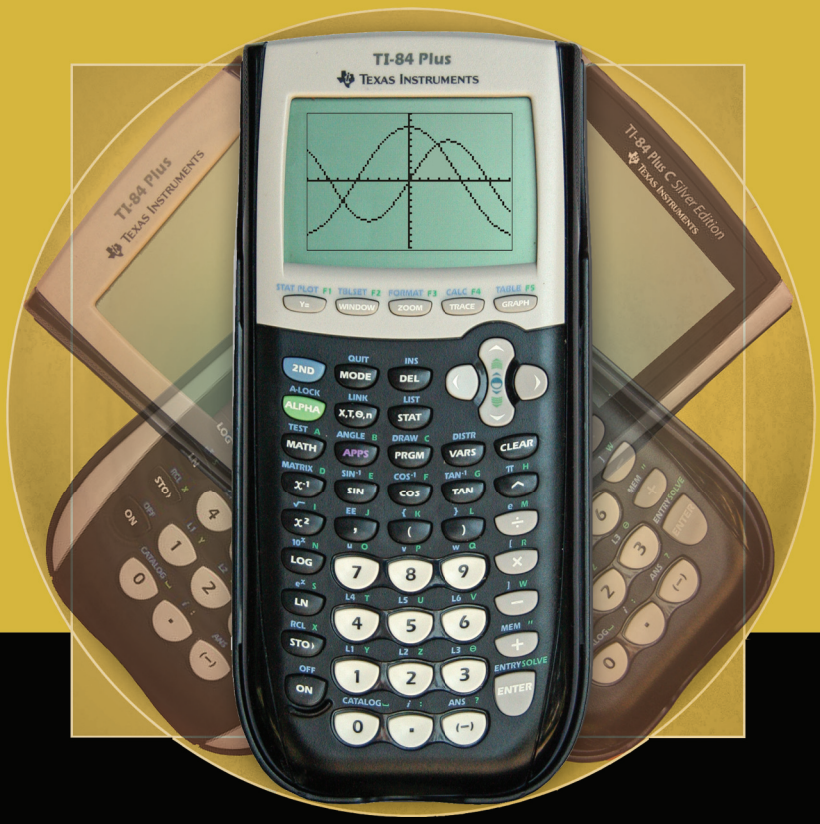


Lessons Learned from *Programming the TI-83 Plus/TI-84 Plus*

Christopher R. Mitchell

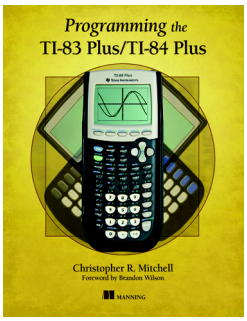
MANNING

Lessons Learned from *Programming the TI-83 Plus/TI-84 Plus*

CHRISTOPHER R. MITCHELL

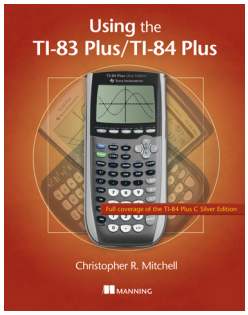
MANNING
SHELTER ISLAND

Save 40% on These Two Great Books!



Programming the TI-83 Plus/TI-84 Plus is an example-filled, hands-on tutorial that introduces programming with the TI-83 Plus and TI-84 Plus graphing calculators. This engaging and easy-to-read book immediately immerses you in your first programs and guides you concept by concept, example by example. You'll learn to think like a programmer as you use the TI-BASIC language to design and write your own utilities, games, and math programs.

$29.99 | 352 pages | Published September 2012



Using the TI-83 Plus/TI-84 Plus gives you a hands-on orientation to the calculator so you'll be comfortable with its screens, its buttons, and the special vocabulary it uses. Then, you'll start exploring key features while you tackle problems just like the ones you'll see in math and science classes.

$24.99 | 280 pages | Coming Summer 2013

This exclusive discount is available only at manning.com.
Just enter **13tisav** in the Promotional Code box when you check out.
There is no limit, so you can distribute this code
to parents and students or use it to purchase class sets.

For quantities over 20 copies, contact Candace Gillhoolley
(cagi@manning.com) for additional discount pricing.

Lessons Learned from
Programming the TI-83 Plus/TI-84 Plus

by Christopher R. Mitchell

Special edition eBook

contents

preface

When I was 13 years old, I received my first graphing calculator. It was Christmas, and my biggest present under the tree turned out to be a TI-83. I was thrilled. I first used it just for math, but over the course of a few months, I became more curious and discovered that I could write programs directly on the calculator. The calculator's guidebook didn't really help with programming, other than demonstrating an interesting Sierpinski Triangle demo. Undeterred, I set off to teach myself calculator programming, though I never thought of it in such definite terms.

I first learned to display text on the screen and then to make simple animations. I discovered that I could also ask the user for input and thus make simple math programs to check my homework results. Soon, classmates began passing around arcade games they had found for their calculators, so I dug into the source code for those games and found out how these worked, using my new skills to try my hand at some games of my own. Over the years I grew more competent, including learning to write z80 assembly, a more complex but much more powerful language than TI-BASIC. I started an online community around graphing calculator programming called Cemetech (pronounced "KEH-meh-tek") that thrives as a hardware and software development haven to this day. I continued to pursue programming as well as my lifelong love of hardware and electronics. I earned two degrees in electrical engineering and one in computer science; I'm now working toward my doctorate. I credit much of my love of programming and engineering to those first faltering steps with my graphing calculator.

Graphing calculator programming is a fun and rewarding way to get started with any kind of programming, to develop logic and problem-solving skills, or just to have

fun with the challenge of working with such a device. If you're a student or teacher, especially of math or science, the programs you write for your calculator can speed up annoying, repetitive calculations or help you check your work. You can enjoy the feeling of accomplishment from completing a useful utility or a fast-paced game for your calculator. Figure 1 illustrates a few projects from experienced calculator programmers.

But why write calculator programs; why not just jump straight to programming a computer? The short answer: the opportunity to learn fast, have fun, surmount the challenges of a programming platform, and get started right away. Chances are, you and your students already have graphing calculators if you're reading this book. If you don't, then you can get one for less than $100. The TI-83 Plus, TI-83 Plus Silver Edition, TI-84 Plus, and TI-84 Plus Silver Edition covered in this book are all cheap, widely available, and widely owned graphing calculators and can all run each other's programs. Calculators are small and portable, great to carry around and whip out

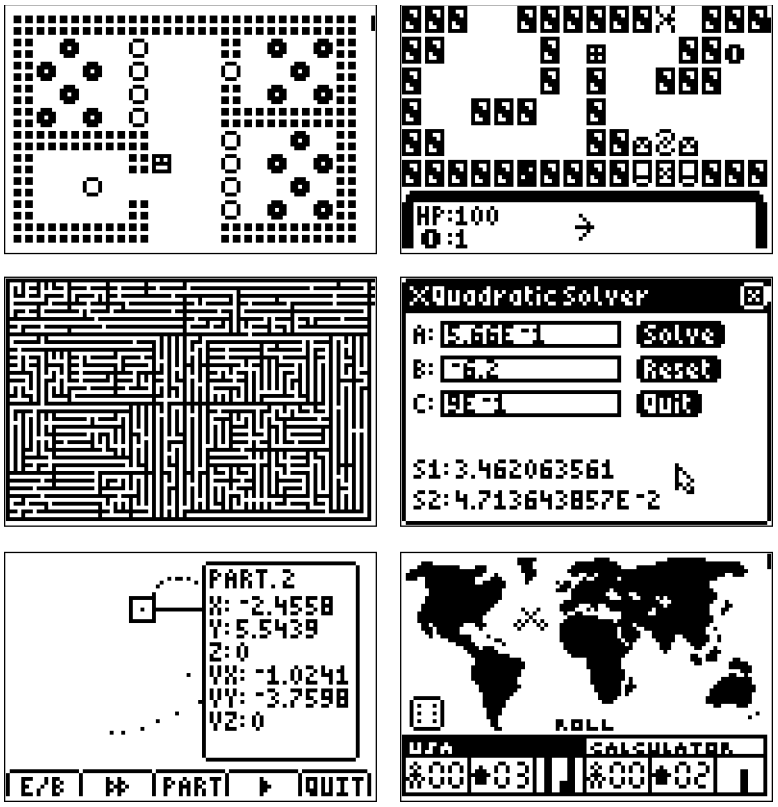


Figure 1 Screenshots from assorted examples of completed and freely available TI-BASIC calculator programs and games. Top row: "Donut Quest II" by Mikhail Lavrov, "Midnight" by Zachary Tuller. The middle and last rows both contain screenshots from programs by the author, Christopher "Kerm Martian" Mitchell. Middle row: "DFS Maze Generator" and "DCSQuad Solver"; bottom row: "ParSim Particle Simulator" and "World Domination I".

when you have some downtime to work on your programming but don't have or want to carry around a laptop. They last for months, not a few hours, on a single charge or set of batteries.

Programming is a fun and rewarding career or hobby. It's great to hone problem-solving skills and to learn to think more analytically. It's gratifying to develop an idea for a program and, after planning and hard work, to successfully bring that idea to fruition. Students may find that they enjoy the satisfaction of surmounting challenges, of learning to optimize their programs to make them small and fast, of sharing their finished work with friends and with users around the world.

Programming calculators is a great pursuit in its own right and teaches most of the skills necessary to easily pick up computer programming languages. Learning to program can spur a passion for STEM subjects. Many of the past and present graphing calculator programming stars started as bored or curious students and now have advanced degrees or high-paying jobs in programming and engineering. The book I wrote for Manning, *Programming the TI-83 Plus/TI 84 Plus*, will teach you everything you need to know to think like a programmer, instilling an intuition for translating an idea into a program and thinking your way around challenges that you'll find useful in a wide variety of technical pursuits. The three lesson plans in this special edition eBook are excerpted from that book.

To learn more or to order the book, please go to manning.com/mitchell. Purchase of the print book includes free eBook versions in PDF, Kindle, and ePub formats.

about this eBook

In this section you'll learn more about *Programming the TI-83 Plus/TI-84 Plus*, the book from which most of the material in this eBook is extracted. The eBook presents three sample lesson plans to go along with the sample book content included, though you are free to design your own lesson plans from these or any other sections of the full 13-chapter book:

- Sample lesson plan A: Get started with programming—is an introduction to calculator programming. Students learn to create a first Hello World program, then move on to a more advanced Quadratic Formula solver. A third program, a guessing game, can be presented in-class or used as a take-home assignment. Students are challenged to experiment with each program to add new features.
- Sample lesson plan B: Understanding event loops—presents a full game called Mouse and Cheese, including source code, program flow diagrams, and screenshots. It teaches students who have some TI-BASIC experience how to build a complete game from start to finish; it helps them understand the math and structure behind it. The material provides suggestions for extra features and modifications students can make.
- Sample lesson plan C: Programs and pixels—for experienced students teaches how to create interactive graphics by directly manipulating pixels. Students learn how graphics and text are composed of pixels on computer and calculator screens. They write and test two programs, one of which moves a mouse cursor around the screen, the other of which lets users paint on the screen.

Let's start with a look at the material covered in the full *Programming the TI-83 Plus/TI-84 Plus* book.

About Programming the TI-83 Plus/TI-84 Plus

This eBook will give you a taste of *Programming the TI-83 Plus/TI-84 Plus*, and how it could be a part of your classroom. You could create an entire course around teaching your students programming via graphing calculators, or you could integrate calculator-programming lessons into a math, science, or technology curriculum. This eBook picks out three sections from the book that form the basis for the sample lesson plans. Many of the sections in *Programming the TI-83 Plus/TI-84 Plus* include new skills followed by exercises and examples, making them easy to adapt into lessons.

Programming the TI-83 Plus/TI-84 Plus consists of 13 chapters, divided into three parts. It also has three appendices, which summarize skills, commands, and resources that any calculator programmer might need. Part 1 of the book focuses on introducing programming skills that are important for TI-BASIC programming but apply to almost any language you might want to learn.

- Chapter 1 introduces graphing calculators and calculator programming, outlining why learning TI-BASIC is important and relevant. It presents your first three programs: a Hello World program, a guessing game, and a quadratic equation solver.
- Chapter 2 presents input and output on the homescreen, including displaying text and numbers and getting strings and values from the user.
- Chapter 3 covers conditionals and comparisons, the building blocks for creating programs that make decisions.
- Chapter 4 completes the picture of controlling program flow in TI-BASIC with labels, loops, menus, and subprograms, all of the structural features that you'll need to create arbitrarily complex programs.
- Chapter 5 steps back to detail the process of designing, creating, and debugging a program in any language. It illustrates each step with a running TI-BASIC example.

Part 2 of *Programming the TI-83 Plus/TI-84 Plus* takes the basic framework from part 1 and teaches additional commands and features necessary for more professional and complete programs. These include graphics, interactivity, and the proper use of the many data types your calculator understands, such as matrices, lists, strings, and pictures.

- Chapter 6 teaches you how to create fun, interactive programs and games with event loops. As with many other lessons, it wraps the TI-BASIC focus in skills you'll be able to bring to many other languages you may explore. This chapter culminates in a full Mouse and Cheese game for your edification.
- Chapter 7 discusses your first true graphics tools, presenting the concepts and commands for turning individual pixels on and off. It shows how to draw small and large text anywhere on the screen and reinforces the lessons of the chapter with two demo programs: a painting tool and a mouse cursor subprogram.

- Chapter 8 expands further on graphics and graphing, covering creating and manipulating graphs from inside programs, as well as drawing with points, lines, circles, and other shapes. It introduces the commands for storing and recalling pictures on the graphscreen.
- Chapter 9 rounds out the second part of the book with an overview of the many data types your calculator can handle and the important commands for manipulating each. It walks through strings, lists, matrices, real and complex numbers, and random numbers, and it concludes with a complete framework for a role-playing game (RPG) that you can expand and enhance on your own.

The third and final part of *Programming the TI-83 Plus/TI-84 Plus* goes into advanced concepts and may be particularly engaging even if you have prior TI-BASIC or programming experience. It covers optimization, hybrid BASIC, and the rudiments of assembly.

- Chapter 10 details how to optimize your programs for speed and size, presenting TI-BASIC–specific tips without losing sight of the more general programming lessons for proper optimization.
- Chapter 11 shows hybrid TI-BASIC and the hybrid BASIC libraries and includes a discussion of the major libraries, where to find them, and how to use them.
- Chapter 12 introduces a new programming language, z80 assembly, giving you enough detail about binary, decimal, hexadecimal, and assembly commands and program flow to spur you to explore it more on your own.
- Chapter 13 concludes with ideas about where you can go with programming and calculator programming from here. It also discusses hardware development and hacking and how such a hobby ties into calculator programming.

