ОРДЕНА ЛЕНИНА
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.Келдыша
РОССИЙСКОЙ АКАДЕМИИ НАУК

Абакумов М.В.

О некоторых методах визуализации сеточных данных

Москва
2003

Аннотация

Предлагается модифицированный метод построения линий уровня основанный на алгоритме «квадродеревьев». Описывается метод, позволяющий формализовать операции скроллинга, масштабирования и печати изображения. Рассматривается способ эффективного доступа к сеточной функции и метод реализации «быстрой» цветовой заливки в Win32. Обсуждается универсальная структура файла данных трехмерного расчета и реализация высокоскоростного доступа к таким файлам. Описывается методика генерации видео в формате MPEG4 в операционной системе Win32.

Abstract

A modified method based on the quadtree algorithm for contouring functions of two variables suggested. A method for formalization of zooming, scrolling and printing operations proposed. The way of efficient grid function access and the «quick» area fill method in Win32 considered. A universal structure of the 3D calculation file and realization of high-speed access to such files discussed. A technique of MPEG4 video generation in Win32 described.

Содержание

Введение

Пожалуй, без преувеличения можно сказать, что каждому специалисту в области вычислительной математики неоднократно приходилось сталкиваться с задачей визуализации результатов математического моделирования. При этом, поскольку круг решаемых средствами вычислительной математики задач неуклонно расширяется, возникают новые постановки задач визуализации и методы их решения. Однако среди этого многообразия можно выделить некоторый класс задач, возникающих достаточно часто и допускающих универсальную постановку. Это позволяет, в свою очередь, реализовать методы их решения в виде библиотеки подпрограмм, на основе которой впоследствии можно разрабатывать программные средства визуализации того или иного математического эксперимента. Наиболее известным и, по мнению автора, удачным примером такой библиотеки является программный комплекс ГРАФОР (см., например, [1,2]). Созданный достаточно давно, он «поддерживается» разработчиками, а потому широко используется в вычислительной практике и в настоящее время.

В рамках данной работы рассматривается, безусловно, гораздо более «узкая» задача, которая состоит в следующем. Предполагается осуществлять постобработку результатов трехмерного (или двумерного) нестационарного расчета на неравномерной ортогональной сетке в декартовой, цилиндрической или сферической геометрии. Результатом такого расчета в общем случае является набор сеточных функций, полученный на некоторые последовательные моменты времени. Эти трехмерные сеточные функции (скалярные или векторные) и подлежат визуализации. А поскольку сформировавшиеся стандарты построения изображения функций трех переменных по-прежнему отсутствуют, автором рассматривается лишь задача построения изображений их сечений при фиксированном индексе по одной из пространственных переменных, то есть двумерных сеточных функций в полярной или декартовой геометрии. В этом случае общепринятыми методами изображении являются построение линий уровня и цветовая «заливка». Векторное поле обычно отображается совместно с линиями уровня в виде направленных стрелок, длина которых соответствует модулю вектора. Есть и другие методы, например LIC (Line Integral Convolution) [3], основанный на упорядочивании изначально случайной текстуры вдоль линии тока. Однако полученное изображение в этом случае не представляется возможным отображать совместно с картиной линий уровня скалярной функции.

Автор отдает себе отчет в том, что соответствующие методы достаточно известны и неоднократно реализованы в различных вариантах, в том числе и в ГРАФОРе. Тем не менее, задача эффективной реализации таких методов с учетом новых возможностей, предоставляемых операционными системами семейства Win32, не перестает быть актуальной.

В ходе данной работы автором была разработана и реализована программа визуализации результатов трехмерного нестационарного расчета на неравномерной сетке `ClcView` (**Cal**culated Data **View**er) . Программа универсальна в том смысле, что позволяет осуществить настройку на файлы данных конкретного расчета, если структура этих файлов подчиняется достаточно «свободным» требованиям. После настройки программа позволяет выбирать из файлов данные, необходимые для ее работы, и строить картину линий уровня (совместно с картиной векторного поля) заданных сечений трехмерных сеточных функций в декартовой или полярной геометрии. Реализованы следующие операции с изображением: масштабирование, скроллинг, увеличение заданных фрагментов, печать видимого фрагмента изображения на принтер (с полным использованием его разрешающей способности) или сохранение в графический файл (поддерживаются форматы BMP, WMF, PCX, JPEG). Наряду с линиями уровня осуществляется построение цветовой «заливки» в произвольной палитре (до 256 цветов), задаваемой пользователем. С полученным изображением также возможны все перечисленные выше операции. Реализована также так называемая пакетная обработка результатов расчета. При пакетной обработке задается набор файлов

данных результатов, соответствующий последовательным моментам времени. После этого без дальнейшего вмешательства пользователя либо строится последовательность изображений заданных сечений сеточных функций, сохраняемая в графические файлы, либо генерируется видео, например, в формате MPEG4, сохраняемое в файл AVI (`audio-video interleaved`).

В данной работе сама программа `ClcView` более упоминаться не будет. Важно лишь то, что она реализована, работоспособна и позволяет, в частности, провести тестирование использованных в ней алгоритмов. Далее же вниманию читателя предлагается обсуждение методов, разработанных в ходе реализации этой программы (и некоторых других [4]). При этом автор сознательно уклоняется от обсуждения аспектов, связанных непосредственно с объектно-ориентированным программированием в той или иной среде. Фрагменты программного кода немногочисленны и приводятся только тогда, когда без этого не обойтись. Основное же внимание уделено общему описанию методов и тех возможностей Win32, которые позволяют их эффективно реализовать, используя произвольную систему программирования. В работе также приводятся результаты тестов, которые могут быть интересны сами по себе, безотносительно рассматриваемых задач.

Далее коротко перечислим рассматриваемые в работе вопросы. В п.1 предлагается модифицированный метод построения линий уровня основанный на алгоритме «квадродеревьев»[5], достоинством которого является высокая надежность и простота реализации. В п.2 предложен общий метод отображения двумерных данных, заданных своими «физическими» координатами, позволяющий осуществлять с результирующим изображением операции масштабирования, скроллинга и копирования на другое устройство. В п.3 рассматриваются вопросы, связанные с эффективностью доступа к сеточной функции и ее интерполяцией. В п.4 описываются способы реализации «быстрой» цветовой заливки в Win32 (более 200 кадров в секунду, без учета времени на вычисление цвета). В п.5 обсуждается универсальная структура файла данных трехмерного расчета и реализация высокоскоростного доступа к таким файлам. В п.6 описывается методика генерации видео в формате MPEG4 в операционной системе Win32. Приводятся результаты тестирования, демонстрирующие возможность генерации полноформатного видео по данным расчета с производительностью 3-4fps (frame per second) на подробных сетках.

Работа выполнена при поддержке РФФИ 03-01-00311, Минпромнауки, государственный контракт №40.022.1.1.1106.

п.1. Метод построения линий уровня функции двух переменных

В этом пункте речь пойдет о методе построения линий уровня, в основу которого положен алгоритм «квадродеревьев», впервые описанный в работе [5]. Данный алгоритм базируется на предположении о том, что во всех точках прямоугольной области декартовой системы координат Ω каким-либо образом задана функция двух переменных $f(x,y)$. Задача построения линии уровня сводится к построению кривых в области Ω, все точки которых удовлетворяют уравнению $f(x,y)-l=0$, где l – заданный уровень функции, лежащий в пределах от $\min\limits_{\Omega} f(x,y)$ до $\max\limits_{\Omega} f(x,y)$.

Суть алгоритма «квадродеревьев» состоит в поиске отрезков линии уровня на основе рекурсивного разбиения исходной области на подобласти. Опишем данный алгоритм в той же форме, что и в работе [5], а именно с помощью фрагмента программы на языке Pascal [6], который наряду с прочими возможностями поддерживает рекурсивный вызов процедур.

```
procedure CreateTree (Depth:Integer; X,Y,Dx,Dy:Real);
 procedure SubDivide;
 begin
  CreateTree(Depth+1,X,Y,Dx/2,Dy/2);
  CreateTree(Depth+1,X,Y+Dy/2,Dx/2,Dy/2);
  CreateTree(Depth+1,X+Dx/2,Y,Dx/2,Dy/2);
  CreateTree(Depth+1,X+Dx/2,Y+Dy/2,Dx/2,Dy/2);
 end;
begin
if Depth<SearchDepth then SubDivide else
 if ContourPresent(X,Y,Dx,Dy) then
  if Depth<PlotDepth then SubDivide else Plot (X,Y,Dx,Dy);
end;
```

Параметрами подпрограммы `CreateTree` являются: `Depth` – глубина рекурсии (уровень вложенности рекурсивного вызова); `X,Y` – координаты вершины прямоугольной области; `Dx,Dy` – длины сторон прямоугольной области.

Первоначальное обращение к подпрограмме `CreateTree` производится с параметрами исходной прямоугольной области Ω и нулевым уровнем вложенности рекурсии.

Внутри подпрограммы `CreateTree` описана процедура `SubDivide`, в которой текущая прямоугольная область, заданная параметрами `X`, `Y`, `Dx`, `Dy`, разбивается на четыре равные прямоугольные подобласти, с вершинами `(X,Y)`, `(X,Y+Dy/2)`, `(X+Dx/2,Y)`, `(X+Dx/2,Y+Dy/2)`, и длинами сторон `Dx/2`, `Dy/2`. К каждой из полученных таким образом подобласти снова применяется подпрограмма `CreateTree` посредством рекурсивного вызова с увеличенным на единицу уровнем вложенности.

Теперь рассмотрим саму подпрограмму `CreateTree`, в функционировании которой ключевую роль играют два глобальных параметра метода: `SearchDepth` - глубина рекурсии при поиске отрезка линии уровня и `PlotDepth` - глубина рекурсии при прорисовке отрезка линии уровня. Параметр `SearchDepth` определяет точность поиска отрезков линии уровня. Пока значение уровня вложенности `Depth` меньше чем `SearchDepth`, продолжается процесс рекурсивного дробления исходной области на подобласти, который обеспечивается процедурой `SubDivide`. В результате дальнейшему анализу будут подвергаться подобласти (ячейки поиска), длины сторон которых в `2` в степени `SearchDepth` раз меньше соответствующих длин сторон исходной области Ω.

Таким образом, исходная область разбивается на ячейки поиска, каждая из которых анализируется на предмет прохождения через нее линии уровня посредством функции `ContourPresent` с параметрами `(X,Y,Dx,Dy)`, задающими ячейку поиска. Эта функция возвращает значение `True`, если через данную ячейку поиска проходит линия уровня, и `False` в противном случае (о конкретной реализации этой функции речь пойдет позднее). Если функция возвращает `False`, анализ данной ячейки поиска заканчивается, и происходит возврат из подпрограммы `CreateTree`, при котором управление передается в соответствующую точку процедуры `SubDivide` подпрограммы `CreateTree` на единицу меньшего уровня вложенности. Если для ячейки поиска функция `ContourPresent` возвращает значение `True`, то необходима дальнейшая ее обработка, имеющая целью прорисовку отрезка линии уровня, содержащегося внутри ячейки. Для этого ячейка поиска подвергается дальнейшему дроблению посредством рекурсивного вызова процедуры `SubDivide` до тех пор, пока уровень вложенности рекурсии не достигнет глубины вложенности прорисовки `PlotDepth`. При этом процессу дробления подвергаются только подобласти, для которых функция `ContourPresent` возвращает значение `True`, для остальных же процесс обработки заканчивается. Когда для текущей подобласти уровень вложенности `Depth` достигает глубины вложенности `PlotDepth` и функция

ContourPresent возвращает значение True, подобласть считается ячейкой прорисовки и к ней применяется процедура Plot. Процедура Plot осуществляет рисование отрезка прямой линии, который аппроксимирует отрезок линии уровня, пересекающей ячейку прорисовки.

Для полного завершения описания алгоритма осталось подробнее остановиться на деталях реализации функции ContourPresent и процедуры Plot. Сначала приведем один из возможных вариантов функции ContourPresent:

```
function ContourPresent (X,Y,Dx,Dy: Real):Boolean;
var F1,F2,F3,F4:Real;
begin
 F1:=Fl(X,Y);
 F2:=Fl(X+Dx,Y);
 F3:=Fl(X+Dx,Y+Dy);
 F4:=Fl(X,Y+Dy);
 ContourPresent:=(F1*F2<0)or(F2*F3<0)or(F3*F4<0)or(F4*F1<0);
end;
```

Здесь вычисляются значения функции $f_l(x,y) = f(x,y) - l$ в каждой вершине прямоугольной подобласти, заданной параметрами (X,Y,Dx,Dy), и определяются знаки произведения значений функции на концах отрезков всех ее границ. Если указанные произведения для всех границ неотрицательны, то считается, что линия уровня не пересекает подобласть.

Сразу отметим, что в случае, если размеры ячеек поиска, определяемые параметром SearchDepth, не достаточно малы, возможны варианты, когда процедура будет работать некорректно. Например, если замкнутый контур линии уровня целиком находится внутри ячейки поиска или линия уровня пересекает границы подобласти четное число раз. По этой причине корректный выбор параметра SearchDepth определяется конкретным видом функции $f(x,y)$ и параметрами исходной области Ω. Кроме того, пока остается открытым вопрос, как трактовать ситуацию, когда какое-либо из указанных выше произведений обращается в нуль, то есть линия уровня проходит через одну из вершин подобласти. В этой ситуации, вообще говоря, необходимо решать вопрос о принадлежности линии уровня к смежным областям, для которых данная вершина общая, что заведомо внесет нежелательную неоднородность в алгоритм. Пока оставим данное замечание для дальнейшего обсуждения.

Основная идея реализации процедуры Plot состоит в том, что для границ ячейки прорисовки (в случае если произведение значений функции на концах отрезка границы отрицательно) находятся точки пересечения линии уровня с отрезком границы. Здесь наиболее целесообразно воспользоваться линейной интерполяцией функции по значениям на концах отрезка границы. Более сложные способы нахождения точки пересечения заведомо приведут к большему числу обращений к функции $f_l(x,y)$, а поскольку вычисление значений этой функции само по себе может быть трудоемким, эффективность метода в конечном счете только снизится. Кроме того, качество прорисовки всегда можно увеличить за счет увеличения параметра PlotDepth.

При корректном выборе параметров метода получаются две точки пересечения линии уровня и границы ячейки прорисовки (линия уровня пересекает два отрезка границы), которые соединяются прямой линией. Исключение представляет случай, когда таких точек пересечения четыре. В этом случае пары точек соединяются отрезками прямой линии, например, крест-накрест.

Подытоживая описание алгоритма (далее будем называть его базовым), остановимся на вопросе выбора параметров SearchDepth и PlotDepth. На рис. 1.1 проиллюстрирована работа алгоритма для случая SearchDepth=2, PlotDepth=4 (более жирными ли-

ниями показаны разбиения на подобласти пока значения `Depth` не превышают `SearchDepth`). Как уже отмечалось, наибольшее влияние на эффективность метода оказывает выбор параметра `SearchDepth`, который в наибольшей степени определяет число обращений к функции $f_l(x,y)$. С этой точки зрения, параметр желательно выбирать минимальным. Однако при малых, применительно к конкретному случаю, значениях параметра `SearchDepth` возможна "потеря" участков линии уровня, а также отсутствие прорисовки мелких замкнутых контуров. Так, например, если линия уровня пересекает отрезки границы ячейки поиска четное число раз, то для такой ячейки функция `ContourPresent` возвратит `False`. Поэтому оптимальное значение этого параметра необходимо выбирать исходя из специфики конкретного случая. Увеличение параметра `PlotDepth` менее существенно влияет на быстродействие алгоритма, так как разбиению подвергаются только подобласти, имеющие точки пересечения с линией уровня (см. рис. 1.1).

