

HANDS-ON INTRO TO GAME PROGRAMMING

BY CHRIS DELEON



WITH CODE FOR 6 CLASSIC GAMES

*This book is written for beginning game developers of all ages.
I want others to have an easier time learning this than I did!*

*Thanks goes out to Maanie Hamzaee for his logistical support
and Laura Schluckebier for her patience, love, and editing.*

Special thanks to Harold Bowman-Trayford of Renegade Applications, LLC for his generous and detailed editing assistance. Check out his game development work at @RenegadeOwner on Twitter. Many of the fixes in Version 3's Section 1 text are thanks to his observations and feedback.

For helpful feedback on preview copies I'd like to thank Ichiro Lambe, Kohta Wajima, and David Eugene Gabrell. For their willingness to work with in-process drafts and offer input along the way thanks to Ben Owens, Adrian Sanders, June Rockwell, Paul Diaz, and Chinua White. Another Version 3 improvement is thanks to David "Daitallica" Parfitt's email.

I'd also like to thank my sixth grade teacher Diane Watson for granting me my first opportunity to teach others how to have fun by making things on computers. I'm still happily doing it.

INTRODUCTION

BEGIN BY MASTERING FUNDAMENTALS

How do all great chefs begin? They first follow some recipes.

How do pianists and singers begin? They learn to perform recognizable tunes, beginning with time-honored classics.

How do craftspeople begin? They make a series of generic projects to gain familiarity with their tools and techniques.

In all subjects we have to first learn how to make anything at all before we can to make anything original and awesome. First attempts should be small, unoriginal, and unimpressive. This puts them right at the edge of a newcomer's abilities, building a bridge of increasing complexity toward mastery.

The advice most often given to beginning game developers follows a pretty consistent pattern: Start small. Remake retro games. Do the simplest games possible. Just get practice.

When I went through this same sort of sequence starting out, I had to struggle to piece it together from different resources.

I'm here to help you get going, feature by feature. I'll give you clear step-by-step recipes for how to craft game mechanics inspired by time-honored classics. In the process you'll gain a ton of practical familiarity with common tools and techniques.

THIS BOOK IS NOT REALLY ABOUT THESE CLASSIC GAMES

The point is not so much to make these specific games. The point is to get you practical experience, equipping you with reusable concepts that then empower you to create your own more original videogames later. Classic games give context and shape to introductory gameplay coding and design skills. Unless you choose to customize them, the games made here really exist primarily just to situate specific learning exercises.

THIS BOOK IS NOT PRIMARILY ABOUT LEARNING HTML5 AND JAVASCRIPT

You'll be writing a ton of JavaScript for HTML5 that works, but the book is not really about that.

Depending upon your priorities as a game developer, you may decide later to switch to another programming language. I stick to the simplest and most straightforward HTML5 JavaScript possible, mostly avoiding its advanced features. I don't even involve any CSS or jQuery. I instead wrote the example code in a way that mirrors the structure and abilities of other programming languages which are also used for game programming (C/C++, C#, Obj-C, AS3, Java, etc.).

The one exception is that, unavoidably, in a few key parts of the programs there's a particularly "JavaScript way" to get input, load images, play sounds, etc. Those operations vary by programming language, although there's at least always something roughly analogous. Because there's no way out of dealing with JavaScript particulars there anyway, I use those spots as opportunities to show and discuss a JavaScript-specific capability or two so that you'll have those in your tool belt or be able to recognize them when reading other code.

If you choose to stick with HTML5 then this will still be a strong starting point for that too. You'll get results and momentum before diving into its language-specific tricks.

On a related note, I've also opted to avoid using any outside libraries for these examples. I focus on standard HTML5 JavaScript and canvas capabilities. Libraries, which bring together well-tested code for common operations, are great! However they each have their quirks and limitations, they reflect one of many ways to do things, and they hide from a beginner what's happening under the hood. By learning how canvas works by default, and making a tiny library of your own, you'll gain a sense for what libraries are doing and how.

THIS BOOK IS ABOUT MAKING VIDEOGAMES, AT HOME, BEGINNING TODAY

Keep a browser open for testing while working through this book, and have the example solutions near. This is not a book that you should simply read, any more than you'd simply read a cookbook without making any foods from it.

From the other end of that: reading the final source code end-to-end, top-to-bottom will not be very helpful. Games get developed in iterative layers. Source is given in stages here because that's how games get made. This is the real process.

Tinker with it. Retune it. Extend it. Do things! That's what this is all about. Comment out lines. Experiment with changing the numbers. Adapt exercises to suit your interests and skills. Apply what you learn from making one game to add features to another, even if it's not explicitly suggested in the book.

EACH OF THE SIX GAMES HAS A COMPLETION EXERCISES CHAPTER IN SECTION 2

Each half of this book serves a different purpose for the six types of games covered. Each game appears in each section.

The first half walks you through creating, from blank files, the starting point with core gameplay for each of the game types. This also introduces game programming fundamentals. I've included full example solutions for every step in that section.

The second half of the book is a series of challenges to bring each of the games toward completion. The training wheels are off at that point. These don't include example solutions, although some example assets are still provided in case you want to focus purely on the code and design aspects. It's up to you to figure them out based on your learning from the first half and, when necessary, searching and experimentation. That's critical, because that's how most game programming happens – without exact examples to turn to when stuck.

AND SO BEGINS YOUR GREAT JOURNEY TOWARD CREATING YOUR OWN WORLDS

It may not be obvious at first just how related these simple classic games are to modern gameplay experiences. The deeper you get into it the more clearly you'll see that aside from beautiful 3D graphics and high quality sounds the very same types of motion, collision, logic flow, input handling, basic artificial intelligence, and iterative design processes are at the heart of virtually every computer game. You're on your way to writing your imagination into a machine to share it with others. I look forward to playing what you'll make.

Chris DeLeon
December 2014

SKILLS BY GAME AND SECTION

LEARNING GOALS AT A GLANCE

	Section 1 Core Gameplay Recipe	Section 2 Completion Exercises
<u>Game Type 1</u> Tennis Game (Tennis Mania)	Graphics Update Loop Mouse Input Linear Motion Ball Bouncing Collisions (Simple)	Difficulty Tuning Custom Text Display AI Bounce Prediction Motion Trail Title Screen
<u>Game Type 2</u> Brick Breaker (Brick Breakanoid)	Tile Grid, Modifiable Collisions (Simple II)	Lives Score Thick-Ball Grid Collision
<u>Game Type 3</u> Racing (Turbo Racing)	Tile Grid, Decorative Keyboard Image Graphics Local Multiplayer Angular Motion Friction Collisions (Intermediate)	Collisions (Advanced) Level Design Lap Verification Real-Time Measurement Simulated 3D Jumps Waypoint-Based AI
<u>Game Type 4</u> Warrior Legend (Warrior Adventures)	Tile Grid, Lock & Key Tile Transparency Collectibles Character Controller	Animation Ammo Limits Directional Melee Advanced Gating Grid-Aware AI
<u>Game Type 5</u> Space Battle (Space Battle Deluxe)	Projectiles Wandering AI Momentum Edge Wrap Collisions (Object)	Spectacle Wrap-Aware AI Item Drops Difficulty Waves Saved High Score
<u>Game Type 6</u> Real-Time Strategy Game (RTS Robot Commander)	Group Control Mouse Selection Distance-Based AI Dynamic Formations	Pathfinding Area of Effect Resource Management Team-Based AI

TABLE OF CONTENTS

OVERVIEW

Thanks.....	2
Introduction.....	3
Skills by Game and Section.....	7
Table of Contents - Overview.....	8
Table of Contents - Detailed.....	9
Getting Started.....	17
Section 1: Classic Game Programming Recipes	
Tennis Game (Early 1970's Arcade-Style).....	25
Brick Breaker (Late 1970's Arcade-Style).....	88
Racing (Early 1980's Console-Style).....	123
Warrior Legend (Mid-1980's Tiled Dungeon).....	209
Space Battle (Projectiles, Wrap, and Enemies).....	246
Real-Time Strategy (Early 1990's Desktop-Style).....	279
Intermission: Sound Exercises	
Game Sound.....	365
Section 2: Completion Exercises	
Tennis Mania.....	391
Brick Breakanoid.....	415
Turbo Racing.....	440
Warrior Adventures.....	466
Space Battle Deluxe.....	487
RTS Robot Commander.....	513
Epilogue.....	551
Section Bonus - Mini-Extras.....	552
Version Change History.....	553
Image Credits.....	558

TABLE OF CONTENTS

DETAILED

Thanks.....	2
Introduction.....	3
Skills by Game and Section.....	7
Table of Contents - Overview.....	8
Table of Contents - Detailed.....	9
Getting Started.....	17
Section 1: Classic Game Programming Recipes	
Tennis Game (Early 1970's Arcade-Style).....	25
Included Bonus: A Video Course Version of Tennis Game, Section 1....	26
Step 1 – Create Your Initial HTML File.....	27
Step 2 – Fill the Background with Black.....	39
Step 3 – Draw the White Ball.....	45
Step 4 – Change Ball Position Automatically.....	49
Step 5 – Get the Ball Moving Smoothly.....	53
Step 6 – Variable for the Ball's Speed.....	58
Step 7 – Bounce Ball Off of Right Edge.....	59
Step 8 – Bounce Ball Off of Left Edge.....	61
Step 9 – Give the Ball Vertical Movement.....	63
Step 10 – Draw the Player (Left) Paddle.....	64
Step 11 – Create Rect and Circle Functions.....	65
Step 12 – Use Mouse to Position the Paddle.....	66
Step 13 – Reset the Ball If the Player Misses.....	69
Step 14 – Draw the Right Player's Paddle.....	72
Step 15 – Player (Test) Control for Right Paddle.....	73
Step 16 – Let the Ball Go Off the Right Edge.....	74

Step 17 – Put Text On the Screen for Score.....	75
Step 18 – Tracking the Left Player's Score.....	76
Step 19 – Add Score for the Right Side Paddle.....	77
Step 20 – Direct Ball by the Paddle Height Hit.....	78
Step 21 – Computer Controlled Right Paddle.....	80
Step 22 – Resolve Shaking by the A.I. Paddle.....	81
Step 23 – Set Up a Maximum Score to Win.....	82
Step 24 – Pause the Game After Anyone Wins.....	83
Step 25 – Let the Player Know How to Reset.....	85
Step 26 – Dashed Line for Net to the Middle.....	86
Step 27 – Removing Update Markings.....	87
 Brick Breaker (Late 1970's Arcade-Style).....	88
Step 1 – Get a Ball Bouncing on Screen.....	89
Step 2 – Adding a Paddle to Block the Ball.....	90
Step 3 – Ball Control from the Paddle Spot Hit.....	91
Step 4 – Space Below the Player Paddle.....	92
Step 5 – Fix the Ball Hitting the Paddle Side.....	93
Step 6 – Draw the Bricks with Nested Loops.....	94
Step 7 – Support for Removal of Bricks.....	96
Step 8 – Detect When the Ball Hits Bricks.....	101
Step 9 – Bounce the Ball When it Hits Bricks.....	110
Step 10 – Handle Collision to Brick Sides, Pt. 1.....	111
Step 11 – Handle Collision to Brick Sides, Pt. 2.....	115
Step 12 – Reset Bricks Once the Last is Gone.....	119
Step 13 – Remove 3 Brick Rows and Ball Cheat.....	120
Step 14 – Erase All Update Markings.....	122
 Racing (Early 1980's Console-Style).....	123
Step 1 – Copy in Brick Breaker to Start From.....	124
Step 2 – Removal of Player Paddle.....	125
Step 3 – Fill the Screen with Bricks & Big Grid.....	126
Step 4 – Support for Hand-Designed Tracks.....	128
Step 5 – Loading a Car Image for the Player.....	131
Step 6 – Spinning the Car Image.....	133
Step 7 – Putting Image Rotation in a Function.....	134

Step 8 – Using Key Press and Release.....	135
Step 9 – Cut How the Car Has Been Moving.....	137
Step 10 – Driving the Car in a Spiral.....	138
Step 11 – Speed and Turning with Arrow Keys.....	141
Step 12 – Support for Key Holding.....	144
Step 13 – Rolling to a Stop, & No Turn While Still.....	147
Step 14 – Set Car Start Position in Track Data.....	150
Step 15 – Walls that Block Car Movement.....	155
Step 16 – Separate JavaScript From the HTML.....	158
Step 17 – Split the Code Into Multiple Files.....	159
Step 18 – Load and Show Track Tile Graphics.....	163
Step 19 – Reduce Risk for Bugs in Loading Art.....	168
Step 20 – Organize Files Into Folders by Type.....	173
Step 21 – Add Three Kinds of Decorative Tiles.....	174
Step 22 – Organize Track Art Into an Array.....	178
Step 23 – Reduce Calculations For Track Draw.....	181
Step 24 – Create a Class for Car Code Reuse.....	185
Step 25 – Preparing Key Input for Player Two.....	191
Step 26 – Adding Player Two’s Car.....	195
Step 27 – Different Graphic for Player Two.....	197
Step 28 – Respond to Crossing the Finish Line.....	199
Step 29 – Declaring a Winner by Name.....	204
Step 30 – Reset Both Cars When Either Wins.....	205
Step 31 – Cleaning Out the Update Markings.....	208
 Warrior Legend (Mid-1980’s Tiled Dungeon).....	209
Step 1 – Copy in Racing Code to Start From.....	210
Step 2 – Removing Second Player’s Car.....	211
Step 3 – Renaming Files and Variables.....	212
Step 4 – Generalize Naming in Player Code.....	213
Step 5 – Walk Like a Warrior – Key Changes.....	215
Step 6 – Walk Like a Warrior – Slide to Move.....	218
Step 7 – Bigger and Fewer Tiles on Screen.....	221
Step 8 – Switching in Warrior & Dungeon Art.....	223
Step 9 – Renaming Room Variables in Code.....	225
Step 10 – Support for Transparency in Tile Art.....	230

Step 11 – Creating a Dungeon Layout.....	233
Step 12 – Using Keys to Unlock Doors.....	236
Step 13 – Cleaning Out the Update Markings.....	245
Space Battle (Projectiles, Wrap, and Enemies).....	246
Step 1 – Copy a Version of Racing to Start.....	247
Step 2 – Clearing Out All Track Elements.....	249
Step 3 – Replace Car Code and Art with Ship.....	250
Step 4 – Edge Wrap for Ship.....	251
Step 5 – Slippery Space Without Brakes.....	252
Step 6 – Separating Facing From Drift.....	253
Step 7 – Giving the Ship a (Stationary) Shot.....	254
Step 8 – Giving the Shot Direction.....	259
Step 9 – Blocking Fire Until the Shot Expires.....	262
Step 10 – Shared Code Between Ship & Shot.....	264
Step 11 – Adding an Enemy UFO.....	271
Step 12 – Crashing Into and Shooting the UFO.....	275
Step 13 – Cleaning Out the Update Markings.....	278
Real-Time Strategy (Early 1990's Desktop-Style).....	279
Step 1 – Creating A New Minimal Foundation.....	280
Step 2 – Add Stationary Player Character.....	282
Step 3 – Random Start Spot, & Click to Move.....	283
Step 4 – Walking Unit in Direct, Straight Line.....	285
Step 5 – Swarm of Units.....	294
Step 6 – Spread Destinations to See Group.....	299
Step 7 – Click and Drag Selection Lasso.....	302
Step 8 – Use Selection Box to Choose Units.....	307
Step 9 – Lasso Box Working in Any Direction.....	311
Step 10 – Command Selected Units.....	313
Step 11 – Rows Formation for Moved Group.....	318
Step 12 – Enemy Units on Map.....	327
Step 13 – Click on Enemy for Attack Command.....	330
Step 14 – Separate File for Input.....	336
Step 15 – Units That Attack When Ordered.....	340
Step 16 – Enemies That Wander Automatically.....	342

Step 17 – Enemies That Attack When Close.....	343
Step 18 – Remove Ghosts From Both Teams.....	352
Step 19 – Only Remove Units When Destroyed.....	357
Step 20 – Declare a Winning Team.....	360
Step 21 – Keep Units on the Battlefield.....	361
Step 22 – Removal of Update Marks from Code.....	363
Intermission: Sound Exercises	
Game Sound.....	365
Phase 1 – Make a Short Sound, Find a Song.....	366
Phase 2 – The Simplest (Unreliable) Approach.....	368
Phase 3 – Detecting Format Compatibility.....	369
Phase 4 – Play Sound on Click Event.....	371
Phase 5 – Switch to the Longer Sound File.....	373
Phase 6 – The Hiccup Workaround.....	374
Phase 7 – The Redundant Buffer Workaround.....	375
Phase 8 – Wrap the Extra Buffer in a Class.....	377
Phase 9 – Encapsulate the Sound Class.....	381
Phase 10 – Looping Music Track.....	385
Phase 11 – Getting the File Ready for Game Use.....	388
Section 2: Completion Exercises	
Tennis Mania.....	391
Exercise 1 – Triple the Height of Both Paddles.....	392
Exercise 2 – Tune Target Score and Balance It.....	393
Exercise 3 – Invisible Player.....	395
Exercise 4 – Easier to Return Serve.....	396
Exercise 5 – Add Sound Effects.....	397
Exercise 6 – Ball Speed Increase During Play.....	398
Exercise 7 – Refactor Into Separate JS Files.....	400
Exercise 8 – Add Image Graphics.....	401
Exercise 9 – Move Paddles Away From Sides.....	403
Exercise 10 – Two Player Mode With Keyboard.....	405
Exercise 11 – Title Menu.....	406
Exercise 12 – Programmatic Giant Score Numbers.....	407
Exercise 13 – Ball Trail Effect Using Array.....	408

Exercise 14 – Computer Player Sees Bounces.....	410
Exercise 15 – Compute Precise Paddle Hit Point.....	412
Exercise 16 – Unblockable Corners (Optional).....	413
Brick Breakanoid.....	415
Exercise 1 – Organize Code Into Separate Files.....	416
Exercise 2 – Keep and Display Score.....	417
Exercise 3 – Life Limit and Full Reset.....	418
Exercise 4 – Click to Serve the Ball.....	419
Exercise 5 – Fix Ball Stuck Along Boundary.....	420
Exercise 6 – Replace Coded Art with Images.....	422
Exercise 7 – Display Lives as Row of Icons.....	424
Exercise 8 – Title Screen.....	425
Exercise 9 – Ball Speed Increase.....	427
Exercise 10 – Extra Lives for Score Milestones.....	428
Exercise 11 – Add Audio.....	429
Exercise 12 – Bricks Layout From Array in Code.....	430
Exercise 13 – Different Kinds of Bricks.....	432
Exercise 14 – One Brick Collision Per Trip (Retro).....	433
Exercise 15 – Give the Ball Thickness for Collision.....	434
Exercise 16 – Temporary Power-Up Drop Downs.....	437
Exercise 17 – Make and Support Multiple Levels.....	439
Turbo Racing.....	440
Exercise 1 – New Tile Types: Grass and Oil Slick.....	441
Exercise 2 – Day/Night or Theme Tile Sets.....	442
Exercise 3 – Basic Car-to-Car Collision.....	443
Exercise 4 – Terrible Computer Controlled Car.....	444
Exercise 5 – Keep and Show True Race Time.....	445
Exercise 6 – Nitro Boost.....	446
Exercise 7 – Support Multiple Tracks.....	447
Exercise 8 – Ramp Tiles and Airborne Cars.....	449
Exercise 9 – Sound Effects (Advanced!).....	451
Exercise 10 – In-Game Track Editor GUI.....	452
Exercise 11 – Laps With Checkpoints.....	453
Exercise 12 – Collision at Front and Rear of Car.....	455

Exercise 13 – Larger Scrolling Track (One Player).....	459
Exercise 14 – Better Computer Driver.....	461
Exercise 15 – Weather Effects (Visual + Steering).....	464
Exercise 16 – Zoom Camera for Two Player Big Map.....	465
 Warrior Adventures.....	466
Exercise 1 – Four Facings for Player Sprite.....	467
Exercise 2 – Player Health and Harmful Obstacle.....	468
Exercise 3 – Flying Bat Enemy.....	469
Exercise 4 – Melee Attack in Faced Direction.....	470
Exercise 5 – Color Coded Keys and Doors.....	471
Exercise 6 – Recorded Sound Effects.....	472
Exercise 7 – Tile-Wandering Enemy.....	473
Exercise 8 – Player Ranged Attack with Ammo.....	474
Exercise 9 – Enemy Ammo Drops, Limit Max Ammo.....	475
Exercise 10 – Projectile Enemy with Facing.....	476
Exercise 11 – Adjacent Rooms.....	477
Exercise 12 – Items to Increase Total Hearts, Heal.....	478
Exercise 13 – In-game Tilemap Editor.....	479
Exercise 14 – From Key Colors to Gating Powers.....	481
Exercise 15 – Inventory Menu.....	482
Exercise 16 – Room Change Scrolling Effect.....	483
Exercise 17 – Up/Down Stairs With Fade.....	484
Exercise 18 – Boss Fight.....	485
Exercise 19 – Find Some Playtesters and Playtest.....	486
 Space Battle Deluxe.....	487
Exercise 1 – Multiple Shots Handled as an Array.....	488
Exercise 2 – Using Score to Incentivize Behavior.....	489
Exercise 3 – Custom Sci-Fi Art.....	491
Exercise 4 – Title Screen.....	493
Exercise 5 – Drifting Destroyable Space Rocks.....	494
Exercise 6 – Enemies That Can Shoot.....	495
Exercise 7 – Multiple Enemies in Array.....	497
Exercise 8 – Fragment Rocks.....	498
Exercise 9 – Player Weapon Types.....	501

Exercise 10 – Vanishing Weapon Item Drops.....	502
Exercise 11 – Difficulty Waves Progression.....	503
Exercise 12 – Local Multiplayer Co-Op/Vs Support.....	504
Exercise 13 – Generated Twinkling Star Field.....	505
Exercise 14 – Particle Effects Explosions.....	506
Exercise 15 – Enemies Aim Considering Wrap.....	508
Exercise 16 – Save and Clear Local High Score.....	509
Exercise 17 – Player Ship Selection.....	510
Exercise 18 – Bot Player for Co-Op and Vs Play.....	512
RTS Robot Commander.....	513
Exercise 1 – Clear Select on Right Click.....	514
Exercise 2 – Draw Targeting Line.....	515
Exercise 3 – Find and Edit Sounds.....	516
Exercise 4 – Unit and Background Graphics.....	517
Exercise 5 – Unit Health Bars.....	521
Exercise 6 – Change Health Color For Low or Max.....	522
Exercise 7 – Bounded AI Thinking Counter.....	523
Exercise 8 – Show Health Only on Some Units.....	524
Exercise 9 – Draw Lasers That Show Missing.....	526
Exercise 10 – Heavy Unit Type.....	527
Exercise 11 – Color Key Collision Affecting Mobility.....	529
Exercise 12 – Fog of War Grid or Per-Pixel.....	531
Exercise 13 – Scrolling Camera with Minimap.....	532
Exercise 14 – Projectile Attack Splash Damage.....	534
Exercise 15 – Unit Draw Order From Back to Front.....	536
Exercise 16 – Player-Like Computer Player.....	539
Exercise 17 – Strategic Computer Army Behaviors.....	541
Exercise 18 – Grid Obstacles and A* Path Finding.....	547
Exercise 19 – Resources, Build Menu, & Recyclers.....	549
Exercise 20 – Buildings and Tech Requirements.....	550
Epilogue.....	551
Section Bonus - Mini-Extras.....	552
Version Change History.....	553
Image Credits.....	558

GETTING STARTED

UNZIP / EXTRACT / DECOMPRESS THE FILES

Many of the files need to use other files. Images and scripts won't be accessible to other files if the folders are still zipped.

EXPOSE FILE EXTENSIONS

Expose file extensions (.html, .js, .png, etc.) if you haven't yet done so. Search the web for how to show file extensions on your operating system if you're unsure how to do this. It's never more than a few steps, and can be easily undone in case you share the computer with another person who doesn't wish to see file extensions. This information needs to be shown when programming because it reveals the full names of files as they're needed for references in code. This is also helpful to ensure that files with names like Main.html or Input.js actually have those extensions, versus really being named Main.html.txt or Input.js.txt with the ending hidden.

HOW TO TEST YOUR HTML FILES IN-BROWSER

Drag and drop your HTML file onto a browser like Internet Explorer, Chrome, or Firefox. Most modern browsers even include a JavaScript or developer debugging console in their menus that will be very useful if doing this. That console is where you'll see error messages and line number listings if something in your code isn't working right. If you ever drag

your HTML file onto the browser and see a white page or something's malfunctioning, check that debug console.

If testing your code this way, you can use any plain text editor for creating and editing your HTML and JS files. One like Sublime Text, Notepad++, or TextWrangler with programmer features like syntax highlighting (my next point), auto-indent, line numbers, and file tabs can be helpful as the code grows.

Sublime Text is a popular choice, and has some advantages. However Notepad++ for Windows and TextWrangler on Mac are both free. Sublime Text has a free trial if you'd like to try it, but it will randomly remind you to buy a license for it.

Alternatively, there are integrated development environment (IDE) options that are better suited for complex debugging tasks and managing a greater number of files for larger projects. Those won't be needed for the sort of introductory projects here. But improved text editors like Sublime Text, Notepad++, or TextWrangler will be easier than just Notepad.

ABOUT SYNTAX HIGHLIGHTING

Code in the book will often be various colors, like so:

```
function moveEverything() {
  if(ballX > canvas.width) { // if ball has moved beyond the right edge
    ballSpeedX *= -1; // reverse ball's horizontal direction
  }

  ballX += ballSpeedX; // move the ball based on its current horizontal speed
  console.log(ballSpeedX);
}
```

Your code (and the included example code) won't necessarily be those same colors, and don't need to be. The colors aren't part of the program code. Many text editors that are used for

editing code apply different colors automatically to common keywords, math operators, numbers, text strings, and so on. This is to make the code a bit easier to look at, so I've done the same for code in this book. But if you don't see colors in your text that's not a problem as far as the code working. If you want the colors, search for a text editor that supports syntax highlighting (ideally, for JavaScript specifically).

ON USING THE SECTION 1 EXAMPLE SOLUTIONS

If you're ever unsure about where the new code goes or how it should all look when put together, feel welcome to check out the corresponding example solutions that are included for the first half the book.

If you're stuck, hitting an error you're having trouble resolving, are very new to coding, or simply feel that you'd benefit more by moving forward than from spending more time puzzled, it's not cheating to take a peek at the answers. I really do intend for people to refer to them. That's why they're there.

As mentioned in the introduction, for Section 2's challenge exercises to bring the projects forward you're going to be more on your own. Besides being a way to test your learning and programming comfort without shortcuts to turn to, since Section 2 doesn't include solutions those exercises can be assigned in classroom, workshop, and homeschool settings without need for concern over anyone buying an answer key.

In each example step for Section 1 I've made it easier to find what's been added or changed by putting a special four slash /// comment right of every line edited as part of that step. Since it begins with the standard // comment mark it

won't affect what the code does. Find-in-file in any text editor can be used to cycle through any step's latest changes.

In the interest of your long-term programming ability, I strongly encourage *typing into your own file* whatever is different from your present solution and the answer. I know it's tempting to copy and paste lines or chunks from the examples, but that's terrible for learning. Doing that is a fast and dangerous way to wind up lost due to missing important details. Retyping brings your focus to specific changes in the code that you'll want to recognize and be able to recall and understand for Section 2, and even more so for when you're totally on your own making original games in the future.

COMPLETELY NEW TO PROGRAMMING?

If you've already done at least some programming before, and are here to make the jump from non-game programming into game programming, jump right in!

If, on the other hand, you're totally new to programming of any kind, I still suggest wading in to see whether you'll be comfortable learning from context and the provided explanations. If you're finding that too steep though or feeling lost, as another option you may wish to switch over to the original Processing/Java version of this package for a smoother introduction designed specifically for people with zero programming experience. That version's still included in your download at no added charge. Its main differences:

- (1) The older version uses Processing, based on Java, instead of HTML5/JavaScript. Processing was made specifically to simplify learning interactive graphical

programming of the kind done for games. Processing is similar enough to JavaScript that when you return to this newer version of the book most of your Processing learning will carry over.

- (2) That older version focuses first and heavily on the practice exercises, with the “write from scratch steps” for authoring the starter code coming after. That order of doing things may be easier to wade into – it immediately gives context.
- (3) The older version includes links to a series of videos I recorded giving code demonstrations that I created for teaching first-time programmers how to program real-time interactive applications. Those videos can help provide a better handle on the jargon and early programming issues, helping you get the most out of the game coding material.
- (4) That version had a game called Bomb Dropper instead of this package’s Warrior Legend.
- (5) It’s considerably shorter.
- (6) Space Battle was handled in a different way than the other five games, demonstrated in a series of refactoring steps.

If you do take a detour through the older package to get up to speed on core programming concepts, you’re welcome to hop back over to this updated HTML5/JavaScript edition at any time when you’re comfortable doing so. The languages and programming techniques are closely related. You may even find it a healthy exercise getting exposure to some of the ways that programming languages can vary.

DISCLAIMER ABOUT DELAYED COMPLEXITY IN PROGRAMMING TECHNIQUE

My training philosophy involves keeping things as simple and straightforward as possible, adding complexity only when it helps us get results. For example, until a project is large enough to need multiple files, I'll keep the whole game in one.

The code is written in a style that prioritizes readability and understandability for someone new to programming games. Program architecture and optimizations can be learned later as another layer after you've mastered enough coding to get various types of games playable in a straightforward way.

I'm also not covering object-oriented programming (OOP) or software design patterns here. Those are higher order code concepts with their own distinct vocabulary and puzzles. I believe these are valuable. Too often though they are taught too early to be of practical use to new programmers. These are brilliant solutions to challenging abstract problems which arise in medium and large projects, but on smaller games they tend vastly overcomplicate what can be a simple task.

I do, however, try to ease in introductions to these ways of thinking about programming. I do so only when it benefits the specific problem at hand.

The amazing books and websites that cover OOP and design patterns will be most valuable, I believe, to someone who has first run up against the very challenges and scale limitations that people invented those approaches to solve.

ONE LAST THING – ERRORS ARE A NORMAL PART OF PROGRESS

Don't get discouraged when you encounter errors along the way. I'll repeat this later. It's ridiculously important.

I work with many beginning game developers. All too often I meet someone who feels like they're a bad programmer since they frequently encounter errors when trying to run their latest code. However that's a perfectly normal part of programming. That doesn't just apply to beginners. I've been doing this for many years and I still get errors in my code. Then I fix them.

When you're walking to get someplace outdoors and there's suddenly a tree in the way, how do you respond? Do you feel bad, like you planned your walking wrong? Do you get angry about it? Do you turn around and give up? No, of course not! You simply walk around it. Well, that's exactly how working through program errors has to be. Don't even break stride.

Figuring out how to diagnose and repair the error feedback (or unintended results on screen) is important practice for a game developer. A little frustration along the way is a positive sign that you're pushing yourself to learn. It means you're working through challenges that you didn't yet know how to do, so that you can come away from it knowing how to do it. You'll even have playable evidence that shows you've done it.

Welcome to videogame development!

SECTION 1

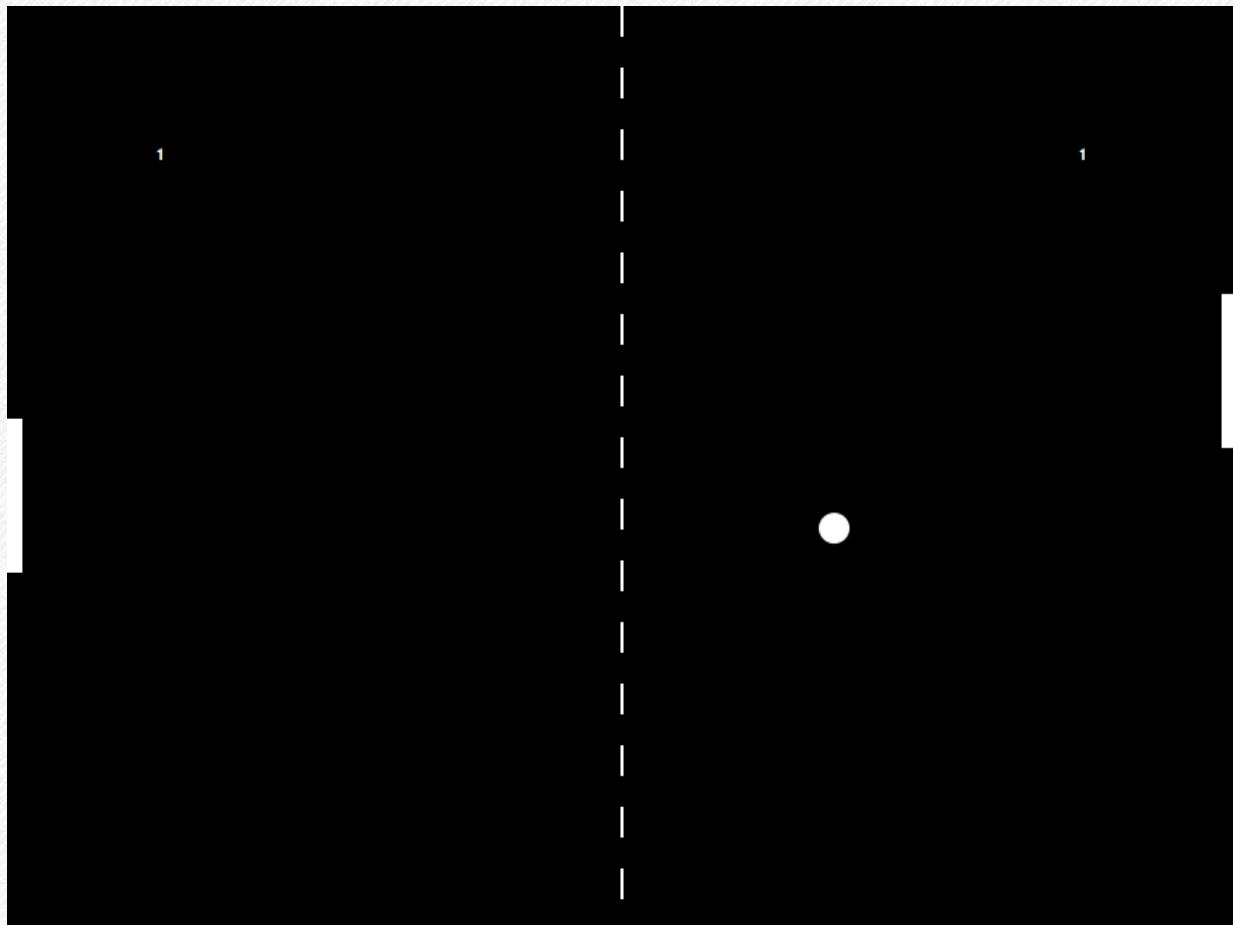
CLASSIC GAME PROGRAMMING RECIPES

HOW TO WRITE SIMPLE PLAYABLE CODE

GAME 1: TENNIS GAME

EARLY 1970'S ARCADE-STYLE

Contrary to popular belief, computerized games similar to this style were not the first to appear in public for mainstream play. Space Battle, covered later, actually bears more similarity to the first commercial arcade game. Yet it was the simplicity of this play experience that captured the world's imagination, rocketing the game industry toward the scale we see today. That same powerful simplicity makes it an ideal first project.



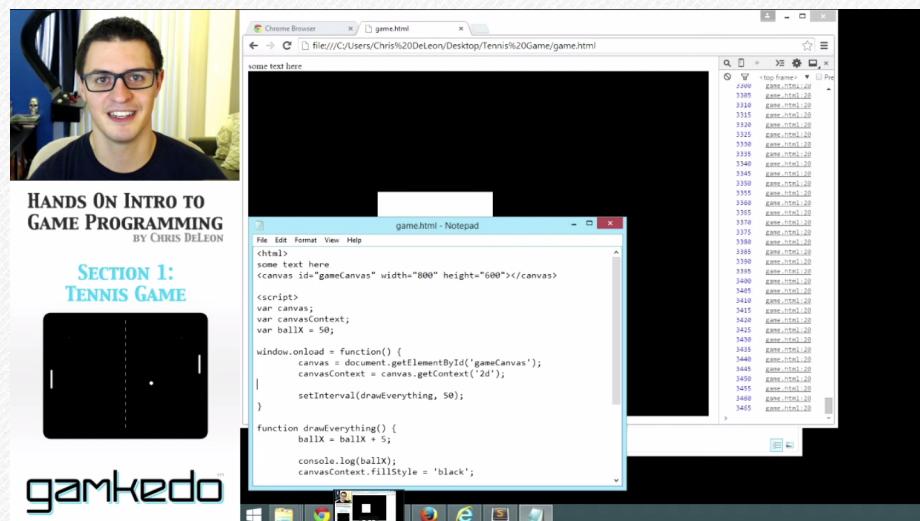
Included bonus: a video course version of tennis game, section 1

I've prepared a video course version of this chapter. This is the very same way that I cover the material when I use this book as my textbook for training new game developers. You can watch the video files (an internet connection is required) and find the resulting code for each video step in this folder:

Section 1 Tennis - Video Version\

The material is adapted for video, so the code and steps are slightly different than those in this book's text. They wind up nearly the same though, and address the same topics. (The main difference is only in a little extra html from Step 1 just ahead, which keeps Firefox from giving a harmless warning.)

No matter whether you work from the book's steps or follow this video for Tennis Game you'll be equally well prepared.



A video version covering Brick Breaker, Racing, and parts of Warrior Legend is available separately or as part of a bundle.

tennis game step 1

create your initial html file

Create a new text file, rename the file extension from .txt to .html, and give the file some baseline HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta content="text/html;charset=utf-8" http-equiv="Content-Type">
<meta content="utf-8" http-equiv="encoding">
</head>
<body>

</body>
</html>
```

In case you're completely new to HTML and this text looks alien, it's really nothing to worry about. Though this approach to game development is often referred to as HTML5 for short, you won't be fiddling with much HTML directly. Most of the game code will be in JavaScript, which runs as part of HTML.

The various elements above – doctype, utf-8 specification, lang setting, head and html tags – won't change during development. You'll see them stay the same through every step. No need to memorize that stuff, either, since it will be the same for all HTML5 files. It can be looked up and copied in when starting a new game project. That's still what I do.

To make room for JavaScript, add a `<script>` and `</script>` section where the middle line was left blank. Anything between the `<script>` tags will be JavaScript instead of HTML. HTML is mainly suited for motionless web page content and formatting, but with JavaScript in the `<script>` tags you can program

videogame functionality through handling graphics, input, and logic even after the page has finished loading.

Graphics and motion code will all be in the <script> area. First though, write a console.log() call inside window.onload, like so:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta content="text/html;charset=utf-8" http-equiv="Content-Type">
<meta content="utf-8" http-equiv="encoding">
</head>
<body>

<script>
window.onload = function() {
    // window.onload gets run automatically when the page finishes loading
    console.log("Hello World!");
}
</script>

</body>
</html>
```

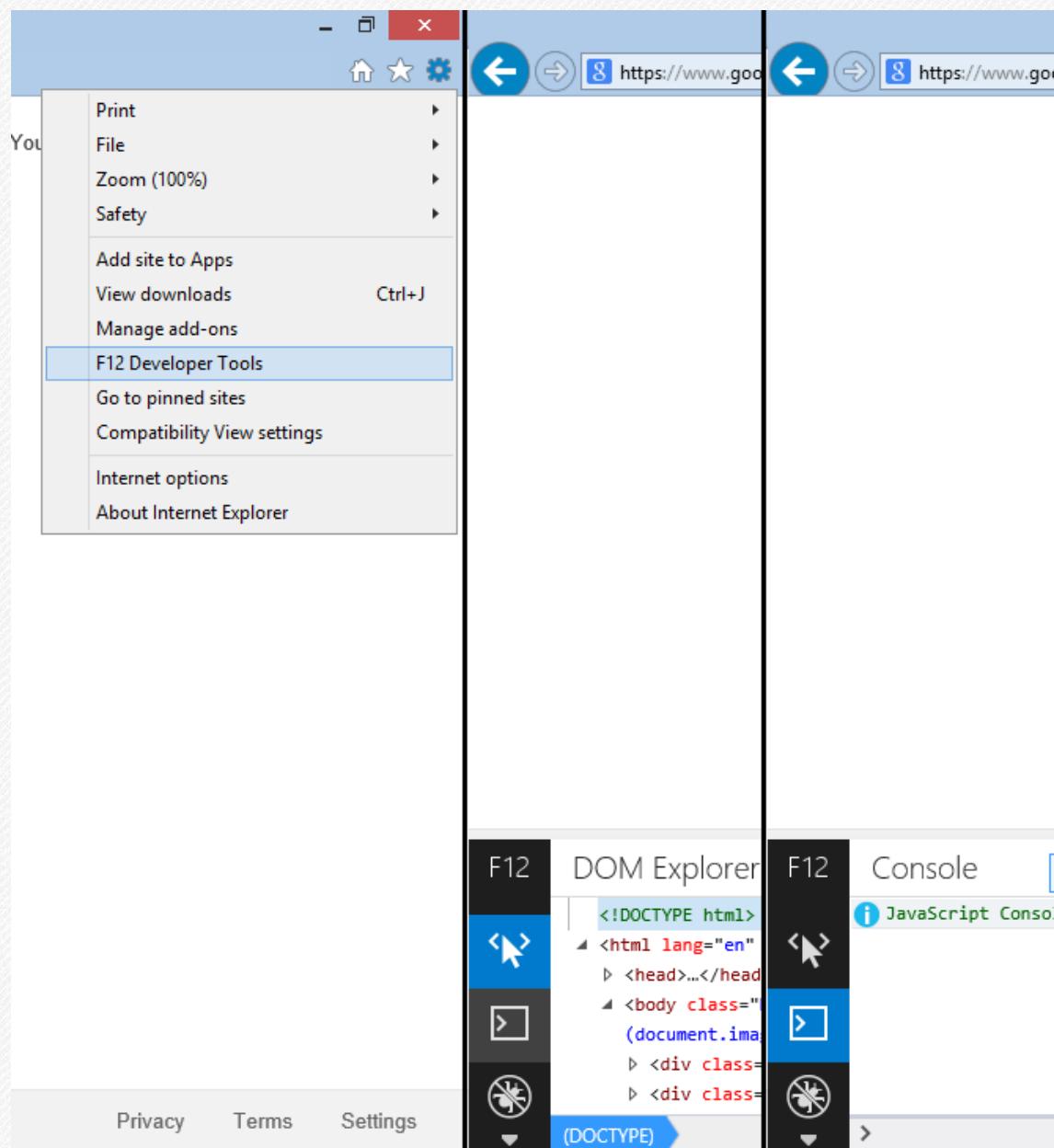
The window.onload function is special in that, as indicated by the comment (text right of // ignored by the program), that block is run once when the page finishes loading. It's possible to use JavaScript without window.onload, however by waiting until all code is loaded before your program starts you gain a lot better flexibility in how you order and organize your code.

To view the text output by console.log() you'll need to open the web browser's console log. This feature goes by many names: web console, developer mode, JavaScript console, etc. In addition to showing console.log() output, this is also where you'll see line numbers from the computer for errors in the code. It's very important to know how to open your browser's console for troubleshooting when the code doesn't work right.

Here's an illustrated guide to how to open your console:

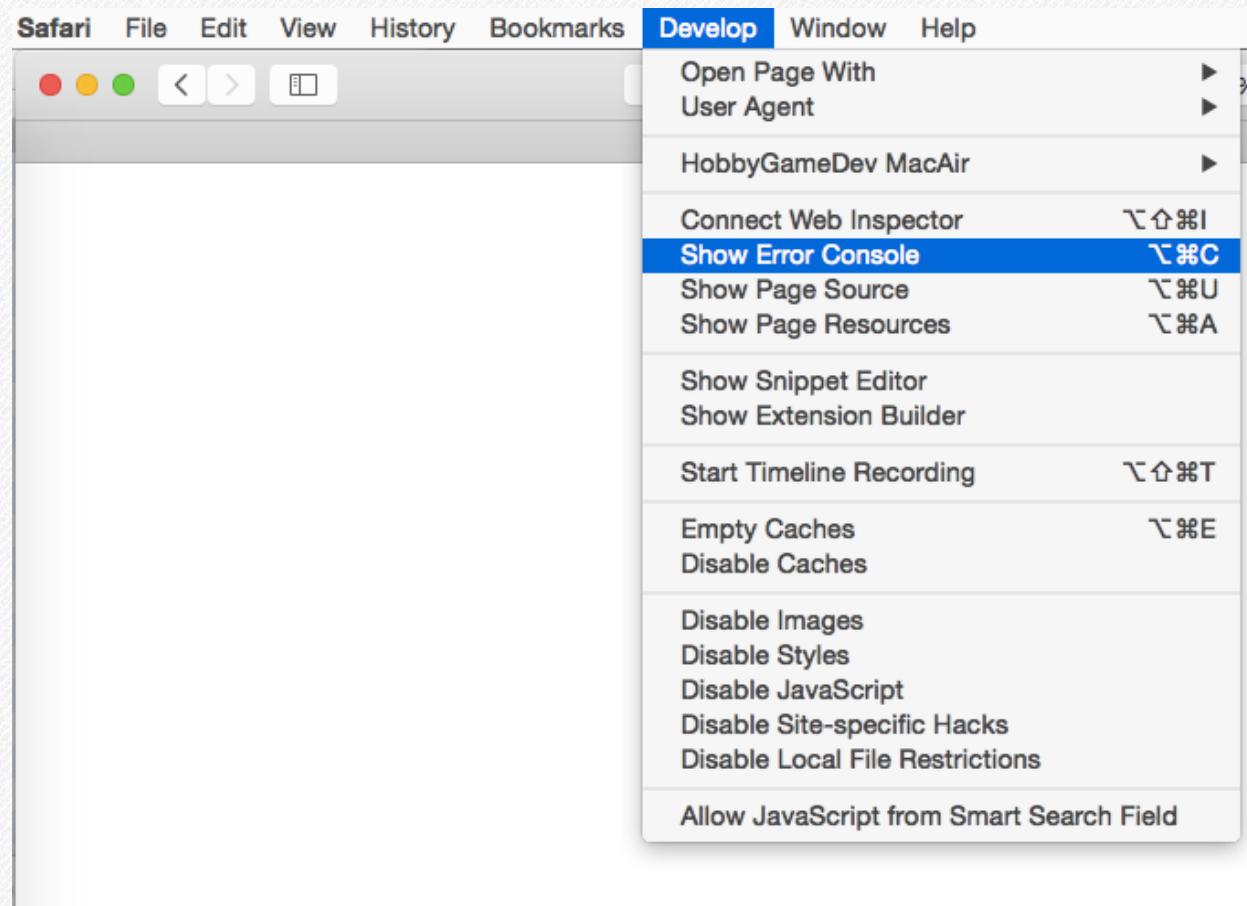
OPENING THE JAVASCRIPT CONSOLE IN WINDOWS INTERNET EXPLORER

Click the Gear in the top right of the window, then choose Developer Tools from the menu. At the bottom you may initially see the DOM Explorer, but to change it to Console Mode click the second option, a rectangle with a greater-than in it. *Then refresh the page, because Internet Explorer only shows log text reached in code while the console is already open (!).* Also with Internet Explorer you might need to click on “Allow Blocked Content” after opening your file to enable JavaScript to run.



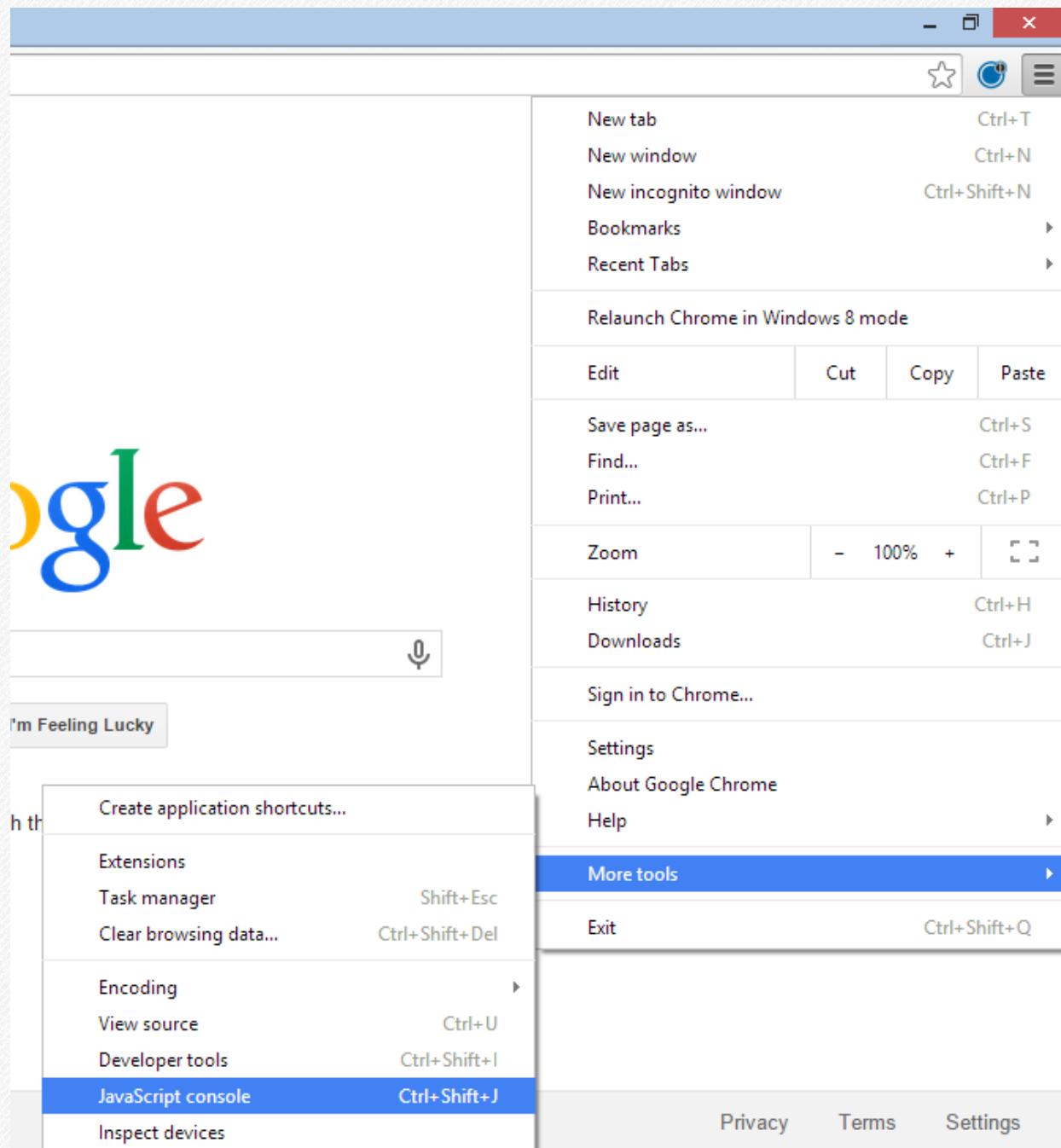
OPENING THE JAVASCRIPT CONSOLE IN APPLE SAFARI

In the menu bar select the Develop dropdown. Show Error Console is the option that you're looking for in there.



