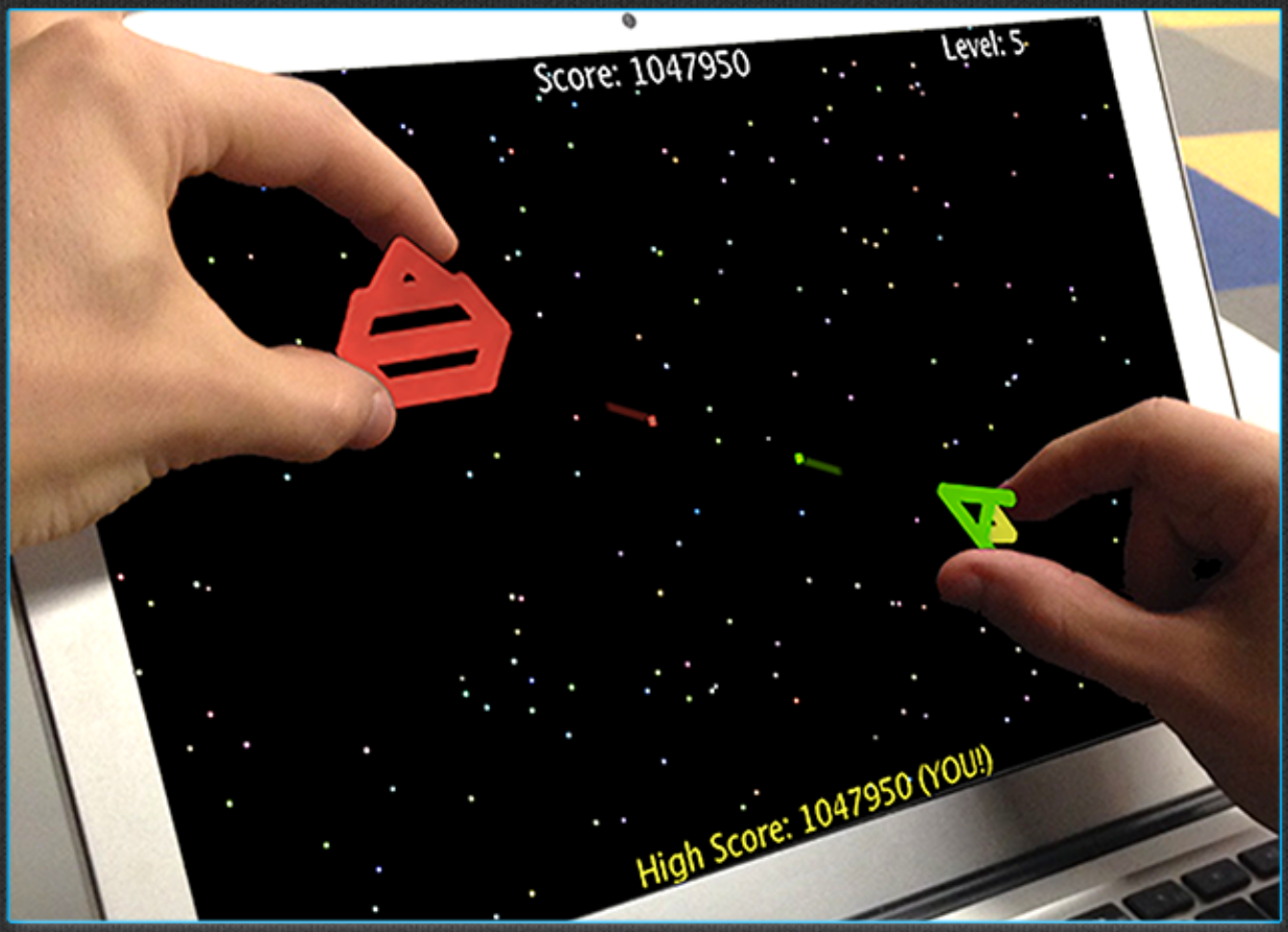
# Hands-On Intro to Game Programming

By Chris DeLeon




With code for 6 classic games

*I'd like to thank Maanie Hamzaee, Laura Schluckebier, Allison Morrow, my brother John, mother Judy, and father Louie. Thanks for your years of encouragement, feedback, and support. Nothing in life is ever a completely solo effort.*

1 Introduction ..... 6
1.1 Hello and Welcome ..... 6
1.2 About Processing ..... 7
1.3 Coding Style Disclaimer ..... 7
1.4 Never Programmed Before? ..... 8
1.5 Installing and Setting Up the Source Examples ..... 10
1.6 About the Bonus Code Included ..... 10
2 Tennis Game ..... 12
2.1 Warm-Up Exercises ..... 12
2.1.1 Triple the height of the player paddles ..... 12
2.1.2 Tune the score required to win ..... 13
2.1.3 Invisible player ..... 14
2.1.4 Try implementing reflect-style ball behavior ..... 15
2.2 Practice Exercises ..... 17
2.2.1 Ball-speed increase as it stays in play ..... 17
2.2.2 Graphics ..... 18
2.2.3 Move player paddles away from sides ..... 19
2.2.4 Add sound events and effects ..... 20
2.2.5 Two player mode option (keyboard controls) ..... 21
2.3 Challenge Exercises ..... 23
2.3.1 Ball trail using an array of past positions ..... 23
2.3.2 AI that accounts for reflections ..... 24
2.3.3 Calculate intersection/intercept when crossing paddle line ..... 25
2.4 Write From Scratch Steps: Tennis Game ..... 26
3 Brick Break ..... 31
3.1 Warm-Up Exercises ..... 31
3.1.1 Code organization: multiple files ..... 31
3.1.2 Keep and display score ..... 32
3.1.3 Life limit then reset ..... 33
3.1.4 Click to serve the ball from the paddle each round ..... 33
3.1.5 Keep ball from getting stuck off screen ..... 33
3.1.6 Minor optimization: brick countdown ..... 34
3.1.7 Images for bricks, paddle, ball, background ..... 35
3.1.8 Icons for lives ..... 36
3.1.9 Title screen ..... 36
3.2 Practice Exercises ..... 37
3.2.1 More points for hit chains ..... 37
3.2.2 Ball-speed increases ..... 38
3.2.3 Extra lives for score milestones ..... 38
3.2.4 Power-up functionality: fireball ..... 39
3.3 Challenge Exercises ..... 39
3.3.1 Bricks loaded from 2D array in code ..... 39
3.3.2 Different brick types ..... 40
3.3.3 Implement modern brick collision ..... 40

3.3.4 Try classic ball-brick collision rules ... 41
3.3.5 Load level from file ... 41
3.3.6 Code organization: classes ... 42
3.3.7 Power-ups: cannon, multiball, sticky ball, points ... 43
3.3.8 Series of levels (and design them!) ... 44
3.4 Write From Scratch Steps: Brick Break ... 44
4 Bomb Dropper ... 52
4.1 Warm-Up Exercises ... 52
4.1.1 Adjust timing and scoring to your liking ... 52
4.1.2 Difficulty modes with different settings ... 53
4.1.3 Keep high score (in session) ... 54
4.1.4 Find and add appropriate creative commons music ... 55
4.2 Practice Exercises ... 55
4.2.1 Two-player mode (keyboard for attacker) ... 55
4.2.2 Save high score to file, give way to reset it ... 56
4.3 Challenge Exercises ... 57
4.3.1 Items that should be dodged ... 57
4.3.2 Non-random adversarial AI ... 57
4.3.3 Particle effects for explosions ... 58
4.4 Write From Scratch Steps: Any Specific Classic Game ... 59
5 Racing ... 64
5.1 Warm-Up Exercises ... 64
5.1.1 Headlights on Player 2 car ... 64
5.1.2 New tile types: grass, oil slicks ... 65
5.1.3 Day/night or theme tile sets ... 66
5.2 Practice Exercises ... 67
5.2.1 Load track images as array ... 67
5.2.2 Collision at front and rear of car ... 67
5.2.3 Laps with checkpoints ... 69
5.2.4 Boost/nitro each lap ... 71
5.2.5 Car-to-car collision (basic) ... 72
5.2.6 Single player option (random P2 movements for now) ... 73
5.2.7 Transition phase at start/end of stages ... 74
5.2.8 Race time based in real-time ... 74
5.3 Challenge Exercises ... 75
5.3.1 Ramp tiles and airborne cars ... 75
5.3.2 Better computer driver ... 77
5.3.3 Scrolling for larger track, Pt. 1 (1 player only, for now) ... 77
5.3.4 Scrolling for larger track, Pt. 2 (2 player split-screen) ... 79
5.3.5 Rain weather mode, with slippery driving/drifting ... 79
5.3.6 Multiple tracks ... 80
5.3.7 In-game track editor GUI ... 82
5.4 Write From Scratch Steps: Racing ... 83
6 RTS Game ... 98
6.1 Warm-Up Exercises ... 99

6.1.1 Try to win! ... 99
6.1.2 Clear select on right click ... 102
6.1.3 Spread out formation ... 103
6.1.4 Remove targeting line ... 104
6.1.5 Make attack line look more like a laser ... 104
6.1.6 Hide health unless selected or near mouse ... 105
6.1.7 Change health color when low or max ... 107
6.2 Practice Exercises ... 108
6.2.1 Background graphics ... 108
6.2.2 Unit graphics with team coloration ... 109
6.2.3 Heavy unit ... 112
6.2.4 Color key collision affecting mobility, range, and visibility ... 114
6.3 Challenge Exercises ... 116
6.3.1 Scrolling camera with minimap ... 116
6.3.2 Fog of war ... 117
6.3.3 Projectile attacks with splash damage ... 118
6.3.4 Draw order from front to back ... 119
6.3.5 Non-random computer team behaviors ... 121
6.3.6 Resources, build menu, and harvesters ... 121
6.4 Write From Scratch Steps: RTS Game ... 122
7 Space Battle: Full Game in Steps ... 135
7.1 About the Seven Code Steps ... 135
7.1.1 spaceBattle0: First Code ... 135
7.1.2 spaceBattle1: Core Gameplay, Section Comments ... 136
7.1.3 spaceBattle2: Splitting Into Functions ... 136
7.1.4 spaceBattle3: Organizing with Files ... 136
7.1.5 spaceBattle4: Refactoring to use Classes ... 137
7.1.6 spaceBattle5: Arrays and Score Display ... 138
7.1.7 spaceBattleFinal: Title Screen, High Scores, Levels, Stars ... 138
7.2 Warm-Up Exercises ... 139
7.2.1 Custom graphics ... 139
7.2.2 Put star code in its own file ... 140
7.2.3 High scores list ... 141
7.3 Practice Exercises ... 141
7.3.1 Different ship types ... 141
7.3.2 Different weapon types per ship ... 142
7.3.3 Two player support ... 142
7.4 Challenge Exercises ... 143
7.4.1 High scores initials ... 143
7.4.2 Power-ups dropped by enemies, vanish if not collected ... 143
7.4.3 Enemy UFOs accounting for screen wrap ... 144
7.4.4 Particle effects explosions ... 144
7.4.5 Computer-controlled ships with player-like weapons ... 145
7.4.6 Ship and upgrade economy ... 146
7.5 Write From Scratch Steps: Space Battle ... 146

# 1 Introduction

## 1.1 Hello and Welcome

I want to help you make videogames. I want to help you gain the ability to create worlds and original mechanics, and to share what's in your imagination with others. Before digging into the new and huge stuff, though, I'm a firm believer in the value of first learning some classics and fundamentals.

How do great chefs begin? By following known recipes.

How do piano players begin? By learning how to play recognizable tunes of an appropriate skill level.

How do craftsmen and craftswomen begin? By going through a few small standard projects that lead to familiarity with the tools, terms, and techniques.

I apply a similar mentality for learning videogame development. It's an approach that has helped me significantly. I've seen it work for peers and students. Similar advice appears again and again in infinite variations in online game development forums:

Start small.

Smaller.

No, smaller than that.

Make some simple retro games.

Well I'm here to help you do just that, one feature and facet at a time.

If you've already done some programming before, and are here to transition from general programmer to game programmer, jump right in! If on the other hand you're totally new to programming, I recommend first checking out section 1.4 for some general programming intro videos I've prepared. Game development brings some particular challenges, atop the fundamental challenges of programming, and a handle on the most common jargon and beginner issues will really help you get the most out of the content that's here.

This isn't the typical kind of source code that comes out of released game projects. That tends to be tough to learn from for a myriad of reasons. I've specifically authored these projects in a simplified manner to be more easily understood by newer programmers. I've put a lot of work and time into preparing an approach to the material that I believe strikes the right balance between guided tips, so that you've got a helpful head start on working out the details, while still keeping the problems open-ended enough to provide an authentic recreation of the same sorts of practical implementation challenges that you'll be facing for a lifetime of making videogames.

I recommend keeping the Processing compiler open with the example code while browsing this document. It may also be helpful to keep open a browser to the platform's reference pages at `Processing.org`. Feel free to skip around between exercises, games, and sections. I've put them in an order that makes sense to tackle

first to last within each project, however whenever possible I've done my best to author them in a way that allows most of the exercises to stand alone. Practice exercises on any game are harder than Warm-Ups for any game later in the document. Page number isn't difficulty level. Jump around!

Speaking of which: please don't just read the code end-to-end, top-to-bottom - source code is not a book, and that's certainly not how I wrote any of it (see the Write From Scratch Steps to better understand the order in which these things come together). Tinker with it! Retune it! Extend it! Become more comfortable and familiar with programming through consistent, deliberate, and varied practice. Play the games a little to get to know the mechanics. Try commenting things out. Experiment with numerical tweaks. Adapt the exercises to suit your interests and ability. Take what you've learned from working on one game and apply it to another. These exercises are intended as a starting point, some initial direction and guidance. They are not an exhaustive list of what programming can do. Programming can do virtually anything!

Be on the lookout for something that you're genuinely excited to work on. Taking that on is when and how you're going to learn the most.

Happy Developing!

Chris DeLeon

## 1.2 About Processing

Processing is a free development environment available for PC, Mac, and Linux available from `http://processing.org`. Processing was created to enable beginning developers to think and build real-time interactive visuals without getting bogged down in the overhead of setting of core libraries or wrestling with compiler flags and makefiles. Unlike other introductory development environments that aim to abstract away common programming elements (functions, variables, loops, conditionals, arrays, classes, and so on) Processing reveals and focuses on those aspects. That's incredibly useful to us as game creators, since other major programming languages used for game development including C/C++, ActionScript 3 (Flash), HTML5/JavaScript, C# (Unity scripting included), and Java (the underlying foundation for Processing) share the same essential programming elements, making it easier to transition to new platforms.

Note that during the development of these projects I tested with Processing 2.0.2, 2.0.3, and the latest at the time of this writing, version 2.1. All versions seemed to work equally well. If however any of the projects don't compile or don't behave as expected with a newer version of Processing, it may be worth trying it with one of these versions to see whether that's the cause.

## 1.3 Coding Style Disclaimer

Newer programmers sometimes expect that there's only one way to get a program to do something. A variation of this thinking is the impression that even

if there are, perhaps, roundabout or inefficient ways to do the same task, surely there is one best way to achieve the result? There are however even many different best ways to write a program to achieve the same purposes, varying not only in superficial layout differences, equivalent phrasing, and variable naming choices, but also in more substantive ways depending on how we measure what's best. Differences in organizational structure, control flow, and programming features used affect tradeoffs between run-time efficiency, programming time, extensibility, readability, maintainability, security, etc. There are many different ways to write the same program.

For the source included in these projects I've prioritized readability and simplicity. Above all, it's important that the code makes sense to new game programmers. This is not intended as a guide to help intermediate-level programmers learn how to write more efficient or advanced code prepared for collaborative programming, it's for helping someone figure out as quickly as possible how to get the gameplay working. I stick with basic programming elements in straightforward patterns. This isn't just a purely educational approach though, as a similarly simplified style has a place as well in certain situational demands of professional and hobbyist game development. Whether rapid prototyping to demonstrate and test an idea, fitting more features into an already tight schedule, or keeping code simple to minimize emergent bugs, the style here has immediate application to full videogame development tasks.

Lastly, since experience and comfort vary among novice game programmers, in these examples I've included a range in complexity. For Space Battle the range is in the differences between the functionally similar (mostly just "refactored" - meaning cleaned up) middle-progress steps. In the other game examples the range varies among projects, with some being relatively flat (in which case reorganization into files or classes happens in exercises) while others come a bit more structured. The level of detail provided in the Write From Scratch Steps also varies between examples. So if the Write From Scratch Steps that you're browsing isn't at the right level of abstraction for what you're looking for today, jumping to another may work.

## 1.4 Never Programmed Before?

In the fall 2013 semester at Georgia Tech I was a Teaching Assistant for LMC 6310, Computer as an Expressive Medium. Outside of my time in the lab helping students with their projects, on my own time I developed a series of free introductory programming videos. For students in the course, this served as review material. Meanwhile, hosting it on YouTube enabled us to share that content with people not at Georgia Tech. These videos can be found here:

http://www.hobbygamedev.com/beg/programming-lab-videos/

If you've never programmed before, you're of course welcome to dive right into these exercises and sample code. I suspect you'll still figure out and learn quite a

lot in the process. However if you're the type of learner that feels more comfortable first building up a bit from the basics, preferring to better prepare yourself with context and concepts in order to get more out of the exercises and game code when cracking these example projects open, most of those free videos I prepared as an extension of that course's material can come in handy. Even if you have already learned some programming basics before but would like a little review, then of course these videos may be helpful in that case, too.

The sections available at the time of this writing, including direct video URLs (a reminder: web addresses, like programming code, are case-sensitive - capital and lowercase count as different characters!), are as follows:

Programming 1: Get Processing, What Can Processing Do, Text Log, Rectangle, Comments, Mouse
https://www.youtube.com/watch?v=aye6f7YnEuw

Programming 2: Bouncing Ball, Creating Functions, Refactoring, Splitting Project Into Files
https://www.youtube.com/watch?v=lQJinfBpfYk

Programming 3: Essential Programming Features
https://www.youtube.com/watch?v=M0m3mzUeu1E

Programming 4: Classes and ArrayLists for Bouncing Balls *(very important concepts in game programming, used in many of these example projects!)*
https://www.youtube.com/watch?v=GSk3TBqalw8

Programming 5: Dynamic Trail, Image File, and Font
https://www.youtube.com/watch?v=VMWzZdbSKYY

Programming 6: Save Image, Title Screen, Key Input
https://www.youtube.com/watch?v=DG8INpy4bls

Programming 7: Using Twitter API in Processing *(not as relevant for real-time games, could easily skip this one)*
https://www.youtube.com/watch?v=gwS6irtGK-c

Programming 8: Reading Local Text Files and Web Text
https://www.youtube.com/watch?v=T_AvPslt3Po

The last batch in the series, consisting of videos 9-12, is especially useful for understanding some common game programming concepts:

Programming 9: 2D Grid Tile Map and Keyboard as Buttons
https://www.youtube.com/watch?v=FmAe0Y7JPoE

Programming 10: Basic AI Chase Exercise
https://www.youtube.com/watch?v=qQlBseEziC4

Programming 11: Ball Roll Minigolf Motion Exercise
https://www.youtube.com/watch?v=SqdaFwte19c

Programming 12: Color Key Collision Map with AI
http://www.youtube.com/watch?v=3AA9Hs5RfsA

## 1.5 Installing and Setting Up the Source Examples

The .pde source files included in the zip file along with this document are Processing code, which requires the free Processing development environment from `http://Processing.org` to be run. Once Processing is unzipped and run, the entire source directory included with this ebook can be copied into where your sketchbook folder is located for Processing. That folder is typically in `User/My Documents/Processing` for Windows, or `User/Documents/Processing` for Mac. After copying the unzipped game code directories into that Processing folder, exiting then reopening Processing should list the source, still organized in its subdirectories, within the projects Sketchbook (Sketchbook is in the `File` menu when Processing is open). Even if for whatever reason the sketches can't be found listed under Sketchbook there are still other ways to open and modify the projects: use `Open...` from Processing's `File` menu to pick any .pde file in the directory of a project. That will open all files in that project folder as tabs.

The source files need to be unzipped from the compressed folder that they are initially downloaded in, since only then can the files access one another and to the image, audio, and level data files. Something important to be aware of with Processing is that it requires a project's main source file share the same name as the directory containing the file - so if you've duplicated a project and renamed the folder as a way to get started, that's perfectly fine (a great idea, even!), however the main source .pde file will also need to be renamed to whatever the containing folder's new name is.

Set up demo video: http://www.youtube.com/watch?v=fGw-rUjx6o8

## 1.6 About the Bonus Code Included

In addition to the videogame example source code and exercises, you'll find a `commonBonus` subfolder, as well as a `bonusPrototype` folder. Inside `commonBonus` are a number of additional demonstration Processing sketches, including `mini-demos`: tiny starting points each showing just one or two features, and `free-from-hobbygamedev.com` which has a number of other examples. In the `lab-examples` folder you'll find the end code written for the YouTube introductory videos I listed above. The `plane-src-examples-all` and `Platformer` projects are early precursor to the types of exercises designed for this *Hands-On Intro to Game Programming*:

both were written as gameplay demonstration code for workshop and educational purposes. For more about the Plane example, see the following HobbyGameDev entry online:

http://www.hobbygamedev.com/int/the-making-of-a-10-thunderbolt-2/

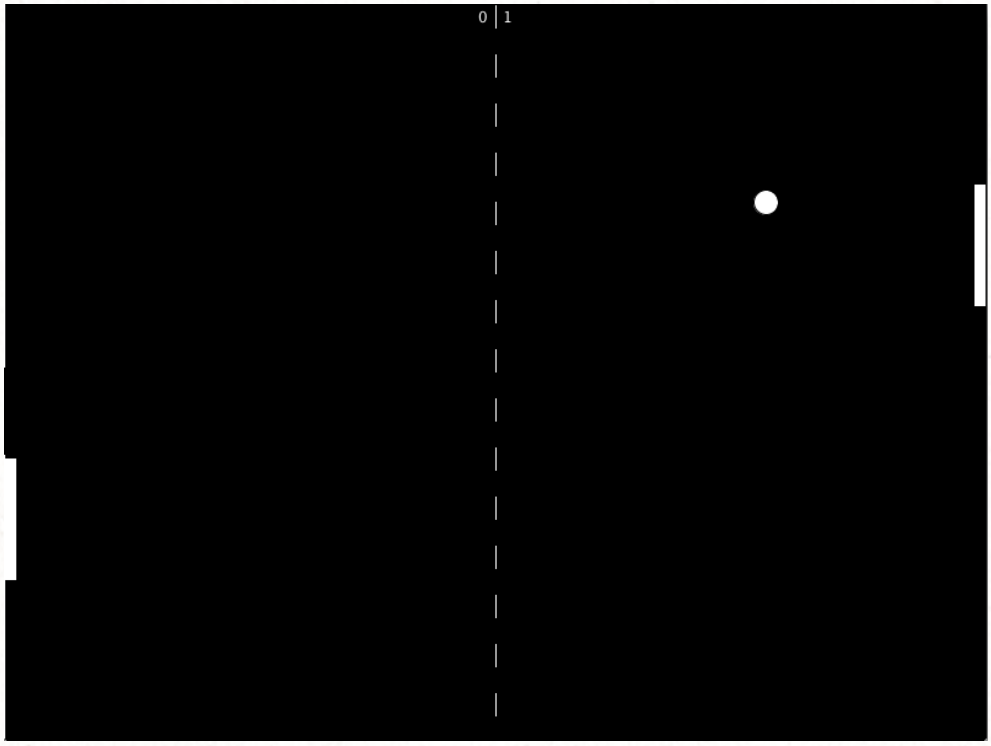
For more about the Platformer example, including some exercise tasks:

http://www.hobbygamedev.com/int/platformer-game-source-in-processing/

The Plane and Platformer examples include demonstrations of many concepts that are not represented in the other game source examples provided, including particle effects, scrolling camera, and an on-screen level timer. Borrowing source from these two advanced examples to mix into the classic game examples or Space Battle code can be a great exercise in becoming more comfortable with the code for each feature.

The `bonusPrototype` code folder is different from anything else included, in that it specifically was not authored as an educational demonstration. It's a raw prototype that I hacked together to explore with a gameplay concept. In addition to wanting to share an example of how hacky prototype code gets, even for experienced developers, I also want to invite you to explore on your own how the prototype might be evolved into a game. For now it's just a system to toy with, however with some resource constraints, initial configuration options, and modes it could be crafted to have a healthy mix of tactical and surprise elements. Two versions are included: one that is pixels only, and another featuring expressive faces that move about, interfering with the battle lines.

# 2 Tennis Game




## 2.1 Warm-Up Exercises

### 2.1.1 Triple the height of the player paddles

Numbers are one of the easiest things to change in game code. They often don't even require a full understanding of the logic, so long as they're well labeled and not changed too dramatically. Fortunately, code is full of numbers.

Figure out how to make the paddles three times as tall, so that it's much harder to get the ball past either side. The increase in size should affect both their appearance and their collision area.

**Details:**

The variable you're looking for is one with a name that is in ALL_CAPS. By historical convention[1] variables named in all caps are used for tuning variables, set only once in the program code, in contrast to gameplay variables that are expected to change during play.

[1] In the old C programming language `#define` values, typically written in ALL_CAPS, are, among other uses, a handy way to pull constant values out of the code, centralizing and labeling them.

Now that you know how to make a super large paddle, why not try a super small one? How small or large can you make the paddle and still win? If you'd like to restore the paddles to their original height before contining, I set them at 100.0, although there's really nothing magic or ideal about my answer. Leave it set to any height value you prefer.

On that note: it's worth noting before continuing that it's smart to have a recovery plan and simple backup habit[2] in case you get stuck somehow midway through a change that isn't working out. For getting back to how the project started, you can always just unzip the projects back out of the original zip file. Where backup matters more is when the knot happens with a project you've been making personalized changes to, completing several exercises in addition to introducing your own tweaks. If it gets in a mess, you'll want to have at least a working version of source from the night before to fall back to.

One easy way to locally save a backup is to compress the sketch folder that you're working on into a zip file. It quickly copies and archives a snapshot of your work. For an added bit of security, consider periodically copying the most recent zip file to a cloud folder like Dropbox or Google Drive. Even simply e-mailing yourself the file as an attachment is better than nothing. Good programmers keep regular backups, and not only on the same local machine!

### 2.1.2 Tune the score required to win

We're still starting pretty slow and easy here. Don't worry though - it'll pick up soon! For now, see whether you can find where to change how many points are needed to win. What goal value feels right to you?

**Details:**

Even though changing the right value in code from 3 to 40, or to 1, is a simple change to make, much like the paddle height it can have a big effect on how the game plays. Setting the goal to 1 makes the game end too soon, and with only 1 goal to win the role of luck may overwhelm skill. Setting it to 15 might fatigue or bore players. That doesn't mean 3 is necessarily the best number though, it's worth experimenting with. Or maybe 3 is the best number to certain kinds of players, people with a certain attitude about competition, while others think of 3 as too fleeting, or taking too long.

What I want to call attention to, here in the beginning, is how even though game design (deciding which number to play to) and programming (making it so) aren't totally the same, the skills are closely related, and worth practicing together.

[2] Recovery plan here is purely keeping in mind the risk of potentially losing work on a Processing project, if it gets into a tangle that's hard to get back out of, and there's no recently saved back up copy of it. There's no real risk from Processing, nor the kind of code in these packs, relating to any other kind of recovery or data loss. Worst-case scenario an infinite loop (badly written `if` or `while`) could stall Processing, but `Ctrl+Alt+Del` on Windows or `Apple->Force Quit` on Mac could then be used to exit Processing before simply reopening it.

Changing that number involves game programming. That is simple programming, sure, but it still involves being set up to run the code, getting into the source, and changing it. If, as a matter of game design, you decide that instead of making the game end after something like 3 points or 5 points, you want to try some different metric like declaring the winner as whoever scores the most points in 2 minutes, that's still game design, but testing that will involve a little more programming than just finding and changing a single number. Becoming more comfortable with game programming opens up new options to test and explore your game design ideas. Meanwhile, paying attention to game design questions and considerations can help you make more resourceful choices and tradeoffs in how you invest your programming time!

### 2.1.3 Invisible player

One of the things you may have already noticed in the code are the comments. Comments are any text to the right of `//` marks on the same line. That's a signal to the programming environment to ignore the text instead of trying to handle it as code. Programmers use comments to leave notes, so that in the future we (or someone else) can more easily make sense of what we were thinking when we wrote some funky code. Other times we use the `//` comment marks left of one or more lines of code to tell the computer to temporarily skip the line(s) so we can check what happens when the rest of the program runs.

Look for where in this game's source code the left player's paddle gets drawn to the screen. Add a `//` comment mark at the start of that line (or multiple lines if multiple lines are involved) to temporarily prevent the paddle from being displayed, then try playing without it. Once done, uncomment that line and test to verify that you've brought the paddle's appearance back.

**Details:**

Were you able to find the right line to comment out? (Answer in footnote:[3]) If so, what were the clues that tipped you off that you were headed in the right direction? If it took a few tries to find it, don't sweat it. Videogame programming is full of experimenting, making and fixing little errors, and trying things as we go. It's not at all a bad sign when something doesn't work right the first time, or when the compiler throws an error that needs to be traced out and resolved. That's just a normal part of the programming process.

What did you observe while playing with the invisible paddle? Perhaps you noticed that even though the paddle wasn't drawn to the screen, you could still play the game by moving the mouse, trying to mentally picture where the paddle would belong, the center due left of your mouse movements. That works as long as what you commented out was only the draw code. If you instead commented out the call to `moveAndDrawPlayerPaddle()` in `draw()`,you may have seen the ball occasionally returned by the left but couldn't control the invisible paddle, which is

[3] In the `moveAndDrawPlayerPaddle()` function, put the `//` marks to the left of its second line: `rect(0,paddle1Y,10,PADDLE_HEIGHT);`

because in addition to drawing the image that function call is also where the left paddle's position gets update based on the mouse.

What I'm aiming to illustrate here is that what's happening on the screen and what's happening in the code don't have to line up. An invisible paddle can still return the ball if the math is right. For that matter, a visible paddle can let the ball right through, given some small changes to the code elsewhere. For fairness we often want what's shown on the screen to be a true and accurate representation of what's happening in the game's logic. However as game developers we're often fiddling with a work-in-progress, in which case there's no guarantee that what we see and what's in the code are appropriately matched up. When the two go out of sync, there are various programming practices for navigating the more abstract types of errors: `println()` statements to trace code execution flow or output values, selective commenting like we've just tried here, calling upon `Edit->Auto-Format` in Processing's menus to help fix our indentation while searching for signs of mismatching braces, and so on. But one of our best tools for debugging issues in real-time game programming is finding ways to do the opposite of what happened here: rather than having the visible go invisible, find ways to make the invisible (like which unit a robot is targeting) visible while testing and developing (by drawing an extra line that the player would never need nor want in the final version, for example). These temporary development features can then be toggled in your code, so that you can put the program into debugging mode for development, or switch it to release mode before sharing the game[4].

One aspect you might have observed in trying to play with an invisible paddle is that not only was it hard to block the ball, but it was even harder to predictably aim the ball. (Did you know that in most Tennis Style games the ball can be aimed? Many modern players don't know this!) Let's dig more into that for our next exercise, in which we're also going to be picking up the pace...

### 2.1.4 Try implementing reflect-style ball behavior

One of the important gameplay features of a classic ball-and-paddle style digital tennis game is the ability to aim the ball based on which part of the paddle blocks it. Specifically: when the ball collides with the center of a paddle, it returns in a flat horizontal line, which is in turn easier for the other player to block or carefully aim it. As the ball hits further from the paddle's center it takes on a steeper angle, in proportion to how far from the center it hit. A little above the middle shoots the ball upward a bit, far above the middle directs it steeply upward, a little below the middle sends it down a bit, and so on.

From a game design perspective this creates a compelling risk-reward trade-off. Players are incentivized to risk missing the ball by hitting nearer to the edge of their paddle, since doing so makes a steeper, faster hit, harder for the opponent to block. Without this feature players would always try to center the paddle at where

[4] You can use Processing's `File->Export Application` menu feature to compile the game's code into a standalone program, for easier distribution to people without Processing installed!

they expect the ball to cross the line, to leave more margin for error in estimation or motor movement. With this feature, using that cautious technique gives the opponent the easiest possible shot to block or aim. Instead, skilled players usually try to hit the ball near their paddle's edge, increasing the chances that even an experienced player will miss the ball.

Fortunately, this feature is implemented already in the example project, as provided. An error some beginning game developers make when trying to create this style of game is to overlook the whole ball control dynamic. Instead they'll implement simple ball reflection, such that no matter which part of the paddle gets hit it bounces off as though the whole player's side were another flat wall.

To compare for yourself the difference in gameplay that results from that error, can you figure out how to modify this game's code to use simple ball-reflection behavior rather than the ball control feature?

Since we may not want to keep it this way as we move on to other exercises, adding a toggle value in your code as a new `boolean` (a variable set to `true` or `false`) or an `int` (here set to `0` or `1`), or simply figuring out a section to comment or uncomment, to switch between these two styles may be handy.

**Details:**

Find the four lines where the `deltaY` variable appears in code. As implemented in the code provided, vertical speed gets recalculated upon the ball's paddle collision. Instead, to make the ball reflect, all we need to do is skip the code that handles the calculation. Then vertical speed is preserved.[5]

We could achieve this effect by simply commenting out the lines that deal with `deltaY`, in both places to affect collision with both paddles. However what we'd like to do instead of commenting it out this time is declare a new `boolean` value at the top of the file that we can then use to manually turn on or off ball control versus simple reflection play. We should name that `boolean` value something based on its usage - `reflectInsteadOfAim` will do quite nicely. Now we just need to check using `if` whether it's `false` to decide whether to allow code flow to reach the `deltaY` lines in the event of paddle collision.

Depending on the starting vertical speed of the ball and how quickly the computer paddle can move, it may now be impossible to score! Initialize that new `boolean` back to `false` before we proceed so the game will again be possible to win by changing the ball's vertical speed and/or aiming away from the opposing paddle's position during play.

[5] Reminder: unlike in geometry or algebra math classes, in 2D computer game development and other technical 2D graphics programming, the y axis is often downward positive instead of upward positive. `(0,0)` is the left-top, `(width-1,height-1)` is the right-bottom. So a positive Y speed means downward movement, negative Y speed means upward. This has to do with old technical conventions, based on how video memory gets mapped to the screen.

## 2.2 Practice Exercises

### 2.2.1 Ball-speed increase as it stays in play

The ball speed is currently set quite fast. It starts fast, stays fast, and ends fast, without ever changing. While this tuning can be helpful to prevent a match for going on forever, it can be especially frustrating for a new player since it can be challenging to return even the first volley. One common pattern for solving this kind of tuning issue in videogame design is to have the difficulty start easier, then progressively get more intense as play continues. Create a way to have the ball's speed increase during the match, such that it begins with slow movement at the start of each serve, gradually picking up speed (up to some practical maximum limit) based on either the time passed or number of consecutive hits.

**Details:**

Take a look at `ballSpeedX`. Its initial value is currently all we need to notice about it, since it stays the same throughout the program, except for being made negative when traveling left. `ballSpeedY` is the variable for vertical ball speed, though for that one, changing its initial value won't change much since that only affects serves, after which `ballSpeedY` gets set each time it strikes a paddle. To have vertical speed increase too we'll need to fiddle with where `ballSpeedY` gets set upon hitting the paddle.

As an initial step, it'll help to first set up a way to get the ball to move faster or slower, even if not changing it yet during play, by creating and setting a single new `float` value in code to represent the ball's overall speed multiplier. Increment that number during play, up to some maximum limit, in a section of code that you know will be called regularly during the round: each time a paddle hits it, every draw frame, etc. Remember to reset the number back to its initial value where the ball gets scored on either side. Remember that `ballSpeedX` is negative when going left and positive when going right, and the sign on `ballSpeedY` flips when going up instead of down - part of why we're creating an additional variable for increasing speed is that if we tried to change `ballSpeedX` directly we'd need to treat it differently based on which direction it's moving it (put another way: checking against negative maximum if less than zero, i.e. moving left, as opposed to greater than zero, i.e. moving right).

Tuning an increase amount to be detectable but reasonable may take some experimentation - if too small or too large the effect may not be apparent, or may seem to cause the ball to instantly vanish, sending it flying off screen in a single frame of motion. Consider the screen's pixel dimensions, the ball's starting speed, and the frequency of where your speed-up code gets called - that means only once every few seconds if sped up on hit, but around thirty times every second if done directly in your `draw()` update cycle!

Are there reasons why you prefer the ball speeding up only on contact with a paddle, rather than constantly throughout the ball's movement during play? What

are some justifications for handling the ball speed increase one way instead of the other?

### 2.2.2 Graphics

The game is currently just white shapes. Make and load image files for the paddles, ball, and background (net and playfield) instead of using the solid shape drawing code.

**Details:**

You'll need to create new `PImage` variable for each image you'd like to load. In addition to defining the label of that type, you'll need to load the corresponding image file with `loadImage("filename.png")` - typically done in `setup()` since we only need to load the files into memory once each - in addition to adding a line of `image()` code to draw the corresponding image in the appropriate location.

To figure out locations, look for where rectangles and ellipses are currently drawn in the code. It may be handy to temporarily draw both the old rectangle or ellipse and the new image to ensure that the two are overlapping as expected in terms of center point, shape edges, and overall dimensions.

A single `PImage` can be reused arbitrarily many times in the application - there's no need to load separate images, nor the same image multiple times, if you want both paddles look the same, for example.

To avoid having a white square behind the ball you'll need to use transparency when authoring the ball image, which works differently than any painted color but is only supported by certain formats[6]. I recommend saving the ball file as `png`, which supports smoothed, partial transparency. Photoshop or the free alternative from `GIMP.org` can be used to author image files with transparent backgrounds. Search the web for `transparent background` and the name of your image software to learn how to use transparency and layers[7].

If instead of authoring images you choose to find and use images off the Internet, odds are good they will not already be the perfect sizes needed for your program. Although there is a way for the image to be resized in code, it often works out better for measurements, positioning, and file size to instead resize and save over the image using an image-editing program instead.

Be aware that whereas the built-in `ellipse()` function used for the ball automatically centers on the x,y coordinate pair provided, the built-in `rect()` function drawing the paddles treats the x,y coordinate as the top-left of the rectangles. Using `image()` to display graphics also uses the x,y as the top-left like

[6] If you're not seeting the .png or other file extensions, do a web search for how to set your computer to no longer hide extensions on known file types. Extensions on filenames are useful to distinguish different formats, and need to be known to import files in code.

[7] Speaking of image format and file extensions: Processing, and most programming environments, cannot directly import the Photoshop PSD files or `GIMP.org` xcf files. The image will need to be exported as png or one of the other formats supported by Processing.

`rect()`. Therefore although `image()` can use the same draw coordinate as `rect()`, for drawing the ball image its coordinates ought to be offset by half the width and the half height of the image/ball to keep the image centered on the location used in collision logic. Otherwise we'll experience the sort of bug mentioned in the Invisible Player warm-up exercise: a mismatch between what the player sees compared to what's happening in the code's movement logic.

For a background image, it will help to make it the same dimensions as the game's window. The windows size can be found as the arguments[8] to `size()` in `setup()`. When drawing the image remember to replace both the `fill()` calls at the top of `draw()`, the `rect()` near there which wipes the screen, and the call to `drawNet()` which was programmatically creating the dashed line in the center of the field. Don't accidentally draw the background image after the paddle and ball though - because `image()` calls overlap anything drawn before it and the background is as large as the whole window, getting the draw order backward could make everything else appear invisible by blocking view of it before the screen updates at the end of `draw()`.

### 2.2.3 Move player paddles away from sides

The paddles are currently flat against each edge of the screen, so that there's no margin between the player and goal zone. This makes for simpler programming, since in combination with the paddles being very thin it means that we can treat the ball crossing the screen edge as the only time we need check for paddle collision. The downside of this is that it means there's no time after players miss the ball in which they can still see the ball moving behind them. That's potentially a meaningful moment in the play: it gives players more opportunity to make sense of what just happened. It also adds a moment of internal celebration or regret for players, when the consequence of a miss can be witnessed even after it's too late to do anything about it.

For this exercise, modify the code so that the paddles are noticeably separated from the edge. What distance they are from each edge is up to you. For ease of tuning to find an ideal position, it would be preferable to connect paddle distance from the edge to a new tuning value at the top of your code.