The appendices provide a quick reference to material, supplementing and coalescing the contents of the chapters:

- Appendix A is a crash course in using your graphing calculator. Chapters 2 onward assume a basic set of general calculator skills, and appendix A reviews all of these skills in case you don't feel entirely comfortable with your device.
- Appendix B summarizes all of the commands found throughout the chapters and includes usage examples and syntax.
- Appendix C lists valuable resources for seeking programming help, finding additional programs for inspiration and source code examination, and tools to facilitate BASIC and assembly programming.

Prerequisites, curriculum, and resources

Who are you? This special eBook is aimed at teachers, but you might also be a student or a professional. If you (or your students) have never programmed anything before, then you or they have a whole world of amazing things that programming can enable

you to do and learn in front of you, and I'll be honored to guide you forward. This book is primarily aimed at the budding programmer. I'll lead you through graphing calculator programming, but I'll help you keep an eye on programming in general and teach you concepts you can apply to almost any language.

For those who have toyed with programming before, for calculators, computers, or another platform, I hope *Programming the TI-83 Plus/TI-84 Plus* can teach you how to learn more, to write and understand complete programs, and to have fun doing so. For advanced programmers, either for calculators or something else, I want to provide a great reference guide for calculator programming, advanced topics and optimization tricks, perhaps get you interested in z80 assembly programming, and give you another perspective on programming as a hobby and as a career.

Prerequisites and curriculum

I'll teach you everything you or your students need to know to write complete programs for graphing calculators. I assume that you have no prior knowledge of calculator programming or any sort of programming. I'll teach how to think like a programmer and how to apply problem-solving skills to take any program you might want to write, break it down into pieces, and code each one. The chapters ahead are designed to teach everything you need to know, from the basics up to the most advanced tricks for creating very fast, very small, very fancy programs.

The full set of skills the book teaches could be broken out into lesson plans, creating a full curriculum of in-class and take-home materials. Alternatively, select skills could be used as enrichment exercises in math, science, or programming classes, with the rest of the material assigned for independent study. For simple math programs and games, the first 4 or 5 chapters are sufficient for students to get started. Part 2, encompassing chapters 6 through 9, teaches using the keyboard and pixel-based graphics for powerful, interactive programs and games. A full introductory programming course built TI-BASIC course would likely need to encompass The final four chapters teach advanced skills that could either be offered as supplementary material advanced students or in a follow-up course.

Resources

The purchase of *Programming the TI-83 Plus/TI-84 Plus* includes free access to a private web forum run by Manning Publications, where you can make comments about this book, ask technical questions, and receive help from both the author and from other readers. The forum can be found at http://www.manning.com/mitchell. This page contains information on how to register on and use the forum, what kind of help is available, and the rules of conduct.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online Forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print. You can also ask technical questions on the author's forum, Cemetech, which has a special subforum for this book at http://www.cemetech.net/forum/f/70 (or http://cemete.ch/f70). Appendix C of *Programming the TI-83 Plus/TI-84 Plus* lists many more online resources, including places to download and publish programs, development tools, and emulators.

about the sample lessons

Although *Programming the TI-83 Plus/TI-84 Plus* was written to appeal to students, it can also be used as an effective teaching guide. This section lists objectives, materials and prerequisites, procedures, evaluation and assignments, and standards necessary to build lessons around each of the three excerpts included in this eBook.

Sample lesson plan A: Get started with programming

The first chapter of *Programming the TI-83 Plus/TI-84 Plus* gives students a crash-course in graphing calculator programming, introducing three programs: a Hello World program, a Quadratic Formula solver, and a number-guessing game. This lesson introduces students to basic programming concepts by understanding, programming, and using these three programs.

Duration: Two 45-minute to 1-hour sessions.

Audience: Grades 8-12

OBJECTIVES

Students will:

1. Learn to create and run programs on their graphing calculators
2. Discover the building blocks of simple programs while learning to read source code
3. Gain confidence in experimenting with and modifying existing programs

MATERIALS AND PREREQUISITES

- One TI-83, TI-83 Plus, or TI-84 Plus family graphing calculator per student, or per pair of students.

- Students should be familiar with graphing calculators as a math tool.
- Sections 1.2 through 1.4 of *Programming the TI-83 Plus/TI-84 Plus* (in this eBook).
- *(optional)* The full text of *Programming the TI-83 Plus/TI-84 Plus* for reference.
- *(optional)* Presentation methods like the TI-ViewScreen, a graphing calculator running on your computer (http://cemete.ch/emu), or the TI-SmartView software.

PROCEDURES

1. Introduce students to the concept of a graphing calculator as a pocket computer. As discussed and diagrammed in section 1.1 of *Programming*, graphing calculators and computers both have RAM, a processor (CPU), input (keyboard), output (screen), and a hard drive (Flash ROM on the calculator)
2. Explain what makes graphing calculators closer to computers than solar 4-function calculators: the ability to run new software to add new features (or solve math problems, or play games).
3. Ask students if they have programming experience, and if so, what languages they have explored, and what they have created. Introduce TI-BASIC, the language in which they will be programming their calculators.
4. (Optional, suggested for older students) Explain the value of programming as a life skill, and calculator programming as a fast, fun way to learn how to write programs.
5. If possible, demonstrate creating a new program named HELLO, typing in the source code for the Hello World program, and running the program. Refer to section 1.2 in *Programming* for full details.
   a. Use presentation methods like the TI-ViewScreen, a graphing calculator running on your computer (http://cemete.ch/emu), or the TI-SmartView software for this step.
6. Instruct students to try writing and running the program on their own calculators
   a. Students who complete the task quickly should be encouraged to try changing the text, or making the program display more text. Challenge students to experiment and not to worry about breaking the program.
   b. Incorrectly typed programs can be used to introduce the idea of debugging or troubleshooting programs (which is covered in detail in chapter 5 of *Programming*, and would make an edifying full lesson).
7. Discuss lessons learned from writing and running the program; use the students' insights on the program to transition to the second program, the Quadratic Formula solver. Explain that the program performs the same math you would do by hand to compute roots of quadratic equations, but more quickly and reliably.
8. Depending on the age and technical experience of the students, lead them step-by-step through typing the QUAD Quadratic Formula solver on their calculators, or let them work on it in small groups. The program may also be given as

a homework assignment if you feel the scope of the program would require too much class time.

   a. Programming intuition discussion—After all of the students have typed and tested the program, ask them to try to explain, in words, what the program does. See if they intuitively get the idea of the calculator holding a "pointer" to where it is in the program as it executes the program, and how each command it reads moves that pointer.
   b. (optional) Before or after the programming intuition discussion, walk them through the program line-by-line, explaining how it works.
   c. Math intuition discussion—See if they can explain how it solves the Quadratic Formula. How is its methodology similar to and different from how they would perform the same calculation by hand? What is the determinant, and how is it used in the program? What do variables A, B, and C do (from a programming standpoint and a math standpoint).

9. Instruct students to write and test the GUESS program on their calculators. Once again, encourage them to try to figure out what went wrong if their program doesn't work correctly (but provide help after they put in some effort to avoid frustration). Lead a discussion on how the program works, what a loop is, and how the program makes "decisions" about what line of code to execute next. Ground the discussion in the program flow diagrams in section 1.4 of *Programming*.
   a. Invite students to brainstorm how the GUESS program could be expanded and modified to work differently or to add additional features.

Evaluation and Assignments

The Quadratic Formula solver and/or the guessing game can be given as homework assignments. Students should write and test the program(s) on their graphing calculators. Students can show proficiency by demonstrating their programs in class. To test the students' absorption and ability to synthesize the material more fully, students can be asked to explain in words or diagrams how the program(s) work, and/or to present a modified version of the program. Students should not be expected to be able to substantially modify the program flow of these introductory programs.

Standards

- ISTE/NETS Standard 1a and 1b (Creativity and innovation)
  - Designing modifications to the three programs presented
- ISTE/NETS Standard 4c (Critical Thinking, Problem Solving, and Decision Making)
  - Deducing how each of the programs works
- ISTE/NETS Standard 5b (Digital Citizenship)
  - Collaborate with classmates to understand how the programs function
- ISTE/NETS Standard 6a and 6c (Technology Operations and Concepts)
  - Learn and understand programming concepts and logic, troubleshoot programs

Sample lesson plan B: Understanding event loops

By chapter 6 of *Programming the TI-83 Plus/TI-84 Plus*, students understand conditionals and Boolean logic, program flow, and loops. If they read and understood chapter 5, they know general programming skills for designing, planning, creating, and debugging a complete program. This lesson leverages those skills to let students write, understand, and modify a complete game, called Mouse and Cheese.

Duration: One or two 45-minute to 1-hour sessions. If one, students should be assigned the second half of the procedure (in which they experiment with expanding and modifying the Mouse and Cheese game) as homework.

Audience: Grades 8-12

Objectives

Students will:

1. Conduct a hands-on investigation of what an event loop is, and how event loops create interactive games and programs.
2. Improve their ability to read and understand source code.
3. Explore their creativity and burgeoning programming skills while designing and implementing new game features or rules.

Materials and prerequisites

- One TI-83, TI-83 Plus, or TI-84 Plus-family graphing calculator per student, or per pair of students.
- Students should be familiar with graphing calculators as a math tool. Students should have calculator-programming experience, ideally from chapters 1 throiugh 5 of *Programming*.
- Section 6.3 of *Programming the TI-83 Plus/TI-84 Plus* (in this eBook)
- *(optional)* The full text of *Programming the TI-83 Plus/TI-84 Plus* for reference.
- *(optional)* Presentation methods like the TI-ViewScreen, a graphing calculator running on your computer (http://cemete.ch/emu), or the TI-SmartView software

Procedures

1. Lead a brief review of students' graphing calculator programming skills. Verbally quiz them on the proper use of input and output commands, conditional statements, and loops. Guide them through reviewing the necessary steps for bringing a program from idea through completion (see chapter 5 of *Programming*).
2. Explain what the game they'll be writing and exploring does: let the user move a symbol representing a mouse around the screen chasing a piece of cheese. The cheese is placed at a random location, and each time the mouse eats the cheese, a new piece of cheese appears. As time passes, the mouse gets hungrier, and if the mouse starves, the game ends. The mouse's hunger is indicated by a bar at the right side of the screen.

   a. Make sure they understand the concept of using random numbers to decide where the cheese will appear each time. Motivate the discussion with the analogy of rolling an 8-sided and a 15-sided die (corresponding to the two `randInt(` commands).

3. Ask the students to type out the program on their calculator and test it.
   a. Advanced students may be asked to instead create the program from scratch without referring to the solution in the book, and to explain how they turned the description.
4. Facilitate a discussion among the students about how the program works, challenge students to ask questions about specific pieces of the code and answer each other's questions. For example, what does the `If K / Output(B,A,"[one space]` sequence do, and why is that a conditional?
5. Ask students to experiment with the program to significantly change the rules of the game, or to add a major new feature. Students should be prepared to explain why they picked the feature or rule they picked, how they designed the changes or additions to the code, and to demonstrate the result. This will likely require sufficient time to be assigned as homework or as a second in-class lab.
6. For an advanced or third session, students can explore the modification in section 6.4 of *Programming*, where the cheese moves randomly as the mouse is chasing it.

Adaptations

If students have TI-84 Plus C Silver Edition color-screen calculators, their homescreens are 26 columns and 10 rows instead of 16 columns and 8 rows. Challenge the students to modify the program to take the full screen on their calculators. Get them to make the mouse be able to move around the whole screen, to put the hunger bar at the far right, and to make the cheese be able to appear anywhere on the screen.

Discussion questions

- What is the initialization section for? Why is it outside the main program loop?
- The diagrams in the text say that a single big loop is bad, and two nested loops are good. Why? What makes the two solutions different? Amount of code, ease of programming, speed of the final program?
- What changes would be necessary to turn the `Repeat` loop into `While` loops? Why are `Repeat` loops used in the code provided?
- What's the difference between a non-blocking and blocking command? What is `getKey`? What about `Input`?

Evaluation

Students can be quizzed on a program similar in structure to the Mouse and Cheese game, or can be asked to write code for a simple program with similar structure. They can be asked to diagram the flow of such programs and/or to annotate the source code with descriptions and arrows showing how the program works and makes flow decisions.

STANDARDS

- ISTE/NETS Standard 1a and 1b (Creativity and innovation)
  - Designing modifications to the three programs presented
- ISTE/NETS Standard 2d (Communication and collaboration)
  - Students can work together to design and implement the new rules or features.
- ISTE/NETS Standard 4c (Critical Thinking, Problem Solving, and Decision Making)
  - Deducing how each of the programs works
- ISTE/NETS Standard 6a and 6c (Technology Operations and Concepts)
  - Learn and understand programming concepts and logic, troubleshoot programs

Sample lesson plan C: Programs and pixels

Chapter 7 of *Programming the TI-83 Plus/TI-84 Plus* teaches students about the graph-screen. Previously, they could only print characters on the screen; now they can manipulate individual pixels. This lesson lets them explore two such programs, one of which lets them moves a cursor around the screen and the other is a painting program.

Duration: One or two 45-minute to 1-hour sessions. If one, students should be assigned the second half of the procedure (in which they test the PAINT program) as homework.

Audience: Grades 8-12

OBJECTIVES

Students will:

1. Discover what pixels are, how to manipulate them, and the additional graphics abilities their programs and games gain with direct pixel manipulation.
2. Improve their confidence in creating a program from scratch given a specification, and troubleshooting their results.
3. Think critically about the positive and negative results of programming languages' features.