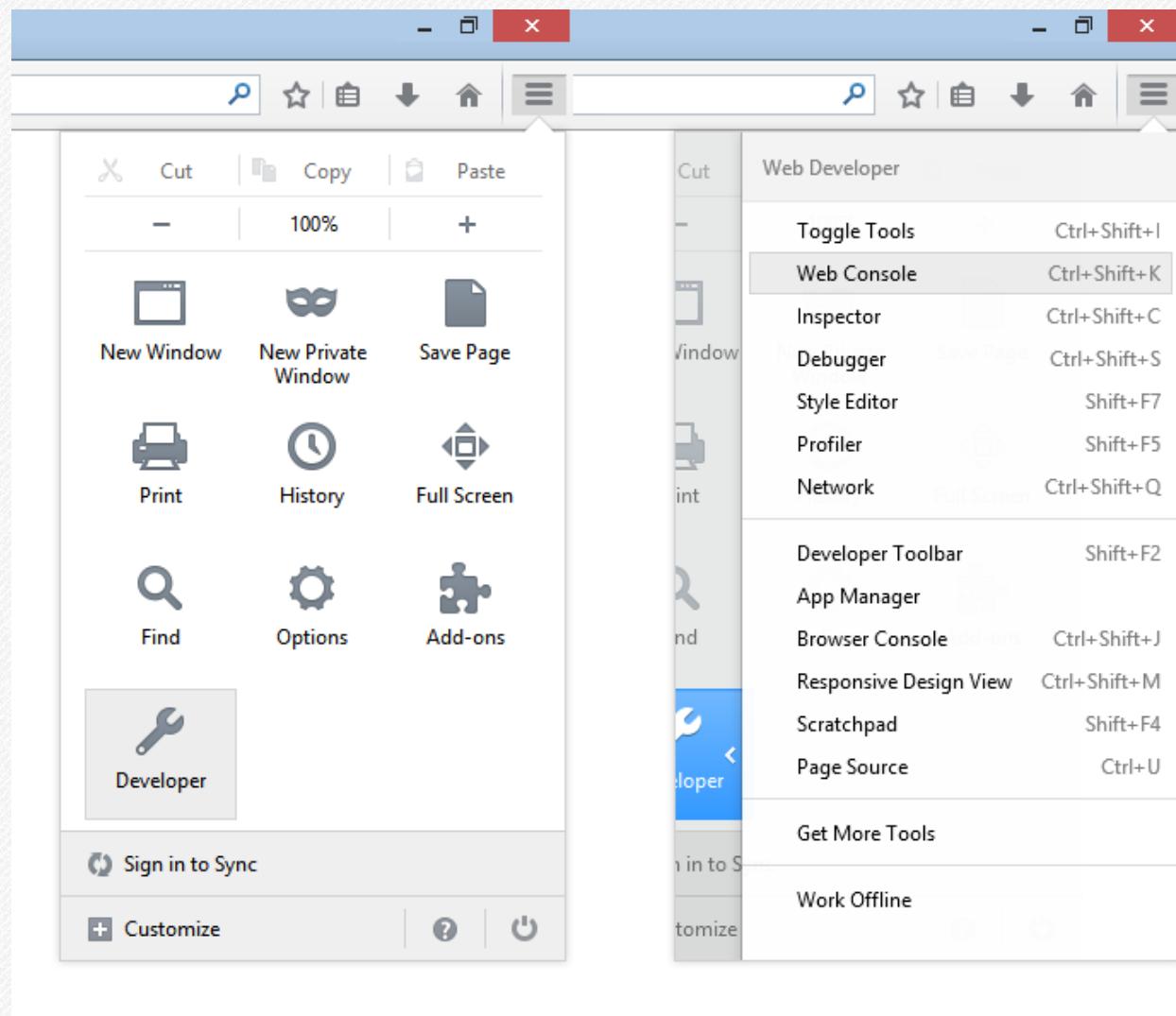
OPENING THE JAVASCRIPT CONSOLE IN GOOGLE CHROME FOR WINDOWS OR MAC

Click the three horizontal bars button in the top right. It's under "More Tools" as "JavaScript console."



OPENING THE JAVASCRIPT CONSOLE IN MOZILLA FIREFOX FOR MAC AND WINDOWS

If you're using Firefox, click on the button in the top right of the browser that has three horizontal bars, then the Developer wrench icon, followed last by the Web Console option.



WARNING ABOUT BRIEF ADDITIONAL DISCUSSION FOR NEW PROGRAMMERS

If you've done any programming before (including non-game programming) and already have some idea what you're looking at in the code so far, you may wish to skip to the **Expected Result** at the end of this section. Any time you're feeling like something I'm describing covers what you're already learning feel welcome to skim through to the part that next interests you. Everyone begins their game development journey with a different background and level of computer familiarity. Next I'll take a minute to break it down in further detail for the people who may be joining us that have not programmed before at all. You may also find the next section or two largely skippable for the same reason. The pace will pick up. See you when it does.

...giving experienced programmers a chance to skip ahead to the Expected Result section...

It's about time those people left, am I right? Goodness. You'd think 2015 people would be past all that, but nooo, they barge right in and read this section too that they were specifically asked not to read. What nerve.

Anyhow, "Hello World!" is a common first statement to print for new coders. My first program printed it in 1996. People have been doing it since the mid-1970's. Welcome to the club!

The curly braces { } mark the scope of the function. That's jargon for saying that any code before the { or after the } won't be included in what window.onload will run once the page finishes loading. You can add more console.log() lines between those braces and you'll see them print out in your JavaScript console from top-to-bottom in whatever order you put them in.

Mini-exercise: have it print your name to the console log. It's strange how good that can feel if you're completely new to this stuff. Here's an example of that going on, and then some:

```
window.onload = function() {
    // window.onload gets run automatically when the page finishes loading
    console.log("Hello World!");
    console.log("Here's another line of text going to the JavaScript console");
    console.log("Jimmy Joe Bob!");
    console.log("PROGRAMMING RULES OMG WHEEEEEEE!!1!!1!ONE");
}
```

The plain English words to the right and on the same line as the // markings are a comment. A program completely ignores comments when running. These are useful to leave notes for your future self or other programmers. In example solutions I'll use comments to explain weird code nearby.

The // comment marker can also be added to the left of any line to get the browser to skip running that code, making it easy to test how your program works differently (or doesn't!) with that one line taken out. This is better than deleting the line because you can bring it right back by removing the comment marks from it. So, for example, by changing that previous example code by adding // comment markers left of one of the four lines, like so...

```
window.onload = function() {
    // window.onload gets run automatically when the page finishes loading
    console.log("Hello World!");
    console.log("Here's another line of text going to the JavaScript console");
    // console.log("Jimmy Joe Bob!");
    console.log("PROGRAMMING RULES OMG WHEEEEEEE!!1!!1!ONE");
}
```

...that would be functionally equivalent to typing this instead:

```
window.onload = function() {
    console.log("Hello World!");
    console.log("Here's another line of text going to the JavaScript console");
    console.log("PROGRAMMING RULES OMG WHEEEEEEE!!1!!1!ONE");
}
```

The only difference between the version with and without the comments is purely that humans can see them. The purpose of comments is mainly for developers. Keep in mind though that whenever you post an HTML file on the internet anyone can use a standard browser's View Source option to see and read your comments alongside the rest of the code. Don't say anything in them that you wouldn't want a stranger reading!

If you ever need to put a comment in the HTML area of code, HTML also supports comments, but it uses different notation. A comment in HTML is made between <!-- at the start and --> at the end. It can be on one line or it can span multiple lines.

Anything between the <script> and </script> tags has to be JavaScript instead of HTML, so <!-- HTML comments --> can't be used there. JavaScript does have a similar form of start-to-end comments marked by a /* at start and a */ at the end (the // is only for one-line comments).

Here are examples of both (you don't need to type this code):

```

<!DOCTYPE html>
<html lang="en">
<head> <!-- here's an HTML comment -->
<meta content="text/html;charset=utf-8" http-equiv="Content-Type">
<meta content="utf-8" http-equiv="encoding">
</head>
<body>
<!--
As long as it's between the start
and end marks then it's a valid
comment and won't show up on the
page. But people can View Source
in their browser and see it!
-->

<script>

/*
Here's how a multi-line comment
looks in JavaScript. Just like
the HTML one it won't be used by
the browser, but is still visible
when someone does a View Source.
*/



window.onload = function() {
    /* you can still use the start-to-end style comments for one line */

    // window.onload gets run automatically when the page finishes loading
    console.log("Hello World!");
    console.log("Here's another line of text going to the JavaScript console");
    /* or to put a comment before the end of a line */ console.log("Jimmy Joe Bob!");
    console.log("PROGRAMMING OMG WHEEEEEEE");

}
</script>
</body>
</html>

```

The semicolon at the end of each program line other than the ones that end with an open brace { or close brace } is required by virtually all programming languages that are typically used for developing computer games. I specify *virtually all* because there are exceptions. In particular: *JavaScript is actually one of those exceptions*. These programs I'm walking you through will work the same if you don't end each line with a semicolon. JavaScript does its best to assume where they're needed, based primarily on where you break to the next line.

That said, it's really good as a long-term habit to add a semicolon to the end of your lines anyway. This is for three reasons:

1. Sometimes JavaScript assumes incorrectly where a semicolon ought to be, leading to a program that runs slightly differently than what you expect from how your code looks. The situation is rare, but that means you're even less likely to suspect or catch it when it's the problem. That can be a real hair-pulling nightmare to troubleshoot.
2. By being in the habit of adding semicolons at the end of your lines you'll be better prepared to quickly adapt to other programming languages later, most of which will require the line-ending semicolons. Most programmers wind up using a number of different programming languages over the years. It's not realistic to think JavaScript is the only one you'll use.
3. Many other coders who are used to seeing line-ending semicolons may be distracted by their absence if you write your programs without using semicolons to end lines, even if they are technically optional.

I'm not even kidding about #3. That's a real thing. Someone sees enough code in enough programming languages over the years and suddenly looking at script that lacks semicolons can seem as distracting as a research paper handed 2 u in txt.

At some point in the future other programmers will see code you write. Maybe you'll be working together on a project, maybe you're asking for help or feedback, or maybe someone is curious to inspect how you made something happen. If your

code lacks semicolons the first thing you may hear is, “jeez, why don’t you use any semicolons?” instead of something meaningful or useful. It’s like the programming equivalent of annoyingly bad grammar.

Indenting code is technically optional too. If you ever show another programmer code where the indentation isn’t done at all or is done all haphazardly thought you’ll need to fix it before they’ll look at it. Global variables (usable from anywhere in the code) don’t even have to be declared in JavaScript, as they’ll be created automatically when used. This is another aspect of coding though where, as a habit, it’ll help to write clear code.

Coming back to this specific code, there’s actually only one line so far that the (optional) ending semicolon is needed for:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta content="text/html;charset=utf-8" http-equiv="Content-Type">
<meta content="utf-8" http-equiv="encoding">
</head>
<body>

<script>
window.onload = function() {
    // window.onload gets run automatically when the page finishes loading
    console.log("Hello World!");
}
</script>
</body>
</html>
```

EXPECTED RESULT

You won’t see anything yet upon dragging this file onto your browser to test it. You should just see a blank white page.

Open the JavaScript / Developer / Web Console mode in your browser, and there you should see “Hello World!” printed out.

tennis game step 2 fill the background with black

To make a graphical game you'll need a <canvas> element in your HTML. Set its pixel dimensions to 800 by 600. That's large enough to look alright, but not so large that it's going to have issues fitting on most older displays still widely in use:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta content="text/html;charset=utf-8" http-equiv="Content-Type">
<meta content="utf-8" http-equiv="encoding">
</head>
<body>
<canvas id="gameCanvas" width="800" height="600"></canvas> <!-- //// -->
<script>
```

That <!-- //// --> is because I'm putting a /// comment in the example solutions on every line that I change or update for each step. The /// mark can stand on its own to the right of a JavaScript line since the first // marking in it identifies the next two as part of a harmless comment for the browser to ignore. That canvas line is a change to the HTML though, where // comments don't work, so it's wrapped in an HTML comment.

Then in the JavaScript <script> block, set up variables for the canvas and its drawing context, saving references to them in the window.onload function:

```
<script>
var canvas; ///
var canvasContext; ///

window.onload = function() {
    console.log("Hello World!");

    // save the canvas for dimensions, and its 2d context for drawing to it ///
    canvas = document.getElementById('gameCanvas'); ///
    canvasContext = canvas.getContext('2d'); ///
}
```

The “var” shown before the canvas and canvasContext labels sets up a data container for each **variable**.

Many other programming languages require different types of variables for storing different types of information. Common examples include ‘string’ for text, ‘int’ for whole numbers (i.e. **integers**), ‘float’ for decimal or fractional numbers (**floating point**, a technical term for these), and custom types for complex groups of related data like the canvas or its context.

JavaScript, on the other hand, lets you designate any data as “var” and will infer from how you set and use it what type of data it contains. If you set a variable like this (you don’t need this or the next one, these are just examples)...

```
var firstName = "Laura";
```

...then the program will see that it’s a text string. If you do this:

```
var favoriteNumber = 3;
```

...then the program will recognize that it’s a whole number.

The program has no understanding of what the “firstName” or “favoriteNumber” mean, those are arbitrary labels that you can decide. Those are the variable names. The value for firstName could even be set to 3 instead of a string, but that’s a terrible idea. The main purpose of naming a variable is to give yourself a reminder of what you mean to do with that piece of data, or to put that another way, what the data represents.

The equal sign in programming doesn’t work the same way as an equal sign in math class. There’s no balancing of equations happening in code. A single equal sign acts as an assignment

operation, calculating or taking the value from the right of it then saving that value into the variable to the left of it. (I say *single* equal sign because a double like == works differently – that checks if the values on both sides of it are the same.)

JavaScript knows what type of data the variable is intended for based on what you save to it. When you set the contents of the canvas variable, it doesn't care whether you name the variable “canvas.” What matters is that you set it to store a reference to “document.getElementById(“gameCanvas”)” – corresponding to the id=“gameCanvas” label of the <canvas> in the document (which refers to the page's HTML).

With that canvas reference you can get its width and height, the 800 by 600 from the HTML element. This way the code can still work even if you change the size of the canvas. With the canvas dimensions in JavaScript you can do other things, too, like checking if a ball's position has left the boundaries of the game, or calculate the center position on the screen.

The canvasContext variable is a reference that you'll be able to use for draw calls to update the canvas's graphics. You'll use it to paint rectangles, circles, text, and loaded image files.

Here's how to draw a black rectangle over the whole canvas:

```
window.onload = function() {
    console.log("Hello World!");

    // save the canvas for dimensions, and its 2d context for drawing to it /////
    canvas = document.getElementById('gameCanvas'); /////
    canvasContext = canvas.getContext('2d'); /////

    canvasContext.fillStyle = 'black'; /////
    canvasContext.fillRect(0, 0, canvas.width, canvas.height); /////
}
```

Where ‘black’ is specified as the `fillColor` you could instead give any hex-style Red Green Blue color. To use pure red, for example, this format would do the same as setting it to ‘red’:

```
canvasContext.fillStyle = '#FF0000';
```

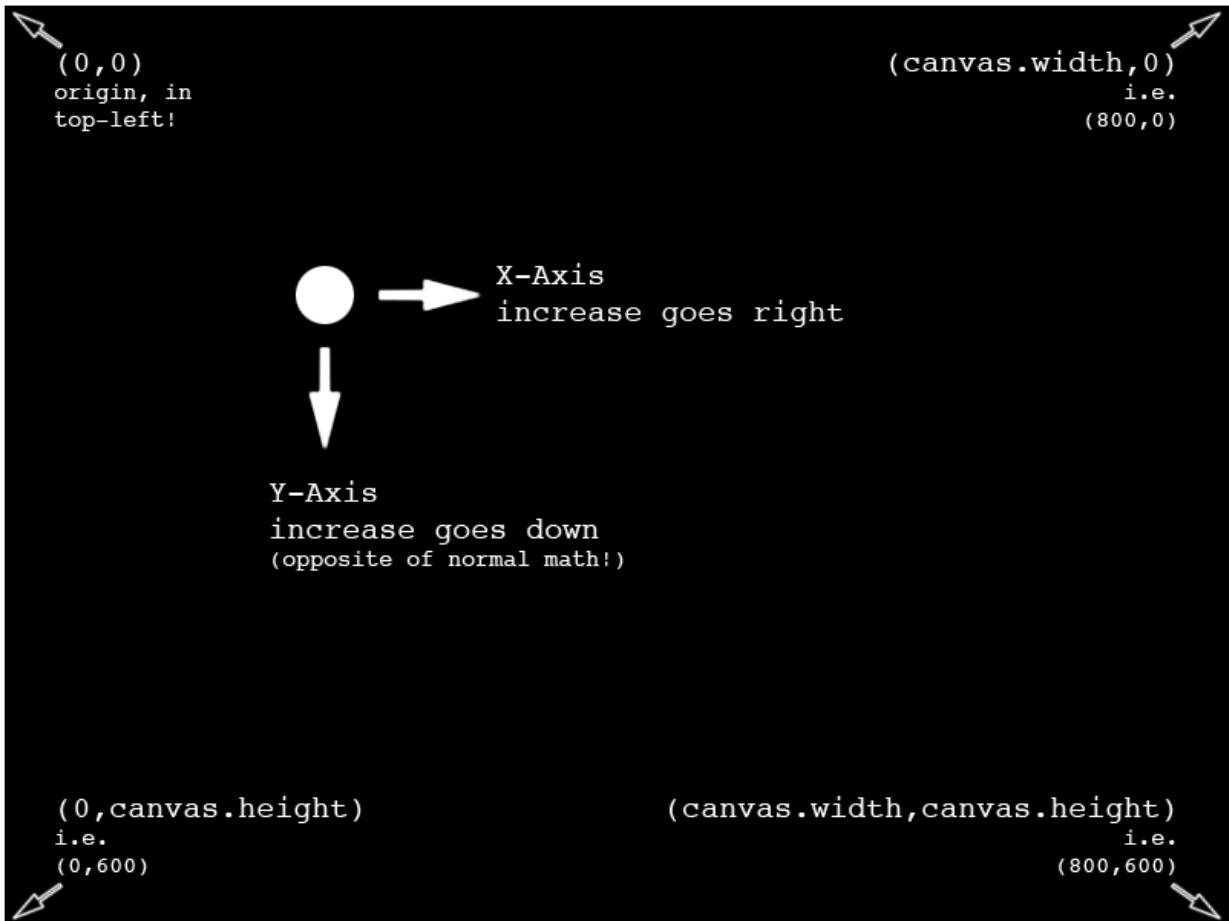
There are several common named colors that can be set by name. If you know or look up hex colors though any color is possible. If you’ve never used those and are curious search for “hex colors” on the web. I won’t use those elsewhere, though.

This function is showing why it’s useful to have the `canvas` and `canvasContext` variables prepared:

```
canvasContext.fillRect(0, 0, canvas.width, canvas.height);
```

That draws a filled rectangle the size of the canvas that’s snug up against the top-left corner of the window. The first number for `fillRect()` is how many pixels from the left side of the canvas the rectangle starts. The second number is how many pixels from the top side of the canvas the rectangle starts. Another way of looking at it is that those first two numbers represent the `x` and `y` for the coordinate where the top-left corner of the rectangle will be drawn.

In math classes the origin point at $(0, 0)$ is usually in the bottom-left corner, and an increase in y is upward. In many computer graphics contexts – this one included – the origin at $(0, 0)$ refers to the *top-left* corner of the display, and an *increase in y is downward*. This will matter soon when drawing and positioning the paddles. The important thing here though is that the first two zeros in that argument refer to the top-left corner of the game’s canvas area.



The third and fourth arguments are for the width and height of the rectangle to fill in. You could put numerical values in there directly. For example, you can create a 100x200 pixel square 50 pixels from the left and 75 pixels from the top by changing that `fillRect()` call to use these values:

```
canvasContext.fillRect(50, 75, 100, 200);
```

By instead using `canvas.width` and `canvas.height`, the program ensures that the filled rectangle will be the exact size of the canvas area specified in the HTML `<canvas>`: 800 by 600.

The use of this generic filled rectangle function may already be familiar if you've programmed graphics in another environment before. If this is your first time putting graphics on the screen

by programming, take a few moments and try filling in different numbers for the arguments of the `fillRect()` call. Save the file and refresh the browser between changes to get a sense for the numbers relate to the rectangle's position and dimensions.

The `canvas.width` and `canvas.height` values can also be used as part of an expression. If you wanted to draw the black rectangle as half the width of the game's whole canvas, only 23 pixels tall, and up against the bottom-left corner of the canvas, you could write the following:

```
canvasContext.fillRect(0, canvas.height-23, canvas.width/2, 23);
```

Where you'll really see these values used is any time the center of the screen, a distance from an edge, or a position along a boundary is needed. Using those variables ensures the code stays right even if you change the canvas dimensions.

Notice too that the dimensions, and thus corner, refer to the 800x600 canvas within the HTML. That's not the same as the bottom corner, nor the dimensions of the web page or browser window. This type of draw code can only draw on the canvas area. To better mark the edges of that drawable region, set the `fillRect()`'s arguments back to their first values and fill the whole canvas before proceeding.

EXPECTED RESULT

The page should now show an 800x600 black rectangle. If you're not seeing the rectangle, ensure that you've saved the text file and refreshed the browser page to see the changes reflected. Otherwise, check the browser console for errors.

tennis game step 3 draw the white ball

Before drawing a second shape, cut/paste the black rectangle draw lines over into a separate function, and call that function from the window.onload setup code:

```
window.onload = function() {
    // removed a console.log() debug statement from here /////
    // save the canvas for dimensions, and its 2d context for drawing to it
    canvas = document.getElementById('gameCanvas');
    canvasContext = canvas.getContext('2d');

    drawEverything(); /////
}

function drawEverything() { /////
    // clear the game view by filling it with black
    canvasContext.fillStyle = 'black';
    canvasContext.fillRect(0, 0, canvas.width, canvas.height);
}
```

This is the most basic use of a function: Here you grouped a block of code under a shared label. Now you can run it all by merely calling that function's name. This use of a function does not accept any input arguments and does not return any output values. By simply giving a descriptive, human-readable label to the block of code between its braces using a function in this way makes the code more readable and easier to grow.

For the code inside of drawEverything() to run, the label needs to be called from within window.onload. You can test that easily by commenting out drawEverything()'s line in window.onload, saving, then refreshing the page. That should then show no black rectangle until you remove the recently added comment marks.

In your new drawEverything() function, at the very end of it after the black background rectangle has been drawn, draw a

20x20 filled white circle (diameter of 10) anywhere visible on the screen.

In my example solution I'll put the circle 75 pixels from the left edge and 75 pixels from the top.

Here's how the whole file looks:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta content="text/html;charset=utf-8" http-equiv="Content-Type">
<meta content="utf-8" http-equiv="encoding">
</head>
<body>
<canvas id="gameCanvas" width="800" height="600"></canvas>

<script>
var canvas;
var canvasContext;

window.onload = function() {
    // removed a console.log() debug statement from here /////
    // save the canvas for dimensions, and its 2d context for drawing to it
    canvas = document.getElementById('gameCanvas');
    canvasContext = canvas.getContext('2d');

    drawEverything(); /////
}

function drawEverything() { /////
    // removed another console.log() debug statement from here /////
    // clear the game view by filling it with black
    canvasContext.fillStyle = 'black';
    canvasContext.fillRect(0, 0, canvas.width, canvas.height);

    // draw a white circle /////
    canvasContext.fillStyle = 'white'; /////
    canvasContext.beginPath(); /////
    canvasContext.arc(75, 75, 10, 0, Math.PI*2, true); /////
    canvasContext.fill(); /////

} /////
</script>
</body>
</html>
```

Although for this step I'm still showing the `////` for changed lines, and the whole program so far with each instruction, the

program will quickly grow too large to do that. You'll still find the `///` marks in the example solutions outside the book. You'll know which lines are changing because aside from a few next or previous lines for context, the code shown on the pages here will often only be the ones that I add or change.

You've likely noticed that drawing a filled circle takes several more lines than the rectangle. Each of those is doing a different part of the work:

1. `fillStyle` sets the color. ('white' is another built-in color)
2. `beginPath()` tells the browser that this is a new, separate shape, rather than a continuation of previous shapes.
3. `arc()` is describing the circle. It takes five arguments. The first two are the x and y coordinate of the center (75 from left, 75 from top). The third is the radius (10 pixels). The fourth and fifth numbers are the start and end angles (0 to 2π radians, for a full circle). The fifth argument indicates that the arc should be drawn counterclockwise (true), though directionality makes no difference since it's for a full circle.
4. `fill()` will color in the shape formed by the recent arc or whatever other draw code has been used since `beginPath()`.

Before you draw more shapes, you'll be grouping these lines of circle and rectangle drawing code into new functions, giving you an easy way to draw additional colored circles with one line of code for each.

The white circle must be drawn after painting the canvas black or it won't show up. Just like with having multiple `console.log()`

statements in a row, code runs from top-to-bottom. With graphics code this means that later draw calls overlap any prior ones. If the white circle gets drawn then the black rectangle gets painted over the whole canvas, the white circle winds up painted over or conceptually behind the black rectangle.

Even though the code within `window.onload` runs step by step, line by line, top to bottom, the time that the computer needs for doing so is practically instantaneous, far too fast to be perceptible. It's not as though putting these lines out of order you'd see a circle drawn, and then the black rectangle, it's all happening much faster than your screen can even show, meaning you would only see the black rectangle.

EXPECTED RESULT

You should now see a small white circle drawn within the large 800x600 black rectangle.

Take a moment to check the developer console. Even though there are no `console.log()` statements in the code, every few changes the console is good to check for errors.

That's true even if the game seems to be working as expected. Not all issues are visible in what the program does. Those kinds of deeper issues can be harder to deal with when they surface and have an impact later. Fixing errors is always easier to do before piling new layers of functionality atop the existing codebase. This approach also reduces how much recently changed code you'll have to check through for issues.

tennis game step 4

change ball position automatically

Set up two new variables to use as the ball's horizontal and vertical position in the canvas. Set each to 75 where they're declared. Update the circle code to draw the ball with those coordinate variables as the first two arguments that indicate the circle's center point.

Here's an excerpt showing the latest `<script>` block:

```
var ballX = 75, ballY = 75; /////
var canvas;
var canvasContext;

window.onload = function() {
    // .. skipping over the unchanged definition here on the book page ...
}

function drawEverything() {
    // clear the game view by filling it with black
    canvasContext.fillStyle = 'black';
    canvasContext.fillRect(0, 0, canvas.width, canvas.height);

    // draw a white circle
    canvasContext.fillStyle = 'white';
    canvasContext.beginPath();
    canvasContext.arc(ballX, ballY, 10, 0, Math.PI*2, true); /////
    canvasContext.fill();
}
```

Now that you have variables for the ball's position, rather than hard-coded numbers, you'll be able to change the variable values during the game to change where the ball gets drawn.

To see the ball drawn in a different position it will need to be drawn more than just the first time. Much like a cartoon, animation in a computer game requires a series of images, each step having minor changes to how it looked the frame before. With the program drawing repeatedly, adjustments made to the position variables will be observable in real-time.

In window.onload replace the drawEverything() call with this:

```
setInterval(drawEverything, 1000);
```

That line uses JavaScript's built-in setInterval feature to call any function given to it (here: the drawEverything() code) every N number of milliseconds, where N is the second argument. 1,000 milliseconds is the same as 1 second, so what this line will do for you is call the drawEverything() code once each second. It will start doing so as soon as the page has finished loading, since the instruction is given within window.onload.

Whereas before the program could only run its code once from the top of window.onload through whatever it called once from there, now the program has a part that will be repeating over and over as time progresses. This makes it possible for us to simulate and display movement. Before, the code could only construct a still picture. Now the code can show motion.

As-is, every time the drawEverything() function gets called it is drawing the same shapes in the same positions. To see the utility of using a variable for the ball position, change it by some amount each time draw() is called by setInterval(). To do that, add this early in drawEverything() between its { and }:

```
ballX += 50;  
console.log("ballX is now: " + ballX);
```

Now seems like a good time to stress that program code is case sensitive. If you typed ballx there instead of ballX the program won't work as expected, because ballx is seen by the code as a different variable name than ballX. Whenever writing variable names or function names it's important that they're always capitalized the same in all places that they're used.

With that, the ball will move 50 pixels right every time the draw function gets called by setInterval(). The console.log() will display on the JavaScript console the current horizontal position of the ball right after each moment that it changes.

Quick aside for new programmers: that `+=` adds the value on the right to the value on the left, *then saves the outcome into the variable on the left*. Consider (don't add) this snippet:

```
var b = 100;
console.log(b);          // logs, as expected: 100
console.log(b + 25);     // logs, due to addition: 125
console.log(b);          // logs, still: 100
console.log(b += 25);    // also logs: 125
console.log(b);          // logs, since b changed: 125
```

Typing `b + 25` or `ballX + 50` doesn't actually change the value saved to that label. The equal sign is the assignment operator and is necessary to change the stored value.

`A += B` is actually a shortcut to writing `A=A+B`. The purpose of an equal sign in programming is to set the variable on its left to whatever is calculated by the expression on its right. As said earlier, it has nothing to do with balancing algebra equations. This is also why in programming you can write `ballX = 75` to change `ballX` to 75, but the line `75 = ballX` is useless and not valid because only variable labels can be placed on the left side of an equal sign. 75 is not a valid variable label. In a similar fashion to `+ =` you can use the `- =` symbols to subtract, `* =` to multiply, and `/ =` to divide a label on their left by some value to the right.

EXPECTED RESULT

For the first second the game will not yet run `drawEverything()` and you'll see a blank white page. At the end of that second

and for each second thereafter you'll see the white ball drawn onto the black rectangle background, moving 50 pixels right each time it gets drawn. With each new draw update, the console will announce the ball's latest horizontal position. Left running long enough, the ball will jump off the right side of the viewable canvas area, but as evidenced by the console numerical print reports it'll keep on moving rightward virtually indefinitely. (At 50 pixels/second: 5.7 million years.)

tennis game step 5 get the ball moving smoothly

Getting the ball moving smoothly rather than in big jumps won't require much of a code change. Still, doing so will open up motion and input response possibilities in the way that's needed for real-time gameplay. All that has to be done is more frequent, smaller updates. That means instead of increasing ballX by 50 pixels once per second, increase it by 2 pixels thirty times per second.

Separating out the motion logic from the drawing instructions will reduce future issues. Mixing those two together can lead to confusion about which objects are updated or still in their old positions during collision tests. Plus, to later fix or improve the game's movement or graphics code, you shouldn't need to dig through both to locate the part you're working on. Remove the line from the top of drawEverything() that added 50 to ballX, and remove that console.log() statement there as well. Make a new function, defined just before drawEverything(), that adds 2 to the value stored in ballX:

```
function moveEverything() {
  ballX += 2; // move the ball to the right by a small increment
}

function drawEverything() {
  // clear the game view by filling it with black
  canvasContext.fillStyle = 'black';
  // ... etc, rest of drawEverything() is unchanged here ...
```

Now the question is where to call moveEverything(). You *can* call the new moveEverything() function directly from the top of drawEverything(), but that kind of muddies the desired separation between them a bit. Instead, change setInterval() in

window.onload so that it now calls both moveEverything() and drawEverything(), doing both at a small enough interval to happen at 30 times every second:

```
var framesPerSecond = 30;
setInterval(function() {
    moveEverything();
    drawEverything();
}, 1000/framesPerSecond);
```

There're two things to notice here.

First, rather than solve for “how many milliseconds between each update will result in 30 frames per second” I instead let the computer solve that math problem for us by creating a local variable and using it in a simple calculation. This makes it easy to see in the code how many frames per second you set the program to run at, and also easy to change it if needed.

By local variable, I mean I put the var before framesPerSecond right here inside the window.onload function, unlike the other var values which were set up outside the functions. Whereas the var values for ballX and canvasContext have to keep and maintain their values between update frames, the frames per second value won't be needed again after its use here. By using var to create this variable inside of window.onload it will be lost when the function reaches its closing brace }. If you try to console.log(framesPerSecond) in drawEverything() you'll see console errors since the value label won't exist outside the function in which it was declared.

The second change here is in the function that setInterval() is now calling repeatedly. To get setInterval() calling two functions every frame, they're wrapped in a new function

defined on the spot. That's what the `function() {}` notation accomplishes. Unlike other function definitions shown so far this one is anonymous – doesn't have a label. It won't need one. It's only needed there, as an argument for `setInterval()`.

Using that function defined on the spot, or inline, is identical to the doing this (no need to change your code in this way):

```
// ... the rest of window.onload above this point is unchanged
var framesPerSecond = 30;
setInterval(callBothMoveAndDraw, 1000/framesPerSecond);
}

function callBothMoveAndDraw() {
moveEverything();
drawEverything();
}
```

However, doing that produces an unnecessary helper function `callBothMoveAndDraw()`. It has only two lines, which are not likely to change much or ever be called from anyplace else. The code which may need expansion is already tucked away nicely into `moveEverything()` and `drawEverything()`. For small functions which won't need much work done to them, making one up on the spot that serves its purpose is a viable option. If it winds up growing in complexity you can always go back to pull it into a separate function.

One last time, here's how the latest script block fits together (for future changes, refer to the provided example code if you're unsure how pieces fit together):

```

<!DOCTYPE html>
<html lang="en">
<head>
<meta content="text/html;charset=utf-8" http-equiv="Content-Type">
<meta content="utf-8" http-equiv="encoding">
</head>
<body>
<canvas id="gameCanvas" width="800" height="600"></canvas>

<script>
    // variables to keep track of ball position
    var ballX = 75, ballY = 75;

    // save the canvas for dimensions, and its 2d context for drawing to it
    var canvas, canvasContext;

    window.onload = function() {
        canvas = document.getElementById('gameCanvas');
        canvasContext = canvas.getContext('2d');

        // these next few lines set up our game logic and render to happen 30 times/second
        var framesPerSecond = 30;
        setInterval(function() {
            moveEverything();
            drawEverything();
        }, 1000/framesPerSecond);
    }

    function moveEverything() {
        ballX += 2; // move the ball to the right by a small increment
    }

    function drawEverything() {
        // clear the game view by filling it with black
        canvasContext.fillStyle = 'black';
        canvasContext.fillRect(0, 0, canvas.width, canvas.height);

        // draw a white circle
        canvasContext.fillStyle = 'white';
        canvasContext.beginPath();
        canvasContext.arc(ballX, ballY, 10, 0, Math.PI*2, true);
        canvasContext.fill();
    }
</script>
</body>
</html>

```

EXPECTED RESULT

The white ball should now move slowly in a straight line toward the right until it leaves the the black viewable area.

This latest change now has the program changing and moving automatically on its own. In this way it can continue to perform calculations over time a long as it's open. The draw code in previous steps ran once then remained inactive. Running code even while the player isn't providing fresh input is central to all motion and smooth interactive experiences.

tennis game step 6 variable for the ball's speed

Rather than having the ball's movement speed hardcoded as 2 pixels per frame, create a new variable which you'll set up to be the ball's current horizontal movement speed. Start it as 2. To do this, directly below where ballX and ballY are set up as 75, add this line:

```
var ballSpeedX = 2;
```

Merely declaring that variable and putting a 2 in it does not change any functionality though. The name of that variable, "ballSpeedX," has literally no meaning to the computer. It's up to you to plug it into the code where it will fulfill the purpose suggested by its name, down in the moveEverything() function:

```
function moveEverything() {  
    ballX += ballSpeedX;  
}
```

Previously, the ball always moved at the same speed to the right, even as it passed the edge of the screen. Now, you'll be able to change the speed of the ball by writing code that changes ballSpeedX during play.

EXPECTED RESULT

No change to what the program does. The white ball will still drift slowly off the right side. The ballSpeedX variable never changes from its default initial value of 2. Try setting that number to other values though then refresh the browser to see what different values do. 4 will go twice as fast right, 6 will go to the right even faster. What happens when you set it to -2?

tennis game step 7 bounce ball off of right edge

Add a new if-condition in the moveEverything() function to determine whether the ball has moved beyond the right edge of the canvas. If it has, reverse the ball's direction by making its speed negative. At the end of the function, add a console.log() of ballSpeedX to display its speed at each moment in the browser console:

```
function moveEverything() {  
    if(ballX > canvas.width) { // if ball has moved beyond the right edge  
        ballSpeedX *= -1; // reverse ball's horizontal direction  
    }  
  
    ballX += ballSpeedX; // move the ball based on its current horizontal speed  
    console.log(ballSpeedX);  
}
```

You could write if(ballX > 800) instead of using canvas.width, but referring to its label has two benefits. (1.) If you later choose to rethink your canvas size, this updates automatically. (2.) why we typed 800 is less self-explanatory than seeing that it's a reference to the total width of the canvas.

Multiplying the ball's horizontal speed by -1 will work to reverse the speed, since it preserves the ball's overall rate while reversing direction. When the speed is negative it will move left, since it means that each frame the ball's distance from the left edge will decrease.

To make this faster to test in-browser, I suggest also bumping the ball's starting horizontal speed from 2 up to 6.

EXPECTED RESULT

The ball should still move right at the same speed, except when its center "hits" the right wall it should bounce back left. Given a little more time to continue moving in this version the ball will then go beyond the left edge, leaving the play area on that side instead.

By opening your browser's console you should see the ball's speed right (6 if you increased it as suggested, 2 otherwise). Its repeat counter should count up 30 times each second until the ball touches the right side, at which point a new line should begin that instead prints the negative version of the same number while the ball travels back left.

tennis game step 8 bounce ball off of left edge

Duplicate the previous functionality, except now for the left edge of the screen. In other words, have the ball reverse its horizontal direction when its center crosses the left edge of the canvas play area.

Remove the `console.log()` that was added in the previous step to help keep the console clear.

Even though `console.log()` can be a very powerful and handy tool to know how to use, and has been helpful up to this point, if you get careless about using it while the program grows you can soon wind up with a cluttered, far less useful console. Log messages you don't need anymore makes spotting errors or current `console.log()` messages more difficult. Now that the game is running dozens of frames per second I'll reduce use of `console.log()` in the examples, instead showing other ways to surface debug values while working.

That said, any time you're working on fixing a bug and want to use `console.log()` temporarily to expose a value, go for it. Once you're finished why you added it, remove the `console.log()` call before moving on so that they won't accumulate.

There's no example code in the book for this one. It's a direct change to the pattern. See if you can figure it out. That said, if you find yourself feeling stuck, or are restless to move forward, there's no harm in referring to the solution given outside the book in the example source.

EXPECTED RESULT

The ball should now bounce back and forth indefinitely in a horizontal line. There should no longer be any numbers being printed out to the console.

tennis game step 9 give the ball vertical movement

The past several changes have collectively accomplished making the ball move at a variable rate/direction horizontally, bouncing off the left and right edges of the playfield area.

Repeat those steps except now adding a new variable to manage the ball's vertical speed, and making the ball bounce off the top and bottom edges of the playfield in addition to the left and right sides.

No example code in the book for this one, either. The pattern for the vertical y axis is similar enough to the horizontal x axis movement that you can probably accomplish this step by mimicking the pattern for what's already working. Though again, if you get stuck, check out the example code provided.

EXPECTED RESULT

Now the ball should bounce around off all four sides of the playfield area. It's like the start of an early 1990's screensaver.

tennis game step 10 draw the player (left) paddle

In drawEverything(), after clearing the screen with a black rectangle, draw a small stationary white rectangle 250 pixels down along the left edge of the screen which is 10 pixels wide and 100 pixels tall. This will be for the player's paddle.

Here's the turning point I've been building up to: I'm going to be showing much less code in the book from here on out, at least up until introducing more advanced concepts later.

It may take experimentation, making it do the wrong thing and head scratching, and searching the web. That's a fine and normal part of videogame programming. If it doesn't work the first time, that just means you're still working on it.

If you're coming into this totally new to programming, don't be shy at all about checking the example solutions for reference. You'll have opportunities to practice these concepts in the Section 2 exercises later. If seeing working examples helps get you that far comfortably then there's no shame in using them.

EXPECTED RESULT

The game should look and work the same, except now there should be a thin white rectangle against the middle of the left canvas edge.

tennis game step II

create rect and circle functions

Now that there are two filled rectangles in the code – the background and the paddle – reorganize those lines to be done by a function that combines the rectangle parameters with the color. Above drawEverything(), define:

```
colorRect(topLeftX, topLeftY, boxWidth, boxHeight, fillColor)
```

This should set `fillColor` then draw a filled rectangle using the `fillRect()` call. Replace the draw code for both rectangles in `drawEverything()` with calls to that function. One rectangle is a large black one clearing the screen, the other is a small white one for the left paddle. After `colorRect()`'s definition create another in the same fashion to contain the circle draw code:

```
colorCircle(centerX, centerY, radius, fillColor)
```

Replace the draw code for the ball with a call to that function. Once those two functions are filled in, `drawEverything()` should look like this but display the same visuals when tested:

```
function drawEverything() {
    // clear the game view by filling it with black
    colorRect(0, 0, canvas.width, canvas.height, 'black');

    // draw a white rectangle to use as the left player's paddle
    colorRect(0, 250, 10, 100, 'white');

    // draw the ball
    colorCircle(ballX, ballY, 10, 'white');
}
```

EXPECTED RESULT

Nothing about the game's appearance or functionality should change from this step. The draw code for the shapes in `drawEverything()` should now be simplified.

tennis game step 12

use mouse to position the paddle

Introduce a new variable to keep track of the player's paddle vertical position, as well as a constant value ('const' rather than 'var') for the paddle's height (referring to its tallness, not to its vertical position on screen). 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 vertically where the mouse is, half above the mouse and half below. The horizontal position should stay locked along the left edge, with no change based on mouse input.

This is a step where a little code in the book I think will be helpful again since it's new material. The first thing to do is write a function which can calculate the position of a mouse:

```
function calculateMousePos(evt) {
  var rect = canvas.getBoundingClientRect(), root = document.documentElement;

  // account for the margins, canvas position on page, scroll amount, etc.
  var mouseX = evt.clientX - rect.left - root.scrollLeft;
  var mouseY = evt.clientY - rect.top - root.scrollTop;
  return {
    x: mouseX,
    y: mouseY
  };
}
```

If that function looks ugly, don't fret. Like some of the other set up code it's things that you won't be seeing much elsewhere in these examples. This function contains and conceals the messiness so that you won't need to think about it or deal with it directly in these games to grab the mouse coordinates.

So that the function isn't a magical block box, here's what's going on. The function's only argument is the mouse event's information, which includes the click position within the browser window. What you really want though is the mouse position relative to the game's canvas, accounting for how the page may have been scrolled. That's what's going into finding mouseX and mouseY. Next, those are returned as separate x and y values, so they can be handled as a coordinate pair. The data pair is an Object Literal, a JavaScript feature that won't come up again until loading multiple image files much later.

In window.onload hook up the 'mousemove' event to call that function and use its y position to update paddle1Y:

```
canvas.addEventListener('mousemove', function(evt) {  
    var mousePos = calculateMousePos(evt);  
    paddle1Y = mousePos.y - (PADDLE_HEIGHT/2); // minus half height, to center it  
});
```

This code connects the canvas 'mousemove' event – which triggers each time the mouse moves within the canvas – to the short function written directly as the second argument. The mouse event, evt, then gets passed to calculateMousePos(), which returns mousePos.x and mousePos.y as its output. Lastly, half the paddle's vertical height is subtracted from the mouse's vertical position, to align the paddle's vertical center with the mouse, rather than the top edge.

Again: if the code in this step is a bit gnarly, don't sweat it. Most of what you're writing for gameplay won't involve returning multiple values or accounting for browser scroll. This pattern will only be used again when connecting other input

events, such as keyboard presses and mouse clicks. The gameplay code that it connects to will follow a simpler form.

Since there are a number of separate pieces to this step, here's a quick bullet list to make sure nothing's missed:

- Set up a variable, paddle1Y, for the paddle's vertical position
- Set up a const, PADDLE_HEIGHT, set to 100
- Add the calculateMousePos() function defined earlier
- Hook up a 'mousemove' event listener to grab the mouse x,y coordinates using calculateMousePos() which then sets the paddle's vertical position to half the PADDLE_HEIGHT above the mouse's y-coordinate
- Set the vertical coordinate of the paddle's colorRect() to be the paddle1Y variable

Partly because there are so many parts involved, if you haven't yet taken me up on the offer to open up the example solution files to double check your work, please feel welcome to do so. It really isn't cheating! Guess and check as you go can be a healthy part of testing your understanding incrementally, or learning new things by reading the code as a demonstration.

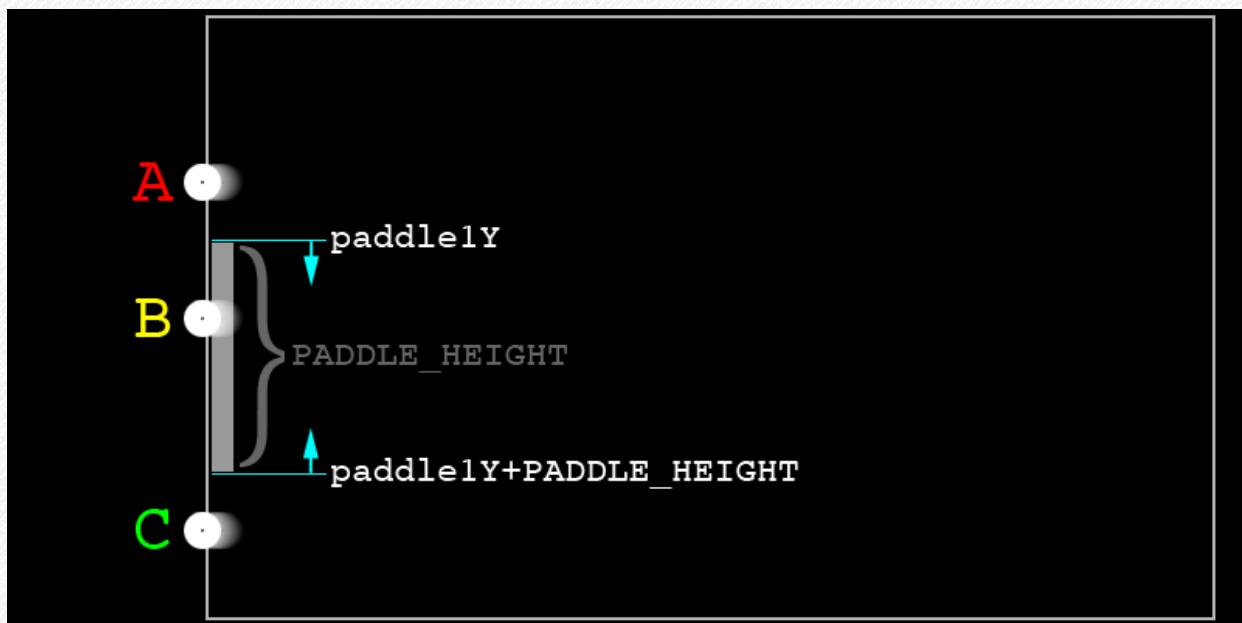
EXPECTED RESULT

Moving the mouse up and down should now keep the left paddle vertically centered on its position over the playfield. At this point the paddle doesn't have any effect on the ball, meaning the ball will still bounce off the left edge no matter whether it lines up with the left paddle or not.

tennis game step 13 reset the ball if the player misses

Now reset the ball 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 a new if-condition to check whether the ball's vertical position variable is both below the top of the paddle and also above the bottom of the paddle. Ignore the thickness of the paddle at this time. Only consider when the ball is crossing the edge of the screen.

Consider three different vertical regions where the ball's center could be as it reaches the left edge of the canvas:



In position A the ball's vertical position, ballY, is less than paddle1Y. Spatially, that means that A is above the paddle.

In position C the ball's vertical position is greater than paddle1Y+PADDLE_HEIGHT. Since paddle1Y is the top, and the paddle is PADDLE_HEIGHT pixels tall, that comparison is finding whether the ball is below the paddle's bottom edge.

Spot B in the center, not too high or too low. In that region ballY is greater than the top (paddle1Y), and also less than the bottom (paddle1Y+PADDLE_HEIGHT). You can check whether two or more comparisons are all true by using `&&` in an if-statement to connect separate considerations with “and also.” Here’s that, in the context of some other example logic:

```
if(ballX < 0) { // if ball has moved beyond the left edge
    if(ballY > paddle1Y && ballY < paddle1Y+PADDLE_HEIGHT) {
        ballSpeedX *= -1; // reverse ball's horizontal direction
    } else {
        ballReset();
    }
}
```

The outer if-statement is just a standard single comparison, determining whether the ball is crossing the screen’s left edge. Notice that the closing brace for that first statement is on the last line, enclosing the whole if-else pair. This ensures that either of those will only happen when the ball passes the left edge of the canvas.

The inner if-statement uses `&&` to tell whether the ball is in region B from the diagram: both below the paddle’s top *and* *also* above the paddle’s bottom. If so, the ball’s horizontal speed is reflected. Otherwise – which is what the corresponding else case refers to – it must be in the same area as examples A or C, and so it should reset to center.

To perform that reset, make a new `ballReset()` function that sets the ball’s horizontal and vertical position variables to the center (half the canvas width as `y`, half the canvas height as `y`).

EXPECTED RESULT

When the ball crosses the left edge if it lines up with the paddle it will bounce back horizontally, otherwise it will instantly reappear in the center of the playfield.

tennis game step 14 draw the right player's paddle

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

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. Because the value for paddle thickness (10) will now be used in multiple places within the source, create a new const at the top of the script to use as a label anywhere the number occurs in code for that purpose. For example, in addition to being used as the width argument of the rectangle function for both player paddles, in order to position the left side of the right paddle its position should be written in code as:

```
canvas.width - PADDLE_THICKNESS
```

...rather than:

```
canvas.width - 10
```

...or (least readable of all, and least automatic to maintain):

790

EXPECTED RESULT

There should be a white rectangle on the right side of the playfield which is identical to the player's, except unlike the player's it doesn't yet either move or have any influence over whether the ball bounces. At this point, the ball still bounces off the right edge every time, even if it misses the right paddle.

tennis game step 15 player (test) control for right paddle

Before automating the right paddle's movement (giving it A.I., often written as just AI, short for "artificial intelligence"), connect that paddle's vertical movement to the mouse height so that you will control the right paddle instead of the left one.

This will be helpful in testing whether the right side paddle's collision code works right, giving us control over whether it is in position to hit or miss. To make testing easier, whenever the ball resets flip its horizontal direction, so that the ball will not get stuck repeatedly missing the left paddle.

EXPECTED RESULT

The left paddle should now sit motionless, whereas the right paddle will be kept vertically centered on the mouse.

The ball will reset if it misses the left paddle, or bounce if it hits the left paddle. The ball will still bounce no matter which part of the right playfield edge it touches, whether or not it lines up with where the right paddle is.

The ball's horizontal direction will flip each time it resets. This got added at this time to help with testing the upcoming right paddle collision functionality, but it makes sense to leave it in. This change has the effect that whoever last scored then gets to serve, choosing how to reflecting the automatic initial shot.

tennis game step 1b let the ball go off the right edge

Reset the ball when it makes it past the right side of the canvas, unless the right paddle deflects it. This should work the same as when the ball hits the left side of the screen.

When the ball is crossing the right edge, check whether the ball's vertical position is below the right paddle's top and above the right paddle's bottom. If so, flip the ball's horizontal speed by multiplying it by negative one. Otherwise, the ball passed the edge, in which case the ball needs to be reset.

EXPECTED RESULT

The right paddle should now deflect the ball if it blocks the right edge when the ball tries to pass. The right paddle is still under mouse control which should make this easy to test when both blocking and not blocking the ball.

Be sure to test blocking and missing the ball both very high and very low relative to the paddle, to ensure that how you're checking for collision matches up to where you're drawing the paddle.

Depending on how you handle collision, it may clip through the inside corners of the paddle if moving diagonally. No need to worry about that detail yet. What's important here is that the ball should reset if it doesn't line up vertically with where the right paddle is when crossing that side.

tennis game step 17

put text on the screen for score

Add white text to the playfield, using the default font and text size. Any text in any position you can see easily will do. For the example solution I'll write the word "stuff" at 100 pixels from the left edge and 100 pixels from the top edge. This is where score text can be displayed next.

UI elements like score text should never be overlapped by playfield objects like the ball. Make this text draw at the end of the `drawEverything()` function, ensuring that it will overlap anything else drawn as part of the current frame.

EXPECTED RESULT

Actual gameplay functionality will be the same as from the previous step: the ball should reset if it passes either paddle, otherwise reflecting back to the playfield.

The only change here is visual, namely that now the word "stuff" should appear in small white letters someplace visible on the playfield the whole time.

tennis game step 18 tracking the left player's score

Add a new variable to keep track of the left player's score.

Start the value at zero. Increase it by one each time the ball passes the right edge without being blocked by the paddle.

Perform the score increase right before resetting the ball from it passing the right edge. By updating the score before calling `ballReset()`, that function will be in a good spot to later check whether either player has enough points to be the winner.

Change the on-playfield text to show that score value instead of "stuff" or your other placeholder word.

EXPECTED RESULT

The text that said "stuff" should now instead begin as "0" and increase to "1" then "2" and so on each time the ball makes it past the right paddle. This score should *not* change when the ball makes it past the left paddle. The mouse will still control the right paddle. That will make it easier to test both reflecting the ball (no score change) and letting it pass (score value displayed increases by one).

tennis game step 19 add score for the right side paddle

Add another text element on the right side of the playfield, and another number variable for the right paddle's score. Follow the previous scoring step logic to increase the right players score whenever the ball resets by passing the left edge.

As part of this change, after testing blocking from right and scoring against the left side, switch mouse control in the code back to be for the left paddle's height to then do the opposite blocking and scoring test as the left paddle.

EXPECTED RESULT

The player should once again be in control of the left paddle, whereas the right paddle will sit motionless.

The left score should now increase whenever the ball misses the right paddle. The right score should increase whenever the ball misses the left paddle.

There's no maximum score at this time. The scores can rise virtually indefinitely, up until hitting the incredibly high limit of how large a number the computer will store in this data type.

tennis game step 20 direct ball by the paddle height hit

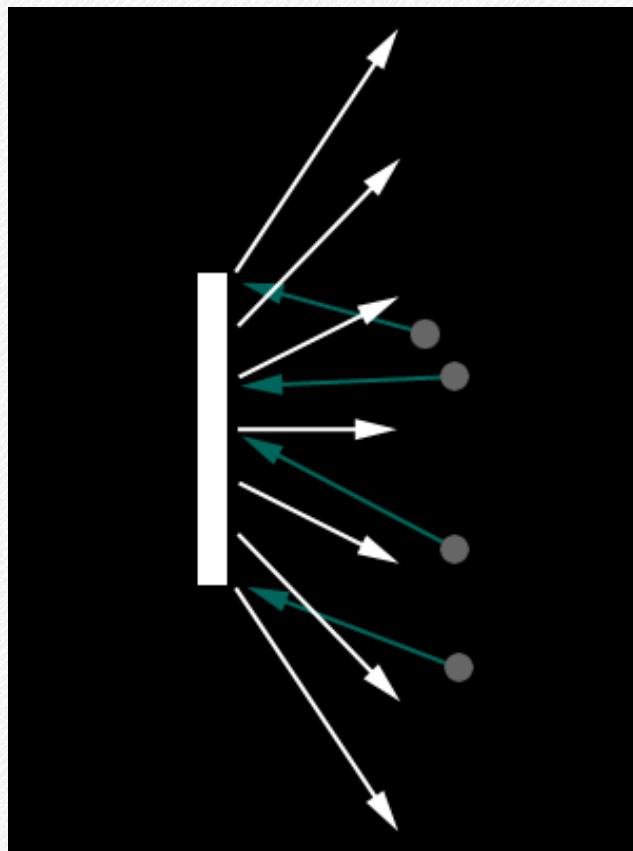
There's currently no way for the player to affect where or how the ball will move after bouncing off the paddle. There's no incentive to try for risky blocks near your paddle's edge. It's always best to line up the center of your paddle with where the ball seems to be headed, leaving maximum margin for error.

To increase player control over the ball, while also adding excitement by rewarding risk taking during play, change the ball-paddle collision response code to allow aiming. Base the ball's vertical speed on the ball distance from the center of the paddle at the time of collision.

If the ball hits the center of the paddle it should go straight back in a horizontal line. The higher on the paddle the ball strikes, the more steeply upward it should ascend. Or, the lower on the paddle the ball strikes, the more steeply downward it should descend.

Let horizontal speed remain the same, independent of the vertical speed. In addition to being easier to code, this makes steeper returns both harder to anticipate and 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 vertical position. Switching which paddle's position updates with the mouse may be helpful to test that this change is working right for both sides.



No matter what speed and direction the ball had coming into the paddle, and no matter how the paddle was moving at the time of collision, the ball return angle is determined by where on the paddle the ball hits. Hitting the center returns the ball horizontally. Nearer to either edge returns it steeper.

EXPECTED RESULT

Hitting the ball against either paddle nearer to the upper edge of the paddle should return it with a more upward speed.

Hitting the ball closer to the paddle center should shoot the ball back closer to horizontal. Hitting the ball lower with the paddle should set it moving more steeply downward.

tennis game step 21 computer controlled right paddle

The right side paddle should be computer controlled rather than sitting still or following the mouse. The mouse y should be controlling the left paddle, so the computer can have full control over the right paddle.

Create a new function named moveComputerPaddle() and call it at the start of moveEverything(). This new function needs to perform a couple of comparisons to make the right paddle move up if the ball is currently above its center, and have the right paddle instead move down if the ball is currently below its center.

Set a modest maximum speed limit to ensure that it's possible for the player to get the ball past it. Since the same speed number will be used in two places in the code (both upward and downward adjustments) define it as a const value at the top of the script, which has the added benefit that it will have a descriptive label in code and can be later tuned easily.

Experiment with tuning that right side paddle's movement speed until it's possible to get the ball past the computer controlled paddle, but not too easy to do so consistently.

EXPECTED RESULT

At this point you can now play against the computer. The left paddle should follow the mouse, while the right paddle will automatically try to line up with the ball as quickly as it can.

The right paddle will shake when it's nearly lined up with the ball. The next step will get rid of the shaking.

tennis game step 22 resolve shaking by the a.i. paddle

Due to the ball control added earlier that affects the ball's vertical speed based on where it hits the paddle, the current artificial intelligence control is at a disadvantage for trying to center the right paddle on the ball.

Add to the code in moveComputerPaddle() so that if the ball is vertically within 35 pixels either above or below the right paddle's center it will sit still. Put another way, only move the right paddle up if the ball is more than 35 pixels above its center, and only move the right paddle down if the ball is more than 35 pixels below its center.

This deadzone gives the ball a chance to hit other parts of the paddle, returning it at steeper and more challenging angles. This alleviates the shaking effect seen in the previous step, since with a 70 pixel buffer of inaction the paddle will no longer need to alternate jumping its center back and forth across the ball's height every other frame on shallow angles.

EXPECTED RESULT

The right paddle will no longer shake during shallow incoming shots. The right paddle is now more likely to return the ball at steeper angles than it was before.

tennis game step 23

set up a maximum score to win

The game is currently endless. There's no maximum score or total reset functionality. Set a maximum number of points to play to. Playing the first to three will work well for testing. It's low enough to happen quickly, even when doing it a second or third time in a row.

Check at the top of the ball reset function whether either score is at or beyond that value. If either player has a winning score, instantly start a new round by zeroing both player scores in addition returning the ball to center field.

EXPECTED RESULT

The game can now be played until one side or the other wins. In this version, reset immediately continues play, meaning that to see who wins you just have to be paying attention to the scores when the last point gets earned.

tennis game step 24 pause the game after anyone wins

Instead of letting the game reset instantly when either player wins, as soon as either player scores the winning goal pause all gameplay and display a win message. Hide the ball and paddles at this time, leaving only the final scores showing along with centered text indicating which player ("left" or "right") won the round. To center text use the `textAlign` property of the graphics context:

```
canvasContext.textAlign = 'center';
```

Much like how setting color on the graphics context continues to apply to all later graphics calls until it's changed, adding that call once in the start of your program will center all text as long as you don't change it to 'left' or 'right' someplace else. This works perfectly fine for this particular game, since even centering the score values will look fine (slightly better, even).

Upon the next mouse click, reset both player scores. Don't do it before the click though, since the match values are needed for the win screen) and then allow the ball and paddles to be drawn again.

This change will involve a new game state in which the only graphics drawn are the black background rectangle and the foreground text. Rather than check in many different places whether one score or the other reached the limit, instead create a new variable `showingWinScreen` which can be flipped to true in a single place and then checked elsewhere.

`fillText()` uses whichever `fillStyle` color was last set for the canvas context. With the paddle and draw calls potentially being skipped, that could now be the black background color. One way to deal with this issue is to create a helper function that sets the text color right before calling `fillText()`, much like the `colorRect()` or `colorCircle()` functions set color before `rect()` and `arc()` calls.

Even though the paddles and ball are invisible their logic and actions are still being updated. The score could even continue increasing while displaying the end screen, unless you don't let the program do so. Add an extra if-condition at the top of `moveEverything()` to quit out of the function prematurely with "return;" if the round already ended.

EXPECTED RESULT

The game now presents a message indicating who won, plus the end scores for both sides. Nothing else should be on the screen when the round is over. The game remains on this end state until the user chooses to start the next match. Although this change required changes to several places, it was a valuable polish step to go from a game that was technically playable as a prototype to a more presentable experience.

tennis game step 25 let the player know how to reset

This step is going to seem oddly imbalanced with the previous one. Even though it won't take much code, it's extremely important and deserves special attention.

The win screen doesn't yet provide any instructions indicating what the player needs to do to advance. Should they try spacebar? Trust the game to restart automatically if they wait?

Don't make people guess that clicking is the right thing, and don't expect them to read your mind.

On the win screen add a second line of white centered text, near the bottom of the playfield region, that tells the player that they can click to start a new match.

EXPECTED RESULT

Identical to previous step, but with one extra line of instruction added to the win screen along the bottom.

tennis game step 2b dashed line for net to the middle

For a bit of visual decoration, add a "net" to the middle of the playfield which shows up while the ball is in play. 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 use a for-loop and a rect() call to do this. There are several other ways to make it happen.

EXPECTED RESULT

White dashed line divides the playfield during play. It should not be shown while viewing on the start or win screen.

tennis game step 27 removing update markings

Since I've been marking code changes with //// comments, I've added this one extra step to clear out the most recent of those for the corresponding example solution. I.e. this is identical to Step 26, except with the //// markings removed.

EXPECTED RESULT

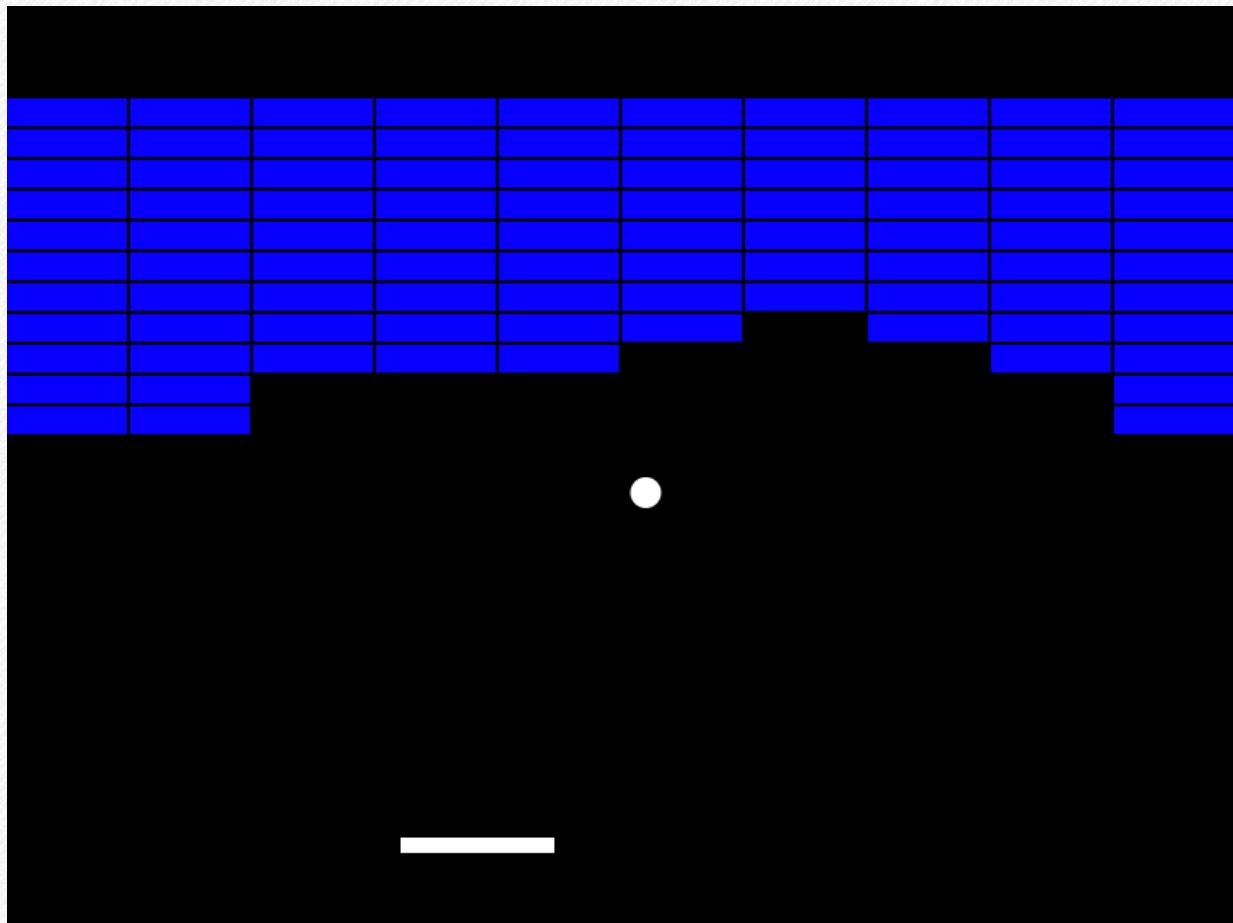
Same gameplay. Very slightly cleaned up code.

You've done it! As of now your project is in the same state as the starter code provided for use with this game's exercises in Section 2. First, I recommend putting your thoughts about Tennis Game on the back burner for a bit by taking a detour through setting up the starter code for Brick Breaker and the other games next. When you return to Tennis Game in Section 2 for its exercises you'll be seeing it with some fresh eyes and having accumulated new skills across the other project types.

GAME 2: BRICK BREAKER

LATE 1970's ARCADE-STYLE

Of special significance in this game project is the use of grid-based collision. That's a concept central to tile-map games like Racing and Warrior Legend coming up later, or in retro platformers. This same grid or tile collision is still in many indie, casual, mobile, and web games made today. It can be seem strange at first, but stick with it. The more you see it and the more that you do with it the better it will click.



brick breaker step 1 get a ball bouncing on screen

Follow steps 1-9 from Tennis Game. The main foundation of this project also begins with a ball bouncing off most edges of the screen.

EXPECTED RESULT

You should have a black rectangle playfield within which a white ball bounces off each of the four edges indefinitely.

NOTE ON PROCESS AND PRACTICE

While you *could* copy in the example code solution provided that corresponds to step 9 of Tennis Game, if you're new to game programming I encourage redoing steps 1-9. Those steps cover a lot of fundamental aspects of game motion and basic display functionality. More comfort with them will pay off later. This second time through those steps should also go much faster since you're now familiar with what's being done, how, and why. For later, larger projects, copying in code from a previous project's end state will be a standard first step.

On the other hand, if you've seen enough game programming or HTML5/JS code to be comfortable with those basics and want to keep moving forward more rapidly, copy in that step 9 example solution from Tennis Game, and I'll see you soon for Step 2 of Brick Breaker.

brick breaker step 2 adding a paddle to block the ball

Repeat steps 10-13 from Tennis Game, except instead of having the mouse slide your paddle rectangle along the left edge of the screen to block the ball from reset, have your paddle move along the bottom edge of the screen. Keep the paddle centered on the mouse's current horizontal position. Reset the ball to the screen's center whenever it passes below the bottom edge without being blocked by the paddle.

EXPECTED RESULT

Moving the mouse side to side should keep the paddle centered on the mouse's horizontal position. The paddle should be able to bounce the ball off the bottom edge of the screen. If the ball slips past the paddle along the bottom edge then the ball should instantly reset back into the playable area.

brick breaker step 3

ball control from the paddle spot hit

Implement step 20 from Tennis Game to achieve ball control based on where the ball hits the paddle. Instead of varying vertical speed based on the vertical distance from the paddle's center as in Tennis Game, here you'll vary horizontal speed based on the horizontal distance from the paddle's center. This is because the Brick Breaker paddle is rotated on its side, relative to Tennis Game's paddle.

EXPECTED RESULT

If the ball gets blocked by the leftmost part of the paddle, it should send the ball off at a shallow angle left. If the ball gets blocked nearer to the center of the paddle, it should take a proportionally more upward angle. Stopping the ball near the right edge of the paddle should cause a rightward shot.

Ball control was important for Tennis due to the competitive head-to-head nature of the game. Here, it's important since without it the player cannot hit a particular brick, except by keeping the ball in play and hoping it will eventually wind up there. That would especially become an issue when most bricks are removed and only a few remain.

In rare situations, it's now possible to get the ball stuck along the left or right edge, so long as the ball intersects the paddle very close to the edge of the screen and at a part of the paddle which promptly steers it deep enough out of bounds. The risk of this happening will be greater after the addition of click-to-begin ball release for an exercise in Section 2, after which a slightly later exercise will discuss ways to prevent that bug.

brick breaker step 4 space below the player paddle

Move the paddle up from the bottom of the playfield so about 10% of the playfield's height is below it. To do so define a constant, named PADDLE_Y, and set it to 530 or 540 for use as the paddle's top edge coordinate (540 is 90% of 600, but the paddle is 10 pixels thick and drawn from its top edge – as long as it's roughly 10% though the game will play the same).

This small adjustment will be nice for providing a second of watching the ball after being missed before the round resets. Be sure to update not only the paddle's draw position but also modify the ball-paddle collision code to take into account the paddle's height and its thickness, rather than simply treating the edge of the screen as the paddle's height.

Whereas the code for paddle collision or bottom drained ball could previously share the same edge check, now that the paddle and drain edge are different heights you'll need to split up the code into separate if-conditions.

In my provided example code, to keep things simple, I'll treat the ball as just its center point for collision testing. However if you'd like to account for the ball's thickness in the ball-paddle collision this would be a fine step to experiment with that.

EXPECTED RESULT

Same as the prior step, except now the paddle should be about an inch above the bottom of the screen.

brick breaker step 5 fix the ball hitting the paddle side

The paddle's simple vertical reflection of the ball upon collision can result in messy, unpredictable behavior if the paddle is moved quickly atop the ball while it passes partway through the paddle's path of motion. If the ball's vertical movement component isn't enough to clear the paddle completely upon reflection it keeps alternating vertical direction until it slides out horizontally.

If you're having trouble seeing this issue, experiment with increasing the paddle's thickness and then attempt to bounce the ball from the left or right side of the paddle. To overcome this glitchy behavior, have the paddle only affect the ball if the ball is currently moving downward, i.e. it has positive y speed.

EXPECTED RESULT

The ball will only ever move upward when it contacts the paddle, independent of which part of the paddle it touches. Temporarily increasing the paddle thickness, to more reliably hit with the paddle's sides, can make it easier to verify that this code fix works as expected.

brick breaker step 6

draw the bricks with nested loops

Define the following constant values for use in positioning and defining the game's upcoming brick wall:

- collision width of each brick, BRICK_W, set to 80
- collision height of each brick, BRICK_H, set to 20
- purely visual gap between bricks, BRICK_GAP, set to 2
- number of brick columns, BRICK_COLS, set to 10
- number of brick rows, BRICK_ROWS, set to 14

I specify “collision width” and “collision height” as well as “purely visual gap” above because the bricks will be, visually, only 78x18 pixels, but treated as 80x20 for collision. The gap will be subtracted from the brick dimensions purely to make it easier to see where rows and columns are divided. The ball will not, for example, be able to meaningfully fit into that gap. When, in a Section 2 exercise, images get added for the bricks that visual gap will no longer be necessary to tell one brick from the ones next to it. Without that gap here though the wall would just show up as an undifferentiated big blob of blue.

For this step also add a drawBricks() function that uses nested for-loops (one covering each row, inside one for each column) to display a blue rectangle at every brick position. Nested is just programmer jargon meaning that one loop is within the {scope} of another, as opposed to one taking place followed by the other. In this case, it means for each column of the

brick grid, check each row within that column. The next page shows an example of what that looks like for this use.

There won't be any support for removed bricks yet. Nor will this handle collisions between balls and bricks. The ball will pass right through bricks for now. All that's happening so far is that there will be a filled grid of blue bricks drawn on screen.

Call the drawBricks() function in the drawEverything() {scope}, after the background gets cleared. If the grid of bricks gets drawn earlier in the cycle than the background, it will be covered over by the filled black rectangle. Here's an example of how the brick drawing function could be written:

```
function drawBricks() {
    for(var eachCol=0; eachCol<BRICK_COLS; eachCol++) { // in each column...
        for(var eachRow=0; eachRow<BRICK_ROWS; eachRow++) { // in each row in that col
            // compute the corner in pixel coordinates of the corresponding brick
            // multiply the brick's tile coordinate by BRICK_W or BRICK_H for pixel
            var brickLeftEdgeX = eachCol * BRICK_W;
            var brickTopEdgeY = eachRow * BRICK_H;
            // draw a blue rectangle at that position, leaving a small margin for BRICK_GAP
            colorRect(brickLeftEdgeX, brickTopEdgeY,
                      BRICK_W - BRICK_GAP, BRICK_H - BRICK_GAP, 'blue' );
        } // end of for eachRow
    } // end of for eachCol
} // end of drawBricks()
```

EXPECTED RESULT

A full grid of blue bricks should now be visible. The ball will pass right through them at this point without any effect on the bricks or the ball's motion.

brick breaker step 7

support for removal of bricks

In order for a ball collision with a brick to remove the brick, you'll need a different variable for each brick that can store, check, and change for whether that particular brick is present.

Brick conditions could each be kept as a separate value either true/false, or 1 for present and 0 for gone, as variables like brick1, brick2, brick3, and so on. That would be a pain to deal with though, would involve a lot of repetitive duplicate code, and it wouldn't be very flexible to quick adjustments made to how many bricks are in the game.

Instead, an array will be used. An array is simply a group of values that can be set or retrieved based on their index, or address in the overall list. Instead of having separate brick1, brick2, and brick3 variables, you can have a single array in which brick[0], brick[1], and brick[2] store the condition of the first 3 bricks (the first element of an array is always 0).

By storing the brick states in an array you can write code that will work independent of how you may later change the total number of bricks via BRICK_COLS and BRICK_ROWS. The size of the array can even be based on the multiplication of those dimensions, ensuring there's one value for each brick:

```
var brickGrid = new Array(BRICK_COLS * BRICK_ROWS);
```

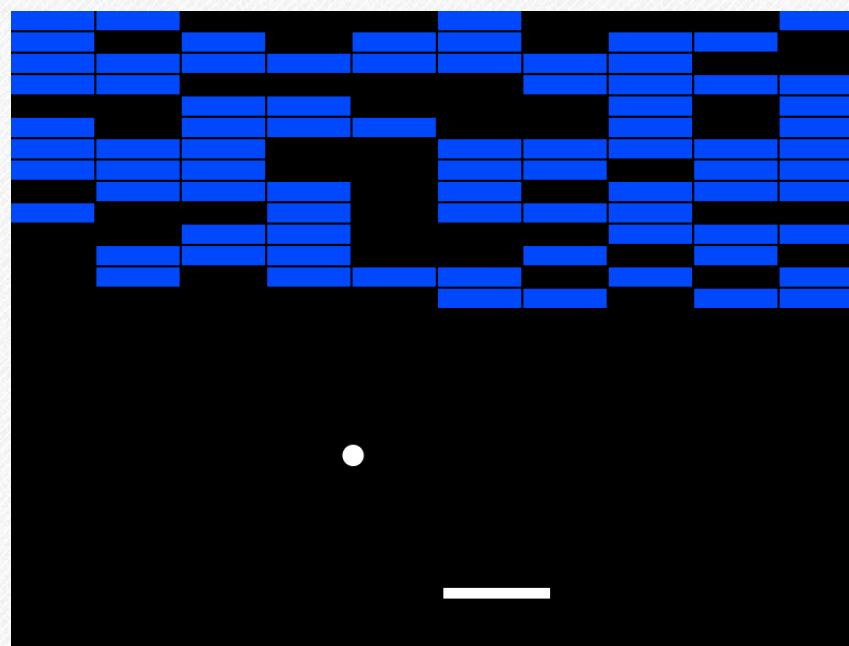
Declaring that at the top of the program ensures that you'll be able to access it from anywhere else in code, and that any changes made to it will be remembered between update frames. If it were declared within a function that wouldn't work.

Using an array will also make it possible to directly calculate which brick the ball overlaps, which is a far more efficient way to detect collisions and cause brick removal. Instead of testing the ball against all bricks, because bricks have consistent dimensions along each axis the position of the ball can be translated by simple division into a brick's index in the array.

A few helper functions can simplify dealing with the array:

resetBricks(), which you'll call from window.onload set up code, should set the value in each position randomly to either 1 or 0. For now give each brick a 50-50 even chance of being there for not when the game starts. This will help show that the newly updated drawBricks() function will hide missing bricks.

```
function resetBricks() {  
    for(var i=0; i<BRICK_COLS * BRICK_ROWS; i++) {  
        if(Math.random() < 0.5) { // only fill in half the bricks, to test display  
            brickGrid[i] = 1;  
        } else {  
            brickGrid[i] = 0;  
        }  
    }  
}
```



isBrickAtTileCoord(brickTileCol, brickTileRow) should return true if a value of 1 is found in the brickGrid array at the index at position (brickTileCol + BRICK_COLS*brickTileRow)

```
function isBrickAtTileCoord(brickTileCol, brickTileRow) {  
    var brickIndex = brickTileCol + BRICK_COLS*brickTileRow;  
    return (brickGrid[brickIndex] == 1);  
}
```

To visualize the numerical mapping from a tile position (ex. 3 bricks over, 2 bricks down) to a position in the brickGrid array, picture the indexing in brickGrid broken into rows like so:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,  
10,11,12,13,14,15,16,17,18,19,  
20,21,22,23,24,25,26,27,28,29...
```

Since the number of positions in each row is BRICK_COLS, adding one more multiple of BRICK_COLS would bump the position down to the row below it. Adding just a few to the position corresponds to other bricks in the same row. That's how the (brickTileCol + BRICK_COLS*brickTileRow) converts from a brickTileCol horizontal position and brickTileRow vertical position into a single address into the array.

In this case the 3rd brick from the left, 2nd brick from the top corresponds to array index 14. Since 0 is the leftmost column and topmost row, brickTileCol for that spot is 2, brickTileRow is 1. BRICK_COLS is 12 since there are 12 positions per row, so the expression (2+12*1) yields the brick address 14.

The code `brickGrid[brickIndex]` accesses the brick at the spot calculated, which will have either a 1 (signifying that the brick

is present) or a 0 (meaning that the brick is missing). That's what the return value is doing with (`brickGrid[brickIndex] == 1`).

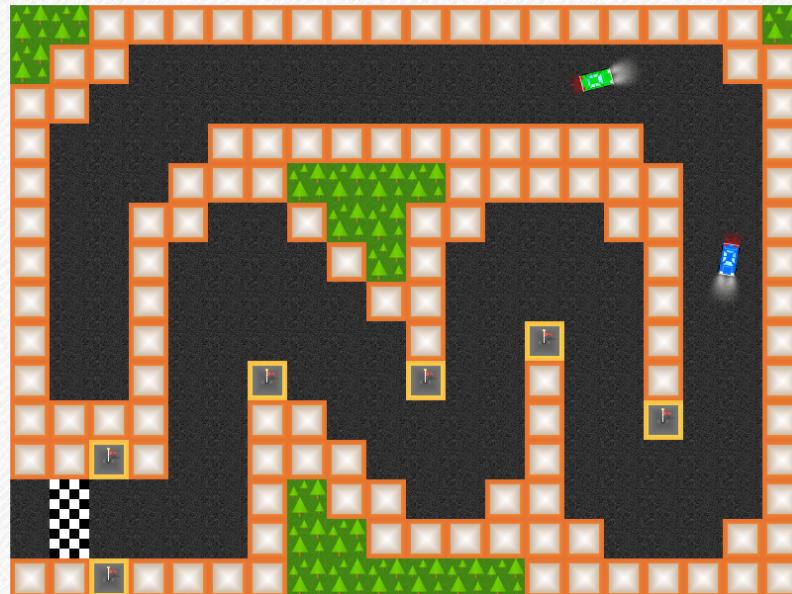
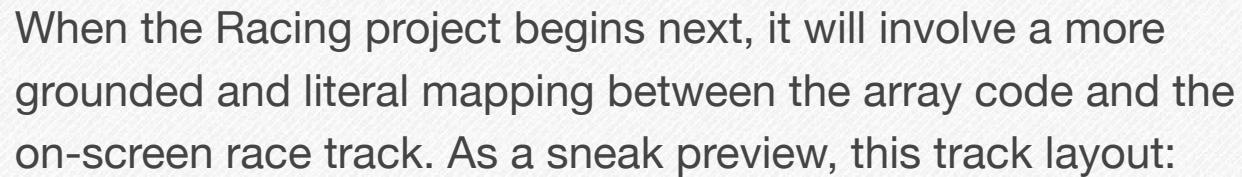
Using the `isBrickAtTileCoord()` function in the `drawBricks()` function can make it so bricks only show if their corresponding position in the array has a 1 to signify the brick's presence.

```
function drawBricks() {  
    for(var eachCol=0; eachCol<BRICK_COLS; eachCol++) { // in each column...  
        for(var eachRow=0; eachRow<BRICK_ROWS; eachRow++) { // in each row within that col  
            if( isBrickAtTileCoord(eachCol, eachRow) ) { // if the brick is present  
                var brickLeftEdgeX = eachCol * BRICK_W;  
                var brickTopEdgeY = eachRow * BRICK_H;  
                colorRect(brickLeftEdgeX, brickTopEdgeY,  
                    BRICK_W - BRICK_GAP, BRICK_H - BRICK_GAP, 'blue' );  
            }  
        } // end of for eachRow  
    } // end of for eachCol  
} // end of drawBricks()
```

EXPECTED RESULT

The ball will move through the bricks, but you should see a random number of the bricks as missing. Until you've made the last suggested change, replacing the randomization with filling the whole grid, reloading the page in your browser should randomly switch which bricks are missing each time.

Another expected result, if you're relatively new to coding, may be just a bit of confusion. Arrays can be a bit awkward to warm up to. An array is merely a list of variables, but the idea of indexes (or addresses) to different positions in the array is more abstract than each variable simply having a separate label in the code. If you're feeling a bit unsure about the whole array thing, stick with it. In the last few Brick Breaker steps you'll have more chances to work with the grid in a less abstract way.



That will get set up in the code in an array defined like so:

That's not nearly as technical as it looks. Its numbers directly correspond to the different kinds of tiles in the picture. See it?

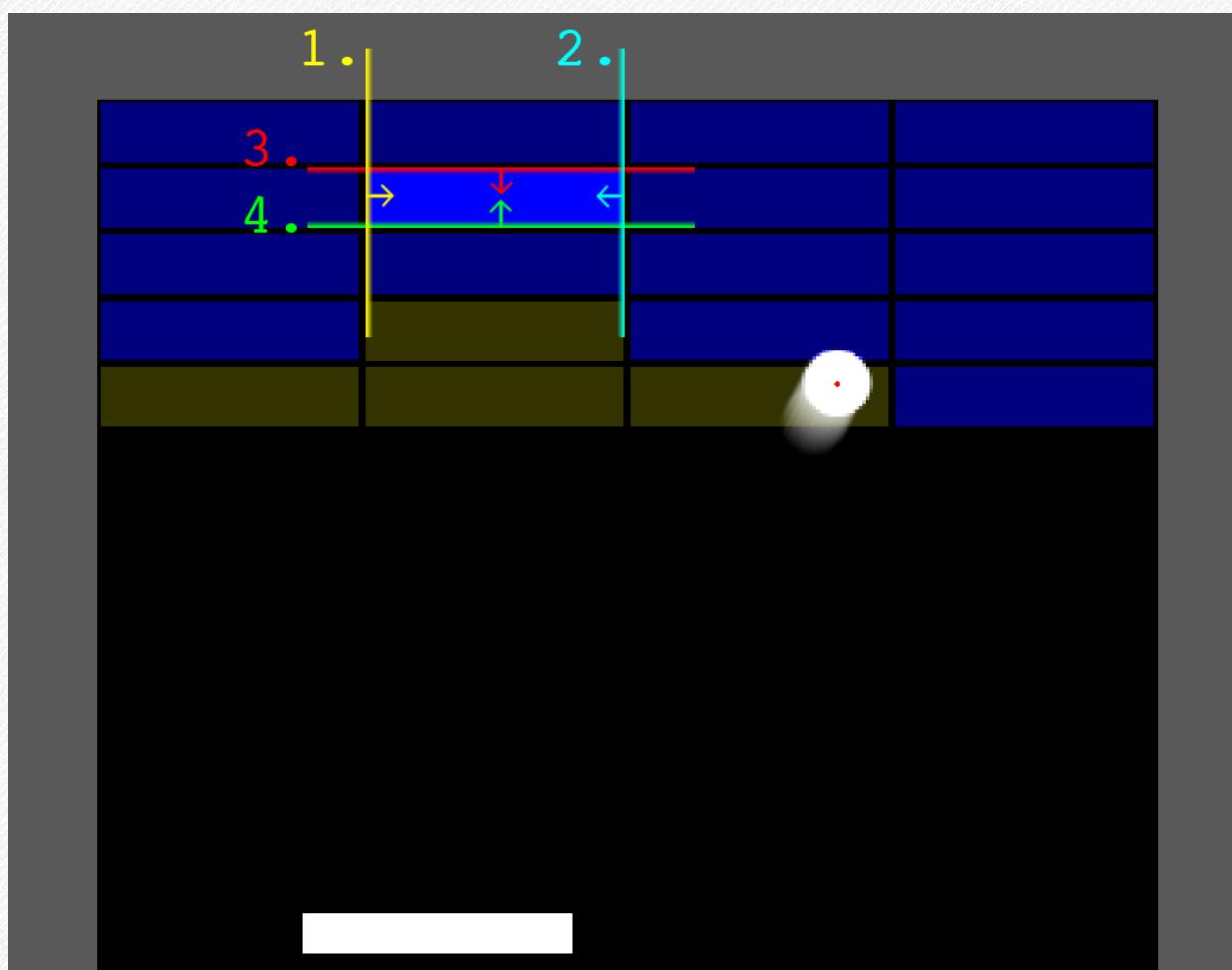
When I was getting used to using arrays for collision detection and drawing code, that type of visual helped it click for me.

brick breaker step 8

detect when the ball hits bricks

Collision between the ball and bricks could be handled by checking each brick against the ball in the same way that the paddle tested for ball overlap. Bricks are rectangles too, right? In other words for each and every brick, you could check every frame if all 4 of these tests are passed:

1. Is the ballX right of this brick's left edge? And...
2. Is the ballX left of this brick's right edge? And...
3. Is the ballY below this brick's top edge? And...
4. Is the ballY above this brick's bottom edge?



We *could* do that. *Could*. But it's not actually necessary to!

There are a lot of bricks on the screen. What if instead of checking every brick against the ball each frame you could instead calculate directly which brick the ball is over, if any?

The trick here relies on the fact that the tiles adhere to such a systematic, regular pattern. Because every bricks has the same size, their corners are spaced in multiples, or regular intervals. This is the same reason why the bricks could be drawn by multiplication with that nested double for-loop, instead of each brick separately storing its own position:

```
function drawBricks() { // old version, without check if brick missing, don't copy
    for(var eachCol=0; eachCol<BRICK_COLS; eachCol++) {
        for(var eachRow=0; eachRow<BRICK_ROWS; eachRow++) {
            var brickLeftEdgeX = eachCol * BRICK_W;
            var brickTopEdgeY = eachRow * BRICK_H;
            colorRect(brickLeftEdgeX, brickTopEdgeY,
                      BRICK_W - BRICK_GAP, BRICK_H - BRICK_GAP, 'blue' );
        }
    }
}
```

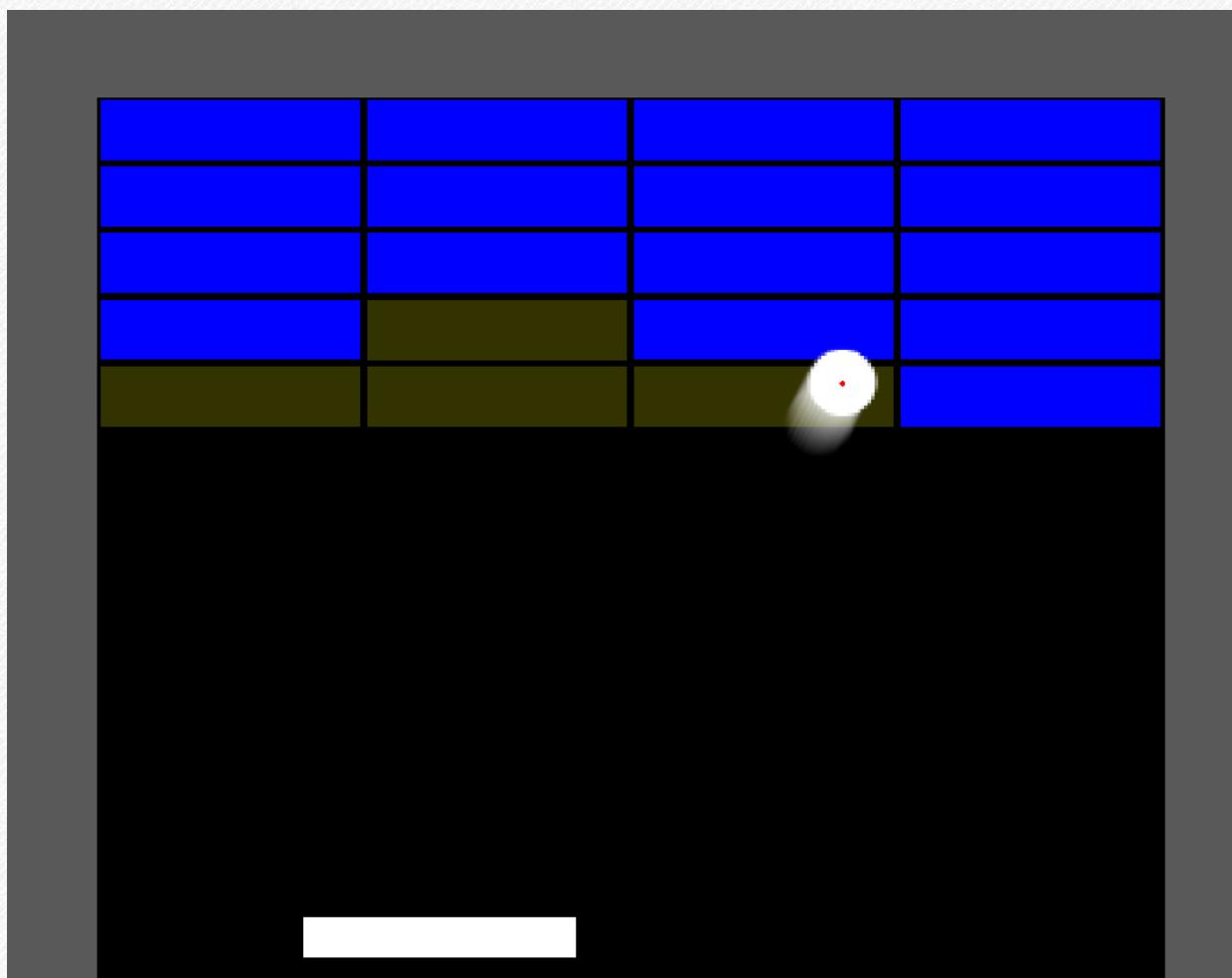
Multiplication there was used to translate from brick positions (column and row, ex. 3 bricks over and 2 bricks down) to pixel spot for draw code (corner at $3 \times \text{BRICK_W} = 3 \times 80 = 240$ pixels from the left, $2 \times \text{BRICK_H} = 2 \times 20 = 40$ pixels from the top).

We can use division for the opposite effect: to translate from a pixel coordinate – say, the ball's current position at ballX, ballY – to the brick column and row, or how many bricks over and down the brick is that's under the ball.

This method of collision detection by division to find where a point is within a grid is used in virtually all tile-based games. It's the exact same technique that will be used in the Racing

and Warrior Legend games covered later in this book. It's also at the heart of classic side-scrolling platformers, digital board games like chess or checkers, and even many puzzle games that operate on a regular square or rectangle tiled grid.

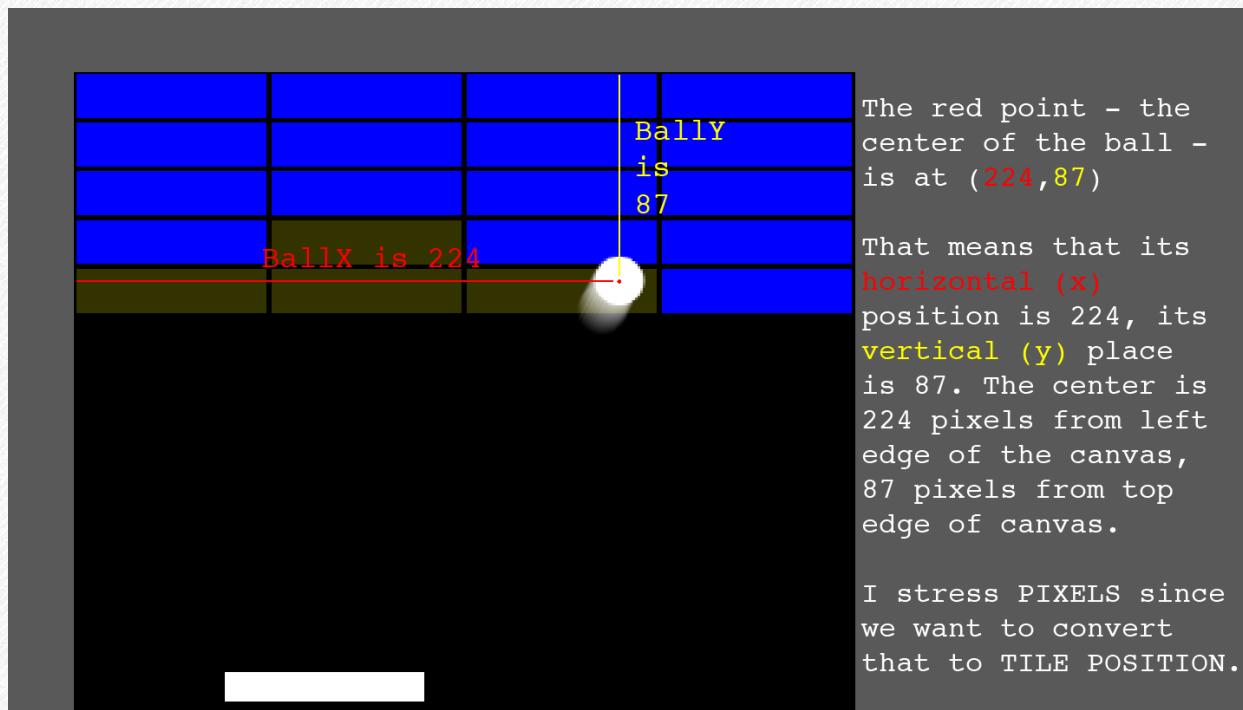
Because this concept is so fundamental to so many types of game experiences, let's take a moment to step through an example of how it works in an example case. For simplicity, let's say the playfield is only 4 brick columns wide. Bricks are still 80 pixels wide (BRICK_W) and 20 pixels tall (BRICK_H):



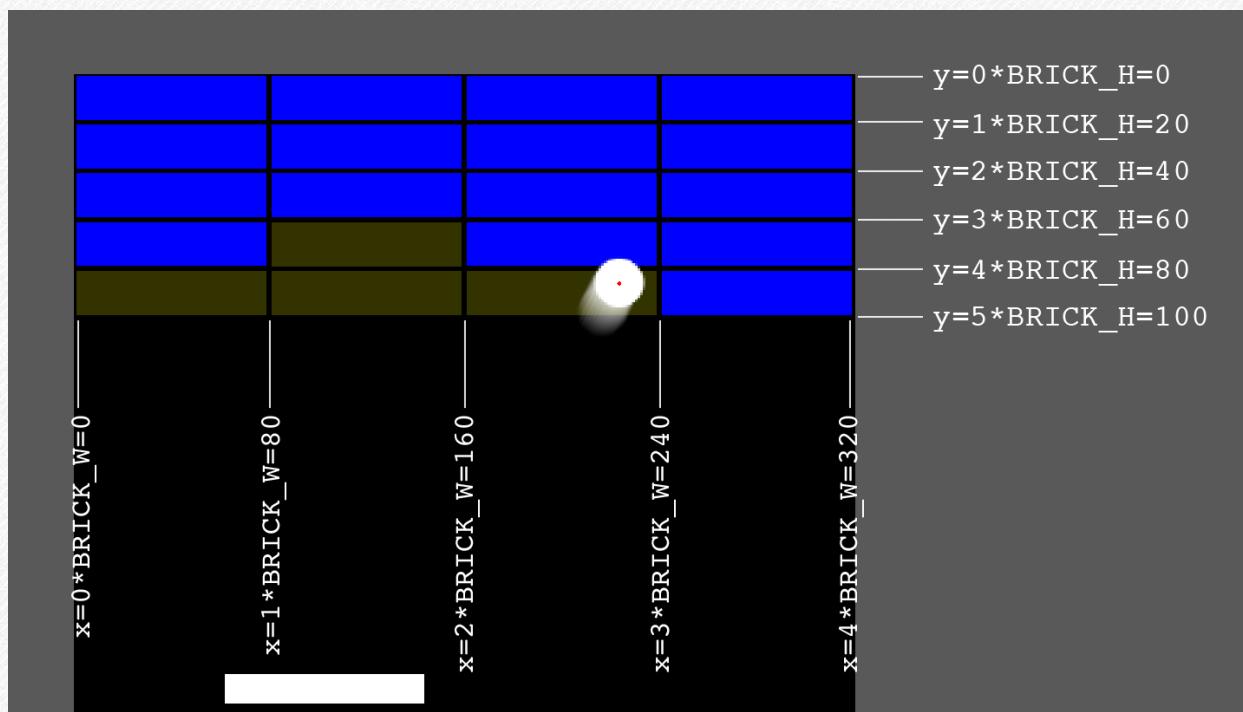
The yellow darker bricks are just to show the space where the bricks have been removed. Put another way, those show where their corresponding place in the brickGrid array is set to

0 instead of 1. Naturally in the game they'd show up as pure black, but I've drawn them here just to show their spacing.

For this example ballX is 224 and ballY is 97. To be extra clear:



Check out these distance markers for each row and column:



The first step is to translate from a pixel coordinate to the brick's tile position as column and row. Here, the ball is in brick tile column 2 ("2 whole bricks from the left" – the leftmost is column 0) and row 4 ("4 whole bricks from the top"):

Tiles C,R	Column 0	Column 1	Column 2	Column 3
Row 0	Tile 0,0	Tile 1,0	Tile 2,0	Tile 3,0
Row 1	Tile 0,1	Tile 1,1	Tile 2,1	Tile 3,1
Row 2	Tile 0,2	Tile 1,2	Tile 2,2	Tile 3,2
Row 3	Tile 0,3	Tile 1,3	Tile 2,3	Tile 3,3
Row 4	Tile 0,4	Tile 1,4	Tile 2,4	Tile 3,4

Remember: that illustration is only here to show the answer that we want, and why. It's not showing how to determine the brick position (column 2, row 4) from pixel (x 224, y 87).

