Hayaku: Designing and Optimizing Finely Tuned and Portable Interactive Graphics with a Graphical Compiler

Benjamin Tissoires [1,2]
[1] DSNA DTI R&D
7, avenue Edouard Belin
31055, Toulouse,
France
tissoire@cena.fr

Stéphane Conversy [2]
[2] Université de Toulouse, ENAC, IRIT
7, avenue Edouard Belin
31055, Toulouse,
France
stephane.conversy@enac.fr

ABSTRACT

Although reactive and graphically rich interfaces are now mainstream, their development is still a notoriously difficult task. This paper presents Hayaku, a toolset that supports designing finely tuned interactive graphics. With Hayaku, a designer can abstract graphics in a class, describe the connections between input and graphics through this class, and compile it into runnable code with a graphical compile chain. The benefits of this approach are multiple. First, the front-end of the compiler is a rich standard graphical language that designers can use with existing drawing tools. Second, manipulating a data flow and abstracting the low-level run-time through a front-end language makes the transformation from data to graphics easier for designers. Third, the graphical interaction code can be ported to other platforms with minimal changes, while benefiting from optimizations provided by the graphical compiler.

Author Keywords

Human-Computer interfaces, User Interface Design, Methods and Applications, Optimization

ACM Classification Keywords

H.5.2 User Interfaces: GUI.

General Terms

Design, Languages

INTRODUCTION

Interactive graphics development is a notoriously difficult task [18, 19]. In particular, rich interactive systems design requires finely-tuned interactive graphics [13], which consists of a mix of graphical design, animation design and interaction design. Subtle graphics, animations and feedback enhance both user performance and pleasure when interacting [16]. The success of the iPhone demonstrates it: finely tuned widgets, reactive behavior, and rich graphics together make the iPhone interface superior to other products. Designing such systems is a recent activity that has rarely been supported explicitly in the past. Yet, their quality is essential for usability. Unfortunately, developing such software is not reachable by all stakeholders of interactive system design. This requires highly trained specialists, especially when it comes to using very specific graphic concepts *and* optimize the rendering and interactive code. Hence, there is a clear need for making interactive graphical programming more usable.

Moreover, even within a given style of computing (either web or mobile), new means of thinking, designing and developing interfaces arise every couple of years. For example, we successively saw the rise of Java2D, Adobe Flash, Adobe Flex, Microsoft dot net, XAML, SVG, WMF, Web 2.0 interfaces programmed in javascript in the browser (with the Canvas and HTML5), OpenGL etc.[1]. In order to design and develop interactive systems on those platforms, interface designers have access to a plethora of toolkits, usually incompatible with one another. This results in the failure of *reusability*, one of the most praised property in computing: designers have to redevelop existing software in order to port it to another platform, with the associated drawback of not reusing well crafted and tested software. For example, the menu subsystems that have reached a good level of usability in traditional desktop platforms (i.e. Windows or MacOSX) are poorly imitated in Web 2.0 interfaces, where the user is for instance required to follow a tunnel strictly when navigating in a hierarchical menu. Hence, there is a need for the ability to reuse existing software, especially if we assume that new platforms will keep appearing in the future (see WebGL for example).

This paper addresses the two requirements presented above: design usability and reusability of finely-tuned interactive gra-phics. In particular, we introduce Hayaku, a toolset that targets interactive graphics that Brad Myers refers to as the “insides” of the application [21], and that no widget toolkit can support. After a review of related works, we present the exact audience that we target, and the requirements of such an audience. We then present the toolset using three use cases, and some of the internal mechanisms that implement

[1] . . . , Cairo, Qt, Prefuse, Protovis, iPhone SDK, Open Handset Alliance’s Android, Palm WebOS to name a few more

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EICS’11, June 13–16, 2011, Pisa, Italy.
Copyright 2011 ACM 978-1-4503-0670-6/11/06...$10.00.

its features. We finally provide a number of elements to evaluate the toolset according to our claims. Related Work This work is related to two topics in the user interaction software and technology community: methods to design interactive systems, and graphical toolkits.

Interactive System Design Methods

Chatty et al. [6] present a method and associated tools to involve graphic designers in interactive system design and development. Programmers and graphic designers first agree on a conceptual and simple SVG skeleton of the scene. While programmers code the interaction with a low quality representation, graphic designers can work on their design in parallel. Since programmers and designers respect a contract, the production of the final system consists in the replacement of the low quality representation by the designer's one. However, the tools oblige the graphic designer to use a library to transform the high-level language (SVG) to a lower-level one (a Tk-like canvas) with a lesser expressive power. This hinders exploration of alternative design since changing gra-phics implies many manipulations to reflect the change in the final application. Furthermore, when optimizing code, the approach falls back to a sequential process: programmers have to wait for designers' solutions before optimizing by hand the rendering code, and designers have to wait for optimizations to assess if their design is usable.

Microsoft Expression Blend makes heavy use of XAML to describe the graphical parts of the application. Like Intuikit, the aim is to separate the graphical description from the functional core of the application. The designer can produce one design per C# class that has to be drawn, but he still needs to manipulate the low level code in order to implement interactions and animations. The concept of "binding" allows programmers to link the graphical shapes and the source objects. The Adobe Flex and Flash suite also provides a means to separate the graphical description from the functional core. However, even if the designer can rely on Flash to build her graphical components, she has to develop the rest of the application using the ActionScript language. Furthermore, there is no abstraction of the graphics, nor a way to express properties with a data-flow. Finally, even if Flex runs on a variety of platforms in its own window, it is not possible to embed the graphics among the graphical scene of another application.

