Research Article

Development of the Object-Oriented Dynamic Simulation Models Using Visual C++ Freeware

Alexander I. Kozynchenko¹ and Sergey A. Kozynchenko²

¹*Finnstastigen 84, Bro, 197 35 Stockholm, Sweden*

²*Faculty of Applied Mathematics and Control Processes, St. Petersburg State University, Universitetsky Prospekt 35, Petergof, St. Petersburg 198504, Russia*

Correspondence should be addressed to Alexander I. Kozynchenko; alexander_kozynchenko@yahoo.se

Received 22 September 2015; Accepted 14 December 2015

Academic Editor: Luis Carlos Rabelo

Copyright © 2016 A. I. Kozynchenko and S. A. Kozynchenko. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The paper mostly focuses on the methodological and programming aspects of developing a versatile desktop framework to provide the available basis for the high-performance simulation of dynamical models of different kinds and for diverse applications. So the paper gives some basic structure for creating a dynamical simulation model in C++ which is built on the Win32 platform with an interactive multiwindow interface and uses the lightweight Visual C++ Express as a free integrated development environment. The resultant simulation framework could be a more acceptable alternative to other solutions developed on the basis of commercial tools like Borland C++ or Visual C++ Professional, not to mention the domain specific languages and more specialized ready-made software such as Matlab, Simulink, and Modelica. This approach seems to be justified in the case of complex research object-oriented dynamical models having nonstandard structure, relationships, algorithms, and solvers, as it allows developing solutions of high flexibility. The essence of the model framework is shown using a case study of simulation of moving charged particles in the electrostatic field. The simulation model possesses the necessary visualization and control features such as an interactive input, real time graphical and text output, start, stop, and rate control.

1. Introduction

This paper is primarily intended for the researchers who focus on the development of scientific continuous and hybrid simulation models of dynamical systems and have programming skills in C++ and Windows. The dynamical systems evolving with the flow of time are described by sets of differential equations, ordinary or partial ones. Such systems are the main part of diverse modelling applications belonging to different scientific and engineering domains as high-energy and accelerator physics (see, e.g., Hajari et al. [1]), ship navigation (Sandaruwan et al. [2]), electric power systems (Yusop et al. [3]), mechatronics and robotics (Ferretti et al. [4]), aerospace (Kozynchenko [5]), process control (Sivakumaran and Radhakrishnan [6]), and so forth. These applications, being characterized by advanced theoretical level of underlying mathematical models, can be

multidomain and have rather complex structures including frequent decision-making and intelligent control units.

The problem of choice of programming language and simulation environment which could be most suitable for developing such complex and unconventional models from the viewpoint of their universality, flexibility, effectiveness, and interactive features comes to the fore. There exists specialized ready-made software, such as Maple, Mathematica, MathCAD, and above-mentioned Matlab and Simulink, which provides a range of opportunities sufficient for a number of applications of average complexity, especially for the models consisting of standard subcomponents. On the other hand, a number of domain-specific languages have been developed for the simulation purposes, such as the one described by Ewald and Uhrmacher [7]. However, it seems rather difficult to find the adequate tools for more complicated models. Even the use of the *declarative* languages

like Modelica [8] is also questionable in this case, because the solution based on *imperative* (and compiled) languages, the C++ first of all, usually provides better effectiveness and flexibility. An example of applying the imperative language Object Pascal (Delphi) to the problem of simulation and optimization of the particle beam emitter has been shown by Kozynchenko and Svistunov [9]. However, the rapid development environments like Delphi or its counterpart C++ Builder are proprietary software, whereas the use of open or free tools would be more suitable in case of lack of funding.

It should be noted that the problem of numerical simulation and visualization using Microsoft Visual C++.NET (which is a proprietary software) has been considered by Salleh et al. [10, 11]. The main feature of the approach discussed in the book is the support of MFC (Microsoft Foundation Classes) library in developing the graphical user interface applications. However, the essential issue of simulation of dynamical systems with the real time graphical output and simulation rate control is not addressed in the book.

To sum up, we can say that the problem of developing an appropriate flexible interactive graphical interface and framework for complex object-oriented dynamical simulation models still remains actual. This interface and framework should provide the following desirable features and properties for the dynamical simulation models focused on complex scientific and engineering problems:

- (1) Usage of efficient (presumably compiled), flexible, and widespread programming language, as well as a mainstream integrated development environment (IDE).
- (2) Ability to draw graphs in progress, tracing out the change of state variables and other parameters with respect to the simulation time.
- (3) Ability to interrupt the program execution, to resume and continue it, and to change the simulation rate over the time.
- (4) Ability to make required inputs in the course of simulation, using several windows if necessary.
- (5) Ability to display the selected numerical output in progress and simultaneously with displaying the graphical output.
- (6) Usage of open-source and/or nonproprietary software.

In the paper, we try to show an example of a simulation framework that possesses the above features. The modelling application is taken from electrodynamics domain and describes the motion of charged particles in the field of an electrostatic lens. The model is developed in the standard unmanaged C++ using the Visual C++ Express 2010 Edition as a development environment. When creating a new project, the “Empty Project” template has been selected for simplicity, because it only fixes the file structure and, for the rest, allows the developer to create the project of own type. One can choose the “Win32 Project” template as well, and on

the “Application settings” page the option “Empty project” should be selected. We focus on a Win32-based application, that is, an application that handles Windows messages directly using message loops. The pure Win32 C++ code is used to create the project from scratch. Such an alternative as writing a code with the help of the MFC library is not available, because the MFC is not supported in the Express Edition. The designed program relates to so-called dialog-based applications which use a dialog box as a main window. The project consists of the set of header files, *field.h*, *particle.h*, *rk4_4eq.h*, and *resource.h*, two resource files, *script.rc* and *script2.rc*, and a source file, *main.cpp*.

2. Structure of the Simulation Model

First of all, the general outline of the simulation model to be studied is provided below. The diagram shown in Figure 1 displays the class and function relationships and is helpful in introducing the subject. The model has an ordinary structure of the desktop dialog-based Win32 two-threaded programs and includes the following global functions and classes:

- (i) The function *WinMain* as an entry point at the main thread.
- (ii) The application-defined callback functions *ModalDlgProc* and *ModelessDlgProc* that process the messages in the modal and modeless dialog boxes.
- (iii) The application-defined callback function *ThreadProc* that is executed by the second thread and performs the application-specific algorithms.
- (iv) The application classes *Field* and *Particle* describing the motion of a charged particle in the field of the electrostatic immersion lens.
- (v) The function template *RK4_4eq* which defines a Runge-Kutta solver carrying out one integration step for a set of ordinary differential equations (ODEs).
- (vi) The helper function *DrawEquipotentialLines* used for the field visualization.
- (vii) The STL vector container *vpp* to keep the *Particle*’s objects.

Again, the model includes also global variables *::pause* and *::sleep* to control the simulation process and two global variables *::particleListBox* and *::particleModelessBox* which denote the particles whose data is to be shown at the list box and the modeless dialog box correspondingly. For the sake of completeness, the other four Windows functions are shown in the diagram: *DialogBox*, *CreateDialog*, *CreateThread*, and *InvalidRect*. The Windows functions marked at the diagram with the sign “Win32” are placed in the Windows-specific header file *<windows.h>* that adds the Win32 API to the project. This diagram does not follow the UML (Unified Modelling Language) standards completely because we tried to show the chains of function calls in one diagram too, not only the class relationships. The description of the model presented below is illustrated by a number of listings shown in Algorithms 2–9 and in Appendix. For simplicity, the code

FIGURE 1: The block diagram showing relationships between classes, objects, functions, and global variables being included in the simulation model.

fragments do not contain some features being important but not affecting the program logic, like exception handling, **private** class sections, and *set* and *get* functions.

3. Description of the Modeling Application

Below we consider in more detail a physical model to be simulated. This is a two-dimensional model that includes a number of hydrogen nuclei (protons) moving in the axial-symmetric electrostatic field of the immersion lens (see, e.g., Humphries Jr. [12], §6.6). The model consists of two classes, *Field* and *Particle*, whose objects are created in the program.