Getting there is done by ordinary division. Divide the ball's x position (227) by the brick width (80) and you'll get 2.8375. Spatially that can be thought of as "2 whole brick widths from the left, 83.75% of the way toward the third brick column." For checking whether the brick is there or not we don't care about the 0.8375 part – a Math.floor() call can round it down to 2. That's the column of the brick under the ball. 2 from the left.

Doing the same for the vertical axis, the ball's y is 87 pixels. Bricks are 20 pixels tall. $87/20$ is 4.35, or 4 whole brick row heights down and 35% of the way through the 5th. Using Math.floor() to round that down, the row under the ball is 4.

If the ball is in the left-most column or the very top row, the fraction resulting from the division will round down to 0, as in

the “zeroth” column or “zeroth” row. That’s not very natural, but it is consistent with arrays using 0 as the first position.

Speaking of arrays: to check or set a brick’s status the column and row has to be translated into a position in the array.

The brickGrid linear array indexes map onto the bricks like so:

Index 0	Index 1	Index 2	Index 3
Index 4	Index 5	Index 6	Index 7
Index 8	Index 9	Index 10	Index 11
Index 12	Index 13	Index 14	Index 15
Index 16	Index 17	Index 18	Index 19

Array shown: {1,1,1,1, 1,1,1,1, 1,1,1,1, 1,0,1,1, 0,0,0,1}

Remember this helper function from the previous step?

```
function isBrickAtTileCoord(brickTileCol, brickTileRow) {
  var brickIndex = brickTileCol + BRICK_COLS*brickTileRow;
  return (brickGrid[brickIndex] == 1);
}
```

Check this out, in comparison to the two prior illustrations:

$0+0*4=0$	$1+0*4=1$	$2+0*4=2$	$3+0*4=3$
$0+1*4=4$	$1+1*4=5$	$2+1*4=6$	$3+1*4=7$
$0+2*4=8$	$1+2*4=9$	$2+2*4=10$	$3+2*4=11$
$0+3*4=12$	$1+3*4=13$	$2+3*4=14$	$3+3*4=15$
$0+4*4=16$	$1+4*4=17$	$2+4*4=18$	$3+4*4=19$

The *4 in each expression is multiplying times the number of columns. The other three numbers in each expression, from left to right, is the brick column, brick row, and array index.

Before handling ball reflection, just getting the ball to wipe out bricks under it as it moves will be a quick way to test whether it's correctly detecting which brick it's currently moving over.

Write a function, `removeBrickAtPixelCoord(pixelX,pixelY)`, to be called near the end of `moveEverything()` with `ballX`, `ballY` as its arguments. As illustrated earlier in this section, it needs to:

- **Divide the ball's pixel position to find the brick's row and column.** Each multiple of brick width to the right signifies one more column over. Each multiple of the brick's height signifies one more row down. The results will likely have decimal parts, so you'll need to use `Math.floor()` to round those down to the nearest whole number. We need to know, for example, whether the brick under the ball is in column 2 or 3. If the division yields column "2.7" that means it's 70% toward the right of column 2 – but that simply means that it's firmly within column 2, nonetheless.
- **Check to make sure that the tile coordinates are in range.** If either brick tile coordinate is either non-negative or greater than or equal to the maximum number of columns and rows, return from the function to avoid an error from checking an illegal array index. This will mainly apply if the ball is below where the lowest brick row is on the screen, but could also momentarily apply if the ball is bouncing off the left, top, or

right edge of the canvas, so that its center point is technically out of bounds for one frame.

- **Translate the column row into the brickGrid array index.**

The expression in isBrickAtTileCoord() is what's needed, but rather than copy and paste it, move that expression into a new helper function named brickTileToIndex(tileCol, tileRow) to handle the addition and multiplication in one place, returning the result of that calculation. Use that same function in both isBrickAtTileCoord() and removeBrickAtPixelCoord().

- **Remove the brick by setting the array at that index to 0.**

This will cause the brick to not appear in later draw cycles.

```
function removeBrickAtPixelCoord(pixelX,pixelY) {  
    var tileCol = pixelX / BRICK_W;  
    var tileRow = pixelY / BRICK_H;  
  
    // we'll use Math.floor to round down to the nearest whole number  
    tileCol = Math.floor( tileCol );  
    tileRow = Math.floor( tileRow );  
  
    // first check whether the ball is within any part of the brick wall  
    if(tileCol < 0 || tileCol >= BRICK_COLS ||  
        tileRow < 0 || tileRow >= BRICK_ROWS) {  
        return; // bail out of function to avoid illegal array position usage  
    }  
  
    var brickIndex = brickTileToIndex(tileCol, tileRow);  
  
    brickGrid[brickIndex] = 0;  
}
```

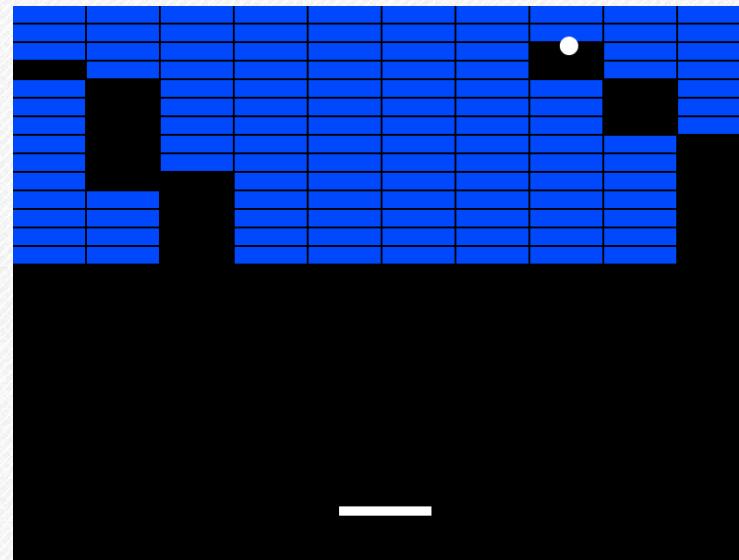
That function uses a new helper named brickTileToIndex() to simplify the translation from a brick's horizontal and vertical position in the grid into the index number for the array. It's really just a nicer way to write the expression that was in the isBrickAtTileCoord() function. It can be used there, too:

```

function brickTileToIndex(tileCol, tileRow) {
  return (tileCol + BRICK_COLS*tileRow);
}

function isBrickAtTileCoord(brickTileCol, brickTileRow) {
  var brickIndex = brickTileToIndex(brickTileCol, brickTileRow);
  return (brickGrid[brickIndex] == 1);
}

```



Rounds are starting with 50% of the bricks missing. Update `resetBricks()` so that instead of randomly leaving out half the bricks, it will set all positions to 1 so bricks begin filled in.

```

function resetBricks() {
  for(var i=0; i<BRICK_COLS * BRICK_ROWS; i++) {
    brickGrid[i] = 1;
  }
}

```

EXPECTED RESULT

The ball should move smoothly through bricks. As the ball moves through them they should immediately vanish. The ball's motion will not yet be affected by the bricks.

The brick wall should be entirely filled in when the game starts. When all bricks are cleared nothing special will happen yet, the ball will just remain in play with nothing to hit.

brick breaker step 9 bounce the ball when it hits bricks

Next have the ball reflect vertically upon collision with a brick.

To do this multiply its vertical speed by -1 whenever the ball overlaps a grid position where there a block is still present.

Rename the function named `removeBrickAtPixelCoord()` to `checkForAndRemoveBrickAtPixelCoord()`, to fit that it will now check for the brick. Return true if a brick is present in that position. Use that to signal in `moveEverything()` a ball bounce when the ball overlaps any brick.

(Confession: we'll soon go back to letting the collision function directly affect ball movement changes. I wanted to show this technique though because it's much closer to how the tile grid will be used for Racing and Warrior Legend collisions later.)

Additionally, now that the bricks bounce the ball it will be an issue that the ball starts in the top-left, inside the wall. Call `ballReset()` near the bottom of `window.onload` to fix that.

EXPECTED RESULT

The ball should now start in the center of the screen. Any brick it contacts will cause its direction to flip, removing that brick. Bricks can be cleared but will not come back, even if all are removed.

The ball will not, as of this step, bounce horizontally off the sides of bricks. Upon overlapping a brick from the side the ball's *vertical* direction to flip and the brick will be removed.

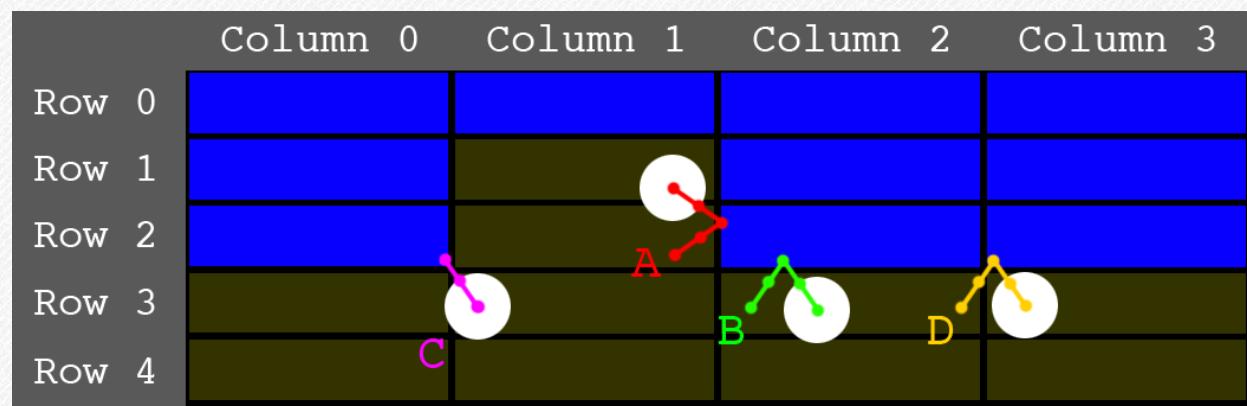
brick breaker step 10 handle collision to brick sides pt. 1

When the ball hits any brick from any heading it currently always rebounds vertically. Put another way, it can't yet tell if it hit the side of a brick and respond appropriately.

To make it easier to hit brick sides while testing this feature, temporarily double BRICK_H to 40 and halve BRICK_ROWS to 7. With fewer, taller bricks you'll hit the sides more often. For the illustrations I'll stick to the normal proportions though. The math and specific case conditions will work out the same.

The method that I'll present here is only one of many different ways that the distinction can be made as to which side of a brick the ball hit. The approach that I'll explain is on the simple side but behaves right in most cases. It's the same technique that I've often used for my own similar game projects.

Consider 4 different ball bounces (balls are at end positions):



Situation A depicts the ball bouncing off the side of a brick.

Situation B shows the ball bouncing off the bottom of a brick.

Situation C is the ball hitting a brick corner and rebounding.

Situation D shows a special edge case. More on that in a sec.

Each situation's line has little circles to show the ball's center in consecutive frames. For collision only the ball's center point is considered. Remember that the ball jumps a few pixels each frame, 30 frames per second. It's not really moving smoothly.

The main trick here is to pay attention to whether the ball, at the time of collision, has in the past movement step changed rows or changed columns. The algorithm begins like this:

1. Is the ball on a brick that hasn't been removed? If so...
2. Find the ball's previous position by subtracting its speed.
3. Compute the tile row and column for that previous position.
4. Compare the previous row and column to the current ones.

Notice how in scenario A the ball has moved from Column 1 to Column 2 for the frame when it hits the brick. To have changed columns at the moment of collision the ball must have entered from the brick's side, so flip its horizontal speed.

In scenario B the ball has moved from Row 3 up to Row 2 at its moment of impact. Since the collision happened from a change in row, it must have hit the top or bottom of a brick. Either way: flip the ball's vertical speed.

Scenario C shows a tricky corner case. The ball crossed a grid intersection diagonally at the time of collision. The frame prior to the collision, the ball's center was in a different row and in a different column. When both change at once, reverse both the horizontal and vertical speeds, so it bounces off the corner.

The code for making this distinction is surprisingly compact:

```
function breakAndBounceOffBrickAtPixelCoord(pixelX,pixelY) { // updated the name!
  var tileCol = pixelX / BRICK_W;
  var tileRow = pixelY / BRICK_H;

  // we'll use Math.floor to round down to the nearest whole number
  tileCol = Math.floor( tileCol );
  tileRow = Math.floor( tileRow );

  // first check whether the ball is within any part of the brick wall
  if(tileCol < 0 || tileCol >= BRICK_COLS ||
    tileRow < 0 || tileRow >= BRICK_ROWS) {
    return false; // bail out of function to avoid illegal array position usage
  }

  var brickIndex = brickTileToIndex(tileCol, tileRow);

  if(brickGrid[brickIndex] == 1) {

    // ok, so we know we overlap a brick now.
    // let's backtrack to see whether we changed rows or cols on way in
    var prevBallX = ballX-ballSpeedX;
    var prevBallY = ballY-ballSpeedY;
    var prevTileCol = Math.floor(prevBallX / BRICK_W);
    var prevTileRow = Math.floor(prevBallY / BRICK_H);

    if(prevTileCol != tileCol) { // must have come in horizontally
      ballSpeedX *= -1;
    }

    if(prevTileRow != tileRow) { // must have come in vertically
      ballSpeedY *= -1;
    }

    brickGrid[brickIndex] = 0; // remove the brick that got hit
  }
}
```

Also replace the if statement where the function was called...

```
if( checkForAndRemoveBrickAtPixelCoord(ballX, ballY) ) {
  ballSpeedY *= -1;
}
```

...with one line since the function now handles ball motion:

```
breakAndBounceOffBrickAtPixelCoord(ballX, ballY);
```

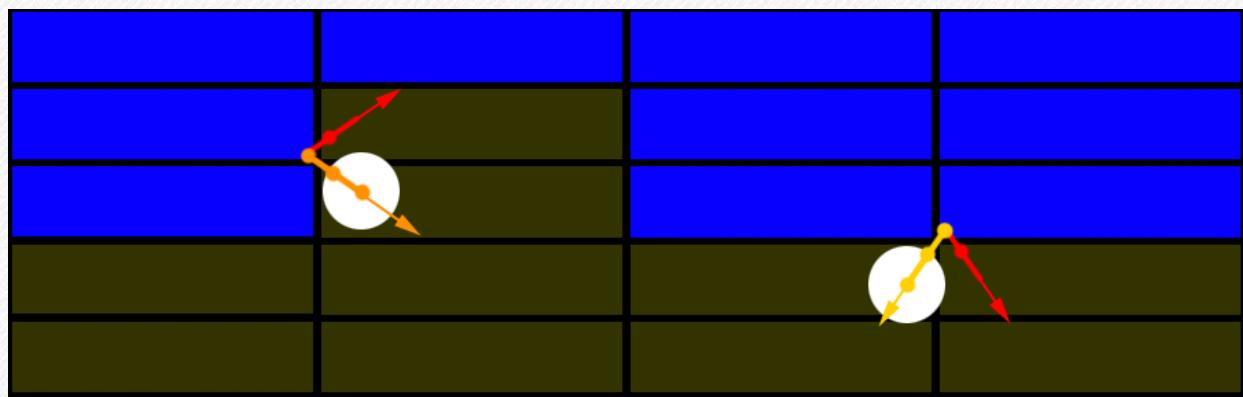
If that's all we did, it would still be an improvement over the previous behavior, which only flipped the ball's vertical motion. However that C condition, the diagonal case, left as-is would lead to a fair amount of distractingly wrong-looking outcomes, such as scenario D. In both cases, the ball crosses diagonally as it goes from open space into a brick tile.

However, in case D the ball in case is hitting the bottom of a long, flat edge. The upright side is blocked by another brick. There's no way that the ball should flip its horizontal direction. Clearly in this case a vertical bounce is what ought to happen.

We'll next improve the collision test to handle that case better.

EXPECTED RESULT

The ball will now bounce off the sides of bricks, too. However if the ball hits a crack intersection between bricks it can move unexpectedly, reversing completely when it really shouldn't. Just for this one step the bricks should be twice as tall and half as many vertically for easier testing against side collisions.

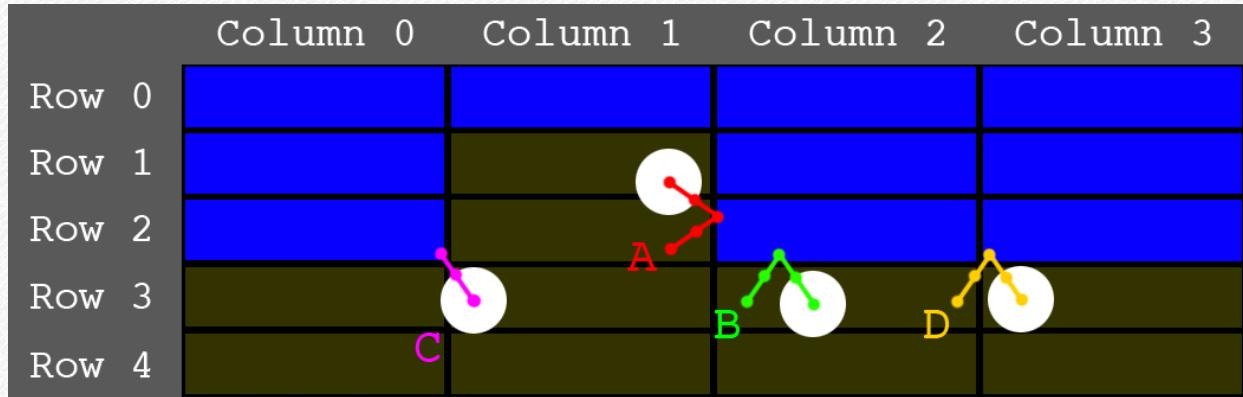


The current problematic behavior happens when the ball crosses a diagonal up against a solid row or column of adjacent bricks. As-is, the ball will now return backward in these two cases. It would look more natural if the ball's motion instead followed the red arrows after these hits. To do so we'll need to look not only at the brick hit and the tile the ball was previously on, but also at the nearby brick positions that might be covering either edge of the brick being struck.

brick breaker step II

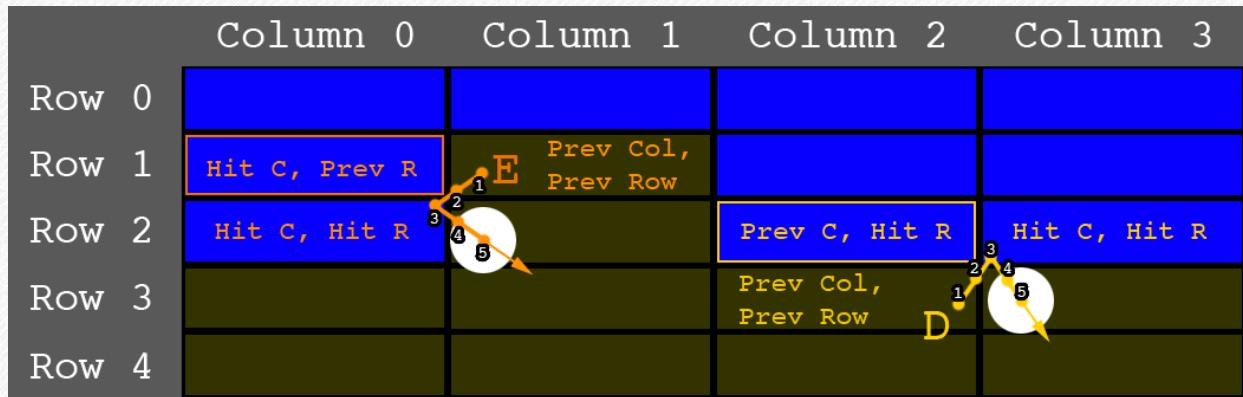
handle collision to brick sides pt. 2

In the previous step, ball movement was affected by brick collision in a way that worked fine for cases A, B, and C here:



(This is the same illustration from the previous chapter, copied here for reference and comparison.)

Since D is the weird failing case, it will help to pick it apart in finer detail. Consider too another collision scenario, E, which is like D in that it crosses a diagonal at the moment of collision. In case E though a horizontal movement reflection will be the one that looks more correct given where bricks are nearby:



If we don't do something special to detect and handle these kinds of corner crossing cases, then they'll be treated the same as scenario C, reversing both horizontal and vertical

movement. Do you see why a ball in either situation D or E completely reversing its direction would look wrong?

The ball's center positions over time are marked 1-5, with 5 being the last frame. The ball-to-brick collision for D and E is at time step 3, coming from its position at time step 2.

Hit C and Hit R in that illustration show the column and row of the collision at time step 3. Prev C and Prev R are the ball's column and row prior to collision at time step 2 (equivalently, take the ball's position at time of collision and subtract its speed along each axis, since that's where it was one frame ago, and find the tile for that pixel position).

The new magic is what happens when those tile coordinates get mixed. It's not obvious from the numbers that anything meaningful is at the tile position described by (Hit C, Prev R) or the tile at (Prev C, Hit R). On the illustration I've highlighted these mixtures with outlines. These coordinates point to the adjacent corner bricks from the direction of the incoming ball. By using those mixed tile positions you can check whether a brick is nearby blocking a hit brick's side or top/bottom edge.

In D, for example, at time step 3 it hits the brick at (3,2). Subtracting its current speed shows that at time step 2 it was in tile (2,3). Mixing those to get (2,2) we can determine that a brick was covering the left side of the brick we struck, and so the ball should not reflect horizontally. It cannot have come from the left. The left side of the brick hit is not exposed.

Mixing hit and previous tile positions in the other order, (3,3), and checking the tile grid for whether a brick is there, it's an

open position. Since the bottom surface of the struck brick was exposed it will look fine for the ball to bounce vertically.

Switching over to example E, it also first detects a collision at time step 3, when it's over tile position (0,2). At time step 2 the ball was last over tile position (1,1). Crossing those to check if the ball could have come in vertically we get position (0,1), where there's a brick. The top is blocked, so no vertical motion reflection should occur. Mixing the tile coordinates the other way yields (1,2) which is wide open, so the side isn't blocked. The ball's motion can be reflected horizontally.

The last case to handle is how the ball should behave if both of the adjacent sides are blocked by other bricks. The only time that can be the case is if we hit an inside corner armpit, so in that case reverse the ball's direction completely by flipping both the ball's horizontal and vertical speed values.

Here's how it all comes together in code:

```
function breakAndBounceOffBrickAtPixelCoord(pixelX,pixelY) { // updated the name!
    var tileCol = pixelX / BRICK_W;
    var tileRow = pixelY / BRICK_H;

    // we'll use Math.floor to round down to the nearest whole number
    tileCol = Math.floor( tileCol );
    tileRow = Math.floor( tileRow );

    // first check whether the ball is within any part of the brick wall
    if(tileCol < 0 || tileCol >= BRICK_COLS ||
       tileRow < 0 || tileRow >= BRICK_ROWS) {
        return false; // bail out of function to avoid illegal array position usage
    }

    var brickIndex = brickTileToIndex(tileCol, tileRow);
```

```

if(brickGrid[brickIndex] == 1) {
    // ok, so we know we overlap a brick now.
    // let's backtrack to see whether we changed rows or cols on way in
    var prevBallX = ballX-ballSpeedX;
    var prevBallY = ballY-ballSpeedY;
    var prevTileCol = Math.floor(prevBallX / BRICK_W);
    var prevTileRow = Math.floor(prevBallY / BRICK_H);

    var bothTestsFailed = true;

    if(prevTileCol != tileCol) { // must have come in horizontally
        var adjacentBrickIndex = brickTileToIndex(prevTileCol, tileRow);
        // make sure the side we want to reflect off isn't blocked!
        if(brickGrid[adjacentBrickIndex] != 1) {
            ballSpeedX *= -1;
            bothTestsFailed = false;
        }
    }

    if(prevTileRow != tileRow) { // must have come in vertically
        var adjacentBrickIndex = brickTileToIndex(tileCol, prevTileRow);
        // make sure the side we want to reflect off isn't blocked!
        if(brickGrid[adjacentBrickIndex] != 1) {
            ballSpeedY *= -1;
            bothTestsFailed = false;
        }
    }

    // we hit an "armpit" on the inside corner, flip both to avoid going into it
    if(bothTestsFailed) {
        ballSpeedX *= -1;
        ballSpeedY *= -1;
    }

    brickGrid[brickIndex] = 0; // remove the brick that got hit
}
}

```

Change the BRICK_H and BRICK_ROWS back to 20 and 14 respectively. That will increase the frequency of corner cases.

EXPECTED RESULT

Gaps between bricks will no longer send the ball backwards. Bricks should now be back to their shorter, normal height.

(Historically, the first games in this genre did not distinguish ball collision with the side of bricks! Gameplay ramifications of that design decision will be explored in a Section 2 exercise.)

brick breaker step 12 reset bricks once the last is gone

Once the player clears the last brick there's nothing left to do. Add code to refill the bricks if none remain when the ball next touches the paddle. (Waiting until that moment ensures the ball won't be inside the wall when it resets.)

There are multiple ways to test whether any bricks are left. You could scan the entire brick grid checking each spot for a non-zero value. A slightly more efficient way is to keep a counter for how many bricks the level has. Set up the counter where the brick wall gets filled in, decrease it by one with each brick removed, then reset the grid (and the counter) if the counter is zero when the ball next touches the paddle.

Testing this change could chew up a lot of time playing. To speed up development add a few lines to the mouse move handling code to immediately update the ball's position to the current mouse position. Now clearing bricks will be way faster.

EXPECTED RESULT

Moving the mouse should reposition the ball. Do this to wipe away all bricks, then touch the ball to the paddle... when the ball is moving downward, otherwise bounce it off the ceiling first – remember that, due to an earlier step, the ball ignores the paddle while it moves upward! All bricks will then reappear if the new code works.

It may be worth testing field reset twice in a row within a single play session to ensure nothing about your approach falls apart as a result of the reset event.

brick breaker step 13 remove 3 brick rows and ball cheat

The bricks going all the way up to the top of the screen makes it too hard for the player to get a ball stuck up there behind the bricks. Half the fun in a game like this one is trying to bury the ball behind the bricks without carving too large a hole, causing it to run amok bouncing around bashing bricks before it finds its way back to the paddle.

An easy way to create a margin at the top of the screen is simply to set the first 3 rows of bricks to 0, instead of 1, when the bricks get reset. In other words, have them start missing. Or if you'd find it conceptually easier to handle this another way, immediately after filling the whole grid run code that removes all bricks from the first three rows.

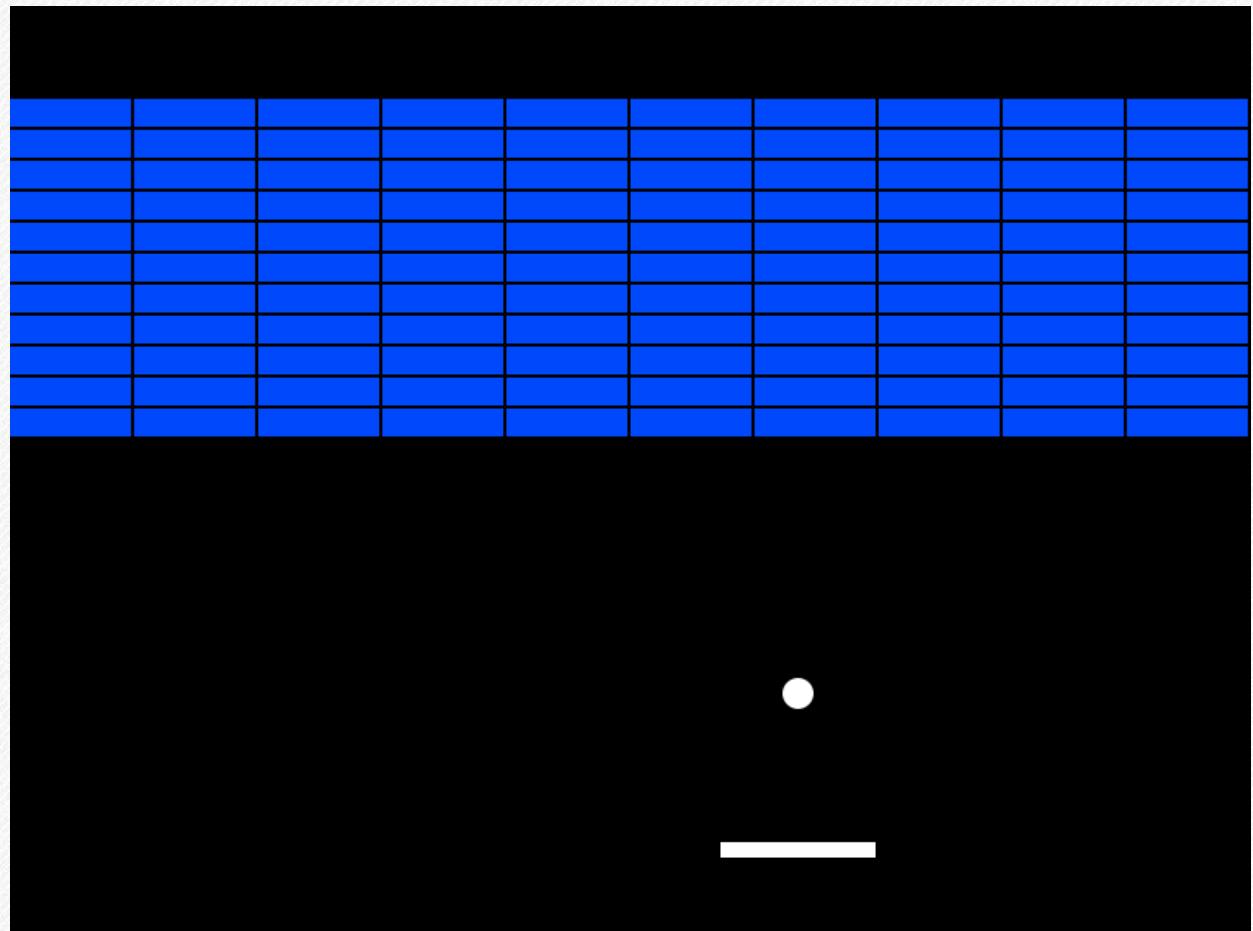
Be sure to do this in a way that preserves the validity of the bricks hit counter or the game may not reset when the last brick gets cleared. Use the mouse movement ball control cheat to verify that clearing all bricks still causes the bricks to all reset.

Once that cheat has been used to confirm that the new empty back row margin hasn't messed up the brick count or wall reset functionality, remove or comment out the lines that move the ball when the mouse moves.

EXPECTED RESULT

The top 3 rows should begin empty, having no bricks in them.

Moving the mouse should no longer reposition the ball.



brick breaker step 14 erase all update markings

The example solution for this step is the same as the previous step, except with all update comment marks //// erased.

EXPECTED RESULT

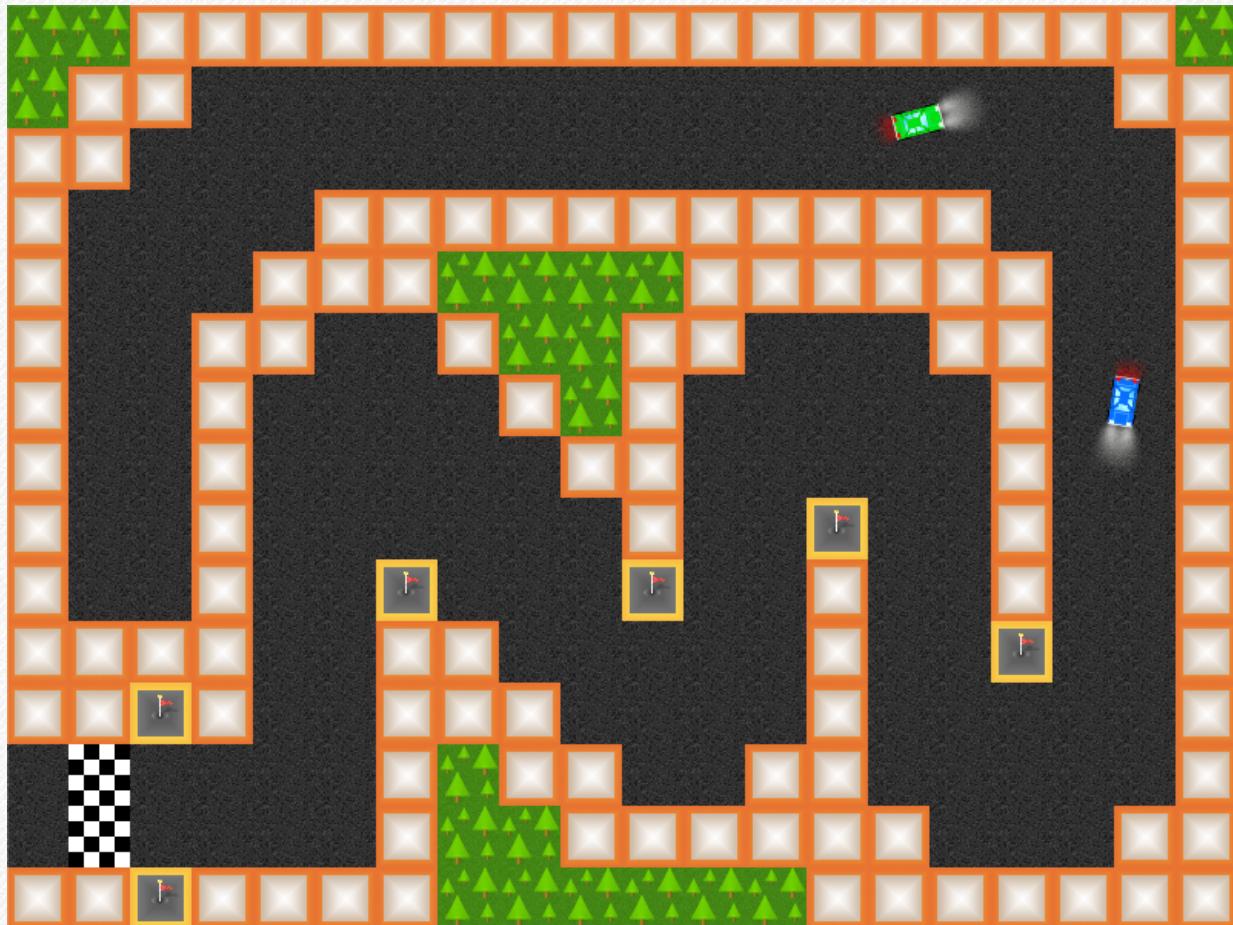
The game should play just as it did in the previous step.

As before with the Tennis Game, I suggest working through building the starter code for Racing and the later game types included before turning to the exercises for Brick Breaker. This way you'll get practice with and exposure to more gameplay concepts that will be useful for adding features to this game in Section 2.

GAME 3: RACING

EARLY 1980's CONSOLE-STYLE

This project is quite a leap forward from the 1970's designs. To make a project this sophisticated you'll need to organize the code in several ways to keep it manageable. Many of these same concepts will also be essential in the next project, Warrior Legend. Don't be put off by this chapter's length. It's long, but in a way you're working on both games at once!



racing step I

copy in brick breaker to start from

Because this project will involve grid-based collision for the cars, begin by duplicating the final core step of Brick Breaker. In the steps ahead, you'll be removing functionality that this game doesn't need, followed by renaming any variables and functions that this game will repurpose. You'll have a partially working game core to build upon from there.

Start by making an identical copy of Brick Breaker as it is after completing the final step. If you didn't do that yet and don't wish to go back to it, my example solution for this step is a copy of the last solution step from that project.

EXPECTED RESULT

Brick Breaker's core functionality with no changes yet.

Note: doing this to start a project is not "cheating." Building off past code projects is an incredibly common thing that many developers do when starting a new project. Pick a previous project you've made that has overlap with what you need, tear it down to keep only what's relevant, update code wording to match its new use, and build from there. I do it all the time. Even if a new project has no overlap, I still prefer to begin with at least a simple project that displays images and takes input.

The more complete a project is with fancy program-specific code, the more work will be needed to carve it down to a reusable foundation. That's one reason why I'm covering core code for each game before going further on any of them.

racing step 2 removal of player paddle

You'll no longer need the paddle at all. Remove any variables and functionality from the code related to the paddle.

Remove any 'mousemove' related code since the racing game will not use the mouse.

Note, however, that you'll be keeping the bricks since those will become the track walls. The ball will remain too, to be repurposed as player one's car.

While you're going through the code, rename any reference in code to BRICK to TRACK, Brick to Track, and brick to track. In other words take care to preserve capitalization during the rename. A find-and-replace-all functionality with ignore capitalization turned off (often the default!) can quickly make a mess of your code if you're not careful.

Because these occurrences happen on so many lines of the code, I won't mark these variable renaming changes in the example code with the /// per line as I do with the other modifications being made.

EXPECTED RESULT

Same as before, except now with the paddle removed. Since the paddle was the only interaction, the current state is not interactive. The full brick wall should be on the top half of the playfield, while the ball will repeatedly reset to the middle of the screen then fall right off the bottom to then reset again.

racing step 3 fill the screen with bricks & big grid

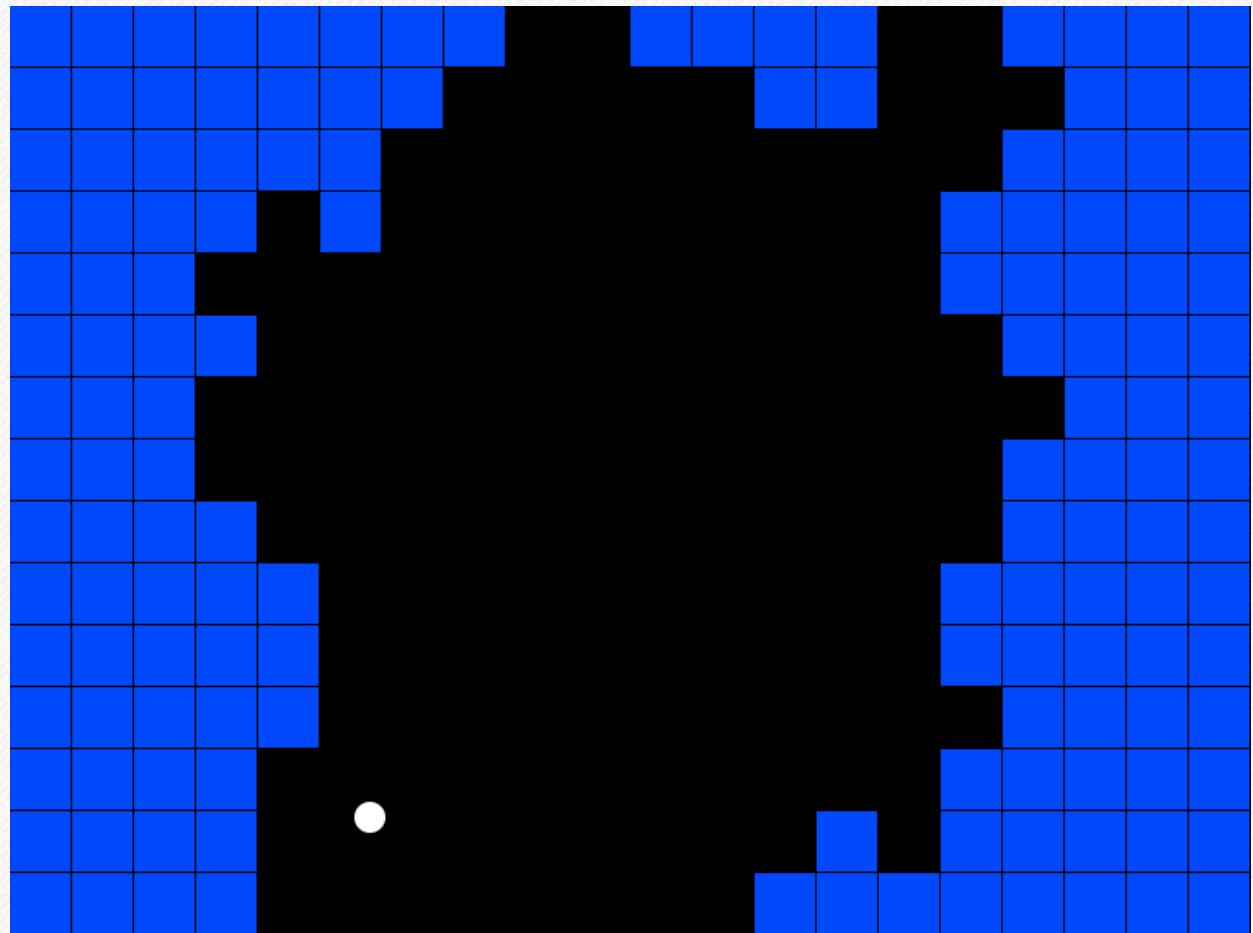
Resize the brick wall's parameters to be bigger and square, and to cover the whole screen with the race track pieces. Set both TRACK_W and TRACK_H to 40. Set the track size to be 20 TRACK_COLS by 15 TRACK_ROWS, since 20×40 is 800 (canvas width) and 15×40 is 600 (canvas height). TRACK_GAP can be 1 just to make it easier to see where one tile stops and the next one starts.

Much as you renamed instances of track in the previous step, rename all occurrences of "ball" in the code with "car" since over the next few changes you'll be repurposing the ball's position, draw, and collision code to become one of the two cars. It's again good to be aware of capitalization sensitivity when doing any find and replace in program files.

To fill in the whole screen with tiles modify the resetTracks() function so that it no longer skips rows to leave a 3 tile margin along the top of the canvas. All bricks should begin on screen.

EXPECTED RESULT

The full playfield will begin covered in square blue tiles, and the ball will begin automatically carving out from the center. Since the ball resets each time it escapes off the bottom of the screen, after a few minutes the scene will stabilize with the ball resetting and repeatedly going right off the same spot along the bottom.



Left running, the game as of this version will stabilize looking something like this.



racing step 4

support for hand-designed tracks

Currently, when the game starts, the track of bricks is totally filled in. That was the desired setup for the Brick Breaker game but it won't make for a suitable race track. Since the race cars won't remove track tiles or walls on contact, before turning the ball into a car, edit the track layout to leave room for the car to move.

The track layout I'll define in the example code looks like:

This is a perfect time to break the rule about not copy/pasting code from the book or sample solution files. Please don't type all that. Or if you want to make your own, I created this one by first forming a 20 by 15 (TRACK_COLS by TRACK_ROWS) grid of 1's separated by commas. Then I carved the road into the walls by putting in zeroes to form a path two tiles wide leading from one place in the map to another. Tracks that can support laps will be covered later, for now a start-to-finish one shot path will be easier to work with. Your track layout doesn't have to look as much like intestines as mine does, though.

The `resetTracks()` function – previously `resetBricks()` – smashes over all positions with 1's to start with a full grid. You no longer want that to happen. Remove the `resetTracks()` function and also the call to it over in the `window.onload` initialization code.

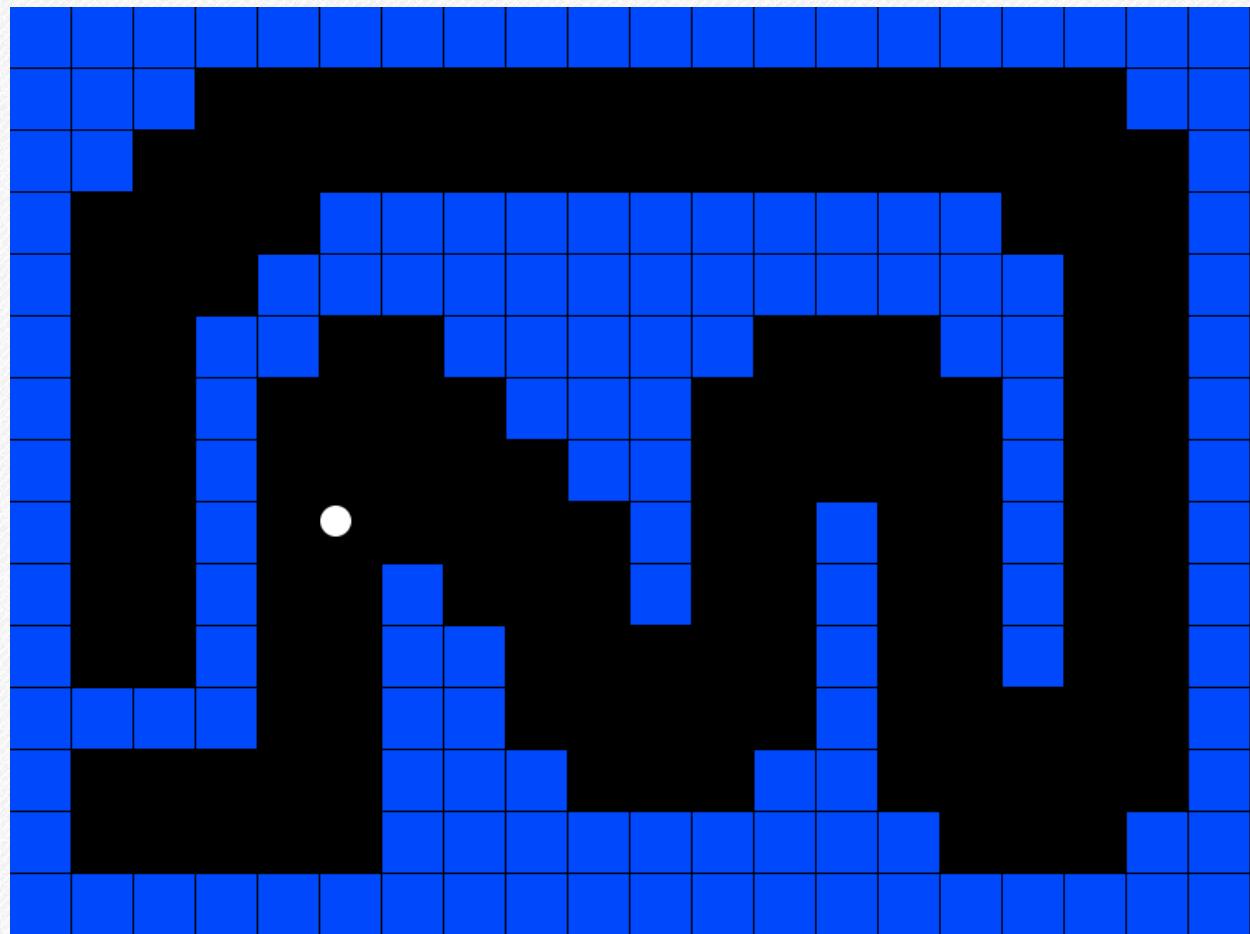
Remove any use of the "tracksLeft" variable. Tile sections will no longer be removed. You won't need to worry about tracking how many are left, and you won't need to reset them.

Remove one line from `breakAndBounceOffTrackAtPixelCoord()` that erased the tile overlapped by the ball/car. To reflect this change rename the function, both where it's defined and where it's called, to `bounceOffTrackAtPixelCoord()`. It's not supposed to do any "breakAnd" in this particular game.

In `carReset()` begin the car an extra 50 pixels to the right. Otherwise it will begin stuck shaking between two adjacent walls. If you're not clear on why this matters or what I mean by it, try not adding that 50 pixels to see the effect. That's how I noticed the need for it – I didn't know this would be necessary until I ran this step's code and saw it stuck where it started. Later on the car's start position will be set up as part of the grid tile information. This plus 50 is a quick fix until then.

EXPECTED RESULT

You'll see a race track, with the road bending around a bit like pixelated intestines. The ball will zig-zag back and forth around the black road area between the blue square tiles. The wall tiles should no longer be erased when bumped.



racing step 5

loading a car image for the player

To begin turning the ball into a car, you'll first need to load a car image to draw instead of the ball's white circle.

Set up a new variable named carPic. Initialize it to document.createElement("img"). Prepare another variable named carPicLoaded. Set it to false:

```
var carPic=document.createElement("img");
var carPicLoaded = false;
```

That second variable will be used to prevent the game from referencing the image before it had time to finish loading.

When tested locally, all files are already on the same computer, so it can load incredibly quickly. You could get away without that type of check for a locally tested game. However, time needed for files to load is a very real part of the process to be familiar with. Thinking ahead about it now will later make sharing the game easier. I'll later show a more general way to do this for a whole list of images being loaded, but this code's a little simpler for dealing with only one.

Just as the whole program runs the window.onload code when it finishes loading, an image variable can also be given an onload function to run when it completes loading. Give carPic a short onload function of its own, set up in window.onload, that will change carPicLoaded to true:

```
// load car image
carPic.onload=function(){
    carPicLoaded = true; // dont try to display until it's loaded
}
```

In the car's draw code you'll be able to check the value of carPicLoaded and skip using the image if it hasn't loaded yet.

Next in window.onload, outside of and after carPic.onload, set the image variable's source to the name of an overhead car image. I included player1.png as an example that also has headlamps and taillights to make it a bit easier to see which way the car is facing.

```
carPic.src="player1.png";
```

The file itself should be located in the same directory as the game's html file, since no relative path information is given.

Where you called colorCircle() at the end of drawEverything() call a carDraw() function instead. In that new function, check if the image has been loaded, and if so draw it centered at the carX, carY position.

Centering the image can be achieved by subtracting half the image's width and height from the car's position.

```
function carDraw() {
  if(carPicLoaded) {
    canvasContext.drawImage(carPic, carX-carPic.width/2, carY-carPic.height/2);
  }
}
```

EXPECTED RESULT

This will function the same as the previous step, except now instead of a white ball there should be a car image facing right. It will bounce left-to-right through the walls like the ball did.

The car won't yet face the direction that it's moving in. It has no way to rotate at this time.

racing step 6

spinning the car image

Before connecting the car angle to key input or drive motion, first continuously spinning the image will make it easier to see whether the rotated graphic code is working as expected.

Add a new variable to hold the car's rotation angle, carAng. In the carDraw() function, increment the car's angle by 0.2 so the car will continually spin, making it easy for us to check the graphic rotation before you hook up keyboard movement. In the block of code within carDraw() in which you've confirmed that the car's image has finished loading, do the following to connect the car's spin to the graphic:

```
canvasContext.save(); // allows us to undo translate movement and rotate spin  
canvasContext.translate(carX,carY); // sets the point where our graphic will go  
canvasContext.rotate(carAng); // sets the rotation  
canvasContext.drawImage(carPic,-carPic.width/2,-carPic.height/2); // center, draw  
canvasContext.restore(); // undo the translation movement and rotation since save()
```

Notice that the arguments for drawImage() here are different than they were in the previous step. Namely, carX and carY are not part of the arguments, since .translate() is handling those.

