be a triangulation of the sphere. If the resulting representation is also not understood, the decomposition process may be repeated until objects that can be handled are generated.

`intersect` :
: The `intersect` method is the foundation of ray based operations used for ray tracing, ray casting, picking or collision detection.

4.3 Component Layer

Any BOOGA application is the combination of different components, each of them performing a primitive

operation on the data structures from the framework layer. Another responsibility of the component layer is the traversal of the data structures manipulated by the components. Different traversal strategies are available for various situations (see below).

Hierarchy of the Component Layer

The components are divided into four categories, as shown in Figure 5, and directly map to the operation types shown in Figure 2.

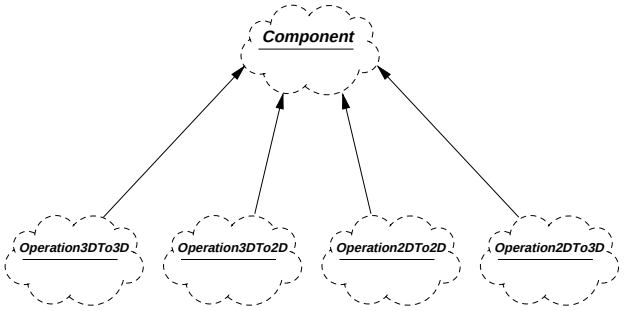


Figure 5: *Simplified hierarchy of the principal component classes.*

Mechanisms of the Component Layer

Components are the active parts within a BOOGA application, offering two types of interfaces:

- The *service interface* is used to invoke the component (method `execute`). It receives a 2D or 3D scene as its input parameter and returns a 2D or 3D output scene. Components should always perform an elementary operation. If dedicated to special purposes they are not very likely to be reused.
- The *management interface* is used for configuration and error handling purposes. This interface allows configuration of the component's behaviour and is used to access additional results and retrieve information from the component.

Components always need a composition model. BOOGA's composition model is trivial since all components offer a standardized way for invocation, namely the `execute` method. The composition mechanism is based upon C++ method calls. Since the protocol is so simple and standardized there would be little effort necessary to implement a composition language or even a visual composition tool. Such a tool could easily be built based on BOOGA itself. Section 5 gives an idea on how a visual composition of components could look.

Traversals

Generally a component has to traverse the graph of the input scene to process every single object. Since most operations either do not care about the order in which the objects are visited or they need to traverse the graph in a predefined way, the traversal process is isolated and mapped to its own hierarchy of traversal classes (according to the Iterator Pattern [5]). This adds the flexibility to use a single component together with different traversals, depending on the component's needs. As an example, in an interactive application, a special traversal object may be used that visits the objects in the scene graph in breadth-first order. As soon as an event occurs (i.e. dragging of the mouse to perform a movement of an object), the traversal process is stopped and the event can be handled by the application. This technique allows fast responses to user requests. On the other hand the scene graph may be pruned.

Each object visited during the traversal is passed to the active component for processing. A component may reject an unknown object. It is the responsibility of the traversal object to react appropriately. The recommended default behaviour is to call the `decompose` method to get an alternative representation of the object.

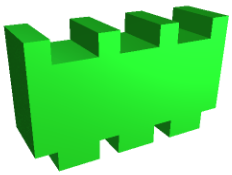
5 Graphical Notation

This section describes a graphical notation which proved helpful to visualize applications (see Section 6) written using the component approach of BOOGA. Each component type is represented by its own brick. The dimension of data flowing through a component is denoted by means of the number of holes for input data and elevations for output data. This notation will be used in the next section illustrating some applications built with BOOGA.

`Operation3DTo3D`

The `Operation3DTo3D` component processes 3D scenes. The result is the possibly transformed input and/or a new 3D scene. Examples of this type of component are parsers or editors.

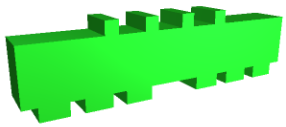
Input scene : 3D
Output scene(s) : 3D



The input scene bypasses the component and may be changed as a result.

. . . . . .

Input scene : 3D
Output scene(s) : 2 × 3D

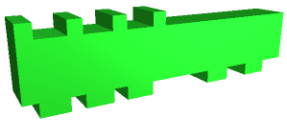


A 3D scene is generated, and the possibly modified 3D input is bypassed.

`Operation3DTo2D`

The `Operation3DTo2D` component generates a 2D scene based on 3D input. Examples are all kinds of renderers.

Input scene : 3D
Output scene(s) : 3D+2D

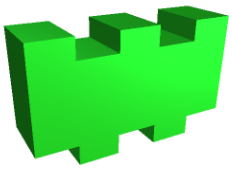


A 2D scene is generated, and the possibly modified 3D input is bypassed.

