Improving Introductory Programming Courses by Using a Simple Accelerated Graphics Library

Thomas Larsson and Daniel Flemström
Department of Computer Science and Electronics
Mälardalen University
Sweden

Abstract

We present a platform independent and hardware accelerated graphics library, which has been found to be a suitable educational tool for novice programmers. The purpose of the library is to change the nature of the assignments and projects used in introductory programming courses to something that fascinate and stimulate the students, e.g., game creation. We describe our experiences from using the presented graphics library in four different course instances of our introductory C++ course. The course surveys show that most students found the approach interesting and fun. As faculty, we could clearly see how many students became highly engaged in their projects and some of them accomplished solutions way beyond our expectations. In particular, compared to the programming courses we have given in the past, in which a standard framework for creating window applications was used, we have noticed a significant improvement in terms of the quality of the students' project solutions.

CR Categories: K.3.2 [Computers and Education]: Computer and Information Science Education—Computer Science Education

Keywords: programming, teaching, motivation, graphics, games

1 Introduction

The motivation that drives the students is of vital importance for what they accomplish during their education. This is especially true in programming courses, where students have to spend a lot of time practicing, i.e. writing and debugging code [Jenkins 2001]. But what makes students in introductory programming courses interested and enthusiastic? Despite that there are no simple answers to this question, we have noticed that there are some types of projects that can make a difference. Graphics, multimedia, and game applications seem to fascinate a broad category of students [Guzdial and Soloway 2002]. In particular, many of our students seem to enjoy programming video games.

Therefore, we have experimented with this idea of using a graphics library in four different instances of an introductory course on C++ programming. Students entering the course were assumed to have previous programming experience equivalent to five weeks of training in imperative programming. As a final project in the course, we let the student implement a sprite based video game in 2D. We wanted the complexity of theses games to resemble the complexity of classical games like e.g. Pacman, Asteroids, Space Invaders, or Tetris, including graphics, interactivity, and sound. To make this a reachable goal, we implemented the Simple Accelerated Graphics Library (SAGLib). The main contribution of the library lies in the unique simplicity in the way it provides access to hardware accelerated computer graphics for novice C or C++ programmers. Other existing tools are either designed to be used with other programming languages, have unnecessarily complex APIs for novice students, do not support the creation of fully interactive graphics applications, or do not exploit hardware accelerated graphics.

The results of using our library were mainly satisfactory. In course surveys, as well as in class-room conversations, the students expressed that by using the simple graphics library, the programming experience became visually rewarding, which served as an extra motivation factor for them to produce better and more interesting project solutions. It also turned out that the library in itself introduced no, or very little, extra complexity, compared to programming text-based applications. Thus, the students were able to focus on the design and programming of the application, without worrying about low-level graphics details.

The remainder of this paper is organized as follows. In the next section, relevant prior work is reviewed. Section 3 briefly presents the design of the library and some minimal example applications are given. How the library has been integrated in our introductory C++ course is described in Section 4, and some examples of student projects are also given. Then, in Section 5, the evaluation results as well as the experiences gathered from using the library in our courses are discussed. Finally, our conclusions are given in Section 6.

2 Related Work

Increasing the motivation by using graphics in introductory programming courses is not a new idea. Roberts reports on using a C-based Graphics Library for the first programming course with good results [Roberts 1995]. Another suggested approach supports the creation of simple graphics applications, while keeping the simplicity of text-based programming by letting the compiler automatically create a graphical user interface for programs that otherwise would appear to be text-based. For example, this makes it possible for students to create simple games like tic-tac-toe, checkers, or battleship. More sophisticated graphics programs, however, including e.g. animations, are not possible in this case [Carlisle 1999].

A more powerful approach is to provide a simple graphics library that hides as much of the code complexity as possible that arise when using advanced graphics and windowing systems, but still enables the possibility to create powerful graphics programs. Astrachan and Rodger report that the key benefit of using their library is that interactive graphics generate student interest and enthusiasm, which often lead to increased learning and mastery of the course material. In addition, in many cases, the visual feedback from the programming assignments immediately reveals the existence of bugs, which provides guidance in the debugging process [Astrachan and Rodger 1998].

Childers et al. present EzWindows, which is a simple and portable graphics library for teaching object-oriented programming using C++. The main purposes of their library are to help students grasp the object-oriented paradigm as well as make it possible for students to create programs that resembles the look of other programs in modern desktop computers [Childers et al. 1998].

