

# Graphic Rendering Considered as a Compilation Chain

Benjamin Tissoires<sup>1,2,3</sup> and Stéphane Conversy<sup>2,3</sup>

<sup>1</sup> DGAC / DSNA / DTI / R&D.

7 avenue Ed. Belin, 31055 Toulouse, France

[Benjamin.Tissoires@aviation-civile.gouv.fr](mailto:Benjamin.Tissoires@aviation-civile.gouv.fr)

<sup>2</sup> ENAC, Laboratoire d'Informatique Interactive.

7 avenue Ed. Belin, 31055 Toulouse, France

[stephane.conversy@enac.fr](mailto:stephane.conversy@enac.fr)

<sup>3</sup> IRIT - IHCS, Université Paul Sabatier.

118 route de Narbonne, 31062 Toulouse Cedex 4, France

**Abstract.** Graphical rendering must be fast enough so as to avoid hindering the user perception/action loop. Traditionally, programmers interleave descriptions and optimizations to achieve such performances, thus compromising modularity. In this paper, we consider graphic rendering as a compilation chain: we designed a static and dynamic graphical compiler that enables a designer to clearly separate the description of an interactive scene from its implementation and optimization. In order to express dependencies during run-time, the compiler builds a dataflow that can handle user input and data. We successfully used this approach on both a WIMP application and on a demanding one in terms of computing power: description is completely separated from implementation and optimizations while performances are comparable to manually optimized applications.

**Keywords:** interactive software, computer graphics, compiler, dataflow, modularity.

## 1 Introduction

Interactive systems have to be efficient. In particular, graphical rendering must be fast enough so as to avoid hindering the user perception/action loop. In addition, as any other software, interactive systems have to be modular, in order to maximize maintainability and reliability. The need for modularity is even more important with interactive systems. Making software modular minimizes the cost of modification. As designing good interactive systems requires designers to implement, test, and tweak a large set of alternative solutions iteratively, modular software maximizes the quality. Traditionally, programmers implement graphic rendering in interactive software using an imperative paradigm. They use graphical libraries, and often introduce optimization during the first stages of development so as to maximize performances. This leads to code in which description and optimization are interleaved, which hinders designers' ability to rapidly test new designs. It can even harm safety, as manual

optimization may change the graphical semantics and introduce bugs that are noticeable only with precise situations.

Computer science literature contains solutions for these kinds of problem. Researchers have designed compilers, i.e. systems that transform a high-level language to a low-level one. They enable programmers to focus on description, while leaving low-level optimization to the compiler. In order to address the problems encountered by interactive systems programmers, we introduce in this paper a new approach to graphical rendering implementation. We consider the transformation from input devices and data to graphics as a compilation chain. We design a static and dynamic graphical compiler: it enables a designer to clearly separate the description of an interactive scene from its implementation and optimization.

We first describe three scenarios illustrating how today's designers implement graphical rendering and cope with description, efficiency and modularity. Based on these examples, we explain why graphical rendering implementation can be considered as a compilation chain. We describe the principles of the graphical compiler, and report on the results we obtained with two examples.

## 2 User Interface Development Scenarios

In this section, we present three scenarios concerning the development of user interfaces. These scenarios are the basis of our reflexion.

### Using Graphic Toolkits

Since the rise of the WIMP (Window Icon Menu Pointer) paradigm, most programmers use User Interface toolkits, such as Motif or Qt. UI Toolkits allow programmers to rapidly construct an interface by juxtaposing widgets, i.e. independent units of graphics and behaviour, on the interface. However, the widget model is not suitable for the implementation of post-WIMP interactions. WIMP interfaces implicitly use a model where widgets are juxtaposed, and they can not be used in scene where graphics lay on top of each other. For example, programmers can not use widgets to implement a radar image that contains flight elements on top of sectors. In addition, programmers do not have access to the inner mechanisms of the toolkit. Hiding implementation details eases use and prevents misuse, but it also prevents some of the optimizations that may speed up the rendering process [15] [10]. There exists a few post-WIMP toolkits [3], but they are internally optimized for a specific part of the rendering process (e.g. culling small or out-of-screen ZUI items).