MATERIALS AND PREREQUISITES

- One TI-83, TI-83 Plus, or TI-84 Plus-family graphing calculator per student, or per pair of students.
- Students should be familiar with graphing calculators as a math tool. Students should have calculator-programming experience, ideally from chapters 1-5 of *Programming*, and should understand event loops and `getKey`, ideally from chapter 6.
- Section 6.3 of *Programming the TI-83 Plus/TI-84 Plus* (in this eBook)
- *(optional)* The full text of *Programming the TI-83 Plus/TI-84 Plus* for reference.
- *(optional)* Presentation methods like the TI-ViewScreen, a graphing calculator running on your computer (http://cemete.ch/emu), or the TI-SmartView software

PROCEDURES

1. Explain what a pixel is, and how to use the `Pxl-On(`, `Pxl-Off(`, and `Pxl-Change(` commands to directly manipulate pixels on the calculator's graphscreen.
2. Invite students to brainstorm what they can do differently in their programs with the ability to turn individual pixels on and off.
3. Demonstrate the CURSOR program in action, if possible, and explain the speed tradeoff that comes with directly manipulating pixels in TI-BASIC. Explain that this limitation is unique to the TI-BASIC language. If necessary, briefly remind students of the difference between an interpreted and compiled/native language.
4. Ask students to type out and test the CURSOR program (including the ZCURSORB subprogram) individually or in small groups. Challenge them to explain the differences between ZCURSORA and ZCURSORB, and what values for variables A and B would make ZCURSORB behave the same as ZCURSORA.
5. See if students can change the mouse cursor into a different sprite or shape. Students should first sketch out their design on paper, ideally graph/grid paper, then figure out the `Pxl-Change(` commands that will make their modification work.
6. If students have a TI-84 Plus C Silver Edition calculator, see if they can make their sprites more colorful, and if this adds any complications to erasing sprites.
7. Assign the PAINT program as homework, or as a second in-class lab. Students should write the PAINT program without referring to the code given in *Programming*. They may first be asked to simply move a pixel around the screen using `getKey` and an event loop, then add the ability to "raise" and "lower" the virtual pen so that the point either does or does not leave a trail behind it as it moves. Advanced students may also be asked to add a "snapshot" key to store the current drawing to a Pic variable, if they have read chapter 8 of *Programming*.

ADAPTATIONS

Students with TI-84 Plus C Silver Edition calculators will need to slightly modify these programs to work properly on their calculators. Encourage students to find out the similarities and differences between the graphscreen on the TI-83 Plus/TI-84 Plus and TI-84 Plus C Silver Edition calculators. They should design and test their planned modifications to the programs to see if they work and debug/troubleshoot problems that arise.

DISCUSSION QUESTIONS

- What is a pixel? Why is it important to understand how to manipulate individual pixels? Why not only use helper routines and APIs that can draw images and text? Who writes those APIs and routines?
- What is a sprite?
- What does `Pxl-Change(` do when you use it twice in the same spot? How can you apply the `Pxl-Change(` operation to use the same code to draw and

erase a sprite? What if you overlap two different sprites? How will they look? Can you still erase both sprites completely? If necessary, demonstrate this on a blackboard.

- Why are the `Pxl-Change(` commands put into a subprogram? Why would they have to be duplicated twice otherwise, and why is this bad? Is there any time it could be good to repeat the same code multiple times? (Hint: what if calling a subprogram is very slow?)

STANDARDS

- ISTE/NETS Standard 1a and 1b (Creativity and innovation)
  - Designing modifications to the three programs presented
- ISTE/NETS Standard 2d (Communication and collaboration)
  - Students can work together to design and implement the new rules or features.
- ISTE/NETS Standard 4c (Critical Thinking, Problem Solving, and Decision Making)
  - Deducing how each of the programs works
- ISTE/NETS Standard 6a and 6c (Technology Operations and Concepts)
  - Learn and understand programming concepts and logic, troubleshoot programs

Let's get started

This introduction outlined what you will find in this special edition eBook as well as what is covered in the print edition of *Programming the TI-83 Plus/TI-84 Plus*, from which most of the material in the eBook was excerpted. We started with a look at why your students should learn to program, from improving problem-solving and logic skills to sparking interest in STEM fields. I also explained why students should learn to program calculators specifically, including how it will help them build and improve a variety of skills. We looked at the lesson plans to be presented in this eBook, and a roadmap of the full *Programming the TI-83 Plus/TI-84 Plus* book. After a brief detour to cover the necessary prerequisites and helpful tools, you saw the three sample lesson plans covered in this eBook. I hope that you find this taste of *Programming the TI-83 Plus/TI-84 Plus* to be educational and inspiring, and that you impart a love of programming to your own students!

Sample lesson plan A Get started with programming

Just from using your graphing calculator for math, you already know some programming. The math operations in TI-BASIC programs are identical to the math operations you type at the homescreen, and with many operations, such as manipulating graphs, you can build off the skills you've already learned using your calculator for school or work. The programming commands have names taken directly from English, such as Input, Repeat, and many others. The calculator even makes it easy to track down your programming mistakes, taking you directly to errors it finds so that you can correct them.

In this lesson plan, you'll take your first programming steps, diving right in with your first three calculator programs. You'll learn to display the text "Hello, World" on your calculator's screen and then create a math program to solve the quadratic equation and a number-guessing game. Ready? Let's get started!

A.1 Hello World: your first program

No instruction in a new language would be complete without plenty of well-annotated example programs to demonstrate each new concept learned. To jump directly into TI-BASIC programming, this section shows you the TI-BASIC version of the simplest program imaginable, universally called Hello World because it prints that phrase on the screen. I'll present an overview of the two major types of programming languages, interpreted languages and compiled languages, while showing you TI-BASIC. You'll see the source code for the program, and I'll teach you how to test it on your own calculator. First, you need to know a few background details about the TI-BASIC language and how it compares to other languages you may know or have heard about.

A.1.1 Before you begin: notes on the TI-BASIC language

The programming language that's commonly known as TI-BASIC isn't officially called by any name by Texas Instruments itself and isn't technically a variant of the BASIC (Beginners All-Purpose Symbolic Instruction Code) language. But like BASIC, it's an interpreted language and shares many traits with that inspiration, so the name TI-BASIC has stuck.

Almost every language can be classified either as an interpreted or a compiled language; a high-level comparison of the two is provided in table A.1 along with a few representative examples of each. See the sidebars "What's an interpreted language?" and "What's a compiled language?" for more details.

Table A.1 Interpreted versus compiled programming languages

	Interpreted language	Compiled language
Execution speed	Slower	Faster
Preprocessing	None needed	Source code compilation
Syntax error checking	During execution	Before execution
Executed by	Interpreter program	Computer's processor
Examples	TI-BASIC, JavaScript, Java, Python	C, C++, Haskell, Fortran

For both types of languages, programmers type in the series of commands that will make up the program in a list of lines, a list called the program's source code. For both types, execution generally proceeds from the top of the program downward, although you'll see in section A.3 how conditional commands, loops, and jumps can redirect execution.

What's an interpreted language?

A calculator or computer directly reads these programs, interpreting on the fly what the program will do. It reads each line of the program, figures out what that line is directing it to do, acts on it, and moves to the next line. If there are syntactical errors, such as sequences of commands that don't make sense, missing pieces of commands, and the like, the interpreter won't find these until it reaches the error while running the program. Interpreted programs are generally slower than compiled programs, because the interpreter must translate each line of the program into a form the computer's processor can understand and make sure the line has no errors before it gives that line of the program to the processor.

You'll now see the first of three TI-BASIC programs meant to immerse you in the basics of the language. I'll present the source code of a Hello World program and explain it. I'll walk you through the steps to type it and test it on your own calculator. If you have

prior experience with TI-BASIC, some of the details in the coming examples may be extraneous, but you'll certainly still learn more about each command, its proper use, and special tricks and features of each as you read. Let's jump into your first program: Hello World.

What's a compiled language?

A compiled program goes through an intermediate process called compilation before being run. A compiler's job is to take the code the programmer has typed and convert it into a program that can be run directly by the computer or calculator's processor before it's run. Because the compiler must examine a program for errors and translate it, much as an interpreter does, it can find some programming errors during the compilation process. After they're compiled, these programs generally run faster, because they're directly executed by the processor with no interpreter spending processor time.

A.1.2 Displaying "Hello, World"

The source code for the Hello World program in TI-BASIC is among the simplest programs you can write, consisting of a single line of code. In any language, Hello World is traditionally the first program presented, and it shows "Hello, World" or some variation thereof on the screen; our version of this program is shown in action in figure A.1.

Figure A.1 Output of Hello World program

Even though it's a tiny toy program, it's useful for introducing the fundamentals of what a program is, how you create a program, and what happens when you run a program. Without further ado, here's the source code for Hello World.

```
PROGRAM:HIWORLD
:Disp "HELLO, WORLD"
```

Name of the program, not a line of code

A line of code, with a command (Disp) taking one argument ("HELLO, WORLD")

Why "HIWORLD", not "HELLO WORLD"?

In the example code, you can see that the Hello World program is named HIWORLD rather than HELLO WORLD, which seems a bit confusing. There's a good reason: calculator programs can have only uppercase names of at most eight characters, containing letters and numbers (but no spaces). Every program name must also start with a letter. Therefore, a name like HELLO or HELLOWOR or HIWORLD is allowed, but 1HELLO and HELLOWORLD and HELLO WORLD are all invalid.

The program shown consists of two pieces: the name of the program (the first line) and the source code for the program (in this case, the second line). Every command has a one-word name and takes zero, one, or more arguments. The command here is `Disp`, short for *display*, and instructs the calculator to display a line of text on the screen. I give it one argument here, the text to be displayed: `"HELLO, WORLD."` In programming parlance, a piece of text to be used or displayed is called a *string*. This line displays the string "HELLO, WORLD" on the screen. Notice that there's no explicit instruction telling the calculator to stop executing the program. Instead, whenever the interpreter reaches the end of a program, it takes that as an implicit command to end the program.

TYPING THE PROGRAM ON YOUR CALCULATOR

If you'd like to type this program into your calculator to try it, you'll first need to create a program named HIWORLD. Start at the homescreen of the calculator, the area where the cursor flashes, and where you can type math and perform the following steps:

1. Press [PRGM] to get to the Program menu, where you'll spend much of your time as you learn to program your calculator.
2. Press [▶][▶] (the right arrow key twice) to switch to the NEW tab, and press [ENTER].
3. The calculator will ask you for a name for your new program; you can type HIWORLD with the keys [^][x^2][–][7][×][)][x^{-1}], the keys over which the letters H, I, W, O, R, L, D are written in green.
4. Press [ENTER] again to create a blank program with the name HIWORLD, as shown in figure A.2.

Figure A.2 Creating a program named HIWORLD

You'll then be able to type lines into your program.

> **Typing out tokens: Disp vs. "D" "i" "s" "p"**
>
> As I will remind you several times in your early experiences with TI-BASIC, commands are something called *tokens*, which means that the `"Disp "` command is a single entity, not the series of characters "D," "I," "s," "p," and a following space. One important side effect of this is that you can't type out DISP as letters and expect it to work; the calculator won't understand what you're trying to do. You must use the tokens found in each of the menus.

To type the one line of code in this particular program after you created the new, blank program, continue to follow these steps:

5 Press [PRGM] from the program editor, which brings up a menu full of programming commands that you can use. It has three tabs, labeled CTL (Control), I/O (Input/Output), and EXEC (Execute program). You can press the left- and right-arrow keys to switch which of the three tabs is visible and the up and down arrows to scroll through each menu. You first need `Disp`, which is the third item in I/O. Press [▶] to go to the I/O tab; then press [▼][▼][ENTER] or just [3] to select 3:Disp. In every menu, you can either move the highlight over the number of the item you want and press [Enter] or press the number itself on the keypad, in this case [3]. This will paste the `Disp` command into your program.

6 After you have the `Disp` in the program editor, you need to type the string you want it to display. To type "HELLO, WORLD", you'll first need the quotation mark, [ALPHA][+]. HELLO is [ALPHA][^], [ALPHA][SIN], [ALPHA][)], [ALPHA][)], [ALPHA][7]. Notice that unlike a computer keyboard, you don't hold down [ALPHA] and tap the key from which you want a letter; instead, you press and release [ALPHA] and then press and release the other key. The space character is [ALPHA][0]; see if you can find the letters for "WORLD" on your own, and don't forget the ending quotation mark. When you've finished, your program should look like figure A.3.

Figure A.3 The source of HIWORLD, on a calculator

> **Alpha and second functions of keys**
>
> To type out the letters in the string, you need to know that each key on the calculator has three functions, with a few exceptions. The normal function of each key is written on the key itself, such as [9] or [SIN] or [GRAPH]. Most also have a second function, accessed by pressing the [2nd] key followed by the key in question. Most have an alpha function too, found by pressing [ALPHA] and then the key. The second and alpha functions are labeled above each key and color-coded to match the modifier ([2nd] or [ALPHA]) key.

When you've finished, press [2nd][MODE] to quit the program editor and go back to the homescreen.

With the source code of the Hello World program entered into your calculator, you can now run it.

A.1.3 Running the Hello World program

Your experience with any programming language including TI-BASIC will be cycles of coding, testing, and fixing bugs and errors. If you're still in the program editor, press

I made a typo! Now what?

If you make any errors while typing, press the [DEL] key, which will erase whatever is currently under the cursor (not before it, like Backspace on a computer keyboard). The normal mode of the program editor is Replace, which means if you move the cursor over a character or token and press something else, it will be replaced. If you instead want to insert at the current cursor position, press [2nd][DEL] to go into Insert mode.

[2nd][MODE] to quit back to the calculator's homescreen so that you can run this program to test it. Press [PRGM] to open the Program menu, this time to the list of programs you have on your calculator. Notice that the function of the [PRGM] key depends on context. From the homescreen, it brings up a list of programs, whereas from the program editor, it shows a list of commands. A few of the other keys on the calculator have similar context-dependent functions.

Use the arrow keys to find HIWORLD in the list of programs, and press [ENTER]. This pastes `prgmHIWORLD` to the homescreen, a command to the calculator to run the program named HIWORLD. Press [ENTER] again to execute the command and run the program. You should see something like the screenshot back in figure A.1, reproduced in figure A.4.

Figure A.4 The output of the Hello World program again, as in figure A.1

When you run this or any other TI-BASIC program, the interpreter starts at the beginning of the program and executes each line sequentially unless instructed otherwise. For this program, it first sees the `Disp` command and knows the program wants to display something. You can display matrices, lists, numbers, or strings; the quotation mark immediately after `Disp` tells the calculator that you want to display a string, so it searches for the second, concluding quotation mark. Once the interpreter finds that second quotation mark, it knows what the string to be displayed is (HELLO, WORLD) and puts the string on the screen. It then goes to the next line of the program, but because there's no next line, the program ends. When a program ends and returns to the homescreen, the calculator almost always prints "Done," shown in the screenshot in figure A.4.

Lessons of the Hello World program

The Hello World program is close to the simplest TI-BASIC program you can write, but it's a useful stepping-stone for becoming more familiar with entering and running programs, as well as getting a first glance at what happens when a program is run. You have now been introduced to the difference between compiled and interpreted programs.

How about a program that might be useful to you in math class?

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

SOLUTION WITH REAL ROOTS
a = 1, b = 2, c = 1

$$x = \frac{-2 \pm \sqrt{2^2 - 4*1*1}}{2*1} = \frac{-2 \pm \sqrt{0}}{2}$$

$$x = -1$$

SOLUTION WITH IMAGINARY ROOTS
a = 1, b = 4, c = 5

$$x = \frac{-4 \pm \sqrt{4^2 - 4*1*5}}{2*1} = \frac{-4 \pm \sqrt{-4}}{2}$$

$$x = -2 \pm 1i$$

Figure A.5 The Quadratic Formula for finding the roots of $0 = ax^2 + bx + c$ (top). The samples show a = 1, b = 2, c = 1 (left), which yields a single real root, and a = 1, b = 4, c = 5 (right), which yields distinct but imaginary roots.