Toolkits

Many toolkits address the problem of performance: Prefuse [14], Jazz [4], Piccolo [3], and Infoviz Toolkit (IVTK) [9] for instance. Performance is maximized by using specialized data structures explicitly (tables for Prefuse and IVTK), or hidden data structures (spatial tree for Piccolo). The first limitation of this approach is that the language used to describe graphics is both inappropriate and not rich enough: describing graphics in Java code with SwingStates [1] is verbose, Java concepts do not match graphics exactly, and rich graphics created with tools for graphic design cannot be directly used in the toolkit. Rich graphics toolkits exist, such as Batik[2], but they are not efficient performance-wise. Furthermore, the problem with these toolkits is that even if they are efficient, they force the toolkit user to work with a specific language and a specific run-time. For instance, users of toolkits can not use Prefuse to write a C or C++ application.

Other works use compiler-like optimizations to produce efficient graphical code (Java3D, LLVM [15] with Gallium3D[3] in Mesa). However these tools are only accessible to low level graphical programmers that manage to write code for the graphic card directly. They are not supposed to be used by the average interactive application developer, with basic understanding of the factors that accelerate rendering.

The solution we propose here consists in helping the production of efficient code for heavy graphics handling. In order to compile the graphical part, we rely on a dataflow, and a mechanism that is able to track the dependencies between input data and graphical elements. Dataflow has been used in graphical interactive toolkits (Icon [8] for the input, and Garnet [25] for the constraints), and have been showed to help building interactive systems efficiently. However, the main difficulty with such a system is to make it fast for both graphical rendering *and* the dependency updating mechanism.

This point is addressed in [24], which introduces a compile chain for interactive graphical software. This work shows that using a graphical compiler (GrC) together with a dataflow leads to good performance. However, the tools were more a proof of concept than a real toolset: the authors present a way to implement optimizations, but do not detail how programmers of the graphical interface can connect all parts together. Another problem of the GrC is that it generates a program that is linked to the runtime of the GrC. This forces the designer to describe all graphical parts of the application with the GrC. However, when dealing with high performance applications, there are parts of the code that the programmer still wants to write manually, in order to maximize the performances. Meanwhile, this programmer may not want to write every piece of the software, if only because they already exist.

The work we present in this paper improves on the concepts described in [24]. In particular, we show how our new tool can help the programmer and the graphic designer to use graphical compiling in a simpler manner, and what benefits can be gained from the new functionalities. We also improve it by allowing it to be modular and thus producing embeddable components. Finally this modularity allows us to turn this proof of concept into a real compiler that can handle multiple graphics back ends and run-time modules.

TARGET AUDIENCE AND REQUIREMENTS

The work presented here targets members of interactive system design teams. A large body of work aims at supporting interactive system design. For example, Participatory Design (PD) partly aims at facilitating production and communication between all designers, be they user-experience specialists, graphic designers, users, programmers [17]. PD employs multiple means to elicit design and communicate it

[2] http://xmlgraphics.apache.org/batik/

[3] http://wiki.freedesktop.org/wiki/Software/gallium

efficiently in groups where people do not share the same culture. Use cases in the form of stories, drawings and mock-up [5], paper prototypes [23]: all tools aim at maximizing expression, exploration by iteration and understanding by culturally different designers.

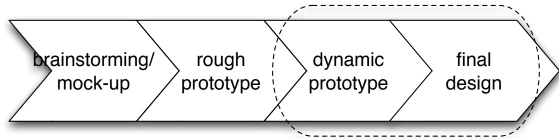

Figure 1. Targeted activity

After these tools have led to initial static prototypes, the designers have to work on dynamic, graphical interactive prototypes [2] [6] (figure 1). As said in the introduction, part of the work is the design of finely-tuned interactive graphics, which consists in a mix of graphical design, animation design and interaction design. The quality of the artifacts designed during this stage in the process is essential for usability. The overall user experience of interacting depends on how well all features (be they graphical, animation, behavior) mix together: the designer must address all concerns at the same time, and dispatching the task between a graphical designer and a programmer does not work anymore. Hence, this activity requires designers with skills in graphic design, animation, interaction design and programming. Our work especially targets this kind of designers.

As demonstrated by Artistic Resizing [7], we think that technical support has a great influence on the experience of designers engaged in the activity. A recent survey analyzes how designers design and program interactive behaviors with current tools [19]. Among the findings, the designers expresses that "the behavior they wanted were quite complex and diverse [...] and therefore requires full programming capabilities"; that "the design of interactive behaviors emerge through the process of exploration [...] and that today's tool make it difficult to iterate on behavior or revert to old versions"; "Details are important, and you never have them all until full implementation"; "I can represent very exactly the desired appearance. However, I can only approximate the backend behaviors"; and they want to do "Complex transitions / animations."

Based on these concerns, we propose a set of requirements for our tools. Similarly to paper prototypes in PD, tools should *maximize expression, exploration, and communication* between designers. Maximizing expression requires rich graphics, hence a toolset should be able to handle *heavy graphical* scenes, with lots of *subtle* graphical properties. Designing such scenes requires efficient design tools, such as vector graphics editors. However, in order to be usable in interactive system, the toolset should *deliver enough performance*. Maximizing exploration implies a system in which changing things (e.g. a graphical property) should be as inexpensive as possible, i.e. with as *little manipulation as possible* required to reflect the change in the subsystems.

TOOLKIT DESIGN AND CONCEPTUAL MODEL

Designing a system that addresses all the requirements above is beyond the scope of this paper. In this work, we describe Hayaku, a tool set that partly addresses these requirements. In particular, we address *richness of expression*, *exploration*, *performances*, and *reusability*. Hayaku mainly focuses on the rendering part of the application. It also provides hooks to implement interaction with the user and communication with the rest of the application. The functional part of the application (what happens in the system when the button is pressed for instance) is out of the scope of the paper.