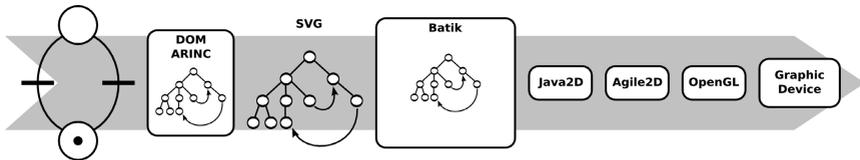


Fig. 1. The chain used in the model of ARINC 661 by [2]

## Model-Based Approach

Conversy et al. in [8] present a model-based approach to separate behaviour from rendering. The idea is to describe the behaviour of the application with Petri Nets together with a conceptual model of the interactive elements, and the rendering with an SVG scene (Scalable Vector Graphics [22]). When user input occurs, the Petri Net modifies the conceptual model, which in turn is transformed into a new SVG scene through an XSLT stylesheet (Extensible Stylesheet Language Family Transformations [23]). The SVG scene is then redrawn on the screen (Fig. 1). This model-based approach allows the designer to clearly separate descriptions of appearance and behaviour (look and feel), to use models based on formalism, and to use SVG, which is an exchange format between coders and graphic designers ([7]). However, the execution process of this chain is costly in terms of performance: each time a change occurs; the whole transformation chain is triggered, and slows down the system. Moreover, the system is based on completely separated stages: each intermediate data structure is completely rebuilt, and does not benefit from invariant behaviour of the front stages. Since there can be seven stages between the Petri Nets and the final pixels, performances are extremely low. Thus, the system is completely modular, but not reactive enough to be used in real-time.

## Working With the Graphic Device to Optimize Performance

One solution to render fast interactive applications is to work at a low level of programming, with the help of libraries close to the hardware, such as OpenGL (Open Graphic Library<sup>1</sup>). At this level, programmers can use optimizations that mainly consist in caching a maximum amount of data or commands on the graphic device. For instance, the programmer can use display lists - a record of a list of OpenGL commands that can be called at once - or memoization of a computed image into a texture.

However, working at such a low level forces the programmer to interleave description of the graphical scene and optimizations. Moreover such optimizations need to be known by the programmer and programmed by hand, and influence his way of writing the application at the cost of readability. These optimizations speed up the whole application but as they are too tightly linked with the rest of the code, it is hard to change either the description or the optimization.

## Discussion

These three scenarios show that with the available tools and methods, a programmer has to do the job of a compiler to build non-standard modular and efficient user interfaces. He has to allocate registers (OpenGL texture or display list), to manage caches of data (render into textures), and to reorganize his optimizations in order to have the fastest code possible. He can even implement parts of a Just In Time compiler (JIT), by designing optimizations triggered at the run-time (such as display lists).

---

<sup>1</sup> OpenGL, The Industry's Foundation for High Performance Graphics: <http://www.opengl.org>

### 3 Graphical Rendering = Compilation Chain

In this section, we explain why the graphical rendering process can be considered as a compilation chain. Then we define the notion of a graphical compiler (GC<sup>2</sup>) and of intermediate graphical languages.

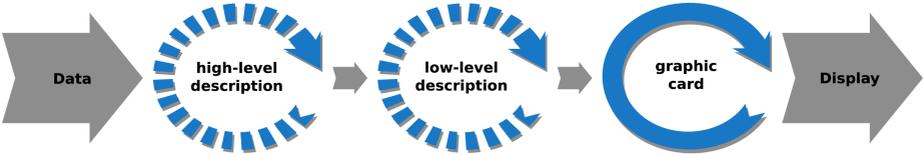


Fig. 2. The “classical” rendering process

#### Why it Is a Compiler Problem

Writing an interactive scene needs several steps to produce the final application. Fig. 2 shows what most programmers do: before trying to display something, some data are needed; then, these data are transformed into a high-level description; if the rendering process needs it, this high-level language is usually displayed and a loop analyzes this language in order to apply changes that occur between two frames (this is the case of scenario number 2). When high performances are needed, the programmer converts by hand the high-level description into a lower-level one, which in turn is rendered to the screen (scenario number 3). This requires the programmer to implement a scheme in which the programmer has to take care of the synchronization between a high-level API and a lower level one. This figure also shows the different refresh loops that are used in graphic rendering. The solid loop to the right symbolizes the video controller that scans the video memory at each refresh of the screen. The two dotted loops symbolize the fact that the loop can be either on the high level, or on the low level. Thus, in scenario number 2 (the model-based one), the loop is placed on the high-level description, and in the scenario number 3 (the OpenGL one) the loop stands on the low-level description.