`Operation2DTo2D`

The `Operation2DTo2D` component processes 2D scenes. The result is the possibly transformed input and/or a new 2D scene. Examples of this type of component are postscript writers or image processing operations.

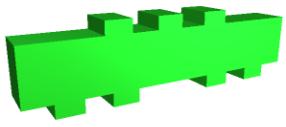
Input scene : 2D
Output scene(s) : 2D



The input scene bypasses the component and may be changed as a result.

. . . . . .

Input scene : 2D
Output scene(s) : 2 × 2D



A 2D scene is generated, and the possibly modified 2D input is bypassed.

`Operation2DTo3D`

The `Operation2DTo3D` component generates a 3D scene based on 2D input. Scene reconstruction algorithms are examples of this type of component.

Input scene : 2D
Output scene(s) : 2D+3D

A 3D scene is generated, and the possibly modified 2D input is bypassed.

Creation

The following two operations create new empty scenes of type 2D or 3D respectively.

Input scene : —
Output scene(s) : 3D

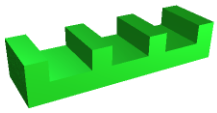
. . . . . .

Input scene : —
Output scene(s) : 2D

Deletion

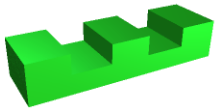
The last two operations delete 2D or 3D scenes.

Input scene : 3D
Output scene(s) : —



. . . . . .

Input scene : 2D
Output scene(s) : —



6 Applications

BOOGA applications are combinations of components. This section gives three examples to demonstrate how applications are built using this approach.

Wireframe Rendering

A wireframe renderer is commonly used to rapidly produce an image of a scene. Each 3D object is represented by a series of appropriately colored line segments. Usually a wireframe renderer produces a pixmap, whereas in our environment the renderer handles a 3D input scene and generates a series of 2D primitives as output, e.g. line or point objects. A pixmap is generated by executing a rasterizer component after the wireframe renderer has done its work. The following components are used to build the whole application (see Figure 6):

- A `Parser3D` component reads an arbitrary scene description file from disk and produces a scene graph of 3D objects.
- The `Wireframe` component generates 2D objects based on the input provided. The result may contain vector or raster objects.
- A `Rasterizer` component is used to scan-convert every object (vector or raster) to a single pixmap.
- The `Display` component displays the generated pixmap directly on the screen.
- `PSWriter` generates PostScript output representing the scene.

Only the following fully functional code[2] is necessary to implement the wireframe application.

```
int main()
{
  // Read a scene from stdin.
  Parser3D parser;
  World3D* world = new World3D;
  parser.execute(world);

  // Compute wireframe view
  // of the scene.
  Wireframe wireframe;
  World2D result1 =
                wireframe.execute(world);

  // Write postscript file to stdout
  PSWriter writer;
  writer.execute(result1);

  // Generate pixmap.
  Rasterizer rasterize;
  World2D* result2 =
              rasterize.execute(result1);

  // Display the world directly
  // on the screen.
  Display display;
  display.execute(result2);

  // Clean up.
  delete result1;
  delete result2;
  delete world;

  return 1;
}
```

Figure 6 shows the wireframe application graphically. Each block corresponds to a BOOGA component.

[2]Error checking and command line argument parsing is skipped to keep the code as short as possible.

Figure 6: *The structure of the wireframe application.*

Computing the 2D Convex Hull

Computing the convex hull of a finite point set is a common task in the field of Computational Geometry [13]. The following steps are required:

1. Read the point set.
2. Compute the convex hull.
3. Display the resulting polygonal hull shape together with the point set on the screen.

These three steps are handled by separate BOOGA components:

1. A `Parser2D` component is used to read the point set.
2. The process of computing the convex hull has two steps:
   (a) Find all the point objects by executing the `Collector<Point>` component.
   (b) Compute the convex hull for the point set found (`ConvexHull2D` component). The result is visualized by adding line segments to the scene.
3. The `Display` component displays the final result.

This application is depicted in Figure 7. Only one special purpose component (`ConvexHull2D`) had to be developed. All other components are standard components used in many other applications.