General Idea: the Interaction Designer is in Charge

As said before, this activity requires designers with skills in graphic design, animation, interaction design and programming. Omniscient individuals that possess all skills are rare, if existing. In order to tackle this problem, teams include specialists in each domain, and design is distributed among the members of the team. In particular, graphical designers and computer scientists (or more precisely interaction programmer) are among the kind of specialists involved in the design of interactive software.

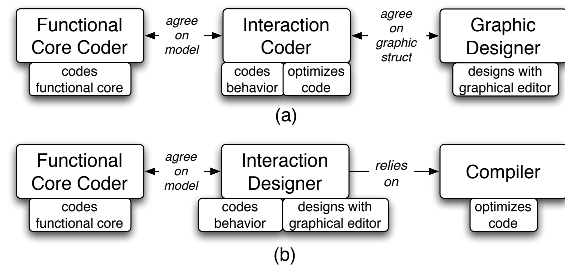

Figure 2. Role repartition with Intuikit and XAML (a) and role repartition with Hayaku (b).

The general idea of our approach is wider than the Intuikit approach [6]: instead of acknowledging the irreconcilability between graphical designers and interaction programmer, and maximizing communication between two different specialists, we tried to make the programmer's concerns accessible to the graphical designer. More precisely, what we target is a graphic designer that has basic programming skills, and that the tool empowers. As depicted in Figure 2, Hayaku provides the required graphical expressive power, while offloading optimizations to the graphical compiler. This turns the *interaction coder* and the *graphic designer* into an *interaction designer*. Again, we assume that the artefacts produced at this stage in the design should be done with all concerns (graphics and code) in mind, and thus by a unique person, or a very close team that share tools and artefacts. The approach is similar to Artistic Resizing: instead of describing with code the behavior of graphical elements under size change, Artistic Resizing enables graphic designers to express the behavior with means closer to their knowledge.

We provide the interaction designer with a tool chain that uses a standard vector graphics editor (Inkscape or Adobe Illustrator) as its first link. This has two advantages: the designer leverages on her experience with such tools, and she

can express graphics using the full expressive power of the tools. The other links of the toolchain consist in a compile chain that takes two inputs: graphics elements edited with the graphical editor, and abstractions of graphical element to control them. The remaining of this section enumerates the main features of Hayaku. One of the contributions of this work is the identification of those features. The goal of the paper is to present the concepts used by the tool, and show why they are adapted to the activity we target. Though there is not enough information to fully describe the system because of limited space, the concepts presented here can be used by readers if they want to design a similar system.

Abstract and Control Graphical Elements The graphic editor stores the drawings in an SVG description. SVG drawings are like "classes" of graphical objects. In order to use SVG drawings in a real application, the designer has to provide three descriptions, all written in JSON[4]. The first one is the "conceptual language" shared with the functional core coders, and serves as a bridge between the functional core and interactive graphics. As in [6], the designer and the rest of the team must agree on a common data structure, or "models" which also acts as classes of concepts. This language is illustrated by the right part of Figure 3. The second one describes how the models defined in the first description is related to SVG graphics, by connecting fields of the models to nodes and attributes in SVG drawings with a mini, data-flow-like functional language. It is similar to a stylesheet. In Figure 3, the connections are represented by the lines between the SVG part and the abstract model part. Finally, the last description is the "scene", i.e. a list of instances of the classes (not represented on Figure 3).

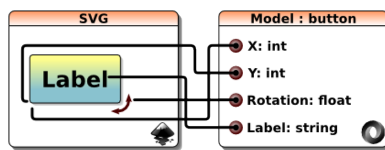

Figure 3. Representation of the connections (the black lines) between the graphical classes (the SVG) and the model.

Though this conceptual model of application design seems complex, it is no more than existing ways of writing code: the first JSON description can be considered as a class definition, the second one as a stylesheet, while the last one corresponds to the instantiation phase of classes at the launch of a program. The only addition is the SVG description, which corresponds to "graphical classes" definitions.

Fast Application Generation

Hayaku includes a compiler that takes the SVG description and the three JSON files as input, and generates an application. The compiler uses various strategies to maximize compile speed and launch speed of the generated application. This allow for rapid fixes and tests, and thus efficient exploration of design.

[4]Javascript Object Notation

Fast and Portable Code Generation

As many compilers, the graphical compiler is able to optimize the generated code. Thanks to a data-flow analysis, and user-provided hooks, the code allows the use of complex graphics (expressive power) with a rendering speed compatible with interaction. Furthermore, the compiler is able to target different graphical back-ends, such as OpenGL or Cairo. This guarantees that the design is portable.

Generate Whole Application or Embeddable Code The compiler can generate either a stand-alone application, or embeddable code. With traditional toolkits, embedding is often limited to a window that the host application displays next to its own windows. The kind of embedding that we target is more useful: graphics should appear inside an existing scene of the host application. Such embeddable code allows for creation of dynamic applications, in which the number of graphical elements is not known at compile-time. This also allows designers to use the compiler as a translation tool between SVG and a run-time environment. More generally, this transforms our toolset in a toolkit for graphical toolkit design (a toolkit of toolkits).

USE CASES

In order to illustrate our approach, we describe how to use Hayaku to implement three different kinds of applications. Though the descriptions look like a tutorial, they enable to understand and assess how a designer is supposed to use the features provided by the tool, and help evaluate how efficient the features are at supporting the designer's activity. The first one is a basic multi-touch application that enables multiple users to move and resize simple graphical objects. It is not very rich in terms of graphics, but since it is simple, it allows for a gentle introduction and short code examples. The second one is a more graphically complex application: a resizeable keyboard with a fish-eye effect that is activated only if the size of the keyboard is too small. The last example is a generic pie-menu that can be reused in an existing application.