If you're worried that this looks complicated or hard to come up with exactly each time, remember that this is why code lets us define functions. There'll be no need to memorize this or type that much every time you want to rotate an image. Soon you'll wrap this up into a reusable function, so that later, for player two's car, you'll be able to draw it rotated with one line.

EXPECTED RESULT

The car graphic will be spinning constantly.

racing step 7

putting image rotation in a function

The code to display a rotated image is a confusing chain of calls to achieve a single purpose. Wrap it up in a new function:

```
drawBitmapCenteredAtLocationWithRotation(graphic, atX, atY, withAngle)
```

Then, you can cut the code that was being used to display the car, and copy that into the { body } of that function. Replace the graphic, position, and angle values with the function's parameters so you'll be able to reuse this function for other cases that need different images and other position values.

In carDraw() call that function with the arguments (carPic, carX, carY, carAng) instead of doing the whole series of canvasContext instructions right there.

EXPECTED RESULT

No change to program functionality. The code will be slightly easier to read and less prone to errors that would be caused by later getting rotation and translation instructions out of order.

racing step 8

using key press and release

It's time to start setting up keyboard control. Keyboard input in HTML5 is done by hooking functions for press and release events. Every key on the keyboard has a unique numerical keycode to match against in the press and release functions.

Add a new text element to the HTML page, outside and below the canvas, above where your `<script>` block opens. The text can use a `<p>` tag, with an attribute of `id="debugText"`:

```
<canvas id="gameCanvas" width="800" height="600"></canvas>

<p id="debugText">(recent text display or temporary debug output)</p>

<script>
  // variables to keep track of car position
  var carX = 75, carY = 75;
  // (etc., rest of the program code continues below)
```

There you'll print out the keycode that matches any key pressed. This will promptly show whether your key response functions are being called as intended. This will double as a way to easily get the numbers that correspond to specific keys that you'd like to use. In later steps it will be a handy way to have on-screen text outside the game that can be overwritten any time you'd like to show a memory value while troubleshooting. Using the attribute `id`, "debugText", you'll be able to set that text from JavaScript with a line like this one:

```
document.getElementById("debugText").innerHTML = "replacement text";
```

In the `window.onload` block of initialization create two event listeners that trigger when any key gets pressed or released:

```
document.addEventListener("keydown", keyPressed);
document.addEventListener("keyup", keyReleased);
```

Above and outside of the window.onload initialization block, define the keyPressed and keyReleased functions. Each needs to take an argument of (evt) – short for event – from which you'll be able to check the key's keyCode and display it as text in the HTML's <p> “debugText” block below the canvas:

```
function keyPressed(evt) {
  document.getElementById("debugText").innerHTML = "KeyCode Pushed: " + evt.keyCode;
}

function keyReleased(evt) {
  document.getElementById("debugText").innerHTML = "KeyCode Released: " + evt.keyCode;
}
```

EXPECTED RESULT

The game's functionality won't be any different.

Pressing and releasing keys should now change the text below the canvas to reveal which keycodes are pressed.

For example, pressing the up arrow will replace the text at the bottom to read "KeyCode Pressed: 38" and when that key comes back up that will change to "KeyCode Released: 38". Pressing the other keys reveals that the left arrow is 37, the down arrow is 40, and so on.

racing step 9 cut how the car has been moving

To tell that the changes to car motion work, you'll first need to temporarily remove the automatic motion and spin that has been applied so far. It will move in a very different way soon.

To simplify the movement code, first remove all the edge checking and tile collision code from moveEverything(). Also remove the line that was automatically increasing the car's rotation angle. In terms of motion this leaves only the two lines that add the car's speed x and y variables to the car position.

Lastly, eliminate the initial movement values by zeroing out carSpeedX and carSpeedY where they are declared.

EXPECTED RESULT

The car should now sit motionless in the middle of the playfield, facing right. It won't respond yet to key input.

racing step 10 driving the car in a spiral

Keeping separate x and y speed values worked fine for the original ball bouncing motion, since the ball has neither a "front" direction nor any forward power of its own. By contrast, the car will always be facing in a particular direction. Not only will the car's facing affect which way it accelerates in when the gas pedal is down, turning and changing the car's direction will also immediately redirect the heading of its current speed.

Rather than treating the car as a bouncing point (storing carX, carY, carSpeedX, and carSpeedY), you'll now need to deal with it as a position, direction, and speed (storing carX, carY, carAng, and carSpeed). Setting carSpeed to a negative value, for example, will send the car in reverse, meaning opposite the direction currently stored as carAng. There's already a carAng leftover from spinning the image. Replace the carSpeedX and carSpeedY values with a single new variable, carSpeed.

Remove the lines in moveEverything() where code was adding the separate x and y values to the car's x and y position. Replace these with trigonometry calls to convert the linear forward or backward speed to separate x and y components:

```
carX += Math.cos(carAng) * carSpeed;  
carY += Math.sin(carAng) * carSpeed;
```

How or why does this work? The trigonometric functions cosine and sine, or cos() and sin() in code, return the length of each leg of a right triangle that has a hypotenuse 1.0 long and an inner angle of the value provided (carAng, in this case). These functions can be thought of as answering the question,

"If a car is driving 35 KPH bearing 60 degrees, what percent of that speed is going east, and what percent is going north?"

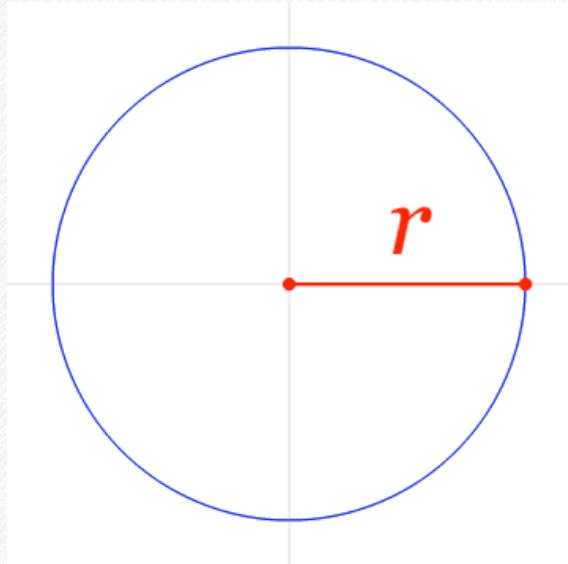
The answer is $\cos(60^\circ) = 0.5$, or 50% eastward and $\sin(60^\circ) = 0.8660$, or 86.6% north. To find its actual speeds in each direction multiply by the total speed: $0.5 \times 35 \text{ KPH} = 17.5 \text{ KPH}$ eastward, and $0.8660 \times 35 \text{ KPH} = 30.3 \text{ KPH}$ northward.

Before hooking up keyboard inputs, add automatic changes to the movement variables to show that they work as expected. At the top of `moveEverything()`, add a line to increase `carAng` by 0.04 (note: the `sin` and `cos` trig functions expect radians, so each frame the car spins by $0.04 \times 180.0/\pi = 2.3$ degrees), and another line to raise `carSpeed` by 0.02 each frame. This will make the car constantly turn slowly to the right, building up speed from the moment the game begins to run.

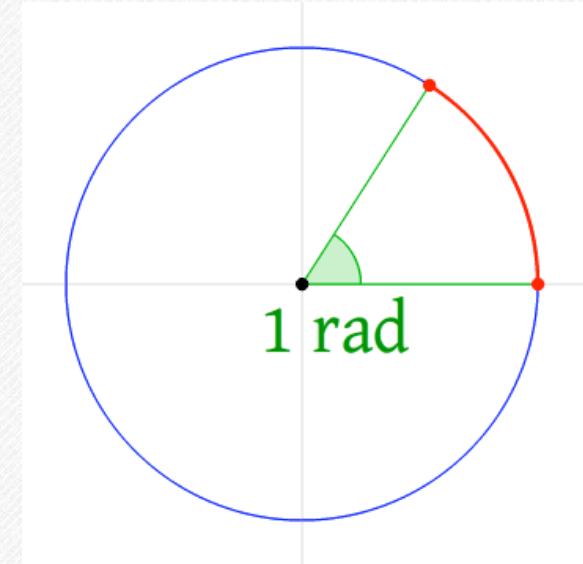
(If you're unfamiliar with radians, instead of measuring an angle in 360 degrees it measures angles by how many times a circle's radius is wrapped along the outside of a circle to cover that arc. The same measures work regardless of circle size since circles share the same proportions. See opposite page.)

EXPECTED RESULT

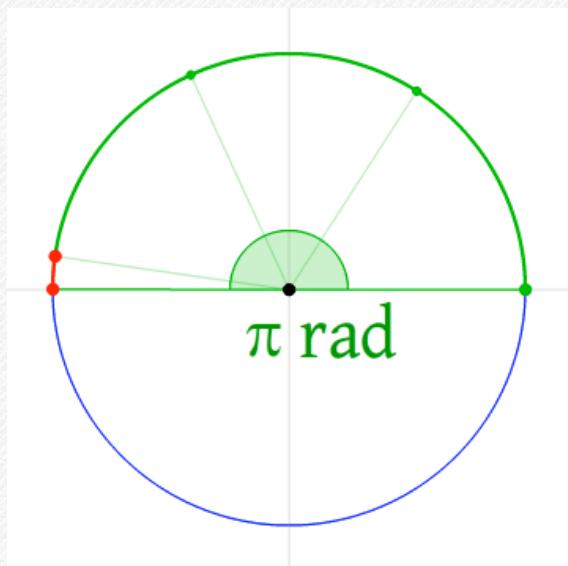
The car will start in the center of the playfield, then spiral outward, driving slowly at first and building up speed gradually. The car will be going right through any walls in its way, and won't have any reaction to crossing the edge of the screen aside from going out of view. Left running until the car's spiral has grown large enough, the car will go out of sight until the page is refreshed to reset the game.



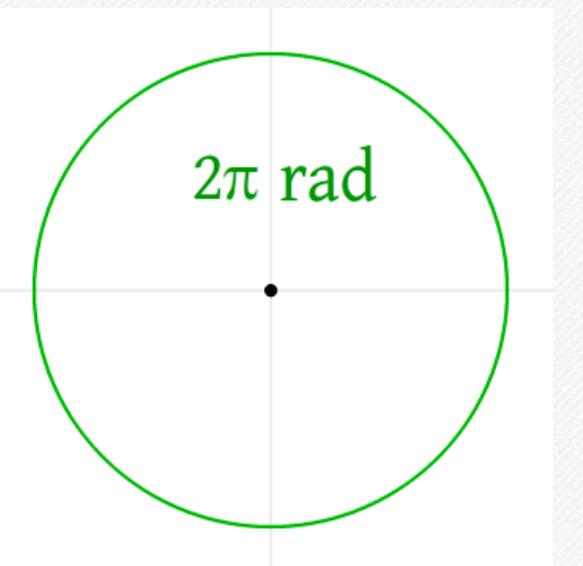
Circle showing its radius, r



Radius laid along edge, 1 radian



$3.14159\dots$ (π) radians is 180°



So 2π is one full spin, or 360°

The above images showing radians as an angle measure are by Lucas V. Barbosa, who placed them in public domain in 2013.

racing step II

speed and turning with arrow keys

Before driving the car smoothly by holding down control keys, as an intermediary step, have the motion variables change each time the drive keys get pressed. Holding the key to make the car turn or speed up will involve a few more variables. For now, tapping the arrows will cause the car to speed up, slow down, or turn in big, instant increments of 45° at a time.

The `keyPressed()` function is still rewriting the text below the canvas to show the `keyCode` variable's values for every key pressed and released. You can use this to check the number for each arrow key. For example, pressing the up arrow reveals that it sets `evt.keyCode` to 38. This means that within `keyPressed()` you can check whether `evt.keyCode` is equal to 38. If it is, run whatever code you want activated when the up arrow gets pressed.

Matching against the arbitrary magic number 38 would make the key controls code very difficult to read or update later. Define four new constants early in the code, setting one number for each of the four arrow key's codes. Since you the up arrow's code is 38, the first of the four will be:

```
const KEY_UP_ARROW = 38;
```

It's set to `const` rather than `var` so that it can't accidentally be overwritten later (the `keyCode` for a key never changes). It's named in all capitals to remind us whenever we see or use it that it's a constant. The program would behave in exactly the same way if `KEY_UP_ARROW` was named `keyUpArrow`, but

then it would look a lot like our variable values as opposed to standing out as a const. This type of convention is purely a decision left to us as the programmers.

Instead of checking if evt.keyCode equals 38 in keyPressed(), you can compare it to KEY_UP_ARROW. That's much easier to read, understand, or make changes to later.

Accelerate the car by another 1.5 pixels/frame if the code for the key pressed matches the up arrow:

```
if(evt.keyCode == KEY_UP_ARROW) {  
    carSpeed += 1.5;  
}
```

Reminder: one equal sign in code is an assignment operation that sets the value to the left of it, but a double equal sign is a comparison. If evt.keyCode and KEY_UP_ARROW have the same value then (evt.keyCode == KEY_UP_ARROW) will be true, allowing the code within the if's { scope } to run.

Also, just to emphasize something important but potentially counterintuitive that's going on here: the keyPressed() function will get called no matter which key goes down. There's no way to connect it to, for example, only call the function if the letter J key gets pressed. All key presses activate this same function and within it individual keys can be filtered for via comparison to the evt.keyCode value.

Do likewise for defining and checking against a new const named KEY_DOWN_ARROW. When pressed, subtract 1.5 from carSpeed. Do the same for new const keyCode values KEY_LEFT_ARROW and KEY_RIGHT_ARROW, except instead of adding or subtracting to carSpeed have them add or

subtract $0.25 * \text{Math.PI}$ to carAng. (`Math.PI` is a built-in approximation for, pretty clear I think, the math constant pi.)

Why $0.25 * \text{Math.PI}$? As mentioned in the previous step, angles in HTML5 for image `rotate()` or trig's sin/cos functions expect radians rather than degrees. In radians a full spin is $2.0 * \pi$. One eighth of that, $0.25 * \pi$, is one eighth of a full rotation. In more typical measure that's an increment of 45° . This way the vehicle can drive in the 4 cardinal directions, North-East-South-West as well as the intermediates (NE, SE, SW, NW).

Include the following at the end of your `keyPressed()` function:

```
evt.preventDefault();
```

This will block the up and down arrow keys as well as the spacebar from serving their default functionality in the web browser. For many modern browsers arrow keys and space cause the page to scroll if there's room to do so.

Lastly, remove the automatic turning and acceleration from the top of `moveEverything()`. That was only in there to test the new movement code before hooking up these control keys.

EXPECTED RESULT

Left and right arrows turn the car in 45 degree increments. Tapping the up arrow makes the car move forward faster, and tapping the down arrow has the opposite effect, slowing it down or putting it in reverse. Currently the car still drives through walls unaffected and can leave the viewable area.

racing step 12

support for key holding

Tapping keys isn't how anyone wants to control a car.

Keys and their default interface to the computer have been designed for typing. As such their built-in support doesn't care whether they're held down. To make the keys work more like gamepad action buttons you'll need to add a little code so that they can trap a held down state.

Instead of taking action in code only in the moments that a key's position changes between up and down, you can use those instants to update a button state which will reflect whether the key is currently held down. If it got depressed but hasn't come back up yet that must mean it's being held. The control logic can check and respond to the held/not-held (true or false) condition instead momentary status change signal. Set up new true/false variables after the key constants at the top of the script that correspond to each control:

```
// keyboard hold state variables, to use keys more like buttons
var keyHeld_Gas = false;
var keyHeld_Reverse = false;
var keyHeld_TurnLeft = false;
var keyHeld_TurnRight = false;
```

In the function that detects key presses, instead of directly increasing or decreasing variables like carAng or carSpeed, set the corresponding key's variable to true.

For example, the up key will now work like this:

```
if(evt.keyCode == KEY_UP_ARROW) {  
    keyHeld_Gas = true;  
}
```

In the keyReleased function, do the opposite. Set the corresponding hold boolean (true/false variable) false when the matching evt.keyCode is detected.

At this point go ahead and remove the debug line that was displaying the keyCode to the text on the site below the canvas when keys are released. It was just there to figure out which codes corresponded to the arrow keys.

At the top of moveEverything(), check for each key hold value, and respond based on what numerical effect you want holding that key to have. So, for example, with the left arrow key:

```
if(keyHeld_TurnLeft) {  
    carAng += -0.03*Math.PI; // same as: carAng -= 0.03*Math.PI;  
}
```

Adding $-0.03 * \text{Math.PI}$ is the same as subtracting $0.03 * \text{Math.PI}$, but I opt in this case to add the negative to further differentiate this line from how the similar one for keyHeld_TurnRight looks. I'm more likely to spot a difference in what's right of the equal sign than in seeing that one's using $+=$ and the other $-=$ for it.

I adjusted the turn rate from $0.25 * \text{Math.PI}$ to a smaller fraction since this now happens way more often: every frame while the key is held, rather than once per key press. When the gas key is held down, add some amount to carSpeed, perhaps around 0.5. Do this for the other arrow keys as well. You should now be able to use the keys like buttons, holding them down to cause a smooth effect in your game.

EXPECTED RESULT

Holding the up arrow key speeds the car up, holding the down arrow key slows the car down. Holding the left or right arrow keys will cause the car to turn slowly.

The car will not yet slow down on its own – any forward or backward speed applied to the car will be maintained until holding the opposite direction key to counteract it.

The car will still go through walls and off the edge of the viewable area.

racing step 13

rolling to a stop, & no turn while still

The car should roll to a stop on its own. Add a const near the top of the code, GROUNDSPEED_DECAY_MULT, and set it to 0.94. That will be used in code to mean, "retain 94% of the car's speed each frame." By shaving off 6% of its speed each frame the car will roll to a stop whenever it isn't being actively powered. At the end of the moveEverything() function multiply the carSpeed against this to continually shave off a fixed percentage of the car's speed.

Instead of burying tuning numbers deep in the code where they are used, they should be meaningfully named constants set near the top of the file. This makes it easier to tune the car's motion later. It also makes for code that's easier to read.

Define as constants values for DRIVE_POWER (0.5, perhaps), REVERSE_POWER (about 0.2, cars reverse more slowly than they drive forward), and TURN_RATE (0.03 feels ok to me). Put a negative sign in front of TURN_RATE for one direction instead of having a different value for left and right. Find any numbers that were typed directly into the moveEverything() function and replace them with their new const labels.

While introducing some control constants, something else to add and account for is a minimum turning speed. This will prevent the car from spinning in place, since cars (unlike, say, tanks) can't turn without moving forward or backward. I'll name the const MIN_TURN_SPEED and set it to 0.5. Check in moveEverything() whether the car is going at least this speed,

and only turn the car in response to holding left or right arrow keys if the car is moving fast enough. To ensure this works correctly when the car is in reverse, too, the comparison should be against the absolute value of carSpeed, which can be determined in HTML5 by using Math.abs(carSpeed).

More code cleanup: keyPressed() and keyReleased() currently have copy and pasted information, namely which keys correspond to which variables. The only difference between them is whether the function sets a corresponding key hold variable to true or false. If you later change which keys control the player but forget to make the change in both places, you'll wind up with broken controls.

To simplify the code and reduce potential mix ups later, write a shared function that both keyPressed() and keyReleased() can call on to centralize the input matching for each keyCode. Set the hold state to true or false based on a passed in argument.

This way, the press and release functions can be guaranteed to acknowledge the same keys and always apply the correct true or false setting. The helper function can take this form:

```
function setKeyHoldState(thisKey, setTo) {  
    if(thisKey == KEY_LEFT_ARROW) {  
        keyHeld_TurnLeft = setTo;  
    }  
    // ...other keys need to be checked and handled in the same way...  
}
```

This eliminates the similar-looking chain of four if-statements in both keyPressed() and keyReleased(), leaving only:

```
function keyPressed(evt) {  
    setKeyHoldState(evt.keyCode, true);  
    evt.preventDefault(); // without this, arrow keys scroll the browser!  
}  
  
function keyReleased(evt) {  
    setKeyHoldState(evt.keyCode, false);  
}
```

EXPECTED RESULT

Arrow keys can be used to drive the car. When the forward and back arrow keys get released the car will slow down until it comes to a halt (technically it keeps moving very, very slowly but after a point that change is so minor that it won't produce a change to the car's position on screen). The car should no longer turn in response to the left and right arrow keys when the car is sitting still or moving very slowly.



racing step 14 set car start position in track data

Before preventing the car from driving through walls, you'll want a way to ensure the car doesn't start inside a wall.

Rather than hardcoding a new start position through trial and error, add a special marker to the trackGrid array that will be interpreted elsewhere in code as the car's start position. Since the trackGrid uses 0 to signify road and 1 to signify wall, use the next number, 2, to mark the start position for the car.

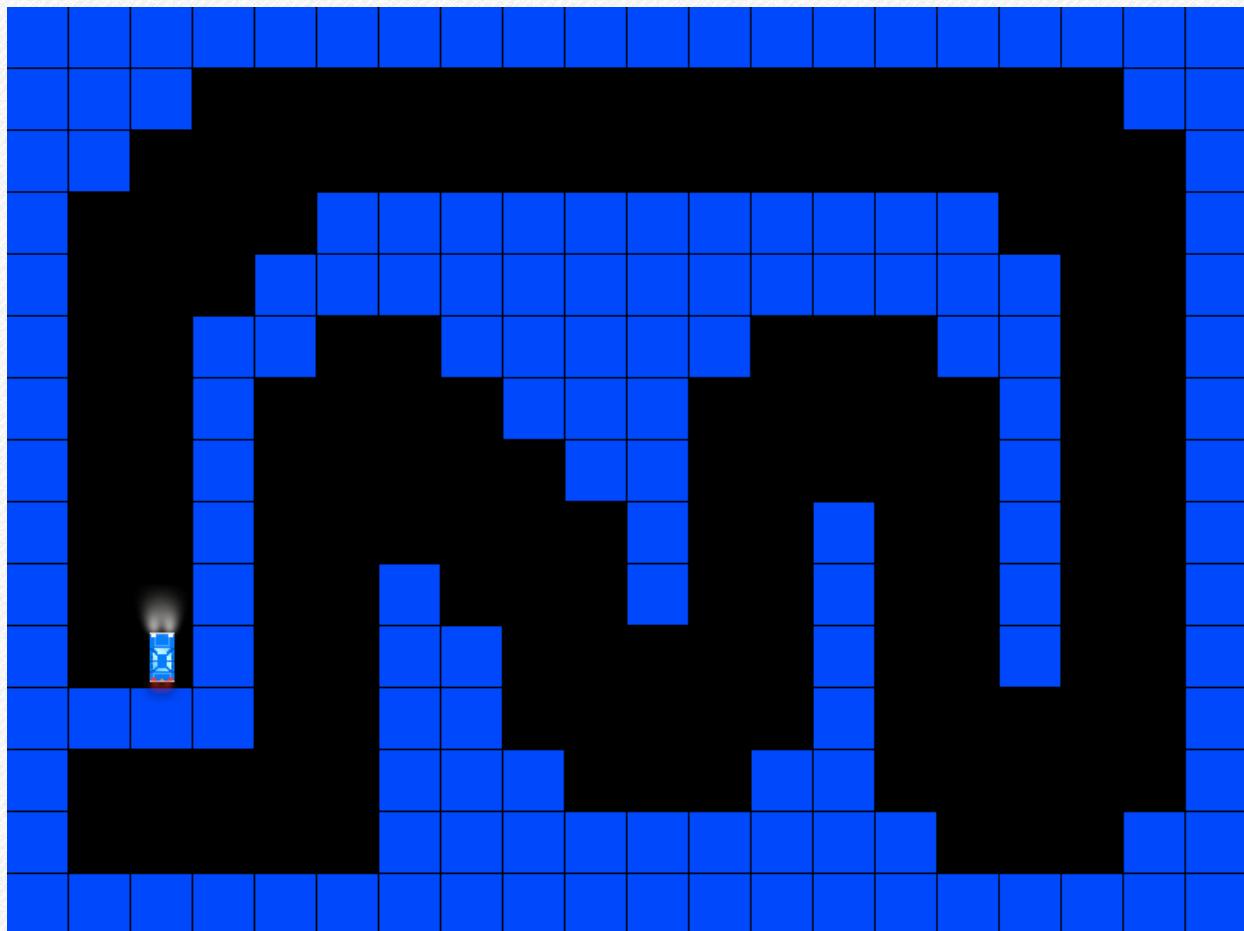
Position it at the beginning of the track, like so:

In the carReset() function, instead of setting carX and carY to the center of the screen, write a for-loop to scan through each index of the trackGrid array in search of a 2. Upon finding the 2, calculate the corresponding row and column from the index:

```
if( trackGrid[i] == 2) {  
    var tileRow = Math.floor(i/TRACK_COLS);  
    var tileCol = i%TRACK_COLS;
```

That math works for tileRow because each row has TRACK_COLS values, therefore if the index of the matching

value is, say, 4-5 multiples of TRACK_COLS, then it must be 4 rows down from the top. Math.floor() is then used to remove the fractional part, leaving a whole number for rows. It matters that the unit belongs 4 rows down, not “4.3” rows down. Math.floor() simply rounds down to the nearest whole number.



As for what that leftover 0.3 means, it's related to how many columns over the car is within that row. Its horizontal spot is 30% (0.3) across the number of columns. If a car is in position 66, at 20 columns per row that's 3 rows down, accounting for the first 60 of those 66 tile positions. That leaves 6. That remainder means the car is 6 tiles over from the left side.

Remember remainder from grade school? It's the number leftover after dividing by some other number. People learn it

before covering fractions and decimal points, then kind of ignore and forget about it thereafter. The remainder can be used to get from tile array index to horizontal tile position. For example, because 46 divided by 20 (where TRACK_COLS is 20) leaves a remainder of 6, tile array position 46 is in horizontal tile position 6. The modulus operator, %, is a programming math operator that calculates the remainder. It takes the form: 46 % 20 equals 6, since $46 / 20$ equals 2 remainder 6.

Now, knowing that the 2 in the grid is some number of tiles from the left, and some number of tiles from the top, you can translate that number back into pixel coordinates for the car. Each track tile is TRACK_W by TRACK_H pixels in size, and you want the car in the center of the tile rather than its top-left corner, which results in the following calculation:

```
carX = tileCol * TRACK_W + 0.5*TRACK_W;  
carY = tileRow * TRACK_H + 0.5*TRACK_H;
```

Looking at the expression to the right of the equals sign, the portion before the plus sign uses multiplication to find the car tile's corner (5 tiles over, 40 pixels per tile, so $5 \times 40 = 200$ pixels in from the left). The portion of those expressions to the right of the plus sign moves the spot half a tile over and half a tile down to get the center pixel coordinate of the tile there.

After setting up the car's position, replace the number in that tile's spot with a 0 – meaning empty road. That way, the collision and rendering code won't need to handle the special case of finding a '2' in the map during game play. At this point in code you already have the index for the tile; it's the for-loop

index used to calculate the tileCol and tileRow, so this will make the swap:

```
trackGrid[i] = 0;
```

This can all be a bit much to follow if you're relatively new to working with a tile map, or to the idea of translating to and from tile (col, row) coordinates and pixel (x, y) coordinates.. When it's time to do more of these calculations in more areas of code, I'll show how to make a simple helper function or two that can centralize the calculation mess. While it's only happening in one or two spots and it's something you're warming up to, it's good practice to reason through it.

Add a debug line, temporarily for this step, to show in the text below the canvas what coordinates of tiles and pixels were calculated. Add this on the next line (reminder: the + can be used to combine strings and variables to format output):

```
document.getElementById("debugText").innerHTML =
  "Car starting at tile: (" + tileCol + ", " + tileRow + ") " +
  "Pixel coordinate: (" + carX + ", " + carY + ")";
```

If you're unsure how the math is working, try repositioning the 2 in the level grid someplace else, and watch how the debug values displayed below the canvas come out differently.

Having found a 2 while scanning through the grid, putting a "break;" statement after it will escape the loop and stop scanning for any more 2's in the grid. This will be handy when adding support for multiple players, so that each car will only consume one 2 start position, instead of finding and erasing all of them from the level data on the first scan through.

Now that the track grid has an additional value with meaning, and is no longer being used as a simple on/off marking for where walls are located for collision and rendering, define a few constants near the top of the script to clarify the meaning of each number used in trackGrid:

```
const TRACK_ROAD = 0;  
const TRACK_WALL = 1;  
const TRACK_PLAYER = 2;
```

I recommend putting these directly under the definition of trackGrid, so they can easily serve as a handy key when rearranging the track map. After defining these constants, the comparison checking for 2 in carReset() should now check for TRACK_PLAYER. Change the line which set the found tile to 0 to set that tile to TRACK_ROAD instead.

Lastly, change the default for carAng to point north instead of east. Remember once again that HTML5 expects angle measures in radians rather than degrees, so instead of initializing carAng to -90, it should be set to (-0.5 * Math.PI).

EXPECTED RESULT

The car should drive the same as it has been, except now it will start facing north from the beginning position on the track that was marked in the grid with a 2.

racing step 15 walls that block car movement

No more driving through walls.

Rename the `bounceOffTrackAtPixelCoord()` function – which hasn't been used in awhile but should still be there – to instead be `checkForTrackAtPixelCoord()`. There's an if-conditional in there that checks whether the track at a given index "`== 1`" in reference to overlapping a wall tile.

If coding like zombies on autopilot (an especially dangerous combination) you might replace that 1 with `TRACK_WALL` since that's the tile const for a grid position set to 1. However what's needed is to check for *track* – meaning road, not wall. Don't to match for `TRACK_WALL`, but for `TRACK_ROAD`.

Replace the if keyword at the start of that statement with a return keyword, so that the whole function will return true if a road tile is found, and otherwise it will return false:

```
function checkForTrackAtPixelCoord(pixelX,pixelY) {
    var tileCol = pixelX / TRACK_W;
    var tileRow = pixelY / TRACK_H;

    // we'll use Math.floor to round down to the nearest whole number
    tileCol = Math.floor( tileCol );
    tileRow = Math.floor( tileRow );

    // first check whether the car is within any part of the track wall
    if(tileCol < 0 || tileCol >= TRACK_COLS ||
       tileRow < 0 || tileRow >= TRACK_ROWS) {
        return false; // bail out of function to avoid illegal array position usage
    }

    var trackIndex = trackTileToIndex(tileCol, tileRow);

    return (trackGrid[trackIndex] == TRACK_ROAD);
}
```

Notice that the last line there behaves exactly the same as:

```
if(trackGrid[trackIndex] == TRACK_ROAD) {  
    return true;  
}  
return false;
```

Instead of returning true when the value inside the if-statement is true, and otherwise returning false, returning the contents of the if-statement directly serves the same purpose. Notice too that no else condition was needed to wrap the return false in the above other way of writing it, since when code gets to a return statement it bails immediately before running any later parts of the current function.

Look a little higher in the code at `isTrackAtTileCoord()`, which is used by the game's draw code. It also has an "`== 1`". Pop quiz: should that be `TRACK_WALL` or `TRACK_ROAD`?

Since it's used to draw where walls are, it should be `TRACK_WALL`. The name of the function should be changed to reflect that, going from `isTrackAtTileCoord()` to something like `isWallAtTileCoord()`, since track means road. Its call in `drawTracks()` also has to be updated to use this name.

As for the actual collision logic, skip to where the `cos()` and `sin()` of `carAng` times `carSpeed` are added to `carX` and `carY` in `moveEverything()`. The task here is to figure out how to involve `checkForTrackAtPixelCoord()` to detect and respond to car overlap with a wall.

Rather than passing `checkForTrackAtPixelCoord()` the car's current position, feed it the car's next position to test. If the car's next position is valid, meaning it's a tile of road rather than wall, then update the `carX` and `carY` variables to those next coordinates. There's no reason to calculate the next

position twice (once to check it, and the second time to update carX and carY), as you can make nextX and nextY local variables to use in both places:

```
var nextX = carX + Math.cos(carAng) * carSpeed;  
var nextY = carY + Math.sin(carAng) * carSpeed;
```

Now you can call `checkForTrackAtPixelCoord(nextX,nextY)` and if it returns true, set carX to nextX and carY to nextY. Otherwise – as an else condition – reset the carSpeed to 0.0 to bring it to an abrupt halt due to smashing into a wall. Even if the car wouldn't move forward over the wall, without that step the car would have to decelerate from its (unused) remaining forward speed before it could back up.

As a slightly different approach you may prefer, instead of setting carSpeed to 0.0 when the car hits a wall you could instead set it equal to `-0.5*carSpeed`. That will make the car to bounce backwards at half the speed it had going into the collision. This is a bit more cartoony, and isn't how I'll handle it in the example solution, but it's worth at least trying out.

EXPECTED RESULT

The car can now race from the start of the maze to the end. If it runs into any track walls it'll stop and need to be backed up. Since the track design fully surrounds the viewable area with collision tiles, there was no need to add special handling for when the car passes the boundaries of the canvas.

(Naturally, if you opted to use the car bouncing approach rather than setting the speed abruptly to 0.0, then instead of the car stopping when it hits walls it should promptly roll in the opposite direction from which it came.)

racing step 1b separate javascript from the html

The program is beginning to grow too long to comfortably keep in a single file. In addition to sheer length, it contains several unrelated groups of functionality: core program, car stuff, track information, input code, and so on.

As the first step toward fracturing the program into smaller files, even separating out the JavaScript into a JS file to load from the HTML file will be a productive start.

Duplicate or copy/paste your whole program source, rename the copied one to Main.js, and open it with a plain text editor. In the Main.js file remove any HTML code that comes above or below the JavaScript. That includes the `<script>` and `</script>` tags at the start and end. Back over in the HTML file, replace everything between, and including, the `<script>` and `</script>` tags with this one line that references the external JS file:

```
<script src="Main.js"></script>
```

Save both files. To test this and future steps you'll still drag and drop the HTML file onto a browser window. There's no need to deal directly with the JS file(s) from the web browser. The opposite is also true: you won't be doing much coding in the HTML file now that you can just edit the JS file.

EXPECTED RESULT

There's no change in functionality. Improved file organization counts as forward progress though, since it's going to make expanding and maintaining the program's functionality faster and easier.

racing step 17

split the code into multiple files

Carving the JavaScript file into more specialized, separate ones will complete what the previous step started.

One quick change that can simplify this step is to move all car movement code out of moveEverything(). Since the car is the only thing in the game that moves, it didn't really matter that its code was all over moveEverything(). To keep moveEverything() in the main file and all car related code in a separate Car.js file, move all the car movement code from moveEverything() to a new function named carMove(), and simply call that function from moveEverything() as its only line of code.

The goal now is to have five separate files: Main.js, Car.js, Track.js, GraphicsCommon.js, and Input.js, where each has code for a particular fraction of the whole game. There are two ways to do this, depending on your comfort and speed with cut/paste and flipping between files:

1. Make four extra copies of Main.js then rename each, deleting blocks that don't fit the name.
2. Make four new blank files with the new names and go through Main.js, cutting and pasting one block at a time from the main file into the others.

I suggest the latter approach, since it decreases the chance of accidentally deleting any of the code from all files, or on the other hand leaving duplicate code sections in multiple files.

This is not the same as organizing the program into classes. At this point it's simply splitting the top, global level variables and functions into named files to break up the code and make it a bit easier to find what you're looking for without needing to dig past everything else in the program. Any function can still call any other function or reference any global variable even if they are defined and mainly used in another file.

What belongs in each file? I'd say give GraphicsCommon.js colorRect(), colorCircle(), and the drawBitmap...WithRotation() functions. Input.js should get all keyboard-related constants, variables, and key handling functions. The currently unused calculateMousePos() function leftover from earlier programs would belong in Input.js if you needed it. Since it's not hooked to a mouse event, or called from anywhere, it can be deleted. Car.js and Track.js should contain any variables, constants, or functions specific to the car or track.

Main.js then holds any core stuff leftover, in particular: the variable declarations for canvas and canvasContext, the window.onload initialization function, and the definitions for moveEverything() and drawEverything().

The main HTML file now needs to have all five files included, so above the line that includes Main.js add:

```
<script src="GraphicsCommon.js"></script>
<script src="Car.js"></script>
<script src="Track.js"></script>
<script src="Input.js"></script>
```

A little more shuffling around for organization before moving forward: the setup handled in window.onload inside Main.js involves a pair of addEventListeners used for input. Those

`addEventListener()` lines should be replaced by a call to a new `initInput()` function defined in the `Input.js` file that contains the replaced lines. Likewise, the `Main.js` `window.onload` function has five lines relating to set up of the car's graphics and start position. Move these five lines into a newly created `carInit()` function in the `Car.js` file and call that function from `window.onload`.

EXPECTED RESULT

Still zero change to program functionality, it should do exactly the same things step 15 did earlier.

What's new and very important here is that instead of having one nearly 250 line file that mixes together variables and functions for every part of the program, you now have a pretty well defined separate place to find and work on each piece of the program. It's now files that are only 30 lines (`Main.js`), 75 lines (`Car.js`), 40 lines (`Input.js`), 19 lines (`GraphicsCommon.js`), and 69 lines (`Track.js`).

This also helps highlight how relatively straightforward the `Main.js` code is, consisting of only some set up, car motion, screen clearing, track drawing, and car drawing. The gritty details of how all that's happening is set apart in the other files that have descriptive labels. It's like the difference between having a garage that's just a pile of random stuff versus having everything in labelled bins – when you need something you can quickly narrow down where to go looking for it.

As an added benefit, you've now started to create a small and simple library of sorts that has a few functions you may wish to reuse in another project. `GraphicsCommon.js`, and to a

lesser extent Input.js, shows how chunks of code can be separated from the more game-specific files.

If you're not using a text editor or development environment that supports having multiple files open at the same time, now would be a great time to find one that does. Sublime Text is a popular option available for Windows, Mac, and Linux. TextWrangler for Mac is free, as is Notepad++ for Windows. Many other options exist, just search the internet to browse text editors for programming.

A quick note about this division into more JS files: there are ways to merge (and compress, or even obfuscate) your code so it's not human readable) multiple JavaScript files together into just one before uploading to the internet. This decreases the total number of files that have to be kept together, slightly speeding up a game's load time by reducing the number of separate file access requests. This won't be covered here, and the game will work fine split across multiple files in the way shown here. I wanted to mention this as a topic you may wish to dig into before public release of your original videogame projects.

racing step 18

load and show track tile graphics

Driving in darkness and bumping into flat blue squares worked fine for testing, but nobody wants to look at that for long.

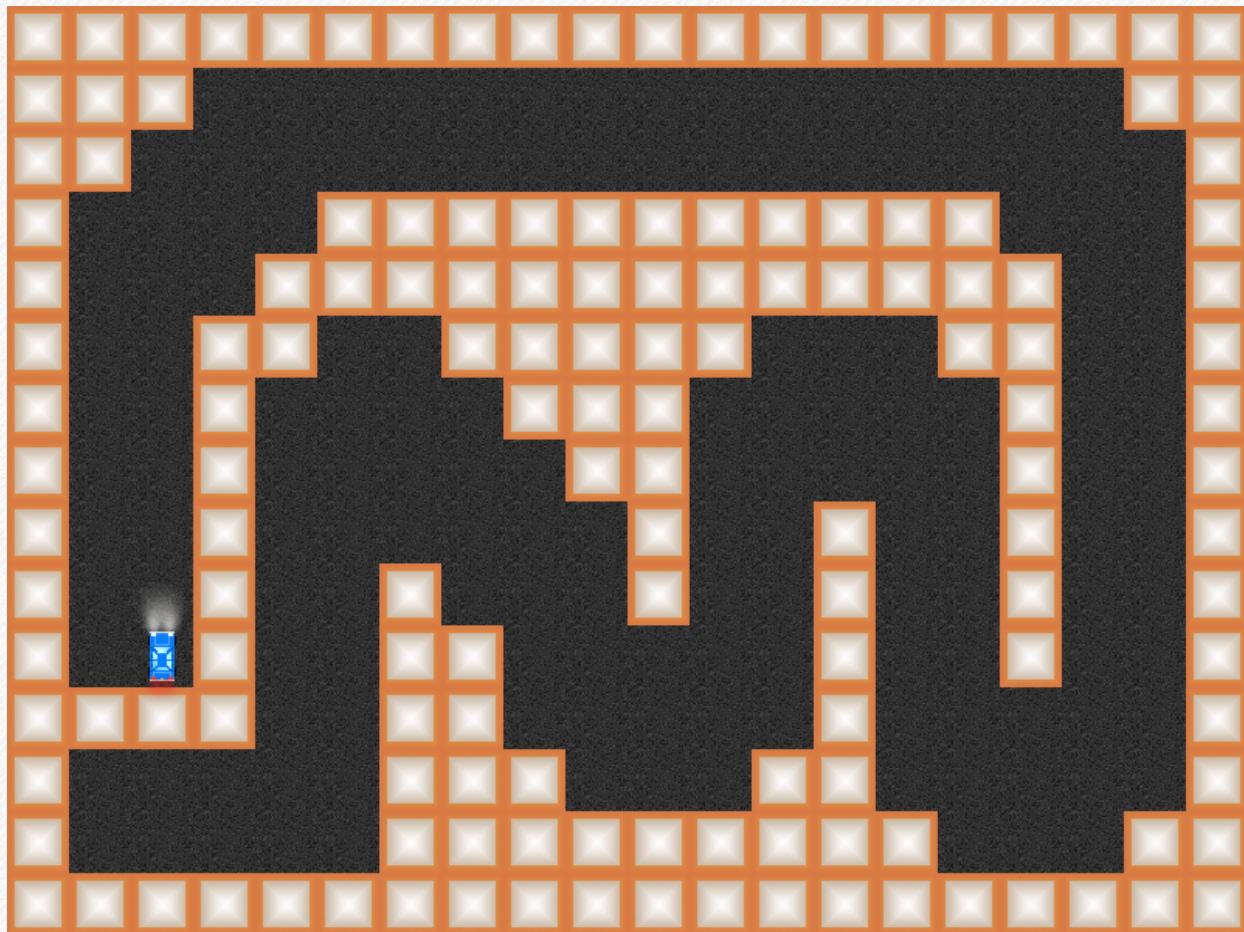
Adding image graphics for the track and walls will help situate the car in a space better resembling where cars ought to be.

This will involve more explanation than many other steps because making this change will involve creating another file and changing how the game code kicks off. The actual total number of new or moved lines of code won't be bad though.

At first you only need two 40x40 images: one for a road tile, and the other for a wall section. I've created examples named `track_road.png` and `track_wall.png` that can be found with the sample solution for this step. You are welcome to create your own. Even if you don't plan to become an artist for game development, getting some functional fluency with making simple images that are recognizable is a valuable skill for prototyping, demonstrating your ideas, or filling in gaps.

In the `Car.js` file, you already have one example of loading a PNG file and displaying it. You could copy that pattern into `Track.js` for road and wall segments. However, there's a good reason not to do that. Notice how the `carPicLoaded` variable in `Car.js` gets used to block the player graphic from being displayed before it finishes loading. If the image hasn't finished loading the program skips displaying it. That means the car might be invisible for a bit after the game starts! Wouldn't it make more sense for the program to wait for graphics to load?

Additionally, keeping a separate true/false value for each image that you wish to use in the game can get unwieldy. The need to check each prior to displaying or accessing the graphic's information (like its width or height dimensions) is prone to mix ups from forgetfulness or copy and paste errors failing to keep the pairs in sync every place they appear.



A more general way to handle this issue is twofold: (1) have a single numerical counter for how many more images you have left to load (2) only start the game after the last image has finished loading, i.e. when that counter reaches zero. To accomplish this, in the game's window.onload function, rather than immediately calling setInterval(), move that function call

into a new loadingDoneSoStartGame() function. Call that new function when that the last image finishes loading.

Instead of starting the move and draw intervals when the program starts, instead use the initialization code to begin loading the images. Each time an image finishes loading it will call a shared onload function to decrease a counter of images there are to load. When the images-to-load counter hits zero that means the last image finished loading, no matter what order the images finished loading in. At that time call the new loadingDoneSoStartGame() function to kick off the game's regular action and graphics updates, with confidence that all images must be available for the code by that point.

Scattering the image declarations across Car.js and, Track.js away from their counter and onload code, seems like a mess. Create another file in the project to keep all images under one roof. ImageLoading.js seems like a fitting name. Add a line to the main HTML to load this file alongside the others, so that its image variables will be available for the Car and Track code.

Move the carPic declaration to the top of the ImageLoading.js file. Right next to it, add two more declarations: trackPicRoad and trackPicWall. Get rid of the carPicLoaded variable, both at its point of declaration and also where it gets checked in the car's draw function. It's made unnecessary by the counter that all images will share. Prepare a variable named picsToLoad at the top of ImageLoading.js. Set picsToLoad to 3 to account for the carPic image plus the two wall images.

Write a function, countLoadedImageAndLaunchIfReady(), that decreases picsToLoad by 1. If picsToLoad equals 0 after that subtraction, call the loadingDoneSoStartGame() function which contains the initialization and setInterval() code that gets moveEverything() and drawEverything() running.

That loadingDoneSoStartGame() function can be connected as the .onload() to be called after each image finishes.

In ImageLoading.js add a function named loadImages() that sets the .onload() function for each image and, on the next line, sets the .src for each image file name.

```
function loadImages() {  
    carPic.onload=countLoadedImageAndLaunchIfReady;  
    carPic.src="player1.png";
```

Do the same for the two track image files, too.

Speaking of the two track images, you'll also need to change the drawTracks() function in Tracks.js to display them, instead of drawing blue squares. Previously you only calculated values for trackLeftEdgeX and trackTopEdgeY when the tile checked had a wall tile present because the space was left blank for road tiles. Now that both road and wall tiles now display an image, move those corner variables and their calculations outside of, and before, the isWallAtTileCoord() if-condition. When the if-test detects a wall, display the wall tile image in that spot. In the else case draw the road tile image.

You have drawBitmapCenteredAtLocationWithRotation() from displaying the car, but it supports centering and rotation, neither of which the wall tiles need. For drawing track sections you can skip the five instruction rotated bitmap draw function,

instead borrowing the single line from it that positions and draws an image by its corner, adjusted for use here like so:

```
if( isWallAtTileCoord(eachCol, eachRow) ) {  
    canvasContext.drawImage(trackPicWall,trackLeftEdgeX, trackTopEdgeY);  
} else {  
    canvasContext.drawImage(trackPicRoad,trackLeftEdgeX, trackTopEdgeY);  
}
```

Lastly, there's a few minor cleanup fixes to do. Since the track is now covering the entire playable and viewable area every update cycle you no longer need to manually black out the whole canvas with that black filled rectangle at the start of each frame. In other words the `colorRect()` at the top of `drawEverything()` can be safely removed. The actual effect on performance may not be significant, but why redraw every pixel twice on the screen twice if there's no reason to?

Additionally, the `TRACK_GAP` const value set near the top of `Track.js` will no longer be needed. That was used to provide visual spacing between tiles when drawing the wall as filled rectangles. Now that each tile is stamping out a copy of the full 40x40 image there's no need to draw borders from the code. For any types of tiles where you wish to show borders you can specify it within the tile image itself, opening up a lot easier control over color, thickness, and other style.

EXPECTED RESULT

The road and walls should appear as tiled graphics, instead of flat blue squares for walls and black for track.

Controls and functionality should work the same as before.

racing step 19

reduce risk for bugs in loading art

Currently if you want to add more images to the program you also have to adjust picsToLoad to reflect that you're waiting for an extra image to finish loading. If you forget to do that the game may start before the last image finishes. Or, even worse, if you remove an image from the game and forget to update that value, the game simply won't start, since its countdown will never reach zero.

Instead of manually setting that value, you can make it one less thing to worry about by starting it at zero and creating a function that increases it by one for each image set to load.

All of these changes will take place in the ImageLoading.js file.

Define a new helper function beginLoadingImage(). It will need two arguments: first which image variable to set up, and second which filename to load into it. Have it set the .src for the image variable to the filename from the other argument. Now for each pair of lines that look like this in loadImages():

```
carPic.onload=countLoadedImageAndLaunchIfReady;  
carPic.src= "player1.png";
```

Replace each of those two lines with one line in this form:

```
beginLoadingImage(carPic, "player1.png");
```

This new helper can also account for the .onload function connection used to tally each image when it finishes loading:

```
function beginLoadingImage(imgVar, fileName) {  
    imgVar.onload=countLoadedImageAndLaunchIfReady;  
    imgVar.src=fileName;  
}
```

So far all that helps is that you're less likely to forget to set up an image's onload correctly. That will be nice when loading more images, but it doesn't fix the main issue yet.

Set picsToLoad to zero where it's declared, instead of hard coding it to three. Then add one more line to the top of beginLoadingImage() to increase picsToLoad by one:

```
function beginLoadingImage(imgVar,fileName) {  
    picsToLoad++;  
    imgVar.onload=countLoadedImagesAndLaunchIfReady;  
    imgVar.src="images/"+fileName;  
}
```

But, since countLoadedImagesAndLaunchIfReady() subtracts one from it, won't this just instantly flip picsToLoad back and forth between 1 and then immediately back to 0 for each image? Usually that won't be the case. It's important to recognize that the .onload function assigned to an image is not called immediately, but rather only after the image has completed loading. In programmer jargon, the .onload function is set to happen asynchronously, in the background, or in a separate thread. Your code execution doesn't wait up for it.

If you call beginLoadingImage() 15 times for 15 images, the game won't start until the onload functions have brought that number back down 15 times. Or 100 times for 100 images.

Take a pause to be sure you have a good sense for why that ought to work before reading on. There's a more robust but less straightforward way to do it that builds on the same idea.

FIXING THE RACE CONDITION

While the above solution will work most of the time, there's a subtle risk hidden in using it. With regard to whether the

`picsToLoad` value might toggle rapidly between 1 and 0 I said that won't *usually* be the case. The "usually" qualifier depends on how quickly each image loads. For an image that is tiny and loaded from the local drive, as happens here for testing the game, there's some chance that the image can load and call its `onload` function before the `picsToLoad` gets increased for another image just a few lines later in your game code.

Sometimes that may happen. Sometimes it may not. Even worse: whether or not that happens can vary due to which computer it's running on. A very fast computer or very fast broadband connection is more likely to finish loading the image incredibly (problematically) quickly than a little older machine or a slower internet connection. This is called a "race condition" since the result is conditional on which of two asynchronous (not waiting up for one another) threads happens to win the race and finish computation first. Race conditions can be tricky to catch, tricky to fix, and tricky to verify have been fixed, due to their innate inconsistency.