Hence, writing an interactive interface consists in a chain of transformations, which can be handled by a compiler:

A compiler is a computer program, or set of programs, that translates text written in a computer language - the *source* language - into another computer language - the *target* language. [1]

In the problem of rendering graphical scene, the data can be considered as an input language and the drawing commands as the target language (Fig. 2). In order to explain the structure of the GC (Fig. 4), we will compare it to the structure of the Java programming environment (Fig. 3). The graphical compiler chain consists in the

---

<sup>2</sup> In Computer Science, GC traditionally stands for Garbage Collector, but in the rest of the article, we will abbreviate graphical compiler by GC.

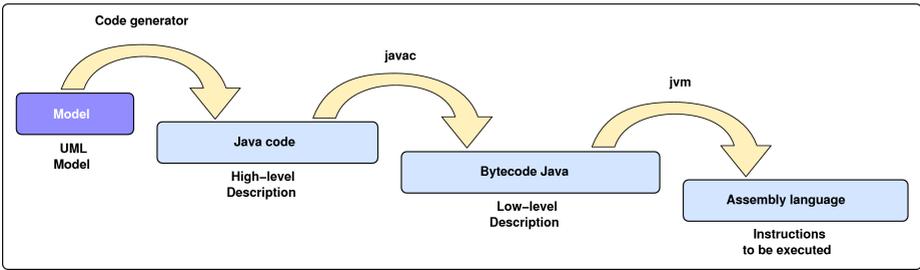


Fig. 3. The compilation chain used in Java when starting from UML...

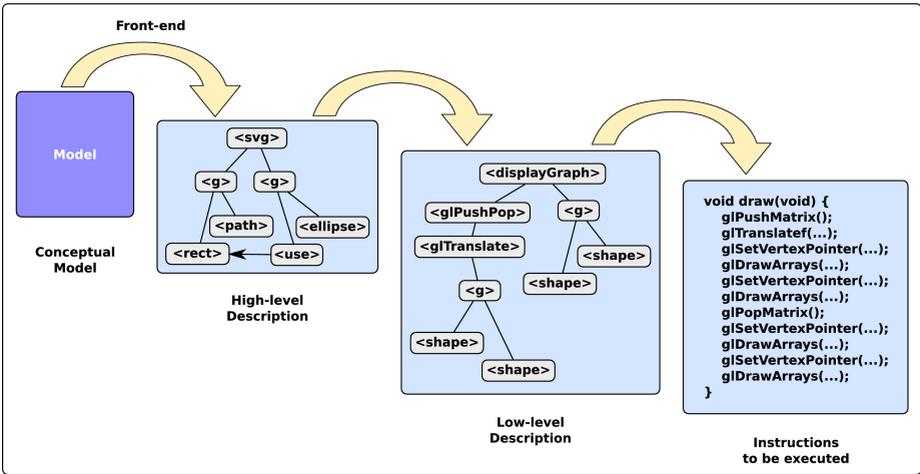


Fig. 4. ...and its equivalent when rendering applications

different transformations between languages. The high-level description of the graphical scene - through an SVG-like syntax - is equivalent to Java code written by the programmer. The low-level description which is strongly linked with the hardware we used at the end (abstracted with OpenGL) is the equivalent of the bytecode produced by the Javac compiler. At the end of the chain, a backend either interprets (JVM) or generates (a native Javac compiler) the instructions that are executed on the hardware.

In addition, the GC includes another front-end, the conceptual model and the rules needed to transform it into SVG. This stage is equivalent to recent environments that generates Java code from UML description. We will see that it allows the GC to handle in a uniform way all the transformations, so that optimizations are applied in the whole program.