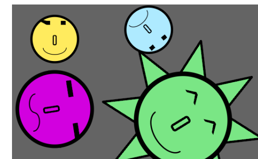

Figure 4. A simple multi-touch application.

Writing a Simple Application

This test-case consists in writing a simple multi-touch application (Figure 4). The interaction consists in controlling in a simple and natural way each of the "heads" that appears on Figure 4. The properties that users of the application can control are the position, size and rotation of each shape.

For the designer, the first phase consists in defining four *graphical "classes"* (here the "head"-shapes) with Inkscape, and save them in a SVG file.

```
{  "model": "SMILEYS",
    "classes": [  {
        "name":"Object",
        "extends":null,
        "attributes": {
            "ID":"key",
            "X0":"vint", "Y0":"vint",
            "SCALE":"vfloat",
            "ROTATION":"vfloat",
            "PRIORITY":"vfloat",
            "Picked_Key":"vint" }},
     { "name":"Object_0",
        "extends":"Object",
        "attributes": {}}]}
```

Figure 5. Model of the multi-touch widget.

```
{"model":"SMILEYS",
  "objects": [
  {"className":"Object_0",
    "file":"demo.svg",
    "graphicalItems": [
    {"name":"smiley_svg",
      "connections":
      {"X0":"smiley_svg.transform.tx",
        "Y0":"smiley_svg.transform.ty",
        "SCALE":"smiley_svg.transform.scale",
        "ROTATION":"smiley_svg.transform.rotation",
        "PRIORITY":"smiley_svg.transform.priority"},
      "picking":
      {"Picked_Key":"smiley_svg"}}]}]}
```

Figure 6. Connection between the model of the multi-touch widget and the graphic parts (smiley_svg).

```
{  "name":"Smileys",
    "model":"SMILEYS",
    "content": [
    {  "type":"Object_0",
        "attributes": {
            "ID":0,
            "ParentID":0,
            "X0":100, "Y0":100,
            "SCALE":0.5,
            "ROTATION":0.0,
            "Picked_Key":-1 }}] }
```

Figure 7. Instantiation of the multi-touch widget.

```
def translate(self,dx,dy):
    self.x0.set(self.x0.eval() + dx)
    self.y0.set(self.y0.eval() + dy)
def rotate(self, dr):
    self.rotation.set(self.rotation.eval() + dr)
def zoom(self, z):
    if self.scale.eval() + z >= 0.1:
        self.scale.set(self.scale.eval() + z)
```

Figure 8. The Python code of the three commands to control the graphical objects.

The third phase consists in defining the *connections* between the model and the graphical part (Figure 6), again in a JSON file. Connections are straightforward and need no explanation. The fourth description pertains to the *scene*, in another JSON file. This file consists in instantiating the different elements of the graphical scene (Figure 7).

The designer has to provide the reactive part of the application, i.e. the connection between input events and reaction of the graphical objects. Since Hayaku focuses on the rendering part only, it does not provide any multi-touch capabilities. Rather, it is up to the designer to describe with the run-time language and input toolkit how events act on the conceptual model, by updating the corresponding fields of the instances. However, when generating the code corresponding to the conceptual model, the toolset offers the possibility to concatenate user-defined code. This enables the designer to abstract behavior (see Figure 8). Furthemore, Hayaku provides a picking mechanism that can be called from user-defined code.

In order to test and launch the application, the interaction designer edits a Python script that contains a call to the function *load* with the three JSON files as arguments (the model, the model-to-svg connection, and the scene). She then launches the command *hayaku* with the script as a parameter. If the compile phase succeeds, Hayaku launches the generated application.

The compilation time for this example is 2.2 seconds the first time. Further recompilations requires 1.9 secs only. The first time of compilation is longer due to some tools that need to be embedded in the final application and that does not need to be recompiled each time a change occurs (OpenGL shaders and utility functions). The application takes less than one second to launch, and runs at 515 frames per second (see Table 2). Again, this application is simple and not demanding in terms of computation power. Still, it shows that the toolkit is reactive enough to deal with high-rate incoming data.

A Fish-eye Keyboard

The second application is a 40 auto-expanding keys keyboard, designed for motor-disabled users (Figure 9) [22]. The keyboard consists in two parts: the keyboard itself, and a one line screen to display the result. The caps-lock key is fully functional: the key mapping changes accordingly. The key *"123"* toggles the numeric mode. Finally, the keyboard can be resized, and at low sizes the keys close to the cursor expand thanks to a fish-eye effect [10].

Figure 9. The test application in action.

This example demonstrates the ability of the toolkit to handle rich graphics with high rendering performance. The design of the keyboard uses a full vectorial description for its components. This leads to high quality graphics even when the keyboard is resized. The design also uses rich graphic properties: gradients, transparency, shadows. . .

Realisation

The graphical part of the application has been realised with Inkscape (Figure 10). In a first SVG file, the designer creates a key by using eight separate graphical layers. The layers are grouped and named in a unique SVG component. In order

to build the global composition, the keys are then cloned, organized and modified to generate an artwork of the final keyboard. The creation of the upper area, including the text display, the backspace key and a background with a gradient completes this artwork. The whole keyboard contains 400 graphical elements.