Just what would happen if each image managed to load super quickly before the `picsToLoad` variable can get raised? The `countLoadedImageAndLaunchIfReady()` function calls `loadingDoneSoStartGame()` if it detects that `picsToLoad` has reached 0, which would happen multiple times if the images loading quickly caused `picsToLoad` to flip back to 0 each time after it hits 1. Since that function is what sets up the game's frame rate logic clock that could then happen multiple times, meaning that instead of updating the car movement based on

its speed 30 times per second it could be happening two, three, four, or more times than that.

To protect the program from that madness you could change how countLoadedImageAndLaunchIfReady() works to ensure that if it has already been called before then it shouldn't set up another update timer. Instead though, let's find a better way to get a count of how many images need to be loaded, such that the target number can be set up to its max automatically prior to any of the images having a chance to start loading.

The core of this fix is that instead of letting each image begin loading right around the time when it gets counted, populate an array with the data needed for loading images. Before sending that data to the loading function set picsToLoad to the number of elements in the array:

```
function loadImages() {
    var imageList = [
        {varName:carPic, theFile:"player1.png"},
        {varName:trackPicRoad, theFile:"track_road.png"},
        {varName:trackPicWall, theFile:"track_wall.png"}
    ];

    picsToLoad = imageList.length; // sets it to 3, since 3 Object Literals in array

    for(var i=0;i<imageList.length;i++) {
        beginLoadingImage(imageList[i].varName,imageList[i].theFile);
    }
}
```

That imageList variable is an array in which each value it contains is an Object Literal in JavaScript. In short, an Object Literal is just a convenient way to group multiple variables together with descriptive labels. Here each Object Literal has two values, a varName and theFile, but those labels are up to us. Pay close attention here to where the two [brackets] are

as opposed to where each of the { braces } are. The brackets mark the start and end of the array list, which here is getting three elements. Each of those three elements is a separate Object Literal that is pairing up the image's variable with its matching file name.

Since that will be the new way to set `picsToLoad`, comment out or remove `picsToLoad` from the `beginLoadingImage` function:

```
function beginLoadingImage(imgVar,fileName) {
    // picsToLoad++; // led to race condition, don't want to do it that way, after all!
    imgVar.onload=countLoadedImagesAndLaunchIfReady;
    imgVar.src="images/"+fileName;
}
```

Now when you'd like to load another image you can just add another line to the `imageList` array, and not worry about whether you remember to connect it to the `.onload` function or include it in the count of images to wait for before starting.

(In `countLoadedImagesAndLaunchIfReady()` the `picsToLoad` value is only used to start the game when the last image is loaded. However, you could use that number there, along with a stored separate value for how many images you load total, to update some kind of loading progress to the player. You could update text showing “3 of 8 images loaded” or calculate the percentage – even use the values to draw a progress bar with filled rectangles. Remember, though, that the loading time will happen far too fast to see, unless your files are online and you have more, larger images or a low bandwidth connection.)

EXPECTED RESULT

No change to functionality.

racing step 20 organize files into folders by type

As another organizational improvement step, before dropping in more track images for use as decorations, go ahead and separate all JavaScript code files from the image files. Having a bunch of both thrown together in the same directory with the main HTML file can make it increasingly hard to keep straight which have been accounted for.

Create two folders in the directory of the HTML file, one named “images” and the other named “js” for JavaScript. All the image files go in the images folder. Move all the JavaScript files into the js folder.

Next, fix the references to these files. Each inclusion of a JS file in the main HTML needs to have “js/” added before the filename to indicate its relative path:

```
<script src="js/GraphicsCommon.js"></script>
```

In ImageLoading.js there's no need to add the images directory to the start of each filename passed into beginLoadingImage – although that would work. A better place to inject it is directly into the src assignment line:

```
imgVar.src="images/" +fileName;
```

EXPECTED RESULT

No change to gameplay.

racing step 2! add three kinds of decorative tiles

To add some visual variety to the map add 3 more 40x40 tile images to the images folder. These will mostly work like walls, but will look different for the sake of decoration. The example versions I prepared and packed with the sample code for this step are:

- track_goal.png – black and white checkered strip, which will not only look different but will also signify the end of the track
- track_tree.png – a group of trees and grass, meant for areas beyond track walls, to better accentuate the boundaries
- track_flag.png – a wall section with a flag on it, which can be used to decorate the inside track tip in sharp corners

In ImageLoading.js declare three corresponding new image variables:

```
var trackPicGoal=document.createElement("img");
var trackPicTree=document.createElement("img");
var trackPicFlag=document.createElement("img");
```

In loadImages() add three new lines to the imageList array to load each of the new PNG files into its related image variable.

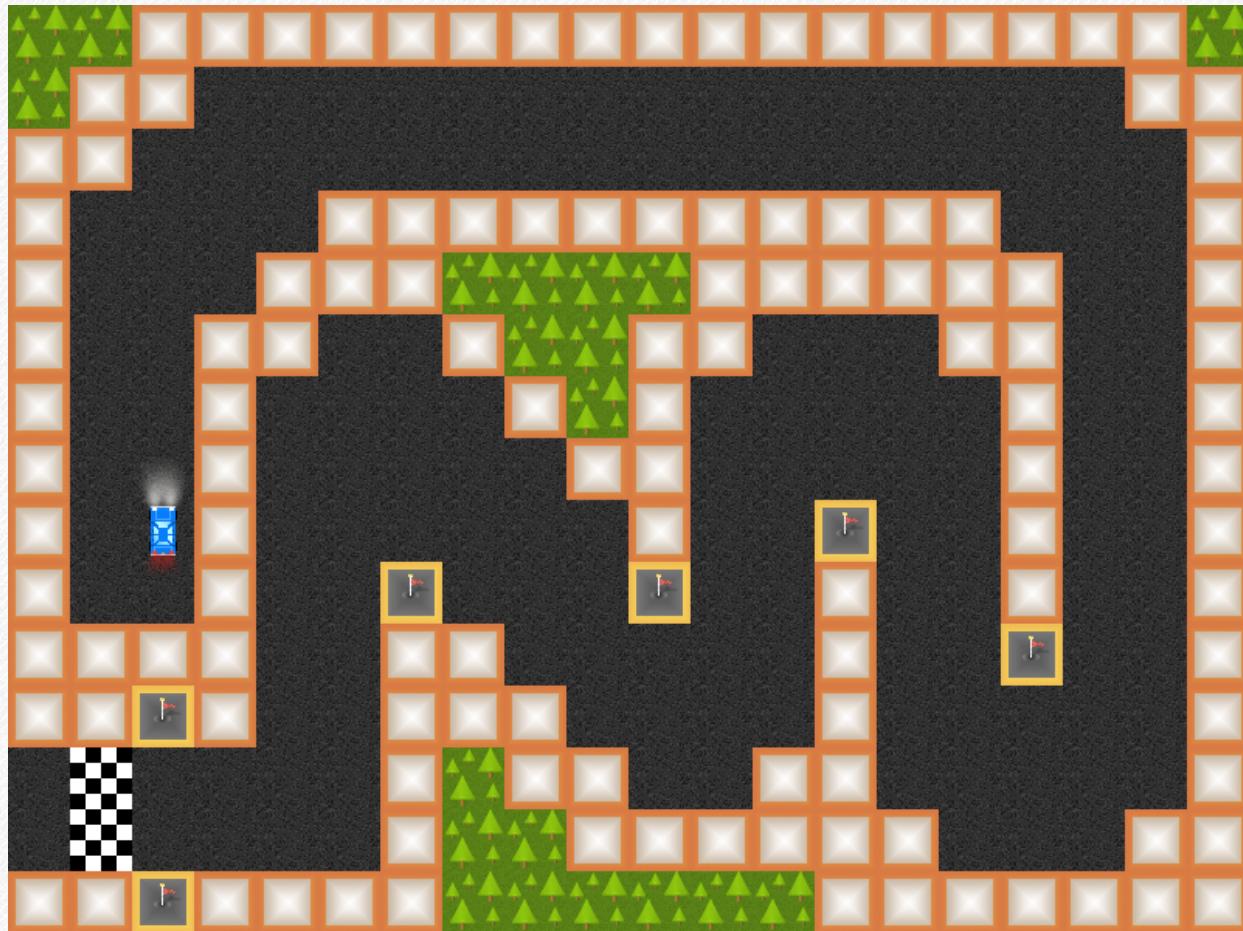
Next, in Track.js, set up three more const values as signs for each of these new tile types to be placed in the track grid:

```
const TRACK_GOAL = 3;
const TRACK_TREE = 4;
const TRACK_FLAG = 5;
```

Update the track definition in Track.js to incorporate these new tile types:



After this step that will produce a track layout like this:



The `drawTracks()` function responsible for displaying all tiles currently only detects the difference between the wall and road tiles, since that's the only thing that `isWallAtTileCoord()`

checks for. To make the track drawing code more versatile, instead of checking "is this tile a Wall" update the code to fetch "which number is at this tile position" and use that to decide in drawTracks() which image to draw.

Earlier you wrote trackTileToIndex() to compute the array index for any column and row. That index can be used to check the tile type in trackGrid at a given location. In drawTracks() right after trackLeftEdgeX and trackTopEdgeY are calculated, add a switch-case statement to translate between the track tile type and the appropriate image:

```
var trackIndex = trackTileToIndex(eachCol, eachRow);
var trackTypeHere = trackGrid[ trackIndex ];
var useImg;

switch( trackTypeHere ) {
  case TRACK_ROAD:
    useImg = trackPicRoad;
    break;
  case TRACK_WALL:
    useImg = trackPicWall;
    break;
  case TRACK_GOAL:
    useImg = trackPicGoal;
    break;
  case TRACK_TREE:
    useImg = trackPicTree;
    break;
  case TRACK_FLAG:
    default:
      useImg = trackPicFlag;
      break;
}
canvasContext.drawImage(useImg,trackLeftEdgeX, trackTopEdgeY);
```

There's a more compact way to do what the above block of code accomplishes. One thing at a time. I'll show how to do that shortly, but first wanted to demonstrate the longer and more explicit version that the other approach improves upon.

EXPECTED RESULT

Decorative track, complete with trees, flags, and a goal line.

The goal tiles currently block the car from driving onto or beyond them. That won't be a problem as you'll later use collision with that tile type to signal race completion.

racing step 22

organize track art into an array

The last part of the previous feature – the switch-case statement – makes logical sense but is prone to errors. Forgetting a break statement or mismatching a track const to an image variable could lead to a confusing bug, and each time you add another tile type you'd have to fiddle with this block of code again. Since there's a one-to-one mapping between the track const numerical codes and the track images, a way to simplify this block of code is to use an array of images for the track tiles instead of separate variables.

Instead of separately declaring trackPicFlag, trackPicTree, and so on, at the top of ImageLoading.js, after declaring carPic, make an empty array to contain all track tile images:

```
var trackPics = [];
```

The idea here is that you'll use the numerical code for each tile type as the index in the trackPics array for the corresponding image. For example, trackPics[TRACK_ROAD] will be loaded with the track_road.png file. This means in drawTracks within Track.js you can replace that long switch-case statement with the following:

```
var trackIndex = trackTileToIndex(eachCol, eachRow);
var trackTypeHere = trackGrid[trackIndex];
// removed useImg variable and the switch-case statement
canvasContext.drawImage(trackPics[trackTypeHere], trackLeftEdgeX, trackTopEdgeY);
```

There's a slight catch to handling the track images this way: whereas you can initialize carPic at the top of loadImages right where it's declared, that approach isn't an option for the track

art array. Instead, write a function to prepare each position in the array to first initialize it to an “img” document element:

```
function loadImageForTrackCode(trackCode, fileName) {  
    trackPics[trackCode] = document.createElement("img");  
    beginLoadingImage(trackPics[trackCode],fileName);  
}
```

This takes advantage of the existing beginLoadingImage() function. Each of the track images can now be loaded in loadImages() with a line like:

```
loadImageForTrackCode(TRACK_ROAD,"track_road.png");
```

The only question remaining is how to get that function to fit smoothly with the imageList array of Object Literals. It wants different input values than beginLoadingImage(). The solution here is to set up the Object Literal differently for graphics used as track tiles. Instead of providing a variable name connected to the label varName, each track type can be given instead through a different label, trackType:

```
function loadImages() {  
    var imageList = [  
        {varName:carPic, theFile:"player1.png"},  
  
        {trackType:TRACK_ROAD, theFile:"track_road.png"},  
        {trackType:TRACK_WALL, theFile:"track_wall.png"},  
        {trackType:TRACK_GOAL, theFile:"track_goal.png"},  
        {trackType:TRACK_TREE, theFile:"track_tree.png"},  
        {trackType:TRACK_FLAG, theFile:"track_flag.png"}  
    ];  
  
    picsToLoad = imageList.length;  
  
    for(var i=0;i<imageList.length;i++) {  
        if(imageList[i].trackType != undefined) {  
            loadImageForTrackCode(imageList[i].trackType, imageList[i].theFile);  
        } else {  
            beginLoadingImage(imageList[i].varName, imageList[i].theFile);  
        } // end of else  
    } // end of for imageList  
} // end of function loadImages
```

As shown in that code snippet the loop through all image data can then just check whether each element in the list has its trackType defined. If one doesn't, then it looks for varName instead. This way the image loading system will be equally able to have new graphics added no matter whether a new image is part of the track tile list or some other standalone image (player 2 car, logo image, interface element, etc.).

Note that the gap in the code between the carPic Object Literal in imageList and the track elements is purely there for us as programmers reading the code. I skipped a line there to emphasize that above the gap I'm doing something a bit different than in the group below it. This is to reduce the likelihood of a copy and paste error happening when adding other graphics in later, or missing that now all of the Object Literals share the same value labels. The code doesn't mind if the two types of image data are mixed together every other line, the grouping is also there only for ease of readability.

EXPECTED RESULT

No change to play experience.

racing step 23 reduce calculations for track draw

While still focusing on the `drawTracks()` function, there are a number of ways that it can be further optimized.

As a rule of thumb it's important to not get too caught up in optimizing code for its own sake unless it has first been measured and proven as the cause of bad performance. On a modern processor many calculations that would seem complex or time consuming if done by hand can be solved virtually instantaneously.

In the case of tile map rendering, the difference between a little optimization effort or a clumsy inefficient approach comes with a big multiplier. Since the track is 20x15 tiles, any work the processor does per tile happens 300 times each frame, or at about 30 frames per second... 9,000 times per second! Anything happening in code so many times per second is at least a candidate for thinking about whether there's a faster way to do it.

The minor optimizations you'll introduce here are all closely connected. The main idea is, instead of recomputing each tile's position values separately, you can use the ordered, consistent grid to reduce or simplify calculations. You can draw the tiles in a predictable sequence. Instead of recalculating each tile's position independently ("how far, in pixels, is 6 whole tiles?") you can increase the draw position per tile ("add one tile width to the previous draw coordinate before

displaying the next one"). The same trick can be done for incrementing the tile's vertical position and the tile index.

To do this, first swap which of the nested for-loops is outside of the other. Here's how the code looked before:

```
for(var eachCol=0; eachCol<TRACK_COLS; eachCol++) { // in each column...
    for(var eachRow=0; eachRow<TRACK_ROWS; eachRow++) { // in each row within that col
        var trackLeftEdgeX = eachCol * TRACK_W;
        var trackTopEdgeY = eachRow * TRACK_H;
        var trackIndex = trackTileToIndex(eachCol, eachRow);
        var trackTypeHere = trackGrid[ trackIndex ];
        canvasContext.drawImage(trackPics[trackTypeHere], trackLeftEdgeX, trackTopEdgeY);
```

The previous for-loop arrangement was scanning top-to-bottom first then advancing left-to-right only upon hitting the bottom of each column. Look at how track data is laid out in the trackGrid variable. It maps to the layout shown in-game, with the top row of numbers corresponding to the top tile row:

```
var trackGrid =
    [ 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4,
      4, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
```

For the code to handle consecutive track positions in the array – position 4, then 5, then 6, then 7, and so on – you should be dealing with column after column left to right and only then moving down to the the next row. Now:

```
for(var eachRow=0; eachRow<TRACK_ROWS; eachRow++) { // deal with one row at a time
    for(var eachCol=0; eachCol<TRACK_COLS; eachCol++) { // left to right in each row
```

This order deals with each map tile in its natural sequence in memory, left-to-right then top-to-bottom just like English writing. Instead of calling trackTileToIndex() to compute trackIndex every time another tile is drawn, now you can rely on their built-in order to do this:

```
var trackIndex = 0;
for(var eachRow=0; eachRow<TRACK_ROWS; eachRow++) { // deal with one row at a time
  for(var eachCol=0; eachCol<TRACK_COLS; eachCol++) { // left to right in each row
    // check trackIndex here to draw the tile
    trackIndex++; // increments to look at next tile!
```

Similarly, instead of using multiplication to separately compute the upper left corner of every image, you can take advantage of knowing that every tile is exactly TRACK_W pixels right of the previously handled one. Add that dimension to a cumulative trackLeftEdgeX variable after each tile in a row. Reset trackLeftEdgeX to zero and increase trackTopEdgeY by TRACK_H when advancing to the next row:

```
function drawTracks() {
  var trackIndex = 0;
  var trackLeftEdgeX = 0;
  var trackTopEdgeY = 0;

  for(var eachRow=0; eachRow<TRACK_ROWS; eachRow++) { // deal with one row at a time

    trackLeftEdgeX = 0; // resetting horizontal draw position for tiles to left edge

    for(var eachCol=0; eachCol<TRACK_COLS; eachCol++) { // left to right in each row

      var trackTypeHere = trackGrid[trackIndex]; // getting the track code here
      canvasContext.drawImage(trackPics[trackTypeHere], trackLeftEdgeX, trackTopEdgeY);

      trackIndex++; // increment which index we're going to next check in the track
      trackLeftEdgeX += TRACK_W; // jump horizontal draw to next tile over by tile width

    } // end of for eachCol

    trackTopEdgeY += TRACK_H; // jump horizontal draw down by one tile height

  } // end of for eachRow
} // end of drawTracks()
```

Make sure to not mix up what trackGrid[trackIndex] means compared to trackPics[trackTypeHere]. The trackIndex variable refers to which tile position in the map is checked. The trackGrid array is used to look up a track position to find which number appears at that spot (ex. 1 for TRACK_WALL).

That value gets temporarily stored as the `trackTypeHere` variable. The `trackPics` array, in contrast, matches a tile type number (`trackTypeHere`) to which picture should be displayed. You could technically cut out that intermediary `trackTypeHere` variable and instead write this eye sore:

```
canvasContext.drawImage(trackPics[trackGrid[trackIndex]], trackLeftEdgeX, trackTopEdgeY);
```

Even though that gets rid of one intermediary variable, it's even tougher to read or make sense of. There's often a personal call to be made about where to draw the line between minor code optimizations versus code readability. In this situation I suggest prioritizing readability over having something like "`trackPics[trackGrid[trackIndex]]`" in the code.

EXPECTED RESULT

Identical gameplay behavior. The game may perform very, very slightly more smoothly, but there's no perceptible difference in the case of a game map so small with so little else happening.

Modern computers are extraordinarily fast, able to compute literally a billion or more operations per second. Even with the gains scaled by 300 tiles per frame the total difference in performance between some redundant multiplication and an extra function call compared to this approach is negligible.

If you are going to optimize before doing any performance tests then at least improving a section in code that runs many, many times each frame is a reasonably smart place to look.

racing step 24

create a class for car code reuse

So far you built this game – and the whole starter code for the simpler games – without needing to program with "classes." Here and now you've reached a point where figuring that out will have real utility. This step isn't about cleaning up or organizing the code for its own sake. Wrapping the car in a class will make it much easier to support a second player.

This fits the situation here because the additional cars will move, look, and function much like the first one. (For anyone coming into HTML5/JS from another programming language, just to be clear the separate JS files have not been forcing or automating separated class definitions. The files got stitched together into a big global chunk of variables and functions.)

You could support extra vehicles the hard way by copy and pasting, then tweaking a duplicated version of every variable and function that's currently used for the car. To do it that way you'd first copy Car.js to create Car2.js, then rename every variable in Car2.js from names like carX to car2X, and every function like carMove() to car2Move(). The downside to doing it that way is that for any improvement, change, or fix to the car behavior, you'd have to manually keep both chunks of similar code tightly in sync with one another.

Maintaining that coupling would very likely involve more error-prone copy and pasting. With the car variables in global scope (i.e. accessible anywhere in the code) the chance of missing a change between car2Speed and carSpeed etc. while you work

on the game is extraordinarily high. That leads to wasted time fixing bugs that can be avoided altogether. All that trouble is if you only ever need two cars, but imagine the inevitable knot of trying to later support four or more cars, with some driven by computer control.

The flexible solution to this kind of problem is to wrap up the car's variables and functionality into a class. The class acts as a shared blueprint for however many cars you wish to create and manage in your code. Each car will be an instance of the new class, all sharing the exact same functionality but using their own unique copy of the same variable definitions. Making this change won't require a ton of new code nor a major rearrangement of what's already there. It will require changing to a slightly different pattern for how the variables in Car.js get used, and how car functions get called from the other files.

The first thing to do is bump the Car.js const definitions to the top of the file, where they won't be part of the Car class definition. The const values won't differ per car. Below the last Car const value, but before setting up the carX variable, insert this new line to distinguish the start of this class:

```
function carClass() {
```

Before there's time to forget, add a closing brace } at the very end of the file to close the class definition.

Starting a class with “function” will look strange if you've seen other programming languages such C++, C#, ActionScript 3, or Java that have a keyword 'class' to use. In JavaScript a

class gets created as a function that contains its own set of variables and other functions.

For every variable within the class function, replace "var " with "this." (with the period after it) – the meaning of which I'll explain in the audio intermission later, but for now it's going to keep the code easier to work with. What used to be this code:

```
var carX = 75, carY = 75;  
var carSpeed = 0;  
var carAng = -0.5 * Math.PI;
```

Now becomes:

```
function carClass() {  
    this.carX = 75;  
    this.carY = 75;  
    this.carSpeed = 0;  
    this.carAng = -0.5 * Math.PI;
```

Functions within the class will be similarly affected, both in definition and where they get called. The initialization function next in the code will go from looking like this:

```
function carInit() {  
    carReset();  
}
```

...to this form, updated in the same place:

```
this.carInit = function() {  
    this.carReset();  
}
```

Follow the same pattern for the other functions, remembering to insert the "this." prefix before any of the class variables or functions. The next function, as another example, will be:

```
this.carReset = function() {
    for(var i=0; i<trackGrid.length; i++) {
        if( trackGrid[i] == TRACK_PLAYER) {
            var tileRow = Math.floor(i/TRACK_COLS);
            var tileCol = i%TRACK_COLS;
            this.carX = tileCol * TRACK_W + 0.5*TRACK_W;
            this.carY = tileRow * TRACK_H + 0.5*TRACK_H;
            trackGrid[i] = TRACK_ROAD;
            break; // found it, so no need to keep searching
        }
    }
}
```

Notice though that the "var" keyword is still being used here for the for-loop counter, as well as tileRow and tileCol. These are strictly local variables that can be forgotten when the function that they're declared in ends.

Further down in the file, as part of the carMove() function, var will appear in this usage too since its values are likewise only needed inside the function (what does change, is the addition of "this." to the car's position, angle, and speed variables):

```
var nextX = this.carX + Math.cos(this.carAng) * this.carSpeed;
```

It is a good idea for readability, though not technically necessary for your code to run, to indent one extra time all the code lines between the carClass() function's opening and closing brace. This can help us keep straight at each line of code how many braces have been opened but not yet closed. Most text editors intended for code have a shortcut key to simplify this. In Sublime Text, select all the lines you wish to indent, then press Tab (or Shift+Tab to go backward). In TextWrangler, use Command plus right bracket:]

Aside from Car.js, the other section of code that requires changes is Main.js where the car functions are called. Since the car's functions have to be called on an instance of the car

rather than the class itself, first set up a new variable near the top of Main.js specifying what label you'll use for this instance of the car (p1 is short for "player 1"):

```
var p1 = new carClass();
```

The “new” keyword is used when creating a new instance of a class. It’s necessary since a class is a more complex set of information, strewn across multiple variables and functions.

Unlike any code within functions, which waits until the full page loads before any of it gets called from window.onload or the functions it spins off with setInterval(), because that p1 var line is in top-level scope it will evaluate immediately when the code reaches that point. For this reason the browser needs to have already seen the definition for carClass(), so make sure the import line for Car.js is above/before Main.js in the HTML file. (If you set p1 to new carClass() in window.onload instead, then it wouldn’t matter if Car.js was listed after the Main.js file.)

Each of the car functions called – carInit(), carMove(), and carDraw() – need to be prefixed with the name of the instance on which they are called. “Instances” can seem abstract to people new to programming, but all it’s doing is specifying which car the function is being called on. This, for example, is the changed line 21 of Main.js above initInput():

```
p1.carInit();
```

In the same fashion, carMove() in Main.js will need to become p1.carMove() and carDraw() will become p1.carDraw().

There isn’t yet support for having a second car look or control differently from the first player’s car. However, having wrapped

the car variables and functions into a class means adding another car will be as simple as separating controls and appearances so they're per-car. You'll be doing just that in the next few steps.

EXPECTED RESULT

No change. Car should still drive as it did before.

racing step 25

preparing key input for player two

To get both cars driving you'll need to connect movement for each to a different set of keys.

The keyHeld_Gas, keyHeld_Reverse, keyHeld_TurnLeft, and keyHeld_TurnRight variables will move from Input.js to the car class. Replace the "var " in their declarations with "this."

```
function carClass() {  
    // variables to keep track of car position  
    // (...skipping over those variables...)  
  
    // keyboard hold state variables, to use keys more like buttons  
    this.keyHeld_Gas = false;  
    this.keyHeld_Reverse = false;  
    this.keyHeld_TurnLeft = false;  
    this.keyHeld_TurnRight = false;
```

Add the "this." prefix before each use of the four keyHeld_* variables in the carMove() function's if-statements.

```
if(Math.abs(this.carSpeed) > MIN_TURN_SPEED) {  
    if(this.keyHeld_TurnLeft) {  
        this.carAng -= TURN_RATE*Math.PI;  
    }  
  
    if(this.keyHeld_TurnRight) {  
        this.carAng += TURN_RATE*Math.PI;  
    }  
}  
  
if(this.keyHeld_Gas) {  
    this.carSpeed += DRIVE_POWER;  
}  
if(this.keyHeld_Reverse) {  
    this.carSpeed -= REVERSE_POWER;  
}
```

A couple of lines after the fourth this.keyHeld_* variable in Car.js gets declared, add a function named setupControls() which accepts four arguments telling the car which keycode

inputs it needs to listen for. Player 1 will get codes for the arrow keys, and Player 2 will get codes for the WASD keys.

```
this.setupControls = function(forwardKey, backKey, leftKey, rightKey) {  
    this.controlKeyForGas = forwardKey;  
    this.controlKeyForReverse = backKey;  
    this.controlKeyForTurnLeft = leftKey;  
    this.controlKeyForTurnRight = rightKey;  
}
```

Here's something different: the controlKeyForGas and related variables above aren't explicitly declared in carClass(). They're simply set as though they already existed. The very act of setting the variable in JavaScript creates it.

This is related to why I'm showing these variables with the "this." prefix. Using the "var" keyword before those values in this.setupControls() would cause the computer to forget those variables as soon as this.setupControls() finished. That won't work because other functions need those codes later. If we used "var" before variables in this.setupControls(), the computer would forget them after this.setupControls().

If the input-related variables here were global the second car's key setup would overwrite the same key variables of the first car's set up. The "this." keyword here connects the variables to each car. (I'll explain other differences between "this." and "var " in the audio intermission.)

At the end of initInput() set up player one's setupControls():

```
p1.setupControls(KEY_UP_ARROW, KEY_DOWN_ARROW, KEY_LEFT_ARROW, KEY_RIGHT_ARROW);
```

In the Input.js file, update the setKeyHoldState() function to accept a new third argument indicating which car you're

checking and setting key information for. This way the button-like hold state functionality can be done per each driver:

```
function setKeyHoldState(thisKey, thisCar, setTo) {
```

In there, change each key from its older hardcoded form, of...

```
if(thisKey == KEY_LEFT_ARROW) {  
    keyHeld_TurnLeft = setTo;  
}
```

...to a version that instead uses the given car's key variables – in other words, those specifically for thisCar (notice that it's in both the if-statement as well as in the key assignment line):

```
if(thisKey == thisCar.controlKeyForTurnLeft) {  
    thisCar.keyHeld_TurnLeft = setTo;  
}
```

The other three keys for right, forward, and reverse need a similar update. In the calls to setKeyHoldState() in keyPressed() and keyReleased(), add p1 (the instance of the first player's car) as a middle argument:

```
setKeyHoldState(evt.keyCode, p1, true); // the form used in keyPressed(evt)  
setKeyHoldState(evt.keyCode, p1, false); // the form used in keyReleased(evt)
```

To be better set up for binding keys to the upcoming Player 2 car, declare and set 4 new constants at the top of Input.js as KEY_LETTER_W, KEY_LETTER_A, KEY_LETTER_S, and KEY_LETTER_D. Const makes more sense here than var since they'll never change during play. As a hint to get you started, KEY_LETTER_W should be set to 87.

If you still have Racing step 11 handy (or my example version) you can run it to determine new key code constants for the WASD keys. Otherwise you could temporarily re-add this line

to the keyReleased(evt) function in Input.js and watch below the game canvas after pressing W, A, S, and D to check key code values for each of those letters:

```
document.getElementById("debugText").innerHTML = "KeyCode Released: " + evt.keyCode;
```

EXPECTED RESULT

Still no change. Even if everything looks and plays fine on the surface, since you've been making changes across many files, it would be prudent to run the game with the error Console showing. Make sure that you're not accumulate subtle errors in your code that might cause bigger confusion later.

racing step 2b adding player two's car

In Main.js mimic the pattern for each appearance of p1 with a matching use of p2. On the line after var p1 = new carClass(); introduce:

```
var p2 = new carClass();
```

On the line after p1.carInit(); introduce:

```
p2.carInit();
```

The same goes for p1.carMove(); and p1.carDraw(); – add lines after each with p2. To support more, or a variable number of, cars you could replace these two instances with a for-loop that places each car in an array. As a first pass into multiplayer functionality though it'll be no big deal to keep these instances separate.

In Track.js add a second car start position adjacent to the original one:

```
var trackGrid =
[ 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4,
  4, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
  1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
  1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1,
  1, 0, 0, 0, 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1,
  1, 0, 0, 1, 1, 0, 0, 1, 4, 4, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1,
  1, 0, 0, 1, 0, 0, 0, 1, 4, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
  1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
  1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,
  1, 2, 2, 1, 0, 0, 0, 0, 0, 1, 0, 0, 5, 0, 0, 1, 0, 0, 1, /// added another car
  1, 0, 0, 1, 0, 0, 5, 0, 0, 0, 5, 0, 0, 1, 0, 0, 1, 0, 0, 1,
  1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 5, 0, 0, 1,
  1, 1, 5, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
  0, 3, 0, 0, 0, 0, 1, 4, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1,
  0, 3, 0, 0, 0, 0, 1, 4, 4, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1,
  1, 1, 5, 1, 1, 1, 1, 4, 4, 4, 4, 1, 1, 1, 1, 1, 1, 1, 1];
```

Both cars won't start in the same position since in the Car.js carReset() function, the first TRACK_PLAYER tile spotted is promptly overwritten as TRACK_ROAD making it look and behave like normal road during play. After the first car is placed, there's one less TRACK_PLAYER (number 2) in the track grid. The second time carReset() runs, it will find the remaining TRACK_PLAYER in the grid, place the second car there, and replace that tile with TRACK_ROAD too.

The one other where calls to p2 need to be added is Input.js, where setupControls() for p2 can be called on the line after p1's, setting up WASD keys to act as arrows for Player 2:

```
p2.setupControls(KEY_LETTER_W,KEY_LETTER_S,KEY_LETTER_A,KEY_LETTER_D);
```

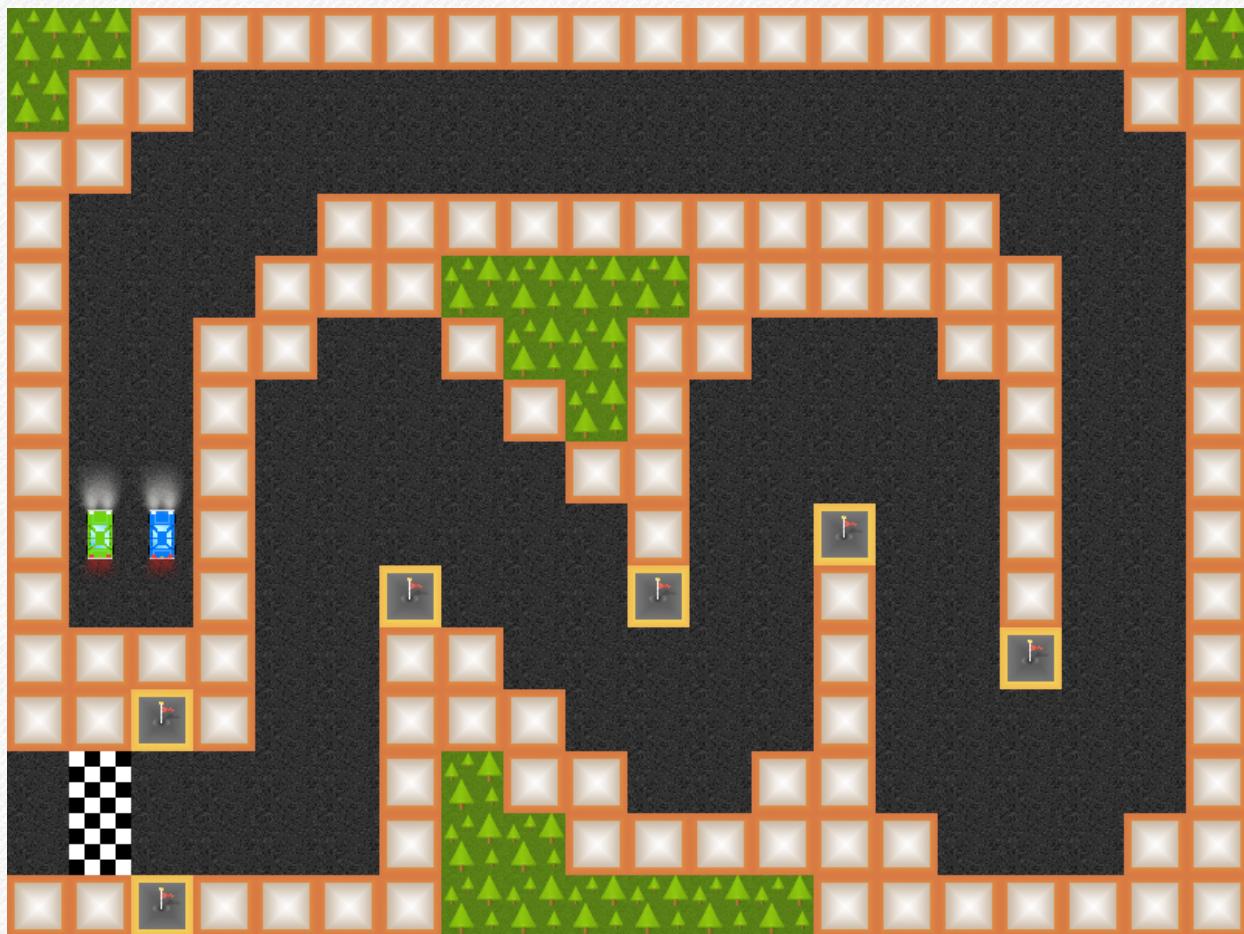
Down in keyPressed() and keyReleased() duplicate each setKeyHoldState() call to deal with the same true or false hold states with the p2 car.

EXPECTED RESULT

There should now be a second car identical to the original. The WASD keys will control it independently. The arrows will still drive the other car.

racing step 27 different graphic for player two

The cars are currently identical. Giving each car a different visual representation will not only look nicer, this also makes a meaningful improvement to playability by reducing confusion among players when the cars are near one another.



Open the player1.png in the images folder with Photoshop, Paint.NET, the editor from Gimp.org, or a similar 2D program that supports partial transparency for PNG files. Recolor the car green, then save this new version with a different filename: player2.png. if you don't feel like messing with image files, you can use the player2.png file provided with the example solution for this step). In the ImageLoading.js file, duplicate the

carPic line to create a similar set up for car2Pic. In loadImages() load the player2.png file into car2Pic, using the same pattern as was used to get player 1's image working.

The car class needs a small tweak to store and use which image it's associated with. Because every car must have its car graphic set, make this part of the initialization function:

```
this.carInit = function(whichGraphic) {  
    this.myBitmap = whichGraphic;  
    this.carReset();  
}
```

Notice that you can still have function arguments in this form. Instead of "function carInit(whichGraphic) {" it is written as "this.carInit = function(whichGraphic) {" like we just did here. It's used for p1, for example, as p1.carInit(carPic).

In carDraw(), take out carPic and in its place put this.myBitmap to use the car's saved image reference.

In Main.js's loadingDoneSoStartGame(), pass each carInit() a reference to one of the two car images:

```
p2.carInit(car2Pic);  
p1.carInit(carPic);
```

I switched the order of the p2 and p1 lines because carReset(), called from carInit(), searches left-to-right, top-to-bottom for the first unused player start location it finds. This ordering ensures that the player two car will start in the position left of the player one car. Since player two drives with the WASD keys on the left side of the keyboard, this is a natural mapping.

EXPECTED RESULT

The left car, driven with the WASD keys, is now green.

racing step 28

respond to crossing the finish line

There's one more chunk of functionality to get the starter code for this Racing game wrapped up and ready to use for the Section 2 exercises. It's going to take a few steps to handle this without trying to take on too much at once.

The finish line now works the same as a track wall. Getting it working is important for even the most basic racing gameplay. A more robust finish line would support multiple laps, but first you can have it reset both players when either crosses the finish. Additionally, it can show who won the race in the debug text below the canvas. The message can stay there for bragging rights through the whole next race until the next time either player makes it to the finish line again.

This is a pretty crudely abrupt way to handle race completion. Figuring out a good way to hook it up isn't as straightforward as the desired result. The only way a car knows whether it bumps a wall – finish line or otherwise – is through the call to `checkForTrackAtPixelCoord()` in `carMove()`. That function only distinguishes between road and non-road tiles. Additionally, as the game is currently programmed, the cars have no way to reset to their start positions, since their original start indicators (the "2" values in the `trackGrid` array) were gobbled up when the game first loaded.

You could check within `checkForTrackAtPixelCoord()` whether a player's car overlapped a finish line tile. This is problematic for multiple reasons. First, its functionality is not clearly implied

by the function's name. Second, you'd also need to pass some information about the car to the track function so the winner could be declared.

Rather than giving the track code new logic responsibilities, I suggest instead moving some of the logic out from the track code over to the car code. This is partly a subjective style choice, but it's supported by a couple rules of thumb. (1) The player car is a dynamic agent, detecting and responding to the world, whereas the road is inanimate. (2) Collisions affect what the car does, but they generally do not change the world. It makes more sense, then, for the car to check the walls and road data, than for the walls and road to check for the car.

Instead of `checkForTrackAtPixelCoord()` making a judgment on its own and returning true or false then, it can be a simpler `getTrackAtPixelCoord()` function that returns the coded index (1 meaning `TRACK_WALL`, 3 meaning `TRACK_GOAL`, etc.) and trusts the code that called it to handle each case properly.

In `checkForTrackAtPixelCoord` replace this block of code:

```
var trackIndex = trackTileToIndex(tileCol, tileRow);
return (trackGrid[trackIndex] == TRACK_ROAD);
```

...with these two lines, which aren't filtering against road type:

```
var trackIndex = trackTileToIndex(tileCol, tileRow);
return trackGrid[trackIndex];
```

To keep your function name indicative of what it does, rename `checkForTrackAtPixelCoord()` to `getTrackAtPixelCoord()`. I used my text editor's Find in Files feature at this point to double

check that it's not called or relied upon anywhere else in the code for its older checkForTrackAtPixelCoord() functionality.

You also need to rethink how you respond to an out-of-bounds request. Previously, the whole function returned either true or false to signify whether a car could safely drive over this tile position. The function defaulted to returning false for any out-of-bounds coordinate to prevent a car from driving off the canvas. Because the function returns a tile type, it doesn't make any sense to return false for out-of-bounds. Instead, if an out-of-bounds tile is requested, return TRACK_WALL so the caller will handle it as impassible territory.

```
function getTrackAtPixelCoord(pixelX,pixelY) {
  var tileCol = pixelX / TRACK_W;
  var tileRow = pixelY / TRACK_H;

  // we'll use Math.floor to round down to the nearest whole number
  tileCol = Math.floor( tileCol );
  tileRow = Math.floor( tileRow );

  // first check whether the car is within any part of the track wall
  if(tileCol < 0 || tileCol >= TRACK_COLS ||
     tileRow < 0 || tileRow >= TRACK_ROWS) {
    return TRACK_WALL; // avoid invalid array access, treat out of bounds as wall
  }

  var trackIndex = trackTileToIndex(tileCol, tileRow);
  return trackGrid[trackIndex];
}
```

Those || marks are “OR bars” in code. They string multiple comparisons into a single condition. If any of the comparisons are true, the whole condition is true. As a reminder >= is used for “greater than or equal to” comparison.

The Car.js carMove() function now has the responsibility of checking for, and handling, a match against the TRACK_ROAD and TRACK_WALL types.

Look for where car collision uses checkForTrackAtPixelCoord() and replace it with the newer name, getTrackAtPixelCoord(). Rather than a true or false variable, it now returns a number indicating which type of track tile the car's center is over. You'll need to test whether it matches TRACK_ROAD since the function won't directly yield true or false for the if-statement.

Because you'll do a similar comparison to match against TRACK_GOAL, save the result of getTrackAtPixelCoord() into a temporary local variable to avoid recalculating it again with each check. Update the debug text below the canvas when either car hits the finish line. The code at the end of carMove():

```
var nextX = this.carX + Math.cos(this.carAng) * this.carSpeed;
var nextY = this.carY + Math.sin(this.carAng) * this.carSpeed;
//// above two lines are unchanged, just showing here for placement context

var drivingIntoTileType = getTrackAtPixelCoord(nextX,nextY);

if( drivingIntoTileType == TRACK_ROAD ) {
    this.carX = nextX;
    this.carY = nextY;
} else if( drivingIntoTileType == TRACK_GOAL ) {
    document.getElementById("debugText").innerHTML = "someone hit the goal line"; //////
} else {
    this.carSpeed = 0.0;
}

//// below line is unchanged, just showing here for placement context
this.carSpeed *= GROUNDSPEED_DECAY_MULT;
```

Driving the full length of the track just to test whether this update does what it's supposed to seems unnecessary. Cheats of various kinds can be helpful for testing. In this case carving a shortcut into the map layout will be a good way to test the change. Here's where I'm putting a hole in the wall next to where the cars start:



Alternatively, you could add a few additional goal line tiles a few tiles north of the cars. Either approach will work equally well for testing, and can be undone once you're satisfied that the finish line functions as it should.

EXPECTED RESULT

Drive the car around from the start position to hit either goal tile. Upon doing so with either car the text below the game's canvas will update to indicate that the end has been reached.

racing step 29 declaring a winner by name

To print the winner, you'll need a way to distinguish the cars by name. While you could use labels like Player 1 and Player 2, it's not immediately clear which is supposed to be which.

Arrow keys are conventionally a primary control scheme, but Player 1 is by console convention the player to the left (though WASD is the left control scheme here). Numerical distinctions are fine in code if consistent, but the player won't see those.

Label the cars by their color instead.

Much as both car instances need their own image, both car instances need their own color label. It makes sense then to add the car's name to the initialization in Car.js:

```
this.carInit = function(whichGraphic,whichName) {  
    this.myBitmap = whichGraphic;  
    this.myName = whichName;  
    this.carReset();  
}
```

On line 76, where the innerHTML is set for the debugText HTML element, the car's name can be inserted, using the + operator to add the car name into a string for context:

```
document.getElementById("debugText").innerHTML = this.myName + " won the race";
```

Then add names to the cars in Main.js where carInit is called:

```
p2.carInit(car2Pic, "Green Car");  
p1.carInit(carPic, "Blue Car");
```

EXPECTED RESULT

Driving either car into the goal line will yield a message below the canvas stating which car last reached the finish line. The race still won't reset yet, though. That's happening next.

racing step 30

reset both cars when either wins

Resetting the car after either car wins the race sounds simple. Reload the map the same way it opened the first time, right? Unfortunately, the trackGrid's car start locations (2's in the grid) were replaced by 0's during the initial game load process.

Change the way carReset() works so the first time it runs, the cars save their initial positions in variables which can be restored from later (this.homeX and this.homeY). Whether or not it's the first time, copy the stored start coordinates (this.homeX, this.homeY) over the car's current coordinates (this.carX, this.CarY). Compare the this.homeX value against the "undefined" keyword to bypass the scan for car start position if home coordinates have already been stored.

```
this.carReset = function() {
    if(this.homeX == undefined) {
        for(var i=0; i<trackGrid.length; i++) {
            if( trackGrid[i] == TRACK_PLAYER) {
                var tileRow = Math.floor(i/TRACK_COLS);
                var tileCol = i%TRACK_COLS;
                this.homeX = tileCol * TRACK_W + 0.5*TRACK_W;
                this.homeY = tileRow * TRACK_H + 0.5*TRACK_H;
                trackGrid[i] = TRACK_ROAD;
                break; // found it, so no need to keep searching
            } // end of if
        } // end of for
    } // end of if car position not saved yet
    this.carX = this.homeX;
    this.carY = this.homeY;
} // end of carReset
```

Since the code was leading to a cascade of more than two consecutive closing braces, I added some simple one-line comments to mark which closing brace matches to which open brace.

To ensure this function is called when either car passes the finish line, add the following lines right after the innerHTML race winner update:

```
p1.carReset();
p2.carReset();
```

It's a sloppy way to get the desired result, but it gets the job done. Here, specific global instances of the car are being used inside the car's class definition. Even though this wouldn't be a great long-term habit, (a) I wanted to show that it's possible, and in a pinch can work, also (b) this clearly isn't the final way that level ending will get handled. This has the undesirable effect though of making the class more tangled with how it's being used from outside the file and for this specific program, making it more difficult to code to read or later repurpose.

This shortcut is merely a placeholder until a more complex solution is developed later. When writing placeholder code, the less time tied up in doing it, the better. This also keeps the small mess in one easy to clean up location. It can be removed later without really needing much untangling.

At this point the code should be ready to test. Though cars will pop back to their start positions when either hits the goal line, they won't yet reset their orientation or speed. This can be addressed by relocating the initialization of `this.carSpeed` and `this.carAng` from the top of the `car` class into the top of the `carReset()` function:

```
this.carReset = function() {
  this.carSpeed = 0;
  this.carAng = -0.5 * Math.PI;
  // ... rest of carReset follows ...
```

The initial values for those two variables may as well be removed from the car's definitions at the top of the class. It would be harmless to leave them there. However it could cause confusion later if trying to adjust the starting angle for the cars, finding then that upon changing those defaults at the top of the class don't have any effect on the game. For this reason, instead of copying the carSpeed and carAng lines totally move them to carReset() as their only set up location.

Once this change has been tested and you're satisfied that both cars reset not only in position but also in angle and speed, restore the wall at the start of the track:

```
var trackGrid =
[ 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4,
  4, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
  1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
  1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1,
  1, 0, 0, 0, 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1,
  1, 0, 0, 1, 1, 0, 0, 1, 4, 4, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1,
  1, 0, 0, 1, 0, 0, 0, 1, 4, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
  1, 0, 0, 1, 0, 0, 0, 0, 1, 4, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
  1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, ///// filled in hole
  1, 2, 2, 1, 0, 0, 0, 0, 0, 1, 0, 0, 5, 0, 0, 1, 0, 0, 1, ///// once done testing
  1, 0, 0, 1, 0, 0, 5, 0, 0, 0, 5, 0, 0, 1, 0, 0, 1, 0, 0, 1, ///// goal line support
  1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 5, 0, 0, 1,
  1, 1, 5, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
  0, 3, 0, 0, 0, 0, 1, 4, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1,
  0, 3, 0, 0, 0, 0, 1, 4, 4, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1,
  1, 1, 5, 1, 1, 1, 4, 4, 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
```

EXPECTED RESULT

When either car wins the race both get reset to their start positions, facing up and with zero speed. The winner is still mentioned by name in the debugText below the canvas.

racing step 3! cleaning out the update markings

For this step in the example solutions I'm removing all the line change //// comment markings.

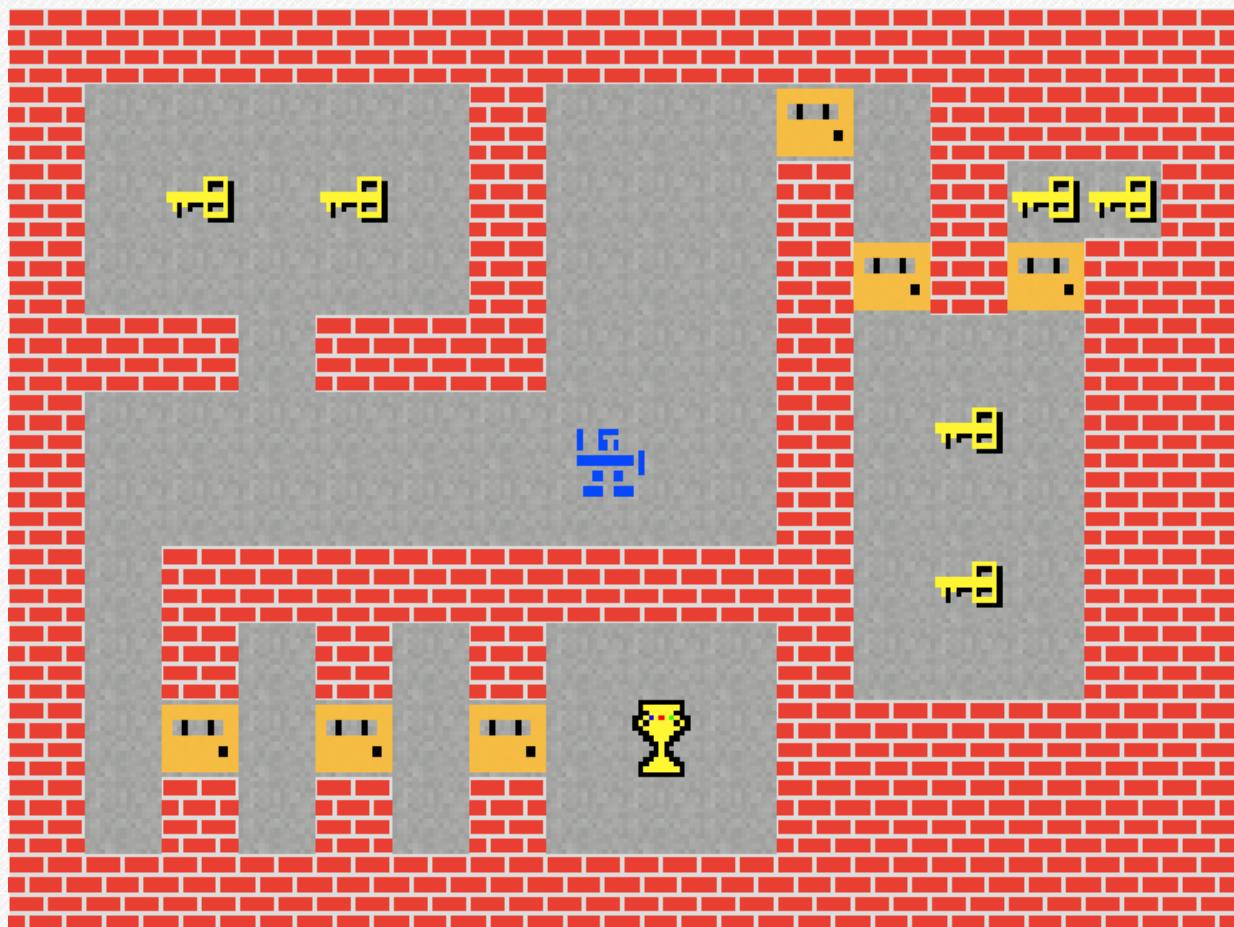
EXPECTED RESULT

Same gameplay. The code is cleaned up for easier reuse or expansion now without all that temporary clutter in the way. Look forward to revisiting Racing in Section 2's exercises, but in the near term it's time to start working on Warrior Legend's core gameplay.