Rasala argues for the use of toolkits in general in any modern first year computer science curriculum. For example, graphics toolkits are needed because graphics is not a built-in feature in many

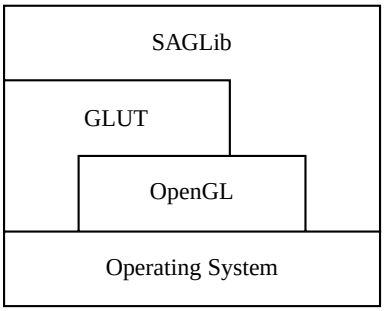

Figure 1: *SAGlib provides a complete layer that hides the operating system, GLUT, and OpenGL for the application programmer.*

languages. Graphics is necessary for many important computing activities, helps in the debugging process by revealing errors, and students are attracted to the discipline by computer-generated images [Rasala 2000].

The integration of a multimedia project in basic programming courses that emphasizes alternative learning styles through collaborative team work has also been proposed [Wolz et al. 1997]. In the media computation approach, suggested by Guzdial as a response to the widespread use of traditional, overly-technical, and abstract course content, all programs written manipulates sound, images, and movies to increase student motivation and hopefully enabling a deeper learning [Guzdial 2003; Guzdial and Soloway 2002]. The leading stars behind the approach were relevance (concrete domain examples), creativity (open-ended assignments), and social (collaboration and sharing of experiences) [Tew et al. 2005].

Meyer presents a new course design for novice programmers using the language Eiffel together with a graphics library called TRAFFIC, which includes some basic elements of a Geographical Information System [Meyer 2003]. The immediate usage of well-designed interfaces, APIs, hiding simple to use, but internally sophisticated, functionality is captured by the name Outside-In, or the Inverted Curriculum, rather than traditional bottom-up or top-down approaches. Data gathered from two course instances indicate student appreciation and higher grades compared to previous course versions [Pedroni and Meyer 2006]

Furthermore, we note that graphics and games are not the only way of letting students work on for them more meaningful and concrete projects. Another good example is to use programming of robots to concretize the projects and stimulate student interest [Lawhead et al. 2003].

Of course, there are many other factors besides the usage of appropriate toolkits and libraries that instructors need to take into account to be successful in teaching introductory programming, for example the choice of programming languages, programming environments, classroom techniques, and type of examination. These factors, however, will not be discussed in this article. An interesting taxonomy of the first two factors, however, has been given by Kelleher and Pausch [2005].

3 Library Design

Roberts [1995] proposed the following four important interface design criteria for their graphics library:

i It must be simple.

ii It must correspond to student intuition.

iii It must be powerful enough for students to write programs they think are fun.

iv It must be widely implementable.

Furthermore, we can add that the usage of a library in a course should be motivated by it giving a direct or indirect benefit with respect to the course objectives.

When designing SAGLib, these were the design criteria we had in mind. Therefore, we decided to provide only a limited number of simple functions. The API hides the inner complexity of OpenGL and GLUT. This makes it possible for the students to focus on data structures and programming rather than graphics programming. This is illustrated in Figure 1. Note that the application programmer only needs to know anything about SAGlib to create simple accelerated graphics applications. Both the C and C++ API provide functionality for

- setting up and displaying a window,
- controlling pens, colors and transparency,
- drawing bitmaps and shapes, with or without simple geometry transformation, and
- simple event handling for mouse, keyboard, resize, and redisplay

For simplicity, our graphics interface is mainly restricted to 2D graphics applications. It supports the drawing of basic 2D shapes, bitmaps, and text. An event-based execution model is used, together with basic timer functionality to support animation. The library also provides basic mouse and keyboard input handling. In our simple setting, this functionality makes the creation of simple video games, or the creation of other types of media rich applications, a reachable goal, even for novice programmers. We note, however, that it has also been demonstrated, through the ALICE system, that novice programmers can develop interactive 3D graphics and animation as part of their initial learning of programming concepts [Cooper et al. 2000].