**Details:**

Drawing the paddles further from each edge isn't the difficult part - for that just adding an offset to the `x` position of the left paddle and subtracting that same value from the `x` position of the right paddle will get the job done. The challenge here is in figuring out how to change the paddle collision code so that the ball can move while behind the paddles, but still responds appropriately when overlapping the paddle. Look in `boundsAndPaddleCheck()` - in the provided implementation

[8] To clarify a question I got from one of my younger students: `size(800,600);` means 800 pixels wide and 600 pixels tall, *not* eight hundred thousand six hundred pixels. In programming we don't put commas in our large numbers - commas are used to separate values.

paddle collision is only checked in the condition that the ball is off the right or left edges of the screen. In order to properly handle collision with a paddle further from the edge it will be necessary to decouple paddle collision from the edge check, treating the ball going past either edge as always scoring a point and resetting the ball. Then in a separate set of `if` conditionals in that `boundsAndPaddleCheck()` function you'll need to check whether the ball is not only below the top and above the bottom of your paddle, but also whether the ball is right of the left edge and left of the right edge.

There are a couple of tricky cases to watch out and account for. The first is that depending on your paddle's thickness and the ball's maximum speed, it's possible for the ball to tunnel through the paddle by crossing in a single position update from one side of the paddle right over to the other, from one frame to the next, without ever overlapping it. This can be accounted for by keeping track of which side of each paddle the ball is on at the end of each frame, and if which side of either paddle the ball is on changes, do the appropriate calculations then to account for collision checking.

The other source of complication is that because the ball's horizontal speed gets flipped horizontally upon paddle collision, there may be cases where the ball gets briefly stuck zigzagging within the paddle before escaping from either edge. There are a few ways to work around this issue, the most common being to check the ball's horizontal direction as part of the collision check (ex. only bouncing off the left paddle if the ball is currently moving left) or to use absolute value of the speed with or without a negative in front to force a sign on the return (so the ball can only go rightward from colliding with the left paddle, and leftward upon colliding with the right paddle).

### 2.2.4 Add sound events and effects

Currently this game is silent. One of the most significant differences between commercially successful early tennis-style games and their unsuccessful counterparts was the presence of sound effects to provide feedback on player paddle hits. Especially in combination with the ball speeding up during play, as is covered in an earlier exercise for this project, the presence of sounds for each paddle hit also helps make clearer the shift in speed and pace as the match progresses.

**Details:**

There are essentially three parts to this exercise: creating (or finding and editing) the sound effect files, writing code to load and play those sound effect files, and then putting that sound playing line in the right areas of code to trigger upon the appropriate events.

For creating or editing sound effects, I recommend a free program called Audacity. Search the web for its title to find an official download of it. It's an audio editing program available for Windows, Mac, and Linux. While it's by no means a total replacement for high-end audio software used by specialists, it covers much of the functionality that we'll need for small-scale projects. It can be used to

trim silence, fade in or out, adjust overall volume or pitch, and so on. It can even be used to generate simple 1970's-style digital tones, which for a classic game is often just the sort of sound that the context calls for. For an easier way to generate slightly more modern tones (1980's sophistication) search the web for as3sfxr, which can be used to quickly and easily create a wide assortment of retro sounds.

To load and play the sounds, the Minim library will be needed. It comes with Processing but isn't included or initialized in projects by default, so it will need to be imported and instantiated. For an example of how to use it, look in the Bomb Dropper project (another game example in this package), searching the project's source code for Minim. In addition to the `import` line, an instance of Minim needs to be declared at a global scope and initialized to a `new` instance in `setup()`. Each sound file should be in the same directory as the project's pde files, or in a `data` subfolder within the project's directory. Sound files are handled in code using `AudioSample` much in the same way that images are treated as `PImage`: declare the variable's label, load it from the filename, then use it (by calling `.trigger()` on the `AudioSample` instance) where appropriate.

Figuring out where to trigger the sound file is a matter of reasoning through which parts of the code are getting accessed when the event in question happens. To attach a sound to a scoring event, look for where in the code the score gets changed, and trigger the corresponding sound there. To attach a sound to where the ball is reflected from a paddle or the wall, look for where in an `if` the ball's horizontal or vertical speed gets flipped, and so on.

### 2.2.5 Two player mode option (keyboard controls)

The game is currently only playable against a simple computer opponent controlling Player 2. Classic tennis-style games often emphasized human-vs.-human play, which here will require changing control of the second paddle from the simple AI routine to another form of input. While there are ways to get a computer to handle the input of multiple mouse devices, it's rare for players to have multiple hardware mice on hand, and so in the interest of making the game playable on standard hardware setups we'll use this opportunity to take on the challenge of using keyboard input to function as a pseudo-analog (analog meaning support for faster/slower action, rather than digital on/off like a button) input.

Can you find a way to use the keyboard to control the Player 2 right side paddle in such a way that it feels like it has a fair level of responsiveness and precision in comparison to Player 1's left paddle mouse input?

**Details:**

Here is an exercise where the basic solution, simply getting keyboard input to move the Player 2 paddle up and down, isn't especially complex. The challenge is buried in getting this done in a way that feels and plays right.

To handle keyboard input at all it will be necessary to write two new functions that, like `draw()` and `setup()`, Processing looks for an automatically connects particular functionality to: `keyPressed()` and `keyReleased()`. The former gets

called when any key goes down, and the latter when any key goes back up. The trick to detect particular keys is to use conditionals to check the value of `keyCode` against directional inputs or `key` against alphanumeric inputs. For example, within `keyPressed()` the comparison `if(keyCode == DOWN)` would identify whether the down arrow was just pressed, whereas `if(key == 'j')` can be used to determine whether the J key was just pressed.

For examples of this kind of key handling, see the `commonBonus/mini-demos/arrowKeys` code for a simple demonstration, `commonBonus/free-from-hobbygamedev.com/plane-src-examples-all/plane_7_plane` for keyboard code mixed into the main source file, or the Racing game (also in this package) for how to set up key holding logic in a separate `keyHandling` file/tab within your project. The gist: keep a `boolean` (`true` or `false`) value for each input you wish to handle, set that to true in `keyPressed()` and false in `keyReleased()`, then in the game logic check the state of that `boolean` variable rather than making changes directly upon key pressed or released. This may seem roundabout but is a really useful concept to treat keys more like controller buttons, which can be held down, rather than as typing inputs which signal either on the moment it's pressed or at some system-dependent interval for automatic key repeat. After disconnecting the AI logic from paddle 2's movements in `moveAndDrawComputerPaddle()`, whenever the corresponding up or down input `boolean` value is `true`, decrease `paddle2Y`, whereas when the `boolean` value for the other direction is `true`, increase `paddle2Y`.

In order to get the input feeling and moving in a more natural way you'll likely need to experiment with having another `float` value to simulate momentum and friction in the movement. Instead of affecting `paddle2Y` directly while the up or down keys are held, instead increment or decrement that new momentum `float`. Then, every frame for the second paddle's movement add the momentum value to `paddle2Y` and somehow decrease the momentum value. One common way to accomplish that: just multiplying the `float` momentum by a value near though below `1.0`, such as `0.96`, means it will decay by 4% of its speed each frame so it glides to a smooth stop.

It can be very difficult to tune the acceleration and deceleration values in the above form of keyboard input to come near the level of control offered by a mouse. If holding the key moves the paddle too rapidly, it becomes too easy to accidentally overshoot the ball. If holding the key moves the paddle too slowly, it may become impossible to get the paddle to the intended location in time, which is an issue that never happens when playing with mouse input. An additional layer of input fanciness that could be used to partly address this limitation is to allow rapid tapping to give the paddle an extra acceleration boost, so that there becomes a different way to input rapid as opposed to smooth movements.

Having trouble making the game fair when played between two people in which one can use the mouse, and the other can use the keyboard? Perhaps consider changing how the left paddle gets controlled, so that rather than using the mouse for that player, different keys on the keyboard are used. With keyboard buttons used by both players, no matter how the input is tuned the match is at least a fair competition.

*(Historical note: the original tennis games in arcades and home consoles were not played with a mouse nor button inputs, but were controlled by rotating dials that raised and lowered each of the paddles on the screen. This was a unique form of input that's difficult to match precisely with either a keyboard or mouse, since unlike a keyboard it could be used to easily indicate movements of varying speed, but unlike a mouse it didn't have such a directly mapped spatial relationship between movements on the controller and movements on the screen.)*

## 2.3 Challenge Exercises

### 2.3.1 Ball trail using an array of past positions

As the ball speeds up during play it can become increasingly difficult for the player(s) to spot and track its motions. One way to alleviate this issue is to draw a trail behind the ball, which simultaneously gives the ball an overall larger visual signature and better represents the ball's current velocity.

In order to draw a trail behind the ball, keep track of some number of previous ball positions. For starters, let's track the past five coordinates, though ideally set up your functionality so that this can be varied easily in code. Draw something at each of those coordinates to visualize that data: ellipses of decreasing brightness, versions of the ball that are increasingly transparent, or for more of challenge, a polygonal trail connecting the points together with thickness decreasing toward the last position tracked.

**Details:**

Keeping an `ArrayList` of positions (`PVector` will do, or create your own pairing of `x` and `y` `float` values) will be the easiest way to make this code flexible for later changing how many trail points to track. An `ArrayList` is a generic container for managing a bunch of class instances, which is computer jargon for saying that if you want to manage in your code a group of enemies, a group of missiles, a group or coordinates, a group of bubbles, or a group of angry images that are treated somehow as a set, then you're probably going to want to get an `ArrayList` involved. Sometimes we take advantage of how an `ArrayList` keeps things in a stable order, as we would be wise to do here for coordinates in a trail, however in many other cases we don't actually care much what order things are in within an `ArrayList`, we just utilize it as a general purpose way to work with a whole set of similar things easily.

Each time the ball's position gets updated, immediately before changing the current coordinate shift all saved coordinates down in the `ArrayList`, then set the coordinates for the first trail entry to the ball's location prior to repositioning. To draw the trail points, in the function where the ball gets drawn, in the lines prior to displaying the ball's current location loop through the trail positions and draw whatever representation you prefer for each point. Remember to write the code in a way that handles the initial situation safely: Either start with all trail points

under the ball's centered position, or be careful to shuffle and draw only trail entries that exist by the time the trail handling code gets handled.

The algorithm described above, which involves iterating over each trail point to copy the coordinates from the entry ahead of it, requires hitting each trail entry every frame, even though really only two points are changing: the lead point gets added and the trail point gets removed, the others in the middle of the list are consistent from each frame to the next. There are a number of more clever optimizations that can be applied to this circumstance to make only minimal calculations or memory changes per frame, such as tracking a head index and cycling that while keeping other list elements in place, however for so few points and on a single object these optimizations will have no impact on game performance. If implementing trails for a fireworks-like or fluid particle system though, in which trails might be much longer and needed for hundreds or thousands of objects simultaneously, those kinds of improvements would need to be considered. For this kind of narrow context, though, optimizing instead for readability and programmer time often make more sense than trying to squeeze out a handful of less processor calculations per update. The difference for a short trail on a single object is only perhaps a few dozen operations, while a modern computer can handle literally *billions* of such operations per second.

If you'd like to get a polygonal trail working, you'll need to use `beginShape()`, `endShape()`, and many `vertex()` calls in-between to set each point. For finding the angle between any two points in the trail, the `atan2()` function will be helpful.

### 2.3.2 AI that accounts for reflections

The computer-controlled player code provided in this example project is very basic: If the ball is lower than it, it moves down, and if it ball is higher than it, it moves up. A more sophisticated and human-like AI would attempt to account for where the ball will cross its side, including anticipating how a wall reflection will affect its trajectory. Calculating the point of reflection (including whether it's on a trajectory to hit either the top or bottom wall before crossing the edge) and its eventual intercept with the edge will involve a little Algebra or Geometry. Of course, having passed those classes before and figuring out how to apply that math in a practical context here are two different things - good luck, and don't be afraid to break out graph paper and a calculator to help in thinking through example cases. As a reminder when searching for non-programming algebra/geometry references - this won't affect the math involved but might lead to some confusion otherwise - in game programming by convention positive y goes downward, with the origin (0,0) at the top-left of the screen, rather than positive y being upward as it is in general math contexts.

**Details:**

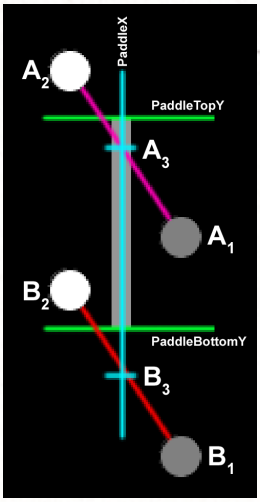
We keep track of the ball's velocity as a separate x and y component, in the form of `ballSpeedX` and `ballSpeedY`. Because slope, the m value in slope-intercept equations, is rise over run, dividing `ballSpeedY` by `ballSpeedX` yields a slope value that can be useful in calculating intercepts. First use slope intercept equations to

determine whether the ball is on a heading to cross the Player 2 edge above the top of the screen or below the bottom of the screen - and if so, that means the ball will be reflected and further calculation is necessary, otherwise the paddle can just move toward that calculated intercept. Remember to compute not for where the ball will cross the screen edge, but rather where it will cross the paddle's line, if you've followed the above exercise to move the paddles away from the edges. Also recall that centering on the ball makes a straight shot back - move aside a bit!

As for when the ball is moving away from the AI player: moving smoothly back toward the center will help it appear more human. One might think that the paddle should simply try to account for where the ball will be several reflections later, including bouncing off the player's paddle, but remember that because ball control is affected in this tennis game version as a function of which part of the paddle the ball strikes, the destination intercept cannot be figured out until after the ball has been struck by the player's paddle.

### 2.3.3 Calculate intersection/intercept when crossing paddle line

As addressed in the earlier exercise about moving the paddles away from the edges of the screen, tunneling is a problem that happens when the ball moves in a single frame from one side of the paddle to the other. At very low speeds this problem doesn't come up, but as the ball speeds up during play it can become a very real issue when using thin paddles and a small ball. The solution described earlier, keeping track of which side of each paddle the ball is on then checking vertical coordinates for collision detection any time the ball switches sides, works quite well but can break down when the ball is moving quickly at a steep angle. Consider the two cases illustrated here, a shot A on the top that should reflect (but would be counted as a miss) and a shot B on the bottom that should be a miss (but would be counted as a hit):




In this illustration, the ball in the top motion A is at position $A_1$ in one cycle and at $A_2$ in the next, which would report

incorrectly as missing the paddle due to being above the paddle in the first position that's across the paddle's horizontal line. Meanwhile the ball following the bottom trajectory B is at position $B_1$ in one cycle and at $B_2$ in the next, giving another false report, but in this case that the ball hit the paddle due to being vertically between the top and bottom of the paddle in its first position across the paddle's line. To handle this case correctly we need to calculate $A_3$ or $B_3$ respectively: the first frame that the ball crosses from in front of the paddle to behind it, we need to then figure out a straight line equation for the ball's trajectory and calculating the value (read: vertical position) at f(x) where x is the paddle's horizontal coordinate.

In the fix for tunneling described earlier, the height at positions $A_2$ and $B_2$ would be checked against the paddle top and bottom edge, since that would be the first position in code on the opposite side of the paddle. The proper fix would calculate for $A_3$ and $B_3$ in the diagram, on the first frame when the ball crosses the paddle's line, then determining based on the vertical position of that intercept point whether the ball hit the paddle (in addition to where along the paddle, for the purpose of determining ball return angle).

This may seem like a pretty silly inconvenience for such a simple game, but whether the ball hits or misses a paddle is the central action of this game: all player attention is on it, wins and losses are decided upon it, and if it's done in a sloppy manner that fails to handle steep shots properly astute players *really will* notice and feel shortchanged by the game "cheating." It's worth figuring out how to get this right!

**Details:**
Look back at the intercept code written for the computer-controlled paddle in the previous step. This solution is very similar. Again you'll divide the velocity components to determine the slope of the ball's motion, and then use that slope the find an intercept where the paddle is positioned horizontally.

## 2.4 Write From Scratch Steps: Tennis Game

Want to recreate this entire game example beginning from a blank file? Test your comfort with the concepts involved by seeing whether you're able to follow the steps below to reconstruct the starter code from nothing.

1. Create a new, empty Processing Sketch.

2. Initialize window size to 800x600.

3. Draw a white ellipse anywhere on the screen, 20 pixels wide and tall. This will be the ball.

4. Add variables to keep track of a ball's horizontal and vertical position, and change the ellipse code to draw the ellipse at that coordinate.

5. Add two to the ball's horizontal position every draw cycle, so that it will appear to move when you press Play in the Processing IDE. (Note: at this time it may leave a smudge trail since the background is not being redrawn yet to erase its previously drawn positions.)

6. To eliminate the smudge trail, clear or wipe the screen each time by filling the screen with black at the start of every `draw()` cycle.[9] Now when pressing play no trail should be left behind and the ball should just appear to move in a straight path.

7. Rather than having the ball move at a constant, hard-coded two pixels per frame, let's create another variable that we can use as the ball's horizontal speed at any given moment. Though initially set to two, by instead using that variable to change the horizontal position each frame we gain the power to reverse the ball's direction at any time by multiplying the value by -1.

8. Presently the ball will go off the right side of the screen if given time to do so. Every frame, check whether the ball has gone past the right edge, and if it has reverse its direction by multiplying that horizontal speed value by -1.

9. Repeat the previous step, though for making the ball bounce off the left edge of the screen. Now when the application is run the ball should bounce back and forth between the sides of the screen.

10. Repeat the past three steps for vertical movement, so that the ball moves at a constant speed diagonally, bouncing not only off the sides but also the top and bottom of the screen.

11. Add a white rectangle along the left edge of the screen that is 10 pixels wide and 100 pixels tall. This will become the player's paddle.

12. Introduce a new variable to keep track of the player's paddle vertical position. Change the paddle rectangle's draw code to be positioned according to the value of that variable. Each frame update that variable based on the mouse's vertical position so that the player's paddle rectangle is centered on the mouse, half above the mouse and half below (horizontal position should stay locked along the left edge, no connection to mouse input).

[9] We only ever see how the screen buffer looks at the end of the `draw()` function - so if we flooded the screen with black there instead of at the beginning of the function we'd never see anything else.

13. Let's now make the ball reset if it misses the player's paddle while passing the left edge. In the code where the ball is detected as past the left edge, add an additional conditional to determine whether the ball's vertical position variable is below the top of the paddle and above the bottom of the paddle (for now ignore the thickness of the paddle, we'll only consider when the ball is crossing the edge of the screen). When the ball's vertical position does fit between the paddle edges, let it bounce horizontally as it previously did for any part of the edge. However when the ball's vertical position is either below the bottom or above the paddle's top, instead of reflecting motion reset the ball to the center of the screen by setting the horizontal and vertical position to half the screen's width and half the screen's height.

14. Draw another white rectangle on the right side of the screen as the same dimensions and style as the player's. This will be the computer's player's paddle.

15. As with the player's paddle, create a variable for setting the right paddle's vertical position, and use it in the rectangle draw to affect where the paddle gets shown.

16. Before automating the right paddle's movement to automatic/AI control, let's first connect its vertical movement variable to the mouse vertical position so that both paddles move at the same time to the same height. This will be helpful in checking whether the right side's paddle's collision code works right, giving us control over whether it is in position to hit or miss at any given time.

17. Just as we had the ball reset if it goes off the left edge where the player's paddle isn't covering, match implementing that behavior for the right side of the screen in connection to the second player paddle's vertical position. In other words: When crossing the right edge check whether the ball's vertical position is within the top and bottom edges of the right paddle, and if so flip its motion value horizontally by setting it negative, otherwise reset the ball's position back to the center.

18. Add text to the screen. Any text in any position we can see easily will do for now, for example we can just show the word "stuff" at 100 pixels from the left and top edge.

19. Add a new variable to keep track of the left player's score. Start the value at zero.

20. Change the text line to show that score value rather than a hardcoded word.

21. In the code where the ball resets for passing the right edge of the screen above or below the paddle, increment the left player's score. Now letting the ball pass the right paddle should show the score increasing in-game.

21. Add another text element on the opposite side of the screen, another variable for Player 2's score, and match the Player 1 scoring steps to increment Player 2's score whenever the ball resets from passing the left edge.

22. Our current collision action between the paddle and ball is causing a simple reflection, meaning that there is no player ball control besides blocking it from passing. If you've already been through the exercises for this project, you may recognize this behavior from one of the warm-up questions. With this functionality there is no incentive to chance risky hits near the paddle edge rather than conservatively always centering the paddle near the ball's incoming position. To increase player agency by providing an element of control, while also adding excitement by incentivizing risky play, set up the ball's vertical speed upon paddle collision to be proportional to the distance from the center of the paddle. This means that if the ball hits the center of the paddle return it straight horizontally, but the further the ball gets above the center the more steeply upward the ball should return and the further below the center the more steeply downward the ball should return. Keep horizontal speed the same independent of the vertical speed, so that in addition to being a harder to anticipate shot the steeper returns will also be moving faster overall. This behavior should work the same for both paddles, but be sure to compute the difference from paddle center from the correct paddle's height (while both are connected to mouse movement making this error would be hard to catch!).

23. The paddle on the right should be computer controlled rather than following the mouse. Disconnect the right paddle's movement from the mouse's y coordinate, and instead have the right paddle move up if the ball is above it, and down if the ball is below it. Setting a modest limit on its movement speed is important in ensuring that the player has a chance to get the ball past it - experiment with tuning that number until it's possible to get the ball past the AI paddle, but not too easy to do consistently.

24. As designed so far, the game is endless. Come up with a maximum number of points to play to - for example, I used three for the example code provided - and when either score reaches that value, pause gameplay to instead display a message indicating which player won. Upon the next mouse click when that message is being displayed both player scores should reset and the ball should be returned to the center of the level.

25. For a bit of visual decoration, add a "net" to the middle of the playfield. Draw a dashed line from the top of the screen all the way to the bottom, two pixels thick, solid white for 20 out of every 40 pixels. For my implementation I

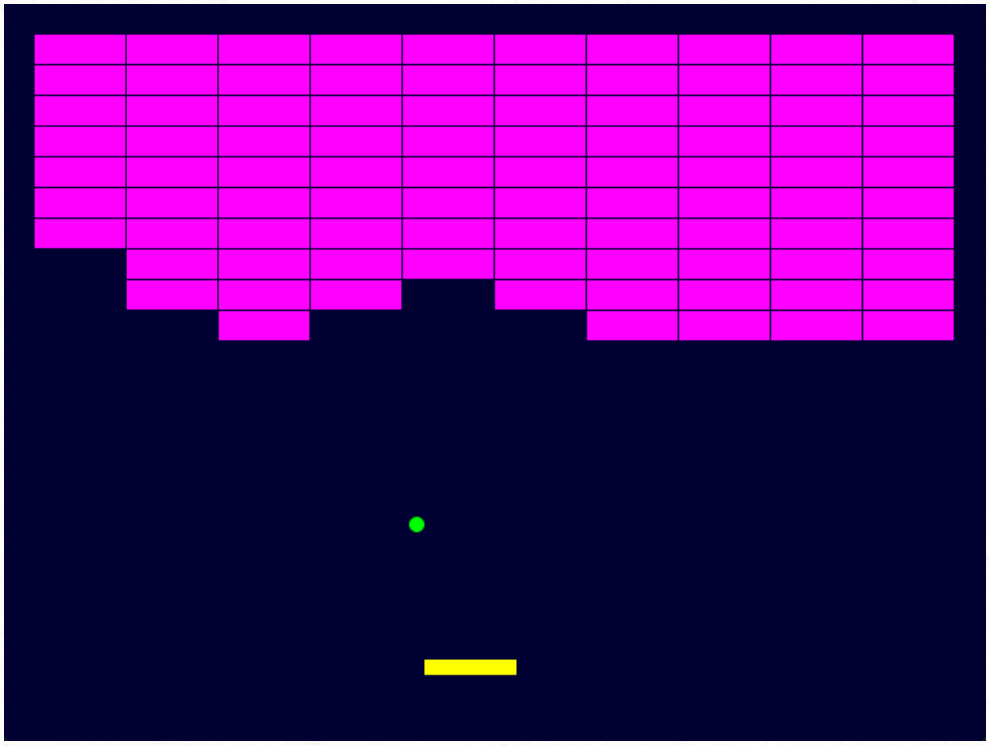
used a `for` loop and a `rect()` call to accomplish this, however there are certainly other ways to do it.

26. Give the game one last tuning and layout pass. Is the effect on vertical ball speed from hitting the ball near a paddle edge too exaggerated, or not significant enough to be worth the risk? Are the scores positioned in a way that makes clear which side has how many points, and they're out of the way of most action though easy enough to spot (consider experimenting with `textAlign()` to center the score displays!)? Are the ball's speed and AI paddle's speed both tuned in a way that makes for a play experience that is neither frustratingly difficult nor boring from being too easy?

27. If you didn't do so while writing this program, it's definitely worth going back while the implementation is still fresh on your mind to do a bit of clean-up and commenting. Perhaps a variable or function name has fallen out of sync with how it's being used, in which case go ahead and rename it in all relevant locations. Any place in your code where something kind of questionable or non-obvious in the approach should be given a line or two of commenting so it will be easier to make sense out of later.

28. You've done it! You've completely rewritten the example code that I included for this game type. Congratulations! If you haven't already been through all of the exercises on the source provided, you are now ready to give them a go on the code that you've written here as a foundation. Even better: If you have been through the exercises on the provided code, consider trying them here again without referring to the hints, or perhaps even without the detailed descriptions, working instead from this PDF's table of contents as a list of brief challenges to tackle.

# 3 Brick Break



## 3.1 Warm-Up Exercises

*Need some help? Stuck or interested in comparing your answers to how I'd accomplish the task described? Check out the code in* `classic-games/example-solutions/brickWarmUpExercises` *for example solutions to the next 9 warm-up exercises. For a video walkthrough of me going through coding these, see:* http://www.youtube.com/watch?v=0qN-7R9C_00 *Remember though that example solutions are* not *available for most exercises designed for the other projects. You'll learn most by finding ways to program the features that you find compelling, and taking the game in a personalized direction that interests you.*

### 3.1.1 Code organization: multiple files

Currently all of the code in this example exists in the same .pde source file. Divide it up into multiple files, each containing only the code and variables for the paddle, ball, or bricks, respectively. For functions or variables that relate to multiple objects (collisions, for example) feel free to keep it in the main file, or in the source file for either of the objects involved in the collision.

**Details:**

In the Processing development environment creating a new file is equivalent to creating a new tab in the project. There's a white circle with a downward facing arrow next to the file's tab - clicking on that and selecting New Tab will present a prompt below to name the new file. Name it, click OK, and then begin using cut and paste[10] to migrate code into the other tab(s). If you decide after naming the tab that you'd like to rename it, the same downward arrow used to create a new tab can be used to rename the one that's currently open. Alternatively, another way to rename a tab is to exit Processing and rename the .pde file. The name of the sketch directory determines which .pde file is the primary one.

### 3.1.2 Keep and display score

The project in its current state doesn't yet keep score. We can later make score more involved, with varying brick types or power-ups with bonuses, but for now earning 100 points per brick removed will be a good starting point. Until we implement lives, we'll just reset the score each time the ball resets.

**Details:**

Although one approach that likely comes to mind is to somehow connect the score to the `bricksLeftInGame` variable, there's a problem with doing so: score should continue increasing when the bricks reset after being cleared. Instead, create another global[11] `int` value for keeping score, to be incremented where the brick gets removed from ball collision. The score can be set to zero in `ballReset()`, and displayed on the screen in the `draw()` function with a `text()` call to show the score value someplace on the screen.

Even though this feature doesn't take much to get it working, it's one of these cases where just a little bit of code can have a big effect on gameplay. Before this, missing the ball didn't matter much. With a score, and with that score being lost when the ball is missed, saving the ball takes on increased significance the longer the player keeps the ball in play. Saving the ball is now the same as saving the score, making the player less willing to risk returning the ball near the paddle edge to achieve desired shot angles.

[10] Select the text to move, then in the menus choose `Edit`, `Cut` followed by moving the cursor to its new desination and using `Edit`, `Paste`. On Mac `Command+X` and `Command+V` will cut and paste, on Windows `Ctrl+X` and `Ctrl+V` are the shortcuts to do it.

[11] Global just means at a scope outside of any functions or classes, making it accessible to anywhere in the code and persistent (retains its value until set to something else), unlike a local variable declared in a function which would be lost after the function ends and inaccessible elsewhere in the code. It doesn't require a different label in code, it's purely up to where the variable's declaration gets placed.

### 3.1.3 Life limit then reset

Having score reset each time the ball is lost seems a bit harsh; typically a game provides at least a few chances before clearing score. Additionally, as long as we're only going to zero the score once every few misses, it makes sense to reset all bricks at that time, too, as a signal of a full reset. Introduce a lives remaining variable, displayed during play, which decreases whenever the player misses the ball. When that lives number drops below zero, reset the score and all bricks.

**Details:**
As with setting up score, this will involve creating a new `int` value for lives and displaying it with `text()`. Inside `ballReset()` where score was set to zero instead decrement the lives value, zeroing the score (also resetting bricks) only when the lives count runs out. Remember when resetting to also to bring the number of lives back up to its starting amount. Because there are now several things that need to happen at once for a full reset, moving that set of operations into a new helper function `resetGame()` will help keep things orderly.

### 3.1.4 Click to serve the ball from the paddle each round

With lives in the game, it's now a problem that the first shot of each round is easily missed. To be played in its current form, the player has to be expecting the ball drop after each loss, rushing right to where the shot will be. Instead let's start each round with the ball just above the paddle and a bit to the side, so that it shoots upward at an angle upon the player's next click.

**Details:**
We'll need to create a new `boolean` value, which we'll set to `true` when the ball is currently in the player's possession. Add a `mousePressed()` function, which Processing will call by default when either mouse button gets pressed. Inside the `mousePressed()` function flip that hold variable to `false`. At the top of `ballMovement()` check whether the hold variable is `true`, and when it is set the ball's position to just above and to the right of the paddle's coordinate to give the ball an angle upon launch (vertical might be too easy to aim). Use `return` at the end of that ball holding condition to bail out of `ballMovement()` prematurely before any other ball motion or logic is called. Lastly, remember to set the holding `boolean` back to `true` when the ball gets lost.

### 3.1.5 Keep ball from getting stuck off screen

With the new serve feature outlined in the previous exercise, it's possible to release the ball off the edge of the screen by moving the paddle up half past the boundary before clicking to serve. Because of how the ball's bounce against outer walls is handled, the ball gets stuck along the edge, zig-zagging tightly up then back down. Unlike many other exercises that are about adding new features, this exercise is about addressing a pretty typical kind of bug in the code: we didn't

explicitly program the unintended and undesirable behavior found, instead it emerged at the intersection of two or more features that otherwise work fine in most normal situations.

**Details:**

The issue lies in `ballMovement()`, where the ball bounces horizontally off the left and right screen edges. There's an `if` statement checking whether the ball is left or right of the screen's boundaries, and if so it reverses the ball's horizontal speed. However if the ball is more than 1 frame of horizontal movement past the screen edge, as can be the case when served from just off the screen edge, then the next logic cycle the ball is still outside the boundaries and will reverse its direction again, this time back outward. So it wiggles.

There are a few different ways to deal with this, all of which in this case work out to function basically the same. In all 3 cases it'll help to split the left and right screen edge checks, currently combined with a boolean `||` (OR bars, meaning either condition can be true), into two separate `if` statements to deal separately with the ball passing the left edge versus leaving the right edge. This split will make it easier for us to respond to the condition differently for each case:

Option A) Check the ball's speed when deciding whether to let the ball collide with each sidewall. In other words: only hit the left wall when the ball is currently moving leftward (checking whether `ballSpeedX` is less than zero), and vice versa. By performing this check, if the ball is off screen but on its way back on screen, the game won't flip its motion back outward.

Option B) Incorporate the `abs()` function around `ballSpeedX` to set the new ball speed to absolute value, with the negative in front of `abs()` for crossing the right edge and no negative for crossing the left edge. This way the ball will be forced leftward when off the right side, and forced rightward when off the left side.

Option C) Jump the ball back instantly to a valid screen position, along the edge, in addition to reversing its lateral speed. This way no matter how far off the screen the ball is, it won't matter since it'll promptly be brought back into the play area, and the next frame its movement will be away from the wall which it's already no longer overlapping.

What are some pros and cons of these different approaches? For a game of this complexity they're all basically just as good, but there are some minor differences in terms of the number of calculations involved on average. If we were writing code to make tens of thousands of particles bounce off the screen edges, rather than a lone ball, such optimizations could be worth considering, however in this situation optimizing for code understandability, programmer time, and/or robustness against other complications are better considerations.

### 3.1.6 Minor optimization: brick countdown

Rather than recounting every brick every frame, we can instead count them only once upon resetting the bricks (or even calculate the number through multiplication, as long as we're using a full brick wall at start) and then count down from that number by one each time a brick gets cleared.

**Details:**

This approach is slightly less robust to potential tallying error if we begin to have multiple places in our code that can clear bricks, or power-ups that clear multiple bricks at once, which risks a situation when the level is cleared and should be reset but won't due to the counter falling out of sync at some point from the actual number of bricks. That said, as long as we're careful to set up our code in such a way that removing a brick will always involve adjusting the counter, too, this shouldn't be too hard to do. Take a look at the variable `bricksLeftInGame`. The way it's handled in the brick drawing code (clearing and counting) can be moved to the brick reset function, and then a line should be added to decrease it by one when bricks collide with the ball.

Testing your implementation of this exercise may be a bit easier by reintroducing a few lines to update the ball's position to the mouse cursor's position each frame. Even if the code seems straightforward, it's prudent to validate that it behaves as expected before moving on. Note that this exercise is purely a (very minor) technical optimization, causing no change to gameplay.

### 3.1.7 Images for bricks, paddle, ball, background

Everything in the game is currently drawn with `rect()` and `ellipse()` shapes made in code. Instead create some image files and use those in place of the programmed graphics.

**Details:**

One handy way to get dimensions, proportions, and the overall look right is to first take a screenshot[12] of the game, then use an image-editing program like Photoshop or one available free from `GIMP.org` to draw a new layer of your own images atop the existing graphics. Then slice those layers apart into the respective separates images to be loaded and displayed. As mentioned in the earlier exercise for graphics on the Tennis game, only a single file and `PImage` variable is needed for all bricks, as they can all share and reuse the same reference - there's no need to create different images on disk or in memory for each brick or anything like that.

As always, beware that `rect()` is by default specifying top-left coordinate with its `x`,`y` arguments, whereas `ellipse()` uses the `x`,`y` as a center point, and these offsets should be considered when using `image()` to display an image file instead. When in doubt, leave the programmed graphics in to be drawn on top of or under the image files and that may be helpful in verifying alignment.

[12] In Windows, try the Snipping Tool, or on Mac Shift+Command+4 followed by Spacebar will save the next window clicked on as an image to the desktop.


### 3.1.8 Icons for lives

Instead of displaying a textual number for how many lives the player has before the game resets, many games prefer to instead show a series of icons to display the remaining tries in a more visually appealing manner. Figure out an icon you'd like to use (heart? ball? star? tiny paddle?), create the image, and in place of the lives text display instead show a horizontal row of that image stamped out once for each remaining try available to the player.

**Details:**
Only one `PImage` should be needed, along with a `for` loop to draw each of the icons. To achieve equal spacing of the icons multiply some constant, which should be the width of the image plus at least a few pixels margin, times the iterator variable used in the `for` loop, similar to how bricks are positioned when drawn to the screen.

### 3.1.9 Title screen

Instead of starting immediately when the program is started or reset, the game should instead pause briefly on a title screen. Create an image in your image-editing program of choice, or construct one programmatically with shape and text code, and have that screen displayed to the player until the mouse is clicked to start the game. Remember to include on the title screen a clear message indicating that the player must click to start. Return to this screen when the player runs out of lives. Display someplace on the screen the player's score from the finished round,

since the player may be too distracted trying to catch the ball to notice the score moments before losing.

**Details:**

Add a `boolean` variable, setting it to true by default, to track and set whether the title screen is being displayed. At the top of `draw()` when that variable is `true` display the title screen image and `return` to bypass the game's other update logic. If you haven't already done so for ball serving add (otherwise add to) a function called `mousePressed()` - which Processing calls automatically upon mouse click - and set the title screen's `boolean` to `false` in there. Remember to set the title screen's `boolean` to `true` in the reset code that you call when the player loses their last ball. While it's certianly possible to display the most recent score on the title screen by not resetting the player's score until the game beginning click, alternatively create another `int` to save the score value before resetting the current value to 0. Where the title screen graphic gets drawn, if the previous score is some number other than 0, have it displayed with a `text()` call.

## 3.2 Practice Exercises

### 3.2.1 More points for hit chains

It's exciting for a player to score many brick hits from a single paddle return, chaining together scoring events. Providing an added scoring bonus for that accomplishment can help the player feel like the non-trivial achievement is recognized by the game. Keep a counter for how many consecutive bricks the ball chains together since the last time it came in contact with the paddle or fell out of play. Simply reset that number back to zero any time the paddle touches the ball or the ball leaves the playing field. Then use that number to make consecutive brick hits worth extra points beyond their value as isolated events, or even just as something to recognize and excitedly commend the player for (Five in a row! Ten in a row! Twenty in a row!).

**Details:**

Implementing this one is relatively straightforward: add an `int` to serve as a counter, set it to zero when the ball is reset and when it collides with the paddle, and increase it by one each time a brick gets removed from a collision. Then use that number as a multiplier on some value to be added to the point value of each brick, say earning an additional 50 points beyond the first 100 for each consecutive brick hit. In that scheme, hitting four bricks after one paddle hit would yield 100, 150, 200, then 250 points for a total of 700 points, instead of the 400 points it would earn without this modification. For what is such a simple change from a technical perspective, this can have a sizable effect on how the game gets played. Optionally: find a way to call attention to this feature, such as displaying the multiplier on the screen, or displaying near the score how much the most recent hit counted for.

### 3.2.2 Ball-speed increases

The game's intensity doesn't yet change, making the experience flat. Having the ball speed up during play can help keep the player engaged. There are, of course, different approaches to increasing the ball speed, basing it for example on the number of times the ball hits the paddle, the number of bricks cleared, or time. One of the original ways to increase the ball speed, which is rarely seen in more recent adaptations but still works quite well, is to increase the ball speed based on the deepest brick hit. This introduces a strategic tradeoff for the player: Try to get the ball back behind the bricks, taking out many bricks in each return but making it harder to return, or aim instead to juggle the ball against the lowest bricks to keep it more manageable.

For this exercise, figure out on what basis you'd like to increase ball speed during play and give it a shot. My recommendation is to base it on the height of a brick when hit, speeding up but slowing down only on ball reset. Feel free to experiment!

**Details:**

If basing the ball speed on the deepest vertical position of brick hit by the ball the `ballHit` value calculated for collision can be used as part of an expression to set a new `ballSpeedY`. The main complications will be keeping the ball from slowing back down while also maintaining proper directionality after collision. The `abs()` function for absolute value may prove helpful.

### 3.2.3 Extra lives for score milestones

The game now has lives, and resets when lives run out, but there isn't yet any way to earn more lives. A well-established convention for earning new lives is to reach score milestones: every 10,000 points, for example. When thinking about how often to award extra lives, take into consideration the number of bricks in a full-screen layout, and of course the effect on score of the chain bonus introduced in an earlier exercise. Update the game's code to award a new life based on the player crossing milestones, and convey that milestone information on the title screen.

**Details:**

It probably goes without saying, but since players can use the chain bonus (covered in an earlier exercise) to score points at larger intervals than 100 points, there's no guarantee that a player's score will exactly equal a milestone in the process of crossing it, so the code will instead need to check for when the player's score is greater than or equal to that mark.

The technical difficulty in this exercise is finding a good way to prevent the game from rewarding the same milestone multiple times. One simple solution for this is to keep another `int` variable in memory as the amount the player has to exceed to earn another life, bumping up that number by some interval the moment the player's score exceeds it. That approach has the added benefit of making the next scoring goal convenient to display on screen.

To prevent a situation in which a very capable player dominates the game indefinitely, design solutions include increasing the distance between milestones (so that the second extra life is harder to earn than the first, for example), limiting the total number of lives that can be held at once, or limiting the number of extra lives that can be earned from milestones before no further extra lives can be won.

Remember to reset any milestone-related scoring code or counters when the player loses and the game returns to the title screen. In other words just because the player exceeded (hypothetical values) 10,000 and 20,000 point milestones for extra lives in a previous play shouldn't prevent them from getting credit for doing it again on their next attempt, that would be a simple bug from unintentionally leaving state unreset between games.

### 3.2.4 Power-up functionality: fireball

In this style of game the fireball ability allows the ball to carve right through bricks without being reflected. It's typically depicted with a different ball appearance. While we might handle it by dropping a collectable item from bricks some small percentage of the time - a nice distraction to encourage the player to attempt catching something other than the ball - for now we'll just connect this functionality to the ball exceeding seven consecutive bounces before it returns to the paddle. This creates a threshold effect where in addition to the game incentivizing chains of hits, the player has a discrete and separately recognized goal to continually work for. Have the ball reset from fireball mode upon hitting the paddle.