GAME 4: WARRIOR LEGEND

MID-1980'S TILED-DUNGEON

For this game you'll once again begin with the previous one. Even though the play experience is different for an adventure game and a racing game, a lot of that has to do with how the player moves and the art loaded. Beyond those differences, both genres (at least in the simplest early games in those genres) use a very similar world representation in memory.



warrior legend step 1

copy in racing code to start from

This step is an exact copy of the Racing functionality in the same state that it ended in from the prior series of steps.

EXPECTED RESULT

It's just the Racing code still. This won't be the case for long.

warrior legend step 2 removing second player's car

Remove the Player 2 instance p2 from the code entirely. This will be a single player game. In addition to any mention of p2 in Main.js, wipe out the call to p2.carReset() in Car.js. The call on the global instance p1 in carReset() can become this.carReset() since there's only one player left. (This removes the need to reference any outside instance within the class.)

Remove one of the "2" values from the trackGrid in Track.js, leaving only one player start position. Letting a second "2" stay in the grid without a corresponding car setup to remove it could cause unpredictable bugs in either our track rendering or movement collision code, since neither specifically handles TRACK_PLAYER type tiles during play. Delete the player2.png file from the images folder, and remove both lines related to car2Pic from your ImageLoading.js file.

Delete the line in Input.js that set up the p2 controls. Also delete the call to setKeyHoldState() on p2 in the input handlers keyPressed() and keyReleased(). You could erase the unused const definitions for KEY_LETTER_W (and also A, S, and D) at the top of Input.js. Those keyCode values are still correct though and they're common keys to bind actions to. I'd say may as well keep them around in Input.js in case you'd like to add support or some purpose for them later.

EXPECTED RESULT

Same racing game features and controls should work, except now with no trace of player 2's car.

warrior legend step 3 renaming files and variables

Rename some of our main project files and global variables to correspond to their new usage.

Car.js will become Warrior.js, and carClass will be renamed warriorClass both at the top of Warrior.js and also where it's referenced in the Main.js file. Don't worry yet about renaming the individual variables and functions that have "car" in them. Those will be fixed up in a different way soon.

Rename Track.js to World.js. Then, in the main HTML file, update the references to Car.js and Track.js to Warrior.js and World.js, respectively.

The GraphicsCommon.js, ImageLoading.js, Input.js, and Main.js files are named in a general enough way to leave their names unchanged.

EXPECTED RESULT

No change to game functionality, but check to make sure that they game still plays (as a simple single player racing game) and isn't producing any new errors due to a missed connection during the renaming.

warrior legend step 4 generalize naming in player code

The next target for cleanup and repurposing is the Warrior.js file. In Warrior.js, do a search for the word “car” and delete every use of it. I don't suggest using “find and replace” though because you'll also want to clean up camel case capitalization (this.carAng becomes this.ang not this.Ang, this.carX becomes this.x not this.X, this.carMove() is now this.move() and so on).

Also remove the word “car” from the few places that it shows up in Main.js's function calls. For the carPic variable in ImageLoading.js which is referenced in Main.js, because “pic” without the “car-“ prefix is too vague, change carPic in to playerPic as a more general term fitting its functionality.

Notice how the code is being generalized in a way that will make it easier to repurpose beyond this specific game. Most games have a player character which needs functions for init, reset, input, move, and draw. The original reason why all car variables were prefixed by "car-" was to differentiate them from any other values in the global scope. Now that the variables are contained in a class, that prefix is built in by the context or class instance: what was carX will be this.x within the class, and something like car.x, warrior.x, or p1.x if referenced elsewhere in other files.

On the other hand, I'm still not suggesting going off the deep end and generalizing code for its own sake, striving to build a one-size-fits-all reusable game engine or template. The driving purpose is still to make a particular game. This is why, at least

for now, I suggested renaming the player class to warriorClass rather than a more abstract term like playerClass. By doing so, you are giving yourself clear permission to write warrior-specific code into this class to get the game done. This avoids getting lost and wasting time theorizing about hypotheticals.

Before worrying about making highly reusable parts, focus first on usable parts. As you work on more types of games, clearer patterns will emerge of what parts are generic and reusable, and which are specific to each game.

EXPECTED RESULT

There's no expected change here to the game's functionality. Making these parts of code a little easier to understand and reuse will speed up recycling and repurposing them for other game projects.

warrior legend step 5

walk like a warrior - key changes

Now that the code indicates the player is a warrior, move on to making the player character move more like a warrior.

Much as you've already been doing, this will start with some under the hood variable renaming to reflect the change from gas, reverse and character relative steering to instead sliding the character north, east, south, and west.

Rename the input boolean values keyHeld_Gas and such on near the top of Warrior.js, to keyHeld_North, keyHeld_East, keyHeld_South, and keyHeld_West. Spelling out the cardinal directions instead of shortening it to keyHeld_N etc. reduces the potential for later confusion over whether these might refer to holding the N, E, S, and W letter keys.

Also update the control setup function's parameter names and assignments in Warrior.js:

```
this.setupControls = function(northKey, eastKey, southKey, westKey) {  
    this.controlKeyForNorth = northKey;  
    this.controlKeyForEast = eastKey;  
    this.controlKeyForSouth = southKey;  
    this.controlKeyForWest = westKey;  
}
```

These are *not* the same as this.keyHeld_North – that's what will be set or checked whenever the this.controlKeyForNorth keycode is held.

Additionally, watch out for the changed order of parameters for keys here. Before, the order was forward, backward, then left and right. Now the expected order is N, E, S, then W. By picking and sticking to a convention like this there won't be

any confusion or mix ups going forward about what order to pass or handle direction-related values in, meaning less to memorize or think through on a case-by-case basis. Head over to Input.js where setupControls() gets called on p1 and reorder the key mappings to match the argument order expected by the function. It did look like this (when the order meant gas, reverse, steer left, steer right):

```
p1.setupControls(KEY_UP_ARROW, KEY_DOWN_ARROW, KEY_LEFT_ARROW, KEY_RIGHT_ARROW);
```

But now looks like this (now that order means north, east, south, west):

```
p1.setupControls(KEY_UP_ARROW, KEY_RIGHT_ARROW, KEY_DOWN_ARROW, KEY_LEFT_ARROW);
```

While in Input.js, make a few updates to the setKeyHoldState() function. Change the thisCar argument (and its eight local appearances in the function) to thisPlayer to catch it up with the variable and function changes elsewhere. References in setKeyHoldState() to variables like controlKeyForTurnLeft need to be changed to controlKeyForWest, while variable names on thisPlayer like keyHeld_TurnLeft have to be updated to keyHeld_West.

Since the behavior of these if-conditions is not affected by their order, it would be harmless to keep them in their original order and replace forward with north, reverse with south, and so on. Since you're editing this bit of code anyhow, you might as well rearrange it to be in the same north, east, south, west order used elsewhere for consistency. This consistency can help identify a transcription mistake or related variable mismatch. The new form for setKeyHoldState(), which includes

the change to its argument name in the declaration from thisCar to thisPlayer, is:

```
function setKeyHoldState(thisKey, thisPlayer, setTo) {  
    if(thisKey == thisPlayer.controlKeyForNorth) {  
        thisPlayer.keyHeld_North = setTo;  
    }  
    if(thisKey == thisPlayer.controlKeyForEast) {  
        thisPlayer.keyHeld_East = setTo;  
    }  
    if(thisKey == thisPlayer.controlKeyForSouth) {  
        thisPlayer.keyHeld_South = setTo;  
    }  
    if(thisKey == thisPlayer.controlKeyForWest) {  
        thisPlayer.keyHeld_West = setTo;  
    }  
}
```

With all the key and movement variable naming fixed up, you're ready to implement the character movement changes. Even though you'll soon be replacing the car steering with walking movement, update any references in Warrior.js's move() to match their new counterparts: this.keyHeld_TurnLeft will become this.keyHeld_West, this.keyHeld_Gas will become this.keyHeld_North, and so on for both right and reverse.

EXPECTED RESULT

No changes. Arrow keys should still drive the car as they did in the Racing game. All this renaming and cleaning up will pay off with some seemingly very rapid changes soon.

warrior legend step 6

walk like a warrior - slide to move

In adventure games like this one the player character typically walks directly north, east, south, or west in a straight line. This is quite different from the way that the cars for Racing drove in curving paths, accelerated, and rolled to a stop.

The soon-to-be-sliding player character will no longer need a `this.ang` variable since the player will no longer spin. Therefore, `this.ang` can be removed from `Warrior.js`'s `reset()` function. The `this.speed` variable won't be needed either since, unlike a car which builds up speed and rolls to a stop, our warrior will either be walking or standing still. Most of the constant values defined at the top of `Warrior.js` were related to engine power and turn speed. Since those no longer apply all of these can be removed:

```
const GROUNDSPEED_DECAY_MULT = 0.94;
const DRIVE_POWER = 0.5;
const REVERSE_POWER = 0.2;
const TURN_RATE = 0.03;
const MIN_TURN_SPEED = 0.5;
```

In their place, put a single new constant for walk speed:

```
const PLAYER_MOVE_SPEED = 3.0;
```

In the `Warrior.js` `move()` function, it's time to replace the chunk of steering and acceleration code with straightforward north, east, south, and west walking.

You can still use local variables `nextX` and `nextY` to check for a potential wall collision before updating the character position, though it will be connected differently this time. First, set `nextX` and `nextY` to the current position, then bump `nextX` and

nextY left or right by PLAYER_MOVE_SPEED based on which movement control keys are currently held.

Walking east or west – right or left – will increase or decrease nextX. Walking north or south – up or down – will increase or decrease nextY. If the position at nextX and nextY is over a road (floor) type of tile then at that time the this.x and this.y coordinates get updated to those adjusted values.

Remember that often in computer graphics, including canvas coordinates for JavaScript and HTML5, 0 is at the top and larger y values are lower on the screen. Therefore, moving up will require subtracting from the vertical position, not adding.

Here's the fully updated move() function in Warrior.js:

```
this.move = function() {
    var nextX = this.x;
    var nextY = this.y;

    if(this.keyHeld_North) {
        nextY -= PLAYER_MOVE_SPEED;
    }
    if(this.keyHeld_East) {
        nextX += PLAYER_MOVE_SPEED;
    }
    if(this.keyHeld_South) {
        nextY += PLAYER_MOVE_SPEED;
    }
    if(this.keyHeld_West) {
        nextX -= PLAYER_MOVE_SPEED;
    }

    var walkIntoTileType = getTrackAtPixelCoord(nextX,nextY);

    if( walkIntoTileType == TRACK_ROAD ) {
        this.x = nextX;
        this.y = nextY;
    } else if( walkIntoTileType == TRACK_GOAL ) {
        document.getElementById("debugText").innerHTML = this.myName + " won";
        this.reset();
    }
}
```

The Warrior.js draw() function no longer needs the angle argument to drawBitmapCenteredAtLocationWithRotation(), as there is not this.ang value to feed it. Input a hard coded 0.0 for that argument to leave the image upright, unrotated. The function will still display the image centered at its location, and keeping the rotation capability in the function may be helpful later for projectiles and certain enemy types.

EXPECTED RESULT

The player car will look a bit silly, always facing to the right as it slides in any direction. When you replace it with a warrior graphic this won't look so bad. The important thing for now is that the arrows can "walk" the player character in any direction. Walls should still block the player's movement.

warrior legend step 7 bigger and fewer tiles on screen

As an intermediary change before switching to adventure world graphic tiles, increase the tile size from 40x40 to 50x50 for slightly larger characters and environment detail. Update both TRACK_W and TRACK_H in World.js to 50.

Since the canvas will still be 800x600 (specified in the game's central HTML file), this also changes the TRACK_COLS from 20 to 16 (since $800/50$ is 16), and TRACK_ROWS to 12 (since $600/50$ is 12). Use an image editing program to resize each of the track_*.png files in the images folder up to 50x50 – no worry about blurriness or pixelation, it won't matter for long since you'll soon replace these with different, more appropriate adventure graphics. If you don't have an image editing program handy or prefer to focus on the code side, you can copy the 50x50 stretched assets from the example solution's image folder.

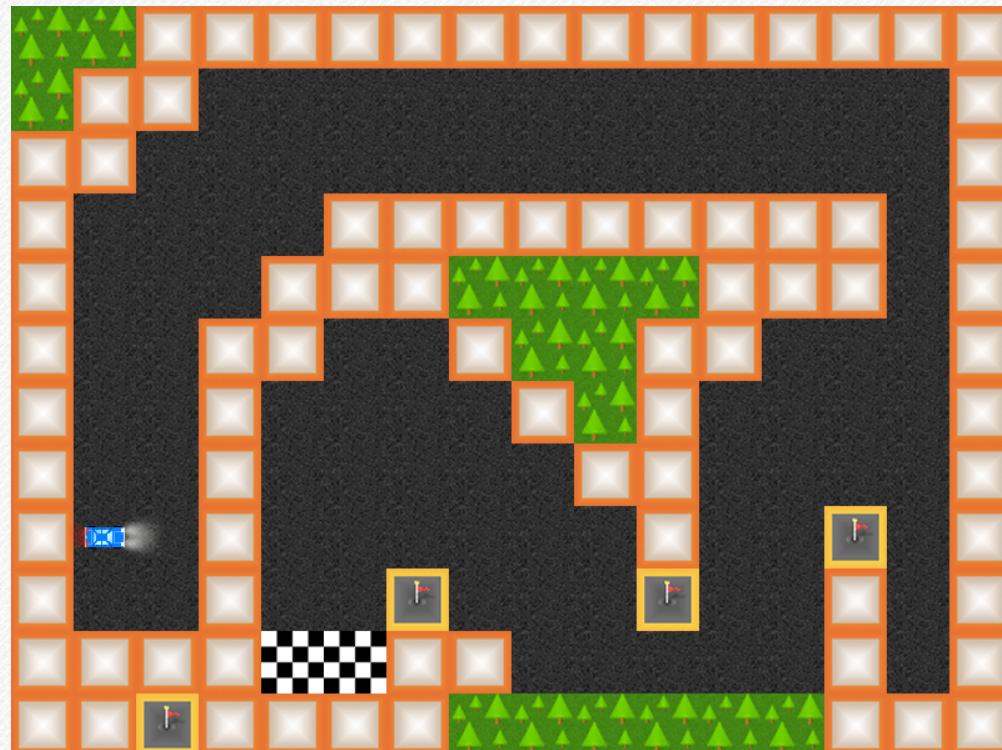
Additionally, you'll need to edit the trackGrid data so it is 16x12 tile codes, rather than the previous 20x15 layout. This might sound hard to do without messing up, but it's really not so bad. Shave off the difference in rows and columns from the right side and bottom. That means removing the last 4 tile values from each row, leaving 16 instead of 20, and trim 3 rows to cut from 12 down from 15. Block edges with wall tiles to avoid potentially unpredictable error from the player leaving the canvas, since that situation is not handled in the code.

Since you probably removed the goal tiles when removing rows, insert a couple of new ones at the end of the track. New trackGrid definition:

```
var trackGrid =  
[ 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
 4, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,  
 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,  
 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,  
 1, 0, 0, 0, 1, 1, 1, 4, 4, 4, 4, 1, 1, 1, 0, 1,  
 1, 0, 0, 1, 1, 0, 0, 1, 4, 4, 1, 1, 0, 0, 0, 1,  
 1, 0, 0, 1, 0, 0, 0, 0, 1, 4, 1, 0, 0, 0, 0, 1,  
 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1,  
 1, 2, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 5, 0, 1,  
 1, 0, 0, 1, 0, 0, 5, 0, 0, 0, 5, 0, 0, 1, 0, 1,  
 1, 1, 1, 1, 3, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, //// added 3's  
 1, 1, 5, 1, 1, 1, 4, 4, 4, 4, 4, 1, 1, 1];
```

EXPECTED RESULT

Larger tiles depicting a somewhat simpler section of track. Arrow keys should still move the player car around on road tiles. Although the tiles are 25% bigger than they previously were, there's no change at this time to the car size. Bumping into the goal line still resets the car back to its start position.



warrior legend step 8 switching in warrior & dungeon art

Replace the track graphics with new graphics that will make more sense for our indoor warrior game. I've provided some examples in my solution, but they're pretty low quality, so be my guest to replace them:

replace player1.png with warrior.png

replace track_road.png with world_ground.png

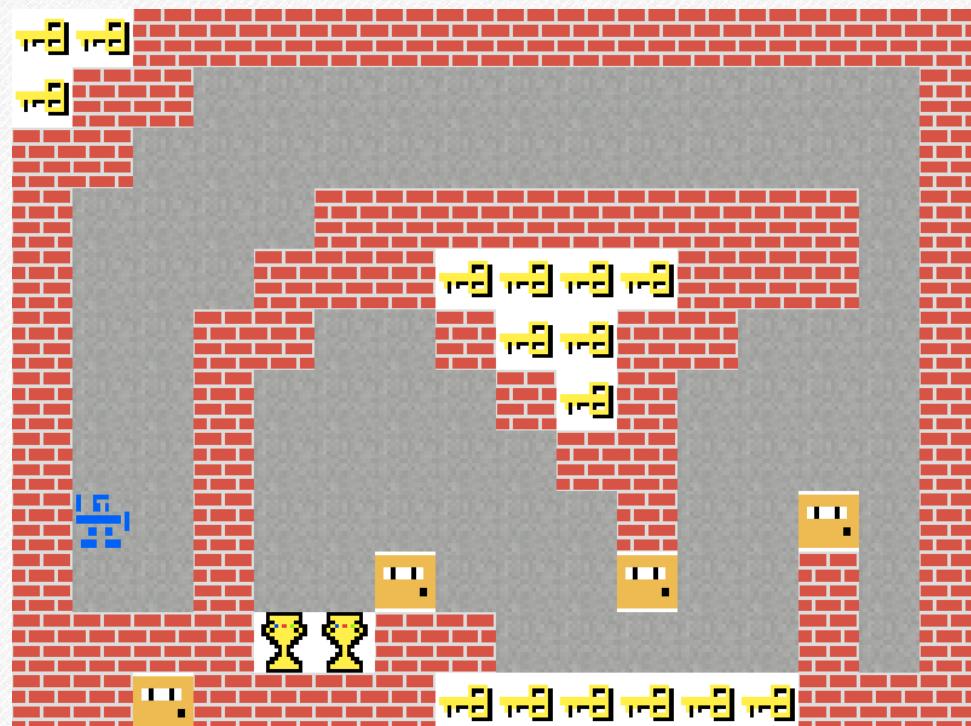
replace track_wall.png with world_wall.png

replace track_goal.png with world_goal.png

replace track_tree.png with world_key.png

replace track_flag.png with world_door.png

In addition to replacing the actual files in the images folder, the filenames in ImageLoading.js need to be updated.



EXPECTED RESULT

The world will now look very different, with a (very simple looking) blue warrior sliding around on a gray floor between brick walls. Doors and human-sized keys will be sprinkled in non-sensical positions on the map corresponding to where there were previously tree and flag tiles for the racing layout.

warrior legend step 9 renaming room variables in code

There are several issues surfaced by this art swap: keys and doors don't function yet, the current map layout doesn't make sense for these new elements, and the floor needs to show beneath all the key, door, and goal tiles. Much to do.

As in previous updates, to maintain sanity it'll be helpful to get the code's variable and function names to reflect their current uses. You don't want to have to remember that the tile type TRACK_TREE *actually* corresponds to a collectible key, TRACK_FLAG corresponds to a door that can be opened, and so on.

Most of these changes will be focused in World.js, however the other files referencing the TRACK_* const labels, trackGrid, or other track functions will be affected as well.

Bear in mind that adventure games tend to take place beyond a single screen. Even the very first graphical game in this design genre, Adventure for Atari in the late 1970s, supported the ability to walk from one screen-sized room to another by walking against a screen edge. This can affect your labels for constants and variables because, while it might be tempting to simply replace the TRACK_ prefixes in World.js with WORLD_, to do so could mix up what's just on one screen with what it is that makes up the whole world.

For the racing game, the track on screen was 100% of the total track. That's not true for an adventure game. The screen represents only one room. Rename the TRACK_COLS and

TRACK_ROWS values to ROOM_COLS and ROOM_ROWS, and trackGrid should become roomGrid. You'll later be able to support switching different values for roomGrid based on the screenful of room tile information found within a larger world.

Before changing TRACK_W to ROOM_W or TRACK_ROAD to ROOM_GROUND, there's a more fitting prefix for these constants than either ROOM_ or WORLD_. What is it that's 50x50 (TRACK_W by TRACK_H), and what sort of thing does TRACK_ROAD or its related const values refer to? Tiles!

I suggest TILE_ as a more descriptive prefix for the _W width and _H height pixel measurements. Here's the updated block at the top of World.js, where I've repositioned the TILE_W and TILE_H values to keep room and tile values grouped in code:

```
// world, room, and tile constants, variables
const ROOM_COLS = 16;
const ROOM_ROWS = 12;

var roomGrid =
  [ 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    4, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
    1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
    1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,
    1, 0, 0, 1, 1, 1, 4, 4, 4, 4, 1, 1, 1, 0, 1,
    1, 0, 0, 1, 1, 0, 0, 1, 4, 4, 1, 1, 0, 0, 0, 1,
    1, 0, 0, 1, 1, 0, 0, 1, 4, 4, 1, 1, 0, 0, 0, 1,
    1, 0, 0, 1, 0, 0, 0, 1, 4, 1, 0, 0, 0, 0, 0, 1,
    1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1,
    1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1,
    1, 2, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 5, 0, 1,
    1, 0, 0, 1, 0, 0, 5, 0, 0, 0, 5, 0, 0, 1, 0, 1,
    1, 1, 1, 1, 3, 3, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1,
    1, 1, 5, 1, 1, 1, 4, 4, 4, 4, 4, 1, 1, 1, 1];

const TILE_W = 50;
const TILE_H = 50;

const TILE_GROUND = 0;
const TILE_WALL = 1;
const TILE_PLAYER = 2;
const TILE_GOAL = 3;
const TILE_KEY = 4;
const TILE_DOOR = 5;
```

The function trackTileToIndex has the word "track" in it, but becomes roomTileToIndex.

The function getTrackAtPixelCoord() is actually returning the tile type at a pixel coordinate, so update its name to getTileAtPixelCoord(). Within that function, change the local variable trackIndex to tileIndex. Accounting for the other const and function name changes, here's how getTileAtPixelCoord() looks after modification:

```
function getTileAtPixelCoord(pixelX,pixelY) {
    var tileCol = pixelX / TILE_W;
    var tileRow = pixelY / TILE_H;

    // we'll use Math.floor to round down to the nearest whole number
    tileCol = Math.floor( tileCol );
    tileRow = Math.floor( tileRow );

    // first check whether the car is within any part of the track wall
    if(tileCol < 0 || tileCol >= ROOM_COLS ||
       tileRow < 0 || tileRow >= ROOM_ROWS) {
        return TILE_WALL; // avoid invalid array access, treat out of bounds as wall
    }

    var tileIndex = roomTileToIndex(tileCol, tileRow);
    return roomGrid[tileIndex];
}
```

World.js's drawTracks() function becomes drawRoom(). Within that function, rename the trackIndex, trackLeftEdgeX, trackTopEdgeY, and trackTypeHere local variables to have "tile" in place of "track" as the prefix. With all the changes, here's how the drawRoom() function winds up:

```
function drawRoom() {
    var tileIndex = 0;
    var tileLeftEdgeX = 0;
    var tileTopEdgeY = 0;

    for(var eachRow=0; eachRow<ROOM_ROWS; eachRow++) { // deal with one row at a time

        tileLeftEdgeX = 0; // resetting horizontal draw position for tiles to left edge

        for(var eachCol=0; eachCol<ROOM_COLS; eachCol++) { // left to right in each row
```

```

    var tileTypeHere = roomGrid[ tileIndex ]; // getting the tile code for this index
    canvasContext.drawImage(tilePics[tileTypeHere], tileLeftEdgeX, tileTopEdgeY);

    tileIndex++; // increment which index we're going to next check for in the room
    tileLeftEdgeX += TILE_W; // jump horizontal draw position to next tile by tile width

} // end of for eachCol

tileTopEdgeY += TILE_H; // jump horizontal draw position down by one full tile height

} // end of for eachRow
} // end of drawRoom()

```

In ImageLoading.js, replace trackPics with tilePics in all locations, and rename loadImageForTrackCode() to loadImageForTileCode().. Rename the parameter for loadImageForTileCode() from trackCode to tileCode.

Also in ImageLoading.js, update loadImageForTrackCode() calls to use the new const TILE_* types that correspond to each new PNG image:

```

var imageList = [
  {varName:playerPic, theFile:"warrior.png"},

  {tileType:TILE_GROUND, theFile:"world_ground.png"}, 
  {tileType:TILE_WALL, theFile:"world_wall.png"}, 
  {tileType:TILE_GOAL, theFile:"world_goal.png"}, 
  {tileType:TILE_KEY, theFile:"world_key.png"}, 
  {tileType:TILE_DOOR, theFile:"world_door.png"} 
];

```

In Main.js, replace drawTracks() with drawRoom().

In Warrior.js, change trackGrid to roomGrid, TRACK_PLAYER to TILE_PLAYER, and TRACK_ROAD to TILE_GROUND. Further down in the move() function, change getTrackAtPixelCoord() to getTileAtPixelCoord(), and change TRACK_ROAD there into TILE_GROUND. TRACK_GOAL becomes TILE_GOAL.

EXPECTED RESULT

No change in functionality. Since these changes involved many references across many files, it's important to check your console log while running the game in-browser to ensure there are no errors.

warrior legend step 10

support for transparency in tile art

Unlike the Racing game's setting, in which every tile type filled the whole tile square, several of the adventure game tile graphics include transparency meant to show the floor behind them. So far those include keys, doors, and the goal chalice.

You could hand copy the floor image into each of the graphics for those other tile types. However, doing so would make it really time consuming to iterate on different floor patterns or colors. It would also increase the total number of tile images required if you later decide to support multiple types of flooring for different kinds of settings (cave, forest, temple...). Here's an opportunity to spend a little extra time now to save much more time and work later.

The key, door, and chalice tile graphics provided already include transparency around them. All you need to do is update the drawing code to display the floor graphic before displaying each of those tile types during the drawRoom() function. The specific section of code to be addressed is in drawRoom() of World.js where the program looked like this:

```
var tileTypeHere = roomGrid[ tileIndex ]; // getting the tile code for this index  
canvasContext.drawImage(tilePics[tileTypeHere], tileLeftEdgeX, tileTopEdgeY);
```

You could add an if-comparison between those two lines, i.e. after the tile type is known but before it gets displayed:

```
var tileTypeHere = roomGrid[ tileIndex ]; // getting the tile code for this index  
if(tileTypeHere == TILE_GOAL || tileTypeHere == TILE_KEY || tileTypeHere == TILE_DOOR) {  
    canvasContext.drawImage(tilePics[TILE_GROUND], tileLeftEdgeX, tileTopEdgeY);  
}  
canvasContext.drawImage(tilePics[tileTypeHere], tileLeftEdgeX, tileTopEdgeY);
```

That extra if-comparison checks whether the tile type to be drawn is one of our types that has transparency. If so, the ground tile is drawn first in the same position. The order of these drawImage() calls is critical since whichever image is drawn later overlaps the image drawn earlier. Since the ground has no transparency, if the ground tile were drawn second, it would completely cover the object drawn first.

Consider what would go into adding another type of tile if this was the solution used:

1. Prepare and load an image for it.
2. Create a new numerical const code to signify it in the map.
3. Update player collision to account for it.
4. (New from this change) In the middle of drawRoom(), check if it's a type that uses transparency and add it to this nondescript chain of or-linked if-comparisons.

Creating a small helper function that exists solely to identify which tile type numbers have transparency accomplishes two objectives. The if-statement in the middle of drawRoom() will be more self-explanatory. Also, by positioning the transparency check right next to where the tile type const values are declared, it's going to be easier to notice and remember to update it when adding other tile types.

The helper function, which I'll place on line 29 below the tile const definitions, takes this form:

```
function tileTypeHasTransparency(checkTileType) {  
    return (checkTileType == TILE_GOAL ||  
            checkTileType == TILE_KEY ||  
            checkTileType == TILE_DOOR);  
}
```

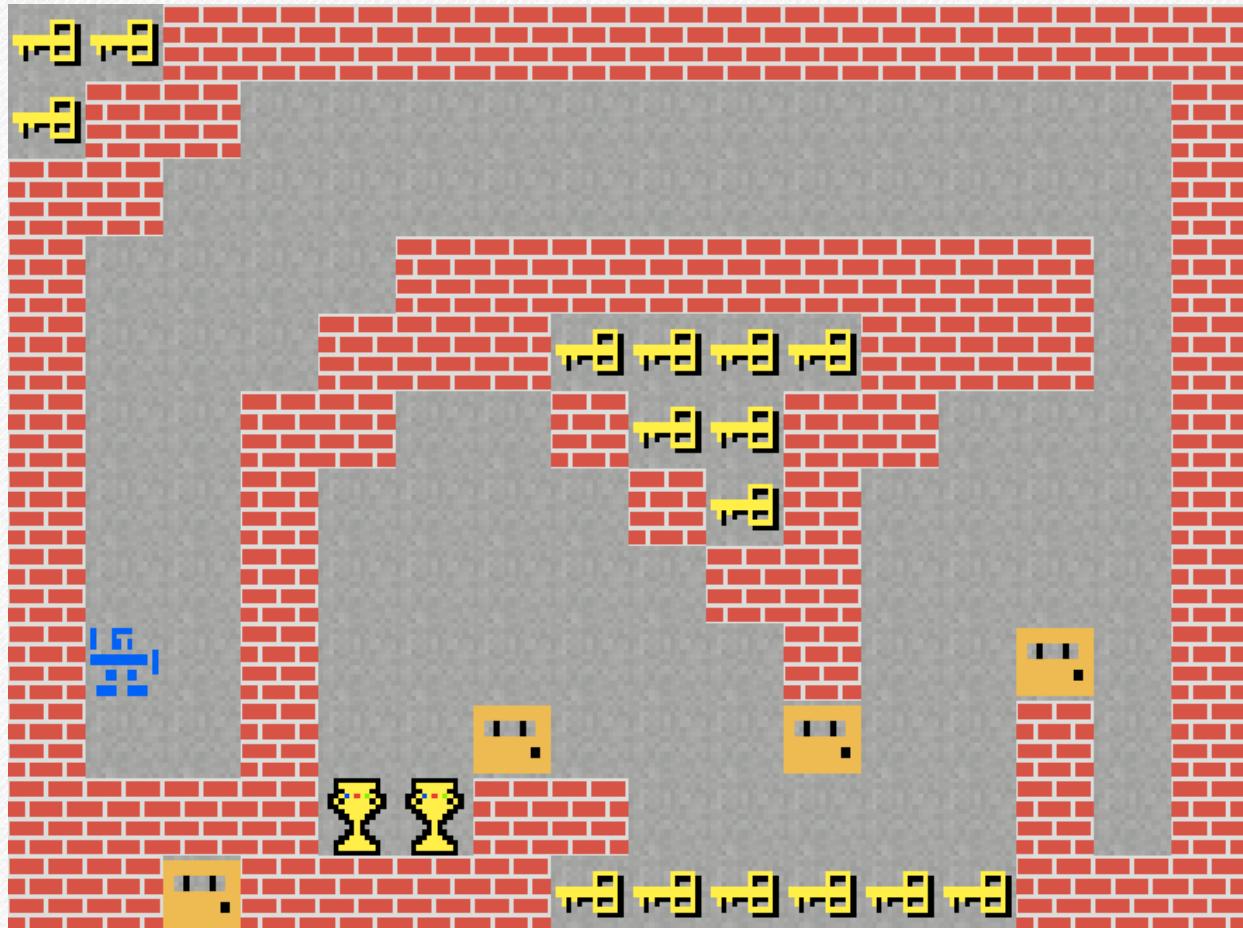
That simplifies the if-comparison in drawRoom(), now on line 69, to this more self-descriptive, human readable form:

```
if( tileTypeHasTransparency(tileTypeHere) ) {  
    canvasContext.drawImage(tilePics[TILE_GROUND], tileLeftEdgeX, tileTopEdgeY);  
}
```

EXPECTED RESULT

Instead of seeing empty void around the key, door, and goal tiles, the floor should now fill in behind/around them.

Gameplay and character movement work the same as before.



warrior legend step II

creating a dungeon layout

The current map layout is an adapted leftover from the cropped Racing game track, which has keys in unreachable places and doors in spots that don't make sense. It will only take a moment to change the number arrangement of roomGrid in World.js to create a more practical test level layout, forming a more usable space in which to test the upcoming extra functionality for collecting keys to unlock doors. You're welcome to experiment and improvise as you'd like with the new layout. Here's the one that I'm going to use near the top of World.js, in the program's global scope:

```
var roomGrid =  
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
    1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 5, 0, 1, 1, 1, 1,  
    1, 0, 4, 0, 4, 0, 1, 0, 2, 0, 1, 0, 1, 4, 4, 1,  
    1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 5, 1, 5, 1, 1,  
    1, 1, 1, 5, 1, 1, 1, 0, 4, 0, 1, 0, 0, 0, 1, 1,  
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 4, 0, 1, 1,  
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1,  
    1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 4, 0, 1, 1,  
    1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1,  
    1, 0, 5, 0, 5, 0, 5, 0, 3, 0, 1, 1, 1, 1, 1, 1,  
    1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1,  
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
```

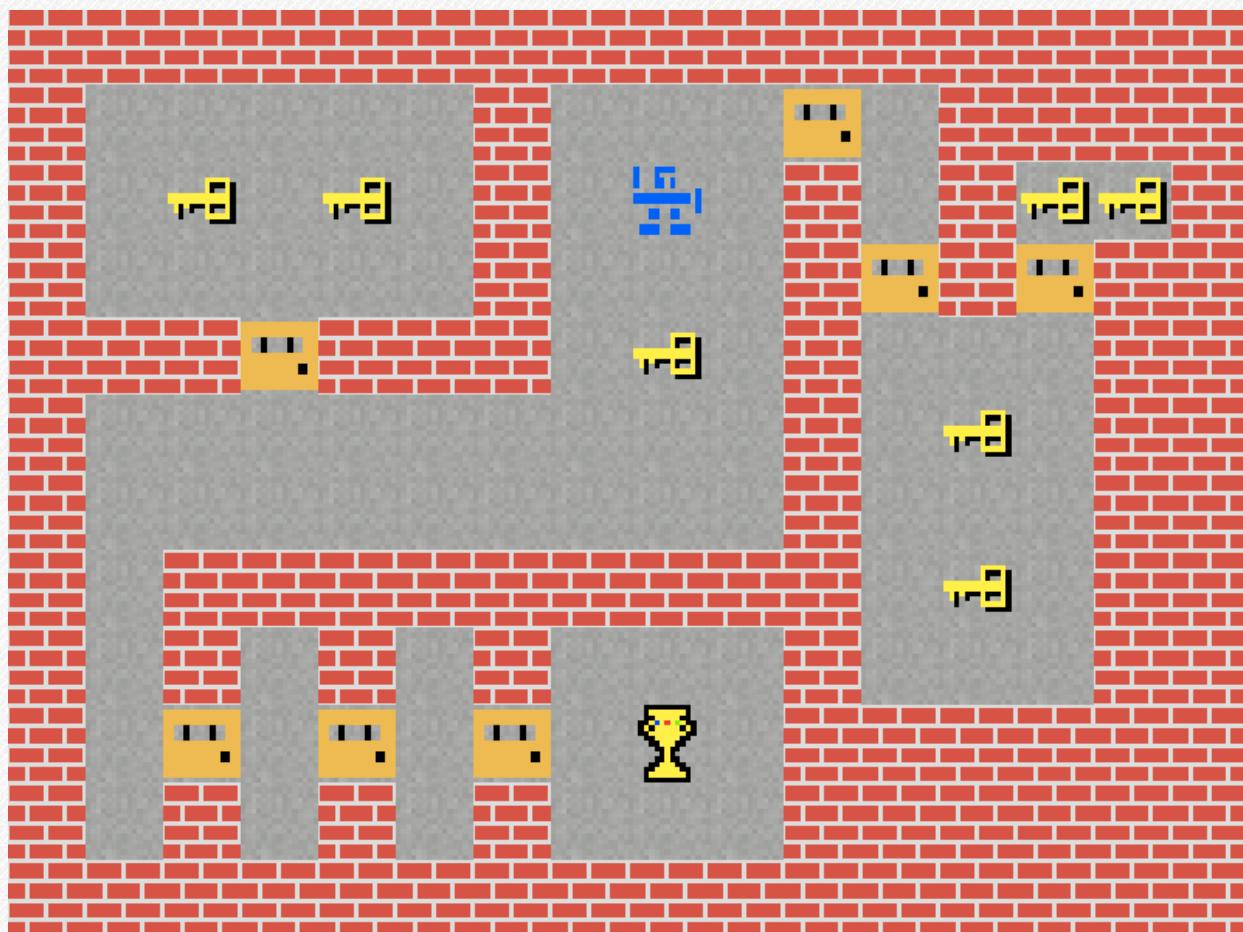
My approach to designing this rudimentary level followed much the same process that I would use for making early playable test spaces in more complex games.

First, I set all the edge spaces to brick wall 1's to avoid dealing with the player stepping against an exposed screen edge (much later this might cause a transition to an adjacent room's tile layout). I then filled the interior space with empty ground 0's to have a blank area to work with. I put a 2 someplace

since I know the player needs a start position, and a 3 to give the map a goal. Then I began sectioning off smaller cells with brick wall 1's.

hen, I experimented with blocking out cells with varying layers of doors (5's) and stashing increasing numbers of keys (4's) in different cells. This formed a sequence in which one key could be used to reach two keys, two keys to reach three keys, then three keys were needed for the goal.

Lastly, I filled in some remaining gaps with brick wall 1's and adjusted both hallway and cell dimensions to improve the appearances in terms of even spacing, cell symmetry, plus a little variety in cell shape, size, and entry/exit placement.



One additional set of considerations I kept in mind relate to our intention of using this layout for quick and easy testing purposes as development continues. Near the player's start position I provided a key, two doors (for quick verification that the single key will unlock the first door, but not the second), and the final goal isn't too far away so that I can cut a quick hole in the map wall whenever I need to test level completion.

EXPECTED RESULT

The room layout will form a coherent – though relatively simple – navigation puzzle, rather than appearing to be a roundabout scrambled mess of misplaced tiles. Keys and doors won't work yet, they'll still block the player like walls.

warrior legend step 12

using keys to unlock doors

When the player bumps a key it will be removed from the map, and the number of keys the player is carrying (indicated for now in the debug text below the canvas) will increase by one. If the player has no keys, doors will work like brick walls. Otherwise, if the player has at least one key upon bumping a door, that door vanishes and the number of keys the player is carrying will decrease by one.

In `Warrior.js`, on the first line of `warriorClass`'s `reset()` function, set a new variable named `this.keysHeld` to 0. Remember that in JavaScript you don't need to explicitly declare this apart from setting it. (Setting x and y at the top of the class with a default just ensures the player character shows up in a position visible on screen even if a level doesn't have a number 2 tile to specify a start location.)

In the `warriorClass move()` function, the `walkIntoTileType` value is compared against `TILE_GOAL` and `TILE_GROUND`.

Touching these types of tiles, or bumping into barriers, doesn't involve changing the value stored at that tile position. You'll need to do so though to pick up KEY tiles and to open DOOR tiles, replacing both with GROUND tiles.

To change the tile where the player is, it will be helpful to temporarily save which tile position's index is being touched. The tile position is found near the end of `getTileAtPixelCoord()` in `World.js`. The function doesn't return `tileIndex` ("tile position 15") though, but instead it gives back `roomGrid[tileIndex]` ("the

tile type found there is a 5, corresponding to a door”). The actual tileIndex position is not stored for reuse, but you’ll need that to change a value there.

Late in the development of Racing, on step 28, the role of interpreting the tile within `getTileAtPixelCoord()` was pushed out to the caller. That involved returning the numerical tile type code (ex. 1 for WALL) instead of letting the function decide which grid positions can be moved through. In `Warrior.js`’s `reset()` function, the `roomGrid` from `World.js` is already checked and edited for the player start position. A similar test in `Warrior.js`’s `move()` function is possible by simplifying `getTileAtPixelCoord()` to return a tile’s position index without first translating it with the `roomGrid`.

(As the program grows and accumulates more interacting parts you may wish to refactor and reorganize the code a bit to prevent direct access to reading or modifying `roomGrid` from outside the `World.js` file. Since the `Warrior` is still the only moving object, this won’t be too much to keep straight.)

First do a Find in Files to verify that `getTileAtPixelCoord()` is only used within `Warrior.js`’s `move()` function. This is to ensure that renaming it and changing how it works won’t break something elsewhere in the code.

In `World.js`, at the end of `getTileAtPixelCoord()`, change it from:

```
var tileIndex = roomTileToIndex(tileCol, tileRow);
return roomGrid[tileIndex];
```

...to this:

```
var tileIndex = roomTileToIndex(tileCol, tileRow);
return tileIndex; // no longer checking to find what number is there in roomGrid[]
```

One downside to this change in approach is: this function can no longer provide a safely usable generic response to a request for an out of bounds coordinate.

If an outside caller, say the Warrior's move() code, requests from this function the index of the tile located 1,000 pixels above the top of the canvas, or far right of the playable area, there will be a problem. Before, the function could provide a placeholder answer, a little white lie about what tile type is there. World.js used to answer, "I checked that spot and it's a TILE_WALL."

Now the Warrior move() function requires a valid index to the roomGrid array. The answer 0 won't work well for a fallback, since that's the top-left visible tile. Negatives, or numbers that are too high, will cause an array access error. From within the function, at best you can catch and log an error message, but that is essentially the same as letting the error happen.

Responsibility to catch the error condition and handle it is now solely over in the Warrior's move() code.

Since you've modified what the function returns, change its name to reflect that. Instead of getTileAtPixelCoord(), rename it in both World.js and Warrior.js to getTileIndexAtPixelCoord(). Here's how the function looks after the changes:

```
function getTileIndexAtPixelCoord(pixelX,pixelY) {
  var tileCol = pixelX / TILE_W;
  var tileRow = pixelY / TILE_H;

  // we'll use Math.floor to round down to the nearest whole number
  tileCol = Math.floor( tileCol );
  tileRow = Math.floor( tileRow );
```

```

    // first check whether the tile coords fall within valid bounds
    if(tileCol < 0 || tileCol >= ROOM_COLS ||
       tileRow < 0 || tileRow >= ROOM_ROWS) {
        document.getElementById("debugText").innerHTML = "out of bounds:"+pixelX+","+pixelY;
        return undefined;
    }

    var tileIndex = roomTileToIndex(tileCol, tileRow);
    return tileIndex; // no longer checking to find what number is there in roomGrid[]
}

```

Back to Warrior.js. Since the function's use and name changed, it can't be handled in the same way that it was:

```
var walkIntoTileType = getTileIndexAtPixelCoord(nextX,nextY);
```

Now it's up to the Warrior move() code to reinterpret the tile index to get the corresponding tile type (door, etc):

```
var walkIntoTileIndex = getTileIndexAtPixelCoord(nextX,nextY);
var walkIntoTileType = roomGrid[walkIntoTileIndex];
```

This spot in the code is now on the hook to safely handle an out of bounds case. As with the older arrangement, set up the TILE_WALL value as the fallback answer to use as the basis for character behavior when checking out of bounds places:

```
var walkIntoTileIndex = getTileIndexAtPixelCoord(nextX,nextY);
var walkIntoTileType = TILE_WALL;

if( walkIntoTileIndex != undefined) {
    walkIntoTileType = roomGrid[walkIntoTileIndex];
}
```

That's kind of gnarly, but it works. The whole reason for this roundabout was so the door and key handling code will be able to easily overwrite those positions in the level layout by using an instruction in this form:

```
roomGrid[walkIntoTileIndex] = TILE_GROUND;
```

Speaking of handling the door and key code in the Warrior move() function, you could continue extending out the same if-else-if pattern of equality comparisons to check TILE_KEY and TILE_DOOR, in this manner:

```
if( walkIntoTileType == TILE_GROUND ) {  
    // let the player complete their current nearby movement attempt  
} else if( walkIntoTileType == TILE_GOAL ) {  
    // set win message, reset round  
} else if( walkIntoTileType == TILE_DOOR ) {  
    // check for key, if available decrease key count and remove door  
} else if( walkIntoTileType == TILE_KEY ) {  
    // remove key and increase key count  
} else {  
    // any other tile type number was found, maybe WALL... do nothing, maybe?  
}
```

That would work. However, this if-else-if chain has some really specific properties that make it a natural fit for using a switch-case statement. The switch-case format was shown during Racing's development to toggle between multiple tile images before the code was updated to use a tile image array.

The switch-case pattern maps well to this situation. The properties which make this a good candidate for using a switch-case instead of an if-else chain are:

- 1) Each comparison is a direct equality check, there are no inequality comparisons, calculations or combined boolean logic expressions involved.
- 2) Every comparison in the chain is made against the same variable, walkIntoTileType.
- 3) The equality checks are made between precise values, simple whole numbers (hiding behind descriptive const labels) like 3 or 5. There's no risk of slight imprecision as could

happen if matching, say, 0.33333 repeating or checking whether a numerical value "equals" pi.

4) Unlike the earlier usage during Racing's development in which a switch-case was used to map number codes to multiple tile images, here each number code needs separate and unique behavioral logic. The switch-case's behavior couldn't easily be replicated or reduced to numerical access into an array of simple values.

The switch-case pattern can be used to accomplish the exact same functionality as the if-else chain previously shown, with the following form:

```
switch( walkIntoTileType ) {  
    case TILE_GROUND:  
        // let the player complete their current nearby movement attempt  
        break;  
    case TILE_GOAL:  
        // set win message, reset round  
        break;  
    case TILE_DOOR:  
        // check for key, if available decrease key count and remove door  
        break;  
    case TILE_KEY:  
        // remove key and increase key count  
        break;  
    case TILE_WALL:  
    default:  
        // any other tile type number was found... do nothing, for now  
        break;  
}
```

To reiterate the dangers of a switch-case: The break statements are very important. Without them code "falls" straight into running whatever's written for the case after it.

For example, if you left out the break statement on the line between the TILE_DOOR and the TILE_KEY cases, walking into a door would execute the door bump code and also

immediately run the key bumping code too. That would cause a bug in which opening a door wouldn't reduce your number of keys, since right after removing one it would immediately also run the code that adds one.

Sometimes, case "fall through" is done deliberately to stack the functionality of two or more adjacent matches, as is shown at the end of the statement with both case TILE_WALL and the default (any other value which lacks a case) leading to the same code.

The potential for bugs that can be frustrating to track down leads some programmers to avoid switch-case statements, preferring explicit if-else comparison chains.

Warrior.js's move() function looks now looks like this:

```
this.move = function() {
    var nextX = this.x;
    var nextY = this.y;

    if(this.keyHeld_North) {
        nextY -= PLAYER_MOVE_SPEED;
    }
    if(this.keyHeld_East) {
        nextX += PLAYER_MOVE_SPEED;
    }
    if(this.keyHeld_South) {
        nextY += PLAYER_MOVE_SPEED;
    }
    if(this.keyHeld_West) {
        nextX -= PLAYER_MOVE_SPEED;
    }

    var walkIntoTileIndex = getTileIndexAtPixelCoord(nextX,nextY);
    var walkIntoTileType = TILE_WALL;

    if( walkIntoTileIndex != undefined) {
        walkIntoTileType = roomGrid[walkIntoTileIndex];
    }
}
```

(Continued on the next page)

```

switch( walkIntoTileType ) {
    case TILE_GROUND:
        this.x = nextX;
        this.y = nextY;
        break;
    case TILE_GOAL:
        document.getElementById("debugText").innerHTML = this.myName + " won";
        this.reset();
        break;
    case TILE_DOOR:
        if(this.keysHeld > 0) {
            this.keysHeld--;
            // one less key
            document.getElementById("debugText").innerHTML = "Keys: "+this.keysHeld;

            roomGrid[walkIntoTileIndex] = TILE_GROUND; // remove door
        }
        break;
    case TILE_KEY:
        this.keysHeld++;
        // gain key
        document.getElementById("debugText").innerHTML = "Keys: "+this.keysHeld;

        roomGrid[walkIntoTileIndex] = TILE_GROUND; // remove key
        break;
    case TILE_WALL:
    default:
        // any other tile type number was found... do nothing, for now
        break;
}
}

```

EXPECTED RESULT

Core gameplay works now. Doors are removed on contact if the warrior has at least one key, and the text below the game shows how many keys the player currently holds. Bumping a key removes it from the map and adds one to the player's key count. The level can be completed if the player gets the keys in the right order. Notably, upon reaching the goal chalice, the player's position is reset, but the doors and keys won't come back, since they were overwritten in the roomGrid array during gameplay. The browser page must be refreshed to play again, or for that matter, to reset the game after unlocking doors in a careless order.