To make the library useful for courses using the imperative as well the object-oriented programming paradigm, we use a two layer API, one in C and one in C++. Also, since our library is implemented on top of standard OpenGL, it becomes very powerful considering the huge improvements over the last few years in commodity graphics hardware. This also means that the library can make use of special 3D graphics functionality internally, for example texture mapping and alpha blending, as appropriate. This approach also make the library portable. Furthermore, we use GLUT for basic window handling [Kilgard 1996], which also is supported on many platforms. Both OpenGL and GLUT, however, are invisible for users of our library[1]. The latest version of our library is freely available on the web [Larsson and Flemström 2006]

Hardware acceleration is automatically enabled since SAGLib is based on OpenGL. We use an orthographic projection with a one-to-one mapping between world positions and pixel coordinates. No z-values need to be specified; they are assumed to be zero. Thus, the 2D graphics image will be specified in the x-y plane.

OpenGL was chosen for several reasons. First of all, we wanted access to the high performance available in commodity graphics hardware in a simple way. Many of the accelerated functions supported in 3D graphics hardware is very useful also in 2D graphics applications, for example double buffering, geometry transformation, primitive rasterization, texture mapping and alpha blending.

[1] It is still possible for skilled students to access OpenGL functionality directly if so wanted.

Function names	
sgDrawPoint	*sgDrawLine*
sgDrawRect	*sgDrawFilledRect*
sgDrawCircle	*sgDrawFilledCircle*
sgDrawTriangle	*sgDrawFilledTriangle*
sgDrawQuad	*sgDrawFilledQuad*
sgDrawText	

Figure 2: *The drawing functions supported in SAGLib.*

The tremendous performance of today's graphics accelerators, also means that everything can be redrawn at each display update without much performance penalty. Furthermore, the graphics rendering is double buffered for flicker-free animation. All this simplifies for the student who can concentrate on the main programming tasks instead of, for example, optimizing the graphics drawing strategy.

The geometrical drawing functions that are supported are listed in Figure 2. These basic shapes are drawn with the current pen and color. An alpha value is also associated with each chosen color. It can be specified as a value between $0 - 255$; that is, from fully opaque to fully transparent. The pen, transparency and color selections will be used until changed by new API calls. This reduces the number of arguments to the functions and the complexity of the API.

Since each color drawn may be transparent, and alpha blending is always enabled, many exiting effects can be created with little effort, for example, sprite animation and clouds. Transparent bitmaps are often unnecessary complex to achieve in other programming environments. In SAGLib, we have simplified internal representation of graphic formats by only using the true color format with 32 bits (RGBA) per pixel. Each pixel may have its own transparency. Thus, a bitmap may have arbitrary many transparent colors.

Also, since font handling usually is quite complex, text output has been reduced to one fixed font. Some students, however, have created their own text styles by drawing custom designed bitmaps. To reduce complexity even further, only one window is allowed per application and we have pre-defined the major application settings, just leaving the window size, position and caption to the user. All event handling has been hidden within the library. The user creates a display drawing function, which the library use as a callback routine. In this function, the API functions for pens, colors, transparency, bitmaps, and shapes may be used. Also, the system calls the display function automatically when for example the user moves or resizes the window.

Since threading is a very complex issue, we have reduced the number of timers to one, which means that the application will run in one thread. In the callback routine of the timer, the code to calculate and update the positions of e.g. moving shapes or sprites should be placed. Together with the support for drawing transparent bitmaps, the timer functionality makes it very simple for the students to include sprite animations in their applications.

3.1 Creating Applications

To make the library more widely usable, it includes support for both C and C++ programming. The C API encapsulates and hides the operating system, GLUT and OpenGL from the application programmer. An event-based execution model is used, which means that the application programmer can subscribe on events by registering callback functions. There are support for keyboard, mouse, redraw, resize, and timer events. An example of a minimal graphics application in C that draws a rectangle in the application's window

```
void myDisplay(void) {
  sgClearDisplay();
  sgDrawFilledRect(500, 20, 200, 230);
  sgFinishDisplay();
}

void main() {
  sgInitGraphics(400, 400,"MyApp", myDisplay);
  sgMainLoop();
}
```

Figure 3: *A minimal C program using SAGLib.*

```
class MyApp:public Application {
public:
  int cx;
  Bitmap bitmap;

  MyApp(void) : cx(0) {
    bitmap.initFromBMP("player.bmp");
    bitmap.addTransparentColor(255,255,255);
  }

  void onDraw(Graphics& g) {
    g.setColor(200, 50, 0);
    g.drawFilledRect(50, 50, 50, 50);
    bitmap.draw(g, cx, 100);
  }

  void onKey(unsigned char key,int x,int y) {
    if(key == 'l') cx = cx + 3;
    if(key == 'k') cx = cx - 3;
    updateDisplay();
  }
};

void main(void) {
  MyApp app;
  app.init("MyApp");
  app.showModal();
}
```

Figure 4: *A simple C++ program using SAGLib.*

is shown in Figure 3.