3.1. Class Field. The electrostatic field is formed by two coaxial cylindrical electrodes having the potentials 0 and -1000 V. The inner diameter and the length of each electrode are 0.06 m and 0.1144 m correspondingly, and they are separated by the gap of 0.0112 m long. On the axis of the left electrode, the needle-like particle emitter of 0.063 m long is situated. The middle part of the immersion lens is shown in the screenshot in Figure 2(a). The field of the lens is described in a class *Field* which is defined in the header file *field.h*. The class includes a two-dimensional array u of the field potentials at the nodes of the square grid with the spacing $h = 0.003$ m plotted at the plane yz , the cross section, through

FIGURE 2: Screenshot of the graphic user interface (a) and a magnified view of the part of modal dialog box with control items (b).

the diameter, of the immersion lens. These potentials are calculated by solving the Laplace equation with the use of the finite difference method and Liebmann's iteration process (see, e.g., Veerarajan and Ramachandran [13], Chapter 11). The class *Field* contains also arrays of pairs (y_i, z_i) of the coordinates of the i th point belonging to one of the equipotential lines to be plotted for potentials $-10, -25, \dots, -990$ V. A data member *zMax* being equal to the length of the immersion lens sets the maximal distance of particle motion along z -axis. This variable is used in the second thread in the function *ThreadProc* as a termination condition of simulation for the given particle (see Algorithm 9, line 21.9).

3.2. *Class Particle.* Another application-related class is the class *Particle* defined in the header file *particle.h* (Algorithm 2). The class contains all variables and constants being necessary for calculating the motion of a proton in the electrostatic field. These are time t , coordinates y , z , velocity v and its components v_y and v_z , kinetic energy W , proton mass m and charge q presented as **static** constants, components of electrostatic force f_y , f_z , the dimensions of the lens (*lengthOfElectrode* and *lengthOfGap*), a set of variables (such as *uLeft* and *yCellTop*) used in calculating the forces in the helper function *FzFy*, a field grid spacing $dy = dz$, and a control variable *kVy* which is used in

the *Particle*'s constructor to set the initial values of velocity components v_y and v_z under the given energy W_0 by the formulas:

$$\begin{aligned} v &= \sqrt{\frac{2W_0 \cdot 1.6022 \cdot 10^{-19}}{m}}, \\ v_z &= \frac{v}{\sqrt{1 + k_{vy}^2}}, \\ v_y &= k_{vy} v_z. \end{aligned} \quad (1)$$

The Boolean flag variable *stop* initialized as **false** shows whether the particle moves inside the lens; it becomes **true** when it reaches the rightmost edge of the lens at $z_i \geq \text{Max}$ and so it falls out of the further simulation. Additionally, the class includes a nested structure *Data*, objects of which keep the information about the current particle's state and are placed in the STL *vector* container *history* for holding the particle's state during the simulation. The data members are initialized in the combined (i.e., both default and parameterized) constructor. Other member functions are four right-hand sides of the ODE's set: $DzDt$, $DyDt$, $DvzDt$, and $DvyDt$. The class also contains a helper function $FzFy$ computing the values of electrostatic forces f_z and f_y used in the member functions $DvzDt$ and $DvyDt$.

4. Description of the Two-Threaded Simulation Engine

Algorithm 3 lists the source file *main.cpp* that contains the function *WinMain*, as well as a set of global variables, constants, and function declarations (lines 10.3–27.3). Among them there are handles (descriptors) to dialog boxes *hModalDlg* and *hModelessDlg*, a handle *hThread* to a thread which carries out the model computations, control variables *pause* and *sleep* responsible for interrupting, resuming, and changing the rate of the simulation, constants *height* and *width* that determine the size of a plot rectangle where the particle trajectories, electrodes, and the field of the lens will be drawn, and scale factors *my*, *mz*, and *kzy* used in displaying the field. Also noteworthy is an STL *vector* container *vpp* which is used for keeping a set of *Particle*'s objects taking part in the simulation. A global variable *particleModelessBox* is introduced to provide the particle's data numerical output at the modeless dialog window, along with a global variable *particleListBox* that indicates the selected particle whose numerical data is expected to be displayed in the list box placed at the modal dialog window.

The function *WinMain* first creates a dynamically allocated object of the class *Field*, initializing the static pointer *Particle::pF* (line 31.3). Simultaneously, the electrostatic field is computed in the invoked constructor of the class *Field*. Furthermore, *WinMain* creates a modal dialog box from the dialog box template resource, using the function *DialogBox*. The corresponding resource-definition script (file *script.rc*) is listed in Algorithm 4. It contains most of resources relating to the modal dialog box and forming the graphical user interface (GUI), such as Windows controls (child windows): buttons,

edit controls, text controls, a list box, and a trackbar. In Figures 2(a) and 2(b), a screenshot of the dialog boxes in the process of simulation is shown (a) together with a part of the modal dialog box with the most important GUI controls and associated resources' IDs (b). The function *WinMain* includes also a message loop that retrieves the Windows messages from the message queue (lines 33.3–38.3). If the messages are intended for the *modeless* dialog box, they should not be processed at the program's message loop. In this case, the function *IsDialogMessage* sends the messages directly to the modeless dialog box avoiding the program's message loop, as is done using the **if** statement in line 34.3.

4.1. Description of the Modal Dialog Procedure. Next we describe the modal dialog box window procedure *ModalDlgProc*; the definition of that is listed in Algorithms 5–8. The function has the following heading:

```
int CALLBACK ModalDlgProc(HWND hwnd,
UINT message, WPARAM wParam, LPARAM lParam
).
```

The function's body begins with definitions of the variables used in designing the graphical interface. These are a handle *hDC* to the device context, a structure defining a set of graphic objects, with some of them also defined here (pens, brushes, and fonts), a variable *psPaint* of type *PAINTSTRUCT* keeping the information necessary for drawing in a client area of the modal dialog box, a variable *rect* of type *RECT* used to keep coordinates of the client area, and temporary text buffers. It should be noted that most of the variables above must be declared as **static** ones; otherwise the graphical information will not be displayed properly. The basic part of the dialog box procedure is a **switch** statement with the controlling expression. The function is intended to be processing the Windows messages sent to the dialog box and passed through the second formal parameter *message*. The first message is *WM_INITDIALOG* which is sent by the system immediately before displaying the dialog box and carries out the required initializations. In the case of sending this message, the following actions are made:

- (i) Creation of the second thread with the handle *hThread* in the suspended state (lines 27.5–32.5).
- (ii) Initialization of the global handle to the modal dialog box *hModalDlg* (line 33.5) in order to be used further in the thread procedure as a control variable.
- (iii) Creation of the modeless dialog box with the global handle *hModelessDlg*, which can display more detailed console-like output for the particles selected during the simulation (lines 34.5–35.5).
- (iv) Completion of creating an edit box (*IDC_EDIT.ParticleModelessBox*) with an up-down (spin) control (*IDC.UpDown*) which is sometimes referred to as a "spinner" control; this is done by calling the functions *SendMessage* to set the edit box as a buddy one and to set the range and the initial zero value (lines 36.5–40.5).

- (v) An analogous procedure accomplished concerning the trackbar control which provides changing the rate of simulation: the functions *SendMessage* in the lines 48.5–51.5 define the range and set the slider to the initial position between two tick marks; another *SendMessage* function gets the slider's logical position that is assigned to the global variable *sleep* (lines 52.5–53.5).
- (vi) Statements in lines 42.5–47.5 and in lines 54.5–55.5 setting the initial values for all edit windows.

4.1.1. Processing the Control Related Messages WM_COMMAND. Another important case we need to focus on is processing the message WM_COMMAND which is sent when the user clicks a button at the modal dialog box. If the user chooses a button “Create a new particle,” the function *GetDlgItemText* retrieves strings from the corresponding edit windows with control identifiers IDC_EDIT_W0 and IDC_EDIT_kVy to the text buffer (Algorithm 6, lines 8.6 and 10.6). Then the strings are converted into numerical values of the variables *W0* and *kVy* (lines 9.6 and 11.6) and are passed to the *Particle*'s constructor. The latter creates a *Particle*'s object that will be kept in the global STL *vector* container *vpp*:

```
12.6 vpp.push_back(Particle(W0, kVy));
```

The size of this container, that is, the total number *N* of particles involved in the simulation, is then passed to the edit window with an identifier IDS_EDIT_PartNum to be displayed (lines 13.6–14.6).