By considering the process of rendering graphical scene as a compiler chain, we expect the following benefits: this architecture makes it possible to separate the description of the graphics and the optimizations; concepts such as optimizations that have been well-studied in the compiler problem can be transposed to the problem of

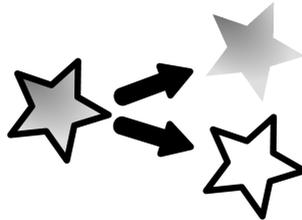
rendering graphics; the high-level description can be abstracted enough to be independent of the final renderer used; the semantics of the transformations used will be clear enough to be able to check rendering.

## Transformations and Languages

**The Conceptual Model.** The first language of the graphical compilation chain is an abstraction of the data. It allows the programmer to separate the presentation and the other parts of the application, i.e. the interaction part and the dialog controller. This part contains elements such as the value used to describe a model of a slider in a WIMP application, or the string of characters of a text field [8].

Once the conceptual model of the elements to be drawn is available, the next step is to transform it in terms of graphical shapes. As said before, we extend the standard model of a compiler by adding a stage on the front: the conceptual model. However, as the GC does not know this specific language used by the programmer, the latter has to give to the GC both the front-end language and the transformation rules to convert his specific language into the high-level language of the GC.

**The High-level Description of Graphics.** This description contains a subset of SVG elements such as *rectangles*, *ellipses*, *path*, *groups*, etc. The scene is described with a graph, with nodes containing geometrical and style transforms. SVG was designed with two purposes: it is a high level language, i.e. it makes it possible to describe complex scenes with a short description; it is also an exchange format between applications and designers. Another advantage of using a SVG-like language is that its structure (a graph) is highly adapted to an implementation in OpenGL.



**Fig. 5.** A shape with a fill and a stroke can be divided into two elementary shapes

Before the low-level description, the GC inserts another stage which consists in converting every shape into a path and the direct cyclic graph into a tree. Thus, a shape made up of a fill and a stroke is divided into two elementary shapes with the same semantic as a group of shapes (Fig. 5). This reduces the language to a kernel, i.e. the minimum set of primitives needed to express the semantics of SVG. It thus minimizes the complexity of the subsequent transformations. Such stages are also included in most standard compilers: they convert the input source into an intermediate representation. This step also allows the compiler to produce an optimized code.

**The Low-level Description.** The GC converts high-level primitives into primitives suitable for the hardware: the low-level description. The previous language is thus converted into a tree containing the instructions needed to render the scene: the display graph. As the current renderer used is OpenGL, this part contains the instructions such as *glPushMatrix*, *glTranslate* or the instructions needed to tessellate and render a path.

## 4 Expressing Dependencies with a Dataflow

The static compiler produces the equivalent of a “binary” program written in the low-level description. Executing the program consists in interpreting the display graph at “run-time”. However, the dynamic compiler executed at run-time needs to know the dependencies of the different variables. We chose to express the dependencies with a dataflow. The GC statically compiles this dataflow. The dynamic graphical compiler does not need to recompile the scene when a change occurs between two frames. For example, if the change consists in the modification of the position of an element, the produced code is the same, except the part concerning the changed variables (Fig. 6).

The programmer needs to specify which variables are input so as to help the compiler to know which parts will change during run-time, and to optimize the produced code. The GC caches all the static data during the static compilation.

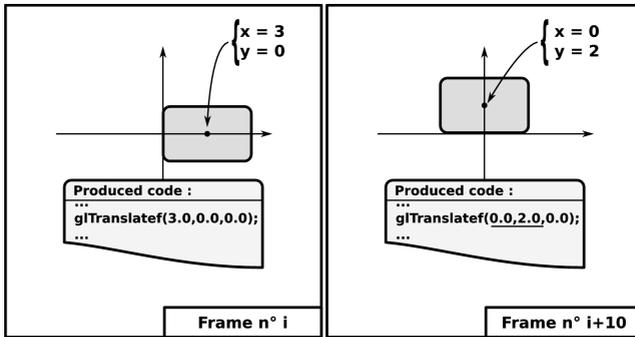


Fig. 6. Changes in the produced code when moving one object

## Implementation

**Language.** The language used for the dataflow is a mathematical one. The designer specifies it by expressing formulas. Our compiler overloads operators in Python<sup>3</sup> to build the parser. For instance, we can write:

```
x0 = var('x0', 5)
y0 = var('y0', 10)
x1 = var('x1', x0+200)
y1 = var('y1', y0+250)
```