The C++ level completely encapsulates the C-level API which means that the students can use object-oriented programming throughout their entire applications. The C++ API consists of three classes. *Application*, *Graphics* and *Bitmap*. Each user application inherits from the Application class. The Application class sets up the window and handles the window events, such as mouse, keyboard, drawing events, timers and so on. Subscriptions for available events are automatically set up in the Application class by the usage of virtual member functions. You may also check the state of a specific key with the function *Application::isKeyDown* which allows games where several keys might be pressed at the same time.

The *OnTimer* method in the *Application* class should be overridden to handle position updates, calculations etcetera. Last in this method, a call to *Application::updateDisplay* will raise the *OnDraw* event. The Graphics class handles double buffering automatically and gives access to the accelerated graphics primitives in an object oriented way. All primitive graphics functions from the C-level API can be found as class methods on the Graphics object.

The example application in Figure 4 creates a window on the

screen, loads a bitmap from file and allows the user to control it with the keyboard. The bitmap acts as a simple sprite, where the color white is specified as completely transparent. It can be noted that the rectangle drawn before the user-controlled bitmap appear to be behind it at all times.

4 Course Design

Here we will present some details of how we have used SAGLib in our C++ course. The purpose of the course is to teach the fundamentals of the C++ language. In the course syllabus the following topics are included:

- Control structures: selection, iteration, recursion
- Functions
- Arrays, pointers, and strings
- Classes and objects
- Operator overloading
- Inheritance
- Dynamic binding
- Templates
- Data structures and algorithms in the Standard Template Library.
- IO streams and file processing

The course objectives state that students, upon completion of the course, will be able to

- Understand the differences between imperative vs. object-oriented programming.
- Writing object-oriented programs in C++ on their own that demonstrate mastery of object-oriented concepts such as abstract data types, encapsulation and information hiding, function and operator overloading, inheritance, and polymorphism.
- Understand the execution model of event-based programs with graphical user interfaces.

Some examples of topics that we usually only touch upon briefly in the course are exception handling, name spaces, and object-oriented analysis and design. Note, however, that these are not programming skills that were removed from the course because of the inclusion of our graphics library. The only thing that we have removed from earlier incarnations of the course is the usage of another more advanced class library for the creation of graphical user interfaces.

The course included lectures, laboratory work, take-home exercises, a project task, and a final written exam. The graphics library was introduced right from the start of the course. An overview of the library and some initial examples were presented already on the first lecture. Also, some initial exercises based on SAGLib were carried out in the first laboratory session.

For example, a concrete and visually rewarding way to practice iteration together with two dimensional arrays is simple image or bitmap manipulation. Some examples of operations that are suitable to be implemented by novice programmers are image color inversion, vertical and horizontal flip, and color to gray scale conversion. An example of the first operation is given in Figure 5. In this case, the inverted RGB color, I, for each pixel is simply given as

$$\begin{aligned} I_{red} &= 255 - S_{red} \\ I_{green} &= 255 - S_{green} \\ I_{blue} &= 255 - S_{blue} \end{aligned}$$

where S is the RGB color of the corresponding pixel in the source image. Other slightly more advanced exercises would be to implement linear filtering such as blur, sharpen and edge detection. These operations involves calculating a new pixel color as a weighted average of a neighborhood of pixels in the source image.