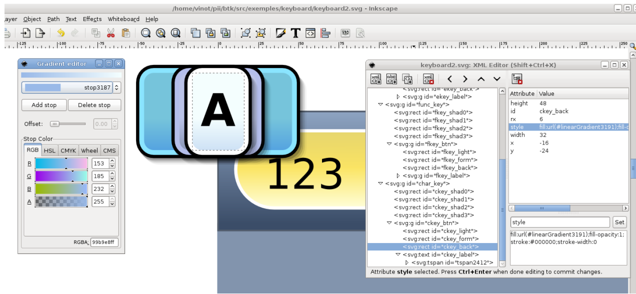

Figure 10. The SVG description of the different components of the keyboard, realized with Inkscape.

Once the global composition is satisfactory, three examples of the different type of keys are put in a separate SVG file, to serve as "graphical classes" : *char_key*, *func_key* and *enter_key*. The graphical components correspond to the component described in the model, and are named accordingly. The blocks that describe the background and the display of the result are also added to this file. A parent class *Key* has been defined to handle the common properties of the different keys. The class is inherited by the different types of key (*char*, *func* and *enter*).

The layout of the keyboard is given in the JSON *scene* file. However, Hayaku does not provide a visual editor for the scene. Thus, the designer has to provide it. Since the production of this file can be laborious, a script has been written to produce it. This script allows the interaction designer to rapidly change the layout of the keyboard by changing some variables in the script, instead of a bunch of values and parenthesis into the JSON file.

The fish-eye effect is implemented by computing the distance between the cursor and each key, and by using this distance to set the scaling property of the key accordingly. Each time the cursor moves, a redraw is triggered, and the key is scaled with is current scale before being drawn.

A Generic Pie-menu

To assess that Hayaku can be considered as a toolkit of toolkits[5], we implemented a generic pie-menu (Figure 11). The objective was to provide an implementation-independent description in order to use it inside an actual, existing application (Figure 12). The pie-menu we designed includes a feedback when flying over a slice: the underlying slice is enlarged. Thus we can not use a mere circle, but several distinct slices. We also need to be able to control the number of elements inside the menu.

[5] here, Hayaku can be considered as a toolkit for building a widget

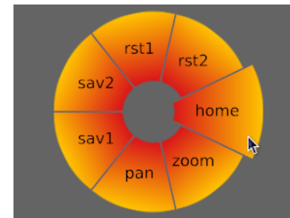

Figure 11. The pie-menu in action.

Realisation

The design in itself resembles the design of the keyboard: we designed the pie-menu to be a set of slices. Each slice has 7 main graphical parameters: a *position*, a *label*, an *angle*, an *internal radius*, an *external radius*, a *rotation*, and a *color* parameter. To describe the scene, we wrote a script similar to the one that generated the keys in the keyboard. The script generates the slices and their parameters according to the number of slices.

The behaviour part maps the picking value of each slice with a callback that changes the internal radius, the external radius and the color as needed. High-level events, such as "*menu 7 has been selected*", have to be generated by the behaviour part, since Hayaku only provides the graphical part of the application.

Embedding in an Existing Application

We have embedded the pie-menu into an existing radar-like application for Air Traffic Control (see Figure 12). This application is written in C++ and makes extensive use of OpenGL. The application is extensible, and provides a mechanism for loading dynamic external libraries. We used this mechanism to plug our pie-menu into this system.

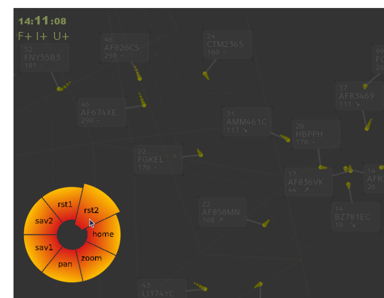

Figure 12. The pie-menu inside a real application.

The steps involved were the following. First, we had to write a C++ class that interacts with the dynamically loaded objects generated by Hayaku. This class is the glue that links the host application and the generated interactive graphics, and factorizes the setup code for all embedded Hayaku code. Then we wrote a subclass specific to the pie-menu, to handle the pie-menu behaviour with respect to user interaction. This subclass represents 112 lines of code. It is a transcription in C++ of previously written Python code, developed during the prototyping phase of the pie-menu widget. As

Figure 12 shows, the pie-menu smoothly integrates into the host application, and does not reduce the frame rate.

This use case shows that it was possible to externalize the creation of widgets and reuse them in other applications. However, in general, existing systems do not support extensions with external dynamic plug-in: in this case, the code generated by Hayaku must be embedded at source-level. The glue between the original code and the graphical part is simpler (just a "#include" at the beginning). Drawing is initiated by calling the exported *draw* function.

TOOLKIT IMPLEMENTATION

How the Toolset Works

The command *hayaku* automatically calls the GrC. The GrC then creates a directory named *BUILD* in which it places all its productions. The JSON files are transformed into Python ones, and a set of C files and their headers are written. Then, the GrC calls *gcc* to compile those C files and produce the object files that can be embedded into C applications. It generates a dynamic library that can be either linked to the run-time of the GrC, or embedded into an existing application.

To reduce compile time, the compiler is able to detect parts that have been modified between two successive compilations, and compiles those parts only. In addition, we designed a monitoring system on the files, and the recompilation occurs automatically whenever a file is modified and saved. The change is automatically reflected in the generated application while it is still running. For example, changing the color of one of the shapes in the example above with Inkscape, and saving the SVG file automatically updates all shapes of this class in the running application. This illustrates the advantage of separating graphics from behavior and using data-flow mechanisms: since the graphical pipeline is clearly delimited, the toolset is able to trigger it at any time, without affecting the behavior of the whole application. Such tools reduce the time needed between envisioning an idea and testing it.

Generation of Portable Code

As we already said in the previous sections, the designer produces the graphical shapes thanks to SVG files. The abstractions and connections between those graphical shapes and the models are given through JSON files. Then, Hayaku loads them into the GrC.