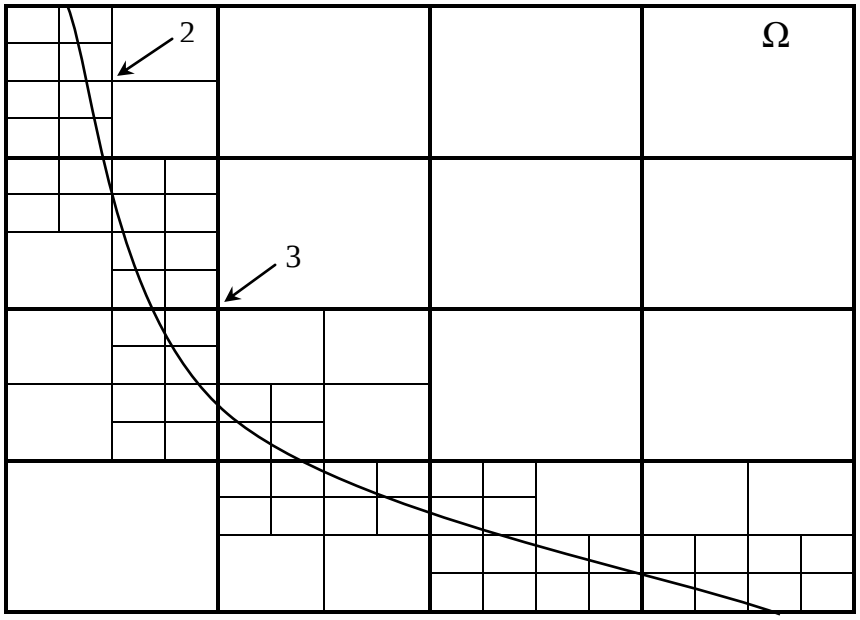

Рис. 1.1

Перейдем к обсуждению достоинств и недостатков базового алгоритма. К несомненным достоинствам необходимо отнести, в первую очередь, простоту реализации. При этом использование рекурсивного вызова процедур, который не поддерживается в ряде языков программирования (например, в некоторых версиях компиляторов языка Fortran [7]) не является существенным препятствием для использования алгоритма. Рекурсивный вызов легко моделируется с помощью организации структуры данных типа стек (Stack LIFO: Last In - First Out)[6], предусматривающую порядок обмена, при котором данные, сохраненные последними, извлекаются в первую очередь. При каждом "фиктивном" вызове процедурой самой себя в стек помещаются значения фактических параметров, локальных переменных процедуры, а также точки входа в процедуру при возвращении в нее. При возврате же на уровень вверх восстанавливаются значения, сохраненные последними и управление передается на восстановленную точку входа.

Следующим важным достоинством алгоритма является возможность выбора его параметров, при которых будет получена сколь угодно подробная картина линий уровня для любой наперед заданной (непрерывной) функции двух переменных.

К недостаткам базового алгоритма необходимо отнести избыточное обращение к функции $f_l(x,y)$, которая может вычисляться достаточно трудоемко. Так на рис.1.1 стрелками отмечены вершины областей разбиения, где значения будут вычисляться 2 и 3

раза соответственно (если в процедуре `Plot` использовать значения, найденные при последнем вызове функции `ContourPresent`). Не сложно представить себе ситуацию, когда количество обращений к функции в вершине совпадет со значением `PlotDepth`.

Далее отметим, что увеличение значения параметра `PlotDepth` не всегда приводит к улучшению «качества» результирующего изображения. Приведем следующий пример (см. рис.1.2).

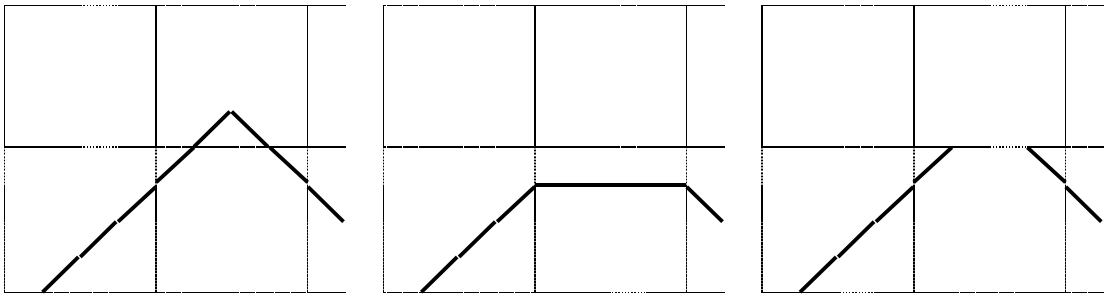

Рис. 1.2

На всех трех рисунках отображены четыре смежных подобласти разбиения (пунктирная линия) на этапе, когда `Depth=SearchDepth`. На первом рисунке (сплошная линия) отображается реальное положение линии уровня. Как легко видеть для верхней правой подобласти функция `ContourPresent` возвращает значение `False`. Далее, если значения `PlotDepth` и `SearchDepth` совпадают, то в результате дальнейшей работы алгоритма получится результат, представленный на втором рисунке. Если же `PlotDepth` превышает `SearchDepth`, то получится результат, представленный на последнем рисунке, то есть линия уровня будет иметь разрыв (не приемлемый в контексте достоверности результирующего изображения) , причем ситуация не изменится при любом дальнейшем увеличении параметра `PlotDepth`. Отметим, что в процессе тестирования алгоритма на реальных вычислительных данных, по которым путем интерполяции строилась функция $f(x,y)$, описанная ситуация возникала весьма часто. По этой причине наиболее надежно всегда выбирать параметры `SearchDepth` и `PlotDepth` равными.

Как уже отмечалось, линия уровня функции может состоять из нескольких, несвязанных друг с другом кривых. Далее такие кривые будем называть элементарными. Задача идентификации линий уровня состоит в нанесении единственной метки уровня на каждую из элементарных кривых. Особенно актуальной эта задача становится в том случае, когда невозможно представление линий различных уровней разными цветами (например, при печати на черно-белом принтере). Отметим, что в описанном алгоритме задача идентификации не рассматривается.

Далее перейдем к описанию модифицированного метода построения линий уровня, предложенного автором. Данный метод, во-первых, полностью решает задачу идентификации линий уровня, во-вторых, минимизирует количество обращений к функции (одно на каждую точку области Ω, задействованную в разбиении ее на подобласти) и, наконец, создает структуру данных, предназначенную для наиболее эффективной многократной прорисовки результирующего изображения средствами операционной системы Windows. Это особенно важно для последующего масштабирования изображения, сохранения его в графический файл, печати и т.п.

Из анализа базового алгоритма несложно сделать вывод, что для минимизации обращения к функции $f_1(x,y)$ необходимо как-то сохранять ранее вычисленные значения, которые могут потребоваться в дальнейшем, а при вычислениях в подобласти разбиения, проверять, находились ли ранее необходимые значения функции в вершинах. Это предполагает организацию дополнительных структур данных (древовидных) с вычисленными

значениями функции и информацией о смежности областей разбиения, что заведомо сильно усложнит метод. Однако есть и более изящное решение. Почему бы не использовать тот же стек, который и так используется для рекурсивного вызова процедуры `CreateTree`. Для пояснения того, как именно, приведем модифицированный фрагмент программы, аналогичный по своим функциям ранее приведенному фрагменту для базового алгоритма.

```
procedure CreateTree(Depth:Byte;X1,Y1,X3,Y3,F1,F2,F3,F4: Single);
var XM,YM,FL,FR,FU,FD,FC: Single;
 procedure SubDivide;
 begin
  XM:=(X3+X1)/2;
  YM:=(Y3+Y1)/2;
  FL:=Fl(X1,YM);
  FR:=Fl(X3,YM);
  FU:=Fl(XM,Y3);
  FD:=Fl(XM,Y1);
  FC:=Fl(XM,YM);
  CreateTree(Depth+1,X1,Y1,XM,YM,F1,FD,FC,FL);
  CreateTree(Depth+1,XM,Y1,X3,YM,FD,F2,FR,FC);
  CreateTree(Depth+1,XM,YM,X3,Y3,FC,FR,F3,FU);
  CreateTree(Depth+1,X1,YM,XM,Y3,FL,FC,FU,F4);
 end;
begin
 if Depth<SearchDepth then SubDivide else
  if ContourPresent(F1,F2,F3,F4) then
   if Depth<PlotDepth then SubDivide else
    Plot(X1,Y1,X3,Y3,F1,F2,F3,F4);
end;
```

Как легко видеть, здесь параметры подпрограммы `CreateTree` отличаются от базовых. Область разбиения теперь задается параметрами `X1`, `Y1`, `X3`, `Y3`, определяющими координаты левого нижнего и правого верхнего углов области соответственно. Также в процедуру извне передаются значения `F1`, `F2`, `F3`, `F4` функции $f_l(x,y)$, вычисленные ранее в вершинах области разбиения, занумерованные в порядке против часовой стрелки от левой нижней вершины. Кроме того в теле подпрограммы `CreateTree` описываются локальные переменные: `XM`, `YM` – для хранения координат середины области разбиения; `FL`, `FR`, `FU`, `FD`, `FC` – для хранения значений функции на серединах сторон и в центре области разбиения. После вычисления этих значений их комбинации с соответствующими значениями `X1`, `Y1`, `X3`, `Y3`, `F1`, `F2`, `F3`, `F4` передаются при рекурсивном вызове подпрограммы `CreateTree` в процессе дробления области на четыре подобласти внутри процедуры `SubDivide`. Важно отметить, что при рекурсивном вызове указанные локальные переменные автоматически заносятся в стек, а при выходе из рекурсивного вызова автоматически же восстанавливаются.

Первоначальное обращение к подпрограмме `CreateTree` производится с нулевым уровнем вложенности рекурсии, параметрами `X1`, `Y1`, `X3`, `Y3` исходной прямоугольной области Ω, и значениями функции `F1`, `F2`, `F3`, `F4`, вычисленными в ее вершинах.

Таким образом, при сохранении прежней простоты реализации и без использования дополнительных структур данных удается добиться, чтобы значение функции в каждой точке области Ω, задействованной в разбиении ее на подобласти, вычислялось ровно один раз. Вычислений значений функции внутри процедур `ContourPresent` и `Plot` более не производится, так как в эти процедуры также передаются ранее вычисленные значения.

В результате проведенных модификаций стек, очевидно, будет использоваться более интенсивно. Однако нетрудно подсчитать, что даже при заведомо избыточных параметрах метода `PlotDepth` и `SearchDepth` равных 100 загрузка стековой памяти составляет менее 10Кб (здесь учтено, что тип `Single` языка `Pascal` соответствует типу `Real*4` языка `Fortran` и занимает 4 байта). При этом 32 битные версии операционной системы Windows позволяют приложениям использовать стековую память вплоть до 2097152 Кб при наличии, конечно, соответствующих физических носителей памяти. Таким образом, увеличение нагрузки на процедурный стек можно признать несущественной.

Далее перейдем к тому, как решается задача идентификации линий уровня. Как уже отмечалось, результатом работы процедуры `Plot` является получение прямолинейного отрезка (или пары отрезков) линии уровня, которые далее будем называть отрезками прорисовки. В базовом алгоритме эти отрезки отображаются на устройство вывода немедленно после их нахождения, и далее информация о них теряется. В модифицированном алгоритме немедленный вывод на устройство отрезков прорисовки не производится, а происходит их сохранение в динамическую структуру данных - однонаправленный список [6]. При этом отрезок задается координатами своих концов в области Ω, то есть никаких преобразований к координатам устройства вывода, да и вообще какой-либо привязки к конкретному устройству не производится. Таким образом, результатом работы алгоритма для заданного уровня l в конечном счете является список всех найденных отрезков прорисовки линии данного уровня. Именно этот список и подлежит дальнейшей обработке.

Целью дальнейшей обработки списка отрезков прорисовки является составление из этого набора отрезков набора элементарных (не связанных друг с другом) кривых линии уровня. При этом элементарная кривая представляет собой связную ломаную, заданную координатами своих вершин, причем отрезок, соединяющий соседние вершины, является одним из отрезков прорисовки. Ломаная является либо замкнутой, тогда координаты первой и последней ее вершины совпадают, либо разомкнутой, тогда крайние вершины ломаной лежат на границе области Ω.

Каждую связную ломаную опять разумно представлять в виде динамического списка ее вершин, поскольку количество вершин заранее неизвестно. Однако, чтобы избежать путаницы, списком по-прежнему будем называть список отрезков прорисовки, а список, представляющий ломаную, просто ломаной. При этом добавление вершины к началу или к концу ломаной будет означать добавление соответствующей записи в начало или конец списка, представляющего ломаную.

Далее под термином «извлечь из списка» будем подразумевать поиск в списке (путем перебора) отрезка, удовлетворяющего каким-либо критериям, с его последующим удалением из списка в случае успешного поиска. Алгоритм состоит в процессе последовательного построения ломаных путем извлечения отрезков из общего списка отрезков прорисовки. Тем самым алгоритм естественным образом завершается, когда список пуст, и достаточно конкретизировать, каким образом из списка извлекается одна ломаная.

Для извлечения очередной ломаной из списка (предполагая, что он не пуст) будем действовать следующим образом. Извлечем из списка первый отрезок и сформируем из его концов ломаную, состоящую из двух вершин. Далее осуществим процесс наращивания ломаной «вверх». Для этого извлечем из списка отрезок, один из концов которого совпадает с первой вершиной ломаной, после чего добавим в начало ломаной другой конец найденного отрезка. Будем циклически продолжать этот процесс до тех пор, пока поиск отрезка в списке успешен. Далее аналогично осуществим процесс наращивания ломаной «вниз», при котором новые вершины добавляются в конец ломаной. В результате будет сформирована ломаная, соответствующая элементарной кривой линии уровня, а из списка удалены отрезки ее составляющие.

Таким образом, по окончании работы указанного алгоритма будет построен набор ломаных, а список отрезков прорисовки окажется пустым. Применяя получившийся метод построения линий уровня для каждого заданного уровня l, мы получим результирующий набор ломаных, представляющих элементарные кривые различного уровня. Этот набор данных является оптимальным для последующей прорисовки и идентификации линии уровня. Действительно, для идентификации достаточно при отображении поставить метку уровня, например, в средней вершине каждой ломаной. Более того, поскольку данные обо всех отображаемых элементарных кривых имеются, можно поставить и решить задачу об оптимальном, например, с точки зрения визуального восприятия, расположении меток.

Прорисовка результирующего отображения будет подробно обсуждаться в следующем пункте, здесь же выскажем некоторые дополнения к описанному выше. Как сразу же ясно компетентному читателю, «слабым местом» описанного алгоритма построения элементарных кривых является сравнение вершин на предмет совпадения, которое сводится к сравнению их координат, т.е. действительных чисел в их машинном представлении. Последняя операция, как известно, не является корректной, поскольку все вычисления проводятся с некоторыми ошибками округления. Первый вариант решения этой проблемы состоит в том, что сравнение производится с некоторым допуском. Однако в этом случае возникает риск некорректной работы алгоритма и в случае, если этот допуск мал по сравнению с ошибками округления (тогда вершины, которые «физически» совпадают, окажутся различными), и в случае, если допуск велик (тогда несовпадающие вершины окажутся при сравнении одинаковыми). Более того, тестирование этого варианта показало, что на практике подобрать универсальную величину допуска невозможно, и алгоритм работает корректно лишь в редких случаях и на «гладких» данных. По этой причине в модифицированном алгоритме предлагается следующее решение этой проблемы, которое показало свою абсолютную надежность.