<sup>3</sup> Python Programming Language - Official Website: <http://www.python.org>

This code builds two inputs  $x_0$  and  $y_0$  and two dependent variables  $x_1$  and  $y_1$ . Building and naming variables allow further references in the description of the scene. For instance,  $x_0$  and  $y_0$  may be the anchor of a shape and  $x_1$  and  $y_1$  the anchor of another shape that has to be moved (200,250) relative to the first one:

```
rect0 = rect(-5, -5, w=10, h=10, fill=(1.0,0.0,0.0),
transform=transform(x0,y0))

rect1 = rect(-5, -5, w=10, h=100, fill=(1.0,0.0,0.0),
transform=transform(x1,y1))
```

**Execution.** Dataflow can have two modes of execution. The first one is interpretation and the second one is compilation. Interpretation is very useful when one wants to debug and test one's design. It allows new variables and formulas to be created at run-time. The counterpart of this flexibility is that it is very costly when it comes to execution, as it requires a tree traversal and the interpretation of each node each time a value has to be computed.

The second possibility, when formulas do not change often at run-time, is to use compilation. The GC implement dataflow compilation by attaching to each variable the function that contains the formula. The execution speeds up but this scheme forces the programmer to do a static compilation of the application.

In the previous example, the GC transforms the declarative description into a list of OpenGL commands. The list of commands contains the two following lines:

```
glTranslate3f(5.0f, 10.0f, 0.0f);
(...)
glTranslate3f(205.0f, 260.0f, 0.0f);
```

The GC remembers the dependences between the input variable (' $x_0$ ' for example) and the produced memory case ('5.0f' here). At run-time, when a change occurs, the executive part of the GC propagates directly the modification towards the memory that is used to render the scene. Such principle avoids the tests needed to know whether a variable has changed.

## Optimizations

As dataflow is a mathematical language and also a functional one, we can apply two types of optimizations. Optimizations can be relevant to the semantic of the functions themselves. For example, writing ' $x+x+x+x+x$ ' can be transformed into ' $6*x$ ', thus reducing the number of operations from five to one (if there is no cache implemented, the access to a variable is costly and the overall cost is then reduced). The GC can also find optimizations more relevant to the implementation, as in all languages, so as to accelerate the time spend inside the propagation of the data.

## 5 Implementation and Optimizations of the Graphical Compiler

### Implementation

We wrote the compiler in the Python language as it allows quick development. Nevertheless, in order to achieve good performances with OpenGL, we wrote the run-time of the graphics in a C module of Python.

The production of the low-level description of the scene follows standard transformation rules. For each element in the graph, the GC produces the corresponding elements. Optimizations are made during the productions by testing whether we should add decorators or not, as seen before.

After the production of the low-level description, the renderer can be executed asynchronously. We designed our toolkit to be asynchronous so as not to penalize all the parts of the process if one is slow. The toolkit uses threads and buffers to implement this mechanism. The list of calls from Python to the run-time uses a strategy similar to the OpenGL double-buffering mechanism. There are two lists available: the first is the one which is executed, and is protected from any changes except local changes coming from the dataflow. The second one allows the compiler to allocate and free the memory needed and is allowed to be modified by other processes.

## Optimizations

The low-level description is what we call a display graph, an abstract tree that represents the graphical code that will be executed eventually.

**Static Optimizations.** We have written our low-level language with the help of design patterns. The help of the design pattern decorator allows the GC to construct the tree so as to avoid tests while walking through it. For example, if the element does not contain any *scale* transformations, the compiler simply does not include the decorator *scale* over the element. The produced tree contains the minimum elements needed to render the scene.

The second possibility offered by this approach is that the compiler can factorize elements by detecting common subexpressions. For instance, if the same transformations occur between two groups, it can factorize them into a bigger group containing the common transformation.