**Details:**

The bonus scoring counter introduced for an earlier exercise is the main data needed for this functionality. Additionally, a `boolean` variable should be added for the ball to track whether it is currently in fireball mode, and a new `PImage` for how the ball should appear when in fireball mode. When the `boolean` variable is `true` skip the ball-reflection code but keep the brick-clearing code in place. How you'd like to handle scoring for the fireball, whether keeping the same incremental bonus or awarding a capped, fixed amount per brick burnt through is of course up to you, but once coming to a decision consider revisiting the extra-lives milestone(s) to be balanced against this new feature. Likewise if seven consecutive bounces for fireball seems like too few, such that it happens too frequently, or too many, such that it never happens, feel free to adjust the tuning for that number as well.

## 3.3 Challenge Exercises

### 3.3.1 Bricks loaded from 2D array in code

Currently the brick grid contains only one type of brick, and always starts as a solid, filled, rectangular area. In order to support different types of bricks, and different layouts as different levels, we'll want to be able to have some control over laying out by hand where bricks start, and which kind of bricks correspond to each

place. Instead of the code assuming all bricks should be filled in as 1 (signifying that there's a brick starting in that position) in `resetBricks()`, have a 2D array defined in the code so that you can manually set up a custom layout of 1's for where bricks should start and 0's for where bricks should always begin missing.

The current collision and drawing code only accounts for bricks in the grid that are mapped to either a 0 or a 1, but that will change in the next exercise.

**Details:**

Look at the example in the World source tab `commonBonus/free-from-hobbygamedev.com/lab-examples/sketch_gridCollide` or `commonBonus/free-from-hobbygamedev.com/Platformer` for examples of how a 2D grid can be initialized in code. In particular, the Platformer example may be more relevant in this case since it has two separate 2D array grids, one used directly for play (and may have changes made to it during play) compared with one that's copied into the play grid upon reset.

### 3.3.2 Different brick types

In the 2D array we currently treat 0 as meaning no brick and 1 as a plain brick. Let's add support for three more brick types: number 2, meaning a brick that can survive two hits before breaking, 3 for a brick that has to be hit three times to break, and a 4 for an unbreakable brick. Create a new image for each of these bricks.

**Details:**

For the sake of simplicity, it's fine for a brick to downgrade types when hit - so hitting a 2-hit brick turns it into a normal 1-hit brick (visually and in how it's represented in memory), and hitting a 3-hit brick would immediately turn it into a 2-hit brick. In addition to updating collision code to handle the different brick strengths, be sure to account for the unbreakable brick type in code involving `bricksLeftInGame` for figuring out when the level is ready to be cleared due to all breakable bricks being cleared.

The ability to specify custom brick layouts in code, introduced in the previous exercise, should be useful for testing these different brick types: set up the whole bottom row as the brick type you're currently working on.

### 3.3.3 Implement modern brick collision

There's currently a simplified approach used in calculating which side of a brick the ball should bounce off of. As a side effect, in situations where the ball bounces against the corner of a brick, even if that brick is flush against others on each side of it, the ball sometimes bounces back both horizontally and vertically instead of reflecting along the axis the player would expect. Find a way to alleviate this by checking for the presence of adjacent bricks.

**Details:**
Although a precision approach is possible, computing line intercepts, it's generally not necessary for this kind of game. Instead, right before letting the ball bounce horizontally or vertically, determine whether there's currently a brick in that direction which would rule out for example the ball coming from the bottom (if there's still a brick below the currently hit one), the ball coming from the right (because there's currently a brick right of the hit one), etc.

*Since this is a popular question among new game developers, I've included an example solution in* `classic-games/example-solutions/brickCollisionTypes` *as the* `brickBreak_collision_modern` *directory. I built it directly atop the 1.9 warm-up sample solution - note that it doesn't account for other exercises for this game between 1.9 and 3.3.*

### 3.3.4 Try classic ball-brick collision rules

In the original games that started this genre, ball to brick physics behaved in a way that modern players tend to see as very buggy and unnatural. The idea is that the ball will never bounce off the side of a brick - except for sides of the screen it will only every bounce vertically - and it can only hit 1 brick per each return trip from the back wall or paddle (passing harmlessly through any other bricks in the way after its one bounce). Try to recreate this behavior.

**Details:**
This style of ball-brick collision has an interesting effect on gameplay. Namely it makes hitting the back wall much more difficult, since hitting any brick blocks the ball from making it further back. This also makes getting the ball to the back wall more valuable, since once the ball gets behind the wall it's much more trapped back there, indiscriminately bouncing upward upon every collision until it can get down to the paddle without bumping a brick on the way. A few other elements factored into the difficulty and risk of hitting the back wall: upon hitting the back wall the ball speed increased (similar in effect to striking any of the back rows of bricks), and *the player's paddle cut to half its width* until the ball reset. The original game made it possible for the paddle to return the ball at only a few discrete angles, rather than on a continuum as we support in this example, which means here we have better control than in the original games for getting a good angle toward the back wall.

*See* `brickBreak_collision_classic` *in the example solution directory* `classic-games/example-solutions/brickCollisionTypes` *for an approach to this problem, built atop the version of the project as of the 1.9 warm-up.*

### 3.3.5 Load level from file

Setting up the level layout in code is handy, but it's not as flexible nor as easy to work with for dealing with many level layouts as having an outside file format.

We could even write a simple editor for it, or use the operating system's folders to collect, share, and manage our different stage layouts. Instead of defining the layout directly in the program, use `loadStrings()` and `split()` to load and interpret a layout from an external text file. For simplicity, the layout can be of the same basic format as it was in the programming code: numbers separated by commas.

**Details:**

Check out the `loadMapLayout()` function in Racing (another game included with this package) with this for an example of loading an outside text file (gameTrack.txt from the Racing game's data folder) to fill in a 2D array for the purpose of level-rendering and collision. You can completely ignore the `TRACK_TYPE_P1_START` and `TRACK_TYPE_P2_START` conditionals for this exercise - those are used to set player start positions from the level file, which doesn't apply to this project.

### 3.3.6 Code organization: classes

As an earlier exercise we divided this project into multiple files. In this state, Processing still groups all the code together into one large, global aggregate file when it's time to compile and run the game. This means there is no real isolation of variables to what they're for aside from the naming convention followed. Let's take a moment to refactor the code a bit so that each of the file tabs isn't just a separate file, but where possible is also carved into its own separate `class`. This will make expanding the program's functionality a little easier for us.

**Details:**

Rather than defining each individual brick as an instance of a `class`, for this game a more appropriate level of abstraction may be to have a `class` for handling all bricks (or, put another way, the collective level layout) as a set. A `class` is a moderately advanced programming concept that would be difficult to introduce in its entirety here, however the gist is that it gives us a way to collect together something's variables and the functions that use or modify them. In addition to defining the `class`, one or more instances of it need to be declared in the program in order for it to be used.

A number of the included source examples (Platformer, Plane, rtsGame, Racing, plus several versions of Space Battle) use the `class` approach to organizing code. If you're not feeling this particular concept yet, don't let it hold you up! Feel free to skip ahead in this game's exercises, or even jump around into another game's exercises for a bit. It's a concept that inevitably you'll get some more exposure to through digging into the other projects. Come back to this one at any time when you think you've got a sense for how you might go about it.

### 3.3.7 Power-ups: cannon, multiball, sticky ball, points

Earlier for the fireball ability I referred to the possibility of having falling power-ups fall from bricks to distract the player from always focusing on the ball and where it's going. For this challenge exercise we'll look at adding those, in addition to various kinds of special functionality to tempt the player to collect them. Functionality for power-ups in games of this genre often includes:

- Cannon - When the player gets this, a small cannon, rocket, or laser gets added to the player's paddle with limited ammo. Clicking fires a shot straight up, which can be used to clear additional bricks.
- Multiball - Upon getting this power-up the ball immediately splits into three balls, each of which behaves identically to the single standard ball. The player's turn isn't over until the last of the balls gets lost.
- Sticky ball - Each time the ball hits the player's paddle it sticks to it, and isn't released until the player next clicks. Where the ball sticks to the paddle determines its outgoing angle upon click.
- Points - Just a sizeable gob of points, good toward reaching the next extra-life milestone.

Some percentage of the time when the player's ball hits a brick, drop a power-up capsule straight downward. If the paddle catches the capsule before it vanishes off the bottom of the screen, award the player the related power-up. Use different capsule patterns and/or colors to signal to the player which power-up the capsule will activate if caught. Upon the ball being lost clear any power-up states (except, of course, points).

**Details:**

There are a few different steps involved in getting this type of enhancement working. First get power-up abilities working when different number keys are pressed. Press 1 for cannon, press 2 for multiball, press 3 to activate sticky ball, etc. That will make it quick and easy to test functionality, rather than playing with the hope of getting a certain one to fall then struggling to catch it. Work on only 1 power-up functionality at a time.

Once they're working with a key press, add a new `class` for the power-up capsule, and give it a constant downward movement in addition to its own graphic. Manage an `ArrayList` of power-up capsules in your code that you can add another to if a random probability check (ex. `random(50) == 1`) passes at the time of any brick hit, or for easier testing purposes at first make them spawn at the mouse upon mouse click. Check for collision between the paddle and the capsule, and when the two overlap remove the capsule from the ArrayList in addition to triggering its related power-up functionality.

Depending upon how flexible you make your multiball implementation, you may have a maximum of three balls, or you may be open to having each of those three split into three each leading to a total of nine in the playfield.

#### 3.3.8 Series of levels (and design them!)

The game now supports external level files, multiple brick types, even power-ups. However we're not yet taking full advantage of all that functionality. Create a dozen or more custom map layout files, to be loaded into the game in sequence as each stage gets cleared. In the process consider adding numbers to define other brick types, such as specially colored bricks that always drop a particular power-up, or if you're feeling devious, a dark skull and crossbones brick that eats the ball on contact.

**Details:**

Level design is a different aspect of game design than the systems or technical design otherwise discussed in these exercises. Experiment! Try many possible layouts, and don't be afraid to discard the ones that aren't really working as well. By default the dimensions of the map layout includes gaps on both sides and along the top - however if these are things you would like to vary between different level layouts consider rethinking the grid dimensions and corner offset(s) to cover more of the screen area before getting too far into making different stages.

As for switching between levels, there's already functionality in this code that detects when there are no bricks left. That would be a great spot in the code to pop up a congratulatory `text()` message on the screen for a couple of seconds before loading a different map's layout from another text file (you could even load something like a mapList.txt file, itself just a text file with map file names on each line to then be loaded in the order listed).

### 3.4 Write From Scratch Steps: Brick Break

This sequence includes some concepts covered in the Write From Scratch Steps for Tennis Game, covered earlier in this document. For that material, such as getting a ball to bounce around between the screen edges, I've condensed the related set of steps into a single conceptual chunk.

1. Create a new, empty Processing Sketch. Turn off stroke in `setup()` with a call to `noStroke()` so that our shapes drawn with code won't have outlines.

2. Set window size to 640x480. Although the `width` and `height` variables built into Processing can be used to check for screen size after initialization by the `size()` call in `setup()`, for this program we'll define our own `SCREEN_W` and `SCREEN_H` at the top of the program, set to 640 and 480 respectively, to be used in screen location calculations for the paddle and bricks even before `setup()` executes.

3. We'll begin by getting a ball bouncing around the screen, without any concern yet about bricks, player paddle, or being able to lose the ball off the bottom. To do this, create four float values at the global, top-level scope: `ballX`, `ballY`, `ballSpeedX`, and `ballSpeedY`. Write a `ballReset()` function that will set `ballX` to the horizontal center of the screen, `ballY` to 20% the screen's height below

the vertical center, `ballSpeedX` to 2.0 and `ballSpeedY` to 4.0. Call this function from `setup()`.

4. Now that the ball is set up we need to move and draw it. Create a new function, `ballMoveAndDraw()`, and call it each frame in `draw()`. Create two more helper functions, `ballMovement()` and `ballDraw()`, and call those from `ballMoveAndDraw()`. Inside `ballMovement()` add `ballSpeedX` to `ballX` and `ballSpeedY` to `ballY`, in addition to checking whether the ball has passed any screen edge and flipping (multiplying by -1.0) the relevant speed variable to bounce it back on screen. In `ballDraw()` set the `fill()` color to bright green[13] and draw an `ellipse()` 10 wide, 10 tall at the current `ballX`, `ballY` position.

5. To prevent smearing of the ball location, we'll need to start each update cycle by erasing everything drawn during the previous frame. At the start of the `draw()` function, clear the screen in one of two ways: calling `background()` with color values for dark blue,[14] or set `fill()` to dark blue followed by a `rect()` call covering the screen's full dimensions. If using the latter method it will be important to ensure `fill()` value is set to a different color before drawing something else (ex. the ball!). At this point, pressing Play to run the program should show a small white ball bouncing around between the four screen edges.

6. Next we'll add the player paddle. On the line before calling `ballMoveAndDraw()` in `draw()` call `paddleMoveAndDraw()`, which we'll define just below `ballMoveAndDraw()`. For the paddle we'll declare three final (unchanging) global values and one variable: a `final int PADDLE_WIDTH` set to 60, `PADDLE_Y` set to 90% of the screen's height (indicating the pixel row 10% of the screen's height above the bottom edge), `PADDLE_HEIGHT` set to 10, and a non-final `float` value `paddleX` which we'll use as the paddle's current horizontal position. As with the ball, write and define two helper functions to call from `paddleMoveAndDraw()`: `paddleMovement()` (just set `paddleX` to `mouseX`, the latter being the mouse's current horizontal position as updated each frame by Processing) and `paddleDraw()` (set yellow[15] `fill()` and draw a rectangle centered on the paddle's position, using the constants and variable associated with the paddle). Running the game should now show both the moving ball and the paddle, with the paddle following the mouse, although the paddle and ball won't collide yet.

7. To handle collision between the paddle and the ball, add a third function to `ballMoveAndDraw()` named `ballHitCheck()`, to be defined just above it. In writing `ballHitCheck()`, introduce a conditional to send the ball upward whenever the ball's coordinate is within the four edges of the paddle, calculated in relation to the paddle's `final` values and horizontal-position variable, and the ball is moving downward.

[13] In RGB, green is 0,255,0 meaning no red, full green, no blue
[14] In RGB, dark blue is 0,0,50 meaning no red, no green, 20% blue
[15] In RGB, yellow is 255,255,0 meaning full red and green, no blue

8. For ball control, similar to that in the Tennis Game code and examples, we'll want the ball's lateral speed to vary in proportion to how far from the paddle's center the ball is horizontally at time of collision. Use subtraction and division to calculate a -1.0 to 1.0 range based on where the ball hit the paddle (-1.0 meaning far left pixel, 1.0 meaning far right pixel, 0.0 if right in center) then multiply times some horizontal maximum speed (I found 4.0 worked well) to derive the new horizontal speed of the ball. This way, hitting the ball left of the paddle's center will aim the shot left, and hitting the ball right of the paddle's center will aim the shot right, on a continuum to enable to player with enough practice to reliably shoot for specific bricks. This type of ball control may seem unnecessary early in the round, but as the number of targets diminishes its importance to moving play forward rises considerably. Test that collisions with the paddle work and produce the ball movements expected. Remember that a `println("any text here")` command can be added any place in code to show in the Processing console when code is reached, which may be helpful in diagnosing whether code is reaching the inside of your conditional or if the problem is in how ball movement is being changed.

9. When the ball is detected as being below the bottom edge of the screen in `ballMovement()`, rather than reflect the ball vertically call `ballReset()` to reload the ball at its starting position.

10. Just as the paddle's positioning was defined largely by constants (`final int` and `final float`), we'll do the same for parameters to describe the brick layout. The constants we'll define for bricks are `BRICK_W` (`float` for the width of each brick, 60.0), `BRICK_H` (`float` for the height of each brick, 20.0), `BRICK_GAP` (`float` for the visual spacing between each brick, 1.0), `BRICK_COLS` (`int` for the number of columns of bricks, 10), `BRICK_ROWS` (`int` for the number of rows of bricks, 10), `BRICK_TOPLEFT_X` (`float` for the horizontal coordinate of the top-left corner of all bricks which can be centered by setting this to `(SCREEN_W-BRICK_W*BRICK_COLS)*0.5`), and `BRICK_TOPLEFT_Y` (`float` for the vertical coordinate of the top-left corner of all bricks, 20.0). We'll use a globally scoped 2D array of `int` values to keep track of where bricks are, setting each position to 1 to signify when a brick is present there and 0 when the brick for that coordinate has been removed. Declare its dimensions based on the `BRICK_ROWS` and `BRICK_COLS` final values like so:

```
int[][] brickGrid = new int[BRICK_ROWS][BRICK_COLS];
```

11. So far the bricks aren't being displayed or used for collision; all we've done so far is set up the needed memory and positioning variables. We're still a few steps from being able to get the bricks working. The first step will be to initialize the brick grid to begin with a 1 in all positions, which we'll interpret in the draw and collision code as meaning that each brick begins present on the playfield.

Write a new function, `resetBricks()`, and call it in `setup()`. In writing your definition of `resetBricks()` do a nestled `for` loop, meaning a `for` loop (for each brick column, in this case) inside of another `for` loop (for each brick row) setting every index of `brickGrid` to 1.

12. In `draw()` add a call to a new `drawBricks()` function to be written next. In `drawBricks()` begin by setting the fill color to magenta,[16] then replicate the nestled `for` loop from `resetBricks()`, except here instead of setting the value to 1 in each brick position instead write an `if` conditional to determine whether the value at that position is 1. When the 1 value is found in any given array position, draw a rectangle (width of `BRICK_W` minus `BRICK_GAP`, height of `BRICK_W` minus `BRICK_GAP`) in a position that accounts for the top left corner of the brick grid (`BRICK_TOPLEFT_X`, `BRICK_TOPLEFT_Y`) and which brick position is being checked (meaning `BRICK_TOPLEFT_X + col*BRICK_W` for the horizontal position, `BRICK_TOPLEFT_Y + row*BRICK_H` for the vertical position). Now, upon pressing Play a full grid of bricks should be displayed in the play area, though the ball will still move right through them.

13. Since the ball movements aren't in the player's direct control, and handling movement reflection could be a distraction from simply checking which brick is hit, before checking for collision between ball and brick, let's first write code that erases bricks under the mouse position. If we can do it for `mouseX` and `mouseY` then we can surely swap those variables to make it work just as well for `ballX` and `ballY`, but the former will be much easier to test in specific cases like along the edges and corners. There's a particularly easy way to enjoy this functionality while working on the project without having to worry later over whether each use of `mouseX` or `mouseY` has been properly replaced: on the lines immediately before our code for checking collision between the ball and the bricks we'll set `ballX` to `mouseX` and `ballY` to `mouseY`, keeping the ball constantly under the mouse pointer

The place to do this is in `ballHitCheck()`, the function we added earlier to check and handle ball to paddle collision. At the top of that function, set the ball position to where the mouse is located then check for which brick position is under the ball. Once we're confident in the implementation we can comment out or remove those lines.

14. Even though it might be tempting, and logical, to simply iterate through each row and column of bricks to check the boundaries of every brick against the ball every frame, that's not necessary since we're using consistent grid dimensions. It will be good practice to instead handle this collision check by using division to verify only the brick position directly under the ball.

The first step is to determine whether the ball is within the space where any bricks can be. Is the ball right of the left edge of the bricks (recall that `BRICK_TOPLEFT_X` is the number of pixels from the left edge where bricks start) and

[16] In RGB: 255,0,255 for full red, no green, full blue

also left of the right edge of bricks (`BRICK_TOPLEFT_X + BRICK_W*BRICK_COLS`, the right side calculated as the leftmost horizontal pixel plus the width of each brick times the number of bricks columns)? Likewise, use similar coordinate math and comparisons to check whether the ball is also below the top edge of where bricks start and above the bottom of the bottom brick. Rather than nestling `if` conditionals remember that the `&&` logic operator can be used to check if multiple conditions are all true.

Provided that the ball is within the area where bricks can be found, next calculate the brick row and column under the ball's position by subtracting the top left offset followed by dividing by the brick's width (for x position) or height (for y position). For example calculating the brick column under the mouse would take the form of:

```
int hitCol = (int) ((ballX - BRICK_TOPLEFT_X) / BRICK_W);
```

This works since every multiple of `BRICK_W` units to the right of `BRICK_TOPLEFT_X` is where the next brick gets depicted. Performing a similar calculation for y values to determine `hitRow` will let you check with an `if` conditional whether the value at `brickGrid[hitRow][hitCol]` is equal to 1, and if so setting the value at that index pair to 0 to clear the brick in that position.

15. At this point running the game and moving the mouse to guide the ball through bricks should cause each to disappear. While we still have such an easy way to wipe out the bricks, lets go ahead and add a bit of code to reload all bricks when the last one gets destroyed.

Near where the other brick `final` values and 2D array of `int` values are declared globally, add a new `int` value called `bricksLeftInGame`. What we're going to do with this is count how many bricks are left on the field each frame. If there are 0 bricks remaining when the ball bounces off the paddle, at that moment we'll reset all bricks. We're going to wait to reset bricks until the ball next touches the paddle in order to avoid the bricks resetting around the ball.

As the first line in `drawBricks()` set `bricksLeftInGame` to 0 to reset the count, then on the line immediately before or after the `rect()` to draw each brick found in the 2D array increment `bricksLeftInGame` by 1. This way when the function is completed `bricksLeftInGame` will be the total number of bricks that were drawn for the current cycle, which corresponds with how many bricks are left in the level. A slight but unnecessary optimization for this is included as one of the exercises, but could be tackled here if preferred: Instead of recounting every brick every cycle, the number could be tallied just in `resetBricks()` and then decreased by one each time a brick vanishes.

The remaining step for brick reset is to find the section of code that handles ball-paddle collision, and within that `if` condition nestling a separate `if` condition to call `resetBricks()` in the case that `bricksLeftInGame` is 0 at the time of collision.

As a warning to watch out for when testing this functionality: in the paddle collision code we verify whether the ball is moving downward (`ballSpeedY >= 0.0`). For the sake of testing this reset condition we'll want to comment out that bit of logic, otherwise only our first time mousing over the paddle will execute the ball-paddle collision. Recall that as part of the ball collision with the paddle we multiply `ballSpeedY` times -1.0 to send it back upward, and that even though we've currently added a couple of lines to force the ball position to reset to under the mouse constantly, that doesn't change that `ballSpeedY` exists independently in memory with either a positive or negative value depending upon whether it has bounced off the paddle or top edge.

16. Comment out or remove the lines in `ballHitCheck()` that were forcing the ball position to be under the current mouse coordinates. If the check for downward ball movement in ball-paddle collision got commented out for testing purposes, as described at the end of the previous step, uncomment that condition to again account for it in order to keep the ball from getting stuck zigzagging between the paddle top and bottom. Now the player should again be able to aim the ball based on how far from the center of the paddle the ball hits it, though for now the ball will just carve through bricks leaving gaps in its wake without ever being reflected. We'll fix that in the next step.

17. There are a number of different ways to handle ball-brick collision. On one end of the spectrum an easy-to-implement approach that looks incomplete is to flip the ball's vertical movement each time a brick gets removed, independent of which side of the brick the ball hit. On the other end of the spectrum it's possible to use algebra/geometry to interpolate an edge intersection to reflect in a physically predictable manner depending upon whether a horizontal or a vertical brick side got hit by the ball. We'll implement a compromise solution here that isn't too involved to program and yields a fairly reliable behavior, although hitting near corners will still yield some slightly unexpected results for now.

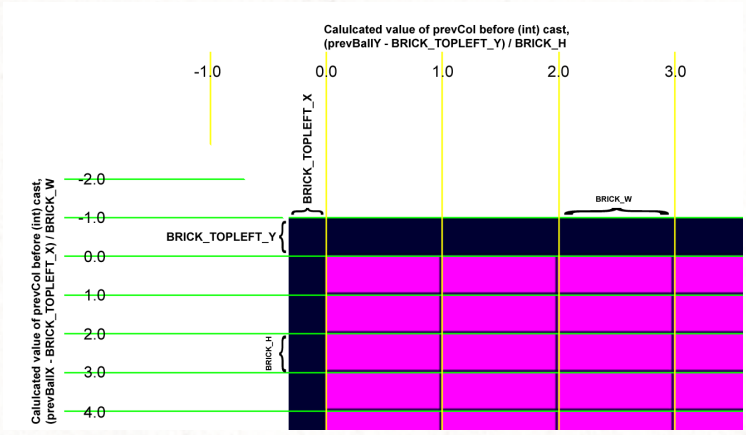
We have the code at this point to determine whether the ball's center is overlapping a particular brick, but what we want to consider for deciding which way to reflect it upon collision is which way the ball is coming from. If the ball is coming from above or below the brick, we can flip the ball's motion vertically. If the ball is coming from left or right of the brick, we can flip the ball's motion horizontally. For the diagonal case we'll reverse both the horizontal and vertical movement, sending the ball back the direction that it came from.

The shortcut we'll use here is to compute the brick position corresponding to where the ball previously was, and compare that to the brick position where the ball currently is. We'll even use an approximation for where the ball was. Rather than keeping track of its past position as a separate variable, we'll just subtract the current ball speed from its position to determine the likely previous coordinate (which doesn't account for the possibility of reflection happening the frame before, but in this context that's often fine).

In the same `if` conditional that clears a brick if one was found for a calculated `[hitRow][hitCol]` index pair create local float values for where the ball was, subtracting the speed values for each axis from its current position on that axis. Then use the same method described earlier - subtract the corner offset then divide by the brick's dimension for that axis - to figure out which brick coordinate corresponds to that previous position.

If the horizontal brick coordinate calculated for the previous position is not equal to the horizontal brick coordinate for the current position, the ball must have hit this brick from the left or right side, so flip its horizontal movement (`ballSpeedX *= -1.0`). If the vertical brick coordinate calculated for the previous position is not equal to the *vertical* brick coordinate for the current position, the ball must have hit this brick from above or below, so flip its vertical movement (`ballSpeedY *= -1.0`). Handling those two conditions independently ensures that if both are true the ball will be reflected completely, on both axes.

As an edge case to watch out for, notice that if the previous ball position would have gone left of the first brick column or above the first brick row, the indexes solved for will be misleading: 0 even though conceptually the answer ought to be -1 to correspond to being an additional row or column over. This is a result of how rounding (truncation, really) happens when casting a negative `float` to an `int` value. If the calculation to determine a brick column index yields 2.6 that means it's over the 3rd brick in. 0.0-1.0 cast to `int` turns to 0 for the first column, 1.0-2.0 cast to `int` turns to 1 for the second column, 2.0-3.0 cast to `int` turns to 2 for the second column etc. The catch is that even though conceptually 0.0 to -1.0 marks the first row left of the grid origin, casting a number between -1.0 and 0.0 to `int` also yields 0, pointing to a different column than it should. Here's an illustration of the values before being truncated (losing their decimal value) by casting to `int`. Bear in mind that (int)(-0.5) and (int)(0.5) both turn to 0, even though -0.5 on the continuum relates to positions left of or above the brick grid:




Although the problem may appear to be a rather complicated one, the solution is fairly straightforward. Since before calculating the brick row and column

corresponding to the previous position we first determine the pixel coordinate of that previous location (by subtracting the ball's current speed from its current position), we can simply check whether that pixel value is left of `BRICK_TOPLEFT_X` or above `BRICK_TOPLEFT_Y`, and if so set `prevCol` or `prevRow` respectively to -1 instead of performing the subtraction and division. This edge case hack will serve our purpose just fine of identifying whether the ball is coming from a different row or column than the one it just entered.

18. Breathe a sigh of relief! True, that one got a little crazy, especially near the end there, but it's another classic style of game that you've now pulled together from a blank file. Good stuff. If you haven't yet done the exercises on the project code provided, you're now ready to dig into them beginning from this foundation you've just created.

# 4 Bomb Dropper


## 4.1 Warm-Up Exercises

### 4.1.1 Adjust timing and scoring to your liking

Just as the two 1970s-style games included were originally controlled with a twistable dial rather than a mouse, the early 1980s game that inspired this classic used that same type of spinning wheel controller. Even though I've tried to make the game approximate the difficulty level and progression of the original, that doesn't necessarily mean that it's still the right tuning for a modern player using a mouse. Check out the Tuning tab/file where I've grouped together most of the variables affecting the game's difficulty: how often extra lives get earned, at what level the game's difficulty peaks, and numerous factors in the speed or eccentricity of bombs and the bad guy.

Does the game feel like it starts slower than you'd like? Or gets too hard too quickly? Does the game never get difficult enough to overwhelm you? Are the bonus lives too far apart for you to ever earn one, or too close together such that you're able to play continuously? Experiment freely with changing the numbers in this Tuning file and observe the effects they have on the game. When the game fits your sense of what's fun to play, you're ready for the next exercises.

**Details:**

Centralizing variables for tuning in one location can be an incredibly useful strategy when developing videogames. Even though the actual usage of these variables is spread throughout the various other files, by putting their initialization in the one place the game's difficulty can be adjusted without needing to root around in the code for where different functionality occurs. The definitions are still grouped into sections, with comments for headers, and some effort went into setting and maintaining a naming convention based on how the constants are used - it's not as though there's a heap here of random and unorganized variables.

This development strategy can make it easier to separate implementation work from design tuning. For a solo developer it may make sense to first get things working, and then later switch gears to focus more on making adjustments to the tuning. For a team of two or more, this sort of approach can enable a non-programmer on the team to help with iteratively working out difficulty tuning, since someone doesn't need to necessarily be able to write or even read the code that applies these variables to simply make adjustments to them. Lastly, by simplifying and making more efficient the act of difficulty tuning, this makes it much easier to fit changes in between testers during a playtest arrangement, in case (for example) the first few testers all find the game unbearably difficult or monotonously easy, and you'd like to get the game better balanced before putting it front of other testers that same day. If the only way to change these values was to wade through the code and start twiddling numbers in the middle of expressions, it might be too risky in terms of introducing a bug.

### 4.1.2 Difficulty modes with different settings

Not all players are looking for the same type of experience from a videogame. It's not merely a matter of ability that some players prefer something a little more rhythmic and relaxing, while others are most entertained by a brutally unforgiving test of their reflexes. Most players fit somewhere in between, and sometimes the same player may feel like playing a game a different way, depending on the player's mood. Additionally, though, there are also real and valid reasons why player ability may vary wildly, from how much of their time they're able to spend on the luxury of playing videogames, to their age, to any number of accessibility considerations. You're never the only type of player for a game - and as a developer you're often more on the expert end of the spectrum due to time spent practicing as the game takes shape.

The variables that we set in the Tuning file are largely the reason for the game's pacing and difficulty. Rather than setting them only to values that make sense for you personally, it's worth experimenting with giving the player a way to select difficulty modes so that they can get an experience that matches their ability and level of investment. Let the player press a number key (1, 2, or 3) or click on a different part of the screen when beginning the game to set their difficulty level to Casual, Normal, or Expert.

**Details:**

To detect a key press add a `keyPressed()` function then use `if` conditionals to check whether the `key` value is '1', '2', or '3'. To detect which region of the screen was clicked, in case you prefer that way of selecting difficulty modes, add some `if` conditionals in the `mousePressed()` function to compare `mouseX` and `mouseY` (the current mouse coordinate, updated by Processing each frame) to the borders of that region. Adding `println(mouseX + ", " + mouseY)` near the closing brace `}` of `draw()` can be helpful in figuring out where the pixel borders are on particular regions mouse regions to check when clicked.

As for how to affect difficulty, the first thing that comes to mind may be to simply have a different set of Tuning variables for each difficulty. This could very quickly get out of hand as a messy and tricky set of carefully balanced data to maintain. Instead a much simpler approach is to identify one or two key variables that can change the difficulty of the game just by being raised or lowered while the other factors remain the same. Setting the difficulty mode would then just set a new `float` variable to use as a multiplier - maybe set to 0.6 for easy mode, 1.0 for normal mode, and 1.3 for hard mode. In a common action game that multiplier might be used to affect player start health or enemy damage, or have an inverse effect on time limits or ammo counts. For Bomb Dropper it probably makes sense to apply it to a variable like the base/initial bomb fall speed, but it could also be applied creatively to make the enemy movements less erratic in easier difficulty modes.

### 4.1.3 Keep high score (in session)

Keeping a record of the player's highest score - even since just opening the game for a session with a few rounds - can make it easier to feel and notice a sense of improvement over repeat plays. Keep track of the highest score the player has earned since turning on the game, and display it in an appropriate way during play.

**Details:**

In the GlobalVals file you'll find where the `score` value is declared as an `int`. You'll need another `int` for keeping the high score. Then you'll need to check whether the `score` value is greater than the high score variable, and if it is just set the high score variable to the current value of `score`.

While it would be fine for a game this simple to perform this comparison check every frame in the `draw()` function (which is where it will need to be displayed with another `text()` call, in any case), as a minor optimization exercise you could instead look for where in the code the score gets changed and only do the high-score update check there. That makes sense of course since we only need to rethink whether a high score has been set when the player's score is increased from the value that it held at the time of our last comparison.

#### 4.1.4 Find and add appropriate creative commons music

While the classic game that Bomb Dropper is inspired by didn't have any music, adding music to a game is a great way to establish a certain tone, reinforcing the game's pace or atmosphere. While becoming a music composer is certainly outside the scope of these exercises, as a solo developer practicing design, programming, and production another option is to look for Creative Commons music that's suited to your game and including it with credit to the person that made the music.

While most Creative Commons music is free to obtain and use, it's not free in the same way that Public Domain audio is. Terms vary by composer, but the most common arrangement is that the music is free to include in your non-commercial game as long as you properly credit the composer for their work.

**Details:**

Kevin MacLeod's Incompetech.com[17] is a great place to search for Creative Commons music, though many other options are out there. Mr. MacLeod has spent many years creating music under the Creative Commons license, and his high-quality work covers a broad range of styles.

Narrowing down what music you want to use for your game sounds easy but can require a great deal of searching and sorting. One process that I've found helpful is to create folders for `fits`, `maybe`, and `no`. Then download any songs you're even open to maybe considering based on the preview, even if you're not quite sure yet. Once the songs are gathered, sort them into those three different folders. Notice that the folder is `fits`, not `best`, nor `favorite`, because it's not about deciding which song you like the most on its own, it's about picking a music track that's appropriate for the game. Once you've narrowed it down to a list of only a few left, run them by a friend to get their input on which seems to best fit the gameplay.

Whereas the sound effects are connected in code using `AudioSamples` that can be played with a call to `trigger()`, for looping music it will work better to instead use `AudioPlayer` which can be started with `loop()` after being loaded.

### 4.2 Practice Exercises

#### 4.2.1 Two-player mode (keyboard for attacker)

A spastic, mostly random artificial intelligence, controls the top character but there's no reason why a second player using keyboard keys couldn't control it instead. At the initial screen create an option for a second player to steer the attack character if the game is started with spacebar.

[17] Go to the "Royalty-Free Music" section of his site then Full Search. At the time of this writing (subject to change) the direct URL for that section is: `http://incompetech.com/music/royalty-free/`

**Details:**

If you've already taken on the exercises for the Tennis Game you may already be familiar with some of the challenges in getting smooth movement out of keyboard keys, even along a single axis of motion. Another option here that would not have made sense for the Tennis Game is to have the character always in motion, and just use inputs as a way to prematurely reverse direction. There's no reason for the top character to ever sit still since, unlike the bottom player here or the paddles in Tennis Game, the top character never needs to block or catch anything. Likewise, there is no need for it to be controlled in a way that would make it easy for the player to get to a specific position in a hurry with precision movements.

Getting a proper two-player mode working may be a little more involved than replacing the automatic motions with keyboard input. Should the game still progress by levels? Should the attack character still automatically drop bombs at a frequency based on the stage number, or should another button be used to manage the drops? When controlling the attacker does a human player expect to be able to speed up and slow down, in addition to reversing direction? There's no right answer to these questions, but they may be worth exploring in further iteration and testing. For the sake of this exercise just getting the two-player mode working at all will suffice.

It's important to include some additional text on the title screen indicating to the player that this two-player mode is possible, and how to start it. It would even be a good idea to somehow indicate on the screen which keys the keyboard player needs to use to play. A videogame needs to be self-explanatory - a player needs to be able to understand it without you there to explain it.

### 4.2.2 Save high score to file, give way to reset it

Although we introduced a high-score feature in an earlier exercise for this game, as implemented now it cannot keep track of the score between play sessions. What we'd like to do for this exercise is figure out how to save and recall that score between sessions, in addition to giving the player an easy way at the game's start screen to clear that saved score.

**Details:**

The functions needed for this are `saveStrings()` and `loadStrings()`, which save a text file and load a text file respectively. There is documentation on these functions through the reference pages at `Processing.org`, however you can also peek ahead to the Space Battle (final version) source code in which there's a similar example in the ScoreAndProgress tab that could be adapted to this case. Although the `saveStrings()` and `loadStrings()` functions are designed to work with `String` arrays, to save and load a lone score value the array only needs a single entry (but must still be handled in code as an array).

## 4.3 Challenge Exercises

### 4.3.1 Items that should be dodged

If the attacker occasionally dropped objects that should be dodged - allowed to pass rather than caught - the game's difficulty could be increased considerably without needing to rely on inhuman speed tuning. If you've already implemented difficulty modes, this feature may be one to allow only in the hardest difficulty mode. The items to be dodged will need their own unique graphic, in addition to the extra code for handling what happens when it hits or bypasses the buckets.

**Details:**

Rather than having a completely different type of object for this, it's probably easier to fit it into the bomb class as a `boolean` toggle that gets set based on a `random()` probability when dropped.

The more difficult side of this challenge is thinking up what type of object the attacker might throw that the player is likely to recognize as something to be dodged rather than caught. Typically in games, something like a bomb would be dodged, and it's a bit backward from convention that in this game the player needs to move toward them rather than away. While in many other games characters would collect coins, rings, fruits or potions, it's not clear from the spatial metaphor of this game that dodging those things would indicate collecting them any more nor less than catching them with the buckets.

One approach is to think of things that would destroy the buckets on contact. Bombs seem like they ought to be in this category, however conceptually what happens in this game is that the water puts out the fuses. If every so often the attacker dropped an anvil (a pretty generic and widely recognized comic shape, and one not too difficult to draw recognizably) it would make sense to steer clear of it rather than letting it hit the bucket.

Another angle would be to embrace the abstractness of this activity, and just create a glowing or flashing light ball that visually stands out from the bombs. Without needing much explanation or introduction this is an object that has the advantage of being easier to distinguish in the chaos. While it might not be understood on the first encounter, the gameplay and rounds are rapid enough that it would be learned on the first time it appeared, and thanks to its abstractness there wouldn't be conflict with expectations based in how objects work or what they mean outside of the game.

### 4.3.2 Non-random adversarial AI

An earlier exercise involved supporting a second player, making it possible to control the attacker with keyboard keys. When a human controls the attacker character rather than it automatically switching directions at semi-random intervals, is the optimal strategy for the attacker player to behave in a similarly random fashion, or does a better technique emerge for sneaking a bomb past the catcher? Think through what considerations or patterns you engage in when

controlling the top character manually, and try to use those as a foundation for programming a more challenging top opponent.

**Details:**

Artificial intelligence (AI) in the broad sense is a very involved and challenging topic, though as an early practice experience with it this is perhaps the most straightforward case possible: one limited axis of movement, in response to one limited axis of movement by the catcher. Unlike the Tennis Game AI in an earlier exercise this opponent doesn't need to account for reflections, however it does still need to appear somewhat rational in its movements and keep the player always guessing. This can be a difficult balance to strike, since making the AI purely rational can render it easy to manipulate, whereas making it purely random leads back to the original behavior.

One common approach to this conundrum is to set up the AI to behave differently depending upon what mood it's in, and then transition between those moods based on some mixture of randomness (to prevent it from being figured out completely and manipulated) and prioritized criteria. Perhaps it has a random mood, similar to its original behavior, a wave mood, in which it just slides back and forth, and a fleeing behavior that makes it try to get and stay at the farthest position it can from the player. By toggling between those three moods, even if just randomly, the player will have a lot more unpredictability to account for than if it always behaves in basically the same way.

### 4.3.3 Particle effects for explosions

Explosions in the game are handled as a series of animation frames, which gets the job done but isn't very dynamic. Every explosion looks the same way every time. To alleviate this we can implement particle effects - a term we assign to a large number of decorative elements (visual only, no mechanical consequences on gameplay) that move, spin, scale, fade, die out, and otherwise operate independently of one another. Depending on the image(s) that we use for the particles and what variables we use for their motions, they can be used to simulate smoke, explosions, even bubbles or splattering mud. Implement particle effects and use them for bomb explosions.

**Details:**

In the `commonBonus` folder the final versions of the `plane` demo (`plane_final`) includes a particle-effects implementation that could be borrowed and adapted for this exercise. Both the `Particle` and the `ParticleSet` files/classes will be helpful. The former is the data structure for each particle, the latter is a managing container for the whole set. Most of the work in adapting that solution to this project will be in simplifying it down to just the parts needed, as the particles from the `plane` demo support flames, destroyable terrain, colored clumps of land, and some other part features specific to that project. All that's needed here is smoke and some form of explosive burst. When copying the .pde scripts for particles from

the `plane` demo be sure to also check for which image files will be needed out of the data folder, and copy it into the data folder for this project as well.