The GrC in itself is written in Python. The GrC is able to produce different types of outputs, in terms of target language and run-time (currently C and Java), and in term of graphical backend (currently OpenGL, and partly Cairo). To be able to reuse the code of the transformations, we implemented our own partial class mechanism. We separate the description of the intermediate languages and the transformation between them. At the beginning of the compile chain, the GrC chooses which languages and transformations it needs to produce the final code by attaching the transformation functions to the descriptions nodes. The trees that are generated can then be transformed just by visiting each node. This mechanism allows us to modularize the graphical compiler and thus to plug different behaviour at different stages as needed.

Generation of Static and Semi-static Code

Most examples are instances of application in which the number of objects is not variable (sliders, pie-menus, keyboard). For other types of applications, such as radar image where the number of flights is in theory not bounded, the data-flow architecture does not allow for simple description and handling of dynamic creation of objects. In this case, Hayaku provides two strategies.

The first one is to consider the number of elements to be displayed bound by an upper limit [24]. This requires to start the application with a pool of available invisible graphical objects, which are allocated to any new data that appear during run-time. In practice, this strategy works well: for example, the number of flights in a sector is bounded by regulation agencies in order to enable a limited numbers of controllers to handle the traffic. It comes at the expense of internal handling of invisible objects (which may hinder performance uselessly) and longer compile time. But the benefits outweigh the drawbacks, since it helps keeping the application simple to write and understand.

The second strategy consists in generating pieces of specific interactive graphical code that can be reused in a larger program. In the radar image, this would consist in designing the graphics for a single flight, and generating the corresponding display code. The main program would then manage creation of new flights and deletion of disappearing ones, and use the display code whenever necessary. With this solution, the compile time is reduced, since the graphical code is not unrolled as in loop unrolling for instance, and the constraint of the upper limit of objects is removed.

Generated code must follow a number of requirements to make it embeddable. First, the generated code has to keep the state of the application. For instance, when working with OpenGL applications, the drawing code has to keep the pipeline in the same state it was before its use. A second requirement is to produce "human readable" code. Since most of the time a designer will connect the generated code to the other application, the names of the functions that are exported have to be understandable by the programmer. For instance, *set0_25_2_1* is less readable than *set_component0_key25_backgroundColor_red*.

Picking Support

The generated code must provide a way to send back information. For instance, when the end-user moves the mouse, the code has to inform the caller that the picking state changed. Hayaku provides a picking mechanism, together with a callback system. The host application has to register callbacks if it wants to be notified by the graphics code, or by the underlying dataflow. Care must be taken when handling picking. For example, a usual picking algorithm consists in rendering the scene in a tiny rectangle around the cursor, and storing each graphical object that owns pixels actually rendered in the rectangle. Applying the same algorithm

in a multitouch application requires as many passes as the number of touches, which is costly. Instead, we used a one-pass color-keying algorithm [12]. Each graphical shape is assigned a unique color in an associative array, and rendered with their unique solid color in an off-screen buffer. Picking shapes consists in reading back the color of the pixel under each touch, and retrieving the corresponding shape from the color with the associative array.

PRELIMINARY EVALUATION

As with any method that aims at supporting design, evaluating a toolset requires controlled experiments, with multiple design teams under different conditions (with or without the tested toolset for example). Such an experimentation is a heavy task, and is beyond the scope of this paper. However, we provide in this section a preliminary evaluation in terms of descriptive power, performance, and usability.

Descriptive Power

We provide two dimensions of analysis to evaluate the descriptive power of the toolkit: the size of the class of visualizations that can be described by the toolkit in a reasonable amount of work, and the simplicity of the description of typical applications. A toolset must target the right balance between the class size and simplicity. A thin class may indicate that the toolset is so specialized that the benefits provided are not very significant. On the other hand, expanding the class usually comes at the expense of simplicity.

Class of Application: previous work showed that the GrC is able to handle basic WIMP interaction (sliders) and graphical scene with a large number of objects, such as a radar image. We showed with the use-cases of this paper that Hayaku can implement multiple types of interactive graphical software: interactors (pie-menus), graphically rich interactive software (fish-eye keyboard), and multitouch applications.

As said before, most examples are instances of application in which the number of objects is not variable (sliders, pie-menus, keyboard). For other types of applications, such as radar image where the number of flights is in practice bounded, a strategy consists in picking objects in a pool of available invisble objects. Hayaku enables to use a second strategy that relies on embeddable, generated code, thus expanding the class of applications.

Using a graphical editor also enables the designer to expand the class of representation he can employ. However, we did not try to design very dynamic applications such as graphical editors with Hayaku because we think that Hayaku is not made for that kind of applications. We suspect that writing such systems would require to twist the conceptual model of application design so much, that it would be too cumbersome to do.

Simplicity: Despite our research in the literature, we could not find a clear definition for simplicity. Thus, we measured it in terms of compactness of the code required to describe interactive graphics, by providing the number of lines of code (LOC) of previously described examples. (Table 1). As said before, the JSON description of the scene (the graphical components of the interface) has been judged as "laborious", and a Python script to produce it has been required. It corresponds to the "generator" column. For example, the 890 LOC for the keyboard have actually been generated by the 210 lines of code generator. As we can see, the amount of code is in the hundreds, which is low considering the richness and variability of the three examples.

use case	conceptual model	model to SVG	scene	generator of the scene
multi-touch	43 LOC	90 LOC	54 LOC	$\emptyset$
keyboard	129 LOC	199 LOC	890 LOC	210 LOC
pie-menu	40 LOC	42 LOC	102 LOC	46 LOC

Table 1. The number of lines of code (LOC) of the different examples.