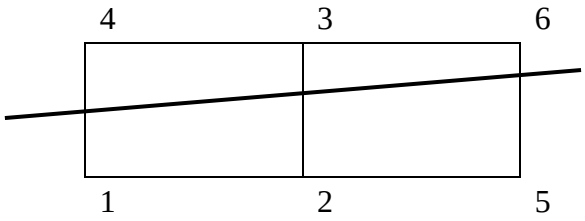

Рис. 1.3

Рассмотрим две смежные ячейки прорисовки и занумеруем их вершины так, как показано на рис.1.3. Предположим, что в ходе работы алгоритма выявлено прохождение линии уровня, в частности, через общую границу ячеек 2-3. Для обеих ячеек, в силу описанного метода, будет применена процедура `Plot` со следующими параметрами:

`Plot(X1,Y1,X3,Y3,F1,F2,F3,F4)` для левой ячейки;
`Plot(X2,Y2,X6,Y6,F2,F5,F6,F3)` для правой ячейки.

Далее учтем, что все значения функций и координаты вершин ячеек вычисляются единожды, а далее по необходимости передаются другим процедурам. Поэтому все одинаково обозначенные параметры действительно идентичны в том смысле, что их машинные представления совпадают с точностью до бита. По тем же причинам можно утверждать, что идентичны `X3` и `X2`, `Y1` и `Y2`, `Y3` и `Y6`. Таким образом, для обеих ячеек передаются идентичные координаты отрезка общей границы и значения функции на его концах. А поскольку порядок параметров процедуры `Plot` также строго определен, не представляет труда вычислять точку пересечения линии уровня и общей границы в соседних ячейках идентичным образом и получать ее координаты, совпадающие в их машинном представлении. Этот подход позволяет корректно использовать сравнение действительных чисел и гарантирует, что «физически» различные вершины не совпадут, поскольку размеры ячей-

ки прорисовки на практике существенного превышают точность машинного представления действительного числа.

Теперь необходимо обсудить ситуацию, когда в ходе работы метода получается, что линия уровня проходит через вершину ячейки прорисовки. Как отмечалось ранее, в этом случае необходимо дополнительно принимать решение, обрабатывать ли отрезок прорисовки в данной ячейке или в какой-либо из смежных. Это влечет за собой существенную неоднородность алгоритма. Для того, чтобы избежать таких осложнений, достаточно «запретить» функции $f_l(x,y)$ обращаться в ноль, то есть принудительно изменять нулевое значение на некоторую достаточно малую величину `ZeroEps`. При отображении это приведет к тому, что линия уровня сместится на «незаметное для глаза» расстояние относительно истинного положения в окрестности такой вершины, что тем более несущественно на фоне общей приближенности построения. В случае же, когда исходная функция является тождественной константой в области Ω, то есть $f(x,y) \equiv l = const$, при попытке построения линии уровня l метод не найдет ни одного отрезка прорисовки. Однако это не представляется критичным, поскольку задача построения линии уровня в этой ситуации сама по себе некорректна, так как любая кривая в области Ω формально принадлежит линии уровня l.

Таким образом, в модифицированном алгоритме построения линии уровня минимизированы затраты на вычисления, связанные с избыточными обращениями к заданной функции $f(x,y)$, однако дополнительно используется алгоритм переборного типа для постобработки отрезков прорисовки, направленный на решение задачи идентификации линий уровня и оптимизирование данных для дальнейшего эффективного отображения, о котором речь пойдет в следующем пункте. В заключение отметим, что данный метод без изменений переносится на случай прямоугольной области в полярной системе координат, если учет геометрии проводить на стадии отображения результата. Кроме того, метод применим и в случае области, отличной от прямоугольной, если исходную функцию доопределить в объемлющем прямоугольнике, например, значением $\min\limits_{\Omega} f(x,y)$.

п.2. Метод «сдвоенного окна» отображения двумерных данных

Как вытекает из сказанного в предыдущем пункте, для решения задачи об отображении линий уровня функции двух переменных на каком-либо устройстве (экране монитора, принтере и т.п.) достаточно отобразить совокупность отрезков, заданных своими двумерными координатами, которые далее будем называть физическими. Более того, этого достаточно для решения целого ряда задач отображения, поскольку любое изображение можно сколь угодно точно аппроксимировать набором отрезков (приписав им определенный цвет). Поэтому далее сформулируем задачу отображения в следующем виде.

Необходимо отобразить набор отрезков, физические координаты которых ограничены прямоугольником с декартовыми координатами левого нижнего угла (X_1, Y_1) и правого верхнего (X_2, Y_2), в окне. Далее для краткости будем обозначать такой прямоугольник (X_1, Y_1, X_2, Y_2) и называть его физическим. Под термином «окно» будем подразумевать прямоугольную область на экране монитора (окно Windows или его часть), прямоугольную область на листе бумаги при печати на принтере и т.п. При этом окно имеет некоторое фиксированное разрешение, то есть состоит из дискретного набора пикселей, каждый из которых может быть окрашен в определенный цвет из некоторого набора. Пиксели расположены упорядоченно по строкам и столбцам таким образом, что к ним можно привязать систему целочисленных координат (wX, wY). При отображении в окне требуется обеспечить многократное увеличение изображения с возможностью его передвижения

(скроллинга) в рамках окна с целью просмотра отдельных фрагментов, а также копирование видимой части изображения из окна на одном устройстве в окно на другом, с учетом различного разрешения, а также пиксельного аспекта окон (далее просто аспект). Аспектом устройства обычно называют отношение физических размеров пикселя по горизонтали и вертикали.

Основную идею метода решения поставленной задачи можно пояснить на следующем примере. Пусть наблюдатель располагается в комнате на некотором удалении от окна (в изначальном смысле этого слова), и ему доступен для обозрения фрагмент расположенного за окном пейзажа. Не меняя своего положения, наблюдатель, очевидно, не может рассмотреть ничего кроме доступного фрагмента, поскольку возможность двигать сам пейзаж отсутствует. Теперь представим, что пейзаж за окном на самом деле написан на большом (существенно превышающем размеры окна) холсте, и имеется возможность двигать холст относительно окна по горизонтали и вертикали. В этом случае у наблюдателя имеется возможность рассмотреть все фрагменты пейзажа, не меняя своего местоположения. Аналогично воспринимается ситуация, когда пользователь персонального компьютера просматривает изображение в окне монитора, используя полосы прокрутки (если изображение не помещается в окно целиком). Однако при этом возникает вопрос, что же выступает в роли холста, ведь физически для «рисования» доступна лишь рабочая область самого окна (`Client Rectangle`). Здесь-то и уместно сформулировать основную идею предлагаемого метода. Наряду с окном на реальном устройстве вывода будем также рассматривать окно, выступающее в роли холста, которое будем называть виртуальным. Поскольку виртуальное окно не имеет физического воплощения, «нарисовать» в нем что-либо можно также лишь виртуально. Поэтому для обозначения прорисовки чего-либо в виртуальном окне будем употреблять термин «отобразить». Далее будем считать, что отображение набора отрезков происходит именно в виртуальном окне, размеры которого, вообще говоря, превышают размеры окна на устройстве. При этом в окне на устройстве непосредственно прорисовывается лишь фрагмент виртуального окна. Последнее предполагает некоторую «привязку» виртуального окна к окну на устройстве, то есть, если угодно, рассмотрение «сдвоенного окна». О том, как осуществляется такая привязка речь пойдет ниже.

Будем исходить из того, что виртуальное окно также состоит из пикселей с координатами (vX, vY), и пиксельный аспект известен. По сложившейся традиции виртуальное окно будем определять координатами верхнего левого (vX_1, vY_1) и нижнего правого (vX_2, vY_2) угла. Важной для дальнейшего подзадачей является привязка физического прямоугольника (X_1, Y_1, X_2, Y_2) к виртуальному окну, то есть, фактически, получение линейных функций перехода от физических координат (X, Y) к оконным (vX, vY). Эти функции перехода определяют, каким образом в виртуальном окне будет отображаться набор отрезков и, в частности, сам физический прямоугольник. При этом понятно, что если физический прямоугольник отображается корректно, то корректно, в силу линейности функций перехода, будет отображаться и результирующее изображение (набор отрезков). Далее под корректностью отображения будем понимать следующий набор требований. Во-первых, оконные координаты физического прямоугольника не должны выходить за рамки виртуального окна (vX_1, vY_1, vX_2, vY_2) (изображение целиком в окне). Во-вторых, размеры физического прямоугольника в оконных координатах максимально возможны при соблюдении первого требования (максимальное использование разрешения окна). В-третьих, прямоугольник центрирован относительно окна (изображение в центре окна). И наконец, должны соблюдаться физические пропорции прямоугольника, то есть, если прямоугольник является квадратом, то именно квадратом он должен выглядеть при отображении (сохранение физических пропорций изображения). От последнего требования в ряде случаев можно отказаться (тогда преобразование координат строится совсем просто),

однако здесь уместно продемонстрировать схему привязки для наиболее «жестких» критериев.

Обозначим физические ширину и высоту пикселя в каких-либо единицах измерения pDx, pDy соответственно. Если известно лишь отношение этих размеров $AspXY$, то достаточно положить $pDx = AspXY$, $pDy = 1$. Далее введем в рассмотрение следующие величины:

$$Dx = X_2 - X_1,$$
$$Dy = Y_2 - Y_1,$$

– ширина и высота физического прямоугольника;

$$FocX = (X_1 + X_2)/2,$$
$$FocY = (Y_1 + Y_2)/2,$$

– координаты фокуса (центра) физического прямоугольника;

$$vDx = vX_2 - vX_1 + 1,$$
$$vDy = vY_2 - vY_1 + 1,$$

– целочисленные ширина и высота виртуального окна;

$$vFocX = (vX_1 + vX_2) \operatorname{div} 2,$$
$$vFocY = (vY_1 + vY_2) \operatorname{div} 2,$$

– целочисленные координаты фокуса виртуального окна (div – операция целочисленного деления).

Далее вычислим отношения соответствующих физических размеров виртуального окна (в единицах измерения пикселя) и прямоугольника (X_1, Y_1, X_2, Y_2):

$$pKx = 2*(vFocX - vX_1)*pDx/Dx;$$
$$pKy = 2*(vFocY - vY_1)*pDy/Dy.$$

Здесь вместо ширины окна vDx берется удвоенное расстояние в пикселях между координатой фокуса $vFocX$ и координатой ближайшего по горизонтали пикселя на границе окна vX_1. Это позволяет гарантировать, что оконные координаты прямоугольника впоследствии не выйдут за границы виртуального окна, как при четной, так и при нечетной его ширине. Аналогично ищется отношение высот.

Далее получим коэффициенты растяжения для переходе от физических к оконным координатам:

$$Kx = \min(pKx, pKy)/pDx;$$
$$Ky = \min(pKx, pKy)/pDy.$$

Окончательно функции перехода от физических к оконным координатам будут выглядеть следующим образом:

$$vX(X) = \text{Round}(vFocX + Kx*(X - FocX));$$
$$vY(Y) = \text{Round}(vFocY - Ky*(Y - FocY)).$$

Здесь учтено, что оконные координаты по оси y увеличиваются в направлении сверху вниз, а физические - в обратном направлении (Round – функция округления до целого).

Функции обратного преобразования координат имеют вид:

$$X(vX) = FocX + (vX - vFocX)/Kx;$$
$$Y(vY) = FocY - (vY - vFocY)/Ky.$$

Далее заметим, что в общем случае после привязки координат не вся область виртуального окна окажется покрытой прямоугольником (X_1, Y_1, X_2, Y_2) в оконных координатах. Действительно, пусть виртуальное окно является квадратным с единичным аспектом, а вертикальный размер физического прямоугольника (X_1, Y_1, X_2, Y_2) существенно превышает горизонтальный. Тогда при отображении на виртуальное окно, указанным выше способом, области по боковым краям окна не будут покрыты прямоугольником, ведь в преобразовании координат заложено сохранение физических пропорций. Эти области, таким образом, не будут задействованы при отображении.

Для дальнейшего потребуется коррекция (урезание) границ виртуального окна таким образом, чтобы в результате прямоугольник (X_1, Y_1, X_2, Y_2) в оконных координатах покрывал его полностью. Такое преобразование имеет вид:

$$vX_1 = \text{Round}(vFocX + Kx * Dx / 2);$$
$$vX_2 = \text{Round}(vFocX - Kx * Dx / 2);$$
$$vY_1 = \text{Round}(vFocY - Ky * Dy / 2);$$
$$vY_2 = \text{Round}(vFocY + Ky * Dy / 2).$$

Для виртуального окна с «новыми» координатами углов функции преобразования координат не изменятся, и отображение в этом окне останется корректным. Далее под привязкой к виртуальному окну (vX_1, vY_1, vX_2, vY_2) физического прямоугольника (X_1, Y_1, X_2, Y_2) будем понимать получение коэффициентов в формулах преобразования координат и последующую коррекцию границ окна.

Таким образом, если предположить возможность отображения отрезка прямой линии, заданного в оконных координатах, в виртуальном окне, то после привязки можно получить корректное, с учетом ранее обсуждавшихся критериев, изображение как самого прямоугольника (X_1, Y_1, X_2, Y_2), так и любого отрезка внутри него, заданного своими физическими координатами. Для этого достаточно использовать приведенные функции преобразования координат $vX(X)$, $vY(Y)$. Однако, как уже отмечалось, собственно прорисовка будет происходить в окне на устройстве вывода, к которому необходимо привязать виртуальное окно.

Будем считать, что окно на устройстве вывода также задано координатами верхнего левого (wX_1, wY_1) и нижнего правого (wX_2, wY_2) угла. Для первичной привязки виртуального окна достаточно приравнять координаты углов виртуального окна и координаты окна на устройстве $(vX_1, vY_1, vX_2, vY_2) = (wX_1, wY_1, wX_2, wY_2)$, определить размеры пикселя pDx, pDy значениями, полученными для конкретного устройства (функция Windows API `GetDeviceCaps`), после чего произвести привязку физического прямоугольника (X_1, Y_1, X_2, Y_2) к виртуальному окну.

Замечание. Здесь и далее автор считает целесообразным приводить ссылки на функции API (Application Programming Interfaces), используемые при реализации рассматриваемых алгоритмов на базе операционной системы Microsoft Windows (Win32). Подробное описание этих функций можно найти в справочном руководстве Windows SDK (Software Development Kit), которое входит в состав руководства любой среды программирования под Win32, а также доступно по адресу[8] в сети Internet.

После указанной операции для изображения отрезка, заданного физическими координатами своих концов, достаточно преобразовать эти координаты к оконным, используя функции $vX(X)$, $vY(Y)$, и произвести прорисовку отрезка прямой линии с полученными целочисленными координатами в окне устройства (`MoveToEx`, `LineTo`). Важно отметить, что прорисовка в окне устройства производится именно с координатами виртуального ок-

на, и после первичной привязки будет получено корректное изображение системы отрезков, «вписанное» в окно устройства.