The considered simulation model allows displaying not only graphs but also numerical data in the console-like fashion. Therefore, after at least one particle is created, the user may choose a particle whose state data could be displayed both at the above-mentioned modeless dialog box with the handle *hModelessDlg* and in the list box placed at the modal dialog box. It is accomplished in the edit box with the up-down control, where the required particle is chosen by the spin button. When the user clicks the button “Confirm,” the numeric input, that is, the particle's number, is retrieved by the function *GetDlgItemInt* and is assigned to the variable *particleModelessBox* (lines 18.6–19.6). The particle chosen for the first time, that is, under the first click of the button “Confirm,” is selected to be displayed in the list box. The number of this particle kept in the variable *particleModelessBox* is assigned to the variable *particleListBox* in line 23.6 (and made visible in line 24.6) which is then used in both threads, remaining unchanged during the simulation. It should be noted that the user can change the particle, whose data are displayed at the modeless dialog box, at any time, using the above procedure.

The modal dialog box contains also two buttons “Run” and “Pause” placing an important part in controlling the simulation process. When the user clicks the button “Run” the program checks if the global variable *pause* equals **true**. If so, the function *ResumeThread* is called and the value of *pause* is reversed to **false** (lines 27.6–31.6). As a result, the hitherto suspended thread with the handle *hThread* is resumed and executes the function *ThreadProc* carrying out the model

calculations. In the case of a need to stop the simulation temporarily, the user may choose the button “Pause.” Since the variable *pause* has the **false** value until this moment, the function *SuspendThread* is invoked, the *pause* reverts to **true** (lines 33.6–37.6), and all calculations in the function *ThreadProc* will stop. And, at last, if the user clicks a “Cancel” button, the dialog box procedure closes the dialog box by calling the *EndDialog* function (line 40.6) as it does in case of processing the WM_CLOSE message (line 2.6). Next up is the case of processing another windows message WM_HSCROLL that is sent if the user wants to change the simulation rate and moves the slider of the trackbar control to the left to increase the rate or to the right to slow it down. The simulation rate is controlled by the global variable *sleep* used further in the function *ThreadProc*. As mentioned above, *sleep* is assigned the value returned by the function *SendMessage* in lines 45.6–46.6. For information, this value is also shown at the edit box with an identifier IDC_EDIT_Sleep (lines 48.6–49.6).

4.1.2. Painting at the Modal Dialog Box: Processing the Message WM_PAINT. Drawing graphical objects at the modal dialog box is carried out when the system generates a message WM_PAINT. It is done either implicitly when the window must be updated after sizing, moving, and so forth, or the application explicitly requests an update, invoking, for example, the function *InvalidateRect*. In our case, these key control actions are accomplished in the function *ThreadProc* (Algorithm 9, lines 36.9–38.9).

The typical processing of the WM_PAINT starts with calling the function *BeginPaint* (Algorithm 6, line 51.6) which passes the painting information to the variable *psPaint* and returns the handle *hDC* to the device context. Then a number of functions are called to prepare the client area at the modal box to draw the interface elements and data (lines 52.6–55.6). When calling the function *GetClientRect*, a rectangle of the client area gets its coordinates; calling the function *SetMapMode* sets the mapping mode between page space and device space. In case of a parameter MM_ISOTROPIC logical units are mapped to device (e.g., display) units with equally scaled axes. The function *SetWindowExtEx* sets the maximum values for the horizontal and vertical axes of the window in logical units, whereas the function *SetViewportExtEx* does the same in device units for the view port, that is, the *device* coordinate system. So, using both functions determines the scaling factor between the window and the view port.

Shown below (Algorithm 7, lines 2.7–5.7) is the fragment that draws the caption of the simulation application “A simulation model of the motion...” Next, in lines 6.7–50.7, the graphic functions are invoked such as *Rectangle*, *MoveToEx*, and *LineTo*. These functions provide drawing a plot rectangle, axes, gridlines, and tick marks (see the screenshot in Figure 2(a)) necessary for displaying electrodes, field lines, particle's trajectories, and so on. The equipotential lines are drawn by calling the helper function *DrawEquipotentialLines* (lines 53.7–54.7). The principal painting, that is, drawing the particles' trajectories in real time, is carried out in the fragments placed between lines 2.8 and 14.8 in Algorithm 8. Both the global vector *vpp* and the *Particle*'s data members, vectors *history* keeping the particles' states, are involved in

the drawing of the trajectories $y(z)$ in the nested **for** loop. The processing of the message WM_PAINT is completed with the fragment written in lines 17.8–21.8 which displays the *vpp*'s index of a particle whose data is to be output at the list box. The final statement of the processing is the call of *EndPaint*.

When the modal dialog box is being destroyed and is removed from the screen, the message WM_DESTROY is sent to the function *ModalDlgProc*. The processing of this message includes the set of function calls of *DeleteObject* to delete the logical pens and the brush.

4.2. Description of the Modeless Dialog Box with the Console-Like Output. The modeless dialog box can be shown even while the modal dialog box exists. Its purpose in this application is to display the numerical data describing the history of the state of a particle having been selected at the edit box with an identifier IDC_EDIT_ParticleModelessBox during the simulation (see Figure 2(b)). The output is carried out by the window function *ModelessDlgProc* in the console-like fashion and requires the vertical and horizontal scrolling to get more display space the modeless box can provide. That is, vertical and horizontal scroll bars are used in the modeless box.

The window function has the similar structure as in case of the modal dialog. It processes the messages in the **switch** selection statement that has eight branches, **case** labels for processing the messages WM_INITDIALOG, WM_VSCROLL, WM_HSCROLL, WM_PAINT, and so on. As the most of the procedure code contains the common, near standard fragments (see, e.g., Petzold [14], Chapter 4, section “Scroll Bars”), the listing of the procedure is placed to Appendix. However, a short description of the procedure is still given below. The resource script placed in the file *script2.rc* and associated with the modeless dialog contains styles WS_VSCROLL and WS_HSCROLL among other ones:

```
STYLE DS_SETFONT | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU | WS_VSCROLL | WS_HSCROLL |
WS_MINIMIZEBOX | WS_OVERLAPPEDWINDOW
```

A number of variables and constants are defined in the window procedure. The constant *NUMLINES* is the maximal number of lines of text involved into the vertical scrolling. The variable *tm* of type TEXTMETRIC defines the font and rows parameters such as width and height of characters and space between characters. The static variables *yPos* and *xPos* are current positions of the scroll bar thumbs for vertical and horizontal scroll bars correspondingly, and *yInc* contains an incremental value for updating *yPos*. The messages WM_VSCROLL and WM_HSCROLL are sent to the window procedure if the scroll events occur either in the window's vertical scroll bar or in the horizontal scroll bar. Processing these messages made within nested **switch** statements (see Appendix, Algorithm 11, lines 46.11–63.11 and 73.11–89.11) with the parameter notification code SB_TOP, SB_BOTTOM, and so forth results in updating *yPos* and *xPos*.

When calling the function *ScrollWindow* in lines 68.11 and 96.11, the system invalidates the client area and generates the WM_PAINT message. One more origin of a WM_PAINT message is the thread procedure, where the function *InvalidateRect*

is called (see Algorithm 9, line 38.9) after completing the nested **for** loop of one step integration of charged particles' motion. The code written in the WM_PAINT selection brunch provides drawing the full set of text lines from the beginning of simulation to the current instant. Each text line outputs the time, position, velocity, and kinetic energy of a particle (i.e., the set of t, y, z, v_y, v_z, v, W).

4.3. Thread Procedure as the Physics Engine. The physics engine of the considered simulation model is placed in the function *ThreadProc*. The function includes the **while** loop (Algorithm 9, lines 4.9–40.9) providing the simulation until the last particle reaches the rightmost edge of the lens. The nested **for** loop (lines 5.9–25.9) carries out one step of integration for all particles which did not reach the right lens edge (i.e., $z_i < zMax$). When a particle gets moved beyond this boundary, its data member *stop* is set to **true**; that is, the particle is excluded from simulation, and the variable *counter* keeping the number of excluded particles is incremented. The integration is carried out by the global function template *RK4_4eq* (lines 7.9–10.9). This function is a solver that processes one step of integration of the system of four ODEs by the Runge-Kutta method of the 4th order. The function template is defined in the header file *rk4_eq.h* and has the prototype as shown in Algorithm 1.