**Dynamic Optimizations.** Working with a tree allows the GC to make optimizations during run-time, to implement a Just In Time compiler (JIT). Nevertheless, walking through the tree has a significant cost in term of instructions to be executed. The time spent to evaluate the display graph, plus the time needed to transform it into graphic call, plus the time of execution has to be inferior to a minimum refresh-time rate (maximum 0.04 seconds per frame to achieve 25 frames per second). To achieve such performance, the run-time transforms this tree into a list of OpenGL calls. This transformation allows caching of operations that have to be executed. It also puts in cache all the tests that need to be done. For instance, the programmer can activate or deactivate a part of the tree through a variable. The produced code is empty if the condition is set to false. By transforming the tree into a list of really executed code, the run-time of the GC avoids a re-evaluation of this test. If the condition changes the run-time re-parses the tree in order to execute the right code. This optimization is known as *dead-code elimination*.

When a change in the inputs that occurs does not imply a rebuilding of the list of OpenGL calls, the dataflow propagates the change by modifying the previously produced code.

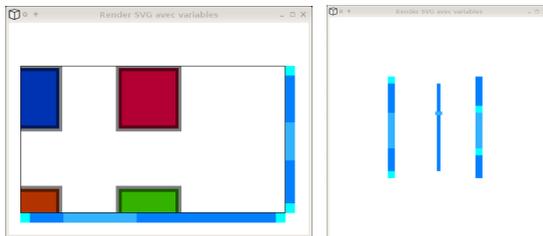
**Other Optimizations.** As we have seen, a graphical compiler can make optimizations over the display graph. The GC can produce both local optimizations and cross-procedural optimizations as it knows the entire display graph. Because of the lack of room, we list other techniques that are available in the GC to speed up the rendering in the light of compiling techniques:

- *Common subexpressions*: the optimizer can detect such graphical common subexpressions and factorize them.
- *Propagation of the constants* corresponds in the graphic field to the operation of caching a maximum amount of data, most of the time on the graphic device.
- *Programmer's hints*: the programmer can specify that a non-trivial or non-detectable optimization concerning his own problem (this optimization corresponds to *aliasing* or the keyword *register* in the C language).
- *Other JIT optimizations*: a Just In Time compiler (JIT) can handle other optimizations that the static compiler can not discover.

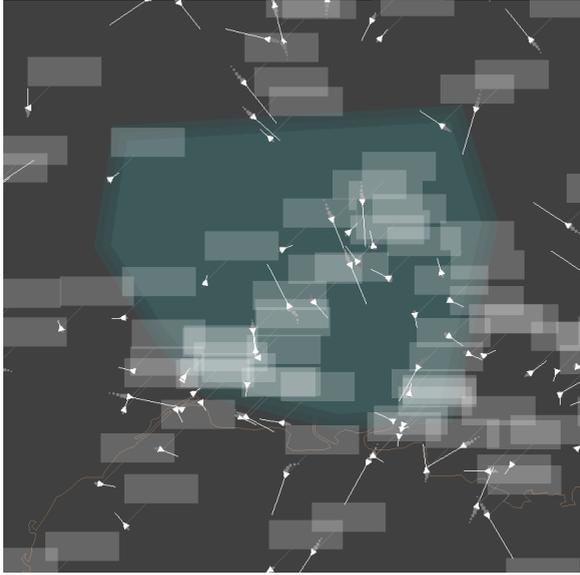
## 6 Results

We assessed the approach by writing two different applications with the GC. The first consists in a demonstration of the use of standard widgets to build a WIMP interface (Fig. 7). This application illustrates scenario number 2. The programmer specifies the conceptual model by specifying the abstraction of the different elements, and then gives to the system the transformations needed to compile the elements to SVG. The GC statically compiles the dependencies and produces the final application. The resulting program contains no interpretation of SVG constructs, as much as a binary does not contain C constructs. As such, it is closed to the minimum program needed to implement this system in the C language with OpenGL.

The second one, a radar view displaying planes (Fig. 8), is demanding in terms of computing power. This application has to display up to 500 planes, each of them made up of 10 elementary shapes. In fact, this proof of concept can display more than 10000 triangles and handle user events with a very low system load: the framerate is up to 500 frames per second. A previous version with a run-time in the Python language with a JIT enabled only reached 140 frames per seconds. The same code without the dynamic compiler and the programmer hints achieved around 4 frames per seconds.