Далее рассмотрим ситуацию (пока абстрактную), когда границы виртуального окна выходят за рамки окна устройства, то есть ширина или высота виртуального окна (vDx или vDy) соответственно превышают ширину или высоту окна устройства (wDx или wDy). Будем считать, что привязка физического прямоугольника к виртуальному окну произведена. Тогда, например, прорисовка отрезка с физическими координатами концов (X_1, Y_1) и (X_2, Y_2) сводится к прорисовке в окне устройства отрезка прямой линии с оконными координатами (vX_1, vY_2), (vX_2, vY_1)(см. рис.2.1). При этом хотя бы один из «оконных» концов отрезка находится вне границ окна устройства. Однако рассматриваемый метод отображения изначально рассчитан на то, что операционная система (в данном случае Win32) позволяет произвести прорисовку линии с такими координатами в окне устройства таким образом, что изображена будет только видимая в окне часть линии, а часть линии вне окна изображаться не будет. Тем самым при прорисовке всего набора отрезков в координатах виртуального окна (в оконных координатах) в окне устройства будет изображен лишь видимый фрагмент изображения, а «лишняя» его часть будет отсечена средствами операционной системы автоматически. Такой подход оправдан, так как отсечение средствами операционной системы (`clipping`) заведомо эффективней любого другого способа, поскольку максимально использует аппаратные возможности. Все выше сказанное остается справедливым также и в случае, когда прорисовку необходимо проводить не во всей рабочей области окна устройства, а лишь в некоторой его прямоугольной подобласти (например, при печати в прямоугольной области листа бумаги). В этом случае достаточно задать эту подобласть в качестве новой (более узкой, чем рабочая область окна) области отсечения (`IntersectClipRect`).

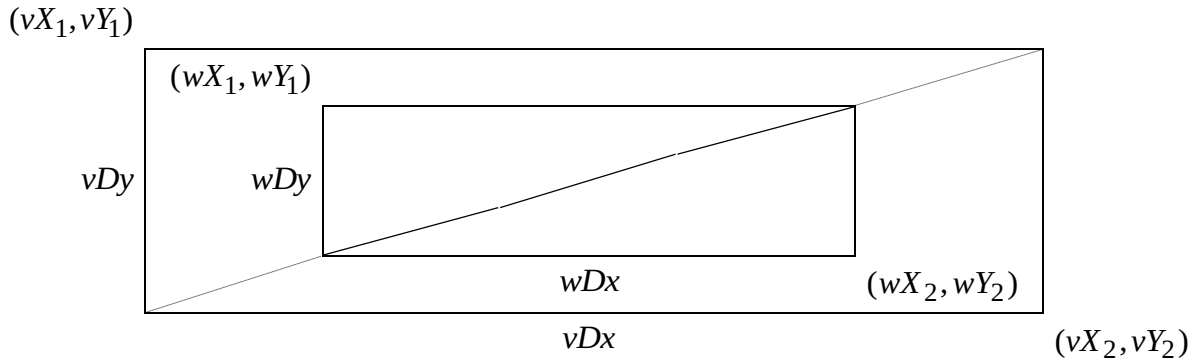

Рис. 2.1

Далее можно перейти к описанию основных операций со «сдвоенным окном». Ранее была введена базовая операция **первичной привязки**, в результате которой изображение «вписывается» в окно, то есть прорисовывается в окне целиком. Теперь снова представим, что границы виртуального окна выходят за рамки окна устройства (см. рис.2.1). В этом случае для обеспечения возможности просмотра любого фрагмента изображения необходимо иметь возможность «передвигать» окна относительно друг друга. При этом передвижение окна устройства в ряде случаев исключено (например, при печати). Зато виртуальное окно, поскольку оно не привязано к конкретной аппаратуре, может быть подвергнуто произвольным преобразованиям.

Как легко видеть, операция прокрутки изображения или **скроллинга** сводится к следующему. Для просмотра произвольного фрагмента изображения достаточно изменять координаты виртуального окна при неизменных его размерах. Обсудим подробно операцию скроллинга по горизонтали, поскольку по вертикали она производится абсолютно

аналогично. Пусть ширина виртуального окна vDx превосходит ширину окна устройства wDx. Тогда, очевидно, необходимо иметь возможность прокрутки в диапазоне от нуля до $sRangeX = vDx - wDx$. При этом крайней левой позиции скроллинга $sPosX = 0$ соответствует координата виртуального окна $vX_1 = wX_1$, а крайней правой позиции скроллинга $sPosX = sRangeX$ - координата $vX_1 = wX_1 - (vDx - wDx)$. Таким образом, для установки позиции скроллинга $sPosX$ из диапазона от 0 до $sRangeX$ необходимо определить координаты виртуального окна следующим образом:

$$vX_1 = wX_1 - sPosX;$$
$$vX_2 = vX_1 + vDx - 1.$$

После такого преобразования нет необходимости производить привязку физического прямоугольника к виртуальному окну заново, если она проведена ранее. Достаточно лишь заново определить значение $vFocX = (vX_1 + vX_2) \operatorname{div} 2$. Остальные коэффициенты в функциях преобразования координат $vX(X)$, $vY(Y)$ не изменятся.

Следующей обсудим операцию **масштабирования** изображения. Эта операция сводится к пропорциональному изменению (увеличению или уменьшению) размеров виртуального окна с последующей привязкой физического прямоугольника и установкой позиций скроллинга. При этом необходимо придерживаться следующего правила. Если физическая точка изображения находится в фокусе (центре) окна устройства, то после масштабирования в фокусе экрана (по возможности) должна находится эта же точка. В этом случае восприятие увеличенного (уменьшенного) изображения является наилучшим. Итак, если необходимо увеличить (или уменьшить) изображение с заданным коэффициентом увеличения Inc, то первоначально необходимо зафиксировать физические координаты точки в фокусе окна устройства, используя функции $X(vX)$, $Y(vY)$ перехода от оконных координат к физическим (предполагая, что ранее привязка физического прямоугольника произведена):

$$OldCentX = X(wFocX),$$
$$OldCentY = Y(wFocY);$$

где как и для виртуального окна координаты фокуса окна устройства имеют вид:

$$wFocX = (wX_1 + wX_2) \operatorname{div} 2,$$
$$wFocY = (wY_1 + wY_2) \operatorname{div} 2.$$

Далее изменяем размеры виртуального окна и совмещаем верхние левые углы виртуального окна и окна устройства:

$$vDx = Inc * vDx;$$
$$vDy = Inc * vDy;$$
$$vX_1 = wX_1;$$
$$vY_1 = wY_1;$$
$$vX_2 = wX_1 + vDx - 1;$$
$$vY_2 = wY_1 + vDy - 1.$$

После этого производим привязку физического прямоугольника к виртуальному окну (которое, напомним, включает «усечение» границ окна для корректного определения диапазонов скроллинга). Затем устанавливаем позиции скроллинга таким образом, чтобы переместить в фокус окна устройства физическую точку, которая располагалась там ранее:

$$sPosX = vX(OldCentX) - wFocX;$$
$$sPosY = vY(OldCentY) - wFocY.$$

В случае, если позиция скроллинга $sPosX$ лежит вне рассчитанного диапазона скроллинга от 0 до $sRangeX$ (подобное может произойти только при уменьшении изображения), определяем это значение ближайшим из граничных значений 0 или $sRangeX$. Аналогично поступаем при определении позиции скроллинга $sPosY$.

Таким образом, если произвести операцию первоначальной привязки, далее можно неограниченное число раз увеличивать и уменьшать изображение (операцию масштабирования), а также «прокручивать» изображение (операция скроллинга) описанным выше способом, причем в произвольной последовательности. Для отображения результатов при этом достаточно после каждой операции просто произвести прорисовку набора отрезков в окне устройства, с использованием вновь полученных функций преобразования координат $vX(X)$, $vY(Y)$.

Далее остановимся на том, до каких пределов можно увеличивать и уменьшать изображение. Начнем со второго. Если в результате масштабирования оказывается, что виртуальное окно целиком содержится внутри окна устройства ($vDx < wDx$ и $vDy < wDy$), то изображение не полностью использует рабочую область окна (разрешение окна), что, очевидно, лишено смысла. Поэтому в этом случае достаточно просто заново «вписать» отображение в окно, то есть произвести операцию первоначальной привязки. Ограничения же на увеличение изображения связаны с разрядностью параметров (типа `int`) в процедурах Windows API, используемых при прорисовке отрезков (`MoveToEx`, `LineTo`) и управлении полосами прокрутки (`ScrollBar`). Тем самым имеют место ограничения на возможный диапазон изменения оконных координат. В Win32 тип `int` ассоциируется с 32-битным целым числом со знаком, имеющим диапазон изменения от –2147483648 до 2147483647. Однако такой «запас» заведомо избыточен, поэтому достаточно ограничить максимальное значение ширины vDx и длины vDy виртуального окна каким-либо значением, обеспечивающим достаточное для практических целей увеличение. Для совместимости соответствующего программного модуля с более ранними версиями Windows и MSDOS вполне достаточно в качестве такового выбрать число 32767. Такое ограничение вполне разумно, поскольку обеспечивает более чем тридцатикратное увеличения на реальных устройствах вывода.

На практике оказывается удобным помимо операций масштабирования и скроллинга иметь возможность указывать прямоугольные фрагменты уже видимой части изображения с последующим их увеличением на всю рабочую область окна устройства («вырезание» фрагментов). Такая операция в свою очередь легко сводится к операции масштабирования. Действительно, коэффициент находится несложно:

$$Inc = \min(wDx/cDx, wDy/cDy),$$

где cDx, cDy - ширина и высота прямоугольного фрагмента. Позиции скроллинга определяются точно так же, как и в случае масштабирования, за исключением того, что значения

$$OldCentX = X(cFocX),$$
$$OldCentY = Y(cFocY)$$

вычисляются в фокусе вырезаемого фрагмента.

Осталось обсудить последнюю операцию со «сдвоенным окном», а именно **изменение размеров окна** устройства. Необходимость такой операции продиктована хотя бы тем, что в операционной системе Windows пользователь имеет возможность изменять размеры окна на мониторе. При изменении размеров окна устройства меняется его разрешение, а также, в общем случае, соотношение его ширины и высоты. Поэтому получить в окне с новыми размерами в точности то же изображение, что было до изменения размеров, вообще говоря, не представляется возможным. Тем не менее, возможно получить изображе-

ние фрагмента с тем же увеличением, который находился «в фокусе» окна устройства до изменения размеров.

Для обработки изменения размеров вводится еще один параметр виртуального окна $GlobInc$, который отражает текущее увеличение изображения. При операции первичной привязки он полагается равным 1, а при каждой операции масштабирования изменяется следующим образом:

$$GlobInc = Inc * GlobInc.$$

Перед изменением размеров окна устройства вычисляются значения

$$OldCentX = X(wFocX),$$
$$OldCentY = Y(wFocY),$$
$$OldGlobInc = GlobInc.$$

После этого производится операция первичной привязки, а далее операция масштабирования с параметром $Inc = OldGlobInc$ и ранее вычисленными параметрами $OldCentX$, $OldCentY$. В остальном операция масштабирования производится также, как описывалось ранее.

В результате, в окне с новыми размерами будет изображено «примерно то же самое», что и до изменения размеров. То есть, та часть изображения, которая была видима ранее, останется видимой, и, быть может, видимой окажется еще какая-то его часть. Если при изменении размеров соотношение ширины и высоты окна устройства не изменилось (изменилось только разрешение), изображение окажется идентичным предыдущему с точностью до качества прорисовки.

Описанная операция изменения размеров окна на самом деле имеет гораздо более широкое применение. К ней, например, можно свести печать видимой части изображения на печатающем устройстве или сохранение изображения в графический файл.

Рассмотрим, например, задачу печати видимой в окне монитора части изображения на печатающем устройстве. Простейший вариант решения этой задачи состоит в копировании изображения рабочей области окна монитора на печатающее устройство, что называется «пиксель в пиксель». То есть, на печатающем устройстве выделяется область с тем же разрешением, что и рабочая область окна, после чего пикселям на печатающем устройстве приписывается цвет в соответствии с цветом пикселей рабочей области окна. Такой подход неприемлем по ряду причин. Во-первых, если пиксельный аспект монитора и принтера не совпадают, в напечатанном изображении нарушатся физические пропорции. Во-вторых, если разрешающая способность печатающего устройства существенно выше чем у монитора, изображение получится попросту слишком мелким. В-третьих, при таком способе печати, как правило, большая разрешающая способность печатающего устройства фактически не используется для получения более качественного изображения.

Таким образом, оправдано считать, что задача печати состоит не в копировании окна монитора, а в прорисовке в некотором окне на печатающем устройстве того же фрагмента изображения, что отображен в окне монитора. То есть, как и в операции изменения размеров окна здесь необходимо получить изображение того же фрагмента с тем же увеличением в окне с другими размерами, правда, на другом устройстве, имеющем быть может другой пиксельный аспект. А поскольку установка пиксельного аспекта ранее предусмотрена в операции первичной привязки, то все действия с виртуальным окном точно такие же, как при простом изменении размеров окна. Различия появятся, очевидно, лишь на стадии прорисовки, которую обсудим далее.

Как вытекает из вышесказанного, после любой рассмотренной операции со «сдвоенным окном» определяются функции $vX(X)$, $vY(Y)$, осуществляющие переход от физических координат к оконным. Осуществляя прорисовку с использованием этих оконных координат, то есть, передовая координаты в качестве параметров процедурам Windows API,

строится требуемое изображение. Однако специфика Windows API состоит в том, что всем подобным процедурам дополнительно передается ссылка на структуру данных, определяющую само устройство, на котором происходит прорисовка, так называемый контекст устройства (`Device Context`). В эту структуру входят графические объекты (`Pen`, `Brush`, `Font`, `Bitmap`, `Palette`, `Region`, `Path`) и графические режимы (`Background`, `Drawing`, `Mapping`, `Polygon-fill`, `Stretching`). Атрибуты графических объектов и графические режимы, собственно, и определяют, каким образом будет осуществляться графический вывод на устройство с использованием функций API «в данном контексте» (отсюда и название).

Перед использованием того или иного устройства достаточно «информировать» `GDI` (`Graphic Device Interface`) о необходимости загрузить драйвер устройства (`Device Driver`) и провести его инициализацию для последующей прорисовки. Результатом этой операции, в частности, является создание соответствующего контекста устройства, который и используется в дальнейшем при вызове функций Windows API. Описанный механизм лежит в основе аппаратной независимости Windows API и позволяет создавать универсальные приложения без привязки к техническим особенностям аппаратуры.

Тем самым, если дополнить операцию изменения размеров окна переопределением (в случае необходимости) контекста устройства, то к этой операции сводится печать изображения на печатающем устройстве, а также сохранение изображения в графический файл. В последнем случае используется контекст, не связанный напрямую с каким либо устройством (`Memory Context`), предназначенный для создания растрового изображения (`BitMap`). Прорисовка в растр, связанный с контекстом в памяти, осуществляется точно также как в окне устройства. Далее растровое изображение может быть сохранено в виде графического файла в том или ином формате.

Таким образом, описанный набор операций со «сдвоенным окном» позволяет полностью решить задачу отображения, сформулированную в начале данного пункта. При этом предложенный метод, хотя и описывался применительно к операционной системе Win32, на самом деле привязан к операционной системе в минимальной степени. Действительно, для применения метода при отображении на экране монитора достаточно, чтобы операционная система обеспечивала возможность прорисовки на экране отрезка прямой линии, а также позволяла производить отсечение (см. выше). А поскольку такие возможности имеют место во всех, известных автору, операционных системах, метод можно признать достаточно универсальным. Тем не менее, возможности Win32, как следует из описанного выше, позволяют использовать метод наиболее эффективно, поскольку работа со всеми устройствами вывода может быть осуществлена однородным образом. Кроме этого, при отображении линий уровня можно существенно повысить эффективность прорисовки, используя дополнительные функции Windows API.