As we can see in the prototype shown in Algorithm 1, the function template imposes some restrictions on the content of the application class *T*; namely, the class needs to have five data members denoting both an independent variable (time t) and four state variables (y, z, v_y , and v_z), a helper function (pointed as *pPrecompute*), and four member functions for computing the right-hand sides of the corresponding differential equations. The helper function makes the calculations common for right-hand sides, thus reducing the run time. The class *Particle* is designed so that these requirements are met.

After an integration step has been completed, the updated values of the state variables are pushed into the *Particle*'s vector container *history* (lines 11.9–20.9). When the **for** loop is completed, the messages are generated to be passed both to the modal dialog box (including the list box) and to the modeless box. In particular, the functions *InvalidateRect* are invoked generating the messages WM_PAINT. As a result, the numerical outputs are updated. When sending messages to the list box via calling the functions *SendMessage* in lines 31.9–34.9, the message LB_ADDSTRING results in adding the new text line to the end of the list, and the message LB_SETCURSOR ensures that the end of the list is visible. The call of the function *Sleep* allows setting the desired simulation rate kept in the variable *sleep*. Then the **while** loop starts the next iteration.

5. Some Simulation Results and Discussion

The considered object-oriented simulation model of particles motion in electric field combined with Win32 dialog-based graphical interface and framework has been tested within Visual C++ 2010 Express environment for different cases to estimate the performance of the simulation. The project

```

template<class T>
inline void RK4_4eq(T* p, //pointer to an object of the class T
    double h, //time step of integration ( $h$  in the class Particle)
    double T::*pX0, // pointer to the independent variable - time  $t$ 
    double T::*pX1, // pointer to the state variable - velocity  $v_z$ 
    double T::*pX2, // pointer to the state variable - velocity  $v_y$ 
    double T::*pX3, // pointer to the state variable - coordinate  $z$ 
    double T::*pX4, // pointer to the state variable - coordinate  $y$ 
    void (T::*pPrecompute)(), //pointer to the helper function  $FzFy$ 
    double (T::*pF1)(), //pointer to the equation of motion  $Dv_zDt$ 
    double (T::*pF2)(), //pointer to the equation of motion  $Dv_yDt$ 
    double (T::*pF3)(), //pointer to the equation of motion  $DzDt$ 
    double (T::*pF4)() //pointer to the equation of motion  $DyDt$ 
);

```

ALGORITHM 1

```

1.2 #pragma once
2.2 #include"field.h"
3.2 #include"rk4_4eq.h"
4.2 #include<vector>
5.2
6.2 class Particle{
7.2     public:
8.2         static const double m, q; //mass and charge of a particle
9.2         double W0, W, v;
10.2         double t, z, y, vz, vy;
11.2         struct Data{
12.2             double t, z, y, vz, vy, v, W;
13.2         } currentData;
14.2         double fz, fy; //components of an electric force
15.2         double dz, dy; //spacing of the field grid along z and y axes
16.2         static Field* pF;
17.2         double uLeft, uRight, uTop, uBottom;
18.2         double yCellTop, yCellBottom, zCellLeft, zCellRight;
19.2         int iTopLeft, jTopLeft;
20.2         vector<Data> history;
21.2         double kVy;
22.2         bool stop;
23.2         Particle(double W0 = 25., double kVy = 0.): W0(W0), W(W0),
24.2             t(0.), z(0.), y(0.), kVy(kVy), fz(0.), fy(0.),
25.2             v(sqrt(2./m*W0*1.6022e-19)),
26.2             vz(v/sqrt(1.+kVy*kVy)), vy(kVy*vz), stop(false){
27.2             dy = dz = pF->h;
28.2             zCellLeft = zCellRight = 0.;
29.2             if(y == 0)
30.2                 yCellTop = yCellBottom = 0.;
31.2         }
32.2         void FzFy(); //computing the force components fy and fz
33.2         double DzDt(){ return vz;}
34.2         double DyDt(){ return vy;}
35.2         double DvzDt(){ return 1./m*fz;}
36.2         double DvyDt(){ return 1./m*fy;}
37.2 };
38.2 const double Particle::m = 1.6726e-27; //kg
39.2 const double Particle::q = 1.6022e-19; //C (Coulomb)
40.2 Field* Particle::pF = NULL;

```

ALGORITHM 2: The header file *particle.h* defining the class *Particle*.


```

1.3 #include<windows.h>
2.3 #include<iostream>
3.3 #include"resource.h"
4.3 #include"particle.h"
5.3 #include"field.h"
6.3 #include<CommCtrl.h>
7.3 #pragma comment(lib, "comctl32.lib")
8.3 using namespace std;
9.3
10.3 HWND hModalDlg, hModelessDlg = 0;
11.3 INT_PTR nResult;
12.3 HANDLE hThread;
13.3 bool pause = true; // a control to make the pause in execution
14.3 int sleep = 0; //a control affecting the rate of running
15.3 const double my = 200./0.06, mz = my, kzy = 0.004;
16.3 const int height = 200, height2 = (int)(height*0.5),
17.3                               width=(int)(height*28./15.);
18.3 const int zLeft = 80, yTop = 70; //coordinates of the drawing area
19.3 vector<Particle> vpp; //container keeping the set of particles
20.3 int particleModelessBox = -1;
21.3 int particleListBox = -1;
22.3 char bufListBox[180];
23.3
24.3 int CALLBACK ModalDlgProc(HWND, UINT, WPARAM, LPARAM);
25.3 int CALLBACK ModelessDlgProc(HWND, UINT, WPARAM, LPARAM);
26.3 DWORD CALLBACK ThreadProc(LPVOID);
27.3 void DrawEquipotentialLines(HDC, HFONT, LOGFONT, HPEN, Field*);
28.3
29.3 int WINAPI WinMain(HINSTANCE hI, HINSTANCE hi2, LPSTR p, int n){
30.3     MSG msg;
31.3     Particle::pF = new Field();
32.3     nResult = DialogBox(NULL, "MYDIALOG", 0, (DLGPROC)ModalDlgProc);
33.3     while(GetMessage(&msg, NULL, 0, 0) > 0){
34.3         if(hModelessDlg == 0 || !IsDialogMessage(hModelessDialog, &msg)){
35.3             TranslateMessage(&msg);
36.3             DispatchMessage(&msg);
37.3         }
38.3     }
39.3     delete Particle::pF;
40.3     return msg.wParam;
41.3 }

```

ALGORITHM 3: The source file *main.cpp* containing the entry point to the program: function *WinMain*.

was built in *release* configuration, so the program is fully optimized. The simulation is carried out on an AMD Athlon™ 64 × 2 Dual Core 2.40 GHz processor. With the aim of getting the CPU (central processing unit) time that elapses when executing the thread procedure *ThreadProc* (physics engine), the function *clock* defined in the header file *<ctime>* has been used. The time step of integration of the particle motion equations is $h_t = 10^{-9}$ sec. Three cases of simulation were considered, which have different sets of output facilities. The first case covers the program version with the graphical output only, the second one adds the text output to the modeless box, and the third one provides also the text output to the list box. In all cases the time delay is set to zero. The results obtained for the cases depending on the sizes of the particle ensemble are presented in Table 1. Each value is an average of a small sample formed as a result of 5–7 runs.

TABLE 1: The elapsed CPU time required for simulating the motion of an ensemble of particles in the thread function *ThreadProc* (no sleep time).