**Fig. 7.** Example of a WIMP application rendered with the help of our GC

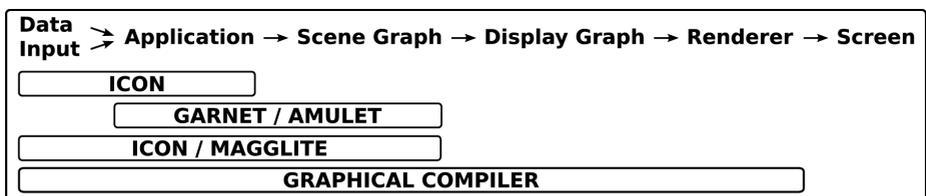


**Fig. 8.** Example of a radar display application rendered with the help of our GC

The GC weighs in 4000 lines of code in Python and the run-time in C is 4000 lines. Applications using the GC are small: the radar view application is made up of only 500 lines and consists only in the description, as expected. Though more feature complete, a previous radar application written in C++ and OpenGL weighs in 85 000 lines.

## 7 Related Work

The use of transformations starting from a high-level description has been studied in the Indigo Project [4]. Contrary to the X11 server, both rendering and interaction are in charge of the *Servir*, the server of the Indigo architecture. This idea of transformations was then extended with the implementation of the set of widgets ARINC 661 [2] and later by the MDPC model [8].



**Fig. 9.** Dataflow span in different toolkits

Many researchers used dataflow language to describe interactive applications (Fig. 9). Some of these dataflows handle data from the inputs to the application (InputConfigurator [11] or Magglite [14]) while others express graphical constraints (Garnet/Amulet [18], [20]). In the GC, the dataflow can manage the transformations from the data and the inputs down to the screen. In [19], the researchers present a way to reduce the storage of the dataflow, which can be a problem in large applications.

The notion of compilation in graphics was introduced by Nitrous, a compiler generator for interactive graphics [12]. However, as in [17], the compiler is only pixel-based, and does not handle the inputs coming from physical devices or from the application. LLVM [16], with its OpenGL stack developed by Apple, can efficiently abstract the description of the interface from the hardware. The JIT included in LLVM can optimize the different shaders available in order to have the most efficient implementation.

Finally, dynamic compilation has been studied with languages such as Smalltalk [9], Self [13], or Java. LLVM can also be executed with a JIT and can do interprocedural optimizations [5].

## 8 Conclusion

In this paper, we have proposed a new approach to graphical rendering, in order to make it both modular and efficient. We show that an interactive application is a list of transformations of intermediate graphical languages, which can be considered as a compilation process. We described how a graphical compiler can help designers and programmers to implement efficient rendering code. The programmer writes a front-end of a language describing the objects to be interacted on, and a transformation function to a high-level graphical API. The graphical compiler can then generate low-level code that implements the application. During the different transformations, the GC detects and applies optimizations in order to generate efficient code. Thanks to the dataflow which is produced at compile time, the dynamic compiler avoids unnecessary recompilation at run-time. The latter can take time to optimize the produced code on the fly.

The architecture we have presented has some limitations. It can not handle dynamic changes of the structure of the conceptual model. With the radar view, flights are filtered out when they are not visible, and the conceptual model elements are recycled for new flights. However, implementing a vector graphic editor is not possible with such description, because it is not possible to know in advance the number of shapes.

Furthermore, the graphical compiler does not handle UI control. Dataflows can simulate control with tests, but a more general approach is needed, such as state machines switching dataflow configurations [6].

However, we showed with two examples that the graphical compilation approach is suitable for a range of applications: static ones, such as WIMP interfaces now found in cockpits, or semi-dynamic, data-bounded ones, such as radar view. Future work includes finding a common language to describe intermediate languages and transformations. This approach leads to verifiable semantics of transforms and languages. We plan to enhance the compiler so as to produce verified code, and make critical systems safer.