A.2 Math programming: a quadratic solver

A Quadratic Formula solver is a great first math program and is often the first math application that budding calculator programmers teach themselves. The Formula is universally taught in algebra or geometry classes when many students first receive their graphing calculators, and the program itself is short and simple. Most important, when you finish typing what in our case is a nine-line program, you have a tool that you can use.

If you're unfamiliar with the Quadratic Formula, it's a method to find the two roots of an equation in the form $ax^2 + bx + c = 0$, or the values of x that make the equation true given values for a, b, and c. Letters a, b, and c represent the three parameters to the quadratic equation in math notation, corresponding to variables A, B, and C in TI-BASIC. The Quadratic Formula is written as shown in figure A.5, along with two sample sets of a, b, and c values.

To solve the Formula, a simple program should ask the user for values for a, b, and c, store them in A, B, and C, then plug them into the equation in figure A.5, and print the values found for x to the user. Unfortunately, there are complications. Suppose that a = 1, b = 4, and c = 5. When you do the math, you'll need to take the square root of $4^2 - 4(1)(5) = 16 - 20 = -4$. As you may know, taking the square root of a negative number yields an imaginary result. Therefore, if $4ac > b^2$, then both roots are imaginary, because $b^2 - 4ac$ will be negative and the square root of a negative number is imaginary. If $b^2 = 4ac$, then the equation has a double root, and if $b^2 > 4ac$, making $b^2 - 4ac$ greater than zero, the quadratic equation will have two distinct (different) real roots. A competent Quadratic Formula solver would detect double roots and imaginary roots and adjust accordingly. For your first math program, you'll write a simple solver that doesn't try to determine imaginary roots (but does warn you about the imaginary roots) and doesn't check for double roots.

A.2.1 Building the Quadratic Formula solver

The TI-BASIC code for the simple Quadratic Formula solver is presented here:

```
PROGRAM:QUAD
:Prompt A,B,C
:If 4AC>B²
```

```
:Then
:Disp "IMAGINARY ROOTS"
:Else
:Disp (⁻B+√(B²-4AC))/(2A)
:Disp (⁻B-√(B²-4AC))/(2A)
:End
:Return
```

Let's work through this program line by line so you can understand how it works. When the calculator executes a program, it starts at the beginning and works its way down line by line, so you're reading the program just as the calculator does. I'll tell you the key sequences used to type each line of code, so that you can test this program on your own calculator. You can also download the source code for all the programs in this lesson plan from the website, www.manning.com/ProgrammingtheTI-83Plus/TI-84Plus.

```
PROGRAM:QUAD
```

This first line isn't a piece of the source code; it's the name of the program. The TI-BASIC editor displays the name at the top of every program, so I'll adopt the same convention. For most programs it doesn't matter what you call the program, as long as it's at most eight uppercase letters and numbers and starts with a letter. For obvious reasons, this program will be called QUAD. If you'd like to type this program into your calculator to try it, you should start by creating a program named QUAD. From the homescreen, press [PRGM], then [▶][▶] to get to NEW, and press [ENTER]. You can type QUAD with [9][5][MATH][X^{-1}], and then press [ENTER] again. I won't walk you through typing this program key by key, but I'll give you plenty of hints to help you type it yourself.

```
[Line 1] :Prompt A,B,C
```

The first line of code of this sample program is a `Prompt` command. This instructs the calculator to prompt, or ask, the user for three variables, named A, B, and C. As mentioned with the Hello World program, commands, sometimes also called functions, consist of a name and take one or more arguments. These arguments can be inside parentheses or, as with `Prompt`, placed after the command. This particular `Prompt` has three arguments, `A`, `B`, and `C`. Users will be asked to enter a number at each of the three prompts that will appear, one each for `A`, `B`, and `C`. If users type anything that isn't a number, the calculator will produce an error message. After this command runs, the three values the user entered will be stored in three variables, or memory locations, labeled A, B, and C. This will allow you to use these values later in your program by referring to the variable names A, B, and C. You can type this line with [PRGM][▶][2] [ALPHA][MATH][,] [ALPHA][APPS][,] [ALPHA][PRGM][,] [ENTER].

```
[Line 2] :If 4AC>B²
```

The second line of the program is a conditional statement, indicated by the `If` at the beginning of the line. Every `If` is followed by a statement that must evaluate to either

logical true or false. Obviously, true indicates that the conditional statement that follows is correct, and false indicates that it's incorrect. If 4AC is greater than B^2, then the statement $4AC > B^2$ is true. Otherwise, it's false. Conditional statements dictate which pieces of the program, or code, are executed. If this case, the section between `Then` and `Else` is run only if the condition evaluates to true, whereas the section from the `Else` to the `End` is run only if the condition is false. From here onward, I'll assume that you understand that [ALPHA][MATH] types the A character and that [ALPHA][anything] types the letter printed at the upper right of the key. I'll therefore represent [ALPHA][MATH] as ["A"] from now on. You can type this line using [PRGM][1][4]["A"]["C"][2nd][MATH][3]["B"][x^2].

```
[Line 3] :Then
```

`Then`, therefore, marks the beginning of the code to run if the conditional is true, or if the roots will be imaginary because $B^2 - 4AC$ will be less than zero. `Then` can be found under [PRGM][2].

```
[Line 4] :Disp "IMAGINARY ROOTS"
```

The fourth line draws or prints a string (a sequence of text characters) onto the screen. In this case, the `Disp` command writes text onto the homescreen. The string to be displayed is offset by quotation marks, so that the calculator knows where the string starts and ends and is `"IMAGINARY ROOTS"`. `Disp` can be typed by pressing [PRGM][▶][3], and the quotation mark is [ALPHA][+]. For this line, it's worth noting that [2nd][ALPHA] will "lock" the calculator in ALPHA mode, letting you type letters without needing to prefix each with [ALPHA], until you press [ALPHA] again.

```
[Line 5] :Else
```

`Else` on the fifth line marks the boundary between the true and false sections of code for the conditional on line 2. If, when executing the program, the calculator finds the condition on line 2 to be true, it will skip directly from the `Else` to the `End` and continue executing downward. If the condition is false, it will skip directly from `Then` to `Else` and execute the code in between the `Else` and the `End` instead of skipping it. You can find `Else` in [PRGM][3].

```
[Line 6] :Disp (⁻B+√(B²-4AC))/(2A)
[Line 7] :Disp (⁻B-√(B²-4AC))/(2A)
```

Lines 6 and 7 perform the Quadratic Formula, calculating the two possible roots. The calculator needs two separate equations because it doesn't know what ± is; you must explicitly tell it to calculate the + case and the – case. An important lesson from typing these two lines, something that trips up many beginner programmers, is that the negative sign, such as -B, isn't the same as the subtraction sign, as in 2–1. The negative sign is written here as a superscript to distinguish it, and is also sometimes shown as (-) in tutorials. It's typed with the [(-)] key, between [.] and [ENTER]. The subtraction sign is typed with the subtraction key [–]. The radical or square root symbol is [2nd][x^2];

notice that just as [ALPHA][key] types the item shown at right above the key, [2nd][key] types the item shown at left above the key.

```
[Line 8] :End
[Line 9] :Return
```

The final two lines end the conditional statement and then end the program. The `End` marks the end of the conditional that started with the `If` on line 2 and continued with the `Then` and `Else`. The `Return` tells the calculator that the current program has completed and that it can return you to the homescreen (or, as you'll later learn, to another program that called this program). `End` is under [PRGM][7], and `Return` is near the bottom of [PRGM], at item E. To save time, you can press [PRGM] and then [▲] until you reach it.

If you are following along and typing in this program, you can now press [2nd][MODE] to quit to the homescreen. Press [PRGM] and scroll down to QUAD and press [ENTER] twice, once to paste `prgmQUAD` to the homescreen, which is an instruction to the calculator to run QUAD, and the second time to start running it. If everything went well, you should see the prompt `A=?` on the screen. If you have problems, double-check that your program exactly matches the code shown.

In case you're having difficulty entering the QUAD program on your calculator or it's not working properly, figure A.6 shows the source code as it should look typed into a calculator.

Next, we'll look at testing the program.

A.2.2 Testing the solver

When you test the program, it will prompt you for three separate numbers, the values for the variables A, B, and C that it will plug into the Quadratic Formula. At each prompt, you can use the number, decimal point, and negative sign keys to type a value; then press [ENTER]. After you provide a value for C, the program will either display the two roots of the equation $ax^2 + bx + c$ or tell you that the roots are imaginary.

You can see the results of testing the program for two samples sets of A, B, and C values in figure A.7.

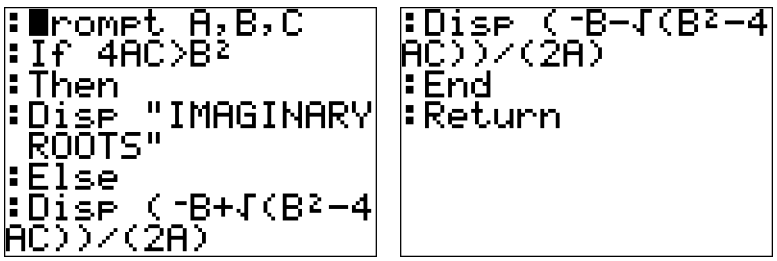


Figure A.6 Source code for `prgmQUAD` typed into a calculator

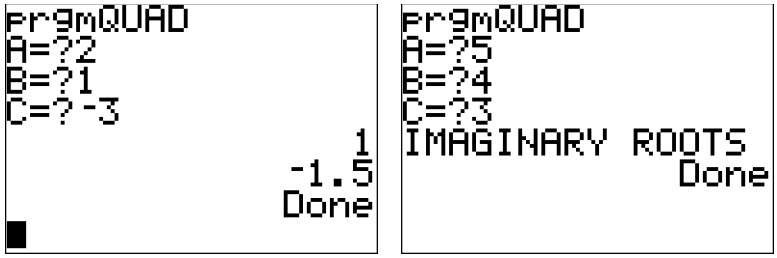


Figure A.7 Two tests of the quadratic solver for different values of A, B, and C. The left image shows the roots of the equation $2X^2 + X - 3 = 0$, which are 1 and –1.5. The right image shows the roots of the equation $5X^2 + 4X + 3 = 0$, which are imaginary roots.

Graphing calculator programming without games would be no fun. I'll present a variety of increasingly complex games that you can write as your programming skills progress, but to get you started, let's jump right into a simple guessing game.

A.3 Game programming: a guessing game

Although graphing calculators are math devices, and they can be used to write complex and powerful math programs, game programming is a perennially popular reason why students and other users start exploring calculator programming. Among the vast range and variety of games that can be written are everything from simple text-based RPGs to complex arcade and 3D games. Here, we'll start with a simple example, a number-guessing game in which the calculator picks a random number and then asks users to guess numbers until they find the correct one. To make it more fun, the program tells users whether to guess higher or lower and challenges them to try to find the number in the fewest possible guesses. This program is nonlinear: execution doesn't simply flow from the top of the program to the bottom and then stop. First, you'll see the source code for the program, and I'll explain it piece by piece. You'll then have an opportunity to type the game into your own calculator and play it.

As in the previous example, if you'd like to try this program on your own calculator, you should start by creating a program, perhaps called GUESS. I'll provide slightly less key-by-key detail on how to type in this particular program, hopefully giving you a chance to exercise the knowledge you gained from the Hello World and quadratic solver programs.

A.3.1 Guessing game source and function

The source code of the guessing game is 13 lines of code and fairly straightforward. It introduces two commands that haven't been presented previously, namely `randInt` and `Repeat`; it also is the first of our three examples to demonstrate the store (→) operator. Glance over the following code and try to get a general idea of how it works. Although the commands are probably new to you, their names should give you a vague intuition into their function.

```
PROGRAM:GUESS
:randInt(1,50)→N
```

Generate the number the player will guess

```
:0→M
:Repeat G=N
:Prompt G
:If G>N
:Disp "TOO HIGH"
:If G<N
:Disp "TOO LOW"
:M+1→M
:End
:Disp "CORRECT AFTER:"
:Disp M
:Disp "GUESSES"
```

Repeat the loop from here to the End until G = N

Ask the player to type a guess

Increment the number of guesses used

UNDERSTANDING THE PROGRAM

To get a basic idea of the flow of this program, look at figure 1.10. Although you might not be familiar with reading flowcharts, I'll try to explain how this shows what the program does by putting you in the shoes of the calculator (specifically, its interpreter). As the calculator, you start at the top of the program at the box labeled START in figure A.8 and keep executing the lines in order unless told otherwise. Let's follow the figure: First, you pick a number at random, the number the player will be trying to guess. Next, you "Ask player for guess." There, you wait for the player to input a number, their guess for the secret number. When the player enters a number, you can then take one of three paths. If the player guessed correctly, take the path labeled "Correct" and display how many guesses it took the player to get the number; then stop. If the number is wrong, then display either "TOO HIGH" or "TOO LOW," and ask the player for another guess.

The arrows that make you, the calculator, return to the same point in the program over and over are part of what is called a loop. Notice that the arrows from "Ask player for guess" to "TOO LOW" and "TOO HIGH" then lead back to "Ask player for guess" and that if the player keeps making guesses that are too high or too low, the program will continue to cycle back to the oval in figure A.8. In the code, the `End` command returns to the `Repeat` command, closing the loop that starts with the section of code between the `Repeat` and the `End`. This program uses the loop to make the program

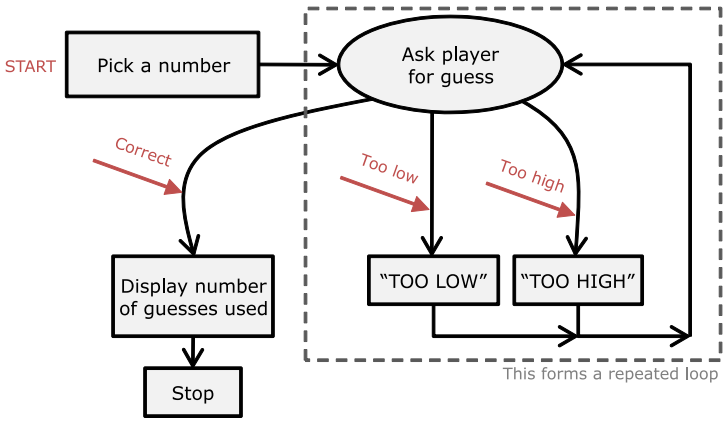


Figure A.8 Guessing-game program flow. The program will keep displaying "TOO HIGH" or "TOO LOW" and returning to ask the player for another guess (a loop) until the player guesses correctly.

keep running until the player's guess is correct, so that the player will be able to keep guessing. Keeping the flowchart in figure A.8 in mind, we'll now take a more methodical look through the source code of the program.