Performance

In Table 2, we show the performances of the three use-cases, compiled with Hayaku, and rendered through OpenGL. For each example, we show the frame rate of the produced code (C+OpenGL), and the time needed to compile it. We differentiate "first compile-time" from "re-compile time", because Hayaku caches some computation between two consecutive compile phases (text fonts for example). The most significant time is the re-compile time, since a designer using Hayaku will spend most of her time doing small increments to her description, and will launch recompilation from time to time.

use case	frames per second	first compile time	re-compile time
multi-touch	∼515 f.p.s.	2.2 sec	1.9 sec
keyboard	∼136 f.p.s.	29.1 sec	8.6 sec
pie-menu	∼400 f.p.s.	10.2 sec	2.9 sec

Table 2. The performances of the different examples.

If performances may not be as good as expected, they could be much higher (8.6 sec re-compile time for the keyboard). The implementation of the toolkit we show here is a prototype (written in the Python language), and could be improved in many ways. For instance, the produced OpenGL code does not use Vertex Buffer Objects, which could significantly improve the run-time performances. In addition, the internal data structures of Hayaku and the GrC (graphs of tiny Python objects) should be changed to decrease compile time.

Usability

Evaluating the usability of a toolkit is an open research problem [20]. For this purpose, we discuss how Hayaku ranks against Cognitive Dimensions of Notation [11], which help make explicit what a notation (i.e a language) is supposed to improve, or fails to support. Cognitive dimensions are based on activities typical of the use of interactive systems. We chose to evaluate the following activities: *incrementation*, *transcription*, *modification*, and *exploratory design*; along the following dimensions: *closeness of mapping, hidden dependencies, premature commitment, progressive evaluation, abstraction, viscosity*, and *visibility*.

Closeness of Mapping: the designer creates (*incrementation*) drawings directly into a graphical editor: it is very *close* to the final product, at least closer than textual graphical language. This allows the use of existing *exploratory design* tools (inkscape), and thus maximizes this property. *Modification* of the graphics is eased since it modifies in turn an SVG file that keeps the same properties (e.g. naming), which in turn is compiled i.e. transformed computationally. Porting can be considered as a *transcription*, and is efficient thanks to the use of a compiler with multiple front-ends and back-ends. The front end of the compiler is the conceptual model JSON file. Since the interaction designer designs the conceptual model, she can make it as close as possible to the domain she models. Hence, closeness of mapping is maximized. However, setting the link between the graphics, the conceptual models, and the data-flow language requires a switch of notation (a graphical editor vs a textual notation).

Hidden Dependencies: the dataflow we provide is not entirely visible. It is difficult for the designer to know exactly what happens once the models and transformations are given. However, the designer is mostly interested in the part of the data-flow he wrote. The part of the data-flow generated by the compiler is less susceptible to be read and understood, except for debugging purpose. *Premature Commitment*: using a graphical compiler inherently prevents premature commitment. For example, changing the run-time environment can occur at any time during the design process. Furthermore, changing a property of the graphics may require a simple recompile to be reflected in the application. Moreover, as Hayaku relies on style-sheets to link the graphical model to the graphical shapes, the design can be rewritten several times without having to rewrite the behaviour part. However, the structure of the graphics must not change too often, since other descriptions rely on it (see viscosity). *Progressive Evaluation:* evaluating a recently modified graphics is immediate. However, evaluating the behavior with respect to the interaction requires to launch the software. Clearly, a tool such as artistic resizing is needed for this kind of activity and concerns.

Abstraction: Hayaku relies on JSON files to *abstract* the graphical model, the connections and the graphical scene. However, if this language is well adapted to represent abstract data, and forces the user to keep it abstract, it is not very well adapted to the human that needs to write it into his/her text editor. In particular, the connection between the models and the graphics would be better defined directly in a graphical editor.

Viscosity: the conceptual model requires all graphical elements to be declared in the JSON file. Hence, if a graphical element is used multiple times (such as the key element in the keyboard example), a change in the "prototype" requires propagating the change in all instances of that element. A solution to this *viscosity* problem is to design a small program that generates all instances from a prototype in a JSON file. This program can be considered as another link in the compile chain, and helps abstract concepts from the conceptual model of the application to be designed.

A change in the conceptual model itself must be reflected into the connection description, and the scene description. This is the problem that the programmer of a C++ class encounters when he adds a field for example: he has to update all calls of the class constructor if a parameter to set up the field is required. Various mechanisms exist to cope with this problem (e.g. a default value), but none is implemented in Hayaku. Similarly, a change in the graphical structure (i.e. the hierarchy of SVG elements) can have a large impact on the model-graphics connection description.

Visibility: currently, the *visibility* of the toolkit is limited. For example, JSON files tend to be verbose and long, which hinders searching or exploratory understanding.

EARLY FEEDBACK FROM DESIGNERS

We provided Hayaku to the graphical designer of the original Fish-Eye Keyboard, and we asked him to recreate it. This designer is used to both design graphics and write interaction code. The designer praised the reliability of the rendered scene. Since Hayaku relies on a graphical compiler, the final generated code does not suffer from a trade-off between speed and power of expression. The final rendered scene is then very close to a static one, produced by Inkscape for instance. Thanks to the expression power of SVG, the graphic designer is not limited when dealing with graphics.

The designer found that one of the most interesting thing was to design the graphical objects by keeping in mind their graphical behaviour during the interaction. This behaviour has been defined by targeting the graphical properties that need to be *connected* to the models in the graphical scene. The evolution of the parameters are then described, either "relatively" with a mathematical expression (similar to the one-way constraints in Garnet [25]), or with a value computed by the behaviour part. For example, the anchor of the shape of a key depends on its width ("FORM_X0": "(self. WIDTH - 100) / -2"): since the width depends on the distance with the cursor, the anchor is updated automatically. Considering all the inputs and outputs of the generated application as a data flow simplified the work of the designer. For instance, implementing the global resize of the keyboard took around 10 minutes, the time needed to understand and implement the solution to connect the two variables screen_width and screen_height to the application. The "connections" allow the graphic designer to quickly build complex behaviour, such as the "fish-eye" function of the keys.