Figure 5: *Results of a simple image color inversion operation.*

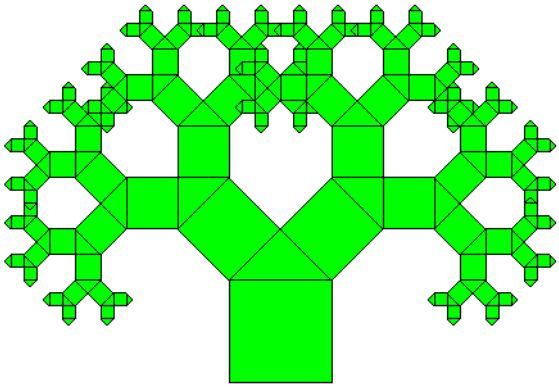

Figure 6: *The Pythagoras Tree — a simple example of a recursively generated image.*

When teaching recursion, which is a concept that many computer science students perceive as difficult, naturally visual examples may contribute to the students' learning and appreciation of this concept. Some concrete examples are various forms of simple fractal images such as the Pythagoras tree, the Koch snowflake, and the Sierpinski triangle (or gasket). In Figure 6, an image of the Pythagoras tree is shown which has been produced by a simple SAGLib program that we use as a code example when teaching recursion. Another interesting recursion example that we have used is a simple version of the flood fill algorithm for painting the interior of arbitrary shapes in a pixel-based image.

Year 2003									
Q	N	R	1	2	3	4	5	Average	Median
Q1	29	18	1	0	1	12	4	4.00	4
Q2	29	15	0	0	6	3	6	4.00	4

Year 2004									
Q	N	R	1	2	3	4	5	Average	Median
Q1	21	12	0	0	3	1	8	4.42	5
Q2	21	12	0	1	6	0	3	3.50	3

Year 2005									
Q	N	R	1	2	3	4	5	Average	Median
Q1	51	31	0	0	6	14	11	4.16	4
Q2	51	26	1	2	11	5	7	3.58	3

Year 2006									
Q	N	R	1	2	3	4	5	Average	Median
Q1	22	14	0	1	2	7	4	4.00	4
Q2	22	14	1	1	4	3	5	3.71	4

Table 1: *Results from course surveys during 2003-2006. Note that N is the number of enrolled students and R is the number of responses we received.*

Later on, somewhere in the middle of the course, take-home assignments are given out in which the students create a very simple example game step by step in order to learn to understand timer events, drawing simple shapes and keyboard handling. In another preparatory assignment, the state pattern is used to implement a rudimentary animated submarine game. The submarine has different animations depending on how it moves on the screen. The purpose of this is to learn handling interactive graphics and game character states.

4.1 Student Projects

As the major programming task or project, the students are given the opportunity to create a simple action game that includes sprite animation, several game levels, saving and loading options, and optionally a high score list and sound effects. The students are also allowed to design a game idea more freely of their own, if they so wish. In this way, they get the opportunity to choose a task that really interest them, potentially increasing their motivation and creativity further. They are required, however, to get their design approved by the course leader to make sure that the level of difficulty of the implementation seems reasonable with respect to the goals of the course. Screen shots from some student projects are shown in Figure 7. Other examples of projects the students have accomplished are remakes of early home computer games such as Manic Miner and Jet-Pack, helicopter and aeroplane games, and sports games.

5 Evaluation and Discussion

The proposed approach has been employed in four course instances during 2003–2006. Evaluation was carried out by course surveys at the end of each course. The following survey questions were used:

Q1 How would you rate the overall quality of this course (1-5)?

Q2 How would you rate the usage of SAGLib in the course (1-5)?

Q3 In what ways did you benefit from the project task?

Q4 What were the strongest features of this course?

Q5 What were the weakest features of this course?

Q6 How would you like to see this course changed in the future?

There were also some additional questions on the survey where the students could give comments on the lectures, assignments, and exercises, and also, they were encouraged to give advices to the teachers. The survey results from the first two questions are given in Table 1.

Regarding the usage of SAGLib, some characteristic positive comments given as answers to the other questions (Q3–Q6) are:

- SAGLib was fun. Even something as simple as incrementing a variable becomes fun if something is happening on the screen.
- SAGLib has been working very well. Good structure and childishly simple to use.
- It's fun with graphics because of the visible results one gets.
- SAGLib is user friendly.
- I think SAGLib is the best I ever have seen.
- It was nice to get graphical results from one's hard work, instead of the usual command prompt.
- SAGLib made it easier so that one could focus on the important things.
- It was a pleasure to work with a rather simple graphical user interface.
- The project was good, making games instead of making programs.