GUESSING GAME SOURCE CODE: A WALKTHROUGH

As I take you through the code for this program line by line, you may wish to type it into your own calculator so that you can test it. For each line, I'll provide extra tips about where to find commands or symbols that you haven't seen before.

The first line of the program contains the `randInt` command, which takes two arguments. In this case, the two arguments are the minimum and maximum possible number that the function should generate.

```
:randInt(1,50)→N
```

As the name suggests, the `randInt` command generates a random integer (whole number) between and including 1 and 50. Figure A.9 demonstrates that this random number is the number the player will guess.

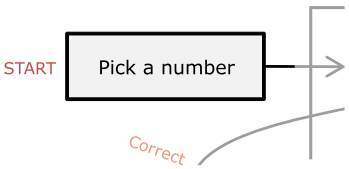


Figure A.9 The `randInt` command serves to pick a number to be guessed.

The small arrow indicates that the random number it returns should be stored into variable N, just as `Prompt N` would ask the user for a number and store it into N. In this case, the calculator generates the number that's stored into the variable, rather than asking the user to type it in. You can find `randInt` under [MATH]; then use the right- (or left-) arrow key to get to the PRB submenu, and choose `5: randInt(`. The closing parenthesis needed to complete the command is the key above the [9] key. You may also not know that the store operator (→), used to update the contents of variables, is on the lower left of the keypad above the [ON] key, labeled the [STO>] key.

```
:0→M
```

Similarly, the second line stores a value to M, but this is simply a zero. The program will be using M to store the number of guesses that the user has made and N to store the target number that the user is trying to guess.

> **Choosing variables to use**
>
> It doesn't matter what variables a program uses to store numbers; in this example, the program could easily use A and B or S and T or C and Z instead of M and N. You have A–Z and many other variables available to you, and it's up to you as a programmer to choose variables that make sense to you. The only exception is the variable Y, which many programmers avoid because certain programming features change the value in Y as a side-effect.

```
:Repeat G=N
:Prompt G
:If G>N
```

Repeat the loop from here to the `End` until G = N

```
:Disp "TOO HIGH"
:If G<N
:Disp "TOO LOW"
:M+1→M
:End
```

The majority of the program is between the `Repeat` and `End` commands, which together enclose what is called a *repeat loop*. In a repeat loop, execution will repeatedly restart at the `Repeat` command every time the `End` command is reached, allowing the code inside the loop to be run over and over again. The program eventually needs a way to get out of the loop; otherwise it'll be looping forever. The solution is a condition on the loop, just as `If` takes a condition. Notice the G = N condition on the `Repeat`; stated more verbosely, this `Repeat` command says "Repeat the loop from here to the `End` until G = N." That means that as long as G isn't equal to N, the loop will continue, but when G becomes N, which means that the user finally guessed the number correctly, the loop will end. The greater than, less than, and equals symbols are all in [2nd][MATH], the Test menu. `Repeat` is [PRGM][6].

```
:Prompt G
```

You have previously seen the `Prompt` command, which will ask the user to enter a value for G (their guess) and then store it into variable G.

```
:If G>N
:Disp "TOO HIGH"
:If G<N
:Disp "TOO LOW"
```

The next four lines will display a message to the user based on the guess that the user entered. If the value for G is larger than the target number (N), the program will display "TOO HIGH." If it's smaller than the target number, the program will display "TOO LOW." Figure A.10 shows the parts of the diagram representing `Prompt` and these two conditional constructs.

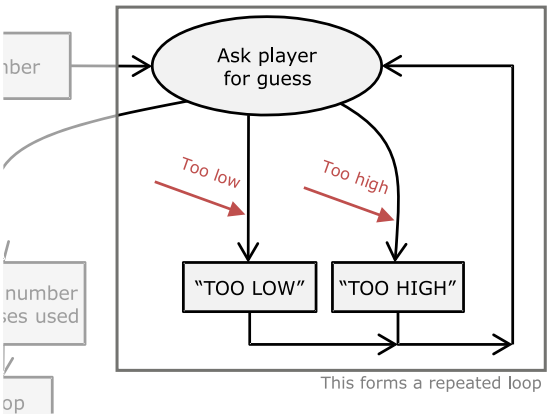


Figure A.10 The loop that alternates between asking players for guesses and telling them whether the guesses are too high or too low

Notice that if the user guessed the correct number, then both G > N and G < N are false, and nothing will be displayed.

```
:M+1→M
```

Execution will continue downward, taking the value in M, adding 1 to it, and storing this value back into M. As we discussed, M contains the number of guesses the user has made so far and is incremented each time the `Repeat` loop is run because the loop runs once per guess.

```
:End
```

The next command read and executed is `End`. `End` triggers the interpreter to reexamine the `Repeat` condition to decide whether to go back and start executing at the `Repeat` again, prompting for another guess, or continue directly to the code under `End`, finishing the looping process. If the condition is true, that is, G = N, then the code after `End` will be executed. If the condition is false, or G isn't equal to N (written as G ≠ N), then the user has still not guessed the correct number, and execution will return to the `Repeat`, `Prompt` section of the code.

```
:Disp "CORRECT AFTER:"
:Disp M
:Disp "GUESSES"
```

The last three lines of the program tell the user that they guessed correctly and also display the number of guesses that the user made before finding the correct answer, as shown in figure A.11.

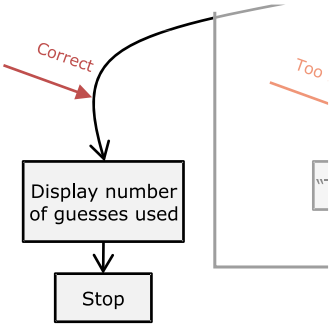


Figure A.11 Displaying the notification that the player won and how many guesses they needed

Notice that although you have only seen `Disp` used with strings so far, `Disp` can also be used to check what number is currently stored inside a variable, in this case M. As with the Hello World program, you don't need to explicitly add the `Return` at the end of the program to tell the calculator that the program has reached its end; instead, the end of execution is signaled by the end of the program file.

With a full understanding of how the program works, you should now try out the program on your own calculator to see it in action.

Typing and testing the program

If you haven't done so already and would like to type this program into your calculator and try it out, you already know where to find many of the commands, such as `Disp`, `End`, `Prompt`, and `If`. Throughout the discussion of the source code, I provided tips about where to find the commands you saw for the first time in this program. As with the two previous examples, you can run `prgmGUESS` by quitting to the homescreen, finding GUESS in the Program menu, and pressing

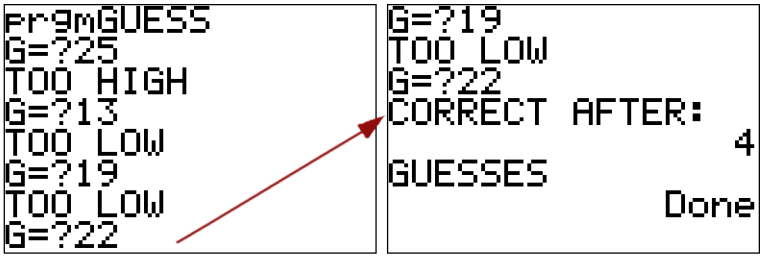


Figure A.12 Playing the guessing game to the correct answer of 22 in four guesses

[ENTER] twice. Figure A.12 shows an example of a game where the player got the correct number in four guesses.

As I discussed in walking through the program's code, the program will pick a random number and repeatedly wait for the user to guess a number and tell them if it's higher or lower than the target number. Once the user guesses the correct number, the program will display the number of guesses made and exit.

A.3.2 Lessons of the guessing game

One of the most important lessons of this particular example is that programs can have arbitrarily complex flows of execution depending on the user's or player's input. Put more simply, what happens inside the program and which pieces of the program are executed in which order are often based on user input. If the user guesses the correct number right away, the program will only go through the `Repeat/End` loop once and will never loop back to the `Repeat`. If the user guesses the same incorrect guess over and over, the program will keep looping over and over. Depending on whether each guess a user makes is higher or lower than the target value, one of two possible `Disp` commands is executed. Indeed, complex games and utilities can be built up from simple pieces such as loops and conditionals combined with concepts you'll learn later such as subprograms and jumps.

A.4 Summary

Graphing calculator programs can be as complex or as simple as you want and are limited only by your imagination and problem-solving skills.

Sample lesson plan B
Understanding event loops

In this lesson plan, you will build a full interactive Mouse and Cheese game with `getKey` and then look at ways to expand the game. Along the way, you will strengthen the knowledge of an important programming concept: event loops, their purpose and construction, and how you can use them to create programs that can respond to events such as keypresses.

B.1 The Mouse and Cheese game

In this game, the player controls a mouse, represented by the trusty capital M. The player will use the arrow keys to move the M around the screen, chasing after pieces of cheese, represented by a small square. To make the game competitive, the mouse's hunger will increase as time passes, regardless of whether or not keys are pressed, which is where your event-loop knowledge will become necessary. To display the hunger, the program will draw a bar at the right edge of the screen made of equals (=) characters. When the hunger bar fills all the way up, the game will end and will tell the player how many pieces of cheese they were able to eat before the game ended. You can see a game in progress and the end of a game in figure B.1.

The program will be structured with two nested `Repeat` loops; the following pseudocode outlines the structure of the program that you'll create:

```
:Repeat until hunger bar fills or [CLEAR] pressed
:[initialize cheese and hunger bar]
:Repeat until hunger bar fills or [CLEAR] pressed or mouse reaches cheese
:[move mouse and increase hunger]
:End
:[handle eating cheese]
:End
```

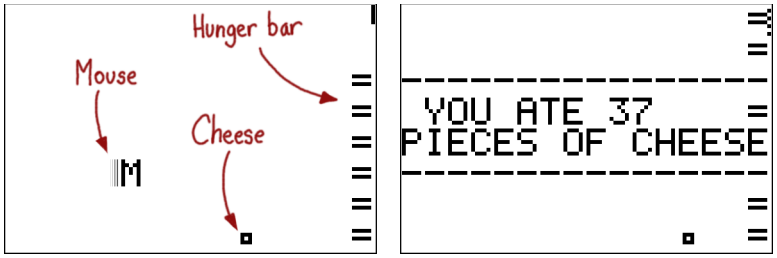


Figure B.1 Screenshots of the Mouse and Cheese game. The mouse (M) is moved by the player to chase after the pieces of cheese (the square). The player loses if the hunger bar (right edge of the screen) fills up, at which point the game tells the player how many pieces of cheese the mouse ate (right).

At the beginning, the mouse's hunger and coordinates will be initialized, and a starting score of zero will be set. At the end, the final score will be displayed to the player. The outer loop will run once per piece of cheese, so it will first create a piece of cheese at random coordinates and display it, then draw the hunger bar, and then start the inner loop. At the end of the outer loop, the player's score will be increased and the mouse's hunger removed if it reaches the piece of cheese. The outer loop ends if the hunger bar is full or the user presses [CLEAR]. The inner `Repeat` loop runs the heart of the game. On every iteration, it checks for keys and moves the mouse accordingly, increases the mouse's hunger, and, if necessary, draws another = sign on top of the hunger bar.

With this basic skeleton in mind as a guide, you can move on to looking at the full TI-BASIC source code for the game.

B.1.1 Writing and running the game

You know where to find `getKey` now, and all of the other commands in the code in listing B.1 should be familiar to you as well. The square symbol on the seventh line of the code might be unfamiliar; you can find it under the MARK menu, [2nd][Y=][▶][▶]. You should type this game into your calculator so that you can try it out and better understand the discussion of how the code works. You'll also want to use it as a base to experiment with tweaks and changes. As with all other programs you've worked with so far, run the program from the homescreen (or your favorite shell). You can press the arrow keys to move the mouse around and press [CLEAR] to end a game early.

Listing B.1 The Mouse and Cheese game

```
PROGRAM:CHEESE
:7→A:4→B
:0→H:0→S
:Repeat H≥8 or K=45
:randInt(1,15→C
:randInt(1,8→D
:ClrHome
```

X and Y coordinates of the mouse

Hunger (H) ranges from 0 to 8; score (S) increases for each piece of cheese eaten

Initialize coordinates of piece of cheese to a random spot on the screen

Outer loop: one iteration per piece of cheese, ends when game ends

```
:Output(D,C,"□
:For(X,1,int(H
:Output(8-X,16,"=
:End
:Repeat H≥8 or K=45 or (C=A and B=D
:If H=int(H
:Output(8-H,16,"=
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]
:If K=24 and A>1
:A-1→A
:If K=26 and A<15
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
:H+.1→H
:End
:If C=A and B=D
:Then
:0→H
:S+1→S
:End:End
:Output(3,1,"----------------
:Output(4,1," YOU ATE
:Output(4,10,S
:Output(5,1,"PIECES OF CHEESE
:Output(6,1,"----------------
:Pause
:ClrHome
```

Only display the cheese once (this symbol is under the MARK tab of [2nd][Y=])

Draw the current hunger bar

Inner loop: runs repeatedly until hunger bar fills, [CLEAR] is pressed, or mouse reaches cheese

Increase hunger on every loop, whether or not a key is pressed

If mouse is at same coordinates as cheese, increase score and remove hunger

End outer loop and finish the game by displaying score

B.1.2 Understanding the game

The Mouse and Cheese game draws on many of the skills that I've discussed thus far, including conditionals, nested `Repeat` loops, using `getKey` in asynchronous event loops, using `Output` to display strings, characters, and numbers, and dealing with integer and decimal numbers. I'll start a detailed look at the code with the initialization at the beginning of the program, work my way through the outer loop and the inner loop, and conclude with the game-ending score display.

The Mouse and Cheese game uses six major variables. The player's mouse is at some (X,Y) coordinates, which are stored in the variable pair (A,B), and each piece of cheese is at coordinates (C,D). Because only one piece of cheese is displayed at a time, it doesn't need to keep track of multiple pieces of cheese. It does need to keep track of the mouse's hunger, which it puts in H, and the player's score, in S.

```
:7→A:4→B
:0→H:0→S

:randInt(1,15→C
:randInt(1,8→D
```

Because the homescreen is 8 characters tall, I decided to make H = 0 mean no hunger, and H = 8 mean full hunger. The score S will increase by 1 with each piece of cheese eaten, but you could easily change it to some other value. You could even give the player more points for getting the cheese faster, or some other more complicated scoring scheme.