## References

1. Aho, V., R., S., Ullman, J.D., Ullman, J.: *Compilers: Principles, Techniques, and Tools*. Morgan Addison-Wesley, Boston (1986)
2. Barboni, E., Conversy, S., Navarre, D., Palanque, P.: Model-based engineering of widgets, user applications and servers compliant with arinc 661 specification. In: Doherty, G., Blandford, A. (eds.) *DSVIS 2006*. LNCS, vol. 4323, pp. 25–38. Springer, Heidelberg (2007)
3. Bederson, B.B., Grosjean, J., Meyer, J.: Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering* 30(8), 535–546 (2004)
4. Blanch, R., Beaudouin-Lafon, M., Conversy, S., Jestin, Y., Baudel, T., Zhao, Y.P.: Indigo : une architecture pour la conception d'applications graphiques interactives distribuées. In: 17th conference on Conférence Francophone sur l'Interaction Homme-Machine, pp. 139–146. ACM Press, New York (2005)
5. Burke, M., Torczon, L.: Interprocedural optimization: eliminating unnecessary recompilation. *ACM Trans. Program. Lang. Syst.* 15, 367–399 (1993)
6. Chatty, S.: Defining the behaviour of animated interfaces. In: *IFIP TC2/WG2*, pp. 95–111. North-Holland Publishing Co, Amsterdam (1992)
7. Chatty, S., Sire, S., Vinot, J.-L., Lecoanet, P., Lemort, A., Mertz, C.: Revisiting visual interface programming: creating GUI tools for designers and programmers. In: 17th annual ACM symposium on User interface software and technology, pp. 267–276. ACM Press, New York (2004)
8. Conversy, S., Barboni, E., Navarre, D., Palanque, P.: Improving modularity of interactive software with the MDPC architecture. In: *EIS (Engineering Interactive Systems) conference 2007, joint HCSE 2007, EHCI 2007 and DSVIS 2007 conferences*. LNCS. Springer, Heidelberg (2008)
9. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the smalltalk-80 system. In: 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 297–302. ACM Press, New York (1984)
10. Dourish, P.: Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications. *ACM Trans. Comput.-Hum. Interact.* 5, 109–155 (1998)
11. Dragicevic, P., Fekete, J.-D.: The input configurator toolkit: towards high input adaptability in interactive applications. In: *AVI 2004: working conference on Advanced visual interfaces*, pp. 244–247. ACM Press, New York (2004)
12. Draves, S.: Compiler Generation for Interactive Graphics using Intermediate Code. In: Danvy, O., Thiemann, P., Glück, R. (eds.) *Dagstuhl Seminar 1996*. LNCS, vol. 1110, pp. 95–114. Springer, Heidelberg (1996)
13. Hölzle, U., Ungar, D.: A third-generation self implementation: reconciling responsiveness with performance. In: 9th annual conference on Object-oriented programming systems, language, and applications, pp. 229–243. ACM Press, New York (1994)
14. Huot, S., Dumas, C., Dragicevic, P., Fekete, J.-D., Hégron, G.: The magglite post-wimp toolkit: draw it, connect it and run it. In: 17th annual ACM symposium on User interface software and technology, pp. 257–266. ACM Press, New York (2004)
15. Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G.: Open implementation design guidelines. In: 19th international Conference on Software Engineering, pp. 481–490. ACM Press, New York (1997)
16. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: *IEEE international symposium on Code generation and optimization*, pp. 75–86. IEEE Press, New York (2004)

17. Peercy, M.S., Olano, M., Airey, J., Ungar, P.J.: Interactive multi-pass programmable shading. In: 27th Annual Conference on Computer Graphics and interactive Techniques International Conference on Computer Graphics and Interactive Techniques, pp. 425–432. ACM Press, New York (2000)
18. Zanden, B.T.V., Halterman, R., Myers, B.A., McDaniel, R., Miller, R., Szekely, P., Giuse, D.A., Kosbie, D.: Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Trans. Program. Lang. Syst.* 23, 776–796 (1994)
19. Zanden, B.T.V., Myers, B.A., Giuse, D.A., Szekely, P.: Integrating pointer variables into one-way constraint models. *ACM Trans. Comput.-Hum. Interact.* 1, 161–213 (1994)
20. Zanden, B.T.V., Halterman, R.: Using model dataflow graphs to reduce the storage requirements of constraints. *ACM Trans. Comput.-Hum. Interact.* 8, 223–265 (2001)
21. ARINC Specification 661-3 Cockpit Display System Interfaces to User Systems, Aeronautical Radio Inc. (2007)
22. Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation (2003), <http://www.w3.org/TR/SVG/>
23. XSL Transformations (XSLT) Version 1.0. W3C Recommendation (1999), <http://www.w3.org/TR/xslt>