## 4.4 Write From Scratch Steps: Any Specific Classic Game

Whereas most of the demonstration projects and exercises included are developed based on a mixture of common features for a particular type of historical game, in this case not enough clones or copycat games were inspired by this classic to exist as a genre or style in its own right - there's just one specific game.

The downside here is that rather than being able to choose between the several different ways that games within the style handled any particular issue (difficulty tuning, collision response, AI, or optional gameplay elements like power-ups) we're left with only a single isolated example of how everything fits together nicely in the one time it was done professionally.

The upside to this situation is that it means we're in a great position to hone in on studying and implementing the features and tuning of a very specific historical game, rather than working with a conceptual average across a variety of implementations. While I don't recommend cloning an existing game as a way to make a name for yourself or as a wise business move,[18] as a practice exercise when getting started cloning a specific game's mechanics is an excellent way to gain a deeper appreciation of design details and implementation challenges.

Seven of the first 10 videogames I developed were inspired by (though not exact clones of) successful and widely recognized classic games. Even as I've become more experienced as a developer I still occasionally go back and practice implementing core engine functionality from older games that I want to understand better. In the cases of those more-exact replicas I never release or advertise publicly my practice remake. I instead pull concepts and code from them into unrelated projects that I'm developing, to make something distinctly new out of the taken apart building blocks. In addition to remaking detailed part of games that I liked during my first year as a game developer I also did this specifically in my second, fourth, sixth, seventh, ninth, tenth, thirteenth and fourteenth years as a developer. I can remember that because I can still see clearly in the games I finished those years which parts I appropriated from my practice remakes.

Rather than providing a walkthrough on how to reproduce the specific mechanics included in the Bomb Dropper example, for this project I'll go through the general steps and considerations that I followed to program this as a practice remake of a classic game. It's my hope that equipped with this approach you'll be

[18] Though borrowing and extending a concept to more fully realize its potential has admittedly yielded huge dividends for Atari's *Pong* (variation on Ralph Baer's *Table Tennis* for his Magnavox Odyssey), Rovio's *Angry Birds* (a cartoony reimagining of *Crush the Castle* by Joey Betz), Notch's *Minecraft* (which began as a remake of *Infiniminer* by Zachtronics), *Guitar Hero* (a blatant riff on Konami's GuitarFreaks), and virtually any game made by Zynga that you may have heard of.

ready to start on a practice remake of your own for any different classic or simple game of your choosing.

1. Identify a game or part of a game that has few, simple objects and interactions, preferably a game that takes place mostly on "one screen" rather than requiring an expansive scrolling space. Remember, practice consists of doing something you already know how to do and want to become more skilled at doing. The purpose of this is to get some wins under your belt, to gain some momentum as a developer. This would be the wrong time to risk biting off more than you can chew - there will be plenty of time to confront that sort of difficulty when working on your original games.

2. Create a new, empty Processing Sketch.

3. Figure out the dimensions of the game you're looking to remake. Even if your version's window will be larger or smaller, often proportions of a screen can affect matters of gameplay movement, puzzle visibility, UI layout, etc. If unable to find the exact resolution, a fallback strategy is to take a screenshot of the game being played in an emulator, browser, or YouTube video (Snipping Tool in Windows to get a screengrab, Cmd+Shift+3 on Mac to save screen capture to desktop) then measuring the screen dimensions with a selection box in an image editing tool. Set the Processing window to a size that matches those proportions but is large enough to see easily on your monitor.

4. The top priority is to recreate the game's feel and functionality. Recreating the exact appearance and sound of the original by finding or screen capping the game's visual assets and finding or recording the audio files may be tempting, but isn't a very productive use of time. Graphics should begin as `rect()` calls then be replaced (if ever) by newly drawn images of your own. Sound effects are usually not essential, depending on the type of game, but placeholder or very simple sounds can often fill the role well enough. If you'd like to practice implementing animation, including aspects like connecting certain jump poses to vertical movement speed, it may be worthwhile to take note of how many different animation frames the original game uses, though dealing with animation will tend to come later in this process.

5. Don't worry about the title screen or non-gameplay menus. Focusing on these elements is another distraction. It may look and feel like progress but contributes little to figuring out the game's mechanics and can greatly slow down iterative testing when there's extra clicking needed to get started for the next few dozen or hundred times the game will be run while working on it. Upon pressing Play in Processing the game should start immediately in action.

6. Rather than trying to think through all of the potential classes and variables needed by the game, it will go much more smoothly to focus on one element of the

game at a time while getting the basic functionality working. Get player movement and input working before doing anything for the enemies, items, goals, or level structure. The player and input will be main way of validating the functionality of those other elements. Player movement and input are also usually central to how gameplay is differentiated between games.

It's often easiest to implement something in a simplified form before trying to layer complexity onto it. Get the character's position to show up at all before trying to get it to move; get it to move before getting it to move right, and so on.

For the player movement and input of the example used for Bomb Dropper, moves are very rapid and straightforward: Like the Brick Break paddle, the buckets simply keep up with the horizontal mouse position, remaining locked at a certain vertical distance from the bottom edge of the screen. One caveat of the player's avatar in this game is that it operates as a stack of three independent objects, one of which gets removed each time the player loses a life, and more of which can be earned by during play (back to the starting number of three) by exceeding score milestones. Before I got anything else in the game working, I set up the player's avatar to be switchable between 0, 1, 2, and 3 buckets, using number keys to toggle between them for testing purposes.

7. As part of the iterative improvement on your implementation, form specific theories about what happens at a detailed level. Write down that hypothesis, since specifics are otherwise easy to forget while switching between tasks. Then take those ideas and go play the original again to verify, invalidate, or otherwise better inform your thinking about the game's behavior in particular cases. What may seem trivial can occasionally play an important role in emergent, rewarded, or optimal player interactions.

In the case of implementing Bomb Dropper, I couldn't remember whether the number of buckets remaining affected the player's ability to catch bombs, or if it was merely a fancy way of illustrating how many lives the player had remaining. At the core of the question was whether bombs could be stopped with the sides of the buckets, rather than only the top of the stack. Checking this against the original game I confirmed during play that, indeed, bombs could be stopped by sideways movements even after passing the top of the stack, so that having fewer buckets remaining did make the game more challenging.

8. Look up FAQ (Frequently Asked Question) files on the game, tips from skilled players, and the analyses of dedicated players. Especially in the case of classic games there's often a treasure trove of documentation and discussion out there about particulars that are not at all evident to someone newer to the game. Tournament competitors, speed runners, and gamers competing for high scores pour over the details, often leaving a wake of surprisingly technical analysis.

In this case, I discovered a few key details that, even as an experienced game developer, managed to elude my investigation since they're so subtle and unusual. By reading up on tips from people that analyzed the original game, I learned that bombs drop more frequently on odd-numbered levels. What a clever trick to keep

the player on their toes, constantly adapting! I also learned that when the player loses, there's a special formula for deciding how many bombs the player will deal with next, and at what difficulty. Because these variables aren't surfaced in the game's interface or otherwise communicated explicitly, and because these ideas are fairly unusual relative to other game implementations, it would have been very unlikely that I could notice these details without doing a simple internet research.

9. One approach to testing the accuracy of your remake is to compare the scores you get in your implementation to the scores that you get on the original. Since scores inevitably fluctuate, comparing one or even a few rounds isn't a very good test - especially for games that can be played in relatively little time, which is generally the target when modeling your work after a tiny game or game segment for practice, it makes sense to play a half dozen to a dozen rounds and record your scores. Are the minimum, maximum, and average scores from both your remake and the original close? If not, and you're confident that the difference isn't indicative of a missing element, consider adjusting your tuning or tweaking to come closer to the challenge of playing the original. Remember that the target of your remake is the feel, not the code, and difficulty is an important factor in how it feels to play a game.

10. It's worth pausing to double-check: Does your game still *look* like the original, in how it moves? Are there irregularities in the original, where obvious patterns are showing in yours? Irregularities give a game or character personality, and in digital space, rarely happen by accident. Does your version look as graceful, rapid, or otherwise fitting of the same adjectives as the original? Even though the specific graphics aren't our focus, the overall sense of pacing and level of predictability during play also factor into how it feels. Paying attention these details is another way to become more aware of these aspects in our own original games, and approaches to refine them even after the game otherwise seems done from a purely functional perspective.

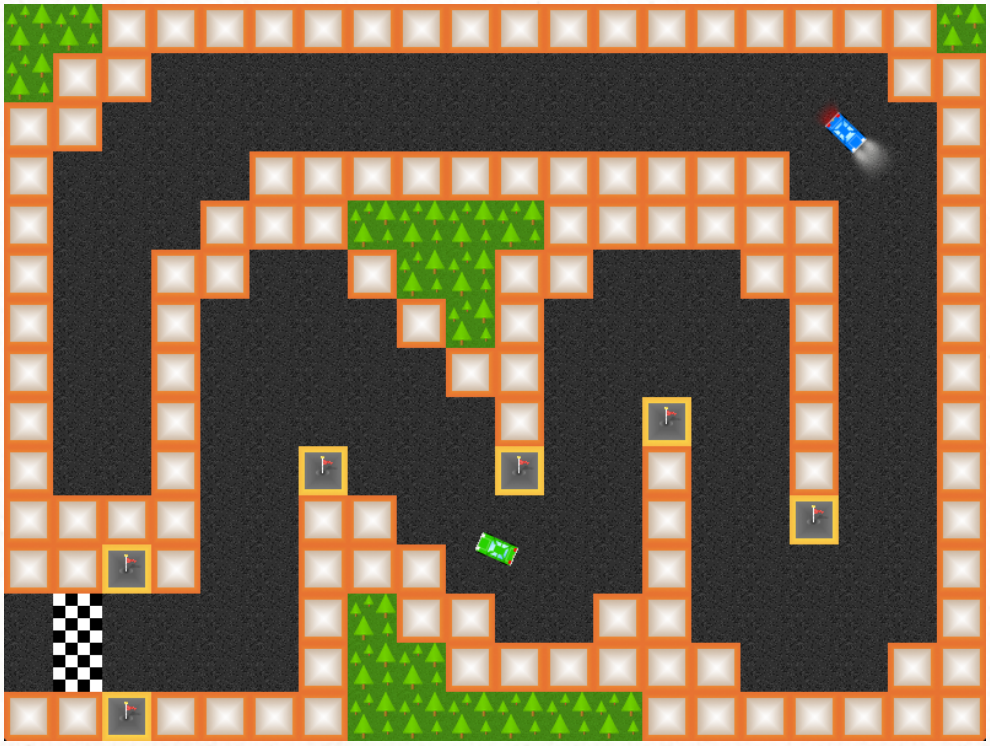
Of course, adjusting the game's tuning to affect how it looks during play may mean unbalancing the work that was just done to get the player's score into a range consistent with the original. What gives? What is more important? For better or for worse, this is the only way to satisfy both goals - there's no shortcut, and both are equally important. Go back and forth another time or two (or however many times it takes) between tweaking for score and tweaking for overall look during play. Each time the changes can be increasingly minor, and with this iterative approach you'll rapidly settle in on values that fit both needs as a compromise.

11. Lastly, go for attention to detail, especially for those non-gameplay aspects of the game that you've been ignoring. Does your version end or even reset instantly when the player loses, while the original version has a delay that lets the player orient themselves? Figure out what you'll need to change in yours to match that effect. Such a seemingly small change can often be surprisingly inconvenient

to code, but also have a big effect on how polished the game feels to play. Ask yourself, and be brutally honest, how is what you have made so far different from, and not up to par with, the game you're trying to model?

To once more stress the real point of this exercise: The goal here is definitely not to knock off someone else's creative work, nor to go through the motions of remaking classic games since it's something experienced developers recommend doing. The goal is simply to become a better developer. It's not enough to just think or talk about what the main objects and interactions are in the game. Digging in on trying to recapture through implementation what works about the original is an exercise that will task the developer to figure out what seemingly small but critical concepts they've been overlooking when considering game development. What's missing from your initial implementation of the game that you find yourself needing to go back to add? In your own future projects, stay especially vigilant to the possibility that your game similarly needs another pass for those kinds of features and details after it otherwise seems complete.

# 5 Racing



## 5.1 Warm-Up Exercises

### 5.1.1 Headlights on Player 2 car

Player 1's car has its headlights turned on, whereas Player 2's car has its headlights turned off. Turn the headlights on for Player 2.

**Details:**

This change isn't in the code at all! It's a simple trick accomplished by using a larger image for the car than is needed for the car's body, with the margin space used for partially transparent gradient that when drawn over other objects in-game looks like headlight glow. Look in the data folder for the player car images, and compare the two images. Turning on the headlights for Player 2 can be accomplished by:

1. Renaming the old Player 2 car image (in case we want it to bring it back later)
2. Duplicating (or copy/pasting) the Player 1 image
3. Renaming that duplicated image to the name of the original Player 2 image (the one the code is already looking for to use)

4. Recoloring the car body in the new Player 2 image to look different than Player 1's

Instead the edits could of course be made to the Player 2 image directly, extending its dimensions then drawing on headlight glow, but the above series of steps ensures that the headlights on both cars will use exactly the same effect. Having art that matches in style goes a long way in helping a game look well put together.

### 5.1.2 New tile types: grass, oil slicks

Collision in this game between the cars and the world is handled in a very similar way to the collisions between bricks and the ball in the earlier Brick Break example project. A 2D array of `int` values stores a different value for each type of stationary ground object in the level: 0 corresponds to open road, 1 to the finish line, 4 to a plain wall, 5 to a square filled with trees, and 6 to a flag post. I skipped 2 and 3 because those have a special purpose: where the game code finds 2 or 3 it uses those to determine start coordinates for the player cars, then fills in a 0 for open road during play. These values don't need to be memorized, or even typed in different places in the code, because they're each defined as a different `final int` at the top of the World tab. This way instead of checking if the tile under the player's car has the value 0, we can check whether it's equal to `TRACK_TYPE_PLAIN_ROAD`.

Create two new types of level tiles: grass and oil slick. In addition to drawing the tile art add them to the track map, and program special functionality for when cars drive over them. When either car drives over grass make it move significantly slower, and while driving atop an oil slick don't let the car turn.

**Details:**

This exercise consists mostly of figuring out the pattern in the existing implementation so that you can extend it for your own. This sort of pattern matching applies not only to the code but also the data files involved. For example, a handy way to ensure that your new tile art for grass and oil slick squares are the correct size and format is to start work on them by duplicating (or copy/pasting) an existing tile in the data folder, renaming the copy, then drawing on top of it. Alternatively this information is of course also in the source: `TRACK_TILE_WIDTH` and `TRACK_TILE_HEIGHT` are both 40, so the image dimensions should be 40x40.

The new track types should be defined as `final int` at the top of the World file for values 7 and 8. Declare the matching `PImage` after `trackFlagWall`. Load the image in `loadMapImages()`. To add test squares to your track edit a 7 and 8 into the gameTrack.txt file from the project's data folder (since it's what gets loaded at the top of `loadMapLayout()`. In `worldDrawGrid()` a couple of new cases need to be added to the `switch` statement (keeping our `PImages` in an array would let us skip that step, but we'll look at that in a later exercise).

Just getting the two extra tile types to show up, as described in the previous paragraph, shouldn't involve much complication. Where things may get a bit

messier, since there are many different ways to do it, is in figuring out how you'd like to connect driving over grass or oil to the consequences on movement and control. `updatePosition()` in the Player tab includes a demonstration of checking whether the player is driving over empty road or the goal, but that isn't necessarily the right or only place to make the player slow down on grass or lose turn ability on oil. Tile positions are very fast for a computer to calculate, and especially with only 2 cars rather than hundreds there's really no harm in using `whatIsAtThisCoordinate(x,y)` elsewhere in the Player `class` to check which tile type is being driven over. Alternatively the `tilePlayerHit` value could be pulled up in scope from local to the `updatePosition()` function to instead be a `class` level variable, making it persist between being computed each frame and accessible to other functions defined in the Player `class`.

### 5.1.3 Day/night or theme tile sets

By switching which graphics get used to draw the track, the same track can be depicted as being at a different time of day, a different season, or even someplace else like a different culture, climate, or planet. Duplicate the existing tile art, mark the copied files with some consistent way to designate which theme they are intended for, and adjust or redraw each tile to fit the theme. In game let the player(s) toggle between the default theme and your new theme by pressing a key - when we later add support for multiple track layouts we can connect different stages to different themes, but this simple key toggle works for testing purposes.

**Details:**

Define a new `int` value in the World file. Based on its value load a different art set. In the keyHandling file check for when a currently unused key is detected in `keyPressed()` using a comparison like `if(key == 'u')`, and use that spot in code to change the theme `int` in addition to calling the function you use to load track `PImage` variables based on that theme value.

If you're looking for a relatively simpler way to generate a whole separate theme, adjusting the image's brightness and contrast can be a way to crudely simulate a different time of day. If changing time of day, a nice touch that can be done to show attention to detail would be switching the car graphics based on the theme as well, so that headlamps are only showing for the later theme. Switching car models for theme could also be used to put snow on top of the cars for a winter theme, though it's worth considering that a theme with snow in it might imply to the player that the cars should handle differently in those road conditions.

*See* `classic-games/example-solutions/racingWarmUpExercises` *for Racing warm-ups answers, and in video:* http://www.youtube.com/watch?v=5MxK9scCpC8

## 5.2 Practice Exercises

### 5.2.1 Load track images as array

As covered in one of the warm up exercises, adding a new tile type currently involves updating a number of different places in code. Any way that we can reduce the number of spots in code needed for adding a new tile type simplifies our work in adding more tile types, in addition to reducing the likelihood of bugs from updates in the different areas falling out of sync.

Instead of having a different `PImage` variable per track section, make a `PImage[]` array named `tilePics` and set up your code to use that instead.

**Details:**

Having a `PImage[]` array named `tilePics` could replace the lengthy `switch` case in the `worldDrawGrid()` function with this:

```
int tileTypeHere = worldGrid[row][col];
image(tilePics[tileTypeHere],tileTL_pixelX,tileTL_pixelY);
```

We could also get rid of the `PImage useThisTileImage` variable in that function. Keeping straight which image index corresponds to each image file is relatively simple since we already have `final int` values enumerated for different tile types.

One complication that arises here is that the `final int` values for track section types include numbers for where each player car starts, even though track graphics are not needed for those numbers. We could load those as tiny or empty dummy files, leave them null/undefined since they never get displayed on the map anyhow, or alternatively use them to hold custom per-theme car graphics (this would be a bit sloppy, but as an upside keeps all art for a switchable theme together in one whole set).

Another solution to that issue would be to rearrange the index values so that instead of having the Player 1 and 2 start positions defined as 2 and 3, they could be moved to the end of the list (or given arbitrarily higher numbers, 98 and 99, or negative numbers) then shuffling your new grass and oil types from values 7 and 8 to 2 and 3, making the same switch in your track file. By moving the map values for player start positions out of the consecutive range of tiles that need track art, the array could then just be sized appropriately to have art for each track tile without needing to deal with gaps.

### 5.2.2 Collision at front and rear of car

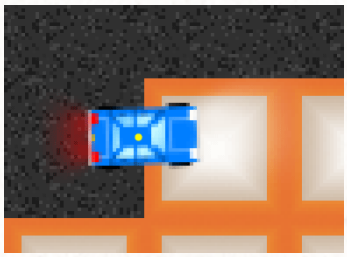
The cars currently can drive halfway into a barricade, because for collision purposes it's treated as a single point at the center of the car. The goal in this exercise is to improve the collision relationship between the car and track obstacles so that the car is no longer able to overlap the barriers, or at least not so severely (a little overlap in certain situations is fine).

**Details:**

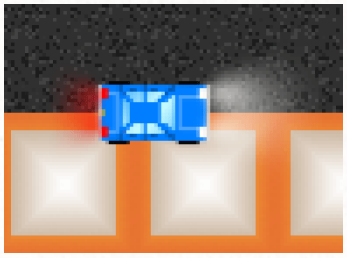
There are a few ways to approach this problem. One fairly natural solution is to check multiple points for the car - rather than only verifying that the spot below the center of the car is open road, with the use of `sin`() and `cos()` the spots under each of the 4 lights on the car can be checked relative to the car's rotation. Alternatively a simpler check would be to use one point at the front center of the car, and one at the rear center, which wouldn't be as thorough of a solution but could eliminate the most distracting and common sort of overlap that happens when the car runs into a wall head on.

Of the many reasonably good solutions available, none are totally perfect. They'll all wind up with some edge case or another where it breaks down and overlaps a bit. When situations like this one come up during game design, where a perfect solution is elusive but many pretty practical solutions can be implemented in a reasonable amount of time, it's worth taking a moment to reflect on who you're more concerned with satisfying: the person playing the game, or the person ignoring the gameplay to test your car's collisions against walls and corners. If the former, and I find that's generally a good answer, just implementing a practical solution and moving on will free up more time to spend energy on other features and aspects that someone really playing the game is more likely to notice and appreciate.

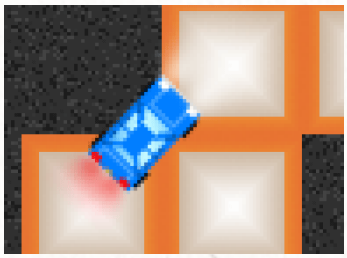
Let's take a visual look at this issue and the tradeoffs of potential solutions:



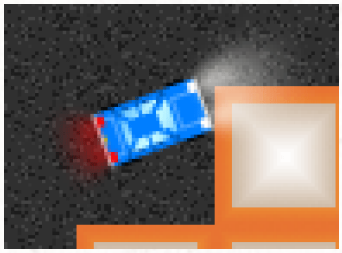
Here's the main issue. The dot in the center shows the point checked for car collision against walls. Because of that, the car can drive halfway into obstacles.



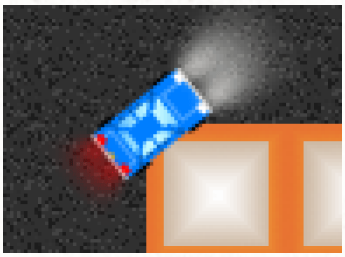
Another side effect of the issue we're trying to correct is that the car can drive with its left or right half atop barriers. The shortcut method of checking for collision at the front center and back center of the car wouldn't prevent this.



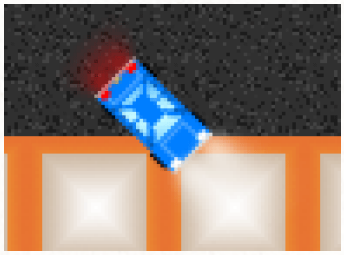
While someone simply playing the racing game will be unlikely to wind up in situations like this, trying to get the car into this position can be a quick way to check a new collision solution. Does it prevent you from getting here? Can the car get stuck?



Checking for collision at the 4 corners of the car, or roughly under the car's four lights, can produce pretty solid behavior. With that solution the car could not drive any farther forward from this position.



As mentioned above though shortcut solutions inevitably still have some special cases that don't work perfectly. Checking the corners/lights would still let the car do this. An additional check at the center of the car's side could prevent this particular goof.



When the car was treated as a single point collision detection could simply hold its position in response to wall contact. Now that we're using non-centered points, it's necessary to take care of how rotation into a wall gets resolved in collision handling code.

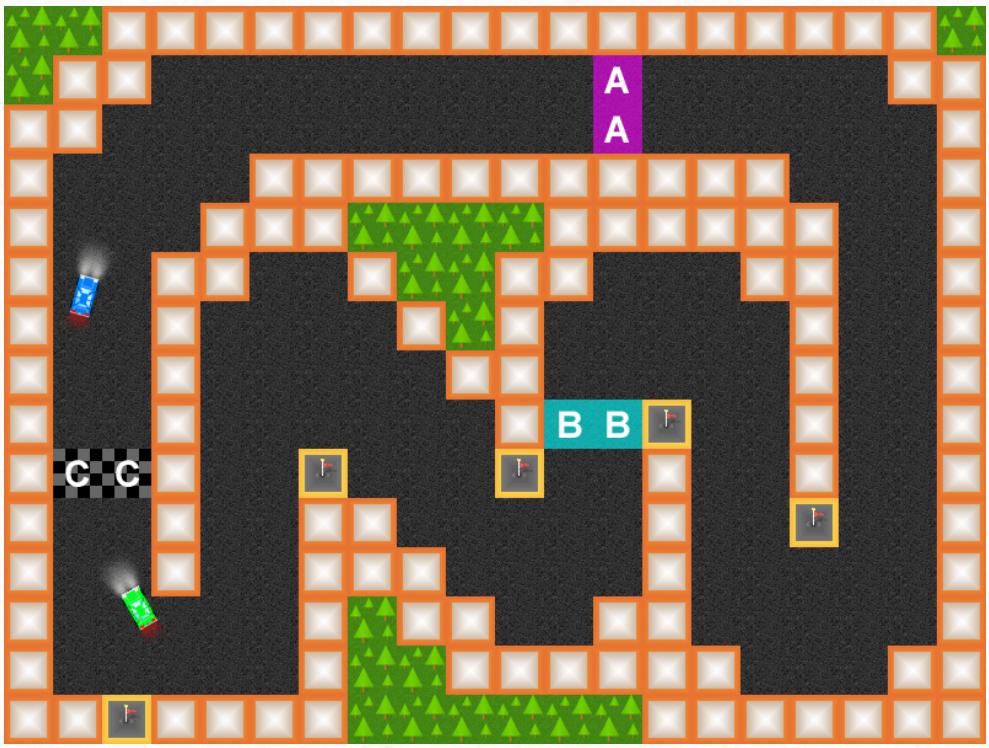
While these improvements and issues are valuable in keeping the car from overlapping walls and other obstacles, for the purpose of having the car respond to road effects like grass, oil slick, or crossing the end line it's still fine to just check the center point and treat that as the value for the whole car. There's no need, for example, to check under all four corners and only treat the car as on grass or oil if some percentage of them are over that type of tile.

### 5.2.3 Laps with checkpoints

The track is set up as a single line snaking around the screen, blocking the player from driving over the goal until following the road to get there. If the wall behind the players were removed, leaving open road there, players could win by going in reverse (since you can edit the track file, you can even try this if you'd like). Because the game does not yet keep track of whether the player has followed the course's path we can't design a track that requires the players to drive multiple laps. For this exercise we're going to add support for laps by giving the game a way to keep track of whether the car is driving the right direction on the track, so that laps won't count if driven backward. Additionally, to utilize this new functionality, modify the game's track so that it's a loop. Start the cars behind the finish line, seal the corner of the map with wall tiles, and require 3 completed laps to win.

**Details:**

One of the most straightforward ways to track whether the player is driving laps in the right direction is to (invisibly!) mark two or more strips of track as distinct checkpoints in addition to the finish line (serving as a visible checkpoint line). Have each car keep track of the last checkpoint it crossed, for comparison to the one next driven over. When doing this a whole strip across the track needs to be marked so that the player can't drive past it without going over it. While in appearance and relationship to car movement these strips would be indistinguishable from normal road sections, they serve as markers to ensure the car has completed a full circuit before counting the finish line as another lap. A picture may help:




While the A and B sections in this illustration would be part of the track data, having their own numbers for use with collision detection, they would appear invisible, or rather as plain track. The C tiles would just look like the finish line, labeled here with a letter for ease of discussion. In order for a lap to count a car must cross A, then B, then C. This will prevent the car from earning laps by driving the course backwards, and also means the player can't just keep driving a little backward then forward over the finish line again to easily accumulate laps.

In code, one simple way to keep track of the last checkpoint driven over is to give each car another `int` - a `progressScore` variable - which resets to 0 when crossing C, 1 if the car drives over section A having last crossed C (meaning when

`progressScore` is 0), 2 if the car drives over section B having last crossed second A (`progressScore` is 1), and counts a lap when the car drives over C having last crossed B (`progressScore` is 2). Crossing any checkpoint out of order should set `progressScore` to -1, which can only be reset to 0 by driving back across C.

Even though these checkpoint regions will need their own `final int` values for the track definition, in terms of car collision code these values will need to be treated as equivalent to open road, so that cars don't collide with these as invisible walls. Look in the Player code for where checks are done comparing to plain road, and ensure that those sections will treat checkpoints the same (in addition to conditionally adjusting that car's `progressScore` variable!). This is also true for the finish line, which previously the code didn't need to treat as drivable area, since the moment either car collided with it the race was won. The checkered goal now also needs to be treated as road.

For this specific track layout, because it's a tube with no intersections, A and B as progress checks don't really need to be spread out very far from the finish line. Spatially A could be immediately following C, and B could be immediately prior to C, the directionality checks would work the same way. On the other hand for a more complex map, such as a crazy 8 with an intersection in the middle, more checkpoints would be necessary to ensure the player follows the right path to get credit for a lap.

### 5.2.4 Boost/nitro each lap

Giving cars a rechargeable speed boost to use each lap adds a bit of depth to the player's decisions while racing. Using it in the right part of the track for maximum effectiveness without losing control, or without causing a counterproductive collision (if/when car-to-car collision gets implemented, it's in another exercise!), is something besides just holding down the gas and taking corners.

Give each car a nitro at the start, and add one for each lap completed - meaning a player should be able to opt to not use it one lap to have two available during lap two, for example. Pick an additional key for each player to use for triggering nitro, which should very briefly give the car a considerable forward speed boost. Display on screen someplace with `text()` how many nitro uses each player currently has remaining.

**Details:**

There are three different tuning variables for the player car that affect how fast it drives: `DRIVE_POWER`, `MAX_SPEED`, and `GROUNDSPEED_DECAY_MULT`. The first variable, `DRIVE_POWER`, affects the car's rate of acceleration. `MAX_SPEED` is how fast the car can go before its speed gets capped at that value. The `GROUNDSPEED_DECAY_MULT` value is the most alien, but plays a central role in the car's speed: every frame the car's speed gets multiplied by this value to continuously lose some percentage of speed. At its default value of 0.94 that means every frame update (around 30 times per second) the car loses 6% of its current speed. Making that number even a little lower, in the ballpark of 0.7, severely restricts the car's movement because it's

constantly losing 30% of its speed, much too rapid a drain for speed to accumulate. Going the other direction, setting it to 1.0, means the car will never coast to a stop even after letting off the accelerator.

In order for the nitro boost to work, one or more of these values will need to be temporarily substituted for (or changed - though to do so you'll need to remove the `final` keyword where they're declared). Simply setting the speed value for the car won't last more than a frame, about $^1/_{30}$ of a second, before the `MAX_SPEED` will promptly cap it back to a lower number and over the next second `GROUNDSPEED_DECAY_MULT` will shave another 6% off it a few dozen times.

To temporarily use differently tuned numbers, it's necessary to keep track concretely of what we mean by temporarily: perhaps a new `int` value in the Player `class` for how many more frames the nitro will last. If the player presses the key to trigger nitro, and that player has at least one nitro currently in inventory (that also being tracked with a new `int` value in the Player `class`), decrease the number of nitro available while bumping that nitro timer up to some tuned number of cycles (30, as an example, would be about one second of boost). In the car's movement code, if that nitro timer is above zero decrease it by one. Also if that nitro timer is above zero, use the nitro versions of those three variables that affect the car's speed.

### 5.2.5 Car-to-car collision (basic)

Creating a realistic response between two automobiles colliding, even a simplification of a realistic response, is surprisingly tricky. There's a lot that we know from experience with cars and watching TV shows about how car brakes, suspension, orientation, and relative speed may cause cars in an accident to slide, spin, or recoil. We won't be worrying about that for the sake of this exercise - of course, no one's going to stop you if you're feeling especially driven to pull that off. But what we will be dealing with is that cars currently don't respond to collisions with other cars at all, they just move through one another as though the other car isn't there. That will not do.

For this exercise, simply detect when the two cars are too close to one another with a distance check between the center points. When the cars get too close to one another immediately return both cars to the positions they held just prior to the collision, and cut the speed of both cars in half. This won't be pretty, but it will definitely be an improvement over cars simply driving right through one another.

**Details:**

For finding the distance between the two cars, for the sake of practice you could write your own helper function to use the distance formula, although Processing has a built-in `dist()` function that does a fine job of finding how far apart two coordinates are from one another. Think about how many pixels long and wide the car body is when coming up with a starting estimate for what constitutes the cars being close enough to trigger collision response, though finding a distance that feels appropriate will likely take some iteration. Since this code

considers both car positions it may make sense for it to be handled someplace in the main Processing file rather than within the Player `class`, since the latter is really best suited to code that applies to each car independently. The code doesn't yet have a built-in record of the car's previous position for resetting it there upon colliding, but creating a few new variables and a function to do what's needed with them should be relatively straightforward. It may be tempting to just undo the car's last movement by subtracting the current speed vector, however since speed may be altered during the crash or from new key press events that approach isn't especially reliable and might lead to situations where the cars get stuck together.

If after implementing code that seems roughly right the cars don't move at all, or don't collide at all, the first thing I'd suspect may be off is the collision distance value that's being compared against how far apart the car centers are. If that number is tuned too large the cars may begin the race trapped inside one another's collision areas. If that number is too small the cars may pass through one another as long as the very center points remain far enough apart.

### 5.2.6 Single player option (random P2 movements for now)

Testing the car collisions can prove difficult with only one player to juggle keys to drive both cars. To help make the tests a bit more dynamic, let's create the option to have a simple, mostly random computer controller take the place of human Player 2. It doesn't need to actually run the race yet, it can just bang around into walls and turn occasionally. We'll take a look at making its actions more deliberate and sensible in a later exercise.

**Details:**

Create a `boolean` value to signify whether the car for Player 2 should abide by human control with the WASD keys or randomized computer control. In the keyboard handling code set up a key to toggle that `boolean` so that the computer control can be turned on or off during play. It would be better to decide that based on a choice made at a title screen, although at this phase of development no such screen exists, and it would be premature to force that in yet as another speed bump to click past every time we need to test the code.

The computer-controlled driver should only be able to manipulate its car with the same mobility functions in the `class` that the human players call when holding down keys. Directly manipulating position or angle could look very unnatural and means the collision code we wrote for walls and cars wouldn't play nice with its motions. Turn like the player turns, accelerate like the player accelerates, and brake/reverse just as the player does, too.

A natural but ineffective approach to making the movements random is to just decide by pure chance each update cycle what the car should do on that frame: gas, reverse, turn left, or turn right. The problem there is that because the code would alternate rapidly between these signals dozens of times each second, on average the car would just sort of wiggle in place, or perhaps lurch forward slowly and erratically since the forward acceleration is more powerful than reverse so on average it'd be due to win out.

Think about how the keys get pressed while playing: decide to press a particular key, hold it for a second or two, and then release. While holding forward or backward the human player will also commit to a turn left, a turn right, or driving straight. To recreate the decisiveness of this input we'll want to add a few more variables to the Player `class` specifically for use when it's AI controlled: how many more frames should pass before we rethink which keys are held, and which keys are held (one approach to handling the computer control for Player 2 might be to just outright hijack the P2 `keyHeld` set of variables, blocking them from being altered by actual key events in the keyHandling file).

### 5.2.7 Transition phase at start/end of stages

Winning the race *immediately* resets the round. There's no congratulatory message, no acknowledgment of which player won, and barely even any indication that the race has reset! Players should have at least a few seconds to catch up to the major event of the race being completed. Stop the cars when the race ends, since otherwise the race results could potentially change, and take a moment to display in a clear, readable way which car won.

**Details:**

Rather than handling game reset right in the code where either car completes its last lap, instead set a variable to indicate elsewhere in the code that the race is over. While this could be a `boolean` value, especially if you would like the win message displayed until either player clicks or presses a key to flip it and continue, if you would like the end screen to time out and advance automatically another approach is to treat it as an `int` count down which gets set to a positive number when the race ends. The car movement code could `return` prematurely if that `int` value is greater than zero. Each `draw()` cycle would simply decrement that value when positive, and within the same `if` conditional that checked if it was positive a check after decrement could identify if it has just that moment reached 0, meaning enough frames passed and the race reset functions can then be called.

For testing this function changing the number of required laps to 1 can save a pretty sizeable chunk of time. Since editing the map's text file is so easy to do it may also make sense to create an alternate test map that's just a short loop, or dig a hole out of the early barricades of the main map and reposition lap checkpoints so that the race can be finished much earlier. Before moving on it would be good to verify that the implementation of round end still plays out as expected on a proper 3 lap race, though for most of testing putting time into going around and around the racetrack alone just to see if it works and how the information is laid out so far would be silly.

### 5.2.8 Race time based in real-time

The `millis()` function in Processing provides the number of milliseconds that have passed since the program started. Use that function to display a real-time

race clock in the game. When a round ends show the winning driver's completed lap times, total race time, and average lap.

**Details:**

Creating a new `int` to keep track of the `millis()` value when the race started is necessary for the timer working on rounds after the first, since otherwise that `millis()` value will just keep on climbing once the program starts, independent of when the race resets. The time we display is then going to be the difference between the current `millis()` value and the one we saved at the start of the race, even though for the first race run on any given session that saved value will likely start at or near 0.

Surprisingly less straightforward than actually keeping the time is displaying it nicely on the screen in a somewhat clock-like format that we're used to seeing. There are Java functions available that can be used to handle the formatting for you, however as practice manipulating numbers and constructing a `String` I think it's worth giving this conversion a shot on your own with a custom bit of code. To get from milliseconds to tenths of a second is a division, but you'll likely want to also display seconds and minutes. The modulus operator % can be useful for getting a remainder, for example to figure out how many seconds are not accounted for by the number of minutes shown.

## 5.3 Challenge Exercises

### 5.3.1 Ramp tiles and airborne cars

Jump blocks can be used to add an element of verticality and danger to the game. Whereas ramps would only work from one side these should work from any direction, so depicting them from overhead as a pyramid shape makes the most conceptual sense. When a drives over it send it "upward" in which condition several things will be different for the car:

1. Draw a black, partially transparent shadow the shape and orientation of the car where the car would be if it were at ground level
2. Display the actual car graphic vertically offset above that shadow, so that the higher it is in the air the more distance there is between the shadow and the car (the shadow should be drawn in code before the car gets drawn, so that if the two overlap the car shows as on top rather than the shadow covering part of the car)
3. No collision reaction with any walls, grass, oil slicks, or other obstacles while it's in the air (checkpoints and goal line should still count)
4. While airborne the car shouldn't be able to gain or lose speed, and should be unable to turn
5. From the moment it begins its upward ascent its downward speed should begin gradually increasing, so that the car lands not long after it hits the ramp

6. The faster the car is going when it hits a ramp, the higher it should be tossed into the air

If the player's car leaves the screen or lands atop an obstacle tile it should lose any checkpoint progress for the current lap and reset to its starting position on the track.

To test this feature add ramp blocks to a section on the main map. Make sure the checkpoints are placed in areas of the track that won't be skipped by using the ramps to take shortcuts. Ideally the placement and tuning of ramps near walls means they aren't safe or consistent to take every time, yet are viable enough to sometimes risk if going fast enough and straight on.

**Details:**

Resetting the car to its start position can be done easily since the Player class already includes a `reset()` function and saved `startX`, `startY` from when the map got loaded. If additional initialization code has wound up in that reset function while completing other exercises, for example clearing the car's lap count or toggling whether it's computer controlled, a `boolean` argument might be added to the `reset()` function for `fullReset` to indicate whether the call is happening at race start or from crashing on a wall. The variable used to track checkpoints progress of course should be reset for either occasion, since otherwise crashing after getting to the second checkpoint would provide a shortcut to complete the lap.

`height` and `verticalSpeed` are both new variables needed in the Player class. They'll be used much like the position coordinates and speed value. Add `verticalSpeed` to `height` each frame, and add some very small negative value to `verticalSpeed` each frame to simulate the effect of gravity. If after the additions `height` goes below 0, cap it at zero, meaning ground level. Upon detecting collision with the ramp set `verticalSpeed` to a modestly large positive number, which will give the car an upward boost if the rest of the related code is working as expected.

Both cars can use the same shadow image, but it will still need to be made in an image editor and loaded as its own `PImage`. The shadow image should just be a black rectangle the same dimensions as the car (no need to deal with the headlight glow stuff), ideally partially transparent. While the car is in the air, vertically raised by some amount based on its current height value above the ground, the shadow will help the player see what the car is lined up with at ground level.

In the World file there's a function called `whatIsAtThisCoordinate()` that is expecting to only handle valid coordinate values that are within the screen space (both greater than or equal to zero, `x` less than `width`, and `y` less than `height`). That worked well enough when walls could be used to prevent players from driving off the screen. Now that the ramp block can be used to potentially go off screen an adjustment to that function needs to be made so that the coordinates are verified as on screen before used to calculate tile array lookup indices, otherwise return the `final int` for a solid wall tile.