Initialization and program structure

You might start writing this game by figuring out exactly how the screen will be laid out. Because the hunger bar will be the last column of the homescreen, the 16th column, the mouse and cheese can both be anywhere in the 15-column by 8-row area starting at the left edge of the screen. The mouse's position can be initialized near the center of that area, at (X,Y) = (A,B) = (7,4). The hunger H should start at 0, and the score S will also be initialized to 0. Next come the two nested `Repeat` loops that form the body of the game.

One excellent question would be why the program uses two nested loops instead of one giant loop. A single giant loop could indeed be used instead, in which you'd have a conditional that would generate new coordinates for the cheese (creating a new piece) every time the mouse reached the cheese, but this would be slow, because the program would have to jump over that piece of code every time through the `getKey` loop, regardless of whether or not the mouse reached the cheese. The left side of figure B.2 shows this incorrect structure, where all of the main game code including creating new pieces of cheese is inside the event loop. The right side of the diagram shows the proper structure, where only updating the hunger bar and updating the

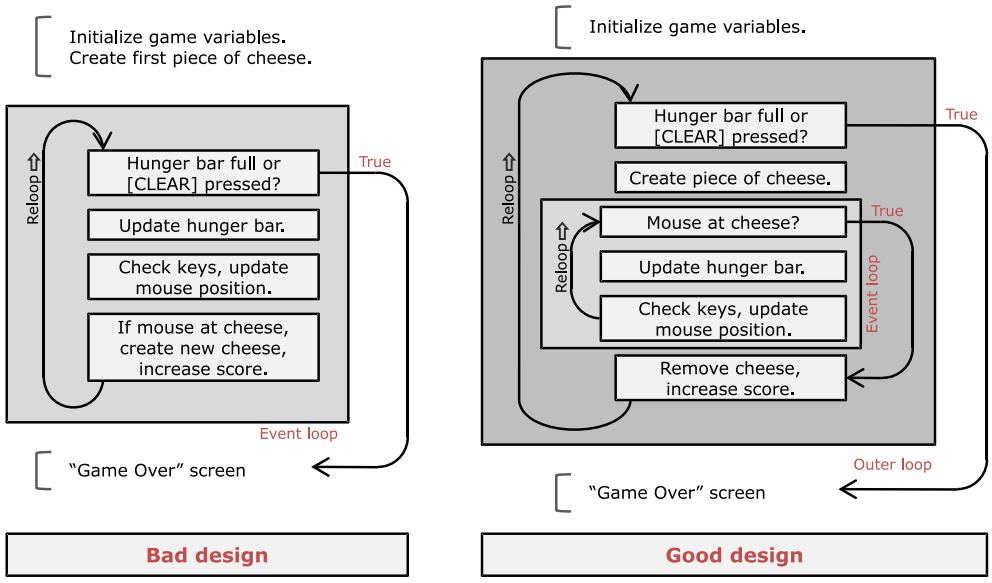


Figure B.2 Putting the full game loop including cheese management inside the event loop (left) and properly reducing the amount of work in the inner event loop (right)

mouse's position are inside the event loop. Creating and destroying the cheese are part of the outer loop.

The outer loop starts by creating the cheese and ends by handling the mouse reaching the cheese; the inner loop only handles moving the mouse around and adjusting the hunger bar. The inner loop terminates when the mouse reaches the cheese to let the outer loop handle the mouse eating the cheese. This is shown in the following pseudocode:

```
:Repeat H≥8 or K=45
:[initialize cheese and hunger bar]
:Repeat H≥8 or K=45 or (C=A and B=D
:[move mouse and increase hunger]
:End
:[handle eating cheese]
:End
```

Look at the conditionals on each of the loops first. `Repeat [condition]` is like telling the program "repeatedly run this loop until [condition] is true". For the outer loop, the program keeps creating pieces of cheese and letting the mouse chase them until the hunger bar is full or the user presses [CLEAR]. Because the hunger bar will get full or the [CLEAR] key will be pressed when the program is inside the inner loop, both the inner and outer loops need to have these two conditions. Therefore, when the hunger bar fills or [CLEAR] is pressed, the inner loop will end, the end of the outer loop will run, and then the outer loop will end too.

THE OUTER LOOP: CHEESE MAINTENANCE

Because one new piece of cheese should be created every time the outer loop starts, the program generates the cheese at random coordinates between the start of the outer loop and the beginning of the inner loop. The `randInt` command can be used to do this:

```
:randInt(1,15→C
:randInt(1,8→D
```

Notice that because the mouse and cheese can only be between X = 1 and X = 15, and Y = 1 and Y = 8, you use these values as the limits for the `randInt` command. After the program generates a new piece of cheese, it should clear the screen, draw the cheese, and draw the hunger bar. Why do you clear the screen instead of just erasing the old piece of cheese using the `Output(D,C,"[one space]` trick? Because when the mouse eats a piece of cheese, the hunger bar becomes empty, and it might have been nearly full. It's much easier in this case to clear the screen, then draw the cheese and empty hunger bar. The space trick would work fine in this case; it would be more complicated. In many of your programs, just as in the eight-direction program, you'll run into several possible ways to do something. In many cases, one of the options will be fast, one will use a small amount of code, and the smallest version may not necessarily be the fastest.

Clearing the screen is easy enough with `ClrHome`, but how shall the hunger bar be drawn? Because I decided hunger can go from 0 through 8, the program could loop X

from 0 to H, drawing an equals sign at (X,Y) = (15,X + 1), which would make the hunger bar grow from the top of the screen downward as the mouse's hunger increases (here, X is being used as a throwaway looping variable; the program doesn't use it after the hunger-bar-drawing loop ends). The program needs to display at row X + 1 instead of row X because 0 isn't a valid Y coordinate on the homescreen: the first row is 1, not 0. But to be extra fancy, we'll make the hunger bar grow from the bottom of the screen to the top. The coordinates can be flipped over from the top to the bottom of the screen by subtracting them from 8. Therefore, 8 – X will be 8 when X = 0, then 7 when X = 1, and so on, so it will grow upward from the last row. Because the hunger always gets reset to H = 0 when the mouse eats a piece of cheese, making the hunger bar nearly empty, the program doesn't need a loop here to redraw the hunger. It could instead have a single `Output(8,16,"=` command. In case you decide to modify the game, to only decrease the mouse's hunger rather than removing it completely when the mouse gets a piece of cheese, this loop will be able to correctly handle drawing any hunger level.

THE EVENT LOOP: HUNGER AND THE SCURRY OF THE MOUSE

Continuing through the program, next comes the inner loop, which is the event loop of the game. This loop will continue to execute, checking the keyboard for activity and moving the mouse accordingly, while simultaneously updating the mouse's hunger and filling the hunger bar displayed onscreen. The hunger bar update is the "other code" section of the event loop, whereas the keypresses that cause the mouse to move are the events that it acts upon. For the sake of analysis, take a look at the body of the inner loop by itself:

```
:Repeat H≥8 or K=45 or (C=A and B=D
:If H=int(H
:Output(8-H,16,"=
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]
:If K=24 and A>1
:A-1→A
:If K=26 and A<15
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
:H+.1→H
:End
```

Add another notch to hunger bar if hunger has increased enough

Only erase mouse if it might move

Repeat until hunger bar is full, [CLEAR] is pressed, or mouse reaches cheese

As you can see, the event loop will terminate on any of three conditions. First, if H is at least 8, the mouse's hunger bar is full, and the game is over. Because the outer and inner loop share this condition, the outer loop will also end if this is true. The second condition is K = 45, which means, as shown in figure B.3 that the [CLEAR] key has been pressed.

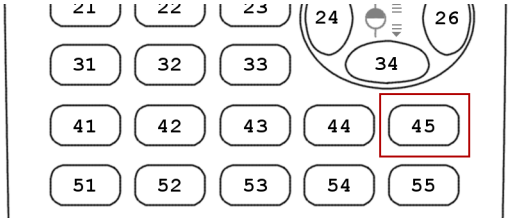


Figure B.3 An excerpt of the keycode chart showing that the [CLEAR] key is code 45

This will end the outer loop as well once the program reaches the outer loop's `End`, because the outer loop has K = 45 as one of its `or`ed termination conditions. The third condition is unique to the inner loop, namely `C=A and B=D`. This expression gets put inside parentheses so that it's evaluated as a whole, which means that the event loop will only end if both C = A, the X coordinates of the mouse and the cheese match, and B = D, the Y coordinates of the mouse and the cheese match. If these grouping parentheses were omitted, or instead the program `or`ed A = C or B = D with the other termination conditions, the event loop would end if the mouse moved into the same column of the screen as the cheese, even if it wasn't also in the same row, occupying the same spot as the cheese.

Next is the somewhat mysterious conditional `If H=int(H`. I said previously that the valid range for the hunger variable H is from 0 to 8. The screen is conveniently 8 rows of characters tall, so H = 0 is the bottommost row, H = 1 is the second to bottom, H = 7 is the top row, and you never have to worry about drawing the hunger bar for H = 8, because that means the game is already over. These mappings of homescreen row to hunger value are shown in figure B.4.

Figure B.4 The Mouse and Cheese hunger bar and its relationship to the value of variable H

The conditional is there to ensure the program only draws the next notch upward on the hunger bar when the value of H reaches another integer. Because hunger ranges from 0 to 8, if the program added 1 to H every time it went through the event loop, the player would lose quickly, because it would take only 8 iterations of the event loop for H to reach 8. To give the player more of a sporting chance, the program only increases H by 0.1 each time through the event loop; it takes 80 individual 0.1s to make 8 (8/0.1 = 80 or 0.1 * 80 = 8), so this gives the player 80 iterations of the event loop to reach the cheese. One thing you can play with, which I'll discuss later, is that if you change `H+.1→H` to `H+.2→H`, for example, the user will only have 40 iterations of the event loop to get to the cheese, and the game will be harder.

Anyway, because the program increases H by these small values, there's no point in displaying the highest notch in the hunger bar every time the calculator executes the event loop, because it will be redrawing the same equals sign 10 times before H reaches another integer. In addition, it would have to use `Output(8-int(H),16,"=`,

because the calculator can't output at decimal locations on the screen, such as (3.4,16). If you don't believe me, try it: you'll get a Domain error. Therefore, the program only displays a new notch on the hunger bar when H reaches an integer. Because int(3.4) = 3, int(5.8) = 5, and int(4) = 4, the only time int(H) = H will be true is when H is an integer already.

After displaying the hunger bar, the program executes what should by now be a familiar set of event-handling commands. It reads the current key being pressed into variable K, erases the mouse if K is nonzero (meaning that a key is being pressed), then handles the arrow keys. As usual, it checks the four arrow keys for the four major directions and also performs defensive boundary checks so that the mouse doesn't move off the edge of the screen. The body of the event loop ends with the update to the hunger value that I've been discussing.

Ending the outer loop and ending the game

The last piece of the outer loop runs after the inner loop ends. Recall that the inner loop can end as a result of three possible conditions:

- The hunger bar is full, meaning that H = 8.
- The user pressed [CLEAR], so K = 45.
- The mouse and the cheese are at the same coordinates, so A = C and B = D.

You only want to award the player another point and reset the mouse's hunger if the inner loop ended because of the third condition. As a reminder, the end of the outer loop and the end of the program look like this:

```
:If C=A and B=D
:Then
:0→H
:S+1→S
:End:End
:Output(3,1,"----------------
:Output(4,1," YOU ATE
:Output(4,10,S
:Output(5,1,"PIECES OF CHEESE
:Output(6,1,"----------------
:Pause
:ClrHome
```

As you can see, the program only resets H to 0 and increments S by 1 if C = A and B = D. If, on the other hand, K = 45 or H ≥ 8, which means that the inner event loop didn't end because the mouse reached the cheese, this conditional code doesn't execute.

The next line of code is a pair of `End` commands. The first `End` closes the `If/Then` statement, whereas the second `End` completes the outer `Repeat` loop. If K = 45 or H ≥ 8, then the outer loop will also end. If neither of those statements is true, which means the inner loop ended because A = C and B = D, then the outer `Repeat` loop will start again, generating a new piece of cheese and redrawing the game screen. If either K = 45 or H ≥ 8, the last seven lines of the program will run. These first output the number of pieces of cheese eaten during the game and then pause until the player

hits [ENTER]. The game concludes by clearing the screen with `ClrHome`, and because the end of the program file is reached, the program ends.

B.1.3 Tweaking the game

Play with this program as much as you want, to try to change the way it works, make it have more features, or even adapt it into a whole new game of your own. The most important lesson of the Mouse and Cheese game is for you to feel comfortable about `getKey` and event loops and create your own games and programs, but a good starting point to fully grasp the concepts here is to first experiment with an existing program like CHEESE. Don't worry about breaking things; you can always start with a fresh copy of CHEESE if you irreparably break the original in your experimentation, or you could exercise and refine your debugging skills to try to track down what happened.

Among the many possible things you could do to expand the program with new features or adjust the existing gameplay, the following stand out:

- Convert the Mouse and Cheese program to do eight-directional movement. You could lift the key-processing sections of code almost directly from program MOVE8D1 or MOVE8D2; you'd only need to adjust it to stop at column 15 instead of column 16, because the hunger bar is in the rightmost column of the homescreen now.
- Make the game have multiple pieces of cheese on the screen at the same time. How would you go about doing this? You could use (E,F) in addition to (C,D) for a second piece of cheese, and then you'd have to change both conditional checks for the mouse and the cheese being at the same coordinates (`A=C and B=D`) to also check for this second piece of cheese. What if you wanted the player to be able to specify the number of pieces of cheese?
- Make the hunger bar run out faster (or slower). There's a single value in the program that you need to adjust to make this happen. Hint: it's the line where the hunger variable H gets set to a larger value. What sort of values could you use? What if the values you use don't add up exactly to 8? Why does the program crash with an ERR:DOMAIN in that case? Or why does the hunger bar not get updated properly?
- Instead of adding more cheese, add a piece of poison or a mouse trap that the mouse must not touch. How could you implement this? What if even being in one of the squares next to the trap could harm the mouse? How about if you made it only slow down the mouse instead of ending the game?
- Make the cheese move randomly around to make it harder to get.
- Make the cheese move specifically away from the mouse at all times. How could you adjust this so that it's still always possible to get the cheese? Among other things, you'd have to add bounds checking for the cheese, too, so that it wouldn't move off the screen in trying to avoid the mouse.

- Make the screen wrap around, so that when the mouse reaches the right edge and goes right, it reappears at the left edge, or it goes to the top edge when it crosses the bottom edge.

The game as presented could be adapted to other themes besides a mouse and pieces of cheese; you could easily make it any sort of game where a character of some sort needs to collect (or avoid) some sort of object.