(a) Graphical output only					
Number of particles	1	10	100	1000	10000
CPU time, sec	0.016	0.041	0.122	1.125	10.32
(b) Graphical output and text output to the modeless box					
Number of particles	1	10	100	200	500
CPU time, sec	0.018	0.044	0.172	0.262	0.634
(c) Graphical output and text output both to the modeless box and to the list box					
Number of particles	1	10	100	200	500
CPU time, sec	9.255	34.19	264.2	513.3	1254

```

1.4 #include<windows.h>
2.4 #include"resource.h"
3.4
4.4 MYDIALOG DIALOGEX 0, 0, zW, yW
5.4 STYLE DS_MODALFRAME|WS_CAPTION|WS_SYSMENU| WS_MINIMIZEBOX
6.4 CAPTION "MyDialogWindow"
7.4 FONT 8, "MS Shell Dlg"
8.4 {
9.4     PUSHBUTTON "Create a new particle", IDB_NewParticle,
10.4         210, 363, 70, 14
11.4     PUSHBUTTON "Confirm", IDB_Confirm, 310, 400, 50, 14
12.4     PUSHBUTTON "Run", IDB_Run, 50, 430, 50, 14
13.4     PUSHBUTTON "Pause", IDB_Pause, 135, 430, 50, 14
14.4     PUSHBUTTON "Cancel", IDB_Cancel 220, 430, 50, 14
15.4     EDITTEXT IDC_EDIT_W0 150, 353, 40, 15, ES_CENTER
16.4     EDITTEXT IDC_EDIT_kVy 150, 375, 40, 15, ES_CENTER
17.4     EDITTEXT IDC_EDIT_PartNum 390,363,25,15, ES_CENTER|ES_READONLY
18.4     EDITTEXT IDC_EDIT_ParticleToModelessBox 270,400,30,15,ES_CENTER
19.4     EDITTEXT IDC_EDIT_Sleep 570,400,40,15, ES_CENTER|ES_READONLY
20.4     LISTBOX IDC_ListBox 550, 70, 200, 250, WS_VSCROLL
21.4     CONTROL "" IDC_UpDown, "msctls_updown32",
22.4         UDS_SETBUDDYINT|UDS_ALIGNRIGHT, 0, 0, 0, 0
23.4     CONTROL "" IDC_TrackBar, "msctls_trackbar32", WS_VISIBLE|
24.4         WS_TABSTOP|TBS_HORZ, 400, 400, 150, 30
25.4     LTEXT "Set initial data\nfor a new particle:", 99, 50,360,65,20
26.4     LTEXT "W0 = ", 100, 118, 356, 30, 10
27.4     LTEXT "Vy/Vz = ", 101, 118, 378, 30, 10
28.4     LTEXT "Total number N\nof created particles:", 102,
29.4         315, 360, 70, 20
30.4     LTEXT "Indicate the particle ( from 0 to N-1 ) \
31.4         whose data you want to display: ", 103, 50, 400, 220, 10
32.4     LTEXT "Displaying data of the first selected particle:", 104,
33.4         550, 55, 160, 10
34.4     LTEXT "0 V", 105, 150, 58, 25, 10
35.4     LTEXT "-1000 V", 106, 360, 58, 25, 10
36.4     LTEXT "Simulation speed", 107, 445, 385, 70, 10
37.4     LTEXT "Increase", 108, 410, 420, 35, 10
38.4     LTEXT "Slowdown", 109, 508, 420, 35, 10
39.4     LTEXT "Delay, ms", 110, 570, 385, 40, 10
40.4     LTEXT "z, m", 111, 463, 169, 15, 10
41.4     LTEXT "y, m", 112, 60, 52, 15, 10
42.4     LTEXT "0", 113, 70, 162, 5, 10
43.4     //Two sets of LTEXT controls for displaying the numerical
44.4     //values for both y-gridlines and z-tick marks of the plot.
45.4     //Relating to the y-gridline:
46.4     LTEXT "0.01", 114, 62, 129, 15, 10
47.4     ...
48.4     LTEXT "-0.03", 119, 60, 255, 16, 10
49.4     //Relating to the z-tick marks:
50.4     LTEXT "0.00", 120, 73, 288, 15, 10
51.4     ...
52.4     LTEXT "0.11", 131, 438, 288, 15, 10
53.4 }

```

ALGORITHM 4: The resource script file *script.rc* for the modal dialog.

The results show a high performance of the simulation model in the first (a) and the second (b) cases, even for large particle ensembles (10^4 particles). However, the third case (c), in which the text output is also displayed in the list

box, is characterised by a much lower rate of computation and output. So this output mode should be applied only for applications not having the high requirements to the simulation speed.

```

1.5 int CALLBACK ModalDlgProc(HWND hwnd, UINT message,
2.5                               WPARAM wParam, LPARAM lParam){
3.5     static HDC hDC;
4.5     PAINTSTRUCT psPaint;
5.5     RECT rect;
6.5     static HBRUSH hBlackBrush=(HBRUSH)GetStockObject(BLACK_BRUSH),
7.5     g_hbrBackground = CreateSolidBrush(RGB(255, 255, 255));
8.5     static HPEN hBlackPen = (HPEN)GetStockObject(BLACK_PEN),
9.5     hBlackThickPen = CreatePen(PS_SOLID, 5, RGB(0,0,0)),
10.5    hWhitePen = CreatePen(PS_SOLID, (int)0.5, RGB(255, 255, 255)),
11.5    hGreyPen = CreatePen(PS_SOLID, (int)0.5, RGB(200, 200, 200)),
12.5    hPinkPen = CreatePen(PS_SOLID, (int)0.5, RGB(255, 150, 150)),
13.5    hBluePen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
14.5     static HFONT fontPotential, fontCaption;
15.5     static LOGFONT lf, lf2;
16.5     strcpy_s(lf.lfFaceName, "MS Sans Serif");
17.5     lf.lfHeight = 8;
18.5     strcpy_s(lf2.lfFaceName, "Century");
19.5     lf2.lfHeight = 18;
20.5     lf2.lfWeight = FW_MEDIUM;
21.5     char caption[] = "A simulation model of the motion of charged\
22.5     particles in the electrostatic field of an immersion lens";
23.5     char bufferLB[3], buffer[500];
24.5     Particle temp;
25.5     switch(message){
26.5         case WM_INITDIALOG:
27.5             hThread = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
28.5             &ThreadProc, NULL, CREATE_SUSPENDED, 0);
29.5             if(hThread == NULL){
30.5                 MessageBox(0,"Error Creating Thread","Message",MB_OK);
31.5                 return 1;
32.5             }
33.5             hModalDlg = hwnd;
34.5             hModelessDlg = CreateDialog(0, "MODELESS_DIALOG", 0,
35.5             (DLGPROC)ModelessDlgProc);
36.5             SendMessage(GetDlgItem(hwnd, IDC_UpDown), UDM_SETBUDDY,
37.5             (WPARAM)GetDlgItem(hwnd,IDC_EDIT_ParticleModelessBox), 0);
38.5             SendMessage(GetDlgItem(hwnd,IDC_UpDown),
39.5             UDM_SETRANGE, 0, 100);
40.5             SendMessage(GetDlgItem(hwnd, IDC_UpDown),UDM_SETPOS,0, 0);
41.5             int pLB;
42.5             sprintf_s(buffer, "%.2f", temp.W0);
43.5             SetDlgItemText(hwnd, IDC_EDIT_W0, buffer);
44.5             sprintf_s(buffer, "%.2f", temp.kVy);
45.5             SetDlgItemText(hwnd, IDC_EDIT_kVy, buffer);
46.5             sprintf_s(buffer, "%d", 0);
47.5             SetDlgItemText(hwnd, IDC_EDIT_PartNum, buffer);
48.5             SendMessage(GetDlgItem(hwnd, IDC_Trackbar), TBM_SETRANGE,
49.5             TRUE, MAKELONG(0, 1000));
50.5             SendMessage(GetDlgItem(hwnd, IDC_Trackbar), TBM_SETPOS,
51.5             TRUE, 200);
52.5             sleep = SendMessage(GetDlgItem(hwnd, IDC_Trackbar),
53.5             TBM_GETPOS, 0, 0);
54.5             sprintf_s(buffer, "%d", sleep);
55.5             SetDlgItemText(hwnd, IDC_EDIT_Sleep, buffer);
56.5             break;