### 5.3.2 Better computer driver

Just as the invisible checkpoints can be added to the track for the game to have additional information not exposed to the player, invisible waypoints can also be added to the map to let the computer driver know where to drive in order to complete laps. Pick a range of higher numbers, for example 50-75, and starting at 50 up to however many are needed set them in the map file in positions that will allow the computer driver to navigate simply by a "connect the dots" approach: driving from each one to the next higher in order until no higher waypoint is available, at which time it will repeat the loop. Rather than having the computer-controlled car randomly slam around into walls, have it switch between different input keys in an effort to guide itself to the next waypoint on its list. It should be able to complete laps on its own, to even win the race if left alone.

**Details:**

Unlike the checkpoint strips on the course, these aren't needed during play and can be replaced with a plain ground's `final int` during level load right after the waypoint information gets collected into an `ArrayList` for use by the computer driver. The mathematical concept of a "dot product" will be useful in determining whether the car needs to turn left or right to point more toward its next waypoint. A practical usage of the dot product is to figure out which side of a line a point is on. That can be used in relation to a line to the car's side or front to answer questions for the computer controller such as, "am I currently facing toward or away from the waypoint?" and, "is the waypoint to my right or to my left?" While troubleshooting your code to get the car to follow the waypoint chain it will be useful to surface or visualize that information in a `text()` call, or to output information to the console with `println()` such as when the car is changing waypoints.

Detecting whether the computer-controlled car is ready for the next waypoint should be done by whether the car's center point is within a fairly forgiving distance of the waypoint.

### 5.3.3 Scrolling for larger track, Pt. 1 (1 player only, for now)

The track presently has to be the size of the screen. Any larger and the cars would be able to drive out of view. We can implement camera scrolling to alleviate this. First increase the track size in both dimensions by a half dozen tiles, to give the game room to scroll. Have the screen shift to center over Player 1's car, except prevent the adjustment from going so far that the area beyond the track can be seen. For now don't worry about Player 2, we'll work on accounting for that second car next.

**Details:**

Increasing the track's overall dimensions is a matter of updating how `worldGrid` gets initialized in the World file. Look for `int[15][20]`. In that declaration of the

2D array it is being set as 15 units tall and 20 wide. As a warning: even though most horizontal and vertical coordinate values list the horizontal, x or width, before the vertical, y or height, for 2D arrays the order is reversed. That happens because technically what `[15][20]` signifies is 15 arrays of 20 arrays, which in an older programming language like C might mean 15 words (rows) that are each up to 20 letters (columns) long. More confusing still is that much like latitude lines are drawn east to west but measure north to south, row numbers extend horizontally and are therefore used to measure vertical offset in the 2D array. Don't feel bad if this seems a bit hard to keep track of. When in doubt it's always an option to write a simple test program to double-check which way is which, or to refer back to some of your past code that already works to figure out how you did it there. Anyhow, back to this particular exercise: to make the track big enough for scrolling will involve increasing that 15 and 20 to larger numbers, plus adding the right number of additional rows and columns to the map text file to fill in for the newly padded dimensions. Be careful to keep straight here which dimension is which, since not defining enough rows or columns in the track text file when it gets loaded may produce an error that'll be tricky to debug since the problem isn't really in the code but is at that point in the data file being loaded.

Create a couple of new global (outside of any classes, for now at least) `float` values named `cameraOffsetX` and `cameraOffsetY`. Think of these variables as representing the distance in pixels that the camera/screen's top-left corner has panned along either axis, so that when both are 0.0 the camera is in the top-left of the track, and as both increase the view should move down and right toward the track's center. While we *could* manually subtract these values from the x and y draw position of every object, so that as we increase `cameraOffsetX` for example all objects would shift their drawing positions that amount left, there's a better way to do this. In `draw()` before any draw code we want moved with the camera call `pushMatrix()` and `translate(-cameraOffsetX,0.0)`, then after that but before draw code for anything that we don't want moved with the camera (interface elements, like screen text or nitro and lap indicators) call `popMatrix()`. An example of this for horizontal camera movement can be found in the `Platformer` example in `commonBonus/free-from-hobbygamedev.com`. `pushMatrix()` and `popMatrix()` are used to combine then forget graphics transformations (sliding, rotating, stretching) that affect any other graphics calls up until the `popMatrix()` call gets reached. No other code besides graphics calls gets affected, so it's fine to perform other game update logic in-between as well. Even though as described here the camera is only moving horizontally, to move vertically as well will just be a matter of replacing the `transform()`'s second argument, 0.0 above, with `-cameraOffsetY`. `transform()` is just the graphics term for movement that's side-to-side or upward and downward, without rotation or resizing.

The other half of this problem is figuring out how to update `cameraOffsetX` and `cameraOffsetY` each frame. The `Platformer` example mentioned earlier includes an `updateCameraPosition()` function that shows an example of centering the camera offset coordinates on the player. Because we use the `cameraOffsetX` and `cameraOffsetY` values as the top left corner of the screen, and want the player's car

in the center, we need to set those values each frame to the position of the car minus half the screen's width and half the screen's height. In the lines that immediately follow that update, before the newly changed number gets used to set screen translation, we bound it by checking if either is less than 0, and setting it to 0 if so to avoid going past the left or top edge, and if either is greater than the track's total width (tile size times tile count along each dimension) minus the screen width (since, again, it's the left edge of the camera/screen we're setting) then we force it back to that number, doing the same accordingly for the vertical axis.

A more complete follow camera functionality showing both horizontal and vertical is available as the `sketch_gridCollide_better` example code provided in the `commonBonus/free-from-hobbygamedev.com/lab-examples` directory.

### 5.3.4 Scrolling for larger track, Pt. 2 (2 player split-screen)

When Player 2 is set to be computer controlled, we'll want to keep the same full screen camera movement introduced in the previous exercise. When Player 2 is switched to be human controlled, instead we'll want to use split screen so that each player gets half the viewing area. Player 2 should have their own camera tracking variables, and be given either the bottom half or right half of the screen while Player 1 gets the other half.

**Details:**

The main concept needed here is the `PGraphics` buffer, which we can draw to with `image()` calls on it as though it were a separate screen, then copying it to the screen with a simple `image()` call containing it as though it were an image file referenced through a `PImage`. That all sounds confusing... even to me, and I just wrote it! Check out the `pGraphicsHalfScreen` source code in the `commonBonus/mini-demos` folder, this is one of those concepts that is much easier to make sense out of by looking at and tinkering with an example than by reading or talking about it.

As a classic tile engine optimization to explore, not all that critical for now but important if the track sizes begin to extend well beyond their current camera movement testing tile dimensions, think about ways to only draw the tiles that are within the camera's view. Since by definition when this works it's impossible to tell from the display that it's being done right, one way to test this sort of optimization is to write the code as though the screen is one tile smaller along each axis, to see the tiles vanishing at either edge of the screen as the camera pans. Once it's working, add back in the missing tile width, and the game should perform just as fast no matter how large the track gets.

### 5.3.5 Rain weather mode, with slippery driving/drifting

Having weather effects in a racing game is a great way to get more mileage out of the tracks that you've already defined. Create a raining `boolean` value, and set it true some percentage of the time when a track loads. When the raining variable is true during play, every frame darken the whole screen, draw constantly changing

rain drops, and affect car behavior so that it's more like driving in wet conditions (feel free to exaggerate the effects of rain, realism here might be less fun than making it more noticeable how the driving is different).

**Details:**

Remember how for the nitro boost we created alternate tuning values for the cars? For the most part rain road conditions can be handled here the same way. When it's raining make acceleration slower, make the brakes less responsive, reduce ground friction (by making speed decay closer to 1.0), and find a way to make turning a bit messier (easier to oversteer by putting momentum into the turning, or find a way to support drifting around corners).

To make it rain only some percentage of the time, set the rain `boolean` at track start time with a comparison between a random value from 0-100 and the number you want as your percentage. For example, to rain 30 percent of the time, `isRaining = random(100)<30;` can do that, since a random value from 0-100 will be less than 30 about 30% of the time it's called.

For darkening the whole screen, drawing a mostly transparent black rectangle over the screen will generally work well enough. For rain drawing many tiny short angled lines randomly around the screen that are a subtle but noticeable raindrop color will do the trick. Having all rain slant at the same slight angle can help it look convincing, since even though we're not tracking rain drop positions between frames or really moving them the slant will help suggest movement to the observer.

### 5.3.6 Multiple tracks

Now that the game supports scrolling camera, a couple new types of tiles, a computer controlled car for single player mode, and we can count laps, the game is finally ready for a set of different track layouts. The code is already set up to load an external track file, gameTrack.txt, in the `loadMapLayout()` function within the World tab. Supporting more tracks will mean just a few additions on top of that:

1. Either have an array of different track file names, or alter the map loading code to support having multiple tracks defined in the same text file (to load, for example, the 3rd map out of 6 that are typed in the same file), whichever you find easiest.
2. Determine how large you want your maximum map, since you'll need to either make them all that size in their text file definitions or otherwise change how `worldGrid` gets set up to account for variable track sizes in memory. I'd recommend doing the latter to keep thing simple - worst case scenario if you want some maps to be smaller than your maximum size, it's perfectly fine to not use the entire possible track area on each level. If it's important that the camera doesn't scroll past the edges of smaller stages, so that they feel just as tight as the larger levels near the world boundaries, you could even add some special signifier tiles to the map definition that your game can interpret while opening the map as the camera's scrolling limits.

3. Give the player some way to select or cycle between the various track files. Using a few extra keyboard keys to go to next/previous stages, letting them click on different regions (tiny thumbnail track screenshots?) on a simple title screen, or randomly picking a new map at the end of each race would all work. For testing purposes while developing a map it will be helpful to automatically load that particular map right at start each time, but no matter how the list of maps gets hooked into the game that should be easy enough to hack in with a temporary line of override code while tinkering.
4. Design the tracks!

**Details:**

The coding parts of this step aren't especially demanding. The challenge here is hidden in those last 3 words: "Design the tracks!" Level design is a separate discipline from game programming, much like art or audio creation it's a form of content authoring that involves separate values, processes, best practices, and ways of thinking. It's worthy of practice because it's an aspect of game development that affects many genres of games, not only racing tracks but of course platformers, corridor action games, even games without obvious architectural space such as puzzle games and scrolling sh'mups since level design there requires figuring out what game pieces come out in what order (or by probability, it partially random), pacing, and so on.

A couple of tips for level design. First, try to think of ways make maps very distinct from one another, in gameplay and also in overall appearance. Do you have a level that uses very wide roads, or very narrow ones? One that is mostly straightaways, or has lots of curves in the path? Some maps that are the size supported and also a few that are the smallest that can still be practical? Don't feel pressured to include every feature and tile type you have in every map, sometimes what can help set a map apart is it's the one that's the most simple and straightforward, without any grass, ramps, or oil slicks and minimal decorative wall types, whereas another course may take place almost entirely on grass, or consist entirely of ramps between walled off sections.

Lastly, remember the huge power that you wield by being not only the level designer for this game but also the programmer and artist: is there a level that you think would be cool to have but it would require a new tile feature, or at least another type of decorative tile to establish the setting? You're always free to dip right back into the programming to add any new types of tiles or gameplay functionality that your level designs require. The only limit is what you can think up, and what you're willing to put the time into figuring out how to do.

As a simple intersection between your programming ability and level design desires, note how as the game is now the cars always start facing north. There's not yet a way to design a map with cars pointing any other direction! What an arbitrary constraint. That was fine when there was only one map and north was the right way for them to point, but not with more maps it'll help to support cars that begin pointing in a different direction. Although there are many different

ways to do this, one approach that's especially appealing because it doesn't require and new tile types or changes to the level format is for the cars to automatically detect which direction the goal line is after a map gets loaded, then point towards it since by convention race cars start behind the checkered goal line.

Other features that can be added to your level designs may involve values or words outside of the actual level grid. Remember how earlier when we supported different tile sets for time of day, season, or environment we suggested that these could be set per level? Just having one number on the first line after the level's grid gets defined can be enough to specify which art theme to load. Alternatively, putting a word there could be used as a filename prefix for a theme, making it possible to create new art themes for new levels without even needing to revisit changes to the code.

Another way to combine your programming practice with your level design practice is to get a head start on the next exercise...

### 5.3.7 In-game track editor GUI[19]

By this time in the game's development there are quite a few tile numbers to keep track of when making a map text file. Multiple values are needed for player start positions, invisible checkpoints, computer car waypoints, and that's not even touching the many varieties of obstacles and decorations. More, if you've found yourself wanting to extend the game engine while designing levels.

Your game engine already supports loading the track art, loading map files, and displaying the two with scrolling camera. For this exercise, create a level editor mode in the game that allows you to open a track file, use mouse interactions to place track elements, and save the track file.

**Details:**

Besides making it vastly easier to design a level with aesthetic considerations in mind, this also gets rid of those pesky problems that inevitably happen while hand editing map files, like forgetting a comma, leaving out a value, or otherwise accidentally breaking the format expected by the program.

The same way that we handled split screen for Player 2 using a `PGraphics` buffer can also be employed to separate a track view pane from a track palette, where the player can click to switch between different track pieces to place when clicking. Matching the mouse click to the track position under it can be done in the same way as the car's collision code, however the `cameraOffsetX` and `cameraOffsetY` values used to `translate()` the screen's graphics have no effect on `mouseX` and `mouseY`, so you'll need to manually account for those offsets in deciding which coordinate to check when the mouse is pressed and the camera is scrolled from its top-left position.

[19] Graphical User Interface

## 5.4 Write From Scratch Steps: Racing

This Racing example is a bit more sophisticated than the earlier game examples. Accordingly, these steps will be a bit higher level in places, trusting that if you're taking this one on you have already gained some comfort with the basic patterns. In other places the steps indicated will be more specific, spelling out variable names or arguments for a function to accept, to reduce the potential for ambiguity or misinterpretation when later steps refer to variables, classes, and functions introduced in earlier steps. This game's code also involves an added layer of organization, which means it will be more steps in (8!) this time before we're really ready to press Play and test what the application has been doing. All that said: you can do it! Good luck, and don't get discouraged if (when!) errors are encountered along the way, that's just part of programming. Figuring out how to diagnose and repair the compiler's feedback (or unintended results on screen) is perhaps the most important kind of practice as a game programmer. Embrace it!

1. Create a new, empty Processing Sketch.

2. Set window size to 800x600.

3. In other programs so far that involve images moving around the screen that can be rotated around their center position, each time we've duplicated the related `pushMatrix()`, `translate()`, `rotate()`, `image()`, and `popMatrix()` code. I kept reusing it in each case because that code is inherently awkward, though nevertheless worth getting used to since it relates to other tricks like how we handle camera offset. Now that we've gained some familiarity with its inner workings, let's go ahead and hide it in a helper function so that we can avoid spending any more mental energy on it in this project. Create a new file named drawRotatedBitmap (to make this easy to include in future projects) and define in it a function `drawBitmapCenteredAtLocationWithRotation()` that accepts as arguments `PImage graphic`, `float atX`, `float atY`, `float withAngle`, to draw the `PImage` passed to it centered at `atX`, `atY` and spun around its center by `withAngle` radians.[20] Even though the program isn't as a whole ready to really test yet, testing this function on its own by calling it a few times in the main source file with temporary `PImage` variables at different positions and angles would be prudent before advancing. Otherwise if will be hard to tell later if the image rotation doesn't work - whether the problem is from how the key presses are being handled, whether the logic is being called, or if the problem is buried in how this helper function is written.

[20] Radians, in case you're more familiar with degrees, is the default unit expected by many trigonometric functions. Whereas degrees divide all possible angles into 360 equally spaced directions, radians are a measurement around a circle in proportion to the circle's radius (thus the name). So in radians $\pi$, predefined as `PI` in Processing, is 180°, $2\pi$ radians is 360°, $\pi/_2$ is 90°, etc.

4. Create a keyHandling file that tracks four `boolean` variables for each player, to be used for Player 1's and Player 2's gas, reverse, left, and right. Also in this file check the `keyCode` value for arrows (Player 1's control keys) and `key` value for letters (Player 2 can use WASD as though they were arrows), setting the corresponding `boolean` to `true` when the corresponding input is detected in `keyPressed()` and back to `false` if detected in `keyReleased()`. As a way to ensure these key mappings don't fall out of sync, write a helper function in that file, `setControl(boolean setTo)`, which can be passed `true` in `keyPressed()` and `false` in `keyReleased()`, so that both can use the same `key` and `keyCode` checks.

5. Create a minimal Player class that at this time only has `float` variables for `x`, `y`, and `ang`, as well as a `PImage` called `myGraphic` for its image. Give the class a constructor[21] that accepts a `String` for which image filename to load as `myGraphics`, and also in that constructor set `x` and `y` to random coordinates within the screen's visible area (ex. set `x` to `random(width)`). Define a function for the class, accepting the following arguments:

```
void considerKeyboardInput(boolean gasKey,
                boolean reverseKey, boolean turnLeft,
                boolean turnRight)
```

For now this function can ignore the `gasKey` and `reserveKey`, but when `turnLeft` is `true` decrease `ang` by 0.045, and when `turnRight` is `true` increase `ang` by 0.045. In order to avoid those two numbers falling out of sync, define a `final float` in the Player `class` titled `PLAYER_TURN_RATE` and set it to 0.045, so that the label can be used in both locations.

6. Add another function to the Player `class`, `drawCar()`, which calls the `drawBitmapCenteredAtLocationWithRotation()` function using the `class`'s variables for graphic, `x`, `y`, and angle values as its arguments.

7. Back in the main source file, declare two variables of type `Player`, named `p1` and `p2` for Player 1 and Player 2 respectively. In `void setup()` right after setting screen dimensions with `size()`, initialize each to a new `Player()` instance, passing each the name of an image (file extension included) in the data directory for an overhead view of a small car pointing right.

8. In `draw()` call `considerKeyboardInput()` on each car, passing in as the four variables the `boolean` values connected earlier to that car's control keys. After those functions call `drawCar()` on each car. Since there's no track being drawn yet to conceal the previous frame's draw operations, call `background(0)` as the first line in `draw()` to avoid seeing a smudge when the cars spin in place.

[21] A constructor is just a function in a `class` that has the same name as the `class` it's in, and no return value (not even a `void` specified), which is called automatically on each instance of the classes at the moment it gets created.

*Feeling a bit lost at this stage? Something not quite working as you think it should, but having trouble sorting out where the error might be? Feel free to compare your solution so far to this one that I put together as a checkpoint up to this step in the rewrite process for Racing, titled* `racing_rewrite_step8` *and available in this directory within the materials:*

```
classic-games/example-solutions/racing-remake-checkpoints/
```

*If at all possible, try to find where you've made an error in your solution, rather than simply copy/pasting code from this or other answer checkpoints, and rather than simply starting fresh here from a duplicate of this directory. It'll help you to learn to identify what errors habitually arise from the ways that you think through typing code, so that in the future when doing your own project and my answer key isn't handy you'll be able to spot and fix (or avoid!) those issues on your own. Our ultimate goal here together is NOT to simply put this Racing game back together like a model airplane, but to learn what you'll need to know to better take on the original ideas that you dream up.*

9. So far so good? All caught up? Great! Let's next give our cars a way to drive forward or backward. While that may sound simple, we're going to introduce a handful of supporting related variables, tuning values, and functions at once for this step. First we'll want four more `final float` tuning values in the Player `class`: one called `DRIVE_POWER` (set to 0.25) for adjusting how quickly the car accelerates, one that's `MAX_SPEED` (set to 3.0) to limit how fast the car can go under ideal conditions, `GROUNDSPEED_DECAY_MULT` (set to 0.94) which we'll multiply by the car's speed every frame so that it gradually rolls to a stop without gas applied, and `MIN_TURN_SPEED` (set to 0.3) which we'll use as a comparison when deciding whether the car is moving fast enough to turn (cars can't turn without moving - without something like this added to our code it would move and act more like a tank or bulldozer!). We'll only need to add a single `float` variable that will change during gameplay - `speed` - which we'll use to keep track of how fast forward or backward (negative values) the car is moving in relation to the direction is pointed.

Let's take a quick time out here to discuss why we want a directional speed value (`float speed`) and angle (`float ang` set up in an earlier step), rather than separate horizontal and vertical speeds as we've done in previous games (`ballSpeedX` and `ballSpeedY` from Brick Break's code, for example). A bouncing ball, unlike a car, does not have a sense of directionality. True, a ball does move in a direction of course, but it does not *face* in a direction in the way that a car does. Without a facing, a ball cannot change which direction it's facing while it's in motion. Since a car is a wheeled vehicle it can move much, much more easily forward or backward than it can side to side, so much so that when the car turns to change which way it's facing, it's simultaneously changing the direction of its

movement. If a car is going 40 mph north and turns due east in a controlled fashion, the car is then moving 40 mph east.

In other words: by keeping the car's motion as speed and angle, rather than the horizontal and vertical components of its velocity, it's much easier for us to keep the car moving forward when it turns, which better reflects the behavior expected of a wheeled vehicle.

10. Next we need to add a couple of new functions to put the `speed` variable and those tuning `final float` values to use. The first will be `gas()`, which will accepted as an argument a `float` called `directionScale`. We'll use that `directionScale` as a way to determine which way the car is moving and with what percentage of its forward drive power. In other words, if we drive forward at 100% with `gas(1.0)`, when we can brake or reverse with 30% power by calling `gas(-0.3)`. The guts of the gas() function are pretty simple:

```
void gas(float directionScale) {
  speed += DRIVE_POWER * directionScale;
  if(speed > MAX_SPEED) {
    speed = MAX_SPEED;
  }
}
```

We multiple the `directionScale` value passed in times `DRIVE_POWER` and add it to the `speed` variable, capping the speed at `MAX_SPEED` in case we've exceeded it with this latest increment. We'll hook the `gas()` function into the `considerKeyboardInput()` function with these new lines:

```
if(gasKey) {
  gas(1.0);
}
if(reverseKey) {
  gas(-0.3);
}
```

11. Even though we're changing the speed variable, until we do something to the `x` and `y` values that we use as the car's display position, the cars can't actually move. What we need is another function in the Player class, called `updatePosition()`, which we'll use to increment the car's position by its current `speed` in the current direction (`ang`) that it's pointing in, in addition to decaying the car's `speed` by multiplying it by our `GROUNDSPEED_DECAY_MULT` tuning value:

```
void updatePosition() {
  x += cos(ang) * speed;
  y += sin(ang) * speed;

  speed *= GROUNDSPEED_DECAY_MULT;
}
```

Remember by the way that the order these functions get defined within the class is not important, as long as each function is given its own space, meaning not overlapping or contained within the definition of another. If you want to put the definition of `updatePosition()` at the end of the Player `class` or at the start of it, as long as it's between the `class`'s braces `{ }` then it's going to be accessible in the same way and perform the same operations. That's of course because functions don't happen in the order they get defined in, but rather in the order they get called in. Speaking of which, on the line before `p1.drawCar()` in the main file's `draw()` function, add `p1.updatePosition()`, and on the line before `p2.drawCar()` also (and right after p1's) in that file's `draw()` function, add `p2.updatePosition()`, so that the position updates are getting called on every update cycle, too.

12. Just as we used `gas()` as a helper function for forward and backward movement, let's also add helper functions for `steerLeft()` and `steerRight()`. Initially these functions can begin with only the `ang` increment or decrement. To incorporate our new `MIN_TURN_SPEED` tuning value, compare whether `abs(speed)` is greater than that tuning threshold, and otherwise don't allow the turn to take place. We're using that `abs()` for absolute value because we only care whether the car is moving when it turns; we aren't concerned with whether it's moving forward or backward at that speed. So we've now changed our `turnLeft` and `turnRight` conditionals in the `considerKeyboardInput()` function to:

```
if(turnLeft) {
  steerLeft();
}
if(turnRight) {
  steerRight();
}
```

And implemented the two helper functions outside of that function, though also in the Player class, like so:

```
void steerLeft() {
  if (abs(speed) > MIN_TURN_SPEED) {
    ang -= PLAYER_TURN_RATE;
  }
}
```

```
void steerRight() {
  if (abs(speed) > MIN_TURN_SPEED) {
    ang += PLAYER_TURN_RATE;
  }
}
```

Now the cars are able to drive forward and backward, and they can steer left to right when moving above a minimum speed.

*I've saved another checkpoint as* `racing_rewrite_step12`*:*

```
classic-games/example-solutions/racing-remake-checkpoints/
```

*At this point our focus can shift to getting the world map working.*

13. Create a new file/tab, and call it World. Define in it two final int values that we'll use as the world's tile dimensions:

```
final int TRACK_TILE_WIDTH = 40;
final int TRACK_TILE_HEIGHT = TRACK_TILE_WIDTH; // i.e. also 40
```

Let's also define in the World tab 5 different numbers that we'll use as codes to distinguish the different sections of our map:

```
final int TRACK_TYPE_PLAIN_ROAD = 0;
final int TRACK_TYPE_GOAL = 1;
final int TRACK_TYPE_P1_START = 2;
final int TRACK_TYPE_P2_START = 3;
final int TRACK_TYPE_WALL = 4;
```

Lastly, we'll declare and set up a 2D array with some values to get us started. I'm making this array 15 rows tall and 20 rows wide, because our tiles are 40x40, as per the first two final int values we defined in this write from scratch step, and our game's window size in pixels is 800x600. 15*40 is 600, 20*40 is 800, so this should cover the whole screen.

```
int[][] worldGrid = { { 4, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 4, 4, 4 },
                      { 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 4, 4 },
                      { 4, 0, 0, 0, 4, 4, 4, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 4 },
                      { 4, 0, 0, 0, 4, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0 },
                      { 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },
                      { 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 1, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                      { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 0, 0, 0, 0, 0, 0, 0 },
                      { 4, 4, 4, 0, 0, 0, 0, 4, 0, 0, 0, 4, 4, 0, 0, 0, 0, 0, 0, 0 },
                      { 4, 4, 4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4 },
                      { 4, 4, 4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4 }};
```

The placement of the 4's, 0's, 2, 3, and 1's aren't terribly important at this point, they're just example input. When I first made that grid I did so by copy pasting it in parts, initially filling out the 20 wide and 15 high in 0's only, then I went back and put in some scattered 4's to help me verify in these coming steps whether it's oriented and lined up as expected when used to draw a track from. As we move forward we'll be loading all those numbers out of an external text file rather than defining them directly in code, but this works for now while we're just getting map functionality started.

14. Next we'll define `worldDrawGrid()` in the world tab, and call it first thing in the main file's `draw()` function instead of using a `background(0)` call to wipe the screen's previous frame. The `worldDrawGrid()` function needs to go through each row and column of the 2D track grid, and for each position check the number there. Based on the number found, a different fill color will be set for the next square drawn. The squares are drawn by multiplying the grid's row and column indices by the tile's dimensions (40x40), in effect letting us display our small numerical grid stretched over the entire screen as a key for how to colorize track sections. As we move forward, we'll be able to check what map section is under a car's position, and affect it's movement or the gameplay logic accordingly.

Here's the function:

```
void worldDrawGrid() {
  for(int row=0;row<worldGrid.length;row++) {
    int tileTL_pixelY = row*TRACK_TILE_HEIGHT;
    for(int col=0;col<worldGrid[row].length;col++) {
      int tileTL_pixelX = col*TRACK_TILE_WIDTH;
      switch(worldGrid[row][col]) {
        case TRACK_TYPE_PLAIN_ROAD:
          fill(200,200,200);
          break;
        case TRACK_TYPE_GOAL:
          fill(255,255,0);
          break;
        case TRACK_TYPE_WALL:
          fill(0,0,255);
```

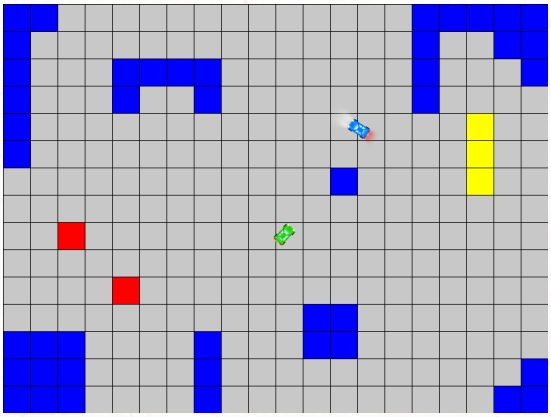
```
        break;
      default: // unexpceted tile type, mark it bright red
        fill(255,0,0);
        break;
    } // end of switch
    rect(tileTL_pixelX,tileTL_pixelY,TRACK_TILE_WIDTH,TRACK_TILE_HEIGHT);
  } // end of for col
 } // end of for row
} // end of function worldDrawGrid
```

Those different case keywords are really just the `final int` numbers we defined in the previous step. Because we defined `TRACK_TYPE_WALL` as 4, that case is really just saying that if a 4 is found at some location in the grid, set the fill color to blue before we get to that `rect()` call.

It's tempting to glaze over some of that math near the top of the nested `for` loop, but that's really what's making this whole thing work, so it's worth taking a closer look at it to try to make sense of what it's saying.

Again: don't forget to call this function at the start of `draw()` instead of `background()`, or we still won't see the map anyway.

Why are `TRACK_TYPE_P1_START` and `TRACK_TYPE_P2_START` not cases in that world map drawing loop? They aren't intended for in-game track information, they're just meant to tell the cars where on the grid to start. Let's set that up next. In the meantime, they show up as bright red for unhandled numbers:



15. For this step we're going to write `loadMapLayout()` in the World file, and call it at the end of `setup()` (make sure it's *after* the lines setting up p1 and p2 as new `Player()` instances!) over in the main source file.

`loadMapLayout()` will scan over every row and every column, checking every tile in turn, and where it finds the number corresponding to a Player 1 or Player 2 start position it will do two things: (1) position the right car in the middle of that tile (and 2) replace that spot in the 2D array with the value for empty road, so that it will be drivable space when the game starts.

```
void loadMapLayout() {
  for(int row=0;row<worldGrid.length;row++) {
    for(int col=0;col<worldGrid[row].length;col++) {
```

```
        if(worldGrid[row][col] == TRACK_TYPE_P1_START) {
          p1.x = col * TRACK_TILE_WIDTH + 0.5 * TRACK_TILE_WIDTH;
          p1.y =  row * TRACK_TILE_HEIGHT + 0.5 * TRACK_TILE_HEIGHT;
          worldGrid[row][col] = TRACK_TYPE_PLAIN_ROAD;
        }
        if(worldGrid[row][col] == TRACK_TYPE_P2_START) {
          p2.x = col * TRACK_TILE_WIDTH + 0.5 * TRACK_TILE_WIDTH;
          p2.y =  row * TRACK_TILE_HEIGHT + 0.5 * TRACK_TILE_HEIGHT;
          worldGrid[row][col] = TRACK_TYPE_PLAIN_ROAD;
        }
      }
    }
  }
```

Provided that we remember to call this function in `setup()`, it should replace where the red squares were showing up with Player 1 and Player 2's cars, no longer in random positions now at start.

*Stuck? Compare to the checkpoint* `racing_rewrite_step15` *in:*

```
classic-games/example-solutions/racing-remake-checkpoints/
```

*Next: collisions with walls and goal line, then loading the map from a file.*

16. When we're drawing grid positions to the screen as track tiles, in order to determine where to draw each tile we multiply the grid number's row and column position by the tile's dimensions. The tiles are defined as 40 pixels wide and 40 pixels tall. So if there's a number 4, which we treat as a wall section, 6 values in from the left side of the grid, then 6*40=240, so drawing the square for it will happen 240 pixels from the side of the screen.

The opposite process also works, when we need to go from a pixel value on the screen to figure out what grid number matches up to it. If the car is driving near the top, left-middle of the screen, and is 244 pixels from the left edge, we can divide by 40, which is 6.1, or without the decimal part (since there's no way to check the grid for a value at position 6.1), position 6 in the grid as described above would be 4, which we treat as a meaning wall. In response to finding that the car has driven on top of a wall section, we could undo its movement, putting it safely back on the road that it was on before it drove into this new tile section.

If you're thinking like I was the first time I was trying to make sense of all this, the numbers involved may seem overwhelming. But the code actually makes it easier for us, because it gives us a way to not deal with the numbers directly, so that we can instead label the numbers we're working with, and then just reason through how we're using those labels. Take for instance how in the previous paragraph I mentioned 4 twice as meaning wall. In the code, we don't actually need to type 4, and we certainly don't need to remember that 4 corresponds to

wall, because instead can just check if the number in the grid under our car corresponds to what we set as `TRACK_TYPE_WALL`.

Nor do we ever actually have to do any multiplication or division, we instead only need to understand how we can use multiplication and division as tools to convert our coordinates from the numerical grid to track squares on the screen, or back from coordinates on the screen into positions on that numerical grid. We do the former to draw the track, and the latter to check for collisions.

What we want now is a new function in the World class that we can wrap up this concept into, so that rather than needing to worry about how or even why we can go from screen coordinates to a grid position, we can just go straight from a screen coordinate - say, where a car is - directly to the number that's under it in the corresponding number grid. This is precisely that function:

```
int whatIsAtThisCoordinate(float someX,float someY) {
  int someTileRow = (int)(someY / TRACK_TILE_HEIGHT);
  int someTileCol = (int)(someX / TRACK_TILE_WIDTH);

  return worldGrid[someTileRow][someTileCol];
}
```

We pass in pixel coordinates, `someX` and `someY`, and it returns as its value whichever track part number is in grid position corresponding to it. So how do we put this one to use? Unlike those other World functions, which we called in the main code path of `setup()` and `draw()`, this one we'll tie closely with the `Player`'s car functionality. Take a look at how we're going to rewrite `updatePosition()` in the Player `class`:

```
void updatePosition() {
    float newX = x + cos(ang) * speed;
    float newY = y + sin(ang) * speed;
    int tilePlayerHit = whatIsAtThisCoordinate(newX,newY);

    if(tilePlayerHit == TRACK_TYPE_PLAIN_ROAD) {
      x = newX;
      y = newY;
    }
    speed *= GROUNDSPEED_DECAY_MULT;
  }
```

Now we don't actually change the player's position unless where the car will be next is on open road. Otherwise the car just stays where it was, which prevents us from being able to drive through walls.

It's worth pointing out that `whatIsAtThisCoordinate()` isn't doing any boundary checking. In other words, it's not making sure that the pixel values requested are within the screen's boundaries and correspond to an actual value in the 2D grid before trying to return a value from that spot. For this reason we're

currently able to drive off the screen, and doing so can cause Processing to halt the game from encountering an error. To keep this from happening, when we get into making our real track layouts we'll surround the full course with walls, keeping the player cars in-bounds.

17. Let's give the Player cars a way to reset when they drive over the finish line area. First, we need a way to save each car's start position when it's first placed by `loadMapLayout()`. To do this we're going to add two float values to the Player `class`: `startX` and `startY`. In `loadMapLayout()` rather than setting `x` and `y` values directly for `p1` and `p2`, let's set `startX` and `startY` instead, then make car `reset()` functions that set `x` and `y` to `startX` and `startY` for that car. This way we'll be set up nicely for resetting the race after either player wins.

So where `loadMapLayout()` previously did this:

```
p1.x = ii * TRACK_TILE_WIDTH + 0.5 * TRACK_TILE_WIDTH;
p1.y =  i * TRACK_TILE_HEIGHT + 0.5 * TRACK_TILE_HEIGHT;
```

It now has this instead (and for `p2`, also!):

```
p1.startX = ii * TRACK_TILE_WIDTH + 0.5 * TRACK_TILE_WIDTH;
p1.startY =  i * TRACK_TILE_HEIGHT + 0.5 * TRACK_TILE_HEIGHT;
```

No changes there other than switching in `startX` and `startY`, which each have to be declared as `float` values in the Player `class` for this to work.

Then let's add a `reset()` function to the Player `class` that will restore its start position. While we're at it, we'll also set its orientation upward:

```
void reset() {
  x = startX;
  y = startY;
  ang = -PI/2;
}
```

Where should we call this car `reset()` function from? Let's make a more general purpose `resetGame()` function in the main source file that calls `reset()` on both cars, like so:

```
void resetGame()
{
  p1.reset();
  p2.reset();
}
```

Call `resetGame()` at the end of `setup()`, so it's after both `p1` and `p2` getting set up and also after `loadMapLayout()`. Lastly, let's do a quick change to the `updatePosition()` function in Player, so that if the track the player isn't of the

road type, it will then try to check whether it may be the goal type, and if so, both cars will be reset:

```
if(tilePlayerHit == TRACK_TYPE_PLAIN_ROAD) {
  x = newX;
  y = newY;
} else if(tilePlayerHit == TRACK_TYPE_GOAL) {
  resetGame();
}
```

The first `if` statement there is the same as it was, we've just now added an `else` condition that's going to reset the game when the cars run into the goal tiles.

18. Now on to loading our track layout from an external file. Using Notepad, TextWrangler, or some similarly very simple editor with minimal features (we don't want any formatting data in the file, just letters) save a new empty .txt (plain text) file and place it in this project's data directory alongside the two race car image files. Then copy this map arrangement into that text file and save it:

```
5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5
5, 4, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4
4, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4
4, 0, 0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 0, 0, 0, 4
4, 0, 0, 0, 4, 4, 4, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 0, 0, 4
4, 0, 0, 4, 4, 0, 0, 4, 5, 5, 4, 4, 0, 0, 0, 4, 4, 0, 0, 4
4, 0, 0, 4, 0, 0, 0, 0, 4, 5, 4, 0, 0, 0, 0, 0, 4, 0, 0, 4
4, 0, 0, 4, 0, 0, 0, 0, 0, 4, 4, 0, 0, 0, 0, 0, 4, 0, 0, 4
4, 2, 3, 4, 0, 0, 0, 0, 0, 0, 4, 0, 0, 6, 0, 0, 4, 0, 0, 4
4, 0, 0, 4, 0, 0, 6, 0, 0, 0, 6, 0, 0, 4, 0, 0, 4, 0, 0, 4
4, 4, 4, 4, 0, 0, 4, 4, 0, 0, 0, 0, 0, 4, 0, 0, 6, 0, 0, 4
4, 4, 6, 4, 0, 0, 4, 4, 4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 4
0, 1, 0, 0, 0, 0, 4, 5, 4, 4, 0, 0, 4, 4, 0, 0, 0, 0, 0, 4
0, 1, 0, 0, 0, 0, 4, 5, 5, 4, 4, 4, 4, 4, 4, 0, 0, 0, 4, 4
4, 4, 6, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4
```

Though this may look a lot like the kind of grid we could lay out in code, we'll need to add and tweak a few lines to interpret it into the data structure we're using. The first step toward doing this will be getting rid of the in-code definition we currently have for `worldGrid`, replacing the big chunk of array definition code with just this:

```
int[][] worldGrid = new int[15][20];
```

We no longer want to define the layout in the code since we'll be getting it instead from the external text file. The 15 and 20 of course correspond to the 15

rows tall and 20 columns wide for our world's track data 2D array. We'll also need to add three new lines to `loadMapLayout()`, that being the ones marked here by comments:

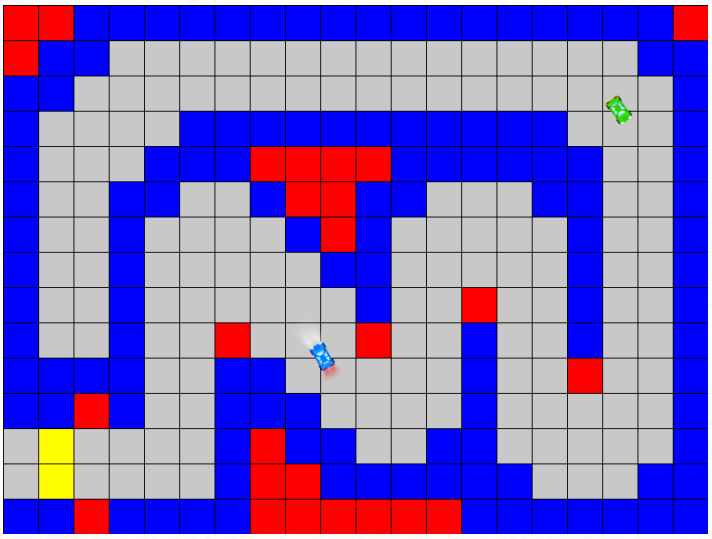
```
void loadMapLayout() {
  String lines[] = loadStrings("gameTrack.txt"); // added

  for(int row=0;row<worldGrid.length;row++) {
    String chunks[] = split(lines[row], ','); // added

    for(int col=0;col<worldGrid[row].length;col++) {
      worldGrid[row][col] = int(trim(chunks[col])); // added

      if(worldGrid[row][col] == TRACK_TYPE_P1_START) {
       // all the same below this mark
```

If the .txt file is in the right place, these three new lines will in turn open the text file, check each line (row) of it at a time, and in doing so carve that row up by commas (designating columns), putting the number found there into the corresponding `worldGrid` position. The `trim()` operation before using `int()` to convert the `String` value to a numerical value is there to cut away any spaces on either end of the number, which otherwise may crash the program. Altogether, this should use the track layout from the text file in the game.