B.1.4 Exercise: going further by moving the cheese

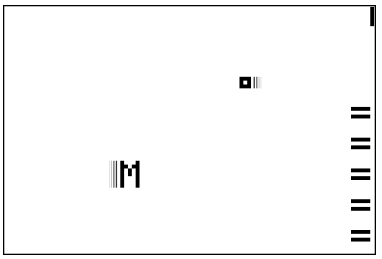


Figure B.5 By allowing the cheese to move randomly around the screen, the game starts to get more challenging.

As a final exercise, you'll try to specifically implement one of the suggestions in the preceding list to see how modifying this program can be done. I'll pick the feature of making the cheese move randomly around the screen to make it harder for the mouse to reach it, as demonstrated in figure B.5. As with all programs you write, you should briefly decide how you'll implement this feature before you dive into the code. Once you know what you want to change, you can then write the new pieces of code. In general, you'll want to directly modify a program when you add new features, but for the sake of this example, I'll first show you the new pieces of code you're adding on their own and then the whole program again with the changes inserted.

The program still deals with one piece of cheese, so it will continue to use (C,D) for the coordinates of the cheese. A new piece of cheese should still appear when the mouse and the cheese reach the same place on the homescreen, so the inner loop can still end when `(A=C and B=D)`. The main change that you need to make is to have the coordinates C and D get randomly modified. Because this must happen at the same time that the mouse is being moved around by the player, these coordinate updates must be in the inner loop. If you put them in the outer loop somewhere, the cheese wouldn't move around until after the player caught it, in which case the feature would be pointless. You know that you want to insert a piece of code somewhere in the inner loop. You should reexamine the original piece of pseudocode for the Mouse and Cheese game to see where you'll need to insert code:

```
:Repeat until hunger bar fills or [CLEAR] pressed
:[initialize cheese and hunger bar]
:Repeat until hunger bar fills or [CLEAR] pressed or mouse reaches cheese
:[move mouse and increase hunger]
:[move cheese]
:End
:[handle eating cheese]
:End
```

As you can see, the cheese must be moved around in the innermost loop at `[move cheese]`, where the program moves the mouse in response to keypresses. Because the

inner loop continues to run whether or not the player presses keys, thanks to the non-blocking quality of `getKey` that I discussed earlier, you can make the cheese continuously move by updating its coordinates in the inner loop.

How can you make it move? I specified that it should move randomly, so you know that you need to use one of the commands that generate randomness, such as `rand` or `randInt`. One possible solution is to add a random integer between -1 and 1 to C and another random integer between -1 and 1 to D. This will make the cheese move in any of the possible eight directions from its starting point or stay in the same place if both random numbers are 0. That should work well, but you still need to make sure that the cheese doesn't go off the screen. Because it's moving around randomly, it could easily get to one of the edges and try to cross the edge, which would yield an exciting ERR:DOMAIN error.

You'll therefore need to do bounds checking. In the original four-direction `getKey` test program, the M only moved to the left (by subtracting 1 from A) if the user pressed the left-arrow key and the M was not already at the left edge:

```
:If K=24 and A≠1
:A-1→A
```

Preliminary movement solution

Your program could do that here, by temporarily putting the two random values into separate variables and then only updating C and D if the updates would keep the cheese on the screen. This particular implementation might look like as follows (remember, in this case the -1 uses the negative sign, the [(-)] key, not the subtraction sign, the [–] key):

```
:randInt(-1,1→E
:randInt(-1,1→F
:If E+C≥1 and E+C≤15
:C+E→C
:If F+D≥1 and F+D≤8
:D+F→D
```

Here the random updates are stored into E and F, and then the program checks if adding those values to C and D will still keep the cheese on the screen. If they will, then the program updates the coordinates with the random changes in E and F; otherwise they're discarded. But for argument's sake, I'll show you an alternative, more efficient method.

Efficient movement solution

Here you'll try a different approach, where C and D are blindly updated with random adjustments and fixed afterward if the random updates accidentally made the cheese go off the edge of the screen. This method saves the use of variables E and F, because you don't need to worry about temporarily storing the results of the two `randInt` commands. But it's a longer segment of code, so it might run more slowly than the previous option. Here's the piece of code that fixes C and D after potentially setting them to values outside the edges of the game space, as just described:

```
:C+randInt(-1,1→C
:D+randInt(-1,1→D
:If C<1:1→C
:If C>15:15→C
:If D<1:1→D
:If D>8:8→D
```

Conditional and conditionally executed command pairs combined on each line

To reduce vertical scrolling while editing the source code, you put the conditional `If` statements and their conditionally executed command (the stores to C and D) on the same line, separated by a colon, rather than hitting [ENTER] in between. As discussed, this is a stylistic choice, and it doesn't change the size or speed of the program. It also has two advantages. First, it saves vertical scrolling, because the set of four direction conditionals are four lines instead of the eight lines they'd take if the program had [ENTER]s instead of colons. This means that this chunk of code is six lines in total instead of ten, so you can see the entire thing on one screen without scrolling at all. Second, it's a way of thinking like a programmer: you know that each `If` statement and the statement that it controls (here, the stores to C and D) are closely related, and you get a visual hint from the two statements being together on the same line of the program.

You can see in the code that C and D are first updated with random values, which might put them outside the game area. The program then runs four separate checks to correct the value of C and D, one for each of the four edges of the area. If C is less than 1, which means the cheese is farther left than the left edge of the screen, the program fixes the problem by setting C equal to 1, the leftmost column of the screen. If C is greater than 15, which means it's either in the column reserved for the hunger bar or off the right edge of the screen, the program sets it to 15 to move it back to the rightmost useable column of the homescreen. The program performs similar checks on D.

REDRAWING THE CHEESE

You need to add one final item: erasing and redrawing the cheese. That's easy enough; you know you need to erase the cheese before updating its coordinates; otherwise the program will be erasing in the wrong place. It then needs to redraw the cheese after updating its coordinates. That leaves the following section of code:

```
:Output(D,C,"[one space]
:C+randInt(-1,1→C
:D+randInt(-1,1→D
:If C<1:1→C
:If C>15:15→C
:If D<1:1→D
:If D>8:8→D
:Output(D,C,"□
```

And that's all the code you need to erase the cheese, randomly move it, and redraw it. You can insert this into the inner loop of the CHEESE program, which we'll now call CHEESE2. You can either put this new block of code before the `getKey`, after all the key-update conditionals, or even after the update to the hunger variable. For the sake

of keeping related pieces of code together, we'll put it after all the keypresses have been processed but before the hunger variable H is updated. The final CHEESE2 program will look like the following listing.

Listing B.2 The Mouse and Cheese game with randomly moving cheese

```
PROGRAM:CHEESE2
:7→A:4→B
:0→H:0→S
:Repeat H≥8 or K=45
:randInt(1,15→C
:randInt(1,8→D
:ClrHome
:Output(D,C,"□
:For(X,1,int(H
:Output(8-X,16,"=
:End
:Repeat H≥8 or K=45 or (C=A and B=D
:If H=int(H
:Output(8-H,16,"=
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]
:If K=24 and A>1
:A-1→A
:If K=26 and A<15
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
:Output(D,C,"[one space]
:C+randInt(-1,1→C
:D+randInt(-1,1→D
:If C<1:1→C
:If C>15:15→C
:If D<1:1→D
:If D>8:8→D
:Output(D,C,"□
:H+.1→H
:End
:If C=A and B=D
:Then
:0→H
:S+1→S
:End:End
:Output(3,1,"----------------
:Output(4,1," YOU ATE
:Output(4,10,S
:Output(5,1,"PIECES OF CHEESE
:Output(6,1,"----------------
:Pause
:ClrHome
```

The newly inserted cheese-moving code

If you'd like to experiment further with the idea of the randomly moving cheese, you could try the other section of random cheese movement code, where I showed you how to provide preemptive protection against the cheese going out of bounds. You could also try making the game slightly easier by running the random updates only sometimes or changing only one of the two coordinates in each inner loop.

With your newfound knowledge about `getKey`, event loops, and more interactive TI-BASIC games, have fun with this example! If you're stuck, try looking at it from a different perspective, or put it down and come back to it later. If you're having trouble seeing how it all fits together, try looking at it in smaller pieces: how the mouse's hunger, score, and coordinates get initialized; how the cheese is placed; how the inner `getKey` loop works; what happens when the mouse gets the cheese; and what happens when the game ends. In particular, look at why this program is written as two nested loops rather than a single big loop with conditional statements. When you write your own `getKey`-based programs, you'll want to be careful to put only the things that must be run repeatedly in your event loop.

B.2 Summary

You should now have a basic understanding of using the `getKey` function as a tool for interactive, fast-paced games. More broadly, you should have a passing familiarity with the idea of the event loop, used to concurrently check for input from the user in the form of keypresses while performing other tasks, such as increasing a hunger level and displaying a hunger bar with that value, running timers, or updating the positions of enemies or items. You should understand the basic differences between an asynchronous event and a synchronous event, because these are important concepts in programming at large. An indirectly related pair of terms that you might have picked up is blocking and nonblocking input, another distinction you'll find yourself using in your calculator programs and in your further programming languages, should you continue on to other platforms.

Sample lesson plan C Programs and pixels

If you tried to represent a graph, a 3D corridor, a puzzle game with expansive puzzles, or any of similar components of thousands of available TI-BASIC programs and games on the homescreen, you'd be disappointed. Luckily, you need not lose hope: your trusty TI-83 Plus/TI-84 Plus calculator has the answers you need in the form of controlling individual pixels in the LCD, the dots that make up any character or image. Although many computer programmers might scoff at having 96 × 64 pixels to work with, you'll soon find that after using the homescreen's 16 × 8 characters, this is more than enough expanse to build complex and feature-rich math and science programs and games that you'll want to play over and over.

In this lesson plan, I'll show you how to turn the individual pixels on and off and demonstrate two full example programs that display text in a smaller font than the homescreen uses.

C.1 Playing with pixels

The TI-83 Plus/TI-84 Plus LCD is made up of 6,144 pixels, tiny points that can be either white or black. The TI-84 Plus C Silver Edition has a 320 × 240-pixel color LCD, and TI-BASIC programmers can use 265 × 165 of those. Clever assembly-language programmers have managed to turn the pixels on and off fast enough to make grayscale, but for TI-BASIC purposes, your programs will be setting pixels to either black or white. I'll introduce four commands that will let you directly manipulate pixels: three to change the color of individual pixels and one to determine the current color of a specified pixel. In this section, we'll begin with the four pixel commands, `Pxl-On`, `Pxl-Off`, `Pxl-Change`, and `pxl-Test`. You'll see how to draw a simple mouse cursor with these commands, then how, in two steps, to make it a moveable mouse.

The first step to combining `Pxl-` commands with your existing knowledge is understanding what `Pxl-` commands exist and how to use them.

C.1.1 Pixel commands

The four pixel-based drawing commands are all quite straightforward. They all take precisely two arguments, and those two arguments are all (row, column), or if you prefer, (y, x). They're all found in the POINTS tab of the DRAW menu, [2nd][PRGM][▶], demonstrated at the left side of figure C.1. Options [4], [5], and [6], the `Pxl-On`, `Pxl-Off`, and `Pxl-Change` commands, all modify the state of a pixel, also shown at left in figure C.1. The `pxl-Test` command is the odd one out, returning a 1 if the specified coordinates contain a black pixel and a 0 for a white pixel. The right side of figure C.1 shows the results from using blocks of nine `Pxl-On`, `Pxl-Off`, and `Pxl-Change` commands on black and white backgrounds, as well as the output value of `pxl-Test` on black and white pixels. As you can see, black pixels are considered on (or 1) and white pixels are off (or 0).

The following snippet of code demonstrates each of the four commands in use:

```
:Pxl-On(1,1
:Pxl-Off(42,80
:Pxl-Change(13,37
:If pxl-Test(42,80
:Then
:Disp "(42,80) BLCK PXL
:Else
:Disp "(42,80) WHT PXL
:End
```

This would turn a pixel near the top-left black, one near the lower-right white, and switch a pixel near the middle of the top edge from black to white or white to black. Because the `Pxl-Off` command turns the pixel at row 42, column 80 to white (off), which is 0 or false, the `If/Then/Else/End` condition displays "(42,80) WHT PXL." Recall that the `If` line is equivalent to the following and that `pxl-Test` returns 1 for black pixels and 0 for white:

```
:If 1=pxl-Test(42,80
```

With the basics of the commands in hand, let's do something fun with them: draw a mouse-style cursor.

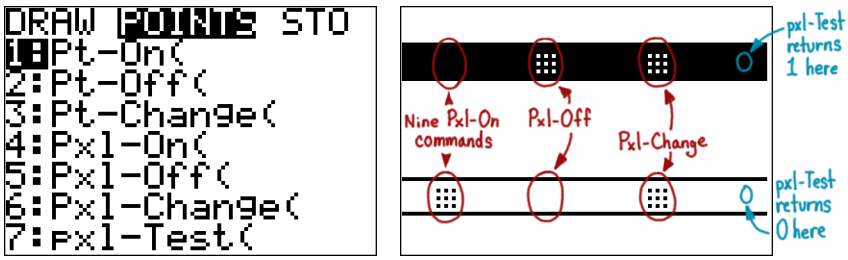


Figure C.1 The four pixel-based commands we'll be using from the DRAW menu's POINTS tab (left) and their effects on black-and-white backgrounds (right)

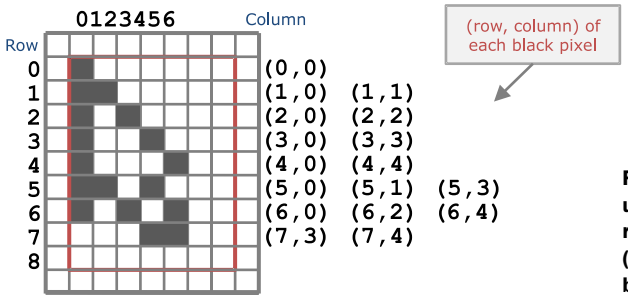


Figure C.2 A "sprite," or set of pixels, used to create a mouse cursor. The right side of the figure shows the (row, column) coordinates of each black pixel.

C.1.2 Drawing a cursor

By the end of the next section, I want you to have a cursor that you can move around your calculator's screen with the arrow keys. A program like MOVETEXT already gives you a good framework to work with, because it lets you move an M around and already handles turning the axes on and off, clearing the screen when appropriate, and working with keys. But the `Text` commands to draw and erase the M will no longer do the trick. For starters, you need to figure out how you'll draw a mouse cursor. Figure C.2 shows a simple example sketched in black and white pixels, spanning eight rows and five columns. Because you know that you want to draw this cursor with `Pxl-Off/On/Change` commands (because that's the only way you know to manipulate individual pixels), you need to start by figuring out the coordinates of each pixel.