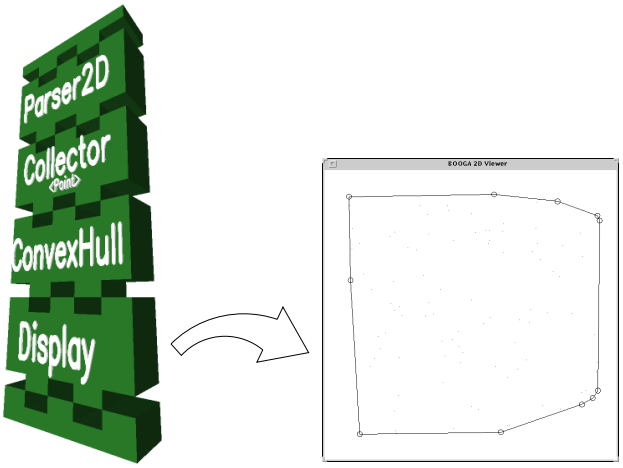


Figure 7: *Computing the convex hull of a planar point set.*

Merge Rendering

The use of components to assemble applications gives great flexibility. The following application combines different rendering engines to achieve faster image generation by splitting a single scene into two separate scenes, and then starts the rendering process for them concurrently. This coarse grained concurrency model may have clear advantages when using a multiprocessor system.

The following components are used:

- The `Parser3D` component is used again to read a scene description.
- The `Split` component is responsible for splitting the scene object into two distinct scenes based on criteria such as *foreground vs. background* objects.
- Two `Renderer` components (e.g. Ray tracer, ZBuffer or Wireframe) are used to render the two scenes. Based on the criterion that has been used to split the original scene, an appropriate renderer may be chosen. As foreground objects need to be displayed very accurately, a ray tracing component would be appropriate, while the background objects could be rendered much faster by executing a z-buffer algorithm.
- In order to obtain a single image, the `Merge` component combines the two results, e.g. based on the depth or alpha channels of generated pixmaps or just by adding the created objects to a new scene object.
- The `Display` component is used to display the result on the screen.

Figure 8 displays the merge renderer application graphically.

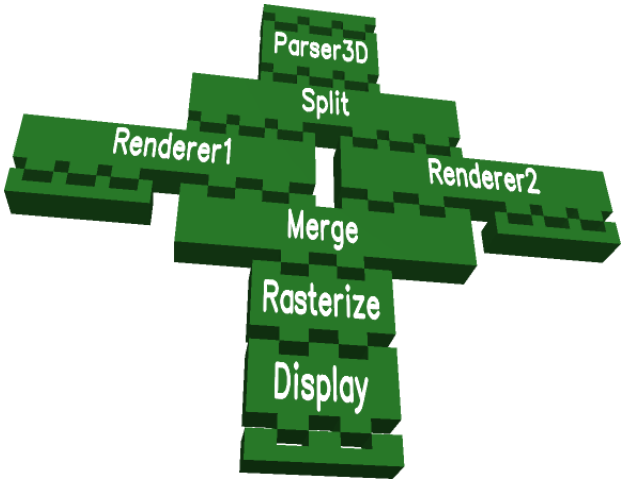


Figure 8: *Combining different renderers to generate the image of a scene.*

7 Conclusions

We have presented a new graphics system designed to deal with most aspects of 2D and 3D graphics. The system is based on a 3-layer architecture consisting of a component, a framework and a library layer built on top of each other. 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 applications very easily. The framework and library layer provide the necessary mechanisms to create the components not already available. However, most applications can be realized by largely reusing existing components.

As soon as a new user realizes the potential of the component approach and has a reliable tool ready to use, he will be highly motivated to spend the additional time to write good, reusable components. Our tool is the BOOGA framework which implements all the necessary protocols and abstractions and provides good default implementations. Since BOOGA completely integrates the component approach, it becomes – after some training by experienced developers – second nature to a new user to reuse and build components.

References

[1] Ekkehard Beier and Uwe Bozetti. A Generic Graphics Kernel and a Customized Derivate. http://www.germany.eu.net/shop/ogefi/dublin.htm, 1996.

[2] Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley, 1992.

[3] James O. Coplien. *Advanced C++, Programming Styles and Idioms*. Addison Wesley, 1992.

[4] J. Gall. *Semantics: How Systems Really Work and How They Fail*. Ann Arbor, MI: The General Systemantics Press, 2nd edition, 1986.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[6] Erich Gamma and André Weinand. Mythos der objektorientierten Programmierung. Seminar IFA, Zürich, 1996.

[7] Ralph E. Johnson. Frameworks. Seminar Zühlke Informatik, Zürich, Mai 1996.

[8] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming (JOOP)*, June/July 1988.

[9] Larry Koved and Wayne L. Wooten. GROOP: An Object-Oriented Toolkit for Animated 3D Graphics. In *OOPLSA '93*, pages 309–325, 1993.

[10] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

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

[12] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[13] Frano P. Preparata and Michael I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[14] Paul S. Strauss and Rikk Carey. An Object-Oriented 3D Graphics Toolkit. In *SIGGRAPH '92*, 1992.

[15] André Weinand. Components: Another End to the Software Crisis? In *Components User Conference (CUC)*, 1996.

[16] P. Wisskirchen. *Object-Oriented Graphics: from GKS and PHIGS to Object-Oriented Systems*. Springer-Verlag, 1990.