*Code up to this point is found as* `racing_rewrite_step18` *in:*

```
classic-games/example-solutions/racing-remake-checkpoints/
```

*Lastly: loading and displaying track segment graphics, setting public and private on the functions and values in class definitions*

19. Let's finally get those decorative track segment images into the game. Draw (or copy from the directory provided with this example project's final step) 5 images, each 40x40, corresponding to a decorative flag wall, a checkered goal area, generic road, an square with some trees on it, and a segment we can repeat for most wall parts. Here are the images provided from my version:

To import these into the program we need to declare a `PImage` for each (or, as described in one of the exercises, and array of `PImages` is also an option). Declare these in the World file to stay organized, since that is where we'll be using them:

```
PImage trackRoad;
PImage trackGoal;
PImage trackWall;
PImage trackTreeWall;
PImage trackFlagWall;
```

Add a function in the World file that we'll be able to call from `setup()` in the main file to connect each of these: `PImage` references in memory to the name of its file in the data folder, like so:

```
void loadMapImages() {
  // images are looked for in the data/ folder of this sketch
  trackRoad = loadImage("track_road.png");
  trackGoal = loadImage("track_goal.png");
  trackWall = loadImage("track_wall.png");
  trackTreeWall = loadImage("track_treeWall.png");
  trackFlagWall = loadImage("track_flagWall.png");
}
```

I'm assuming here that you have named your track image files in the same way that I have, and that they are also saved as PNG format. Otherwise feel free to edit those filenames to fit your images. Call the `loadMapImages()` function in `setup()` so that we can use those image references in the map draw code.

In the World file, on the lines immediately after the definition of `final int TRACK_TYPE_WALL` add two more `final int` definitions to accommodate our additional decorative wall designations:

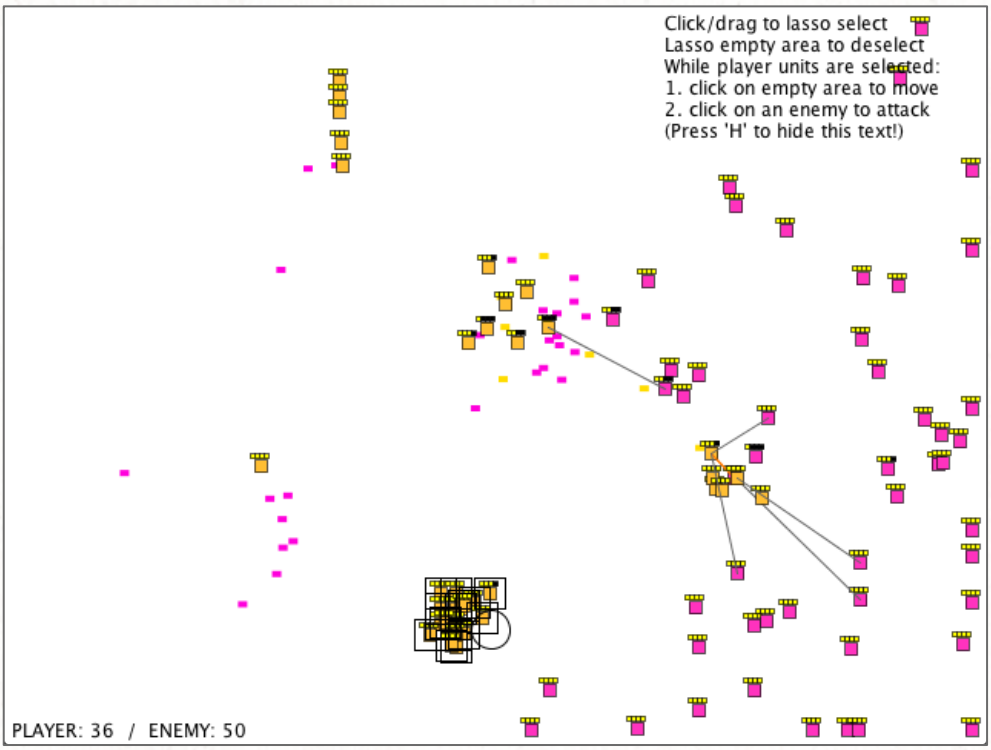
```
final int TRACK_TYPE_TREE_WALL = 5;
final int TRACK_TYPE_FLAG_WALL = 6;
```

20. We're going to need to change how `worldDrawGrid()` is written. Instead of using the different tile types as a guide for which `fill()` color to set up for a `rect()` call, let's instead use them to switch a `PImage` reference to be draw in place of that old `rect()` call by using an `image()` call. The function will wind up taking on this form:

```
void worldDrawGrid() {
  PImage useThisTileImage;
  for(int row=0;row<worldGrid.length;row++) {
    int tileTL_pixelY = row*TRACK_TILE_HEIGHT;
    for(int col=0;col<worldGrid[row].length;col++) {
      int tileTL_pixelX = col*TRACK_TILE_WIDTH;
      switch(worldGrid[row][col]) {
        case TRACK_TYPE_PLAIN_ROAD:
          useThisTileImage = trackRoad;
          break;
        case TRACK_TYPE_GOAL:
          useThisTileImage = trackGoal;
          break;
        case TRACK_TYPE_WALL:
          useThisTileImage = trackWall;
          break;
        case TRACK_TYPE_TREE_WALL:
          useThisTileImage = trackTreeWall;
          break;
        case TRACK_TYPE_FLAG_WALL:
          useThisTileImage = trackFlagWall;
          break;
      }
      image(useThisTileImage,tileTL_pixelX,tileTL_pixelY);
  ...
```

21. The checkpoint for this most recent step of the code is simply the form of Racing provided. If you opted to take on these steps before digging into the exercises, you're now completely ready to start on those from here. Good luck!

# 6 RTS Game




Whereas the other example games included are more fleshed out, this final game is a bit more of a functional skeleton to be built upon. The core gameplay and player interactions are present. It's definitely playable. However there's a lot yet still to be done here before it will have a form more consistent with player expectations for this genre.

Many of the exercises for the earlier games were about optional ways to stretch yourself and engage with the code by building in new features. Here our focus will be on really pushing you step-by-step to put more of this game together all on your own. Some of this difference can be chalked up to the considerable increase in complexity that videogames had by the early 1990's, especially those played on PC. When people new to videogame development ask around for what type of project they should try to take on making first, the answer is very often along the lines of the Tennis Game or Brick Break example projects - definitely *not* Real-Time Strategy! We're well out of the water we could stand in and swimming into the deep end here.

The other reason for this shift is because I'm a teacher at heart, and I recognize that as you gain comfort with this domain as a learner (which hopefully you've been doing for some time now, by taking on the more classic styles of games first) it's necessary for me to back off a bit so that I don't wind up getting in your way by trying to do too much of the thinking for you. You've got this.

As always though we'll start with warm-up exercises to gain some familiarity with the functionality already here before we try to do more with it.

## 6.1 Warm-Up Exercises

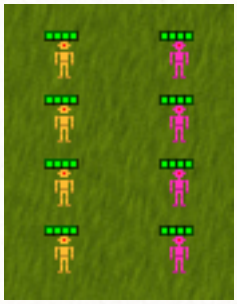
### 6.1.1 Try to win!

This isn't even a programming exercise - it's a gameplay exercise! Before digging into the game it'll help to get a bit of practice playing it effectively. The game is set up the give you as the player far fewer units than the opponent, which means you'll need to use strategy to win instead of just charging in and counting on the randomness of battle to play out in your favor. Can you find a way to play that not only wins consistently, but also maximizes the margin of units you have left when the other army is completely defeated?

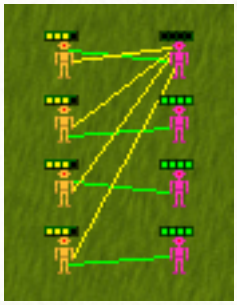
**Details:**

There are a few basic strategies for classic real-time strategy games that derive from the fact that a defeated unit can't return fire but a damaged one can still deal full damage, and others that work in an emergent way from how the most basic unit-AI operates. Those strategies can be exploited in our game.

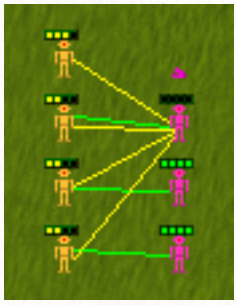
The first techniques have to do with focused fire. Think of this scenario: if there are four units on each side, each with four health, and all able to attack once per second, will the team do better by each picking a separate target, or all firing on the same target for each volley? Let's take a look:



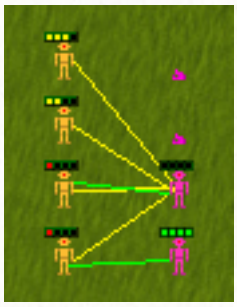
At the start of our example scenario, 4 units all with full health from each side are meeting in the battlefield. The orange team on the left is going to concentrate its fire on one target a time, whereas the purple robots on the right will each pick a different target.



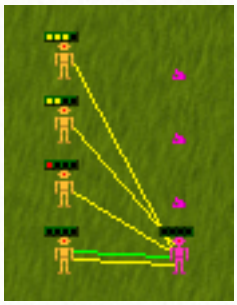
In the first round each unit on the orange side loses 1 health, meanwhile one robot from the purple team is immediately eliminated.



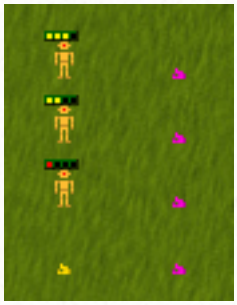
In the second exchange the orange team on the left is again able to eliminate a single purple robot on the right, however this time only 3 of the orange 4 take damage, since the purple team has one less point of attack than they did when this started.



The advantage for the orange side is probably pretty clear by now, but let's let this finish out. In round three, another purple enemy is toast, and now only two orange robots lose another hit point.



In the last volley, the fourth and only remaining member of the purple squad is finished, while the orange team on the left loses a lone member, their first and only loss in this battle.



Result: 3 out of 4 orange units standing, with an average of half their starting health remaining. All 4 purple units have been eliminated. Victory goes to the orange team, by a substantial margin.

It's worth admitting that the above case is a pretty ideal and somewhat contrived scenario for illustrating this point. Rarely do all units on both sides fire at exactly the same time. Every unit is not necessarily within attack range of every other unit in a conflict. The number of units involved, how much damage each does when attacking (potentially accounting for a probability of dealing an inconsistent amount of damage, or with an inconsistent ability to hit its target reliably between each reload delay), and how much health each unit involved are all parameters that may affect the optimal strategy.

For example in an 8 on 8 battle, if all 8 from one team focused fire on a single enemy target a full half of those 8 shots would be wasted on an already defeated target, since they only have 4 health total. There it would be advantageous to break the groups into two sets of 4, each focusing firing on a separate target - again, assuming 100% accuracy, all units have exactly 4 health, every unit begins at full health, both groups have the same number of units in the exchange, units that have taken damage still possess attack capabilities identical to those of a unit with full health, and so on.

Well if it's so gosh darned complicated, then what does all this mean about effective play strategies?

Even though the exact details inevitably vary during the chaos as dozens of robots fire lasers at one another, a handful basic principles hold up that are worth learning from this. (1.) Focusing fire with direct common orders is generally advantageous compared to simply moving your units near enemies then leaving each to randomly pick a target for itself. (2.) Trying to utilize a squad of more than four units against opponents that have a maximum of four health risks inefficiency, unless you're able to rapidly micromanage them as subgroups. A slightly larger group only hedges your bets against the randomness of losing a unit during battle, or serves to further dilute unfocused enemy attacks. (3.) Bringing a unit with low health into a new firefight means you may lose a point of attack early in the exchange - so send those damaged robots back to hide among the main group where there's a lower probability that they will get targeted, and bring forward others that have full health in their place. Conversely: attack enemy groups that have damaged units before taking on groups of enemies with mostly full health, and you'll likely go into that second battle with more working lasers to use. (4.) When possible, send a full squad of four to attack a separated enemy, rather than sending just two. That way the enemy can likely be eliminated nearly instantly, perhaps before even getting an opportunity to deal any damage to your units.

If you're somewhat new to this genre, it's hopefully become clear why it's called real-time *strategy*. There really is a whole lot more to the gameplay here than just sending your crowd into their crowd and hoping for the best. More interestingly, observe how these strategies outlined so far aren't actually even programmed directly into the game! Nowhere do we ever specify explicitly in the code which is a better decision, to focus fire or not, nor what equations could solve for how many units should remain after a firefight between a certain number of units versus a certain number of units depending upon which tactics either side employs. These strategies instead emerge from the dynamic interplay of observable events in the gameplay. A developer of an RTS game doesn't need to put in special effort to ensure that these strategies work, and may not even be aware that these techniques exist in their game.

The second gameplay tip derives from a few different basic features of the simplistic per-unit automatic behaviors. This is also generally applicable in other RTS games, especially older ones where the AI doesn't have checks implemented to prevent this. In concept, this amounts to leading the enemy into an ambush, which

in the history of conflict might even predate throwing rocks. The way this works is that characters will auto-target any unit that gets within attack range, or alternatively will target a unit back when attacked. Characters are also often programmed to pursue a retreating unit once it's already targeted, at least for a modest distance. As explained above, a character walking into a group of enemies can sometimes be defeated before it even has an opportunity to return fire - invariably all the ambushing units have only the one distracted target to attack, resulting in a focused fire situation even without direct orders. In summary: send a few healthy units to pick a fight with an enemy squad (a few so that they can survive the first and nearly immediate counter shot or two), then immediately retreat once you've got their attention, running back across a larger pack of your own units which are waiting to overwhelm the enemy robots once they're in range.[22]

It's worthwhile for a developer to learn functional fluency in the literacy of their players, in order to better recognize how the work can be shaped to enhance what's best about the game, or to spot and repair implementation issues (what may even be thought of as technical or design "improvements" while being put in) that interfere with the game's core functionality. It's not necessary to become a competitive master at the game, but it is useful for a craftsperson making furniture to have actually used that type of furniture before to think about its manner of use.

### 6.1.2 Clear select on right click

These days for cross platform support we typically try to make our games playable with a one-button mouse, since Mac trackpads and some computer mice only have a single button. This is rarely ideal or even comfortable for game types that originally required two-button mice, but it's worth at least ensuring there is some way to play the game for someone that doesn't have two buttons. As implemented, the game already does just that. That said there's no reason we can't also support the second button for the many players that do have it. After all, PCs still vastly outnumbering non-PCs. An extremely helpful feature to support in an RTS game is to let players clear their selection by right clicking. For this exercise, deselect all units when the player presses the right mouse button. (If you are using a Mac laptop and do not have easy access to a right mouse button, the same functionality can of course be tested in a more clumsy fashion by holding the Control key when clicking.)

[22] All that shouting in military movies about holding the line, or trying to suppress the rage of a soldier with a lost temper, and the general emphasis on soldier discipline is largely to prevent that simple pattern from being successfully exploited. While real military engagements are no doubt remarkably more complicated than Hollywood's simplified depictions, it's worth keeping in mind as a reference, since unless you're programming a training simulation your player's expectations are more likely set by Hollywood than reality.

**Details:**

Even though it isn't going to be hard to do once we understand the code in this program, supporting this feature is going to require becoming acquainted with how this game's source fits together. As mentioned in the introduction, there are many different ways to program equivalent behavior, and my priority in constructing these examples has been conceptual clarity, where possible using only introductory programming features. Clever optimizations have generally been avoided in favor of more readable code.

The way that I chose to implement player selection of multiple units is to keep a separate `ArrayList` named `selectedUnits` that has references to those `Unit` entities in the main `ArrayList` (named simply `units`) that are currently highlighted to receive player commands. Every infantry `Unit` in the game, whether defeated or still active, fighting for either side, selected or unselected, is always in the `units` group - that's the list we draw every frame so it has to include everything. Any and every `Unit` that is currently selected, while still being in the `units` list, is *also* contained in the `selectedUnits` list. We draw the selection boxes around all selected infantry by doing a second drawing pass every frame after drawing all army units from both sides, to then draw the selection boxes on those that are also referenced in the `selectedUnits` array. Among other minor benefits like simplifying our code to send command to all selected infantry, this approach also ensures that selection boxes will always appear drawn over the characters, without one character blocking part of another's selection box as could happen if we drew selection boxes on the same pass as drawing the actual `Units`.

How then do we deselect the player's current selection? We simply need to clear the `selectedUnits` list. Where we should do this in code is in the `mousePressed()` function within the mouseSelect file, which again is called whenever a mouse button gets pressed, just like `keyPressed()` is automatically connected by Processing to be called whenever a key gets pressed. Just as we can filter in `keyPressed()` for different arrow keys with `if(keyCode == RIGHT)` in the `mousePressed()` function we can add at the top a check for:

```
if(mouseButton == RIGHT) {
    // add the line here to deselect any currently selected units
    return;
}
```

The `return` line of code ensures that when using the right mouse button the other code in that function, written for left/main clicks, won't also happen immediately following the deselection of units. A `return` line quits out of the function at that line, giving back control flow to the rest of the program.

### 6.1.3 Spread out formation

When multiple infantry, selected by a click/drag motion, are directed to move to a new empty position they gather in an orderly grid formation. While the move

instruction is being passed to each `Unit` a counter is giving each a distinct number, which is then used in a simple pair of expressions to calculate how far over and down from the mouse's click position its unique destination should be. Alas, the `units` currently fit together so closely that it can be a little hard to see where health bars start and end, and this will only get worse when we replace those squares in a later exercise with some little robot graphics. Find where in the code the pixel distances in the formation are, and increase it to 15.0.

**Details:**

Not much needed in terms of further details or advice, since you're basically just hunting for a number and changing it. Of course, while you're at it, and you've got where this value is buried in the code, it would be a pretty good idea to go ahead and define and use a new `final float` constant tuning value, `FORMATION_PIXEL_SPACING`, at the top of the class or file so that tweaking this value later in development won't involve so much searching.

### 6.1.4 Remove targeting line

The gray line drawn from a unit to its target is convenient for debugging purposes, since it makes it clear which characters are targeting others, It visually clutters an already busy scene though, and makes it difficult to really see what's happening in battles after infantry get close together. Let's do away with that targeting line altogether, instead showing only the attack line.

**Details:**

There's only one `line()` call in the Unit file, though it's being used to draw both the targeting line and the attack line with a different `stroke()` set for when it's attacking. Rearrange the logic there so that we'll never see a targeting line, but the colored attack line still appears when units are engaging in a firefight with one another.

### 6.1.5 Make attack line look more like a laser

The way the attack line is presented still looks more like debugging information than an actual attack. Let's make a few changes to it so that it looks more like a laser blaster:

1. Change the color of the laser based on which team is firing it.
2. Randomly flicker the line by only drawing it 30% of the time, making it appear rapid fire. This also helps prevent the laser drawn second from always overlapping the laser drawn first.
3. Increase attack range from 30 to 45 so that we can see more lines when unit groups get near. (This of course has some effect on gameplay, but without any other units or terrain yet these values are still easily adjusted without worrying about balancing.)

4. To further distinguish the robot that's firing from the side being fired at, let's randomly offset the target end of the line by up to 6 pixels in either direction along either axis.

**Details:**

This is of course in the same section of code as the previous exercise. To make the laser colors different per team use an `if` conditional on `playerTeam` to set `stroke()` to one color or another. To make the laser flicker, in addition to only drawing the line when `target != null` and `isAttacking`, add a third condition that must be met: (`random(100)<30`). The attack range is a `final float` value near the top of the file. The shot endpoints can be randomized by 6 pixels in either direction by adding `random(-6.99,7)` to the `target.meX` and `target.meY` arguments of the `line()` call.

Why -6.99 on the left side, but 7 on the right, to move the endpoint by six on either axis? Pixel values get cast to `int`, since a pixel is either on one point of light or the next and can't be between. Calling `random()` with a minimum and maximum value for the range is inclusive on the minimum side (so it technically could be -7 if we used -7 as the first argument, though in practice it would only happen very, very rarely anyhow), and exclusive on the maximum side (so when the second argument is 7, the value returned can be a tiny bit less than 7 but will never be 7 exactly).

### 6.1.6 Hide health unless selected or near mouse

Right now all health bars show for all units all the time. This is cluttering the battlefield with far more information than the player can really use at any given time. We've also been doing enough game programming by now to begin thinking ahead a little, and when we start putting in graphics we're not going to want abstract bars covering the action all the time. Instead of always showing health bars for all infantry, only show health bars for units that are currently selected or are within some distance (let's say, for starters, 60 pixels) of the mouse cursor. This means the player will always be able to easily see the health of his or her own infantry that are prepared for orders, in addition to being able to scan the mouse over enemy troops to quickly size up their crowd for identifying weaknesses, meaning unit clusters with less than full health.

**Details:**

This exercise will be easier to complete if we're willing to accept a slightly lazy trick, ignoring the barely noticeable side effect of doing so. And in practice completing even a medium-sized videogame project alone or on a small amateur team involves making dozens or even hundreds of tradeoffs of that sort, so lazy but good enough solution, here we come!

First, let's make this number easier to tune later by defining a `final int` at the top of the Unit file, `HEALTH_SHOW_RANGE`, set to 60. Within the Unit file's `drawMe()` function, near the top there's a call to `showHealth()` that causes that health bar to always be shown. Wrap it in a new `if` conditional that checks whether the distance

(using Processing's `dist()` function) between `mouseX`, `mouseY` and `meX`, `meY` is less than `HEALTH_SHOW_RANGE`. Since that function will now only be called for a unit when the mouse pointer is close enough to it, we've completed the first part of the exercise. Feel free to try it at this point to check that it works as expected.

Next is where we're going to deal with our slightly lazy shortcut. Recall that in addition to showing a unit's health if it's near the mouse, we also want to show the health when that unit is selected. The individual unit actually has no idea whether or not it's selected. Selection just means that there's an additional reference to it in that secondary `ArrayList` named `selectedUnits`, remember?

As a brute force approach, we could check in the infantry's `drawMe()` code against every reference in `selectedUnits` whether any of them are equal to `this`, which is the programming keyword for referring to the class's instance that the function is currently being executed on. We have 120 units in play, dozens of which could be selected at a time, which means every cycle we would then be checking 120 times dozens of comparisons just to figure out if a unit is selected. We aren't terribly concerned with runtime efficiency for this program, but that's bordering on performance limiting, especially going forward when we introduce a larger scrolling world map. We could of course give each unit a `boolean` variable set `true` or `false` based on whether it's currently selected, but then we've got the same information represented two different ways in the program and we'd basically be just asking for bugs to crop up later when these two manage to get out of sync with one another.

Instead, we'll simply loop through all `selectedUnits` outside of the Unit `drawMe()` function, and there separately call the health bar draw function on each. Doing it this way, if the mouse is within range of the unit, and it's selected, its health bar will get rendered twice - not really a big deal in the scheme of things, and not likely going to be the source of a performance hit. There's a barely noticeable difference in the darkness of the health bar when it gets drawn twice, since the black outline around health bars is partially transparent, so stacking it twice doesn't allow as much of the background to show through. There are various ways we could work around this, any of which would in itself be a little hack: in the loop through selected units skip drawing its health bar if it's within the `HEALTH_SHOW_RANGE` radius, for example. It's probably not noticeable to worry about though, especially once we get a background graphic behind it besides a solid color. Hold off on stressing over a fix for it at this time, and if later in testing it comes up because it's noticeable, a fix can be prioritized for it then.[23]

Alternatively, and more traditionally, enemy health might only be visible when it has been recently attacked, instead of ever showing when near the mouse. This would add some challenge in the form of information hiding, in which to gather facts about the enemy troops the player has to do recon by engaging with them. For now as an expedient shortcut that makes the gameplay a bit more accessible, this distance to mouse check is fine. However, if taking this project further it

[23] Though this tradeoff here is set up a bit for the sake of example, situations and decisions much like this really do come up seemingly constantly when making medium-large sized game projects.

would be good to go back and reconsider implementing that health bar convention instead since it's more consistent with player expectations, unless you have a very good reason not to.

### 6.1.7 Change health color when low or max

When scanning the battlefield to get a sense for where the player or enemy's army is already damaged, trying to gauge the status of many tiny health bars at once can be hard to do at a glance. One way we can help this is to adopt an old convention of coloring the health based on its relative level: green when high/full, yellow when a little damage has been taken, and red when it's near defeat. Change the health bar code to change color based on the particular character's number of remaining hit points.

**Details:**

This will need to be done in the `showHealth()` function defined in the Unit class. Currently in the loop that draws the tiny square for each `hitpoint` on the health bar, there's an `if` conditional calling one `fill()` value for the player having health exceeding that box's draw number (255,255,0 for full red and full green, no blue, making yellow[24]), and a different color value for the player's health being below that box number (0 in all color channels, making for solid black). We still want to show black for missing health, so our new code for health color change should be nestled entirely within the `if(hitpoints>i)` conditional. Here's a case where an else-if chain makes sense, in the form:

```
if(hitpoints==START_HEALTH) {
       // full health, so set fill to green
} else if(hitpoints<=START_HEALTH/2) {
      // health is at or below halfway, set fill to red
} else {
      // health is between half and full, set fill to yellow
}
```

Again, this will need to all be within the `hitpoints>i` conditional, since if health is low we still only want to show 1 or 2 red dots, not a red filled bar. Notice that in the comparisons above I used our tuning value `START_HEALTH` rather than typing in hard coded values like 2, or 4. This is of course to help ensure that our colors signify the same meaning even if we later adjust how much health the units start with. This approach is also a more flexible way to handle it in case we later want to support multiple types of units that begin with different amounts of health; having more than 2 `hitpoints` remaining may qualify as medium level

[24] In case your exposure to color mixing is from a traditional art class, and you're used to thinking of Red, Yellow, and Blue as the primary colors rather than Red, Green, and Blue: you're definitely right, too! The former applies to subtractive color, as when paint gets mixed, whereas the latter isf or additive color, which mostly means when lighting gets mixed. Since a monitor presents color by light mixing, Green is our third primary color instead of Yellow.

health for an infantry robot, however for a tank that's probably well below half health. We'll still need to make some changes to how health gets displayed for those situations, for example adjusting how wide each box is to keep the overall health bar an appropriate horizontal length, but thinking ahead a bit to write this part right saves us a little thinking later.

## 6.2 Practice Exercises

### 6.2.1 Background graphics

Draw a background image - grass, dirt, metal, or some other material. It's important for this to be our first graphic, before dealing with unit appearance, since this way when drawing army units we'll be able to experiment to determine what stands out well enough against the background terrain. Draw a background image the same dimensions as the screen, load it into your program, and display it at the start of each frame instead of (the default functionality provided in the source) using a white rectangle to overwrite the previous frame. If necessary, adjust the coloration and/or brightness of the on-screen help and interface text to keep it legible.

**Details:**

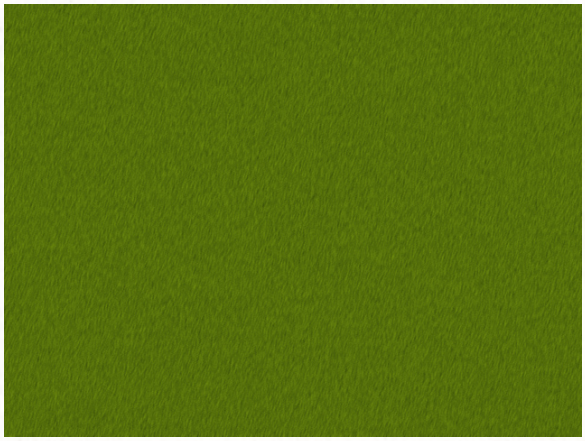
Most images authored for our programs so far have needed to support transparency, meaning that we could have parts of the image where we see through to the background around a circle or edge without it being filled in by in a white or black box behind it. For those images we stuck to PNG format because it provides good transparency support. For the background image however we specifically don't want any transparency, since we're counting on being able to draw it each frame to completely wipe away whatever was leftover in the frame buffer from the previous cycle. The background is also larger in dimensions than our little unit sprites. The combination of not needing to support transparency and wanting to keep the file size down for a larger image makes JPG a good candidate for the background image's format. Thankfully Processing can load and use either of those, along with a handful of other common alternative formats, seamlessly just by indicating the image filename with its different format extension at the end.

There are a few things to keep in mind while drawing the ground terrain. First, at this time in the development, we won't have support for representing obstacle collision. So avoid the temptation to draw rivers, boulders, trees, cliffs, buildings, and so forth, which the troops will walk right through effortlessly. We'll get to at least a version of those features soon, but for now just focus on getting the main ground appearance right.

Because we'll need to see both text and unit art clearly over the ground image it's important to not make it too detailed, busy, or high in contrast. On the other hand if it's too washed out or solid in appearance then it won't look like a ground surface but instead like a picture abstractly displayed behind the action. It's going to take some experimentation to find something that feels right, and after making

unit graphics to go on top of it, it may involve further tweaking or fresh attempts, so don't spend too much time making something yet that you'll grow attached to.

I'm definitely not a specialist in visual art, but one very fast and simple approach that I've found helpful to mash out this kind of asset quickly (at least as a placeholder until I can enlist a better artist's help, or dedicate more of my own time later in the project to a final version) is to create a new image the same dimensions as the playfield (see the `size()` call in `setup()` for those dimensions), fill the whole image with white using the paint bucket, apply a noise filter to cover the image in dots of varying brightness, tint the image dark green for a grassy look, then decrease contrast a bit (one way I approach this for a little more color control is to make a new solid color layer above the background, and adjust its transparency). After another filter or two, including subtle motion blur in an effort to give the grass some length and directionality, here's what I came up with for my background at this point:



When layers are involved Photoshop or `GIMP.org`'s alternative will attempt to save the file as one of their formats that can keep track of the layer data, although both also give ways to export or save a copy as a jpg or other format to have it ready for use in the game. Processing can't directly open Photoshop or `GIMP.org`'s native layer formats.

As for recoloring the `text()` calls to keep them visible on the background, the color of text gets set somewhat counter intuitively by whatever `fill()` call precedes them. In the case of a darker background, putting a `fill()` before the text calls at the end of `draw()` that has 255 for each of the RGB channel arguments will produce white text that can stand out easier.

### 6.2.2 Unit graphics with team coloration

It's time to finally turn those little battling squares into little battling robots! Make a very small (around 7 x 14 pixels - you'll definitely need to figure out how to zoom in using your image editing software to work at this scale!) character image with a transparent background. When drawing the robot, make most of its pixels shades of gray, then in code tint the image based on which army the unit is

fighting for. Also create an image for the robot when defeated, depicting an asymmetrical pile of pixels that are also shades of gray, so that we can also color the scrap pile based on which team it belongs to.

Draw these images centered on the unit coordinate, instead of the colored square that was being draw before for each unit. Increase the dimensions of the selection indicator square so that there's a more comfortable margin between the image and the interface element's lines, and likewise bump the unit health bar's upward a little if needed to make the head room and spacing between a unit and its health more visually appealing.

**Details:**

Why robots, you may wonder, rather than people, aliens, or some other creature out of myths or the imagination? While you're of course free to take the project's appearance and style in any direction you may choose, there are some advantages to robots that make them appealing as a first target:

1. We need to keep these characters incredibly small, which is also going to mean that they're going to be somewhat boxy. It takes a lot less art or animation skill and time to get away with having boxy robots than boxy organic creatures like people or Martians.
2. There's a rich history of highly diverse robot depictions in media that we can draw inspiration from, or we can just as easily take our own spin on it, and unlike a realistic human form where player expectations are clear about how it "should" look, we have a ton of wiggle room for drawing robots and someone is unlikely to feel like robots couldn't take that form or shape.
3. While I opted to put little legs on my robots, meaning later in the project I should probably try to animate their legs while they're moving, robots have the wonderful capacity to be on little treads, wheels, or hover bases, any of which we can get away with not animating at all (or just swapping some gray and black pixels in place) if we're clever about how we draw them.
4. Although particle effects aren't specifically an exercise provided for this project, they are covered in other project challenges, and they could of course be applied here. Robots can (and let's be perfectly honest here, frankly should) explode in a spectacle of smoke, flying debris, and bursting flames when defeated, leaving behind a smoldering scrap pile. Besides adding some visual interest to the battlefield, this can also serve the practical function of signaling to the player where and when units in a battle are getting defeated, in a much more noticeable way than simply falling over. Different explosion colors could even be used for each army, making it easier to see at a glance which side is winning!
5. The way we've set up drawing the attack lines looks like laser blasts, so we're already partway into a sci-fi setting already. We could of course always go back and change that representation in various ways to make it

look more like machinegun fire, arrows fired from a bow, or magic spells, but it's what we've got for now.

6. Whereas we may have an expectation for humans or alien creatures to operate in a semi-intelligent and self-preserving manner, we don't typically hold robots to the same standards. Robots making questionable, simplistic decisions or getting into fights where they're doomed to lose is a lot more believable. It's also a less appalling, which brings us to the last consideration...
7. Robots battling robots is about as uncontroversial as action can get. Since there are people that are uncomfortable with videogame depictions of violence against or between living things, and it is easier to support the exact same gameplay with robots anyhow, it's worth considering that if you keep your army units inanimate you open up your game to reaching more players out there of all ages and a broader range of personal beliefs.

Here's my 7x14 robot, with the blue fill showing up to the boundaries where my image has transparency (filled in here, since otherwise you'd be looking at white on white; the actual image doesn't have the blue in it[25]):

While some of those reasons listed above for using robots may seem a tad lazy, when there's so much work to be done on making a videogame, sometimes there's a very fine line between lazy and cleverly resourceful. When games were still in black and white we saw a lot of games based in space. When 3D graphics couldn't yet display many objects at once we saw a lot of games taking place in very foggy or dark environments. For a window of time in-between when pixels were larger and had fewer shades of colors to work with, we saw a bunch of games with robots. More recently many indie games have been made with enlarged pixels, vector graphics, flat shaded polygons, or other workarounds to do the best they can within their time, budget, and training limitations. Figuring out what you're able to do well in your development situation isn't lazy. It's smart! And, if you're like me in terms of lacking substantial training in animation or graphic illustration

[25] Though while working on an image like this I do fill in a colored background so that it's easier to see what I'm doing. As a historical note: an older approach to image transparency, much less flexible than the PNG format, involved selecting a single color value to have a program treat as transparency by skipping when copying the image from memory to the screen. Like the green screen behind the weatherman these colors were often a vibrant cyan, magenta, or neon green, since the actual game graphics didn't need to use those shades.

techniques, there's a good chance that you too can pull off stiff, imperfect robots a lot better than trying to get away with stiff, imperfect humans.

On to the technical stuff: since we need transparency in order to see the ground behind the robot over its shoulders, between its arms and body, etc. we should save this one in PNG format. The selection box's dimensions are in the `showSelectionBox()` function - for tuning purposes pulling that out into a new `final int` variable, `SELECTION_BOX_SIZE`, at the top of the class makes sense (note where it's being used that the subtracted offset for the x and y position is half of the width, in order to center it over the unit's origin point). While we're editing the selection box anyhow, I'd suggest decreasing the outline's alpha value to make the box less opaque and distracting - this can be done by adding a fourth argument to the `stroke()` call before the `rect()`, for example 90 would signify 90 out of 255 alpha acheiving about 35% opacity. The health bar's height above the origin is of course in the `showHealth()` function.

Coloring the units by which team they're on be done with a `tint()` call instead of `fill()` when drawing the unit graphics. Much like the other graphic color calls, `tint()` will continue affecting any image calls made after it. So after presenting the recolored image call `tint()` with all 3 color arguments at 255 to let the next image calls in the code show their full and proper coloration.

One thing that we're not really dealing with in this draw code, which will be a bit more noticeable now that that we have much less abstract representations, is draw order. If you look closely where there are many units drawn near one another, you can sometimes notice a scrap pile on the ground behind a robot overlapping a robot that conceptually is standing closer to the camera (in math terms, that means with an origin lower on the screen), and the same can also happen between robots, especially when they're each on different teams so that their graphics don't blend together when close. We can address this later in a challenge exercise, but for now it's worth noticing that it's happening. By keeping our units small and graphics simple however we've successfully reduced how noticeable or distracting this artifact is during play.

### 6.2.3 Heavy unit

To support a wide variety of units, we'd want to develop a pretty robust system for adjusting the tuning on each type - such as an external text file for each unit, containing its tuning parameters. Before we get ahead of ourselves trying to design and build a more generalized approach, let's first just figure out a way to hack in support for a new type of unit. This will help us gain an understanding of what tuning values we would want to be able to specify in an external text file, while also giving us a sense for ways that mixing in different unit types can breathe more life and strategy into the game.

**Details:**

Create a new, slightly large character image for the heavy unit. Try to match the same general look as your infantry robot, while also making it visually distinct, and somehow indicating that it's meant to be slower, stronger, and better armed.

Stick mostly to shades of gray that will respond to the army's `tint()` color predictably, though a splash of some other color for highlights is fine. Here's mine - again, blue just added to make it easier to see:

Remember to make a destroyed version of the unit's graphic, too. Even though the used portion of the image won't be anywhere near the full dimensions of the picture, leaving the dimensions the same is an easy way to ensure the scrap heap lines up at the feat of the unit when it gets defeated. If we cropped the scrap heap image and kept it centered on the unit's origin, it might appear to pop up halfway when switching to the pile.

Add a `boolean` in the Unit `class` called `isHeavy`. If this is set to `true`, we'll handle this as our second unit type in terms of its presentation and behaviors. To support many different kinds of units we'd more likely use an `int` to keep track of which unit type this is, which could then act as an index into an array filled with tuning values loaded from text files, but again, one thing at a time!

`MOVE_SPEED`, `ATTACK_RANGE`, and `START_HEALTH` are all currently `final` values, but we'll want these to vary by unit since these are among the most dramatic ways to distinguish unit functionality (on another pass we might also affect reload times, damage done in attacks, accuracy, AI tuning values, etc.). We're going to remove final from those values and get rid of their in-line number initialization, so that now that section of the class definition will look like this:

```
final float MOVE_SPEED_INFANTRY = 1.5;
final float MOVE_SPEED_ROBOT = 0.7;
float moveSpeed;

final float ATTACK_RANGE_INFANTRY = 45.0;
final float ATTACK_RANGE_ROBOT = 180.0;
float attackRange;
```

```
final int START_HEALTH_INFANTRY = 4;
final int START_HEALTH_ROBOT = 16;
int startHealth;
```

In the `randomReset()` function, we'll then set `moveSpeed`, `attackRange`, and `startHealth` to one set of those `final` values or the other based on `if(isHeavy)`. Afterward update `AI_TargetRange = attackRange*4.0`. That can no longer be `final` either, since its calculation is a function of `attackRange`, which we don't set until `randomReset()`. As with the other numbers that are no longer `final`, hanging the name from `AI_TARGET_RANGE` to `AI_TargetRange` helps avoid mix-ups elsewhere in the code over which values are `final`.

When and how does `isHeavy` get set? As a testing shortcut for now we can randomly assign some percentage of units to be this special type, much like how we're randomly assigning teams, by doing this before the comparison that sets the tuning numbers:

```
isHeavy = (random(100)<8);
```

That way about 8% of the units on the field will be heavy units, keeping the battle largely focused on the default infantry but with enough heavy ones involved for their presence to matter and be used strategically.

The last other detail to sort out here is how to get the health bar to show up the right width, rather than extending well beyond the expected dimensions. Since our previous health bar was based on giving each 3 pixels for 4 health points, we can calculate a new width based on the number of maximum `hitpoints` on a unit by accounting for that numerical balance in `showHealth()`, starting with this early in the function:

```
noStroke(); // get rid of outlines, since we'll need smaller slivers
float pixelSkipWidth = 3.0*4.0/startHealth;
```

Then changing the health box `rect()` call like so:

```
rect(meX-7+i*pixelSkipWidth,meY-11,pixelSkipWidth,3);
```

### 6.2.4 Color key collision affecting mobility, range, and visibility

Just a few exercises ago, when drawing a background for the battle area, I mentioned that we'd later add support for different terrain types, so don't get overly attached to it. Later is now!

We could of course implement a grid-based collision and level rendering system, much like the one from the Racing project. That has several advantages, among them the ability to create entirely new levels rapidly by simply rearranging the tiles. However the drawback of the tile approach, as you may have noticed while tinkering on that project, is that the levels wind up chunky and inorganic. We could treat this as a rationale for putting the battle in a giant warehouse, factory, space station, or futuristic urban landscape, where repetition would look fine and the space being laid out on a regular grid would make sense. Indeed, again, many

perfectly fun and successful games did just that, back when tile-based environments were not only a convenient way to make large landscapes but were in some cases the only way videogames on old processors could handle large landscapes at all.

Now though we're going to take advantage of some of the advances in computing hardware that have happened since, to relatively easily create a battlefield with organic, jagged, and gradual edges between zones that are alternatively uneven, marshy, sandy, wooded, flooded, and so on.