You always label graphscreen pixels with the lowest row and column numbers at the top left, so the tip of the cursor is (0,0); the full list of pixel coordinates for this mouse relative to (0,0) is shown at right in figure C.2. If you rendered this entirely with `Pxl-On` commands, it might look something like the left side of the code in the following listing.

Listing C.1 Two versions of a cursor-drawing subprogram, ZCURSORA/ZCURSORB

```
PROGRAM:ZCURSORA
:Pxl-On(0,0
:Pxl-On(1,0
:Pxl-On(1,1
:Pxl-On(2,0
:Pxl-On(2,2
:Pxl-On(3,0
:Pxl-On(3,3
:Pxl-On(4,0
:Pxl-On(4,4
:Pxl-On(5,0
:Pxl-On(5,1
:Pxl-On(5,3
:Pxl-On(6,0
:Pxl-On(6,2
:Pxl-On(6,4
:Pxl-On(7,3
:Pxl-On(7,4
```

```
PROGRAM:ZCURSORB
:Pxl-Change(B,A
:Pxl-Change(B+1,A
:Pxl-Change(B+1,A+1
:Pxl-Change(B+2,A
:Pxl-Change(B+2,A+2
:Pxl-Change(B+3,A
:Pxl-Change(B+3,A+3
:Pxl-Change(B+4,A
:Pxl-Change(B+4,A+4
:Pxl-Change(B+5,A
:Pxl-Change(B+5,A+1
:Pxl-Change(B+5,A+3
:Pxl-Change(B+6,A
:Pxl-Change(B+6,A+2
:Pxl-Change(B+6,A+4
:Pxl-Change(B+7,A+3
:Pxl-Change(B+7,A+4
```

But this only allows you to draw the cursor at the top-left corner of the LCD and gives you no way to erase it again without clearing the screen, which is often something you'll want to avoid if you have many things on the screen and want to erase only one. The program at the right side of listing C.1, ZCURSORB, solves both of these problems. By using `Pxl-Change` commands, running the program once will draw a mouse, changing all the white pixels to black in the shape of the cursor. If you run the program again, `Pxl-Change` will change all those black pixels back to white, erasing the cursor. In addition, the insertion of the A (X-coordinate or column) and B (Y-coordinate or row) variables, here used as offsets, means that instead of being drawn relative to (0,0), the cursor is now drawn relative to (B,A). If you change A and/or B, the cursor will be drawn at a different location.

Now that you have a small program that can both draw and, if called again, erase a mouse cursor at some coordinates (B,A), you can combine this with fragments from MOVETEXT. In the final subsection of this section, I'll challenge you to put this together yourself and then show you how it's done.

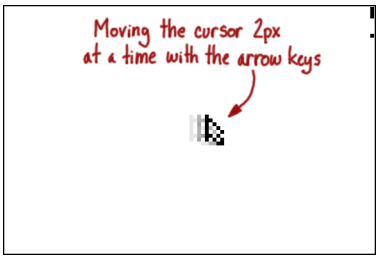


Figure C.3 The results of the CURSOR program you'll write in section C.1.3, a mouse cursor that moves

C.1.3 Exercise: the moveable mouse cursor

For this exercise, you'll create a program to move a mouselike cursor around the graphscreen, as in figure C.3. You'll be modifying the MOVETEXT program from listing C.2 into a new program called CURSOR.

Listing C.2 MOVETEXT

```
PROGRAM:MOVETEXT
:45→A:28→B
:AxesOff:ClrDraw
:Repeat K=45
:Text(B,A,"M
:getKey→K
:If K
:Text(B,A,"[three spaces]
:If K=24 and A≠0
:A-1→A
:If K=26 and A≠91
:A+1→A
:If K=25 and B≠0
:B-1→B
:If K=34 and B≠57
:B+1→B
:End
:AxesOn
```

Repeat this loop until the [CLEAR] key is pressed, at which point the loop ends

Text uses a variable-width font: an M is three pixels wide plus one padding column, a space is one pixel wide. Three spaces are needed to draw over the Ms three pixels of width.

Left-right movement and bounds checking. The M is four pixels total, and the columns range from 0 to 94, so the M can start in column 0 through column 91 safely.

Up-down movement. The M is 6 pixels tall, 5 plus 1 padding row, and the LCD's rows range from 0 to 62. The M can start in rows 0 through 57 safely.

When the Repeat loop ends, be polite and turn the axes back on before terminating.

Listing C.2 introduces a new command, Text, alongside `ClrDraw`, `AxesOn`, and `AxesOff`. If you test `prgmMOVETEXT`, you'll see it lets you move a letter M (slightly smaller than a homescreen M) around the screen, but when you press any arrow key, it moves by only a single pixel in that direction. If you have any graph functions active, you may see them behind the M symbol. If you want to copy and paste the contents of MOVETEXT, use the `Rcl` function. For the actual drawing and erasing of the mouse cursor, use the ZCURSORB program as a. Because it's slower to call `prgmZCURSORB` than to use a single `Text` command, try making the mouse move 2 pixels at a time. As a hint, your program will need to have the two `Text` commands in `prgmMOVETEXT` changed, as well as the numbers used for bounds checking and the numbers for updating A and B, but the structure will remain more or less the same. The only structural change should be putting the `getKey` in a tight `Repeat` loop, because running ZCURSORB repeatedly won't repeatedly draw the M over itself with no visual change, as running `Text(B,A,"M` did. Instead, it will repeatedly draw and erase the mouse in place, an unwanted side effect.

Once you have an idea of what your program will look like, or better yet, have written and tested it, read on for my solution.

THE MOVEABLE CURSOR: SOLUTION

My proposed solution for CURSOR is shown in listing C.3, and it requires that you also have ZCURSORB from listing C.1 on your calculator.

Listing C.3 The moveable cursor program, CURSOR

```
PROGRAM:CURSOR
:44→A:28→B
:AxesOff:ClrDraw
:Repeat K=45
:prgmZCURSORB
:Repeat K
:getKey→K
:End
:prgmZCURSORB
:If K=24 and A≠0
:A-2→A
:If K=26 and A<88
:A+2→A
:If K=25 and B≠0
:B-2→B
:If K=34 and B<54
:B+2→B
:End
:AxesOn
```

The best way to explain a program like this would be to step through it line by line, explaining each chunk.

```
:44→A:28→B
:AxesOff:ClrDraw
```

Initialize A (the X-coordinate) and B (the Y-coordinate) to roughly the center of the screen; then turn off the axes and clear the screen.

> **Multiple commands on the same line**
>
> When you insert a colon between two or more commands, you can put them next to each other without adding a line return (new line) between each command. The only caveat is that you can no longer omit ending parentheses and quotes to save space. The following four commands fit on two lines. This technique is used to save vertical scrolling in long programs.
>
> ```
> :45→A:28→B
> :AxesOff:ClrDraw
> ```

```
:Repeat K=45
```

Repeat the main loop of this program until the [CLEAR] key is pressed.

```
:prgmZCURSORB
```

Call `prgmZCURSORB` to draw the cursor on the screen at row B, column A.

```
:Repeat K
:getKey→K
:End
```

With the cursor on the screen, run a tight `Repeat` loop, which repeats until K is not 0 (there's an implicit =1 or ≠0 when you have a variable by itself as a conditional). Because `Repeat` loops are guaranteed to run at least once, the first run through the loop with no key pressed will set K to 0, allowing the loop to continue without needing to set K to 0 before the beginning of the loop.

```
:prgmZCURSORB
```

When the `Repeat K` loop ends, a key has been pressed; erase the cursor by running `prgmZCURSORB` again:

```
:If K=24 and A≠0
:A-2→A
:If K=26 and A<88
:A+2→A
:If K=25 and B≠0
:B-2→B
:If K=34 and B<54
:B+2→B
```

Perform bounds checking while updating A and B. The left- and right-arrow keys move the X-coordinate A as far left as 0 or as far right as 88. Technically, because the cursor is 6 pixels wide, it could begin in column 89 and span only as far as column 94. Because the program moves the cursor 2 pixels at a time, it would jump from 88 to 90, at which point the program would throw an ERR:DOMAIN when it tried to change a

pixel in column 95, the 96th column. This code also moves the cursor as high as row 0 or as low as row 54. A and B are initialized to even numbers, all updates are in increments of 2, and bounds are checked with even numbers, so all coordinates in this program are always even numbers.

```
:End
:AxesOn
```

End the outer `Repeat` loop, and if it terminates, turn the axes back on (which clears the graphscreen) and reach the end of the program to conclude it.

As with any program you write or that I show you, if there's anything you don't understand, try it out on your calculator, examine and tweak the code, and even add debug statements. If you're still vague on any of the concepts, read on to the next section, where I'll show you a program that lets you doodle on the screen.

C.2 A painting program

For the final full program, I'll expand the concepts of the CURSOR program to let you doodle on the screen. Instead of moving a mouse cursor around the screen that erases after itself as it goes, you'll be moving a single flashing pixel around the screen with the arrow keys. But pressing [ENTER] in this program is like putting a pencil or pen down on a piece of paper: now the flashing pixel will draw a line behind it as it moves. Press [ENTER] again to lift up the writing implement, and when you've finished, press [CLEAR] to quit. To start, take a look at figure C.4, which from left to right shows a sample session using the program.

I'll begin by showing you the code for this PAINT program and explain a bit about how it works. At this point, the flow of such programs should be gradually getting clearer to you, so I'll pick out specific features to highlight rather than tracing it line by line.

Coding a painting program

The source code for `prgmPAINT` is shown in listing C.4. It follows the structure of many programs that are based on moving something around with `getKey`. Like the CURSOR program, it uses two nested `Repeat` loops, an outer one to handle keypresses until [CLEAR] is pressed and an inner `Repeat` loop to flash a pixel on and off while it

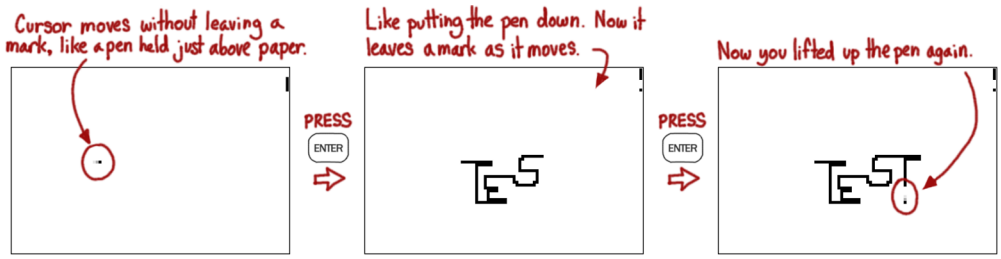


Figure C.4 A demonstration of `prgmPAINT` in action. Moving the pointer around at far left without drawing and then pressing [ENTER] to hold the pen against the "paper," as in the middle screenshot. Pressing [ENTER] again to lift the pen again, as in the far-right screenshot.

waits for any keypress. It does graph setup and variable initialization before the outer loop and cleans up after the end of the outer loop.

Listing C.4 A painting program, `prgmPAINT`, using `Pxl` commands and `getKey`

```
PROGRAM:PAINT
:AxesOff:ClrDraw
:48→A:32→B
:0→P
:Repeat K=45
:Repeat K
:Pxl-Change(B,A
:getKey→K
:Pxl-Change(B,A
:End
:If K=24 and A>0
:A-1→A
:If K=26 and A<94
:A+1→A
:If K=25 and B>0
:B-1→B
:If K=34 and B<62
:B+1→B
:If K=105
:1-P→P
:If P
:Pxl-On(B,A
:End
:AxesOn
```

Turn off the axes and clear the graphscreen

Initialize the (B,A) = (row, column) to about the center of the screen

Initialize the P variable, indicating whether the "pen" is touching the "paper," to 0, or false

Flash a pixel to show the currently selected pixel while waiting for a keypress

Toggle the "pen" touching (P = I) or not touching (P = 0) the "paper"

If the "pen" is touching the "paper," turn this pixel on

Like the MOVETEXT and CURSOR programs, this program uses variables A and B to store the X- (column) and Y- (row) coordinates of the pixel currently selected. It also uses variable P to store the current state of the virtual "pen," whether it's touching the paper (P = 1) or not (P = 0). It controls whether a `Pxl-On` command runs right after any movement command, so that when P = 1, the program turns on the pixel each time it moves to a new pixel:

```
:If P
:Pxl-On(B,A
```

The [ENTER] key toggles P between 0 and 1 using the same math trick as `prgmMODKEYS` in listing 6.5 . When the [ENTER] key is pressed, P is set to 1 – P.

```
:If K=105
:1-P→P
```

If P is 0, 1 – P = 1 – 0 = 1, so P switches to 1. If P is 1, 1 – P = 1 – 1 = 0, so P switches to 0.

The final interesting bit of code is the inner loop that displays the swiftly flashing cursor while the program waits for the user to press a key:

```
:Repeat K
:Pxl-Change(B,A
:getKey→K
```

```
:Pxl-Change(B,A
:End
```

This code is designed to leave the pixel as it originally was when it ends. As figure C.1 and the CURSOR program showed, applying `Pxl-Change` twice to the same pixel changes it back to whatever it started as. The first `Pxl-Change` either changes a white pixel to black or a black pixel to white, and the second `Pxl-Change` switches it back. Because there are two `Pxl-Change` commands inside the `Repeat` loop, no matter how many times the loop runs, an even number of `Pxl-Change`s occur, and the pixel at (row,column) = (B,A) will always be the same after the loop ends as before the loop began. While inside the loop, the pixel will flash on and off fast to indicate that that's the currently selected pixel.

As a final note, let me clarify the logic on the `Pxl-On` command that runs when P = 1 is placed at the end of the outer `Repeat` loop. As always, we want our programs to be as small and fast as possible. Therefore, we only want to run the `Pxl-On` command that draws on the virtual paper when either the "pen" is toggled to P = 1 (touching the paper) or the "pen" moves to a new location due to the user pressing the arrow keys. Both of these conditions are handled when the end of the outer `Repeat` loop runs, because the inner loop ends when K ≠ 0, or a key has been pressed. Thus, the end of the outer `Repeat` loop is a logical place for the `Pxl-On` command.

C.3 Summary

These were your first steps of programming with the graphscreen. This lesson plan introduced working with individual pixels and how you can use the graphscreen in your own programs and games. You saw a few example programs, including a program to move a mouselike cursor around the screen and a doodling program, both of which combined the event loop lessons of the previous chapter with the precision and control of the graphscreen.