```

ALGORITHM 5: Modal dialog box: function *ModalDlgProc* (part 1).

```

1.6 case WM_CLOSE:
2.6     EndDialog(hwnd, nResult);
3.6     break;
4.6 case WM_COMMAND:
5.6     double kVy, W0;
6.6     switch(LOWORD(wParam)){
7.6         case IDB_NewParticle:
8.6             GetDlgItemText(hwnd, IDC_EDIT_W0, buffer, sizeof(buffer));
9.6             W0 = atof(buffer);
10.6            GetDlgItemText(hwnd, IDC_EDIT_kVy, buffer, sizeof(buffer));
11.6            kVy = atof(buffer);
12.6            vpp.push_back(Particle(W0, kVy));
13.6            sprintf_s(buffer, "%d", vpp.size());
14.6            SetDlgItemText(hwnd, IDC_EDIT_PartNum, buffer);
15.6            break;
16.6         case IDB_Confirm: //the button "Confirm" is pressed
17.6             if(vpp.empty()) break;
18.6             particleModelessBox = GetDlgItemInt(hwnd,
19.6                 IDC_EDIT_ParticleModelessBox, NULL, FALSE);
20.6             if(particleModelessBox >= (signed)vpp.size())
21.6                 particleModelessBox = vpp.size() - 1;
22.6             if(particleListBox == -1 ){
23.6                 particleListBox = particleModelessBox;
24.6                 InvalidateRect(hwnd, NULL, FALSE);
25.6             }
26.6             break;
27.6         case IDB_Run: //the button "Run" is pressed
28.6             if(pause == true){
29.6                 ResumeThread(hThread);
30.6                 pause = false;
31.6             }
32.6             break;
33.6         case IDB_Pause: //the button "Pause" is pressed
34.6             if(pause == false){
35.6                 SuspendThread(hThread);
36.6                 pause = true;
37.6             }
38.6             break;
39.6         case IDB_Cancel: //the button "Cancel" is pressed
40.6             EndDialog(hwnd, nResult);
41.6             break;
42.6     }
43.6     break;
44.6 case WM_HSCROLL:
45.6     sleep = SendMessage(GetDlgItem(hwnd, IDC_Trackbar),
46.6                         TBM_GETPOS, 0, 0);
47.6     sprintf_s(buffer, "%3d", sleep);
48.6     SetDlgItemText(hwnd, IDC_EDIT_Sleep, buffer);
49.6     break;
50.6 case WM_PAINT:
51.6     hDC = BeginPaint(hwnd, &psPaint);
52.6     GetClientRect(hwnd, &rect);
53.6     SetMapMode(hDC, MM_ISOTROPIC);
54.6     SetWindowExtEx(hDC, zW, yW, NULL);
55.6     SetViewportExtEx(hDC, rect.right, rect.bottom, NULL);

```

ALGORITHM 6: Modal dialog box: function *ModalDlgProc* (part 2).


```

1.7 // drawing the caption:
2.7     SetBkMode(hDC, TRANSPARENT);
3.7     fontCaption = CreateFontIndirect(&lf2);
4.7     SelectObject(hDC, fontCaption);
5.7     TextOut(hDC, 130, 20, caption, strlen(caption));
6.7 // drawing the plot rectangle:
7.7     SelectObject(hDC, hWhitePen);
8.7     Rectangle(hDC, zLeft, yTop-4, zLeft+width, yTop+height+4);
9.7 // drawing the Z-axis with an arrow head:
10.7    SelectObject(hDC, hBlackPen);
11.7    MoveToEx(hDC, zLeft, yTop + height2, NULL);
12.7    LineTo(hDC, zLeft + width + 15, yTop + height2);
13.7    LineTo(hDC, zLeft + width + 10, yTop + height2 - 2);
14.7    MoveToEx(hDC, zLeft + width + 15, yTop + height2, NULL);
15.7    LineTo(hDC, zLeft + width + 10, yTop + height2 + 2);
16.7 // drawing the horizontal grey gridlines:
17.7    SelectObject(hDC, hGreyPen);
18.7    MoveToEx(hDC, zLeft, yTop + height2*2/3, NULL);
19.7    LineTo(hDC, zLeft + width, yTop + height2*2/3);
20.7    MoveToEx(hDC, zLeft, yTop + height2/3, NULL);
21.7    LineTo(hDC, zLeft + width, yTop + height2/3);
22.7    MoveToEx(hDC, zLeft, yTop + height2*4/3, NULL);
23.7    LineTo(hDC, zLeft + width, yTop + height2*4/3);
24.7    MoveToEx(hDC, zLeft, yTop + height2*5/3, NULL);
25.7    LineTo(hDC, zLeft + width, yTop + height2*5/3);
26.7 // drawing the horizontal black axis with tick marks:
27.7    SelectObject(hDC, hBlackPen);
28.7    MoveToEx(hDC, zLeft, yTop + height + 25, NULL);
29.7    LineTo(hDC, zLeft + width, yTop + height + 25);
30.7 // drawing the y-axis with an arrow head:
31.7    MoveToEx(hDC, zLeft, yTop + height + 25, NULL);
32.7    LineTo(hDC, zLeft, yTop - 15);
33.7    LineTo(hDC, zLeft - 2, yTop - 10);
34.7    MoveToEx(hDC, zLeft, yTop - 15, NULL);
35.7    LineTo(hDC, zLeft + 2, yTop - 10);
36.7 // drawing the vertical grey gridlines:
37.7    SelectObject(hDC, hGreyPen);
38.7    for(int i = 1; i <= 11; ++i){
39.7        MoveToEx(hDC, zLeft+(int)(0.01*mz*i), yTop+height, NULL);
40.7        LineTo(hDC, zLeft + (int)(0.01*mz*i), yTop-1);
41.7    }
42.7 // drawing the electrodes:
43.7    SelectObject(hDC, hBlackBrush);
44.7    Rectangle(hDC, zLeft, yTop-4, zLeft+(int)(12.7*kzy*mz), yTop);
45.7    Rectangle(hDC, zLeft + (int)((28.0-12.7)*kzy*mz), yTop-4,
46.7        zLeft + width, yTop);
47.7    Rectangle(hDC, zLeft, yTop+height, zLeft + (int)(12.7*kzy*mz),
48.7        yTop + height + 4);
49.7    Rectangle(hDC, zLeft+ (int)((28.0-12.7)*kzy*mz), yTop+height,
50.7        zLeft + width, yTop + height + 4);
51.7 // drawing the equipotential lines and labels of potentials:
52.7    SelectObject(hDC, hPinkPen);
53.7    DrawEquipotentialLines(hDC, fontPotential, lf, hBlackPen,
54.7        Particle::pF);

```

ALGORITHM 7: Modal dialog box: function *ModalDlgProc* (part 3).

```

1.8 // drawing the particles' trajectories:
2.8 SelectObject(hDC, hBluePen);
3.8 int z1, z2, y1, y2;
4.8 for(int i = 0; i < (signed)vpp.size(); ++i)
5.8     for(int j=0; j<(signed)(vpp[i].history.size()-1); ++j){
6.8         z1 = (int)(mz*vpp[i].history[j].z) + zLeft;
7.8         y1 = yTop + height2-(int)(my*vpp[i].history[j].y);
8.8         z2 = (int)(mz*vpp[i].history[j+1].z) + zLeft;
9.8         y2 = yTop + height2-(int)(my*vpp[i].history[j+1].y);
10.8        if(y2 > yTop + height)
11.8            y2 = yTop + height;
12.8        MoveToEx(hDC, z1, y1, NULL);
13.8        LineTo(hDC, z2, y2);
14.8    }
15.8 // showing the number of a particle whose data
16.8 // will be displayed in the list box:
17.8 if(particleListBox != -1){
18.8     pLB = sprintf_s(bufferLB, "%d", particleListBox);
19.8     SetBkMode(hDC, TRANSPARENT);
20.8     TextOut(hDC, 720, 60, bufferLB, pLB);
21.8 }
22.8 EndPaint(hwnd, &psPaint);
23.8 break;
24.8 case WM_DESTROY:
25.8     DeleteObject(g_hbrBackground);
26.8     DeleteObject(hBlackBrush);
27.8     DeleteObject(hBlackPen);
28.8     DeleteObject(hPinkPen);
29.8     DeleteObject(hBluePen);
30.8     DeleteObject(hGreyPen);
31.8     DeleteObject(hWhitePen);
32.8     DeleteObject(hBlackThickPen);
33.8     PostQuitMessage(0);
34.8     break;
35.8 }
36.8 return 0;
37.8 }

```

ALGORITHM 8: Modal dialog box: function *ModalDlgProc* (part 4).