To do this we're going to need two different versions our background: one that's a JPG, illustrating pretty visual detail, that we'll show the player, just as we're doing now with the current background, plus another image of the same size that's a PNG (to avoid color inaccuracies due to JPG's slightly lossy compression artifacts[26]). Instead of containing pretty illustrations intended for human viewing, the PNG will just contain blobs of specific color values that correspond to different terrain regions of the JPG, using a different RGB color for high grass, one for mud, one for shallow water, etc. We'll determine what terrain type a unit is standing in - to affect its attack range, movement speed, or other parameters - checking and matching against the color value in the PNG that corresponds to its position on the battlefield.

**Details:**

Even though conceptually this is fairly straightforward, there are a few subtle tricks involved in getting it done well. I've provided an example as the `commonBonus/free-from-hobbygamedev.com/lab-examples/sketch_colorkey` code included with this source pack. At the time of this writing there's even a video on YouTube that I prepared as review material for a class that walks through writing that example code: `https://www.youtube.com/watch?v=3AA9Hs5RfsA`

When coming up with ways to have the terrain type affect unit mobility, consider how it may impact the infantry robots differently than the heavy unit. For example the heavy unit may be much less affected by terrain that slows down infantry, such as mud, but may be unable to cross water at all.

[26] Part of how a JPG manages to make a detailed image so much smaller than other image formats can is by approximating the color data in subtle ways that the human eye barely notices. This is also the trick to how MP3 and M4A files manage to get music so much smaller than a WAV file. The little bit of inaccuracy is fine for human viewing or listening, but if we're counting on the precision of its values to be machine interpreted than we'll want to preserve every pixel's color without compromise. Fortunately, PNG, like GIF, supports something called run-length encoding: if the next 300 pixels of the image from left to right, top to bottom are all an exact shade of red, the file won't save 300 redundant data points but will instead make a note to the effect of what the shade of red is and how many of the next pixels it accounts for. In consequence: for an image with large blocks of consistent color, like our color key collision lookup image, PNG can actually produce a file smaller than JPG, anyhow!

Note however that if terrain is created which certain unit types cannot navigate at all (rather than simply being slowed down or otherwise impaired in functionality when traversing them) there may be player expectation of AI capable of pathing around those barriers to follow orders. Solving that problem would involve an algorithm like A*, pronounced A Star, which admittedly is beyond the scope of even the challenge exercises, but is absolutely worth searching for and learning about on the web if you're curious to go further and get more serious in creating a full RTS gameplay experience.

## 6.3 Challenge Exercises

### 6.3.1 Scrolling camera with minimap

Now that we have our terrain system set up, let's take it beyond the size of a single screen. Traditionally RTS games take place on a map large enough that the player needs to send out scouts and explore the space to figure out where opponents are hiding, or to identify safer directions to approach the enemy units from. While for performance reasons we probably don't want to make a massive area for this gameplay demo, it's still well worth expanding beyond the size of a single screen.

In addition to supporting scrolling camera - one technique for which has been covered in exercises in other games included - also considering implementing for this challenge a minimap that shows all unit positions from the world compressed into a very small part of the screen.

**Details:**

As mentioned above, camera scrolling has been discussed elsewhere, and can be done with two `float` values, one for the camera's offset along each axis, combined with an extra `pushMatrix()`, `translate()`, and `popMatrix()` wrapping around everything in `draw()` other than the interface text. Instead of computing the camera offsets based on a main character, instead when the mouse is near any screen edge the camera offsets should be incremented or decremented accordingly, so that the player can pan around the map by moving the mouse near the edges. Alternatively or additionally, holding the keyboard arrow keys could be used to reposition the screen view relative to the map, another common convention. Remember that when detecting click coordinates or proximity to the mouse, the camera's offset values will need to factored into consideration alongside `mouseX` and `mouseY` which know nothing of the scrolling camera. Increasing the world size will require increasing the size of both the terrain display image and the color key collision image - experiment with some test sizes before committing to drawing a ton of detail on either, in case you discover while experimenting that your initial plan for land mass is more ambitious than your processor is comfortable dealing with while maintaining a fluid frame rate.

The minimap sounds fancy, but is actually a great deal less complicated than one might be inclined to assume. Near the end of `draw()`, after the camera

`translate()` gets removed before interface calls by `popMatrix()`, use a `rect()` call to draw a small black rectangle which is proportional to the terrain, or alternatively scale down your terrain file in your image editor and darken it a bit to use as the radar view, in case you want terrain landmarks on the minimap. Then iterate through all units, calculate its x and y coordinate as a percentage of the terrain's total size along each axis, scale that by the minimap's dimensions for the axis, put down a pixel of one color or the other depending on which team the unit is on, and whalla, minimap. Drawing a rectangle on the minimap to represent where the camera is scrolled to can be done in a similar way, using the camera offset values rather than unit position, to plot the corner of the screen, then size that rectangle proportionally to how much of the terrain the screen's display can show at once.

### 6.3.2 Fog of war

A fog of war system blacks out areas that the player has not yet explored, or, if preferred makes enemy units invisible if they beyond a certain visibility range from any of your own units. Let's get at least one of those fog of war concepts working.

**Details:**

As is often the case in game programming, there are many ways to do this, involving tradeoffs between performance, ease or simplicity of implementation, and differences in how smooth or rough the method appears. For blacking out areas prior to exploration, but leaving them visible once the player has been there, a Brick Break-type of grid system could be employed that draws black squares over the playing area where there's still a 1, and units could semi-frequently update any positions adjacent to their position or their target to a 0 so that the world gradually gets exposed to the player. By default that would look very jagged, although additional gradient edge and corner tiles could be added dynamically to make it look much less boxy.

Alternatively, but a bit slower and at the cost of tying up more memory, a full black image could be drawn after the world map. That could be cleared dynamically with new transparency, similar somewhat to how land gets torn out in the `commonBonus/free-from-hobbygamedev.com/plane-src-examples-all/` examples when bombs drop or the plane crashes.

While the other approach of hiding any enemy units beyond the visibility of your army probably sounds simpler, because it wouldn't scale well for performance to have every unit check every other unit every frame,[27] the group's visibility would ideally be accumulated into something like a grid layer anyway, updated semi-periodically rather than constantly, which the enemy units could then quickly

[27] Confession: actually, at this scale, on a modern computer, we can probably get away with doing just that. Proof: `attackIfInRange()` is basically doing that now, at least for every unit that isn't already under orders to do something else, and could definitely be optimized to only check periodically instead of every frame.

use to look up whether or not they are in an area visible to the player. To avoid the appearance of units popping out of thin air, that kind of visibility system often includes some way of helping the player understand what's going by a visualization of how far their units can see, so it may not fully escape the need to implement something similar to the other form of fog of war.

### 6.3.3 Projectile attacks with splash damage

The heavy unit is already slow with long range - add some tuning tweaks to that unit to also make it have longer reload times and larger inaccuracy, and you've just about got an artillery unit. The only other missing element to make that complete is support in the game for splash damage. That means that rather than only damaging the unit targeted, all units within some radius of the point attacked lose `hitpoints`, possibly as a function of their distance from the epicenter of the blast.

**Details:**

The actual code for evaluating splash damage isn't too tricky. While a brute force solution of checking every possible unit to determine if it's alive, on the other team (unless you'd like to support friendly fire effects for splash damage, in which case ignore that check), and its distance to the attack point isn't very elegant, for something that happens only occasionally and takes up only one frame that's probably a fine way to do it here for now. Likewise introducing real inaccuracy to the shot's destination, rather than the purely decorative minor inaccuracy we imposed on the laser's end points, can be easily done by adding random offset ranges to the x and y target points to be used as the explosion's center.

Where this problem becomes trickier is in its representation. For an explosion, particle effects will probably need to be implemented, similar to the simple types discussed in exercises for the other games in this document. For the shot itself it may no longer be sufficient to do an instantaneous effect; traditionally in war games artillery is depicted as a distinct projectile object, visible as it sails through the air (height can be simulated with a shadow and vertical offset, like the ramping cars in Racing), imposing a noticeable delay between the moment it's fired and the moment it bursts on the ground a quarter to half second later. Getting that to work will likely entail adding a new class for projectile shots, which knows its position, target destination or heading, and weapon type to indicate its visual (the same class could easily be adapted for missiles, arrows, plasma balls, laser segments, lobbed grenades, etc.) and splash damage properties (range radius, amount of damage...). For a unit type that fires projectiles, the attack code will then need to be modified to add a `new` instance of the projectile class to a list of all projectiles, which someplace called every frame from `draw()` needs to be iterated through to update positions, graphics, and handle other logic for projectiles.

### 6.3.4 Draw order from front to back

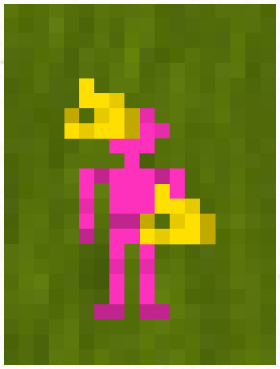
The order that units get drawn comes down simply to what order they happened to be added to the `ArrayList`, which means that half the time when two objects are standing close enough for their images to overlap the one which should be drawn behind the other may instead be the one drawn on top. This is one of those issues that's significantly easier to illustrate than to explain with words:

When a unit's y value positions it higher on the screen than something else, we also want it drawn first. In computer graphics that's called the Painter's Algorithm, since to get overlaps looking right a painter needs to draw background details before dealing with foreground elements that partially overlap them. Actually dealing with this turns out to be sort of a pain, but generally worth addressing, since as long as your game handles draw order wrong and you know to look for it it's going to annoy you.

**Details:**

As always there's a sloppy, fast, practical, mostly good enough solution that's pretty easy, and a more thorough solution that handles more cases.

The quick and easy fix to address most of these situations is to draw all the dead robots first, before drawing all the live ones. At least for the assets in this game so far, in which allied units tend to blend together when closer by one another, and opposing units move constantly and get defeated quickly when near one another, the problematic situation I've illustrated above is practically hypothetical. It just doesn't really happen, or when it does it doesn't happen long enough to be perceptible. What *would* happen, if this fix were not already implemented in the example code provided, far more frequently since defeated units are stationary and permanent on the field, is this:



Thankfully since rubble piles are so low to the ground, it's rare for rubble piles to get in each other's way. So from a practical standpoint, it works quite well enough to first draw all the scrap heaps for dead characters first each frame before then doing a second pass to draw all the living units. Not that further optimization would really be needed in this case, but as noted in a code comment around where that hack occurs in the provided source a relatively simple way to cut down on redundant comparisons would be to separate the dead units into their own `ArrayList` separate from `units`, say, `scrapUnits`, getting added to that new list and removed from the main one the moment they run out of health. That would also simplify some of the other attack code and such, since rather than checking before engaging each target to tell if it's alive, no defeated targets would even need to be considered. Again, that's not essential, but simple enough to do that it's maybe worth taking a crack at it.

If you want to handle the more comprehensive solution, it amounts to literally making sure that you're drawing the units in the same order as their y placement on the screen. There are several ways to go about doing that. One pretty simple approach conceptually is to have a `drawList` which gets cleared at the start of each frame, then objects get inserted into it during the logic updates in an order based on their y position on the screen (this means we can skip considering objects off screen), or just piled into that list and then sorted with a comparison function that evaluates solely based on y coordinate. As one last catch to this approach: so far in code we've been treating the origin as the center of the graphic, but for this type of carefully sorted draw order it would work better for us handle draw offset such that origins became the bottom-center of the character's feet, otherwise tall characters would get sorted out of order in the proximity of shorter ones. Per-type vertical offsets would then need to be added to the selection boxes and health bars to compensate for the variable unit sizes.

One upside to doing the more thorough and proper solution to this approach is that prepares your code to handle more impressive decorations. Once objects draw properly from the back to the front, you can have one or more types of decorative tree Unit entities that sit still and get destroyed by collateral damage. Scattering those around a region on the map that has a color key to function as wooded

territory, could really give more depth to the scene, in addition to making the world feel more alive by having destructible elements in it.

### 6.3.5 Non-random computer team behaviors

Up to this point, the enemy units have just wandered around, serving as practice dummies for the player to test whether their strategy is superior to... sheer randomness. That's not a very rewarding challenge to overcome! Program a computer opponent to manage the enemy robots at a strategic level that's more difficult to defeat.

**Details:**

A classic way to approach writing artificial intelligence is to think about what you're doing when you, a human player with some idea of what's going on, play the game. How, when, and why do you group units into teams of certain sizes? What are you accounting for when you're deciding on a good target area to approach, as opposed to a group that's more likely to win in an exchange? Especially involved are less mathematically clear considerations, like how to employ the heavy units as effectively as possible.

In principle at least, once you get those tactics and considerations programmed the computer should have a huge advantage over the human player, since it can rapidly micromanage with great precision and speed to perform focused fire, calculations about optimal squad size to approach a given target, and swapping damaged units out for healthy ones the instant that it makes sense to do so. Once it's working the armies can be balanced out - recall that in `randomReset()` within the Unit class by default we only gave the player 30% as many units as the computer. If the computer side still shreds through the player's army, consider tipping that balance even further in the human's favor, as a test of your AI programming. How much of a numerical advantage can you give yourself as the player and still be defeated by the AI you made?

### 6.3.6 Resources, build menu, and harvesters

Historically Real-Time Strategy games have been defined in part by their resource management: scouting to find resources (ore, lumber, oil, or some other valuable thing that might be scattered in the wild), sending out resource gatherers to collect and return with those resources, and spending those resources wisely to build more units in preparation for attacking the enemy's units and buildings. Besides adding more importance to recon, and another layer of strategic decision making in figuring out how to spend or save limited resources, this also created room for new tactics like going after the opponent's resource gatherers and supply rather than attacking their combat units directly, and counter strategies like needing to keep guards stationed at resource fields. While it's true that this is no small change to introduce to the game, relative to the ways it expands on the core gameplay, it's a pretty appealing tradeoff between relative coding difficulty and payoff in game depth.

**Details:**

The trick to taking this on successfully is to think about how we've handled other seemingly imposing challenges in putting this game together. First, think about ways you could utilize patterns you already know - dynamic collision grids, image pixel checking/erasing, getting units to move to a destination point - how might you choose to represent the resources in memory, and display them on the screen? Depending on how you opted to implement fog of war, if that was an exercise you took on, resources could be implemented a bit like a fog of war that only a special harvester unit can clear, in which doing so increments a team's available wealth.

The other thing to think about from how we've approached the other difficult challenges is to focus on just doing one thing at a time, making it work in the simplest form possible first, then iteratively improving upon it from there. Trying to sit down and architect a fully robust system supporting many building types, a smart harvester, and so on is biting off more than one person alone should be trying to chew at one time. Get one building working first, have both armies start with the building already there, and treat it as where the harvester needs to return collected resources. Get a harvester first to work at all without computer assistance, needing to be manually moved over resources then back to dump at the collection building, before trying to automate it. Each step of the way, look for a way to add just a little more to its functionality, and before you know it you can have a repairable building, defensive towers, buyable heavy infantry or replacement harvesters, and so on.

## 6.4 Write From Scratch Steps: RTS Game

1. Create a new, empty Processing Sketch.

2. Set window size to 640x480.

3. Create a new file, Unit, in which we'll put code for how the entities will behave. We'll first get one working, with globally-scoped variables not yet organized into a class, then refactor it into a class so that we can then manage many instances of it in an `ArrayList`.

4. In the Unit file add float `meX` and `meY` as its position coordinates, as well as a `boolean` named `isDead`.

5. Also in the Unit file add a `drawMe()` function that displays an 8x8 square centered on the position indicated by `meX` and `meY`. On the lines just prior to that `rect()` call set the color to bright orange if `isDead` is `false`, or darker orange if `isDead` is `true`.

6. One more function in Unit: add a `randomReset()` function that sets `isDead` to false and puts `meX`, `meY` into a random position within the top left quarter of the screen.

7. Back in the main source file call `randomReset()` in `setup()` after setting `size()`, and in `draw()` clear the background with a solid white rectangle as the first step of each frame, followed by calling `drawMe()` to display the test Unit.

8. Create another tab named mouseSelect. Add a `final int` in mouseSelect near the top of the file, call it `SELECTOR_RANGE`, and set it to 12.5.

9. In the Unit tab add a function called `mouseCheck()` which sets `isDead` to `true` if the unit is not yet dead and is also within `SELECTOR_RANGE` pixels of the mouse (check using `dist()`).

10. In the mouseSelect file add a `mousePressed()` function definition that just calls `mouseCheck()`.

*At this point, upon running the game, you should see one bright orange unit appear in a random position within the top-left corner of the screen, and clicking near it should cause it to darken. Now let's change the code so that instead of killing a unit when we click on it, we'll make it the active selection.*

11. Our first step in this direction will involve refactoring Unit to be wrapped in a `class` by the same name. Indent the variable and function definitions in the Unit file accordingly since they're now within a new set of enclosing braces. At the top of the main source file add

```
Unit testUnit = new Unit();
```

And elsewhere in the code any calls to functions defined in Unit should be preceded by `testUnit.` That means for example `drawMe()` in the main source file's `draw()` definition becomes `testUnit.drawMe()` while `mouseCheck()` in the `mouseSelect` file becomes `testUnit.mouseCheck()`.

12. At the top of the main source file add another Unit `newSelection` which defaults to `null`. At first we're only going to support selecting a single unit at a time, and when this is not `null` it will be a reference to the selected unit.

13. Make a new function in the Unit `class` called `showSelectionBox()` which turns off fill with `noFill()`, sets `stroke()` to black, and draws a 20x20 box outline by using `rect()`, centered on `meX` and `meY` as the position.

14. At the end of `draw()` in the main source file add a new `if` condition that calls `showSelectionBox()` on `newSelection` in the case that `newSelection` is not `null`.

15. In the Unit `class`'s `mouseCheck()` function replace the line that set `isDead` to `true` to instead set `newSelection` to `this`.

*Now when running the game, you should again see the bright orange square in the top left, but now clicking on it should select it by putting a box around it, rather than making it darker by killing it.*

16. In the Unit file add two more `float` values called `gotoX` and `gotoY`. These will indicate where the infantry is expected to move to next. When `gotoX` is the same as `meX` and `gotoY` is the same a `meY` the unit is stationary.

17. Add another function to the Unit file, called `gotoMouse()` that sets `gotoX` to the current value of `mouseX` and `gotoY` to the current value of `mouseY`.

18. Create a `mouseReleased()` function in the mouseSelect file. Inside that function check whether `newSelection` is not `null`, and if it is not `null` then call `gotoMouse()` for `newSelection`.

19. Back in the Unit file add a `final float` called `MOVE_SPEED` and initialize it to 1.5. This will determine how fast our unit moves each frame.

20. Add another function to Unit called `moveMe()`, and in it the first thing we're going to do is `return` in the condition that `isDead` is `true`, in order to bail out prematurely from the function and avoid ever moving after being defeated. But the heart of this function will consist of checking the distance between `meX`, `meY` and `gotoX`, `gotoY`, and if that distance is less than `MOVE_SPEED` setting `meX` to gotoX and `meY` to `gotoY` so it locks into its destination. If on the other hand the distance between the points is greater than the speed value, find the angle between the coordinates with `atan2()` and get `meX` and `meY` closer to `gotoX` and `gotoY` by `MOVE_SPEED` pixels by multiplying `MOVE_SPEED` by the `cos()` and `sin()` of the angle found between the points and adding those values to `meX` and `meY`.

21. On the line before `testUnit.drawMe()` in the main source file's `draw()` function, add a call to `testUnit.moveMe()` and test run the program again.

*Now when running the program clicking anywhere except the unselected character shouldn't do anything, then clicking on the character should select it, after which clicking anywhere on the screen should make the character move straight toward where the mouse clicked.*

*Next we'll add more units, separate one army from the other, and support attacking by clicking on a player team unit then an enemy team unit.*

22. For this next set of features we need to change from having only a single `testUnit` to an `ArrayList` containing many instances of Unit. In the main source

file on the line where `testUnit` was declared, replace that line with a declaration of an `ArrayList` named `units` and initialize it to a `new ArrayList()`. Add a `final int` named `UNITS_IN_NEW_BATTLE` at the top of the main source file and initialize it to 120. We're going to keep `newSelection` around because it's going to be helpful to use when we're selecting a single entity instead of a group, or when clicking on an enemy to attack them.

23. Write a new function called `resetUnits()` near the bottom of the main source file, and call that new function from `setup()` after `size()` instead of calling `testUnit.randomReset()`. In the `resetUnits()` function, empty the `units` list with `units.clear()`, then use a `for` loop to add `new Unit()` into the `units` list `UNITS_IN_NEW_BATTLE` times.

24. Put a constructor in the Unit `class` that calls `randomReset()` so that each entity will get a random position assigned when created.

25. Add a new `boolean` to the Unit class, and name that value `playerTeam`. In `randomReset()` set `playerTeam` to `true` 50% of the time using a comparison against a `random()` value. After determining whether `playerTeam` is true for this entity in `randomReset()`, use an `if` conditional to move `meX` and `meY` into the bottom right quarter of the screen if it's `false`, so that enemy units start in the opposite corner of the screen than the player's units.

26. In the `drawMe()` function of the Unit class, keep the `fill()` color orange if this unit is on `playerTeam`, otherwise use light purple.

27. Add another `ArrayList` to the top of the main source file, calling this one `selectedUnits`, and initialize it to `new ArrayList()`. In `draw()` instead of calling `showSelectionBox()` on `newSelection` we're instead going to use a `for` loop to iterate over every Unit referenced in the `selectedUnits` list and call `showSelectionBox()` on those instead. In `resetUnits()` empty `selectedUnits` with by calling `selectedUnits .clear()`.

28. Add a new Unit variable inside of the Unit `class` called `target`. This will be a reference to a unit that this one is supposed to chase and attack. If it is `null`, it signifies that this unit does not currently have an enemy unit to pursue. Set `target` to `null` in `randomReset()`. Also set `target` to `null` at the end of `gotoMouse()` so that when we issue a new move command to an attacking unit we can interrupt any previous instruction or entanglement.

29. Adjust the `mousePressed()` function in the mouseSelect file so that instead of calling `mouseCheck` on `testUnit` we will instead do a `for` loop over every entity in the units list, calling `mouseCheck()` on each Unit in that list. Before that `for` loop, set `newSelection` to `null`, so that if it's no longer `null` after the loop we know in the code that immediately follows that we've just now clicked on a Unit. After

that `for` loop, check whether `newSelection` is not `null`, and if it is not, check whether it's on `playerTeam`. If `newSelection` has `playerTeam` set to `true`, we clicked on an ally so clear the `selectedUnits` list then add `newSelection` to it so we leave the `ArrayList` as containing only a reference to the friendly unit just clicked on. If instead `newSelection` has `playerTeam` set to `false`, it means we clicked on an enemy unit, so if `selectedUnits` has any units in it, loop through each of them and set their `target` variable to point to `newSelection`.

30. At the end of `drawMe()` in Unit, check if `target` is not `null`, and if it is not then draw a gray `line()` from `meX`, `meY` to the target's `meX` and `meY`. This will help show visually that the unit has a target identified, regardless of whether our code for its movement operates as expected, and will generally be helpful for debugging when we start getting multiple units active at once by giving clues of whether we're staying fixated on a defeated unit, somehow got assigned in a mix up to attack a unit on our own team, etc.

31. Add a new `final float` to Unit called `ATTACK_RANGE` and initialize it to 30.0 in addition to another `boolean` called `isAttacking`. Set `isAttacking` to `false` in `randomReset()`.

32. In `moveMe()` inside the Unit class after we passed the chance to quit the function based on `isDead` but before we update our position based on `gotoX`, `gotoY`, first set `isAttacking` to `false` and check whether `target` is not `null`. If `target.isDead` is `true` then set `target` to `null`, so that we should forget a target that gets defeated, and set `gotoX` to `meX` and `gotoY` to `meY` so we'll cease pursuit. Otherwise, meaning `target.isDead` is `false`, we are targeting an active enemy and should engage. In that case compute the distance between `meX`, `meY` and `target`'s `meX`, `meY`. If the distance is less than or equal to `ATTACK_RANGE` then set `isAttacking` to `true` in addition to settling in at position by setting `gotoX` to `meX` and `gotoY` to `meY`. Otherwise if too far away to attack yet, set `gotoX` and `gotoY` to just inside the `ATTACK_RANGE` radius of pixels from the target's position at it's `meX`, `meY`.

33. Just below those `target` and distance checks, see whether `isAttacking` is `true`, and if so call `target.damage()`. This will get more sophisticated one we add hit points to our units, but for now just create a `damage()` function in the Unit `class` that sets `isDead` to `true` and `target` to `null`.

*At this time we should have units that can be clicked on directly to be selected, if on the player team, after which clicking on an enemy team unit should send out unit straight to it (drawing a line to show the attack vector), defeating that targeted unit once we're in range. Our gotoMouse() behavior isn't being called any more in the expected manner, however we'll fix that as part of this next code.*

*Next we'll incorporate lasso selection so we can select multiple units at once.*

34. Create 4 more `int` values at the top of the mouseSelect file: `lassoX1`, `lassoY1`, `lassoX2`, and `lassoY2`. Those first two will correspond to the starting corner of our box selection lasso, the second two will correspond to the position of the corner we're dragging to from the start position.

35. Add a function just below those values called `resetMouseLasso()`, and in it set all four of the lasso `int` values to 0. Call `resetMouseLasso()` in `setup()` in the main source file, and also at the end of `mouseReleased()` in mouseSelect.

36. Create a `drawMouse()` function in the mouseSelect file which turns off fill, sets stroke to black, and draws a circle centered on `mouseX`, `mouseY` with the width and height set to `SELECTOR_RANGE+2.0`. After that, in the same function, check if `lassoX1` is not 0 and `lassoY1` is not 0, meaning that we're in the middle of clicking and dragging, and in that case set `lassoX2` to `mouseX` and `lasseoY2` to `mouseY`, drawing a `rect()` on the next line starting at the corner `lassoX1` and `lassoY1` with the width as `lassoX2-lassoX1` and the height as `lassoY2-lassoY1` (recall that `rect()` does not want the coordinates of each corner, but the corner followed by a width and height of the shape!). Call `drawMouse()` at the end of `draw()` in the main source file.

37. In `mousePressed()` also within mouseSelect, let's add the code to begin the lasso drag. Basically if we didn't click directly on a friendly unit, or if we clicked on an enemy unit but don't have any friendly units, then we want to start a lasso drag. Create two `else` cases (one for if `newSelection` was `null`, the other for if the selection wasn't a player and no friendly units are currently selected), each setting `lassoX1` to `mouseX` and `lassoY1` to `mouseY`.

38. Now to fix `mouseReleased()` and our `gotoMouse()` function usage. In the mouseSelect file at the top of `mouseReleased()`, check whether `lassoX1` and `lassoY1` are both equal to 0, and if so, `return` from that function because that means we've already selected a player or attacked an enemy during our logic in the `mousePressed()` function. Otherwise check if the distance from `lassoX1` and `lassoY1` to `mouseX` and `mouseY` is less than some small value (8.0 works) to determine that the mouse hasn't moved very far from where it started, and we can handle this case by ignoring the lasso rectangle and treating it as a click in open space away from either an allied or enemy unit, which is when we need to issue move commands. To do so, loop through all units in the `selectedUnits` list and call `gotoMouse()` for them.

Otherwise, meaning that the lasso has been dragged more than 8 pixels from where the mouse button when down, it's time to update our selection `ArrayList` to be composed of whatever player team units are within its boundaries. The first step to doing that is to ensure that `lassoX1` is left of `lassoX2` and that `lassoY1` is above `lassoY2`, since that will greatly simplify the comparisons we need to check against the units to see which are inside the box. To prepare those variables and ensure they're each on the right side, declare a local `int` value there called

`tempSwap`, and if the variables are not on the sides that they should be relative to one another, use `tempSwap` to trade them (values `A` and `B` can be swapped using temporary value `C` by doing `C=A; A=B; B=C;`). Next empty the `selectedUnits` list so that units caught in our selection box will be the only units selected after we released the mouse. Lastly, loop through all units, checking whether each unit is on the player's team and also `isInLasso()` is `true`, in which case add units meeting those criteria to the `selectedUnits` list. Define `isInLasso()` as a helper function in Unit like so:

```
boolean isInLasso() {
    return (meX<lassoX1 ||
            meX>lassoX2 ||
            meY<lassoY1 ||
            meY>lassoY2) == false;
}
```

Conceptually that function is just a way of asking about the unit in question: am I left of the lasso's left edge, right of the lasso's right edge, above the lasso's top edge, or below the lasso's bottom edge, because any of those would mean I'm outside the lasso, and if none of them are `true` then I must be inside the lasso and will therefore return `true`.[28]

39. Units now should be selectable by clicking and dragging, however moving them to a new destination will cause all to converge on the same point and overlap, after which they become quite tricky to spread back apart. Instead, let's program them to fill in a formation when directed as a group. To do this we need to improve the `gotoMouse()` command, and calculate the group formation's dimensions in `mouseReleased()` over in the mouseSelect file.

The gist is that we want to figure out how many columns we want the group to line up into to yield about the same number of rows, so that whether we're moving 4 selected units at a time, 9 units, or 36 units, they're stay in a roughly square formation. If we have 4 units we want 2x2, if we have 9 units we want 3x3, if we have 36 units we have 6x6... and in case you're not picking up on the pattern, this is one of those weird occasions when for entirely practical reasons we want to use square root (the `sqrt()` function) in our code. We're working with whole numbers, since units can't be cut into fractions to form a nicer square, so the value will need to be truncated and cast to an `int`. The other part of this question then is thinking through (or experimenting with) how we want to handle

[28] I went back and forth on whether to instead write this in the easier to read positive case, which considers if it's inside all four boundary edges, rather than how it's written here, checking if it's not outside any of the boundary edges. This form, though awkward, I believe can help simplify later reasoning by extrapolation to things like detecting whether two rectangles overlap (asking questions like is my right side left of its left side turns out to be a more productive approach).

more general cases when exact square roots don't work out. I found that adding 2 to the number of selected units before taking the square root yielded group dimensions closer to what I found natural, but your mileage may vary so feel free to experiment and try other values there.

We then need to pass both that formation dimension and the unit's number in the formation to the `gotoMouse()` function. Here's how that changes the code in `mouseReleased()` for when the distance between the lasso origin and the mouse coordinates is under eight pixels so we're treating it as a move input:

```
if(selectedUnits.size() > 0) {
  int formationDimension = (int)sqrt(selectedUnits.size()+2);
  for(int i=0;i<selectedUnits.size();i++) {
    Unit eachUnit = (Unit)selectedUnits.get(i);
    eachUnit.gotoMouse(i,formationDimension);
  }
}
```

Then over in the Unit file's `gotoMouse()` definition:

```
void gotoMouse(int formationPosition,int formationDim) {
  int rowNum = formationPosition / formationDim;
  int colNum = formationPosition % formationDim;
  gotoX = mouseX + 10.0f*colNum;
  gotoY = mouseY + 10.0f*rowNum;
  target = null;
}
```

To be fair, yes, this kind of looks insane, especially if you haven't see the modulus (%, it finds the remainder) operator used very often. The math really isn't that bad though. Let's talk through it, beginning in the code that's calling `gotoMouse()`. First it's deciding how many columns wide the group should file into, by using square root, as discussed earlier. Let's say there are 15 units, and since `sqrt(15+2)` is 4.12, casting that to an `int` we've decided to allow 4 soldiers per column. Each soldier then gets told two numbers as part of its move instruction: what its position number is in the formation, and how wide the formation is allowed to be. By dividing the position number by the width, the soldier can decide how many rows down it should stand (formation width is 4, and because 3/4 truncates[29] down to 0 as an `int` soldier #3 is 0 rows down, because 6/4 truncates down to 1 soldier #6 is 1 row down, because 14/4 truncates down to 3

[29] Rounding takes into account whether a decimal part of a number is closer to the number below or the number above. Truncation, which is what happens when we cast a `float` to `int` or perform division and save it as an `int` just disregards the decimal part of the number, so even 3.997 would be truncated to 3.

soldier #14 is 3 rows down[30]). Whereas division calculated how many rows down the soldier should stand, because the modulus (%) finds the remainder of division by a number, we've used it here to ask how many columns in we should stand from the edge.

Still perplexing? Think back to elementary school math, when we used remainders instead of decimal values when doing division. That's all we're doing here.

Here, check this out:
What's 0 divided by 4? 0, remainder 0. Soldier #0 is row 0, column 0.
What's 1 divided by 4? 0, remainder 1. Soldier #1 is row 0, column 1.
What's 2 divided by 4? 0, remainder 2. Soldier #2 is row 0, column 2.
What's 3 divided by 4? 0, remainder 3. Soldier #3 is row 0, column 3.
What's 4 divided by 4? 1, remainder 0. Soldier #4 is row 1, column 0.
What's 5 divided by 4? 1, remainder 1. Soldier #5 is row 1, column 1.
What's 6 divided by 4? 1, remainder 2. Soldier #6 is row 1, column 2.
What's 7 divided by 4? 1, remainder 3. Soldier #7 is row 1, column 3.
What's 8 divided by 4? 2, remainder 0. Soldier #8 is row 2, column 0.
What's 9 divided by 4? 2, remainder 1. Soldier #9 is row 2, column 1.
What's 10 divided by 4? 2, remainder 2. Soldier #10 is row 2, column 2.
What's 11 divided by 4? 2, remainder 3. Soldier #11 is row 2, column 3.
What's 12 divided by 4? 3, remainder 0. Soldier #12 is row 3, column 0.
What's 13 divided by 4? 3, remainder 1. Soldier #13 is row 3, column 1.
What's 14 divided by 4? 3, remainder 2. Soldier #14 is row 3, column 2.

That's all 15 soldiers accounted for (again, we have a soldier#0 as the 1st, so soldier #14 is the 15th). See how nearly they each sorted into rows, 4 for each row, then each took a separate column within that row based on the remainder of the division?

Don't feel bad if it still seems strange. It takes some getting used to! Let it sink in for awhile. Go away and come back to it later. Like a lot of other math that we use in game programming, it's one of those patterns that begins to find its way into being relevant every so often, and after awhile you get acclimated to it, maybe even recognizing its relevance in a practical problem you're working on, like trying to figure out the x and y coordinate of the 127930th pixel on a 640x480 bitmap (it's at 570 x and 199 y, also figured out by division and modulus of the row's width, which is 640 in that case) since `loadPixels()` treats an entire bitmap as a one dimensional array of color values.

[30] In typical computer science fashion, numbering starts at 0 in arrays and for loops, so there is a soldier #0 in this scheme, and that robot is the one standing in the top-left position directly where the mouse clicked.

Anyhow, the units should now neatly file into an orderly group when a handful are selected and moved around. Totally worth it.

*So far the enemies haven't done much. Let's at least get them to attack the player units if we get too close to them, and then just program them to wander around aimlessly.*

40. We're going to write two functions to help breathe life into the enemy units: `attackIfInRange()`, which will apply to player units too, and `aiUpdate()`, which will only apply to computer team units. Inside `moveMe()` call `aiUpdate()` if `playerTeam` is `false`, and after that, outside the `playerTeam` conditional, call `attackIfInRange()` as part of all unit movement.

For defining `attackIfRange()`, bail out of the function with `return` if `isAttacking` is `true`, or `target` isn't `null`, or `meX` isn't equal to `gotoX` or `meY` isn't equal to `gotoY` - in other words, if this unit already has any other active move or fight orders underway, don't pick a new target. Otherwise iterate over the `units` list, checking each one, and if it's alive, on the other team, and within attack range from our position, make it our target then `break` out of the loop. Now when two units get close enough to one another they should auto-target one another.

41. For `aiUpdate()` we'll need to define some new tuning values at the top of the Unit file:

```
final int MIN_AI_RETHINK_FRAMES = 24;
final int MAX_AI_RETHINK_FRAMES = 70;

final int MIN_AI_WANDER_RANGE = 35; // in pixels
final int MAX_AI_WANDER_RANGE = 100;

final int AI_MOOD_WANDER = 0;
final int AI_MOOD_ATTACK = 1;
final int AI_MOOD_TYPES = 2;
final float AI_TARGET_RANGE = ATTACK_RANGE*4.0;
```

Plus a two new `int` values named `frames_AI_Rethink` and `aiMood`. Set `frames_AI_Rethink` to 0 in `randomReset()`.

In `aiUpdate()` decrease the value of `frames_AI_Rethink` by 1, and if `frames_AI_Rethink` is at or below zero, bump it back up by a random number between `MIN_AI_RETHINK_FRAMES` and `MAX_AI_RETHINK_FRAMES` - this ensures that the AI only updates its thinking irregularly, giving the group a much more organic behavior. Inside that same condition, when this unit has waited enough frames on its previous behavior to be ready for a goal update, is where all the AI update logic goes. The first step of that will be randomly assigning a new mood, using `aiMood = (int)random(AI_MOOD_TYPES)`. Though there are only 2 modes for the AI we're working on for now. This makes it easy to build in other behaviors later, such as

retreating, or going into a base defense patrol. If the new value of `aiMood` is equal to `AI_MOOD_WANDER`, then pick a new goto position in a random direction between `MIN_AI_WANDER_RANGE` and `MAX_AI_WANDER_RANGE` pixels away from the unit's current position. Because that random new position may be off screen, write a `boundsCheck()` helper function that will fit `gotoX` and `gotoY` within the playable area if they're off the screen when the function gets called. If instead `aiMood` is equal to `AI_MOOD_ATTACK`, loop through all units and target the first one found that is alive, on the player's team, and within `AI_TARGET_RANGE` (defined above as several multiples further than the `ATTACK_RANGE`, which means in this case the AI will go looking to pick a fight).

Now that the computer army can harm the player's units, there's another small check we should add to the code: let's deselect any unit that gets shot down while selected, so that we don't wind up looking like we can give orders to crumbled robots. To do that, let's go to where the `showSelectionBox()` function gets called on all selected robots - after all, this is something we only need to check for units that are currently selected, and that's the same `ArrayList` we're looking through at that part of the code to draw the selection boxes. Right there in the code we can check to see whether the unit that we're about to draw the selection box for has `isDead` set to `true`, and if so, remove its index from `selectedUnits` with `selectedUnits.remove(i)` instead of calling `showSelectionBox()` on it. There's one catch to this though: in order to properly remove an entry by its index while we're looping through an `ArrayList` like this, we should loop through the `ArrayList` backward, like so:

```
for(int i=selectedUnits.size()-1;i>=0;i--) {
```

Why is that? Well, think about what happens if we remove the entry in position 4. There would still be an entry at position 4 in the `ArrayList`, but it would be the one that was formerly in position 5, and if we `i++` to access the next entry ahead, we'll have skipped what was in position 5 on that iteration. By going backward through the list, starting at the last entry and working down to the unit in position 0, when we remove an position index from the `ArrayList` we've already dealt with all higher indices in the list, so the shuffling around won't cause us to skip processing any of them. It's perfectly safe to try that loop going the normal direction, and with all units selected you'll notice a subtle flicker when any of them get defeated, but this is a good thing to be aware of since in other cases where changes happen more frequently, the effect of skipping entries can be much more distracting or may introduce cumulative errors in synchronization.

42. So far, units just destroy one another the moment they attack. Instead, let's introduce health bars and slightly randomized reload delays between attacks. Add a new `int` to the Unit, `hitpoints`, and a `final int` for tuning that identifies its max value as 4. Set `hitpoints` to that max value in `randomReset()`, and let's change damage so that instead of just always setting `isDead` to `true`, instead decrement `hitpoints` there, after which do a check: if `hitpoints` is at or below 0,

then set `isDead` to true and `target` to `null`, otherwise (meaning this unit survived the attacks damage) if `target` is `null` (meaning we're not currently under orders or previous plans to target someone else) then set `target` to `fromUnit` so that we'll go after whichever unit harmed us. Where do we get ahold of this `fromUnit` reference that I speak of? We're going to pass it in from our attacker. Make it an argument on the `damage()` function, taking the form of `damage(Unit fromUnit)`, and down where we later call `target.damage()` in the code change that to `target.damage(this)`. Now when one unit attacks another, it shares its identity directly with the attacked, so that if the attacked survives and wants to fight back it knows what to target

43. The other thing we need to do with `hitpoints`, of course, is to display them. Write a `showHealth()` function in the Unit class that draws 4 black outlined squares above the unit, filling one in with yellow for each hitpoint remaining and the rest with black. Call `showHealth()` from `drawMe()`.

44. For reload, define a min and max `final int` as `MIN_RELOAD_FRAMES` and `MAX_RELOAD_FRAMES`, set to 10 and 15 respectively. Also add another new `int` value called `framesReloadTime`. Look for where we have an `if` conditional on `isAttacking` in the Unit's `moveMe()` function. On the line just before it, decrease `framesReloadTime` by 1. Inside the `isAttacking` conditional only allow the `damage()` call to be sent `if(framesReloadTime <= 0)` and then alongside that damage code bump `framesReloadTime` to a random value between `MIN_RELOAD_FRAMES` and `MAX_RELOAD_FRAMES`. This has the effect of limiting the frequency of the attacks, so that health will matter, as otherwise it would be wiped out in 4 consecutive frames (less than 0.2 seconds!).