Напомним, что результатом работы модифицированного алгоритма построения линий уровня, описанного в предыдущем пункте, является набор ломаных (элементарных кривых), заданных физическими координатами своих вершин. Используя функции $vX(X)$ и $vY(Y)$, каждую ломаную можно преобразовать в ломанную в оконных координатах, после чего произвести прорисовку ломаной целиком (функция Windows API `PolyLine`). Оказывается, это позволяет многократно повысить скорость прорисовки по сравнению с вариантом, когда каждый отрезок прорисовывается индивидуально.

В следующей таблице приведены средние отношения времен, затрачиваемых на прорисовку ломаной линии в окне на мониторе целиком (`PolyLine`) и по отдельным отрезкам (`LineTo`), в зависимости от количества звеньев и их длины. Длины звеньев выбирались случайным образом в диапазоне от 1 до ширины ячейки прорисовки (см. п.1). Также случайно выбирались координаты вершин ломаной.

Количество отрезков	Ширина ячейки прорисовки (в пикселях)			
	2-5	10	25	50
5	4.5	4.5	4.4	4.0
10	8.3	8.2	7.9	5.0
25	19	19	11	**5.6**
50	33	26	**12**	5.9
100	50	**28**	12	5.9
250	**55**	32	12	5.9
500 и более	**58**	32	12	5.9

Применительно к окну на мониторе значения ширины ячейки прорисовки 2-5 пикселей характерны для больших значений глубины прорисовки (8-10), при этом характерная длина ломаной составляет 250-500 звеньев. Здесь выигрыш в скорости наиболее существенен (более чем в 50 раз). Жирным шрифтом также отмечены значения, характерные для глубины прорисовки 7, 5-6 и 4 соответственно. Из этих данных вытекает, что при изображении картины линий уровня «среднего качества» (глубина прорисовки 5-6) прорисовка линии целиком оказывается на порядок быстрее, и этот показатель существенно увеличивается с ростом глубины прорисовки.

Замечание. Здесь и далее тестирование проводилось на персональном компьютере с операционной системой Windows XP Professional SP1, имеющем следующую аппаратную конфигурацию: материнская плата ASUS P4C800, процессор P4-2800, двухканальная оперативная память 512 Mb (3200 Gb/s), видеоадаптер на основе чипа GeForce FX 5200 (128 Mb), жесткий диск 7200 rpm Ultra ATA/100.

И, наконец, последнее, о чем целесообразно упомянуть в данном пункте, это каким образом осуществляется отображение линий уровня функции, заданной в прямоугольной области полярной системы координат (круговой сегмент или кольцо). Как отмечалось ранее, процесс построения линии уровня в этом случае не изменится, однако в результате будет построен набор ломаных, вершины которых будут заданы в полярных координатах. На этапе прорисовки в качестве физического прямоугольника достаточно взять минимальный прямоугольник, целиком покрывающий круговой сегмент, а координаты вершин ломаных преобразовать к декартовым координатам. Далее прорисовка проводится точно так же, как описано выше.

п.3. Построение линий уровня сеточной функции

Метод построения линий уровня функции двух переменных, предложенный в п.1, предполагает задание функции в каждой точке прямоугольной области декартовой или полярной системы координат. Поэтому обобщение метода на случай сеточной функции сводится к построению на основе сеточных данных функции непрерывных аргументов. В данном пункте рассмотрим случай ортогональной неравномерной сетки.

Под двумерной ортогональной неравномерной сеткой $\omega_h = \omega_h^x \times \omega_h^y$, заданной в прямоугольнике $[a,b]\times[c,d]$, далее будем подразумевать декартово произведение одномерных неравномерных сеток вида:

$$\omega_h^x = \{x_i; i = 1,2,\ldots n; a \le x_1 < x_2 < \ldots < x_{n-1} < x_n \le b\},$$

$$\omega_h^y = \{y_j; j = 1,2,\ldots m; c \le y_1 < y_2 < \ldots < y_{m-1} < y_m \le d\}.$$

Отдельно выделим два частных случая равномерной одномерной сетки:

$\omega_h^x = \{x_i = a + (i-1)h; i = 1,2,\ldots n; h = (b-a)/(n-1)\}$ - узлы на границах ячеек;

$\omega_h^x = \{x_i = a + (i-0.5)h; i = 1,2,\ldots n; h = (b-a)/n\}$ - узлы в центрах ячеек.

Будем считать, что в каждом узле (x_i, y_j) сетки ω_h определено значение сеточной функции $f_{i,j}$. Построим по этим данным функцию непрерывных аргументов (x, y) путем простейшей (линейной по каждой переменной) интерполяции:

$$f(x,y) = (1-\alpha_y)((1-\alpha_x) f_{i,j} + \alpha_x f_{i+1,j}) + \alpha_y((1-\alpha_x) f_{i,j+1} + \alpha_x f_{i+1,j+1}),$$

где $x \in [x_i, x_{i+1}]$, $y \in [y_j, y_{j+1}]$, а веса значений сеточной функции имеют вид:

$$\alpha_x = (x - x_i)/(x_{i+1} - x_i),$$
$$\alpha_y = (y - y_j)/(y_{j+1} - y_j).$$

Отметим, что выбор простейшей интерполяции оправдан тем, что алгоритм построения линий уровня предполагает многократное вычисление значений функции, и более сложные способы интерполяции могут существенно понизить его эффективность. Кроме того, при таком выборе способа интерполяции отсутствует какое-либо «сглаживание» сеточной функции, что позволяет в конечном счете визуализировать результаты расчетов без привнесенных изменений.

Таким образом, для вычисления функции $f(x, y)$ в произвольной точке (x, y) прямоугольника $[a,b]\times[c,d]$ необходимо для каждой одномерной сетки ω_h^x и ω_h^y найти номера узлов i, такой, что $x \in [x_i, x_{i+1}]$, и j, такой, что $y \in [y_j, y_{j+1}]$, соответственно. Простейший вариант решения этой задачи поиска состоит в последовательном переборе узлов:

```
i:=1;
while (x>x[i])and(i<n) do i:=i+1;
i:=i-1;
```

Здесь для наглядности приведен фрагмент программы на языке `Pascal`. Также можно решать задачу поиска с помощью метода половинного деления:

```
iLeft:=1;
iRight:=n;
while (iRight-iLeft)>1 do begin
 iCentr:=(iLeft+iRight) shr 1;
 if X<X[iCentr] then iRight:=iCentr else iLeft:=iCentr;
end;
i:=iLeft;
```

Тестирование на «заливке» (о которой речь пойдет в следующем пункте) показало, что метод половинного деления существенно превосходит простой перебор по скорости. Если на грубых сетках различия между двумя методами отсутствуют, то на сетке 100x100 достигается ускорение поиска в 5 раз, на сетке 200x200 почти на порядок (!), и этот показатель существенно возрастает с дальнейшим увеличением размерности сетки.

Дополнительно можно повысить эффективность вычисления функции за счет оптимизации формулы для ее вычисления по числу умножений:

$$f(x,y) = f_{i,j} + \alpha_x(f_{i+1,j} - f_{i,j}) + \alpha_y(f_{i,j+1} - f_{i,j} + \alpha_x(f_{i,j} + f_{i+1,j+1} - f_{i+1,j} - f_{i,j+1}).$$

В заключение отметим, что можно обобщить метод построения линий уровня и на случай более сложных сеток, например, треугольных. Для этого в каждой треугольной ячейке сетки можно использовать линейную интерполяцию функции $f(x, y)$, а задача по-

иска сводится к нахождению ячейки, содержащей заданную точку (x, y). Задачу поиска здесь также можно решать как простым перебором, так и более эффективным способом. Для повышения эффективности поиска область, покрытая треугольной сеткой, предварительно разбивается на одинаковые прямоугольные подобласти, число которых является параметром алгоритма. Для каждой подобласти формируется список тех треугольных ячеек, которые имеют с подобластью общие точки. В дальнейшем для поиска ячейки сетки сначала определяется подобласть, содержащая точку (x, y) (2 умножения), после чего перебираются только ячейки из соответствующего списка. Такой способ оптимизации вполне оправдан, поскольку предварительную операцию разбиения и создания списков необходимо осуществить один раз, а обращение к функции будет происходить многократно.

п.4. Цветовая «заливка»

Наряду с построением линий уровня для визуализации функции двух переменных часто применяется так называемая цветовая «заливка». Суть этого способа визуализации состоит в следующем. Предположим, имеется устройство вывода, поддерживающее окрашивание пикселей в различные цвета из некоторого набора (палитры). Функция $f(x, y)$ задана в прямоугольнике Ω декартовой системы координат. Вводится разбиение отрезка от $\min\limits_{\Omega} f(x, y)$ до $\max\limits_{\Omega} f(x, y)$ на диапазоны, число которых совпадает с количеством цветов в палитре, и устанавливается соответствие между диапазонами и цветами. Теперь, если сопоставить каждому пикселю в окне устройства соответствующую подобласть области Ω, то каждый пиксель можно окрасить в цвет, соответствующий диапазону, в котором находится среднее значение функции в данной подобласти. При достаточно большом количестве цветов в результате получится изображение по информативности не уступающее, а то и превосходящее, картину линий уровня функции, построенную в том же окне устройства, поскольку в случае заливки информацию о функции отображает каждый пиксель, то есть разрешение окна используется с максимальной эффективностью. Кроме того, программная реализация здесь крайне проста, что также является немалым достоинством. Однако очевидным недостатком цветовой заливки являются высокие требования к цветопередаче устройства вывода, в то время как для изображения линий уровня достаточно воспроизведения двух различных цветов.

Метод «сдвоенного окна», описанного в п.2, также может быть эффективно использован для реализации заливки. При этом в качестве физического прямоугольника попрежнему задается прямоугольник Ω, а все операции (первичной привязки, скроллинга, масштабирования и изменения размеров окна) осуществляются точно также как и для случая отображения линий уровня (или произвольного набора отрезков). Отличия возникнут, очевидно, лишь на стадии прорисовки. Прорисовку же можно осуществить совсем просто. Действительно, по окончании любой операции со сдвоенным окном, т.е. перед прорисовкой, всегда определены функции перехода от оконных координат к физическим $X(vX)$, $Y(vY)$. Если палитра и соответствие между цветами и диапазонами значений функции определены заранее (например, по линейному закону), то для прорисовки достаточно осуществить следующие действия. Для каждого пикселя с координатами (wX, wY) окна устройства, то есть прямоугольника (wX_1, wY_1, wX_2, wY_2), вычисляются физические координаты (X, Y), где $X = X(wX)$, $Y = Y(wY)$. (В случае, когда область Ω и функция f заданы в полярных координатах, на этом этапе точка (X, Y) приводится к полярной системе.) Если точка (X, Y) не принадлежит физическому прямоугольнику Ω, что возможно, когда изображение не полностью заполняет рабочую область окна устройства (см. п.2), то пиксель с координатами (wX, wY) окрашивается цветом фона. В противном слу-

чае вычисляется значение $f(X,Y)$, и пиксель окрашивается цветом, соответствующим диапазону, которому принадлежит вычисленное значение. При этом описанная процедура по-прежнему не зависит от того, на каком именно устройстве вывода происходит прорисовка. Поэтому без ограничения общности далее будем считать, что речь идет об окне на мониторе и называть его просто окном.

Таким образом, из вышесказанного вытекает, что реализация заливки с использованием метода «сдвоенных окон» или без этого казалось бы не вызывает дополнительных вопросов. Однако это не совсем так. Действительно, пусть рабочая область окна имеет разрешение 800x700. Тогда необходимо 560000 раз произвести операцию окрашивания пикселя в определенный цвет, и кроме того, столько же раз нужно воспользоваться функциями $X(vX)$, $Y(vY)$, 2 операции умножения, и вычислить значение функции $f(X,Y)$, 5 умножений (в оптимизированном варианте) и две операции поиска индекса (см. п.3). Таким образом, в процессе заливки производятся как вычисления, так и многократные вызовы процедур прорисовки пикселя (в Windows API `SetPixel` и более быстрая `SetPixelV`), скорость работы которых напрямую зависит от возможностей аппаратуры, драйвера устройства и `GDI` (см. п.2). Тем самым итоговую скорость заливки в основном будет определять более медленный из двух процессов (прорисовка или вычисления). Тестирование, о котором речь пойдет далее, показало, что прорисовка пикселей непосредственно в окне занимает около 80% процентов времени (для сеток порядка 200x200), необходимого на заливку в целом. Учитывая это обстоятельство, а также то, что уменьшить временные затраты на вычисления в данном случае не представляется возможным, основные усилия необходимо сосредоточить именно на оптимизации прорисовки. При этом увеличение скорости прорисовки на порядок даст выигрыш в общей скорости заливки в 3,5 раза, а это уже вполне достойная внимания цель. Ниже будет описано, каким образом в Win32 можно увеличить скорость прорисовки более чем в 200 раз, что дает возможность пятикратно увеличить скорость заливки, и, фактически позволяет полностью устранить влияние процесса прорисовки на итоговую производительность.

Перед тем, как приступить к поиску возможных путей оптимизации, необходимо обсудить, почему попиксельная прорисовка в окне по-прежнему является достаточно медленной операцией. Ведь в современных персональных компьютерах, как правило, устанавливаются видеоадаптеры, которые имеют собственные процессоры обработки команд (графические ускорители), взаимодействуют с основным процессором по выделенной шине данных AGP (Accelerated Graphics Port), и к тому же оптимизированы для работы с Win32. Дело в том, что аппаратное ускорение эффективно функционирует в том случае, когда команда графическому адаптеру сводится к выполнению большого числа однотипных операций. Казалось бы, при заливке это как раз то, что нужно, поскольку многократно производится одна и та же операция изменения цвета пикселя. Однако графический адаптер как раз и «не знает» о том, что данная операция производится многократно. Ведь при вызове функции `SetPixelV` всякий раз дается команда, и передаются данные, достаточные для прорисовки всего лишь одного пикселя, и неизвестно какая команда последует далее. Поэтому «ускорять» здесь, в общем, и нечего. Отсюда следует вывод о том, что для оптимизации необходимо сначала накопить информацию об изменении цветов упорядоченного набора пикселей (в данном случае прямоугольника рабочей области окна), а затем передать эти данные для «оптовой» прорисовки (быть может по частям).