Many of these comments were repeated with some variations by other students. From all the given comments, it was clear that a strong majority of the students expressed that the usage of SAGLib benefitted the course and their own learning experience in one way or another. To exemplify how fond of the library some of the students have become, we can mention that several students have come visiting us after the course asking if they are allowed to use the library for making their own games at home.

Over these years, only a handful of students have expressed some negative things or feelings about SAGLib. As the following comments show, some of the more skilled students wanted to use more advanced graphics APIs with more features. Unfortunately, some students also ran into problems in their project because of some bugs that appeared in the graphics library, which we could feel made them dislike the library. Here are some characteristic criticisms which were expressed in the surveys:

- SAGLib doesn't feel perfect. Some features are missing.
- I don't understand why we must use SAGLib when there are better well-documented alternatives.
- The graphics becomes slow if one uses many and relatively large images with SAGLib.
- SAGLib was buggy.

The course evaluations, together with our own experiences from interacting with the students during these courses, have made it evident for us that the usage of SAGLib has improved our introductory C++ course. In particular, we could see that the combination of take-home assignments and usage of SAGLib that we used was highly appreciated. It was a strong correlation between the students that finished their take-home assignments and the students that finished their projects on time. Also, at the end of the courses, a final written exam was given, which showed that most students

Figure 7: *Screen shots from four different student projects.*

were able to use their acquired skills, from the take-home assignments and their projects, to solve relevant programming problems satisfactorily.

Over the years we have tried to use OWL (Borland's Graphics Library), MFC and straight WIN32 programming for the final project in our introductory C++ course. All of these libraries required that the student more or less got a finished application to modify. This was perceived to be rather difficult and there was a lot to learn about all classes in respective library. Much time was consumed by trying to use the wrong class in the wrong context. For example, to use transparent bitmaps, students were forced to write or cut and paste code they did not fully understand. The supposed to be fun event driven graphics programming and C++ design were obscured by these frameworks. With SAGLib, on the other hand, there are few functions and classes to master, which means that the students were able to concentrate on solving the intended problem. Since they could concentrate on design and C++ issues, we could see how the quality of their solutions was increased significantly compared to previous years.

We would also like to point out that today's students are well aware of the advanced graphical user interfaces that most applications have in modern windowing systems. If they want to learn programming, but are only allowed to create programs using simple textual input and output they might get discouraged. Furthermore, the usage of graphics in programming might help the instructor in creating a feeling among the students that programming is fun. This can raise the students interest and enthusiasm and make them work harder and learn more. Another benefit of using a graphics system in learning programming is the direct visual feedback you can get as a result of your programming, which may be very helpful when debugging. Another possibility is to use graphics to visualize how algorithms work. Special code for algorithm animation can be provided by the instructor or the students themselves can produce an algorithm animation as a programming exercise.

A possible drawback of using a graphics library in introductory programming courses would be that learning to use the library could consume too much time, and so forcing the instructor to remove other important concepts or exercises from the course. It is therefore important that the graphics system is simple to use and that it provides a fruitful base for learning the programming abilities the course is meant to teach. It is the task of the library to make the programming work easier, and a more joyful experience, not to lay an additional burden upon the students. We have found that SAGLib fulfills these goals extremely well.

Some faculty may also argue that it is better to focus on using functions and classes that are supported in international standards than on using a non-standardized graphics API in introductory courses. Many programming principles and solutions, however, may very well be illustrated by considering the design of the graphics library itself. Also, programmers in industry use different frameworks, toolkits and libraries on a daily basis [Rasala 2000].

Regarding survey question Q6 about future changes for the course, a student wrote: "Don't force everyone to make games". It is true that the library mainly has been used for simple game programming in our courses, but nothing really hinders the library to be used outside this context to be more appealing for a broader category of students. Therefore, we are planning to emphasize the open-ended nature of the programming project more in the future to make it clear that students really have the opportunity to be creative also outside the game programming context.

The good project results we have observed can partly be explained by the fact that the presented course was not the students very first programming course. They were novice C++ programmers, but they had taken a first course in an imperative programming language previously. Therefore, it would also be interesting to try using SAGlib in a course where the students have no earlier programming experience at all. Maybe a somewhat simpler project would be required in such a case.

Interestingly, we have been told that SAGLib already has been used in some programming projects in upper secondary school in cooperation with our university. Also in this case, simple games with 2D graphics, e.g. Tetris, were implemented successfully, and the pupils were reported to be satisfied with the library [Nolte 2006].