In addition to the above-mentioned simulation cases, a comparative evaluation study has been carried out with respect to the performance of a code written in C++ and two other popular object-oriented languages, Java and C#. As a test code, we take the solver for Laplace's equation described shortly in Section 3.1. The desired accuracy in the iterative Liebmann's procedure is set to $\text{eps} = 10^{-7}$. Three projects with identical algorithm coding have been developed: the first is created as a Visual C++ 2010 Express console application, the second is written in Java and run in the Java SE 8 Runtime Environment, and the third is written on the base of the Visual C# 2015 Express as a console application. The C++

and C# projects are built in the release configuration and the Java project, in Run configuration (versus Debug). To determine the execution time in the Java program, we apply the *System.nanoTime* method. As to the C# application, the methods of *Stopwatch* class are used. The execution times obtained as an average of 5–7 runs are as follows: C#, 0.059 sec, Java, 0.036 sec, and C++, 0.016 sec. These results confirm once more such a key advantage of the compiled C++ as the running time efficiency against the widespread alternatives, Java and C#, even provided that the “just-in-time” compilation is applied.

```

1.9 DWORD CALLBACK ThreadProc(LPVOID lpdwThreadParam){
2.9     const double ht = 1.0e-9; //time step of integration, s
3.9     static int count = 0; //number of particles with z > zMax
4.9     while(count < (signed)vpp.size()){
5.9         for(int i = 0; i < (int)vpp.size(); ++i)
6.9             if(vpp[i].stop == false){
7.9                 RK4.4eq<Particle>(&vpp[i], ht, &Particle::t, &Particle::vz,
8.9                     &Particle::vy, &Particle::z, &Particle::y, &Particle::FzFy,
9.9                     &Particle::DvzDt, &Particle::DvyDt, &Particle::DzDt,
10.9                    &Particle::DyDt);
11.9                 vpp[i].v=sqrt(vpp[i].vy*vpp[i].vy + vpp[i].vz*vpp[i].vz);
12.9                 vpp[i].W=0.5*Particle::m*vpp[i].v*vpp[i].v/Particle::q;
13.9                 vpp[i].currentData.t = vpp[i].t;
14.9                 vpp[i].currentData.z = vpp[i].z;
15.9                 vpp[i].currentData.y = vpp[i].y;
16.9                 vpp[i].currentData.vz = vpp[i].vz;
17.9                 vpp[i].currentData.vy = vpp[i].vy;
18.9                 vpp[i].currentData.v = vpp[i].v;
19.9                 vpp[i].currentData.W = vpp[i].W;
20.9                 vpp[i].history.push_back(vpp[i].currentData);
21.9                 if(vpp[i].z >= Particle::pF->zMax){
22.9                     vpp[i].stop = true;
23.9                     ++count;
24.9                 }
25.9             }
26.9         if(!vpp[particleListBox].stop){
27.9             sprintf_s(bufListBox, " t = %.3e  y = %.5f  z = %.5f",
28.9                 vpp[particleListBox].history.back().t,
29.9                 vpp[particleListBox].history.back().y,
30.9                 vpp[particleListBox].history.back().z);
31.9             SendMessage(GetDlgItem(hModalDlg, IDC_ListBox), LB_ADDSTRING,
32.9                 0, (LPARAM)bufListBox);
33.9             SendMessage(GetDlgItem(hModalDlg, IDC_ListBox), LB_SETTOPINDEX,
34.9                 (LPARAM)(vpp[particleListBox].history.size()-1), 0);
35.9         }
36.9         InvalidateRect(hModalDlg, NULL, FALSE);
37.9         if(hModelessDlg != 0)
38.9             InvalidateRect(hModelessDlg, NULL, FALSE);
39.9         Sleep(sleep);
40.9     }
41.9     return 0;
42.9 }

```

ALGORITHM 9: The second thread: function *ThreadProc*.

Summing up this and other advantages of C++, such as greater flexibility through the wide use of pointers and templates and the availability of extensive libraries (Win32 API, STL, Boost, etc.), we have every reason to consider C++ as a premier object-oriented language for scientific applications.

6. Conclusion

The case study of the simulation model considered in the paper provides some general guidelines on how to develop a desktop framework that would have the minimal set of means being sufficient for organizing the simulation process as such,

```

1.10 int CALLBACK ModelessDlgProc(HWND hwnd, UINT message,
2.10                                WPARAM wParam, LPARAM lParam){
3.10     static HDC hDC;
4.10     PAINTSTRUCT ps;
5.10     static HBRUSH g_hbrBackground;
6.10     static int xChar, xCaps, yChar, xClient, yClient, xClientMax;
7.10     static int xPos = 1, xMin = 1, xMax = 500, yPos, yMax;
8.10     int x, y, iPaintBeg, iPaintEnd, yInc, temp, xPosOld;
9.10     static const int NUMLINES = 2000;
10.10    TEXTMETRIC tm;
11.10    static char buf[1000];
12.10    static HFONT font;
13.10    static LOGFONT lf;
14.10    strcpy_s(lf.lfFaceName, "MS Sans Serif");
15.10    lf.lfHeight = 14;
16.10    lf.lfWeight = FW_SEMIBOLD;
17.10    switch(message){
18.10        case WM_INITDIALOG:
19.10            hModelessDlg = hwnd;
20.10            g_hbrBackground = CreateSolidBrush(RGB(255, 255, 255));
21.10            hDC = GetDC(hwnd);
22.10            GetTextMetrics(hDC, &tm);
23.10            xChar = tm.tmAveCharWidth;
24.10            xCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * xChar/2;
25.10            yChar = tm.tmHeight + tm.tmExternalLeading;
26.10            ReleaseDC(hwnd, hDC);
27.10            break;
28.10        case WM_CLOSE:
29.10            DestroyWindow(hwnd);
30.10            hModelessDlg = 0;
31.10            break;
32.10        case WM_CTLCOLORDLG:
33.10            return (LONG)g_hbrBackground;
34.10        case WM_SIZE:
35.10            yClient = HIWORD(lParam); //retrieving the dimensions of
36.10            xClient = LOWORD(lParam); // the client area
37.10            yMax = max(0, NUMLINES + 2 - yClient/yChar);
38.10            yPos = min(yPos, yMax);
39.10            SetScrollRange(hwnd, SB_VERT, 0, NUMLINES, FALSE);
40.10            SetScrollPos(hwnd, SB_VERT, yPos, TRUE);
41.10            xPos = min(xPos, xMax);
42.10            SetScrollRange(hwnd, SB_HORZ, xMin, xMax, TRUE);
43.10            SetScrollPos(hwnd, SB_HORZ, xPos, TRUE);
44.10            break;

```

ALGORITHM 10: Modeless dialog box: function *ModelessDialogProc* (part 1).

as well as the online input and graphical and numerical output. The important feature of this framework as compared to alternative solutions is that it uses the free Win32-based platforms from Microsoft, such as Visual C++ Express 2010 Edition. This widespread IDE allows developing effective desktop interactive simulation models of complex dynamical systems by utilizing the basic Win32 API functions and standard unmanaged compiled C++. The main advantages of this versatile approach are the flexibility and run time efficiency of the solutions developed. Indeed, domain neutral simulator systems based on the usage of C++, the object-oriented, imperative, and compiled language, should run faster than

simulators which use, for example, also object-oriented and imperative, but interpreted, languages like Java (see, e.g., Praehofer and Schoeppl [15]). The caveat connected with this approach is that a researcher involved in the simulation modelling has to possess a strong proficiency in C++ and Win32 API.