В Win32 реализовать этот способ можно, используя первоначальную прорисовку в растр (`BitMap`) совместимого с данным устройством контекста (см. п.2) в оперативной памяти (`Memory Context`). Для иллюстрации этой возможности приведем небольшую подпрограмму на языке Паскаль, параметрами которой являются контекст устройства (`DC`), а также ширина и высота рабочей области окна (`Width`, `Height`).

```
procedure DrawIndirect (DC:hDC; Width,Height:Integer);
var MemDC:hDC;
    OldBitMap,MemBitMap:hBitMap;
begin
 MemDC:=CreateCompatibleDC(DC);
 MemBitMap:=CreateCompatibleBitMap(DC,Width,Height);
 OldBitMap:=SelectObject(MemDC,MemBitMap);
{Вызов функций SetPixelV с парамтром MemDC для прорисовки пикселей}
 BitBlt(DC,0,0,Width,Height,MemDC,0,0,SRCCOPY);
 OldBitMap:=SelectObject(MemDC,OldBitMap);
 DeleteObject(MemBitMap);
 DeleteDC(MemDC);
end;
```

Для создания контекста в памяти используется функция Windows API `CreateCompatibleDC`. Параметром данной функции является контекст устройства (`DC`), а результатом совместимый контекст в памяти (естественно, передаются и возвращаются ссылки). В результате работы данной функции, в частности, инициализируется графический объект `BitMap` во вновь созданном контексте, который изначально имеет ширину и высоту в один пиксель. Чтобы осуществить дальнейшую прорисовку, необходимо задать необходимую ширину и высоту растра, в данном случае совпадающие с размерами рабочей области окна. Однако сделать это напрямую нельзя, поскольку любые изменения графических объектов в контекстах осуществляются в Win32 путем создания новых объектов и указания их контексту для дальнейшего использования. В рассматриваемом случае с помощью функции `CreateCompatibleBitMap` необходимо создать совместимый с устройством растр, где параметрами являются контекст устройства и требуемые размеры растра, а результатом - ссылка на созданный в памяти совместимый с устройством графический объект `MemBitMap`. При этом совместимость заключается в том, что представление пикселей и палитра цветов в этом объекте соответствует формату, используемому для устройства. Далее с помощью функции `SelectObject` вновь созданный объект `MemBitMap` устанавливается в качестве растра контекста `MemDC`, при этом сохраняется ссылка на растр `OldBitMap`, использовавшийся ранее. Далее, с помощью функции `SetPixelV` с параметром `MemDC`, осуществляется прорисовка не непосредственно на устройстве, а в растре, размешенном в оперативной памяти. По окончании прорисовки с помощью функции `BitBlt` осуществляется «быстрое» копирование созданного в памяти растра в окно на устройстве. На этом этапе и предполагается достигнуть прироста эффективности за счет максимального использования возможностей драйвера устройства и аппаратного ускорения. Дальнейшие действия, осуществляемые в приведенном выше примере, необходимы для корректного завершения работы подпрограммы. Сначала в качестве растра контекста `MemDC` вновь указывается прежний объект `OldBitMap`, затем с помощью функции `DeleteObject` удаляется созданный в подпрограмме графический объект `MemBitMap`, и, наконец, с помощью функции `DeleteDC` удаляется контекст `MemDC`. Указанные действия необходимы для высвобождения ограниченных ресурсов GDI и сводятся к принципу «сделай все, как было».

Как показало тестирование (см. далее) при использовании копирования предварительно сформированного растра на графическое устройство (функция `BitBlt`) общая скорость прорисовки увеличивается приблизительно в 2,5 раза. При этом полезным побочным эффектом является отсутствие так называемого «дрожания экрана», поскольку собственно функция `BitBlt` исполняется за время порядка тысячных секунды, и обновление изображения производится незаметно для глаза. Остальное же (по-прежнему существенное) время уходит на большое число вызовов «медленной» функции `SetPixelV`. Таким образом, «бутылочным горлом» теперь становится не прорисовка на устройстве, а первоначальное формирование растра.

Как следует из вышесказанного, для дальнейшей оптимизации необходимо иметь возможность менять цвета пикселей в растре напрямую без использования функции SetPixelV. Казалось бы, прямой доступ к пикселям реализовать не так уж и сложно, ведь функция CreateCompatibleBitMap, используемая в приведенной выше подпрограмме, возвращает ссылку на созданную в памяти структуру типа tagBITMAP. Последняя в свою очередь содержит ссылку bmBits на массив, биты элементов которого представляют цвета пикселей. Изменяя соответствующие биты элементов массива, можно тем самым изменять цвета пикселей растра. Однако проблему составляет то обстоятельство, что способ задания соответствия между битами массива и цветами пикселей является специфическим для каждого устройства и оптимизирован не для удобства доступа, а для эффективного отображения растра на это устройство. Такие растры называются DDB (Device-dependent Bitmaps). По этой причине для прямого доступа к пикселям используются растры DIB (Device-Independent Bitmaps), формат которых жестко определен и не привязан к какому-либо устройству. Структура растра DIB достаточно проста. Растр содержит структуру типа BitMapInfo, в которой содержатся, в частности, данные о ширине и высоте растра, а также о количестве бит, представляющих цвет пикселя. Далее следует массив цветов, элементами которого являются четверки байт типа RGBQUAD содержащие RGB (Red, Green, Blue) представление цвета (1 байт зарезервирован). Это таблица цветов, задающая соответствие между номером цвета в массиве и его значением в RGB представлении. Далее следует битовый массив, задающий цвета пикселей растра, причем цвет пикселя может задаваться номером в таблице цветов. Например, если используется растр для 16-ти цветного изображения, массив цветов содержит 16 элементов, для представления индекса в массиве достаточно 4 бит, и количество бит, представляющих пиксель, также равно 4. (Возможен и альтернативный формат растра, связанный с логической палитрой устройства, который здесь рассматривать нецелесообразно.) В простейшем случае, когда растр представляет изображение True Color, используется 24-битное представление пикселя, и нет необходимости использовать таблицу цветов, поскольку пиксель можно явным образом задать RGB-представлением цвета. Этот формат растра наиболее удобно использовать для реализации «быстрого» доступа, поскольку для изменения цвета пикселя в этом случае достаточно заполнить три соответствующих байта в битовом массиве.

Для создания растра DIB используется функция CreateDIBSection, которой в качестве параметра передается предварительно заполненная структура BitMapInfo. Поля этой структуры определяют характеристики создаваемого растра, в том числе ширину, высоту и количество бит на пиксель, которое задается равным 24. Также процедуре передается параметр ppvBits, в котором возвращается адрес ссылки на битовый массив, представляющий цвета пикселей (остальные параметры для дальнейшего изложения несущественны). В дальнейшем для доступа к конкретному пикселю необходимо учесть, что тройки байт, определяющие цвет RGB, упорядочены по строкам растра (Scan Line) от нижней к верхней (bottom-up), если изначально задано положительное значение для высоты растра (порядок обратный, если это значение отрицательно). Данные для каждой строки выравниваются по границам двойного слова, т.е. дополняются незначащими байтами таким образом, чтобы общее количество байт, представляющее строку было кратно 4. Таким образом, число байт на строку BytesPerLine рассчитывается следующим образом:

```
(Width*3+3)and(-4).
```

Зная это значение, несложно найти адрес тройки байт для пикселя с номером I в строке J (нумерация ведется с 0), а именно:

```
pvBits+(Height-J-1)*BytesPerLine+I*3.
```

Также необходимо учесть, что байты RGB расположены последовательно в следующем порядке: Blue, Green, Red (т.е. реально BGR, однако RGB - общепринятое обозначение).

Таким образом, прямой доступ к пикселям позволяет осуществить «быстрое» формирование растра при использовании DIB. Осталось обсудить, как производится прорисовка сформированного растра на устройстве. Функция BitBlt, о которой речь шла выше, в данном случае неприменима, однако можно использовать функцию StretchDIBits. Данная функция осуществляет копирование произвольного прямоугольника растра DIB в произвольный прямоугольник на устройстве, задаваемом своим контекстом. При необходимости осуществляется масштабирование, а в случае совпадения размеров прямоугольника копирование производится «один в один» с точностью до преобразования цветов к цветам устройства, если оно необходимо. Данная функция является более медленной, чем BitBlt, однако работает достаточно быстро (см. далее), когда размеры прямоугольников совпадают.

Далее приведем результаты тестирования цветовой заливки, реализованной с использованием прямого доступа к пикселям, обсуждавшегося выше. Основным показателем здесь является скорее не абсолютное время заливки, а количество генерируемых кадров (Frame Rate) в секунду fps (frame per second). Тестирование проводилось в окне с разрешением рабочей области 800x700 пикселей на мониторе персонального компьютера, конфигурация которого описывалась в п.2. Сначала приведем данные для случая, когда тестировались различные варианты заливки, но вычислений цвета пикселя не производилось, то есть отсутствовали вычисления значений сеточной функции, поиск цвета в палитре, и цвет задавался одинаковым для всех пикселей.

1. Напрямую в контекст устройства (SetPixelV)	1 fps
2. С использованием контекста в памяти (SetPixelV, BitBlt)	2,7 fps
3. Прямым доступом к пикселям DIB (StretchDIBits)	212 fps
Frame Rate для функции BitBlt	1260 fps
Frame Rate для функции StretchDIBits (без масштабирования)	303 fps

За счет использования третьего варианта заливки с использованием DIB и прямым доступом к пикселям удается уменьшить время, затрачиваемое собственно на прорисовку, более чем в 200 раз по сравнению с «прямым» способом заливки. Это позволяет фактически не учитывать затраты на прорисовку, которые в этом случае ничтожно малы по сравнению с затратами на вычисление цвета. Использование прорисовки через контекст в памяти для заливки не является столь эффективным, однако этот способ можно использовать при прорисовке линий уровня для устранения эффекта «дрожания экрана». Учитывая впечатляющую скорость работы функции BitBlt, к «накладным расходам» при этом придется отнести только дополнительное выделение памяти.

Далее приведем окончательные значения производительности для различных способов заливки. Тестирование проводилось на различных сетках в декартовой и полярной геометрии. Для поиска индексов сеточной функции использовался «быстрый» метод половинного деления, описанный в предыдущем пункте.

Сетка	Способ 1	Способ 2	Способ 3
19x9	0,88 fps	1,80 fps	6.8 fps
120x100	0,85 fps	1,75 fps	4,8 fps
210x310	0,84 fps	1,68 fps	4,5 fps

Таким образом, как и утверждалось ранее, за счет использования эффективного способа прорисовки можно существенно (более чем в 5 раз) увеличить скорость заливки и достигнуть итоговой производительности 4-5 fps на достаточно подробных сетках. Заметим также, что увеличение размерности сетки незначительно влияет на производительность, что обуславливается быстрым алгоритмом поиска индексов при обращении к сеточной функции. Это, в частности, позволяет производить операции с изображением (см. п.2) в «реальном времени» в том смысле, что задержка между поступлением команды и собственно изменением изображения (четверть секунды) вполне приемлема с точки зрения восприятия пользователя.

п.5. Представление сеточных данных

В этом пункте обсудим немаловажный вопрос о том, каким образом возможно осуществить обмен данными между вычислительной программой и программой, осуществляющей изображение. Здесь принципиально возможны два варианта. В первом варианте подпрограмма, осуществляющая изображение, встраивается непосредственно в код вычислительной программы, и отображение осуществляется в процессе счета. Достоинством этого способа является то, что данные, подлежащие изображению, доступны, и нет необходимости дополнительно заботиться об их передаче. К очевидным же недостаткам можно отнести простой вычислений во время построения изображения, что особенно существенно, если дополнительно реализованы интерактивные функции, предполагающие вмешательство пользователя. Второй вариант состоит в передаче данных от вычислительной программы к программе визуализации посредством файлового обмена. Это предполагает постобработку результатов расчета, однако, учитывая многозадачность операционной системы, можно обрабатывать уже готовые результаты параллельно с процессом вычислений. При этом, поскольку скорость реакций пользователя ничтожна по сравнению со скоростью обработки команд процессором, основное время работы программы визуализации уходит на ожидание следующей команды пользователя (если речь не идет о пакетных режимах работы), то есть программа в основном простаивает и не загружает процессор. Таким образом, производительность вычислений при параллельной обработке результатов фактически не снижается. Еще одним важным преимуществом этого способа является возможность многократного доступа к результатам, сохраненным в файлы.

Итак, далее будем исходить из того, что программа, осуществляющая трехмерные или двумерные расчеты (в общем случае нестационарные) в декартовой, цилиндрической или сферической системе координат, сохраняет результаты в виде серии файлов, соответствующих данным, вычисленным на различные последовательные моменты времени. Далее обсудим возможную структуру таких файлов.

Для большинства реальных задач вполне достаточным является представление и последующее сохранение вычислительных данных в формате чисел с плавающей точкой `Real*4`. Сохранять эти данные в файл можно как в бинарной форме, тогда каждое число занимает 4 байта, так и в текстовой, тогда требуется 14 байт (знак, восемь значащих цифр, точка, 4 символа на экспоненту). Для примера при сохранении данных одного временного слоя двумерного газодинамического расчета на неравномерной сетке размерности 420x620 получается бинарный файл размером более 6 Mb. Если в дальнейшем предполагается осуществлять обработку результатов, например, путем построения анимации, то одновременно необходимо хранить порядка тысячи таких файлов, а это уже 6 гигабайт дискового пространства. В текстовом формате каждый файл занял бы в 3,5 раза больше места, и потребовалось бы дисковое пространство в 21 гигабайт. Учитывая емкость существующих в настоящее время жестких дисков, такая разница, мягко говоря, ощутима. Кроме этого, доступ к данным в текстовом файле принципиально невозможно организовать с той же эффективностью как к данным файла бинарного (см. далее). Таким образом,

несмотря на удобство текстовых файлов (легко посмотреть содержимое), от их использования в данном случае, видимо, стоит отказаться.

Далее необходимо конкретизировать формат файла данных. При этом сложность состоит в том, что при разработке программы визуализации, претендующей на универсальность, жестко задавать этот формат не представляется возможным. Действительно, при проведении многомерных нестационарных расчетов вычислительная программа обычно изначально разрабатывается таким образом, чтобы данные временного слоя сохранялись через определенные промежутки расчетного времени. Это позволяет помимо проведения последующей обработки результатов продолжить расчет, например, после неожиданного отключения питания компьютера с момента времени, когда осуществлялось последнее сохранение (а не с начала). Если речь идет о расчетах сложных нестационарных задач, где время расчета исчисляется днями, такая возможность является нелишней. Отсюда следует, что в сохраненных файлах должна содержаться информация, достаточная для полного восстановления всех расчетных данных задачи и временного слоя. А поскольку на стадии разработки программы визуализации, какова будет эта задача не известно, фиксация формата входного файла приведет к необходимости двойного сохранения. То есть возникнет необходимость генерировать файлы внутреннего формата для целей вычислений и фиксированного формата для работы программы визуализации, а это нецелесообразно, учитывая большие объемы сохраняемых данных. По этой причине необходимо определить только общую структуру файла, которая была бы пригодна для сохранения данных произвольных трехмерных или двумерных расчетов. Программа визуализации при этом должна иметь возможность настройки на файлы конкретной задачи, «выбирая» из них только те данные, которые необходимы для ее работы.

Вполне разумным представляется принять следующее соглашение об общей структуре файла вычислительных данных. Будем считать, что бинарный файл формата `Real*4` состоит из четырех блоков:

1. Заголовок файла нефиксированной длины;
2. Трехмерный массив сеточных функций;
3. Координаты узлов сетки (для неравномерных сеток);
4. Прочие данные.

Заголовок файла содержит произвольный набор данных, в числе которых в любом порядке должны быть указаны: размерность сетки по каждому измерению (также в формате `Real*4`), границы расчетной области (концы отрезка для каждого измерения) и расчетное время (не обязательно). Далее следует единый трехмерный массив записей. Каждое поле записи является элементом трехмерного массива, представляющего сеточную функцию, задействованную в расчете. Таким образом, запись содержит значения всех сеточных функций в данном узле. Если расчеты проводятся на неравномерной сетке, за трехмерным массивом следуют одномерные массивы с координатами узлов по каждому из трех измерений. Далее следует блок, который попросту игнорируется программой визуализации, где вычислительная программа может размещать любые необходимые данные, которые по каким-то причинам неудобно разместить в заголовке.