However, there still are some pitfalls. The main problem was the hand writing of the different JSON files. This has been judged as laborious, since the coherence between those files had to be maintained manually. Furthermore, writing a JSON file for the scene is also annoying, since a scene can contain many similar elements. As explained, a solution to this problem is to write a script that generates the scene, which makes it more controllable.

CONCLUSION

In this paper, we have identified that there is a lack of tools to support designers in producing graphically rich, finely tuned

and highly reactive graphical applications. We have presented Hayaku, a toolset that aims at supporting this activity, by turning the interaction coder and graphical designer into an interaction designer. The interaction designer writes the program in a high-level, known language (SVG) and through JSON files that abstract the graphical elements. He then compiles it into an runnable application or embeddable code.

Like the keyboard example shows, the compile time hinders design exploration, and must be improved. We have developed Hayaku in Python in order to prototype it rapidly, and we are aware that parts of the code are sub-optimal (notably trees traversal). Many optimizations can be done to improve that part of the toolset. Future works also include expanding the sets of back ends, both for graphics platform and languages. Finally, using multiple JSON files as a description language is cumbersome, especially when describing the connection between models and graphic models. Specialized tools must be designed, such as a graphical editor.

REFERENCES

1. Appert, C., and Beaudouin-Lafon, M. SwingStates: adding state machines to Java and the Swing toolkit. *Software: Practice & Exp. 38*, 11 (2008), 1149–1182.
2. Beaudouin-Lafon, M., and Mackay, W. Prototyping tools and techniques. In *The Hum. Comp. Inter. Handbook*, A. Sears and J. A. Jacko, Eds. CRC Press, 2007.
3. Bederson, B., Grosjean, J., and Meyer, J. Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering 30*, 8 (aug. 2004), 535 – 546.
4. Bederson, B. B., Meyer, J., and Good, L. Jazz: an extensible zoomable user interface graphics toolkit in Java. In *Proc. of UIST '00* (2000), ACM, 171–180.
5. Buxton, B. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufman, 2007.
6. Chatty, S., Sire, S., Vinot, J.-L., Lecoanet, P., Lemort, A., and Mertz, C. Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proc. of UIST '04* (2004), ACM, 267–276.
7. Dragicevic, P., Chatty, S., Thevenin, D., and Vinot, J.-L. Artistic resizing: a technique for rich scale-sensitive vector graphics. In *Proc. of UIST '05* (2005), ACM, 201–210.
8. Dragicevic, P., and Fekete, J.-D. Support for input adaptability in the ICON toolkit. In *Proc. of ICMI '04* (2004), ACM, 212–219.
9. Fekete, J.-D. The InfoVis Toolkit. In *Proc. of InfoVis'04* (October 2004), IEEE Press, 167–174.
10. Furnas, G. W. Generalized fisheye views. *SIGCHI Bull. 17*, 4 (1986), 16–23.
11. Green, T. R. G. Cognitive dimensions of notations. In *Proc. of HCI'89* (1989), Cambridge University Press, 443–460.
12. Hanrahan, P., and Haeberli, P. Direct WYSIWYG painting and texturing on 3D shapes. In *Proc. of SIGGRAPH'90* (1990), ACM, 215–223.
13. Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S. R. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proc. of UIST '08* (2008), ACM, 91–100.
14. Heer, J., Card, S. K., and Landay, J. A. Prefuse: a toolkit for interactive information visualization. In *Proc. of CHI '05* (2005), ACM, 421–430.
15. Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of CGO'04* (Mar 2004).
16. Mertz, C., Chatty, S., and Vinot, J.-L. The influence of design techniques on user interfaces: the DigiStrips experiment for air traffic control. In *Proc. of HCI Aero IFIP 13.5* (2000).
17. Muller, M. Participatory design: The third space in HCI. In *The Hum. Comp. Inter. Handbook*, A. Sears and J. A. Jacko, Eds. CRC Press, 2007, 1061–1081.
18. Myers, B. Challenges of HCI design and implementation. *Interactions 1*, 1 (1994), 73–83.
19. Myers, B., Park, S. Y., Nakano, Y., Mueller, G., and Ko, A. How designers design and program interactive behaviors. In *Proc. of IEEE VL/HCC '08* (2008).
20. Myers, B. A. Usability issues in programming languages. Tech. rep., School of Computer Science, Carnegie Mellon University, 2000.
21. Myers, B. A., and Rosson, M. B. Survey on user interface programming. In *Proc. of CHI* (New York, 1992), CHI '92, ACM, 195–202.
22. Raynal, M., Vinot, J.-L., and Truillet, P. Fisheye keyboard: Whole keyboard displayed on small device. In *Proc. of UIST '07: poster session* (Oct 2007).
23. Snyder, C. *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces (The Morgan Kaufmann Series in Interactive Technologies)*. Morgan Kaufmann, April 2003.
24. Tissoires, B., and Conversy, S. Graphic Rendering Considered as a Compilation Chain. In *DSV-IS'08* (2008), no. 5136 in LNCS, Springer, 267–280.
25. Vander Zanden, B. T., Halterman, R., Myers, B. A., McDaniel, R., Miller, R., Szekely, P., Giuse, D. A., and Kosbie, D. Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Trans. Prog. Lang. Syst. 23*, 6 (2001), 776–796.