It should also be noted that the considered multiwindows desktop framework may apparently have the possibilities to be used as a basis for more complex and expanded dynamical models with other powerful features, for example, double buffering, to prevent an excessive flickering and to speed up the redraw, parallel computing using MPI (Message Passing


```

45.11 case WM_VSCROLL:
46.11     switch(LOWORD(wParam)){
47.11         case SB_TOP: //the HOME keyboard key is clicked
48.11             yInc = -yPos; break;
49.11         case SB_BOTTOM: //the END keyboard key is clicked
50.11             yInc = yMax - yPos; break;
51.11         case SB_LINEUP: //the top arrow is clicked
52.11             yInc = -1; break;
53.11         case SB_LINEDOWN: //the bottom arrow is clicked
54.11             yInc = 1; break;
55.11         case SB_PAGEUP: //scroll bar shaft above the scroll box
56.11             yInc = min(-1, -yClient/yChar); break; // is clicked
57.11         case SB_PAGEDOWN: // scroll bar shaft below the scroll
58.11             yInc = max(1, yClient/yChar); break; //box is clicked
59.11         case SB_THUMBTRACK: // user is dragging the scroll box
60.11             yInc = HIWORD(wParam)-yPos; break;
61.11         default:
62.11             yInc = 0; break;
63.11     }
64.11     yInc = max(-yPos, min(yInc, yMax - yPos));
65.11     if(yInc != 0) {
66.11         yPos += yInc;
67.11         SetScrollPos(hwnd, SB_VERT, yPos, TRUE);
68.11         ScrollWindow(hwnd, 0, -yChar*yInc, NULL, NULL);
69.11     }
70.11     break;
71.11 case WM_HSCROLL:
72.11     xPosOld = xPos;
73.11     switch(LOWORD(wParam)){
74.11         case SB_PAGERIGHT:
75.11             xPos += 10; break;
76.11         case SB_LINERIGHT:
77.11             xPos += 1; break;
78.11         case SB_PAGELEFT:
79.11             xPos -= 10; break;
80.11         case SB_LINELEFT:
81.11             xPos -= 1; break;
82.11         case SB_TOP:
83.11             xPos = xMin; break;
84.11         case SB_BOTTOM:
85.11             xPos = xMax; break;
86.11         case SB_THUMBTRACK:
87.11             xPos = HIWORD(wParam); break;
88.11         default: break;
89.11     }
90.11     if(xPos > xMax) xPos = xMax;
91.11     if(xPos < xMin) xPos = xMin;
92.11     SetScrollPos(hwnd, SB_HORZ, xPos, TRUE);
93.11     if(xPos == xMax)
94.11         EnableScrollBar(hwnd, SB_HORZ, ESB_DISABLE_RIGHT);
95.11     if(xPos != xPosOld)
96.11         ScrollWindow(hwnd, xChar*(xPosOld-xPos), 0, NULL, NULL);
97.11     break;

```

ALGORITHM 11: Modeless dialog box: function *ModelessDialogProc* (part 2).

Interface), and so forth. Again, applying such up-to-date.NET technology as Windows Forms should be considered as well. Essentially, the presented results can be applied to all kinds of simulation: continuous, hybrid, and even discrete-event

applications. The approach has already been used in developing the research software in the field of designing particle accelerators, which was carried out by S. Kozynchenko and V. Kozynchenko [16–18].

```

98.12 case WM_PAINT:
99.12     if(vpp.empty())
100.12         break;
101.12     hDC = BeginPaint(hwnd, &ps);
102.12     //finding the painting limits:
103.12     iPaintBeg = max(0, yPos + ps.rcPaint.top/yChar - 1);
104.12     iPaintEnd = min(NUMLINES, yPos + ps.rcPaint.bottom/yChar);
105.12     font = CreateFontIndirect(&lf);
106.12     SelectObject(hDC, font);
107.12     SetBkMode(hDC, TRANSPARENT);
108.12     for(int i = iPaintBeg; i < iPaintEnd; i++){
109.12         y = yChar*(1-iVscrollPos+i); //y- and x-coordinates of
110.12         x = xChar*(1 - xPos); //the beginning of the string
111.12         if(vpp[particleModelessBox].history.size() == 0)
112.12             break;
113.12         if((signed)vpp[particleModelessBox].history.size() <= iPaintBeg)
114.12             break;
115.12         else
116.12             if(i < (signed)vpp[particleModelessBox].history.size()){
117.12                 temp = sprintf_s(buf,
118.12                     "t = 3e y = %.5f z = %.5f vy = %.5f vz = %.5f\
119.12                     v = %.5f W = %.5f",
120.12                     vpp[particleModelessBox].history.at(i).t,
121.12                     vpp[particleModelessBox].history.at(i).y,
122.12                     vpp[particleModelessBox].history.at(i).z,
123.12                     vpp[particleModelessBox].history.at(i).vy,
124.12                     vpp[particleModelessBox].history.at(i).vz,
125.12                     vpp[particleModelessBox].history.at(i).v,
126.12                     vpp[particleModelessBox].history.at(i).W);
127.12                 TextOut(hDC, x+10, y, buf, temp); //writing a line of text
128.12             } // to the client area from the buffer temp
129.12             else
130.12                 break;
131.12         }
132.12     EndPaint(hwnd, &ps);
133.12     break;
134.12 case WM_DESTROY:
136.12     hModelessDlg = 0;
137.12     break;
138.12 }
139.12 return 0;
140.12 }

```

ALGORITHM 12: Modeless dialog box: function *ModelessDialogProc* (part 3).

Appendix

See Algorithms 10, 11, and 12.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

- [1] S. S. Hajari, H. Shaker, and S. Doeber, "Beam dynamics design of the Compact Linear Collider drive beam injector," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 799, pp. 172–186, 2015.
- [2] D. Sandaruwan, N. Kodikara, C. Keppitiyagama, and R. Rosa, "A six degrees of freedom ship simulation system for maritime education," *The International Journal on Advances in ICT for Emerging Regions*, vol. 3, no. 2, pp. 34–47, 2010.
- [3] A. M. Yusop, R. Mohamed, A. Ayob, and A. Mohamed, "Dynamic modeling and simulation of a thermoelectric-solar hybrid energy system using an inverse dynamic analysis input shaper," *Modelling and Simulation in Engineering*, vol. 2014, Article ID 376781, 13 pages, 2014.
- [4] G. Ferretti, A. Leva, and B. Scaglioni, "Object-oriented modelling of general flexible multibody systems," *Mathematical and*

- Computer Modelling of Dynamical Systems: Methods, Tools and Applications in Engineering and Related Sciences*, vol. 20, no. 1, pp. 1–22, 2014.
- [5] A. I. Kozynchenko, “Analysis of predictive entry guidance for a Mars lander under high model uncertainties,” *Acta Astronautica*, vol. 68, no. 1-2, pp. 121–132, 2011.
 - [6] N. Sivakumaran and T. K. Radhakrishnan, “Modelling and predictive control of a multivariable process using recurrent neural networks,” *International Journal of Modelling and Simulation*, vol. 28, no. 1, pp. 20–26, 2008.
 - [7] R. Ewald and A. M. Uhrmacher, “SESSL: a domain-specific language for simulation experiments,” *ACM Transactions on Modeling and Computer Simulation*, vol. 24, no. 2, 2014.
 - [8] *Modelica—A Unified Object-Oriented Language for Systems Modelling: Language Specification, Version 3.3*, The Modelica Association, 2012, <http://www.modelica.org/>.
 - [9] S. A. Kozynchenko and Y. A. Svistunov, “Application of field and dynamics code to LEBT optimization,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 558, no. 1, pp. 295–298, 2006.
 - [10] S. Salleh, A. Y. Zomaya, S. Olariu, and B. Sanugi, *Numerical Simulations and Case Studies Using Visual C++.NET*, Wiley, Hoboken, NJ, USA, 2005.
 - [11] S. Salleh, A. Y. Zomaya, and S. A. Bakar, *Computing for Numerical Methods Using Visual C++*, Wiley, Hoboken, NJ, USA, 2008.
 - [12] S. Humphries Jr., *Principles of Charged Particle Acceleration*, Wiley, New York, NY, USA, 1986.
 - [13] T. Veerarajan and T. Ramachandran, *Numerical Methods: With Programs in C*, Tata McGraw-Hill, New Delhi, India, 2nd edition, 2006.
 - [14] C. Petzold, *Programming Windows: The Definitive Guide to the Win32 API*, Microsoft Press, Redmond, Wash, USA, 5th edition, 2002.
 - [15] H. Praehofer and A. Schoeppl, “A continuous and combined simulation platform in Java and its application in building paper mill training simulators,” in *Proceedings of the International Conference on Webbased Modeling & Simulation*, San Diego, Calif, USA, January 2000.
 - [16] S. Kozynchenko, “Development of the object-oriented program in C++ for simulation of beam dynamics in accelerator injection systems,” in *Proceedings of the 23rd Russian Particle Accelerator Conference (RuPAC '12)*, pp. 364–366, Saint Petersburg, Russia, September 2012.
 - [17] V. Kozynchenko and S. Kozynchenko, “Parallel beam dynamics simulation in injection systems taking into account particle interactions,” in *20th International Workshop on Beam Dynamics and Optimization (IVESC-ICEE-ICCTPEA-BDO '14)*, St. Petersburg, Russia, June 2014.
 - [18] S. A. Kozynchenko, “MPI-based software for charged particle beam dynamics simulation and optimization in the injection systems,” in *Proceedings of the 20th International Workshop on Beam Dynamics and Optimization (BDO '14)*, pp. 1–2, IEEE, Saint Petersburg, Russia, June-July 2014.