Для настройки программы визуализации на файл такой структуры достаточно указать позиции (индексы) в файле следующих данных:

```
Размерность по переменной  1;
Размерность по переменной  2;
Размерность по переменной  3;
Граница области X1min;
Граница области X2min;
Граница области X3min;
Граница области X1max;
```

```
Граница области X2max;
Граница области X3max;
Начало трехмерного массива данных.
```

Из трехмерного массива данных для построения изображения в дальнейшем необходимо извлечь двумерную сеточную функцию, соответствующую слою трехмерной сеточной функции при фиксированном индексе по одной из переменных. Если дополнительно на картину линий уровня предполагается наложить изображение векторного поля (например в виде стрелок), то необходимо дополнительно извлекать еще две двумерных сеточных функции, определяющих компоненты векторного поля. Для этого задаются следующие данные:

```
Число элементов в записи;
Номер фиксированной переменной (1,2 или 3);
Номер слоя по фиксированной переменной;
Позиция в записи отображаемой функции;
Позиция в записи 1-й компоненты векторов;
Позиция в записи 2-й компоненты векторов.
```

Далее необходимо определить тип разностной сетки. Если сетка `равномерна`, то для каждой из двух нефиксированных переменных достаточно указать, как расположены узлы: `в центрах` или `на границах ячеек` (см. п.3). Если указано, что сетка `неравномерна`, то координаты узлов двух нефиксированных переменных считываются из соответствующих одномерных массивов в третьем блоке.

И, наконец, последнее, что необходимо задать для окончательной настройки программы визуализации, это систему координат в которой заданы узлы сетки (`полярная` или `декартова`).

Перечисленный набор настроечных параметров является достаточным для того, чтобы осуществить чтение данных, необходимых для работы программы визуализации, за один проход по файлу, что немаловажно, учитывая, что размеры файла могут быть существенными. При этом настройка на файлы конкретной вычислительной программы осуществляется однократно, поскольку настроечные параметры можно сохранить.

Ниже для наглядности приведем две подпрограммы, осуществляющие сохранение данных газодинамических расчетов на языке `Fortran`, который, как известно, наиболее эффективен для вычислений. В первом примере осуществляется сохранение данных трехмерного расчета на равномерной сетке в цилиндрической системе координат. Для этого примера будут приведены настроечные параметры для построения изображения в сечениях как вдоль, так и поперек цилиндрической оси (декартова и полярная системы координат соответственно). Во втором примере сохраняются данные двумерного расчета на неравномерной сетке. Будет приведен набор настроечных параметров, в частности, демонстрирующий, что двумерный случай вполне укладывается в предложенную общую структуру файла данных.

Файл ‘`Arrays3D.for`’:

```
Parameter (NR=120,NF=60,NZ=30)
Dimension R(0:NR+1,0:NF+1,0:NZ+1)
Dimension U(0:NR+1,0:NF+1,0:NZ+1)
Dimension V(0:NR+1,0:NF+1,0:NZ+1)
Dimension W(0:NR+1,0:NF+1,0:NZ+1)
Dimension P(0:NR+1,0:NF+1,0:NZ+1)
Common /Arrays3D/ R,U,V,W,P
```

Подпрограмма `SaveStep3D` (трехмерный вариант):

```
Subroutine SaveStep3D (cFileName)
Include 'Arrays3D.for'
Character*12 cFileName
```

```
Common /ClcInt/ iSvStep,iClcStep
Common /Aria/ R1,R2,F1,F2,Z1,Z2
Common /Steps/ Hr,Hf,Hz,Tau,Time
iSvStep=iSvStep+1
iFile=6
Open (iFile,File=cFileName,Form='Binary',Recl=4,Status='New')
rSvStep=iSvStep
rClcStep=iClcStep
rNR=NR
rNF=NF
rNZ=NZ
Write(iFile) rSvStep,rClcStep
Write(iFile) rNR,rNF,rNZ
Write(iFile) R1,R2,F1,F2,Z1,Z2
Write(iFile) Hr,Hf,Hz,Tau,Time
Do i=1,NR
 Do j=1,NF
  Do k=1,NZ
   Write(iFile) R(i,j,k)
   Write(iFile) U(i,j,k)
   Write(iFile) V(i,j,k)
   Write(iFile) W(i,j,k)
   Write(iFile) P(i,j,k)
  EndDo
 EndDo
EndDo
Close(iFile,Status='Keep')
Return
End
```

В этой подпрограмме помимо данных, необходимых для работы, программы визуализации в заголовке дополнительно сохраняются и другие данные, такие как номера шага по времени и шага сохранения, сами шаги по времени и пространственным переменным (r, φ, z). В запись единого трехмерного массива входят плотность, три компоненты скорости и давление.

Автор отдает себе отчет, что с точки зрения эффективности кода порядок перечисления индексов в тройном цикле приведенного фрагмента не является оптимальным. Поскольку массивы в `Fortran` хранятся по строкам, наиболее эффективным является обратный порядок индексов. Автор имел случай убедиться в том, что простая замена «неоптимального» порядка индексов в циклах на «оптимальный» во всей программе привела при компиляции с помощью оптимизирующего транслятора Compaq Visual Fortran 6.0 [11] к увеличению быстродействия почти в 2 раза(!). Однако в данном случае, учитывая, что сохранение производится относительно редко, то есть через достаточно большое количество временных шагов, разумно, во избежание путаницы, принять «естественный» порядок перечисления индексов при сохранении в файл.

Приведем первую группу настроечных параметров, которые не зависят от того, какой именно слой трехмерных данных подлежит визуализации:

Размерность по переменной 1	3
Размерность по переменной 2	4
Размерность по переменной 3	5
Граница области X1min	6
Граница области X2min	8
Граница области X3min	10

Граница области X1max	7
Граница области X2max	9
Граница области X3max	11
Начало трехмерного массива данных	17
Число элементов в записи	5
Равномерна сетка	
Узлы в центрах ячеек по переменной 1	
Узлы в центрах ячеек по переменной 2	

Для того, чтобы отображать сеточную функцию плотности и поле скоростей в проекции на слой сетки при фиксированной переменной z (например, при `k=1`), в полярной геометрии (r,φ) достаточно дополнительно установить следующие настроечные параметры:

Фиксированная переменная	3
Номер слоя по фиксированной переменной	1
Позиция в записи отображаемой функции	1
Позиция в записи 1-й компоненты векторов	2
Позиция в записи 2-й компоненты векторов	3
Полярная геометрия	

При отображении сеточной функции давления и поля скоростей в проекции на слой сетки при фиксированной переменной φ (например, при `j=10`) в декартовой геометрии (r,z) устанавливаются следующие значения параметров:

Фиксированная переменная	2
Номер слоя по фиксированной переменной	10
Позиция в записи отображаемой функции	5
Позиция в записи 1-й компоненты векторов	2
Позиция в записи 2-й компоненты векторов	4
Полярная геометрия	

Далее приведем пример подпрограммы для двумерного случая.

Файл ‘`Arrays2D.for`’:

```
Parameter (NX=620,NY=620)
Dimension R(0:NX+1,0:NY+1)
Dimension U(0:NX+1,0:NY+1)
Dimension V(0:NX+1,0:NY+1)
Dimension P(0:NX+1,0:NY+1)
Dimension Xi(0:NX+1)
Dimension Yj(0:NY+1)
Common /Arrays2D/ R,U,V,P
Common /Grid/ Xi,Yj
```

Подпрограмма `SaveStep2D` (двумерный случай):

```
Subroutine SaveStep2D (cFileName)
Include 'Arrays2D.for'
Character*12 cFileName
Common /ClcInt/ iSvStep,iClcStep
```

```
Common /Aria/ X1,X2,Y1,Y2
Common /TimeStep/ Tau,Time
iSvStep=iSvStep+1
iFile=6
Open (iFile,File=cFileName,Form='Binary',Recl=4,Status='New')
rSvStep=iSvStep
rClcStep=iClcStep
rNX=NX
rNY=NY
Write(iFile) rSvStep,rClcStep
Write(iFile) rNX,rNY
Write(iFile) X1,X2,Y1,Y2
Write(iFile) Tau,Time
Do i=1,NX
 Do j=1,NY
  Write(iFile) R(i,j)
  Write(iFile) U(i,j)
  Write(iFile) V(i,j)
  Write(iFile) P(i,j)
 EndDo
EndDo
Do i=1,NX
 Write(iFile) Xi(i)
EndDo
Do j=1,NY
 Write(iFile) Yj(j)
EndDo
Close(iFile,Status='Keep')
Return
End
```

Для отображения плотности и компонент скорости настроечные параметры для такого файла имеют вид:

Размерность по переменной 1	3
Размерность по переменной 2	4
Размерность по переменной 3	0
Граница области X1min	5
Граница области X2min	7
Граница области X3min	0
Граница области X1max	6
Граница области X2max	8
Граница области X3max	0
Начало трехмерного массива данных	11
Число элементов в записи	4
Неравномерна сетка	

Фиксированная переменная	3
Номер слоя по фиксированной переменной	1
Позиция в записи отображаемой функции	1
Позиция в записи 1-й компоненты векторов	2
Позиция в записи 2-й компоненты векторов	3

При реализации чтения файла данных в программе визуализации достаточно предусмотреть действия, когда в качестве настроечных параметров появляется нулевая позиция (данные отсутствуют). Например, в простейшем случае вычислительная программа может сохранять всего одну сеточную функцию, тогда отображение векторного поля не производится, и в качестве позиций компонент вектора в записи задаются нулевые значения.

Таким образом, предложенная структура файла данных является достаточно универсальной и может быть использована для хранения данных любого трехмерного расчета. Не составляют исключение и текстовые данные, которые всегда можно разместить в массиве, объявленном как `Real*4`, и сохранить в последнем блоке файла, который может быть использован произвольно.

Как уже отмечалось, скорость доступа к данным в бинарном файле является существенным параметром, определяющим итоговую эффективность программы визуализации. Особенно важным этот аспект становится в том случае, когда осуществляется потоковая обработка расчетных данных, например, при генерации видео на основе кадров цветовой заливки. Если кадры строятся по сеточной функции, полученной в ходе расчетов на последовательные моменты времени (см. п.6), то это предполагает многочисленное обращение к файлам данных, и количество обрабатываемых кадров в секунду будет напрямую зависеть от скорости этого обращения.

В Win32 доступ к файлам можно осуществить несколькими способами. Во-первых, есть возможность использовать встроенные средства языка программирования. В языке `Pascal`, например, достаточно описать файловую переменную и производить доступ к записям в файле следующим образом:

```
procedure ReadFromFile(FileName:String);
var FileReal4:file of Single;
    RealRec:Single;
begin
  AssignFile(FileReal4,FileName);
  ReSet(FileReal4);
  while not Eof(FileReal4) do Read(FileReal4,RealRec);
  CloseFile(FileReal4);
 end;
```

Однако данный способ оказывается крайне медленным, причем настолько, что все усилия по увеличению эффективности заливки теряют всякое значение, поскольку чтение данных занимает время, в разы (см. таблицу далее) превышающее время последующей обработки.

Альтернативный способ доступа к данным файла сводится к использованию функции Windows API `ReadFile`. Достоинством этой функции является возможность организации блочного чтения данных из файла, используя промежуточный буфер в оперативной памяти, ссылка на который передается функции в качестве параметра. Конечно, это требует дополнительных усилий при реализации чтения из файла, однако, как будет видно из дальнейшего, это вполне оправдано.

В следующей таблице приведены результаты тестирования двух вариантов доступа к файлу, причем для функции `ReadFile` варьировались различные значения размера промежуточного буфера. Тестирование проводилось в процессе чтения файлов данных двумерных газодинамических расчетов (см. выше) на сетках 420x620 (размер файла около 6 Мб) и 210x310 (размер файла около 1Mb). Количество файлов выбиралось таким образом, чтобы общий объем данных превышал 1 гигабайт (для усреднения влияния кэширования диска средствами операционной системы). Результаты приведены в терминах количества обрабатываемых файлов в секунду (аналогично fps).

Размер буфера	210x310 (1Mb)	420x620 (6Mb)
отсутствует (Pascal)	0.76	0,13
32b	11	2,0
64b	16	2,9
128b	21	4,0
256b	32	7,4
512b	35	8
1024b	38	8,3
2048b	40	8,3
4096b	41	8,3
8192b и более	41	8,3

Как следует из данных, приведенных в таблице, использование функции ReadFile и промежуточного буфера позволяет повысить скорость чтения файла по сравнению с первым вариантом более, чем в 50 раз (!). При этом нет необходимости использовать промежуточный буфер больших размеров. Вполне достаточным оказывается размер 2-4 Kb, а дальнейшее его увеличение не приводит к повышению производительности (даже если использовать буфер, размер которого превышает размер файла).

п.6. Генерация видео

В настоящее время активно развиваются так называемые мультимедийные технологии. Это объясняется тем, что мощности современных персональных компьютеров достигли уровня, когда в реальном времени стало возможным осуществлять кодирование и декодирование звука и видео, что позволяет проводить оцифровку, хранение и воспроизведение соответствующих данных непосредственно на компьютере. Среди таких технологий важное место занимают методы сжатия видеоданных. Наверное, у всех на слуху такие термины как MPEG2 (от Moving Picture Experts Group) - стандарт сжатия, используемый при записи дисков DVD, и MPEG4 - стандарт, появившийся не так давно, используемый для сжатия данных мультимедиа. Не претендуя на глубокие знания в области сжатия видео, отметим лишь то, что одна из основных идей, лежащих в основе этих методов, состоит в следующем. Из входного видеопотока, представляющего собой последовательность кадров, выбирается первый кадр (key frame). Далее для каждого следующего кадра тем или иным способом, в зависимости от метода сжатия и его параметров, фиксируются изменения относительно предыдущего (важно, чтобы эти данные требовали места для хранения меньше, чем сам кадр). Этот процесс осуществляется для заданного числа кадров, и соответствующая информация вместе с опорным кадром записывается в выходной видеопоток. После обработки этой порции кадров следующий кадр вновь выбирается в качестве опорного, и процесс повторяется. Указанный процесс преобразования входного видеопотока в выходной называется кодированием (encoding). Для последующего воспроизведения видеоданных, которое сводится к отображению кадров с заданной частотой (Frame Rate), необходимо обратное преобразование сжатого видеопотока в последовательность кадров, то есть декодирование (decoding). Отметим, что кодирование можно осуществить и вообще с одним опорным кадром (первым), однако в этом случае для просмотра кадра с заданным номером (или видео с заданного момента времени) придется осуществлять декодирование сначала сжатого видеопотока, что затрудняет позиционирование (time seeking).

Программы, осуществляющие кодирование и декодирование, называются кодеками (codek сокращение от coder-decoder). В Win32 эти программы устанавливаются аналогично драйверам устройств и выступают в роли преобразователя между несжатым и сжатым видеопотоками (последний обычно и сохраняется в файл). Каждый кодек имеет собственные настроечные параметры, определяющие соотношение размера файла и качества изображения (хорошее качество исключает малый размер и наоборот). В большинстве кодеков задается среднее количество бит данных, необходимых для представления секунды видео в сжатом видеопотоке (BitRate). Прочие внутренние параметры устанавливаются автоматически таким образом, чтобы полученный размер файла максимально соответствовал этому значению. Отметим, что различные кодеки поддерживают различные наборы форматов кадра, однако большинство из них поддерживают полноцветные (True Color) кадры, ширина и высота которых кратна восьми.

В Win32 как сжатые, так и несжатые видеопотоки могут храниться в файлах формата AVI (audio-video interleaved). Файлы AVI могут содержать несколько согласованных по времени (time-based) потоков данных (data streams). Как правило, это один видео, несколько аудио потоков (например, различные варианты звукового сопровождения) и титры на различных языках. Для дальнейшего важно, что эти файлы могут содержать и один видеопоток, причем если он является сжатым, то информация о кодеке также хранится в файле.