*Final stretch! Almost done!*

45. Units currently stand there staring each other down while firing, which looks very orderly and not at all like a battle is taking place. Define two more `final int` values, `MIN_DODGE_RANGE` and `MAX_DODGE_RANGE`, and each time the unit fires set a new `gotoX`, `gotoY` position within a random distance between those two values from where the unit was when it fired, giving motion while it reloads. Since having a target overrides prior move orders, also add another variable to the Unit class that's a `Unit` reference called `targetBetweenDodge`,. Upon entering an evasive maneuver, `target` can be set to `null` after saving its reference to `targetBetweenDodge`. If `targetBetweenDodge` has a non-null value when the unit reaches its `gotoX`, `gotoY` position, we can then restore `target` from `targetBetweenDodge`, followed by setting `targetBetweenDodge` back to `null`.

46. One of the problems discussed in the exercises is that defeated units sometimes get drawn on top of active units. To fix this issue, separate the unit draw code into two passes: the first which draws all defeated units, and the second which draws (and updates the logic for) all active units.

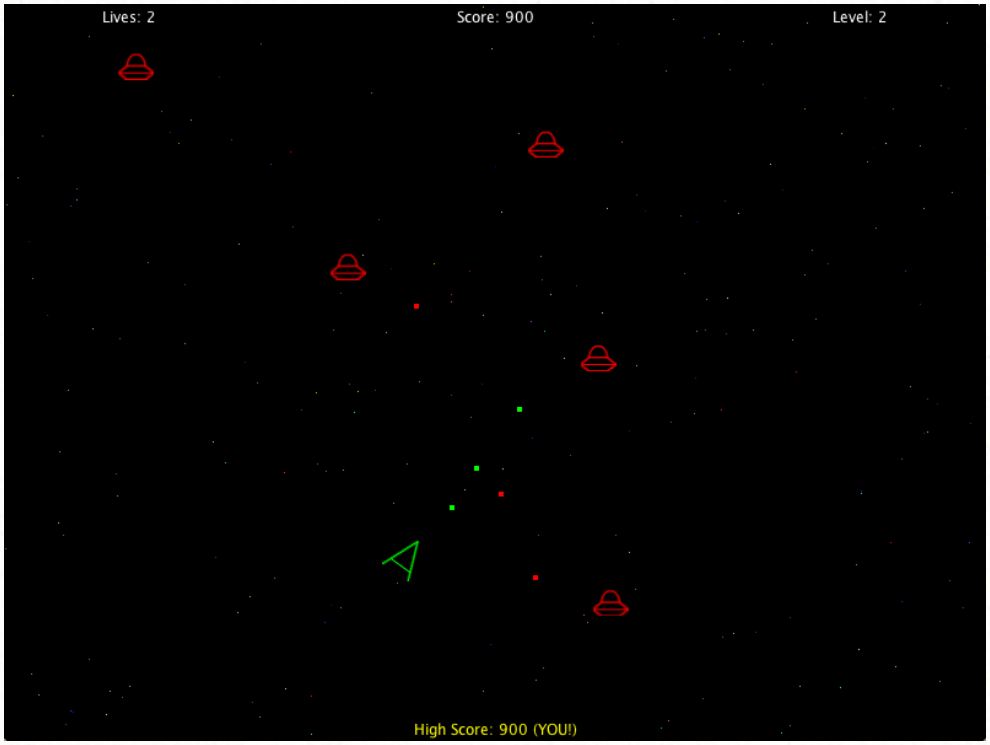
47. Add two `int` values in the main file, one for how many units are active on the player's side and one for how many are active on the enemy's side. Before the `for` loop that calls the move and draw code on all `units`, rest both counters to zero, then increment one or the other (based on whether `playerTeam` is `true` for that unit) for each unit found still alive. Display those numbers on the screen as part of a message using `text()` at the end of `draw()`.

48. When either army has lost all of its units, add `text()` declaring which side is the winner and noting that spacebar can be pressed to reset. Implement a `keyPressed()` function in the main file that detects spacebar presses, and if either army has zero units remaining when the spacebar is pressed reset the round.

49. There are some non-trivial interactions to know for playing this game! Add some instructions text in the corner of the screen, also using `text()` near the end of `draw()`. Since it takes up a lot of screen space to write all the descriptions of input commands, make the last line indicate that the player can press h to hide the text. In `keyPressed()` also check for the h key, and if it gets pressed toggle a new `boolean` value that flips from the full help message to a one line message indicating only that h can be used to reveal the full help.

50. Whew! That was *a lot* to work through. Thanks so much for your patience and determination. If you've opted to take this on before doing the challenges, in order to do the project totally from the ground up: good luck with the exercises!

# 7 Space Battle: Full Game in Steps




## 7.1 About the Seven Code Steps

Space Battle is provided in seven different versions, or steps. Each of these demonstrates a different aspect of making progress on videogame development, with the specific purpose of providing more conceptual tools for how to break down projects of your own original design into manageable steps.

The steps provided are as follows:

### 7.1.1 spaceBattle0: First Code

Player input and movement are often a great place to start. This step of the project accounts for getting some basic functionality all together in the program: handling real-time input, loading and displaying an image, and spinning that image based on the input. It's not fancy, but until this can be figured out there isn't much sense in proceeding with trying to create a whole game. Once these basics are out of the way the elements already in the code for handling image loading and rotation can be easily reused.

### 7.1.2 spaceBattle1: Core Gameplay, Section Comments

This version pulls together one player ship, one enemy ship, and one projectile each. It accomplishes this with code that's nearly as unstructured as can be. With the exception of the keyboard functions, the code basically just runs from top to bottom, without division into classes, files, or even functions.

When I first started making videogames, my code looked something like this. I'm including it here for a few reasons. (1) I wanted to demonstrate that even using just a handful of common programming elements simple gameplay can be put together. (2) Much of the additional work that we put into programming and employing organizational code features has more to do with improving code readability than with accomplishing functionality that we couldn't have otherwise. (3) Grouping related code under comments that describe the functionality of each chunk, and coming up with some consistent naming conventions for variables is a good step toward wrangling an otherwise untamed mess of code into something that will be easier to clean up later.

### 7.1.3 spaceBattle2: Splitting Into Functions

This version of the project exhibits the exact same functionality and appearance as the previous phase, but the code has now been carved up into functions. In the previous step, code was grouped under comment headers, which here have been used as the function labels. While functions are capable of accepting parameters and returning values, a simple `void` function that takes no arguments, as is used throughout this project phase, simply acts as a handy label on a chunk of code that can then be "called" by that label elsewhere in the program.

One of the main benefits of doing this is that whereas code in the previous version was a large, unbroken chunk, code now reads in many places as a sequence of descriptive function names. This can make it easier to follow the logic of the code, to comment out particular parts of the functionality for testing purposes, or to find where certain behaviors get implemented in case you'd like to work on improving them. When `if` statements cause entire blocks of code to be skipped, having that code wrapped up in its own function can make it much simpler to see what the scope of that logic is skipping or including.

Another reason to break code into functions, even when not utilizing arguments or return values, is to minimize redundancy or copy/paste propagation of errors. That benefit isn't as applicable to the code here since we're not really calling any of these functions multiple times, but the power of doing this becomes more clear in later steps when we further generalize some of the redundant code (player shot and enemy shot) to share the same functions and variables.

### 7.1.4 spaceBattle3: Organizing with Files

This step doesn't even change the typing from the previous one, yet it still manages to improve upon the project's readability. The step this time consisted of

taking the functions we just wrote and dividing them up into separate files, grouping together similar functionality based on what the code applies to. We wind up with one file dedicated to player functions, and another to the enemy UFO. One file has just the keyboard code, and another contains the main tuning variables for tweaking how the game plays. While splitting up the functions into their own files I also moved the related variables mostly used by those functions to the same files.

This rearranging isn't strictly necessary. Once again the program does exactly what it did a couple of steps ago. Processing basically just joins all these files together as one long file when it's time to test the game. But every layer of organization that we apply can help us keep our thoughts straight about what we're working on, can make it faster and easier to find a particular part of code we are looking for, and gets us closer to being able to think about one piece of code at a time while being able to mostly ignore the rest.

(To create, rename, or remove a file tab in Processing, click on the downward arrow in the circle to the right of your current file's tab. Processing will then provide a naming prompt down below for the new file. Upon inputting the name a new empty file will be created in your project's directory. All source files in the same directory as your project's main file, a status granted to the file that matches the containing directory's name, are automatically opened with it as part of the same project.)

### 7.1.5 spaceBattle4: Refactoring to use Classes

Here we pull in another important programming concept: classes. Unlike the step we just finished, which focused on rearranging the functions but didn't actually change any of the typing, organizing our code into classes involves getting our hands a bit dirty updating a bunch of the code.

Up until this point we've largely been using consistent naming prefixes on variables and functions to keep straight what each is intended for. `evil` at the start of a variable or function meant it was for the enemy UFO, while `player` was a signal that the value or code applied to the player's ship. This was necessary since otherwise our whole program would have a bunch of position and speed variables without any way to tell which was meant for the player's shot as opposed to the enemy's spacecraft.

When we clean up our code to use classes, what we're doing isn't conceptually much of a change what we did when splitting the variables and functions into different files. One important difference is that by enclosing variables and functions inside a `class` we no longer have to worry about those being accidentally confused with variables and functions of a different `class`. The implication of this is maybe easier to understand than the description: Rather than needing to separately store variables named `playerX` and `evilX`, as the horizontal positions of the player and enemy respectively, the player and enemy can now both track their own version of `x`. Name conflict or confusion is avoided since functions in either `class` by default refer to their own variable by that name, and can only access that variable on

another object by preceding it with a clarification of whose variable we meant to reference.

The other quirk to moving from a file-only organization into different classes is that for each `class` it's not enough to define it - we need to declare an instance of it in order to use it. A `class` is really just a description, a template, a definition for how certain data types can be pulled together with related functions under a common label. To have that `class` store and manipulate actual data we need to add a few extra lines of program code to clarify what label we intend to use for that instance of the `class`. So whereas we only define the enemy UFO `class` one time, when we want to have more than one UFO soon we'll be able to declare multiple instances of that same `class`, rather than needing to duplicate that definition for each UFO.

Lastly, it's worth highlighting that just because many of the files are handled as classes, that doesn't mean everything in the program now is. The keyboard functions, `keyPressed()` and `keyReleased()`, as well as the main initialization and update functions, `setup()` and `draw()`, are functions that Processed is expecting to find at the global, top level, and wrapping those up into a class wouldn't really make sense. Those remain in their previous state, while we're wrapped up those variables and functions that do make sense as a contained object into classes.

### 7.1.6 spaceBattle5: Arrays and Score Display

Here's where we finally can see (and play) a real change to our program's functionality compared with how the game looked in its second phase. With the shot and enemy UFO classes defined, we can manage an `ArrayList` of instances of them. That may sound complicated, and though it will take a few extra lines of code to get that working, it is in concept a pretty simple idea: We'll want many enemy ships and many shots on the screen!

An `ArrayList` can be thought of as a container for `class` instances. Anything we've defined as a `class` we can many of at once, by keeping track of them all in an `ArrayList`. Conceptually, the steps to use an `ArrayList` in this way are to declare the `ArrayList` variable, followed by initializing it to be a new empty `ArrayList`, after which we can `add()` new instances of basically `class` to it. Then we can later loop through the whole list, doing something with or to each one: moving it, drawing it, checking it for collision against the player's projectile, etc.

### 7.1.7 spaceBattleFinal: Title Screen, High Scores, Levels, Stars

In terms of basic gameplay this version has a lot in common with the previous code, though some additional work has gone into making it more presentable. The game now has a title screen, shows high scores (which get saved to the drive between plays), a level progression, limited lives, and decorative stars that twinkle in the background. When making videogames, it's never enough to just make the gameplay work. In order to be received in a meaningful context by our players we need to package that gameplay in a frame that fits their expectations. Polish and presentation really do matter.

In terms of code organization, this last stage of the project utilizes the `extends` keyword to share common variables and functions between derived classes. Previous steps still had a bit of code copy and paste just to get two different objects to, for example, wrap when passing the screen edge. Now WrappingPosition is its own class dedicated to having an `x`, `y` position and a function to `updateWrap()`. By using the `extends` keyword in the declaration of the LaserShot `class`, LaserShot doesn't need its own `x` or y variables declared, nor its own definition of `updateWrap()`, as those get inherited from WrappingPosition. This can also be done in a layered approach, as for example the way that ArmedSpaceCraft (handles the object's shots in an `ArrayList`) `extends` WrappingPosition (meaning it also keeps a position and can wrap), and in turn both the Player `class` and the UFO `class` have an `extends` relationship to ArmedSpaceCraft.

One of the more subtle changes from the previous versions to this final step is how the program code deliberately limits the number of simultaneous shots that the player is allowed to have on screen. As demonstrated in the previous phase, the code here is flexible enough to allow the player to fire constantly, putting as many shots on the screen at once as he or she desires. When early coin-operated videogames were made in the late 1970s and early 1980s they frequently had a strict limit on the number of simultaneous projectiles the player could have on screen at a time. What emerged from that technical limitation was a game mechanic that holds up over time: the player has a strategic decision to make between firing in a tight burst or spreading shots out, accuracy is rewarded with being able to shoot again sooner, and missing is penalized by leaving the players more exposed.

I've added a couple of extra lines of code to the player's shooting functionality to recreate that limitation, since I think it actually plays much better this way than when the player can (and therefore basically should) just hammer constantly on the shoot key. That said, if it's not the way that you like it, or if anything about the way that I prepared this project is not the way that you like it, change it to suit your liking. That's the magic of having the entire source code. You can make this game do anything that you'd like.

But first, in case you'd like some guided warm-up, practice, and challenges before endeavoring to make your own original adjustments, here are the exercises I've prepared for Space Battle, intended to be done with Space Battle Final as the starting point:

## 7.2 Warm-Up Exercises

### 7.2.1 Custom graphics

In terms of programming, this exercise isn't very difficult - it doesn't even involve changing any of the code! Well, it may involve a small amount of code depending on what customization you decide to apply, but it can be done without programming changes. The difficulty here will be conceptual, in deciding how

you'd like the ship, UFO, and projectiles to look, and testing your ability with an image-editing program (Photoshop, `GIMP.org`, etc.). A program that supports image transparency should be used, to avoid a black or white rectangle around the graphic. Additionally for this reason the image type should be kept in PNG.

**Details:**

Look in the data folder of this sketch for the player.png[31] and ufo.png files. Replacing these images will change the ships in the game the next time it is played, without any updates to the code. The player ship is drawn pointing right and gets rotated in code, so unless you're going for a radially symmetric design like a saucer it will be important to keep viewports facing right and thrust pointing left. On the other hand, since the enemy ship is not reoriented while moving, an image should be used that's still appropriate for sliding around without changing the way it's facing.

There are a few places where you may opt to operate a bit on the code for this exercise. The first is commenting out, or at least adjusting, the player ship's dynamic thrust. In the player tab's `drawRocketAndShot()` function look for `if (keyHeld_Thrust)` which is where the yellow flickering flame gets drawn while the player applies rocket force. Another possible spot for code addition would be to use an image for the projectiles rather than a simple `rect()` call. If going for a directional projectile like a laser line or rocket, the angle for rotation can be calculated from its movement as `float ang = atan2(yv,xv);`

### 7.2.2 Put star code in its own file

The sparkling stars in the background are a simple `ArrayList` initialized with hundreds of instances of the `PVector class` (a container for `x,y` position that supports common math operations). The variables and functions relating to those stars are currently intermixed in the main source file. Create a new file/tab in the project for Stars and move the appropriate star-related code there.

**Details:**

This is the same transformation as spaceBattle2 into spaceBattle3, except it only needs to happen to the stars since it's already handled for everything else. The code you're moving should include the relevant variable definitions, meaning for example where an `int`, `float`, or `ArrayList` is declared, as well as the definitions of the functions, however references to some of those variables or calls to those functions may still remain in other sections of the code.

[31] If you only see player and ufo, without the .png extension, it's worth doing a web search for how to set your computer to no longer hide extensions on known file types. The extension characters on a file are very helpful for a programmer to distinguish format differences, and also have to be known to import a file in code.

### 7.2.3 High scores list

Currently the game only stores, displays, saves and reloads a single score value. Having only a single high score on a game can be a bad idea since it may be a fluke performance, unlikely to be exceeded. It's fine while playing to only show the highest score so as to not take up too much space, though the title screen should list the best 10 scores or so. The top entry on that list should be the highest earned yet, with the next being the second highest and so on, with the last entry shown being the lowest value. Fill the high scores list initially with 0's so that the first 10 plays after implementing this feature will get scores listed.

**Details:**

The `loadStrings()` and `saveStrings()` functions that are used to export and import scores from an external file are already designed to handle multiple lines of values. The main changes will be to how the scores get displayed in the game, which will likely involve a `for` loop and an adjustment to the positioning of that information on the screen, and the addition of some basic sorting functionality to keep scores listed in the right order. For a tad extra challenge, if the player's most recent round set a score on the top 10 list, call out that line by coloring the text differently.

## 7.3 Practice Exercises

### 7.3.1 Different ship types

Although currently the player has the same ship for the entire game, it actually takes surprisingly little technical work to create a different ship for the player: Just use a different image, and somehow vary its performance variables. In addition to the default craft also create a nimble scout ship, a heavy attack ship, and an stealth fighter.

How to determine which ship the player uses is a design choice totally up to you. Perhaps every two stages put the player in a different ship (repeating in a cycle), so that the more interesting ships keep the game feeling new as the player makes progress? Or let the player pick their click by clicking on a different ship icon at the title screen? A hybrid approach could be to let players select their ship, but require a high score above a certain value for each of the non-default ships to become available, using them as a reward for improved play.

**Details:**

Think about how the ships might differ in terms of functionality: turn rate, acceleration and deceleration are the main ways to differ, though you could also affect variables like how many shots the ship can have on screen at once, or the how close enemy shots have to get to count as scoring a hit. It's possible to experiment in code first to come up with the different ships before you prepare additional graphics, or the different graphics could just be adjusted copies of the default player ship that tweak the color and size of the image.

If you decide to implement the ships in a way that will be locked by achievement, whether based on highest score achieved or based on the level number currently played, it will be incredibly useful for testing purposes to put in a few simple cheat buttons that will instantly unlock or switch the player's ship. There's definitely no sense in burning precious time trying to get to a certain phase in the game, only to discover that something small needs to be adjusted in the code, followed by replaying that time to verify the fix.

### 7.3.2 Different weapon types per ship

Another way to make the player ships feel different is to give each different attacks or features. Give each ship a unique primary attack as well as a secondary ability (another attack type, something movement related, or some other limited power or unusual property).

**Details:**

This exercise will likely involve some modification to both the LaserShot `class` as well as the shot code in the Player's tab. Ultimately what you choose to implement is up to you, though a few possibilities to consider include long-range shots, twin cannons, shockwave (short-range shot in many directions), guided missiles, mines (just stationary or short-range homing shot with a long lifetime), rail guns, nukes, brakes, random warp, or speed boost.

### 7.3.3 Two player support

Part of what works well about a game that can be controlled with only a few keyboard keys is that it becomes possible to support two players sharing the same display and input. At the title screen, give the player a prompt to start the game with one of two input actions, one of which will begin the game in two-player mode instead of single-player mode. The second player's ship should control like the player's, and be able to fire like the player's. Whether you decide to have the player's share lives and a common score, or each have their own lives and score, is your call. In either case, the game will need to be able to handle the situation in which either player loses first and is out of the action until a restart.

**Details:**

While it's possible for two players to share a keyboard, even for a simple game with relatively few controls it can be difficult to find an agreeable key mapping. WASD for one player and arrow keys for another are a natural choice for directions, and since there is no backward movement in this game the S key and down arrow could activate the secondary attack, leaving only one more button needed per player. Mapping one player on the number pad is tempting since it would afford more elbow space, though many laptop keyboards and compact wireless keyboards lack a number pad altogether. Beware, too, of using shift key for game input, which on some operating systems may pop up an accessibility message when used too rapidly.

The second player could be handled as another instance variable of the Player `class`, or alternatively players could be combined into an `ArrayList` as the enemies are. Whichever way it's handled it will be necessary to update the keyboard-handling code in the Player `class` to check different keys based on which player is currently getting the logic update.

If you are doing these exercises in order and in a common file, there will be added design and implementation challenges in figuring out how to integrate the custom ships with the two-player set up. The possibility of each player piloting a different type of ship sounds pretty exciting! Just because something will be a bit more involved to accomplish doesn't mean it should be passed up. Keep in mind that this isn't a graded assignment. This isn't on anyone else's timeline. The whole point is just to learn, and stretching yourself is a great way to do that.

## 7.4 Challenge Exercises

### 7.4.1 High scores initials

An earlier warm-up exercise included storing a list of scores, but we didn't at that time build in support for entering initials since this turns out to be a little more involved. Now is the time to take it on! When either player has a high score upon running out of lives, acknowledge the player's accomplishment on a new intermediary screen that allows input of initials before returning to the title screen.

**Details:**

While there are many things that Processing is well set up to do, for plain text entry it can be surprisingly tricky to work with in stark contrast to something like HTML forms or an old command-line application.

One tip I'll offer is that there is a better way to do this than to check each key separately in `keyPressed()`. The `key` value is automatically updated by Processing to be the character of the last key pressed, provided that the conditional `if(key != CODED)` is `true` then `key` is just the last typed character. `String`s in Processing can be appended by simply using a plus operator.

### 7.4.2 Power-ups dropped by enemies, vanish if not collected

Power-ups that get dropped by defeated enemies but vanish if not collected quickly enough are a classic game mechanic because they can compel the player to move through a thick pack of enemies mid-combat. Even a simple drop for some bonus points and a more rare drop earning an extra life will change how the game feels and plays, though there's plenty of potential to dream up other types of power-ups. When any enemy gets defeated, spawn a power-up in their place 10% of the time. Set up power-ups to vanish if not collected by the player within some narrow (but fair) interval of time.

**Details:**

Even though conceptually power-ups are nothing like laser shots, functionally in code they have a lot in common. They also need to get spawned on the fly, vanish after their time limit runs out, and collide with the player ship. They might even be set up to move slowly, so that they don't look too static. If you're comfortable with the `extends` keyword by following the pattern of how it's used in this example the power-ups could potentially be derived from the `LaserShots`. At the very least a similar pattern will be used so it's worth taking a deeper look into how the shots work. They're basically enemy shots that do something other than reset the player upon collision.

Icon images will be needed for the power-ups, though when initially developing them different `fill()` values set up before `rect()` calls work well as placeholder. Because spaceship games like this one are notoriously tricky to control precisely, giving power-ups a large collection radius can help reduce player frustration. In this context it's easily justified as the ship having some sort of tractor beam. Alternatively just give the power-ups big icons to fill the relevant collision space.

### 7.4.3 Enemy UFOs accounting for screen wrap

The UFOs aim toward the player by finding the angle between their ship and the player's current location. That works well in most situations, however it leaves them especially vulnerable to a player firing shots from around the screen edge. Change the UFO code to account for screen wrap, either identifying when to aim toward a nearby screen edge to counterattack, or deciding to wander away from an edge if the player is near the opposite side.

**Details:**

The difficulty of this has more to do with game design than it does with technical implementation. What will seem fair to a player? UFOs shooting back across the edge line can make them look intelligent if the player is paying close attention to them, though if that action isn't noticed it might seem unfair to suddenly be shot from an enemy across the screen. Having enemies avoid the edge opposite of the player could seem like a lame solution to the player coming up with a strategic plan of attack. An easier but more heavy-handed solution would be to just prevent shots from wrapping, so that the player can't attack from around the screen edge, either. If the player likes doing it, and its utility is limited enough to keep it from being all the player does, is it perhaps worth leaving in as is?

What do you think?

### 7.4.4 Particle effects explosions

Ships vanish when destroyed. Instead there should probably be some debris, maybe even flashy effects to help signal to the player that a hit occurred. Implement some minimal particle effects to display when ships get destroyed.

**Details:**

Particle effects sometimes involve special optimizations, however for the number, size, and complexity of particles in a game like this, a straightforward implementation will work fine. The particles can basically be thought of (and implemented) as just laser shots or power-ups that don't do collision checks against ships of either team. Alternatively, if you'd like more of a foundation or example to start from:

In the `commonBonus` folder the final versions of the `plane` demo (`plane_final`) includes a particle-effects implementation that could be borrowed and adapted for this exercise. Both the `Particle` and the `ParticleSet` files/classes will be helpful. The former is the data structure for each particle, the latter is a managing container for the whole set. Most of the work in adapting that solution to this project will be in simplifying it down to just the parts needed, as the particles from the `plane` demo support flames, destroyable terrain, colored clumps of land, and some other part features specific to that project. All that's needed here is smoke and some form of explosive burst. When copying the .pde scripts for particles from the `plane` demo be sure to also check for which image files will be needed out of the data folder, and copy it into the data folder for this project as well.

If you've already worked on the Bomb Dropper example, you may recognize that previous paragraph from that game's exercises. Speaking of which: if you already did that exercise, borrowing from your own code for this project would be a great way to start!

### 7.4.5 Computer-controlled ships with player-like weapons

Having implemented different weapon types for player ships, it's time for the computer opponents to keep up with the arms race. Pick one or two of the weapons or abilities that you've implemented for the player(s) and design a special enemy ship type or two to utilize those attacks. Depending on how much more dangerous this enemy type is than the default, either have it randomly show up in matches as one of the normal enemies, or set up one in every four levels as a boss fight against one or more of these special enemies.

**Details:**

As long as you've already figured out the movement code for the player's shot, this exercise shouldn't be particularly tricky to figure out, though it can add a lot of value to the game. A game that has only one type of opponent typically feels unfinished, whereas having even another type or two makes a big difference since you can present them in different combinations.

For an even greater challenge to you as both a developer and as a player, trying to get a computer-controlled version of the player ship, complete with thrust-based movement and turning rather than the UFO-like shuffling around, is a more complicated bit of programming. This can lend more life and interest to boss encounters, and even creates the possibility of having computer-controlled allies show up during play to provide reinforcements.

### 7.4.6 Ship and upgrade economy

If you've done the challenges up to this point you've designed multiple ship types, multiple weapon types, and a system for power-up drops when enemies are defeated. This step is about taking it just a bit further, implementing a concept of in-game money which can be picked up as power-ups, then spent between levels on new ship types, weapon upgrades, or extra lives.

**Details:**
An in-game store, if presented well, is a great way to encourage replay, support another level of strategy, and add depth to an otherwise simple game. Since shop interfaces inevitably have at least a handful of buttons or options that can be clicked, it will likely pay off to plan through the layout of that design in a mock-up form first, whether on paper, in a presentation slide tool, or just as an image. Rather than connecting ship types to particular weapons, disconnecting the two can present a more interesting and granular trade-off for the player, making them decide when and whether it's worth saving up for a different ship rather than new weapons.

## 7.5 Write From Scratch Steps: Space Battle

1. Create a new, empty Processing Sketch.

2. Set window size to 800x600

3. Begin by adding four empty functions that Processing will automatically connect to: `setup()`, `draw()`, `keyPressed()`, and `keyReleased()`.

4. Save the Processing project so that a folder will be created for it. Create a new folder in the project's directory. Name that folder data.

5. Using an image editing program, create a 32x26 image of the player ship, overhead view pointing right in the image, and another image for the player UFO. Save these images as PNG format in the data folder.

6. Load both image files as `PImage`. Display the player image in the middle of the screen.

7. Add `boolean` variables for tracking whether the left or right arrows are being held down. Set the corresponding value to `true` when the arrow key's `keyCode` gets detected in `keyPressed()`, set `false` when that `keyCode` is detected in `keyReleased()`.

8. Use a `float` to track the rotation value of the player's ship. In `draw()` decrease the angle by 0.05 when the left arrow is held, or increase it by 0.05 when

the right arrow is held. Constantly display the player's ship rotated around its center point by the `float` amount.

9. Fill the whole draw area with blue at the start of each `draw()` call, using `background(0,0,255)`, to erase the previous frame's update, so that graphics will appear to rotate cleanly without leaving a smudge.

*At this point the code reflects the state of progress demonstrated in the spaceBattle0 version of the source. Feel free to compare or check your work up to this point against that example.*

10. Add another `boolean` variable, this time for tracking whether the up arrow is being held down.

11. Create two new `float` values for keeping and using the player ship's current velocity, in data as an x speed (positive right, negative left) and y speed (positive down, negative up).

12. Add the player's velocity values to the player's position values every frame.

13. While the up arrow is held down, increase the player's velocity `float` values to move the player's ship in the direction it is currently pointed in. Multiply the `cos()` and `sin()` of the player's angle by a tuning constant, `THRUST_FORCE`, to be declared at the top of the file as `final float` and set to 0.1 as the value where it is created.

14. Make another tuning constant, `final float THRUST_DECAY_MULT`, set as 0.99, to be multiplied against the player's velocity values then saved over them every frame immediately after the velocity values are added to the player ship's position.

15. If the player's position goes off any edge of the screen, adjust the player ship's coordinate so that it appears to wrap immediately to the opposite side of the screen.

16. Create five new variables to track the state of the player's shot: two `floats` for its position on either axis, two `floats` for its velocity along either axis, and one `int` for how many cycles it will exist before it expires.

17. In the `draw()` function, if the player's shot has a positive number of frames left to exist, add its velocity float values to its position float values, and wrap its position if it crosses any screen edge, in addition to subtracting one from how many frames it has left to exist.

18. Also in the `draw()` function, when the player's shot has positive frames left to exist, draw a 3 pixel by 3 pixel white square centered at the shot's position.

19. When the player presses spacebar, if the player's shot has 0 or fewer frames remaining to exist, update its position to the position of the player's ship, update its velocity components to the velocity of the player's ship plus a substantial boost in the direction the player's ship is facing (tuned as a `final float SHOT_SPEED`, set to 4.0), and set its number of frames remaining to exist to a `final int` tuning variable named `SHOT_LIFE_CYCLES` (set to 70).

20. Using an image-editing program create a simple UFO ship that is around 29x23 pixels in its dimensions. Save it as a PNG with transparency in the data folder for this project.

21. Create four `float` variables to manage the enemy UFO's position and goal destination. In `setup()` initialize the UFO's start position to one quarter down the screen's height and one quarter across the screen's width. On the line immediately following, set its initial goal position to its position, so that will begin at the location it is first trying to reach.

22. Define a `final float ENEMY_MOVE_SPEED` tuning variable and set it to 0.8. In the `draw()` function check whether the enemy is more than one measure of `ENEMY_MOVE_SPEED`'s left of its horizontal goal position, and if so move it right by `ENEMY_MOVE_SPEED`. Also check the other direction, whether the enemy is more than one measure of `ENEMY_MOVE_SPEED`'s right of its horizontal goal position, and if so move it left by `ENEMY_MOVE_SPEED`. If neither of those conditions are `true`, set the horizontal goal destination to a new `random()` value up to the pixel `width` of the screen. Apply the same movements and logic to the vertical axis as well, for the y coordinate and y goal position, and limiting its new `random()` value up to the pixel `height` of the screen.

23. In the `draw()` function display the UFO image centered at the enemy's positoin values. The UFO image should never rotate.

24. If the player's shot gets within a certain range of the enemy's ship, that range being defined as a new tuning number at the top of the source file (`final float TARGET_RADIUS`, set to 12.0), reset the UFO's position to where and how it starts, in addition to immediately setting to 0 the number of frames remaining that the player's shot has to exist.

25. Create five new variables to track the state of the enemy's shot: two `floats` for its position on either axis, two `floats` for its velocity along either axis, and one `int` for how many cycles it will exist before it expires. Program it to operate the same as the player's shot: if it has positive cycles left in its lifetime counter increment its position by its velocity on each axis, wrap it in the event of crossing any screen edge, decrease its remaining life cycles by one, and if it gets too close to the player's ship, reset it to its start position along with setting to 0 its remaining

life cycles at that time. Draw a 3x3 white square centered at the enemy shot's location if it has a positive number of frames left to remain in action.

26. Instead of being fired upon a spacebar press in the direction of its ship, the enemy shot should be fired on a given frame if it has 0 or fewer shots left in action and also at that moment a random number up to 150 is less than 3. At that time the enemy shot's position should be set to the enemy ship's position, the enemy shot's should be set to a vector pointing from the enemy ship directly at the player ship albeit offset by a random value ranging from -2.0 to 2.0 on each axis independently, and the enemy shot's frames left in action should be bumped instantly up to the tuning value `SHOT_LIFE_CYCLES`.

27. To avoid the event of a tie, on the occasion that either the player's shot comes within the hit radius of the enemy's ship or the enemy's shot comes within the hit radius of the player's ship, the number of frames left in action for both shots should immediately be set to 0.

28. When the left or right arrow keys are held, instead of decreasing or increasing the player's ship angle by 0.05 instead decrease or increase by a new `final float` tuning variable, `PLAYER_TURN_RATE`, set to 0.025.

29. Fill the screen with solid black instead of blue to erase the previous frame's draw residue.

30. Go back and add 1 line comments as section heads above each chunk of code added, briefly labeling what it does or what it does it for.

*At this point the code reflects the state of progress demonstrated in the spaceBattle1 stage of the source. Feel free to compare or check your work up to this point against that example.*

*The bulk of the changes for the next several code versions follow a conceptual pattern across the entire project, refactoring the code but without any observable changes to gameplay behavior. For additional information about the rationale behind the updates in each phase, consult the first section in the Space Battle documentation,* ***About the Seven Code Steps****.*

31. Split the code into helper functions. The functions generally don't even need to have a return value or accept any arguments, just use it as a way of labeling and separating coherent chunks of code. The function names will likely reflect the header comments put in as the last step, combining information about what that small part of the code codes (move, or draw, for example) and what it does it for (enemyShot, or player, for example).

*At this point the code reflects the state of progress demonstrated in the spaceBattle2 version. Feel free to compare or check your work up to this point against that example.*

32. Create 6 new tabs/files for the project: Player, PlayerShot, Tuning, UFO, UFOShot, and keyHandling. Reorganize the functions and variables into whichever of these tab labels best fits. Leave `setup()`, `draw()`, and potentially other general purpose functions (or functions that otherwise don't belong in the other tabs) within the projects main source file.

*At this point the code reflects the state of progress demonstrated in the spaceBattle3 step. Feel free to compare or check your work up to this point against that example.*

33. Get rid of the Tuning tab/file by migrating any of its tuning values into the most closely related file. If it's unclear where else to move a remaining tuning value, place it back into the main source file. Delete the Tuning tab.

34. Adapt all of the contents of the Player tab into a Player `class`. Declare a single Player type variable in the main source file to act as the player's instance.

35. Adapt all of the contents of the UFO tab into a UFO `class`. Declare a single UFO type variable in the main source file to act as the enemy's instance.

36. Create a LaserShot `class` that accounts for all functionality needed by both the player's shot and the enemy's shot. Give the LaserShot `class` a `boolean` value which serves as its indicator of which team fired it, as for use when determining which ship to check its collision from. Delete the PlayerShot and EnemyShot tabs, instead giving the Player and UFO each an instance of the LaserShot.

37. Change the enemy's inaccuracy to be based on a random adjustment in the angle, rather than a randomized adjustment to the shot's separate horizontal and vertical velocity components at the time of its creation. Make the maximum angular inaccuracy amount determined by a tuning value in the UFO class, named `ENEMY_SHOT_INACCURACY` and set to 0.5. Change the frequency of enemy shots to no longer be based on a `random()` comparison but to instead happen each time a reload cycles timer counts down to zero, at which time it gets set to a random number between `ENEMY_RELOAD_CYCLES_MIN` (a new `final int` tuning value, set to 50) and `ENEMY_RELOAD_CYCLES_MIN` (a new `final int` tuning value, set to 150).

*At this point the code reflects the state of progress demonstrated in the spaceBattle4 phase. Feel free to compare or check your work up to this point against that example.*

38. Color the player ship image green, and the UFO ship image red.

39. Instead of having only one UFO instance in the main file, create an `ArrayList` to contain multiple instances of the UFO `class`. Spawn a number of them specified in the code by a new tuning value in the main source file, `HOW_MANY_UFOS`, set to 4. They should spawn evenly spaced across the width of the screen and 10% down the height of the screen from the top, returning to their own individual start location when hit by an enemy projectile. The player's projectile should only have the power to reset the enemy ship hit, not all at once.

40. Instead of having only one `LaserShot` per UFO and per Player, give each an `ArrayList` of arbitrarily many shots. Shots that have run out of active cycles should be removed from their containing `ArrayList`. As a reminder, that's generally best do during a reversed for loop over the list (or outside of a loop) rather than within a forward loop where that could cause entry updates to get skipped.

41. Add two new `int` values, one to keep track of how many times the player has successfully hit a UFO, and the other to keep trck of how many times the UFO shots have successfully hit and reset the plaer. Display those numbers on the screen along the top.

*At this point the code reflects the state of progress demonstrated in the spaceBattle5 source. Feel free to compare or check your work up to this point against that example.*

42. Create a new tab, WrappingPosition, containing a new `class` of the same name which has only two `float` values, for `x` and y position, and a function that causes those position variables to wrap across screen edges. Use the `extends` keyword to build the LaserShot `class` atop the WrappingPosition class, removing any redunancy from the LaserShot `class` (ex. the LaserShot should no longer declare its own `x` or y float values, and should not have its own code wrap function but should instead use the one defined in WrappingPosition).

43. Create a new tab, ArmedSpaceCraft, with a `class` of the same name that also `extends` WrappingPosition. The ArmedSpaceCraft `class` should contain an `ArrayList` of LaserShots, which it should have functions to reset, update/move, and draw.

44. Use the `extends` keyword to build Player and UFO both atop the ArmedSpaceCraft `class`. Again, remove any redundancy if possible, such that neither the Player nor UFO classes declare or directly manage their own `ArrayList` of shots nor their own `x` or y position nor their own wrapping code, but instead rely upon the functions and variables of the classes that they extend.

45. Create a new `final int` tuning value in the Player class to serve as a limit on how many shows the player can have active on the screen at a time, and call it `PLAYER_SHOTS_AT_ONCE_LIMIT`, set to 3. Disallow the player to have more than that number of shots active at a time.

46. Build in a title screen step for the program, providing instructions to the player and indicating that spacebar needs to be pressed to start. Program a simple decorative animation for the title screen.

47. Color the player's shots green and the UFO's shots red.

48. Give the player a limit to how many times he or she can respawn before being bumped back to the title screen. Set this as a new tuning `final int` variable called `PLAYER_START_LIVES`, set to 4. Display the number of lives remaining someplace easily seen during play.

49. Set up UFOs so that once hit by the player instead of respawning they stay eliminated until the player as defeated all UFOs that spawned at the start of the round. At that time the player should advance to the next round, in which the number of UFO's spawned at the start of that round should be increased by one each time. Their positions should be evenly spaced horizontally, and 10% the height of the screen from the top. Display in an easily seen place on the screen how many levels the player has survived since the title screen. Each time play resumes from the title screen the player should be back to facing only three ships in the first encounter.

50. Remove the score for the UFO team. For the player's score now award 100 points for each UFO defeated, and an additional 500 points times the level number for each level cleared (+500 points for clearing all three enemies in level 1, +1000 points for clearing all four enemies in level 2, +1500 points for clearing all five enemies in level 3, and onward, on top of the 100 points per ship).

51. Show the player's highest score achieved, and save that number to a text file to be retrieved when the program is opened back up. Give the player a way to clear that saved score, and display a message about how to do it on the title screen. Display the high score, when one is saved and was not recently cleared, both on the title screen and during gameplay.

52. While the player holds down the thrust button draw a yellow thrust triangle behind the ship of randomly varying length, to communicate that thrust is being applied while the key is held.

53. Create an `ArrayList` that contains a number of `PVector() x, y` coordinate positions, that exact number set by a `final int` tuning value named `STAR_COUNT` set to 200. When populating the `ArrayList` give each position a random spot

visible on the screen - anywhere horiziontally and anywhere vertically. Every frame immediately after wiping the screen with black and before drawing anything else iterate through all of these coordinates and draw a pixel at that position of a constantly changing random color per pixel, to simulate colorfully twinkling stars.

54. Each time the player gets hit by an enemy shot and needs to respawn, assign it a random horizontal position to reduce the liklihood of it being defeated multiple times in quick succession by a single UFO's attacks.

54. After the player completes a level or loses the last life, let the then one-sided remaining spaceship(s) continue to move for another few seconds in peace while a message informing the player of the event (level complete, and for how much bonus, or game over, and at what final score) before advancing to the next stage or resetting back to the title screen, respectively.

55. If the player's ship and an enemy UFO get within `TARGET_RADIUS*2.0` then destroy the UFO and the player ship, treating both for all purposes as though they had been hit by an opposing shot (the player still gets 100 points for the takedown, but also immediately loses a life).

56. If you haven't already done the exercises by starting from the code that was provided: you're now ready to take them on beginning from the source that you put together from scratch. Congrats!