Before the pupils entered the project, however, they have already been briefly introduced to rudimentary programming of text-based applications at their school.

6 Summary and Conclusions

It is true that there are many difficulties students encounter while learning basic programming skills [Jenkins 2002]. We believe, however, that if students find the programming assignments inspiring, it can motivate them to overcome many of the arising problems.

To be successful, instructors need to consider the motivating factors that drive the students. In fact, many students in computer science find computer graphics to be one of the most interesting subjects in the curriculum [Carlisle 1999]. Also, applications in modern operating systems are expected to have sophisticated graphical user interfaces. Thus, it makes sense to allow the students to learn programming by using a simple graphics API.

When we used our simple accelerated graphics library in our introductory programming courses, we experienced very encouraging results. By providing a simple way to create graphics applications many students show more interest and enthusiasm in the programming exercises. In particular, we have found that the possibility to create arcade-style video games, with fast, flicker-free sprite animation, through hardware accelerated graphics attracts many students.

Of course, incorporating a graphics library into basic programming courses must be done in a careful and proper way to ensure that usage of graphics enrich the learning experience. It is important to note that the usage of the library must not introduce extra code complexity that risks to take the students focus away from the basic programming skills they try to learn. Because of the simplicity of our library, however, we have not noticed any such negative effects.

References

ASTRACHAN, O., AND RODGER, S. H. 1998. Animation, visualization, and interaction in CS 1 assignments. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 317–321.

CARLISLE, M. C. 1999. Graphics for free. *ACM SIGCSE Bulletin 31*, 2, 65–68.

CHILDERS, B., COHOON, J., DAVIDSON, J., AND VALLE, P., 1998. The design of EzWindows: A graphics API for an introductory programming course.

COOPER, S., DANN, W., AND PAUSCH, R. 2000. Alice: a 3-D tool for introductory programming concepts. In *CCSC '00: Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges*, Consortium for Computing Sciences in Colleges, USA, 107–116.

GUZDIAL, M., AND SOLOWAY, E. 2002. Teaching the Nintendo generation to program. *Commun. ACM 45*, 4, 17–21.

GUZDIAL, M. 2003. A media computation course for non-majors. In *ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education*, ACM Press, New York, NY, USA, 104–108.

JENKINS, T. 2001. The motivation of students of programming. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, ACM Press, New York, NY, USA, 53–56.

JENKINS, T. 2002. On the difficulty of learning to program. In *Proceedings of the 3rd annual LTSN-ICS Conference*, 53–58.

KELLEHER, C., AND PAUSCH, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv. 37*, 2, 83–137.

KILGARD, M. J., 1996. The OpenGL Utility Toolkit (GLUT) programming interface, API version 3, November.

LARSSON, T., AND FLEMSTRÖM, D., 2006. SAGLib: Simple Accelerated Graphics Library. http://www.idt.mdh.se/SAGLib.

LAWHEAD, P. B., DUNCAN, M. E., BLAND, C. G., GOLDWEBER, M., SCHEP, M., BARNES, D. J., AND HOLLINGSWORTH, R. G. 2003. A road map for teaching introductory programming using LEGO mindstorms robots. *SIGCSE Bull. 35*, 2, 191–201.

MEYER, B. 2003. The outside-in method of teaching introductory programming. In *Ershov Memorial Conference, volume 2890 of Lecture Notes in Computer Science*, M. Broy and A. V. Zamulin, Eds. Springer-Verlag, 66–78.

NOLTE, T., 2006. Personal communication.

PEDRONI, M., AND MEYER, B. 2006. The inverted curriculum in practice. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, ACM Press, 481–485.

RASALA, R. 2000. Toolkits in first year computer science: a pedagogical imperative. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 185–191.

ROBERTS, E. S. 1995. A C-based graphics library for CS1. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 163–167.

TEW, A. E., FOWLER, C., AND GUZDIAL, M. 2005. Tracking an innovation in introductory CS education from a research university to a two-year college. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 416–420.

WOLZ, U., DOMEN, D., AND MCAULIFFE, M. 1997. Multimedia integrated into CS 2: an interactive children's story as a unifying class project. In *ITiCSE '97: Proceedings of the 2nd conference on Integrating technology into computer science education*, ACM Press, New York, NY, USA, 103–110.