Для реализации операций с файлами формата AVI в Win32 служит библиотека AVIFile. Далее речь пойдет о том, какими конкретно функциями этой библиотеки необходимо воспользоваться, чтобы сгенерировать файл видео, сжатый тем или иным кодеком, из последовательности кадров, построенных, например, путем цветовой заливки. Отметим, что эти функции, как, впрочем, и другие функции Windows API, доступны в любой среде программирования под Win32, в том числе и в среде Compaq Visual Fortran [11], из справочного руководства которой и позаимствована существенная часть сведений, приводимых далее.

Перед началом использования функций библиотеки AVIFile необходимо произвести первичную инициализацию путем вызова функции AVIFileInit (функция не имеет параметров). Далее требуется открыть файл AVI, в который в дальнейшем будет сохраняться сжатый видеопоток. Для этого используется функция AVIFileOpen. Функция возвращает адрес ссылки на открытый файл (ppFile), а ее параметрами являются имя файла (*.avi), режим доступа к файлу (OF_Write or OF_Create or OF_Share_Exclusive) и адрес идентификатора класса обработчика данного типа файла (nil - устанавливается стандартный обработчик, исходя из расширения файла .avi).

Замечание. Под термином «функция возвращает» обычно подразумевается значение функции, однако, учитывая, что большинство функций библиотеки AVIFile *в этом смысле слова возвращают код успешного завершения, разумно здесь пользоваться этим термином иначе, имея в виду механизм обмена данными через параметры.*

Следующим шагом является инициализация (открытие) видеопотока и его привязка к ранее открытому файлу с использованием функции AVIFileCreateStream. Параметрами этой функции являются: ссылка на открытый файл (pFile) и ссылка на структуру типа AVIStreamInfo, содержащую параметры открываемого потока данных. Функция возвращает адрес ссылки на интерфейс открытого потока (ppAVI). Поля предварительно размещенной в памяти структуры AVIStreamInfo, можно изначально заполнить нулями, после чего определить только необходимые для дальнейшей обработки видеопотока значения:

```
AVIStreamInfo.dwScale:=1;
AVIStreamInfo.dwRate:=FrameRate;
AVIStreamInfo.fccType:= $73646976; {streamtypeVIDEO,'vids'}
```

```
AVIStreamInfo.rcFrame.Right:=Width;
AVIStreamInfo.rcFrame.Bottom:=Height;
```

Первые два поля совместно определяют `FrameRate` (`dwRate/dwScale` обычно 25fps), следующее поле указывает на то, что открывается именно видеопоток, далее указываются размеры кадра.

Два следующих действия связаны исключительно с организацией сжатия видео. Если предполагается сохранять несжатый видеопоток, то их можно не осуществлять. В противном случае сначала необходимо выбрать кодек и установить его внутренние параметры (см. выше). Для этого используется функция `AVISaveOptions`, которая отображает стандартный диалог выбора кодека. В диалоге выводится список всех кодеков, установленных в системе, и предоставляется возможность выбора кодека из списка и установки его параметров. Параметрами функции являются: ссылка на вызывающее диалог окно приложения (`hWnd`); дополнительные опции (`uiFlags`), влияющие на конфигурацию диалога (`0`); количество потоков данных (`nStreams`), для которых необходимо определить кодеки (`1`); адрес массива указателей на интерфейсы потоков (в данном случае `ppAVI`); адрес массива ссылок на структуры типа `AVICompressOptions`. В этих структурах возвращаются данные о выбранных кодеках для каждого потока данных. В рассматриваемом случае кодек выбирается для одного видеопотока, поэтому достаточно передать адрес ссылки (`ppCompOpt`) на одну такую структуру, предварительно размещенную в памяти. После вызова указанной функции необходимо освободить ресурсы, выделенные в ходе ее работы, используя функцию `AVISaveOptionsFree` с параметрами `nStreams` (`1`) и `ppCompOpt`.

В результате предыдущего действия фактически происходит только заполнение структуры, определяющей кодек и его параметры, размещенной по адресу `pCompOpt`. Поэтому далее необходимо дополнительно инициализировать сжатый видеопоток на основе этих данных и ранее открытого входного видеопотока. Для этого используется функция `AVIMakeCompressedStream`. Ей в качестве параметров передается ссылка на интерфейс входного потока (`pAVI`), ссылка на данные о кодеке (`pCompOpt`) и адрес идентификатора класса (по-прежнему `nil`). Функция возвращает адрес ссылки на интерфейс инициализированного сжатого видеопотока (`ppsCompressed`). На этом этапе по сути устанавливается некий «посредник» между входным видеопотоком и файлом `AVI`, который осуществляет кодирование данных и их трансформацию в сжатый видеопоток, поступающий непосредственно в файл.

Для завершения подготовки к процессу кодирования видео необходимо установить формат видеопотока, который, в частности, будет сохранен в файле `AVI`. Для этого необходимо применить функцию `AVIStreamSetFormat`. Параметрами данной функции являются открытый поток (сжатый или несжатый, в данном случае `psCompressed`), позиция в потоке для получения формата (всегда `0`, поскольку обработчик файлов `AVI` не поддерживает смену формата), адрес структуры, содержащий формат (`lpFormat`), и размер структуры, содержащий формат (`cbFormat`). Последний параметр необходим, поскольку данная функция применяется не только для видеопотоков, а структуры форматов различных потоков данных отличаются друг от друга. Для видеопотока используется структура, имеющая тип `BitMapInfoHeader`. Ее также достаточно после размещения в памяти инициализировать нулями, после чего установить следующие поля:

```
biSize:=SizeOf(BitMapInfoHeader);
biWidth:=Width;
biHeight:=Height;
biPlanes:=1;
biBitCount:=24;
biSizeImage:=BytesPerLine*Height;
```

Здесь первое поле задает размер самой структуры, два последующих - размер растров, которые в дальнейшем будут передаваться в качестве кадров, количество цветовых планов, количество бит на представление цвета пикселя (в данном случае TrueColor) и размер битового массива растров (см. п.4). После вызова процедуры с указанными параметрами все готово для кодирования последовательности кадров заданного формата. Само же кодирование осуществляется в процессе последовательной передачи кадров в сжатый видеопоток. В качестве кадров здесь выступают растры DIB установленного формата, а именно такие растры используются для «быстрой» генерации заливки (см. п.4), так что дополнительных преобразований кадров не потребуется.

Для передачи очередного кадра в ранее открытый видеопоток используется функция AVIStreamWrite. Ей передаются следующие данные: ссылка на открытый видеопоток (psCompressed), позиция в видеопотоке (lStart), количество кадров подлежащих передаче (lSamples), адрес буфера с данными кадров (lpBuffer), размер буфера (cbBuffer), флаг, ассоциированный с кадрами (dwFlags). Функция возвращает количество фактически переданных (записанных) кадров в переменной по ссылке plSampWritten и общее количество переданных байт (plBytesWritten). Если в качестве двух последних параметров установлено nil, то соответствующие данные не возвращаются. В рассматриваемом случае в качестве lStart необходимо задать номер текущего кадра, lSamples установить равным 1, в качестве lpBuffer передать ссылку на битовый массив растра DIB (pvBits), доступную после создания растра (CreateDIBSection), размер cbBuffer равен размеру битового массива растра BytesPerLine*Height (подробнее см. п.4). В качестве флага кадра передается значение AVIIf_KeyFrame ($10).

Осуществив последнюю операцию для всех последовательных кадров, необходимо завершить работу с видеопотоками, файлом AVI и библиотекой AVIFile. Для этого последовательно вызываются следующие функции:

```
AviStreamRelease(pAVI);
AviStreamRelease(psCompressed);
AviFileRelease(pFile);
AviFileExit;
```

Предназначение данных функции понятно из их названий, отметим только, что последняя функция высвобождает занятые функцией AVIFileInit ресурсы.

Приведенных сведений вполне достаточно для реализации в Win32 сохранения видео на основе генерируемых кадров формата DIB. В рассматриваемом случае кадры строятся путем цветовой заливки (см. п.4) прямоугольной области в соответствии со значениями сеточной функции, полученной в ходе численных расчетов на последовательные моменты времени (см. п.5). На производительность генерации видео здесь влияют следующие операции: считывание и подготовка сеточных данных, заливка (подготовкой растра DIB) и кодирование кадра (включая его передачу и сохранение данных в файл AVI). Вопросы, связанные с повышением эффективности первых двух операций подробно обсуждались выше. Скорость же кодирования существенно зависит от выбранного кодека и его параметров.

Для дальнейшего тестирования автором использовались два различных кодека, отвечающих стандарту MPEG4: DivX [9] и XviD [10]. Первый является коммерческим программным продуктом, второй распространяется свободно и является так называемым продуктом Open Source (исходный текст программы также распространяются свободно). Параметры кодеков выбирались таким образом, чтобы качество видео было максимально возможным. Для этого устанавливались параметры по умолчанию, после чего для кодека DivX изменялось значение BitRate (4000kbps), а для XviD выбирался параметр maximum quality. Размеры кадров устанавливались равными 720x576 (стандарт PAL) и

800x700 (этот размер кадра использовался для тестирования скорости заливки в п.4). Значение Frame Rate также устанавливался в соответствии со стандартом PAL (25fps).

В качестве исходных данных для генерации видео использовались результаты нестационарных газодинамических расчетов, полученные на различных сетках. Поскольку рассчитывалась одна и та же задача, динамика изменения кадров, влияющая на скорость кодирования, была примерно одинакова для всех вариантов тестирования. Формат соответствующих файлов приведен в предыдущем пункте (SaveStep2D).

Производительность генерации видео традиционно измеряется количеством обрабатываемых кадров в секунду frame per second (fps). Исходя из тестов скорости чтения файлов данных и генерации заливки, проведенных ранее, оправдано ожидать, что итоговая производительность заливки в среднем (т.к. зависит от разрешения кадра, сетки и кодека) окажется порядка 3-4 fps при достаточно подробном разрешении кадра. Более детальные результаты приведены в следующей таблице.

Формат кадра и кодек	Сетка 210x310		Сетка 420x620	
	Декартова	Полярная	Декартова	Полярная
320x240 кодирование XviD	18.2	15.3	6.1	5.7
320x240 кодирование DivX	19.0	16.2	6.2	5.8
320x240 без кодирования	21.1	17.8	6.3	5.9
720x576 кодирование XviD	4.8	3.8	3.2	2.6
720x576 кодирование DivX	5.7	4.4	3.4	2.8
720x576 без кодирования	6.5	5.0	3.7	3.1
800x700 кодирование XviD	3.8	3.0	2.8	2.2
800x700 кодирование DivX	4.5	3.3	2.9	2.4
800x700 без кодирования	5.1	3.8	3.1	2.6

Из приведенных данных следует, что усилия по повышению эффективности чтения сеточных данных, доступа к сеточной функции и заливки позволили сделать время генерации кадра сравнимым со временем его последующего кодирования. И если принять за скорость кодирования «в реальном времени» значение 25fps (Frame Rate при отображении видео в формате PAL), то достигнутые показатели не так уж и далеки от этого значения. В частности, на сетке 210x310 практически в «реальном времени» генерацию видео можно осуществить при разрешении кадра 320x240, которого в ряде случаев вполне достаточно.

Несложно также сделать вывод о том, что генерация видео в полярной геометрии является более медленной. Этот результат был вполне прогнозируемым, поскольку обусловлен необходимостью пересчета декартовых координат пикселя в полярные. Эта операция предполагает обращение к обратным тригонометрическим функциям и проводится для каждого пикселя, а потому занимает существенное время.

Выбор кодека не столь существенно влияет на итоговую производительность. В частности, кодек XviD и является более медленным, однако, по твердому убеждению автора, именно его и стоит использовать. Дело в том, что качество видео в результате его работы оказывается более высоким по сравнению с кодеком DivX (при «максимальных» параметрах качества), и кадр видеоизображения практически не отличается от оригинала. Кроме того, немаловажно, что этот кодек распространяется свободно [10].

Далее обсудим немаловажный вопрос о том, насколько ощутимым является сжатие видео. Для этого приведем следующие данные для случая обработки 1000 кадров с разре-

шением 720x576. Если сохранять эти кадры в файлы BitMap (без сжатия), то они займут на жестком диске пространство в 1215Mb, то есть больше гигабайта (!). Столько же займет файл AVI с несжатым видеопотоком. Файл с видеопотоком, сжатым кодеком XviD (с максимальным качеством), при этом занимает всего 6,5Mb, то есть речь идет о сжатии видеоданных почти в 200 раз (!) без потери качества.

В качестве заключения отметим, что описанная выше методика позволяет обрабатывать результаты расчета с минимальными затратами времени и ресурсов. Так для генерации полноформатного видео по результатам расчета в 10000 кадров (6,7 минут) необходимо в среднем около 40 минут и порядка 100Мб пространства на жестком диске для сохранения результата. По сравнению с пространством, занимаемым самими расчетными данными, это обычно несущественно. Отметим, что альтернативный вариант, когда кадры сначала сохраняются на жесткий диск, а видео на их основе генерируется позже с использованием каких-либо дополнительных программ, требует существенно больше ресурсов. Так для сохранения 10000 кадров в разрешении 720x576 дополнительно потребовалось бы более 10 гигабайт дискового пространства, не говоря уже об усилиях и временных затратах на дальнейшую обработку кадров. Кроме того, подходящую программу для дальнейшей обработки кадров попросту необходимо иметь. Например, программа MovieMaker, входящая в состав операционной системы Windows XP, не справляется с задачей обработки даже 500 кадров, не говоря уже о нескольких тысячах. Из вышесказанного следует, что реализация обработки видео внутри программы визуализации открывает принципиально новые возможности, поскольку реальным становится создание полнометражных и полноформатных фильмов на основе обработки результатов вычислений.

Литература

1. Базаров С.Б., Баяковский Ю.М. Комплекс графических программ ГРАФОР в среде WINDOWS. – Препринт ИПМ РАН №30, 2000.

2. Базаров С.Б., Баяковский Ю.М., Сайдалиева Ф.Ф., Скачков А.Ю. Адаптация комплекса графических программ ГРАФОР в операционных системах WINDOWS и LINUX. – Препринт ИПМ РАН №27, 2002.

3. Andreas Sandquist. Dynamic Line Integral Convolution for Visualizing Streamline Evolution. – IEEE Transactions on Visualization and computer Graphics, 2003, pp. 273-282.

4. Абакумов М.В. Система хранения и визуальной идентификации иерархической структуры инженерных сооружений. – Свидетельство о регистрации программного средства в фонде алгоритмов и программ по ракетно-космической технике Российского космического агентства №4069 от 20.01.97.

5. Suffern K.G. Quadtree algorithms for contouring functions of two variables. – The Computer Journal, Vol.33, №5, 1990, pp.402-407.

6. Абрамов В.Г., Трифонов Н.П., Трифонова Г.Н. Введение в язык Паскаль. - М.:Наука,1988.

7. Соловьев П.В. Fortran для персонального компьютера. - М.:Арист, 1991.

8. Windows SDK (www.microsoft.com/msdn/sdk)

9. DivX Pro Codec 5.1.1, DivXNetworks (www.divx.com).

10. XviD MPEG4 Codec 1.01, группа независимых разработчиков (www.xvid.org).

11. Compaq Visual Fortran 6.6, Compaq Computer Corporation (www.compaq.com/fortran).