The game currently only provides about 30 seconds of play. It's clearly not perfect, or anywhere near ready for launch, but this is a solid starting point. Polish can be dealt with through the exercises by extending the existing features.

Here's a short preview of some of the most significant issues and shortcomings that you'll be addressing in the Section 2 exercises for Warrior Legend:

- the warrior currently always and only faces to the right
- the player can slide halfway through obstacles before being blocked by them
- there's no way to attack
- there are no enemies
- there's no way to connect multiple screen-sized room layouts to form a larger space

Before taking a break from Warrior Legend to move on to Space Battle, there's still one last step to get the example code in a state that'll be ready for Section 2 later...

warrior legend step 13

cleaning out the update markings

As usual, at this point it's time to removing any of the last round of line change //// comment marks.

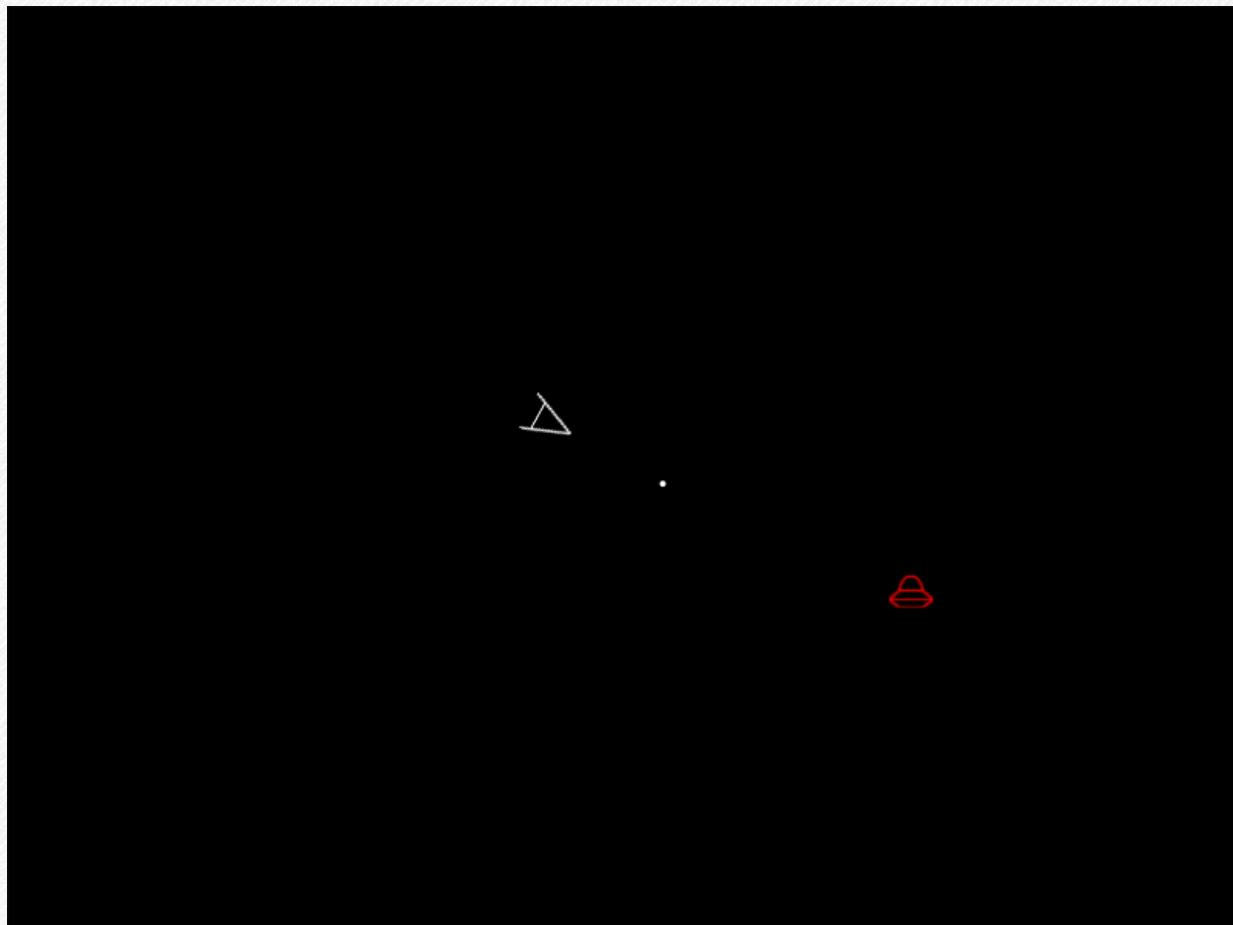
EXPECTED RESULT

Same gameplay. The codebase is prepared for the Section 2 exercises, or as the starting point for a related but more ambitious personal project.

You're making great progress! In the next game starter code you'll be working with momentum, projectiles, edge wrap, and a sweet UFO.

GAME 5: SPACE BATTLE PROJECTILES, WRAP, & ENEMIES

This game style originated in the 1970's – so it's out of chronological order relative to the other games so far – this is because it involves a number of gameplay concepts and technical issues not yet covered. In the starter code, you'll get player projectiles and a basic enemy type working. In the Section 2 exercises, you'll be adding waves of increasing difficulty, more enemies, and various kinds of attacks.



space battle step 1 copy a version of racing to start

For step 1 of Space Battle, it will be easiest to begin from a much earlier step of Warrior Legend, one that still has more in common with Racing's functionality. Even though you added many new features atop the Racing code for the later steps of Warrior Legend, virtually none of those changes are relevant to Space Battle. Many of the changes were related to the tile environment or changing the player's movement code to walk. Space Battle doesn't need a tile environment, and the player movement should work more like the overhead car (vehicle relative turning plus forward motion) than a human walking.

The reason I recommend using an early Warrior Legend step rather than Racing's final step – racing-step31 – is that the first several changes for Warrior Legend removed Player 2 and all references to "car". Because this game won't have a player 2 or car either, by using warriorLegend-step4 you avoid needing to repeat that same tear down work for this game. Copy in warriorLegend-step4 and rename it spaceBattle-step1. You don't want the code of any later step since those started switching how the player movement worked.

In addition to copying in warriorLegend-step4 and renaming the folder spaceBattle-step1, update the HTML file's name to match. This makes it much easier to tell in a browser, or editor, which project is being viewed.

Since this step was copied from an earlier project, search in each of the script files for any //// update markings and clean



them out. Step 4 of Warrior Legend was the step which removed the word “car” from most places in the code, so you’ll see quite a few of them in the `Warrior.js` file, plus a handful in `Main.js` as well.

Using this older midpoint as a starting place highlights an extra benefit to having older versions of your project code available. If you pile changes one on top of another in-place without ever saving back-ups along the way, it’d be more trouble to spin off another project from an older state of the code.

(If you don’t have an earlier step of your work saved to start from, feel free of course to instead copy it in from my provided starter code within the example solutions folders.)

EXPECTED RESULT

You’ll have the single player version of the Racing game’s starter code from an earlier stage of that project.

space battle step 2 clearing out all track elements

Goal for this step: remove World.js and any code or images connected to it. Remember to delete World.js's inclusion line in the HTML file. This change will touch a number of points in the files, but you can check yourself once the World.js file has been cut by looking for any errors in the Java console due to missing references.

The only image remaining in the project will be player1.png. In Warrior.js, do away with homeX and homeY, starting the player at the position this.x and this.y are initialized to (75, 75). Get rid of the second argument (whichName) to the Player class' init function. Additionally, delete the line in that function which saved it as myName. Finally, remove the line in Main.js which initialized this variable to "Blue", since without the finish line, there won't be a use for the player character's name.

Since the track is no longer being drawn each frame, the player car will leave a smudge trail at all prior positions. Fill the entire canvas with black each frame to prevent this.

EXPECTED RESULT

The car drives around in a black void. The entire tile track grid should be gone from both the play experience as well as the project folder's code and images. Driving out of bounds has no consequences, the car simply goes out of visible range.

space battle step 3 replace car code and art with ship

Next, replace Player 1's car image with a ship image. My retro "capital letter A" can be copied from the example solution if you'd like. Whichever image you use should face right, like so:



HTML5 JavaScript for Canvas has good support for fast and crisp line drawing. If the final visual look was meant to keep the spacecraft and UFOs looking like they did on vector/line displays long ago, you could plot and rotate the lines instead of using images for the ships. However the retro look here is just a placeholder, intended to be replaced with your own image. An image is much easier to reimagine and replace than is a list of coordinates defining a single-color set of shapes.

In the ship's reset() function, rather than starting the ship near the top-left corner of the canvas, set it to the center of the canvas (x to half of canvas.width, y to half of canvas.height).

Change all references to "warrior" (including in the Warrior.js filename, and the HTML file's inclusion of it) to "ship". In Input.js, update the thisCar parameter for setKeyHoldState() to thisShip.

EXPECTED RESULT

It's now a spaceship in the void instead of a car, although for now it will still control and move the same. Also, the vehicle will now begin in the middle of the canvas rather than near a corner, giving us more room to maneuver while updating its control code.

space battle step 4 edge wrap for ship

With the track gone, the playfield has no boundaries.

Updating the ship to slide more with momentum from thrust, and removing any brakes or reverse will make it harder to control. That difficulty in control is important to how classic games of this style work. Mastering ship control is part of play.

First, implement edge wrap so the ship instantly pops over to the opposite side when it goes beyond any visual boundary. The superior precision in the existing car-like controls will make it easier to test whether edge wrap is working.

Create a new function for the ship class in `Ship.js`, named `this.handleScreenWrap()`. Call it each frame near the end of the ship's `move()` function. Use a series of if-comparisons to check whether the ship's center exited the canvas boundaries, and if so, add or subtract a canvas dimension (width for x, height for y) to put the coordinate in a new position back within canvas boundaries across from where it slid off.

EXPECTED RESULT

Vehicle will still move the same way, but going off any edge of the canvas will instantly relocate it to the opposite side. Test this by driving in a wavy line around the edges of the canvas. A more polished implementation might support showing part of the image split across the edges when only partway across, but handling that in a robust way also does corners right and accounts for related collisions, adding up to be quite involved.

space battle step 5 slippery space without brakes

The spaceship needs to move like it's in space. To keep more momentum, increase GROUNDSPEED_DECAY_MULT from 0.94 (which is applied in a way that meant every frame it lost 6% of its speed) to 0.99 (so instead, it loses only 1% of its speed per frame). The name should change, too, since this is no longer friction for speed on the ground. Rename it to something more fitting, like SPACESPEED_DECAY_MULT.

Remove the constant REVERSE_POWER and any use of it. Do the same for keyHeld_Reverse (check Input.js too since this will mean one less argument for the setupControls() call). Get rid of MIN_TURN_SPEED, both the const declaration and where it's used in code.

Decrease DRIVE_POWER from 0.5 to 0.15, so the vehicle's acceleration is less abrupt, and rename it THRUST_POWER for consistency with its new usage

EXPECTED RESULT

The ship should take much longer to slow down to a stop, the down arrow should no longer affect it, and it should be able to turn even while sitting still. At this phase, turning the vehicle while it's sliding will still instantly change the direction of the vehicle's motion.

space battle step 6 separating facing from drift

When a moving wheeled vehicle steers, it changes direction.

When vehicle floating in space reorients, it should keep moving in the same direction, uninfluenced by its new facing unless more thrust is applied.

To achieve this effect, get rid of `this.speed` from `Ship.js` since it's a value for linear, forward-backward movement. Instead, make separate velocity variables per axis (I call mine `this.driftX` and `this.driftY`), to use as a step between the application of forward thrust (at magnitude `THRUST_POWER`) and the per-frame updates applied to the vehicle's position. You'll add `driftX` to the `x` position and `driftY` to the `y` position each frame, and use thrust to affect `driftX` and `driftY`. This extra layer of indirection between engine power and movement partially preserves the existing movement even as new thrust is applied.

For sanity, the ship should gradually slow to a stop (even though this is not realistic for space motion, just as edge wrapping is not). Since `this.speed` has been removed, use `SPACESPEED_DECAY_MULT` to decay both `driftX` and `driftY` instead. Do this near the end of `Ship.js`'s `move()` function.

EXPECTED RESULT

The ship should now be trickier to control. To slow down quickly, the ship will need to turn around and apply thrust. Instead of being able to turn a tight corner, applying thrust while turning will make the vehicle awkwardly spin and drift.

space battle step 7 giving the ship a (stationary) shot

Next comes support for a single reusable player shot. Soon the player won't be able to fire again until the current shot's time limit runs out or it hits something, but for now it can instantly restart back at the ship if fired again rapidly. First, create a stationary shot at the player's position when the spacebar is pressed. Movement and reload be added later.

Duplicate or file copy and paste Ship.js, naming this new file Shot.js to match its intended usage. You'll rework this copy to describe the shot's data and logic. The reason we're starting from a clone of Ship.js is because the shot will have some things in common with the ship. The shot needs its own position coordinates, movement variables, and functions to reset, move, draw, and even wrap at the screen edge.

Later, we'll set up a common parent class to contain the shared values and functions. For this pass, simply duplicating the ship's code will keep you free to mold the duplicated code to suit the shot's needs without stressing over whether shot-specific changes may affect the ship's workings. Once this pass is working, you'll be able to step back and look at what is or isn't 100% similar between these two classes and separate what's shared between them into a parent class.

Add Shot.js to the HTML file. In Shot.js replace the three ship const values with 3 different ones that apply to the shot:

```
const SHOT_SPEED = 6.0;  
const SHOT_LIFE = 30;  
const SHOT_DISPLAY_RADIUS = 2.0;
```



Remove the keyHeld_* variable declarations in Shot.js, as well as the Shot.js copy of the setupControls() and init() functions. For the Shot.js version of reset(), set this.shotLife = 0, which will be treated elsewhere as a signal that the shot is inactive and ready to use. Whenever shotLife is 0, the shot won't be drawn, move, or collide. Keep the handleScreenWrap() function, and right below it, create a new function called shootFrom() which accepts the player's ship as its only argument.

In shootFrom(), if shotLife is zero set this.x and this.y to the x and y coordinates of the ship that was passed in. Set this.xv and this.yv to 0.0 for a stationary shot. Set this.shotLife to the const value SHOT_LIFE to signal to other shot code that it's now meant to be active and visible:

```
this.shootFrom = function(shipFiring) {
  this.x = shipFiring.x;
  this.y = shipFiring.y;

  this.xv = 0.0;
  this.yv = 0.0;

  this.shotLife = SHOT_LIFE;
}
```

Later, the shot's move() function will handle movement, but for now its sole purpose is to count down the shot's time limit:

```
this.move = function() {
  if(this.shotLife > 0) {
    this.shotLife--;
  }
}
```

For the draw() function in Shot.js, instead of using a bitmap, call the old colorCircle() function in GraphicsCommon.js to draw a white circle at the shot's position with a radius defined

by the const SHOT_DISPLAY_RADIUS. Wrap that colorCircle() call in an if-condition that checks whether this.shotLife is greater than 0. This will hide the shot if it's not currently active.

Over in Ship.js, create a single reusable shot on the player's ship and call its functions from their counterparts in the ship code. As the first line of the ship's init() function, add a line declaring our reusable shot instance for this ship:

```
this.myShot = new shotClass();
```

On the last line of reset(), in Ship.js, call reset() on the shot:

```
this.myShot.reset();
```

Likewise on the last line of move() in Ship.js, call move() on myShot. On the first line of draw() in Ship.js, call the draw() function for myShot. By putting it in the first line, the shot will be drawn underneath the ship graphic. That will be especially noticeable if you replace the transparent, white-line ship graphic with something more solid.

Add a new function named cannonFire() which accepts no arguments and exists to call the shootFrom() function on myShot. The shootFrom() function needs a reference to which ship fired so it can copy the ship's position to the shot's starting point. The same programming keyword "this" used to reference variables within a class can also be used without the period to represent a whole instance of a class currently running its code. That's useful, for example, where the p1 instance of Ship calls the cannonFire() function, in which case "this" is interpreted as the p1 instance, complete with its position and other data:



```
this.cannonFire = function() {
  this.myShot.shootFrom(this);
}
```

The spacebar still needs to be hooked up to call the ship's `cannonFire()` function. It won't follow the same pattern as the control arrows though. Unlike the arrow keys, the spacebar doesn't need to track whether it's held. Shots are triggered at the moment the spacebar is pushed. For this reason, it doesn't need a true/false hold state variable like the other input keys. It does need a variable to store which keycode is meant to trigger the action, though.

In the `setupControls()` function add an extra argument, `shotKey`, to save as `this.controlKeyForShotFire`. Over in `Input.js`, near the top where other key const labels are defined, add a new const for the spacebar value. Saving you the trip to check what the keycode is for spacebar, it's 32:

```
const KEY_SPACEBAR = 32;
```

On the last line of `initInput()` insert that new const label as an additional argument in `setupControls()`, to fill in for the new argument added to that function:

```
p1.setupControls(KEY_UP_ARROW, KEY_LEFT_ARROW, KEY_RIGHT_ARROW, KEY_SPACEBAR);
```

Here's where the pattern for other keys will no longer fit. The other keycodes are checked in the `Input.js` function named `setKeyHoldState()`. For the firing key, the shot can be started right when the key gets pressed. In the `keyPressed()` function, add an if-condition that will directly call the player's firing function when the player's control key for firing is detected (set to spacebar's keycode in the above `setupControls()` call):

```
if(evt.keyCode == p1.controlKeyForShotFire) {  
    p1.cannonFire();  
}
```

EXPECTED RESULT

Pressing spacebar will leave a small stationary white circle at the ship's current position. The ball will vanish after a second or so. If the player presses spacebar before the previous shot disappears, the shot is instantly relocated to the ship's latest position with the shot lifetime counter started anew. If the ship is in motion when the dot gets "fired" it will remain where the ship's center was at the time of firing.

space battle step 8 giving the shot direction

Making the shot move in the direction the ship is pointing won't take nearly as much code as that previous change. The changes are a little less straightforward, though. First head to the shootFrom() function in Shot.js. Rather than setting xv and yv each to 0.0 for a stationary shot, give xv and yv values based on the ship's heading and motion:

```
this.xv = Math.cos(shipFiring.ang) * SHOT_SPEED + shipFiring.driftX;  
this.yv = Math.sin(shipFiring.ang) * SHOT_SPEED + shipFiring.driftY;
```

The first part of each sum is similar to adding thrust to the ship when the thrust key is held. That's the main factor in the shot's direction if the ship is sitting still when it fires. It is what makes the shot move away from the ship in whichever direction the ship is facing while firing.

The cos() and sin() functions can be used to reinterpret the angle as a percentage of total movement along the x and y axes. In other words, multiplying their output by SHOT_SPEED scales the x and y motion to move at SHOT_SPEED overall.

Notice that the player's thrust code uses a `+=` instead of an `=` when using sin() and cos(), and also that the player's thrust constant (THRUST_POWER, set to 0.15 at the top of Ship.js) is far lower than the spot's speed (SHOT_SPEED, defined as 6.0 at the top of Shot.js). That's because the ship's thrust is cumulative over many frames, but the shot jumps instantly to full speed when fired. The player's ship takes time to accelerate from a standstill.

Next look at the shipFiring.driftX and shipFiring.driftY values on the right half of each sum. The driftX and driftY values in the ship class store the current movement along the x and y axes, and shipFiring is a reference to which ship fired the shot. The right half of the sums is adding the ship's current speed to the shot's initial speed. This way a ship moving forward adds its speed to the shot, so that if the ship is sliding sideways when it fires, the shot will inherit the ship's slide.

It's important to use the driftX and driftY values for carrying ship momentum into the shots, rather than the ship's angle. Remember that the ship can spin to aim without using thrust in order to fire in a direction other than where it's heading.

If you test the game at this point though, the shot won't move yet. The computer won't ever assume anything on your behalf. Even though you've set xv and yv values meant to update the shot's x and y position values, they are not yet used anywhere. To make them work, add these lines in Shot.js's move():

```
this.x += this.xv;  
this.y += this.yv;  
this.handleScreenWrap();
```

Those first two lines will increment the shot's position by its velocity. That third line calls handleScreenWrap(), which was duplicated in Shot.js as part of copying Ship.js as the starting point. As an organizational step later, a single common definition of this function can be shared by both the Shot and Ship classes. In the meantime, this works by having all the code needed for the ship and shot within their respective files, even if there's a little code duplication.

EXPECTED RESULT

Pressing spacebar will send the shot forward from the ship for about one second before it vanishes. The shot should also inherit initial momentum from the ship. Firing across an edge of the playable area results in the shot wrapping promptly to the opposite edge, the same as the ship does when it flies off the edge of the canvas.

To test whether the momentum transfer works, speed up the ship by holding thrust, then release thrust and turn about 90 degrees left or right to leave the ship sliding sideways. Firing from that orientation, you should see the shot move diagonally relative to the canvas, moving straight away from the ship but sliding sideways to keep up with it.

Alternatively, confirm that if you drive forward at full speed and fire without turning, the ship shouldn't "keep up with" the shot. If you spin the ship to face backwards after building up speed, then fire opposite the direction of travel, the shot should move much more slowly relative to the screen.

Tapping spacebar will cause the currently moving shot to be instantly teleported back to the ship. It's time to fix that next.

space battle step 9

blocking fire until the shot expires

Supporting multiple shots at once is a more involved exercise, and will be handled later. For now, what needs to be fixed is that if the player fires while the shot is in motion, it teleports back to the ship mid-flight rather than waiting to finish its time.

Let's block it until it runs out of time or hits something. This way, the currently single shot in the game will be reusable. It maybe seems strange that the shot will be reader sooner if it hits something, but this isn't as broken as it seems. Many classic space and arcade action games limited the number of player shots on screen at a time to 1, 2, or 4. This has the unrealistic but exciting effect of increasing rate of fire as an instant reward for accuracy, and on the other hand this penalizes misses by dragging out the time between shots.

In the Shot.js file, create a new function above shootFrom() that returns true if the shot is ready for use (meaning its life counter is at or below 0) or false if the shot is still in action:

```
this.isShotReadyToFire = function() {  
    return (this.shotLife <= 0);  
}
```

In the Ship.js file's cannonFire() function, use this new function as a condition round the myShot.shootFrom() call, so the shot action will not be activated until the shot is ready for use:

```
if( this.myShot.isShotReadyToFire() ) {  
    this.myShot.shootFrom(this);  
}
```

The reason for creating and checking such a simple helper function isShotReadyToFire() instead of check the number



stored in this.myShot.shotLife within Ship.js's cannonFire() function, is because this reduces the need for the Ship code to know about the Shot's shotLife variable or what it means. This distinction will be increasingly useful to us as the total amount of code in the program grows.

EXPECTED RESULT

Pressing spacebar will still launch a shot as it did before, however tapping spacebar repeatedly now won't yield a followup shot (or interrupt the current shot) until the shot's time limit runs out.

space battle step 10 shared code between ship & shot

The shot and player have a lot of functionality and variables in common. They share coordinate velocity, wrap, and the part of their move functions that update the x and y coordinates by xv and yv (or the equivalent for the ship class, driftX and driftY).

Since driftX and driftY are used the same way in Ship.js as xv and yv are used in Shot.js, rename those variables in Ship.js to xv and yv for consistency. This will simplify extraction of common parts. Also find and update those variable names wherever they are used in Shot.js's shootFrom() function.

Now it's time to create a parent class which will be shared by the Ship and Shot. Duplicate Shot.js to start from. Since it's the shorter and less complicated of these two classes it will require less cutting. Rename the newly copied file MovingWrapPosition.js, and add it to the main HTML file before Ship.js and Shot.js since they will reference it:

```
<script src="js/MovingWrapPosition.js"></script>
```

Next, tear down the MovingWrapPosition.js file and its class, leaving only what's used in both the Ship.js and Shot.js files. Delete the const values at the top of MovingWrapPosition.js, and rename the class to movingWrapPositionClass. Remove the file's draw() function (drawing is handled separately by the ship and shot). Keep handleScreenWrap() since it's the same for both. Keep reset() for now, though you'll be replacing its contents later.

Remove `isShotReadyToFire()` and `shootFrom()` since they're both shot specific and not needed for the ship. In the `move()` function remove the three lines that have the if-conditional checking and decrementing `shotLife`. That code is shot only. Keep the two lines that add `xv` and `yv` to the `x` and `y` coordinates and then call `handleScreenWrap()`. That chunk of functionality is needed for both the shot and the ship.

After all this trimming, the `MovingWrapPosition.js` should be a class with three functions: `handleScreenWrap()`, `reset()`, and the three line version of `move()` that updates the `x` and `y` coordinates and calls the local `handleScreenWrap()` function. For the `reset()` function in `MovingWrapPosition.js`, clear out the `shotLife` code since it's a shot specific variable. In its place, copy the first few lines from the `Ship`'s `reset()` function that zero the `xv` and `yv` (formerly `driftX` and `driftY`) values as well as centering its `x` and `y` coordinates:

```
this.reset = function() {  
    this.xv = this.yv = 0.0;  
    this.x = canvas.width/2;  
    this.y = canvas.height/2;  
}
```

While this isn't explicitly necessary for the shot, since these are the variables `MovingWrapPosition.js` works with, it'll be convenient for the class to have a way to initialize them to sensible, defined, in-bounds default values.

For sharing the `MovingWrapPosition.js` file and class code between the `Ship` and the `Shot`, there are different ways to do it depending on the way in which their related functionality may or may not overlap as you continue working on them.

If you're coming into JavaScript from another Object-Oriented Programming language (if you're not, skip this paragraph), you might assume that there's a keyword or symbol to extend a class. However, unlike OOP languages, JavaScript is actually a prototypal language. It works a bit differently. If you search the web for how to accomplish normal OOP features using JavaScript, you'll find many smart and experienced coders disagreeing about the various ways to reconstruct or simulate class inheritance-like functionality within JS.

As always, we'll just write simple code that gets the job done.

Since my aim is to both prepare you for a variety of languages, and also to help make a connection to other programming you may have tried before, I'll show a minimal way to recreate the most basic aspects of class-like functionality for games.

Focus on Shot.js first. A line or two above the start of the shotClass definition, add this line: shotClass.prototype = new movingWrapPositionClass(); This establishes the basis of the shot class (functionally speaking, its fallback) as the definitions in movingWrapPositionClass():

```
shotClass.prototype = new movingWrapPositionClass();
```

Next, remove the entire handleScreenWrap() function from Shot.js, the version defined in movingWrapPositionClass will take care of this.

The next matter is how to call the movingWrapPositionClass ("superclass" or "parent class") version of the move() function from the shotClass ("sub class" or "child class") version of the move() function. This will also be needed for reset(). The goal is



to perform both the generic functionality (update position by velocity, and wrap position at edges) and some shot- or ship-specific functionality as well.

To call a function defined in movingWrapPositionClass from any place in shotClass, prefix the function name with the “this.” keyword to call it directly. For example, a call in the Shot’s move() function to “this.handleScreenWrap();” would work fine. This won’t work for this.move() or this.reset(), however, since by the time the code reaches them, both have been overwritten/blocked by the Shot.js specific version.

Add the following line on the line above Shot.js’s definition of this.move():

```
this.superclassMove = this.move;
```

As is always the case for variable and function names, the term “superclassMove” is just a label created here to be read by human eyes. That wording has no special significance or functionality to the computer. It could be this.elephant, it’s just saving a reference to the old move() before it is redefined.

Since that line is reached before this.move() is redefined in Shot.js, it stores a handle to the function as it is defined in the superclass movingWrapPositionClass. In the shot class’ version of the move() function, erase the two leading lines that update the x and y coordinates based on xv and yv, and remove the call to handleScreenWrap().

All that will be handled by the parent class’ move() function.

In place of those three removed lines, call the newly saved superclass version of move(), leaving the whole Shot.js move() function looking like this:

```
this.superclassMove = this.move; // saving a reference to the parent class's move
this.move = function() {
  if(this.shotLife > 0) {
    this.shotLife--;
  }
  this.superclassMove();
}
```

This more concise version makes it easy to see that the shot's position doesn't really need to be updated if its life counter is zero. Rearranging the function slightly can make it so that the motion update code in the superclass only happens if the shot is still live (instead of invisibly moving and wrapping):

```
this.superclassMove = this.move; // saving a reference to the parent class's move
this.move = function() {
  if(this.shotLife > 0) {
    this.shotLife--;
    this.superclassMove();
  }
}
```

All that's left is to make similar changes in Ship.js. Above the shipClass definition, add the following line to make a copy of movingWrapPositionClass the start for the ship's class:

```
shipClass.prototype = new movingWrapPositionClass();
```

Delete handleScreenWrap() in Ship.js, as it is redundant with the one defined in movingWrapPositionClass. On the line right above this.move() in Ship.js, save a reference to the superclass' move() function:

```
this.superclassMove = this.move; // saving a reference to the parent class's move
```

Over where the x and y components were being incremented by xv and yv (formerly driftX and driftY), above the



multiplication by SPACESPEED_DECAY_MULT, call the superclass' move() function:

```
this.superclassMove();
```

Seeing this pattern in both files, you may wonder why the common movement function isn't just named something other than move() in the first place (like superclassMove()!), instead of needing to save the local reference before each subclass redefines it. You could do that, and that would work. Doing so would even remove the need to explicitly add the line above the redefinition of the move() function in Ship.js and Shot.js. I'm favoring the extra step of this pattern here because it makes more explicit that the move() function is intended to be run from within the subclass' move() function if overwritten.

This way there's a built-in annotation within the subclass making clear that it's redefining a function from the superclass. It's admittedly a matter of style preference, though.

Follow the same process for reset() in both the ship and shot code as a bit of practice on this pattern. On the line above each local definition of this.reset(), define:

```
this.superclassReset = this.reset;
```

Call that function on the first line of each reset() function. Calling it on the first line ensures the class-specific reset() function can overwrite any values it may need to. In the ship's version of reset(), the lines setting xv and yv to 0.0 can be removed since they appear in the movingWrapPositionClass version of reset(). You could also remove the lines that center the ship, but where the natural reset value for velocity can be



assumed to be 0.0 on each axis, what constitutes `reset()` for a position is more open to interpretation. The center of the canvas is one possible answer among many.

Think of it this way: if you want to change where the player starts, should you look in the `movingWrapPositionClass` version of `reset()`, or the ship-specific version? The latter, I think, makes better sense.

Technically, the superclass version of `reset()` called in `Shot.js` has no practical effect. All 4 of the initialized variables are overwritten instantly when the player fires. When reasoning about the relationships between the superclass and the subclass, if you added code to `movingWrapPositionClass'` version of `reset()`, I think it would be reasonable to expect that code would also affect the shot.

EXPECTED RESULT

Same gameplay as the previous step. You're now in a better position to add enemy UFO ships and other moving, wrapping elements with less copy and paste code duplication.

space battle step II adding an enemy ufo

The player can shoot now, but there's nothing yet to shoot at. That will not do. The game needs a UFO.

Begin by duplicating Ship.js as a new file, UFO.js. That's the best choice since the UFO has more in common with the player ship file (a class definition, needs a graphic, inherits wrapping motion) than it does with the other files so far. In the HTML add the UFO.js on the line after Shot.js, which is also after MovingWrapPosition.js since it inherits it.

```
<script src="js/UFO.js"></script>
```

The enemy UFO will move much differently than the player ship. It will move steadily in a straight line for a fixed interval of time, then update to head a potentially different random direction. Two const values will be enough to tune this: one for how quickly it moves, the other for adjusting how frequently it will change direction. The various const values leftover from copying the Ship.js file won't be needed for the UFO.

```
const UFO_SPEED = 1.9;
const UFO_TIME_BETWEEN_CHANGE_DIR = 85;
```

Next, change the word ship in the file to UFO. It only actually appears three times, two places for the class name, and the third in a nearby comment. The UFO won't need to handle input, so remove any lines with keyHeld variables, as well as the entire setupControls() function. The UFO won't have any ability to shoot yet, so also remove any myShot lines from the UFO.js file and the entire cannonFire() function. The UFO won't

spin like the player ship, so delete any reference to `this.ang` and replace it in the `draw()` function with `0` so it'll draw upright.

Since the UFO doesn't need `const` values for speed decay either, all this editing will leave the UFO's `move()` function, for now, as no more than a call to `superclassMove()`.

Speaking of what's left after all that chopping, the file should now be much shorter. All that remains in `UFO.js` at this point:

- two UFO-related `const` values
- prototype inheritance from `movingWrapPositionClass`
- a class definition that accepts a graphic for initialization
- a `reset()` function to center its coordinates
- a `move()` function which is empty aside from a call to its `superclassMove()` function
- a `draw()` function that displays its graphic

The UFO needs a bit of custom reset and move code to get it moving around. As the last line in `UFO.js`'s `reset()` function, set a new counter variable to zero:

```
this.cyclesTilDirectionChange = 0;
```

While in the `reset()` function, rather than having the UFO default to the center of the canvas where the player is, update those `x` and `y` start values to be a random position anywhere on the canvas, like so:

```
this.x = Math.random()*canvas.width;
this.y = Math.random()*canvas.height;
```



Over in the UFO's move() function, after the superclassMove() function is called, decrement that variable. Whenever it drops below zero, pick a new random angle, set xv and yv to move the UFO at that angle, scaled to UFO_SPEED, and reset the counter to UFO_TIME_BETWEEN_CHANGE_DIR. The whole move function becomes:

```
this.superclassMove = this.move; // saving a reference to the parent class's move
this.move = function() {
    this.superclassMove();

    this.cyclesTilDirectionChange--;
    if(this.cyclesTilDirectionChange <= 0) {
        var randAng = Math.random()*Math.PI*2.0;
        this.xv = Math.cos(randAng) * UFO_SPEED;
        this.yv = Math.sin(randAng) * UFO_SPEED;
        this.cyclesTilDirectionChange = UFO_TIME_BETWEEN_CHANGE_DIR;
    }
}
```

As always, the class definition is an abstract description. To see the UFO in-game you'll need to declare an instance of it. Do that in Main.js, on the line after p1 is declared, following the same pattern, except for the UFO:

```
var enemy = new UFOClass();
```

Look wherever "p1." is referenced in Main.js as a guide for where this enemy instance needs code. One of the next appearances is where p1's init() function expects an image – playerPic in that case.

Before continuing in Main.js it's time to fix ImageLoading.js to bring in a UFO picture. Near the top of that file, set up a UFO image variable:

```
var UFOPic=document.createElement("img");
```

Down in loadImages(), add another load line to imageList to import the UFO's PNG:

```
var imageList = [
  {varName:playerPic, theFile:"player1.png"}, // note: need to add this comma, too!
  {varName:UFOPic, theFile:"ufo.png"}
];
```

For the creation of the graphic, I've provided a classic-looking ufo.png in the sample solution. It's a 29x23 pixel, transparent PNG file, drawn as the front view of an upright generic UFO. With UFOPic set up, return to Main.js and to the line after p1.init(). Now you can do the same for the UFO:

```
enemy.init(UFOPic);
```

Add calls for enemy.move() and enemy.draw() on the lines after their p1 counterparts in the Main.js move and draw code.

EXPECTED RESULT

The UFO should now start at a random position on the canvas, then wander around in straight lines and changing direction periodically. As of yet it the player shots and player ship move right through it with no consequence.

space battle step 12 crashing into and shooting the ufo

The UFO needs a little more functionality for this game starting point to be complete: it needs to be able to harm and be harmed by the player. In lieu of dealing with giving the UFO the ability to shoot on its own quite yet, at the moment the UFO will destroy the player's ship only by crashing into it, whereas the player will be able to destroy the UFO with its shot. At this point, "destroy" means to reset it and overwrite the debug text that appears below the canvas.

First, add a new const value at the top of `UFO.js`, which will be how close (in pixels) something has to be to the center of the UFO to count as hitting it:

```
const UFO_COLLISION_RADIUS = 13;
```

Why 13? Recall that the UFO image file is 29x23 pixels. With a collision radius of 13, the collision circle's diameter will be twice that, or 26 pixels, which falls roughly between the width and height of the ship. If you wanted to use a collision box rather than a distance comparison you could. For a high speed retro game like this, approximating collision checks with circles (i.e. simple distance checks) is quite common.

Next, in `UFO.js`, add a new function which will accept as its arguments an x coordinate and a y coordinate. The function should compute the distance between the UFO's x and y position and the coordinate value passed in, returning true if the distance found is less than or equal to the collision radius const and false otherwise:

```
this.isOverlappingPoint = function(testX, testY) {  
    var deltaX = testX - this.x;  
    var deltaY = testY - this.y;  
    var dist = Math.sqrt( (deltaX*deltaX) + (deltaY*deltaY) );  
    return (dist <= UFO_COLLISION_RADIUS);  
}
```

The Shot.js file will need a new function as well, this one for a shot to check whether it's overlapping an enemy instance:

```
this.hitTest = function(thisEnemy) {  
    if(this.shotLife <= 0) {  
        return false;  
    }  
    return thisEnemy.isOverlappingPoint(this.x, this.y);  
}
```

The if comparison at the top of the function ensures that if a shot is not currently active or visible, it won't accidentally count as hitting an enemy when its previous position happens to be where a UFO is flying through.

In Ship.js, another new function brings together the player ship, the ship's shot, and the enemy UFO:

```
this.checkMyShipAndShotCollisionAgainst = function(thisEnemy) {  
    if( thisEnemy.isOverlappingPoint(this.x, this.y) ) {  
        this.reset();  
        document.getElementById("debugText").innerHTML = "Player Crashed!";  
    }  
    if( this.myShot.hitTest(thisEnemy) ) {  
        thisEnemy.reset();  
        this.myShot.reset();  
        document.getElementById("debugText").innerHTML = "Enemy Blasted!";  
    }  
}
```

If the player ship's center is found to be too close to the enemy ship's center, then the player ship's position resets and the debug text below the canvas updates with the "Player Crashed!" message. If the shot is active (remember: that's checked as part of the shot's hitTest function) and is close enough to the enemy, then the enemy ship resets to a new



position, the shot gets cleared, and the "Enemy Blasted!" message appears below the game.

One more line is needed to make all this work. The recently defined function in Ship.js must be called in Main.js with the appropriate player and UFO instances filled in. It makes sense to put this in moveEverything() within Main.js:

```
p1.checkMyShipAndShotCollisionAgainst( enemy );
```

Even though there's only one player ship, with one player shot, and there's only one enemy ship, these functions are written to accept which enemy UFO is being checked as a target. Later on, that will make it easier to have more enemy ships at once.

EXPECTED RESULT

Player ship and UFO both function just as they did in the previous step, except shooting the enemy UFO will cause it to reappear elsewhere, whereas crashing into the UFO will cause the player ship to recenter. Either event will also update the text below the game's canvas to a message that provides appropriate feedback.



space battle step 13 cleaning out the update markings

Cut the leftover line change /// comment markings.

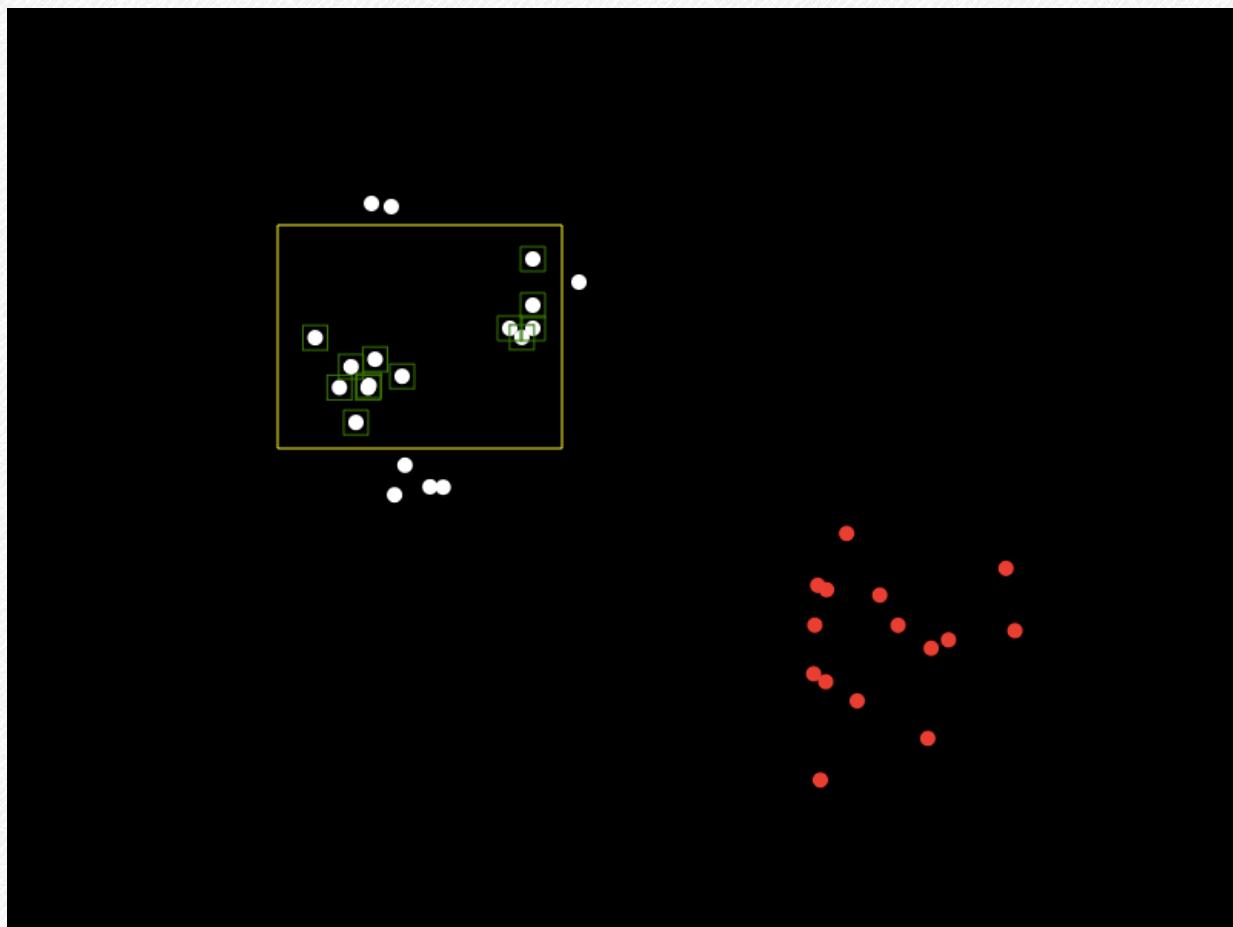
EXPECTED RESULT

No change from step 12. Once again the code is now in better shape for use as a starting point with the Space Battle exercises in Section 2 or for an original project of your own.

The next project is a bit more complex than the ones that you've been dealing with so far. It has more moving parts, more subtleties in its features, and brings up challenges for which these other games haven't scratched the surface. You've been building up to though. You're ready for it!

GAME 6: REAL-TIME STRATEGY EARLY 1990's DESKTOP-STYLE

Previous projects started by copying a related codebase as a first step, tearing it down, then building back up. I chose Real-Time Strategy, a strategy genre about commanding an active military group, as the final game type because that approach won't work here. This project will cover many concepts that weren't needed in the earlier examples. The core looks simple, but there's really quite a lot going on in this type of game.





r.t.s. game step 1

creating a new minimal foundation

For this starting point you'll want an HTML file that sets up an 800x600 canvas, has a debug text `<p>` tag with id set to "debugText", and includes 2 JavaScript files in a relative js\ folder: a GraphicsCommon.js file and a minimal Main.js.

The GraphicsCommon.js can start as a copy of the one built previously with definitions for `colorRect()`, `colorCircle()`, and `drawBitmapCenteredAtLocationWithRotation()`.

The Main.js can be based on an early version of the Tennis Game or BrickBreaker source, since those too were mouse-input games. Here's all you'll need in Main.js at first:

- The canvas and canvasContext global variables (and set up)
- A `calculateMousePos()` function which returns the canvas-relative mouse x and y coordinates
- An empty `moveEverything()` function
- A `drawEverything()` function which is empty except for a call to `colorRect()` to black out the whole canvas at the start of every frame
- The `window.onload()` function to set the canvas variables, as well as setting up calls 30 times each second to both the `moveEverything()` and `drawEverything()` functions
- A function connected to 'mousemove' events that gets the mouse position from `calculateMousePos()` and prints the coordinate to the `debugText <p>` in HTML below the canvas



Altogether, the Main.js file should look something like this:

```
// save the canvas for dimensions, and its 2d context for drawing to it
var canvas, canvasContext;

function calculateMousePos(evt) {
    var rect = canvas.getBoundingClientRect(), root = document.documentElement;

    // account for the margins, canvas position on page, scroll amount, etc.
    var mouseX = evt.clientX - rect.left - root.scrollLeft;
    var mouseY = evt.clientY - rect.top - root.scrollTop;
    return {
        x: mouseX,
        y: mouseY
    };
}

window.onload = function() {
    canvas = document.getElementById('gameCanvas');
    canvasContext = canvas.getContext('2d');

    // next few lines set up our game logic and render to happen 30 times per second
    var framesPerSecond = 30;
    setInterval(function() {
        moveEverything();
        drawEverything();
    }, 1000/framesPerSecond);

    canvas.addEventListener('mousemove', function(evt) {
        var mousePos = calculateMousePos(evt);
        document.getElementById("debugText").innerHTML = // (newline to fit page)
            "("+mousePos.x+","+mousePos.y+")";
    });
}

function moveEverything() {

}

function drawEverything() {
    // clear the game view by filling it with black
    colorRect(0, 0, canvas.width, canvas.height, 'black');
}
```

EXPECTED RESULT

You'll see an 800 by 600 pixel black rectangle on an otherwise blank white page. When you move the mouse over the canvas, the debug text updates to show the current mouse position, measured in pixels.



r.t.s. game step 2 add stationary player character

Create a new Unit.js file in the js folder, and add it to the HTML's list of included JS files. Inside that file, create a class (name it “unitClass” for consistency) with a reset() function that sets its this.x and this.y variables to the center of the canvas dimensions, and this.isDead to false. Also give the class a draw() function which, if this.isDead is false, draws a filled white circle at the unit's x, y position (remember that there's already a colorCircle() defined in GraphicsCommon.js). For the radius of the circle, define a const in the Unit.js file, name it UNIT_PLACEHOLDER_RADIUS and initialize it to 5.

Create a var named testUnit in Main.js. Initialize it to a new instance of unitClass. At the end of the window.onload() function call the unit's reset() function, and at the end of drawEverything() call the unit's draw() function.

EXPECTED RESULT

Same as before, except now there should be a small white circle drawn right in the center of the game's black canvas.



r.t.s. game step 3 random start spot, & click to move

In preparation for adding more Units, in the Unit.js file change the reset() function to start the Unit in a random spot within the left 25% of the canvas and within the top 25% of the canvas:

```
this.x = Math.random()*canvas.width/4;  
this.y = Math.random()*canvas.height/4;
```

Also add support for telling the Unit where to go. Once you have multiple Units, you'll need a way to select them. Since there's only a single test Unit we'll treat it as always selected.

In the unitClass reset() function, after setting this.x and this.y, initialize this.gotoX (set it to this.x) and this.gotoY (set it to this.y). These new variables will dictate the unit's movement target. The Unit will follow incremental movements toward its goto coordinates. Make another const at the top of Unit.js to define the unit's walking speed:

```
const UNIT_PIXELS_MOVE_RATE = 2;
```

Also define a move() function for unitClass:

```
this.move = function() {  
    if( this.x < this.gotoX ) {  
        this.x += UNIT_PIXELS_MOVE_RATE;  
    }  
    if( this.x > this.gotoX ) {  
        this.x -= UNIT_PIXELS_MOVE_RATE;  
    }  
  
    if( this.y < this.gotoY ) {  
        this.y += UNIT_PIXELS_MOVE_RATE;  
    }  
    if( this.y > this.gotoY ) {  
        this.y -= UNIT_PIXELS_MOVE_RATE;  
    }  
} // end of move function
```



The logic here may look confusing, but all it's doing is explicitly evaluating each cardinal direction to determine which way the goto target is in and taking a step toward it along each axis. "Is my goal east of me? If so move east a little. Is my goal west of me? If so move west a little. Is my goal south of me?..."

Remember to add a call to `testUnit.move()` in `moveEverything()` inside `Main.js`, otherwise this new logic won't actually be used.

Lastly, also in the `Main.js` file, add another event listener function below the one that updates the debug text to the mouse coordinates. This new function should respond to the 'click' event by setting the `testUnit.gotoX` and `testUnit.gotoY` variables to the current mouse coordinate:

```
canvas.addEventListener('click', function(evt) {  
    var mousePos = calculateMousePos(evt);  
    testUnit.gotoX = mousePos.x;  
    testUnit.gotoY = mousePos.y;  
});
```

EXPECTED RESULT

Each time the game's HTML file is refreshed the white circle should start somewhere a bit different within the top-left area of the playfield. Clicking the mouse anywhere in the black canvas should prompt the dot to move steadily (though slightly indirectly – more on that in a minute) to the position clicked.



r.t.s. game step 4

walking unit in direct, straight line

Although the test Unit successfully walks to the destination clicked, if you click in random locations you'll notice a strange pattern in how it moves. To move directly horizontal or vertical, it moves straight, otherwise it follows a 45-degree angle diagonal until it lines up horizontal or vertical. That happens since movement along each axis is done separately.

The Unit's horizontal movement is to either full speed, or zero. The vertical movement works the same way. There's no finer variation in speed. Consequently, motion is either straight up and down, straight left and right, or at a 45-degree diagonal from moving the same speed upward and over, and so on.

For the Racing and Space Battle projects `cos()` and `sin()` trigonometry functions translated an angle and a movement speed into separate horizontal and vertical components. That approach needs an angle for movement. Another trigonometry function, `atan2()`, can be used to find the angle between two coordinates (ex. from the character's current point to the target point). With that approach, the `move()` in `Unit.js` could become:

```
var deltaX = this.gotoX - this.x;
var deltaY = this.gotoY - this.y;
var moveAng = Math.atan2(deltaY, deltaX);
var moveX = UNIT_PIXELS_MOVE_RATE * Math.cos(moveAng);
var moveY = UNIT_PIXELS_MOVE_RATE * Math.sin(moveAng);
this.x += moveX;
this.y += moveY;
```

MATH EXPLANATION DETOUR (IT'S GOOD FOR YOU, AT LEAST SKIM IT)

If you're content to know that **the practical use of `atan2` is to get an angle given two points** (generally, the angle given a



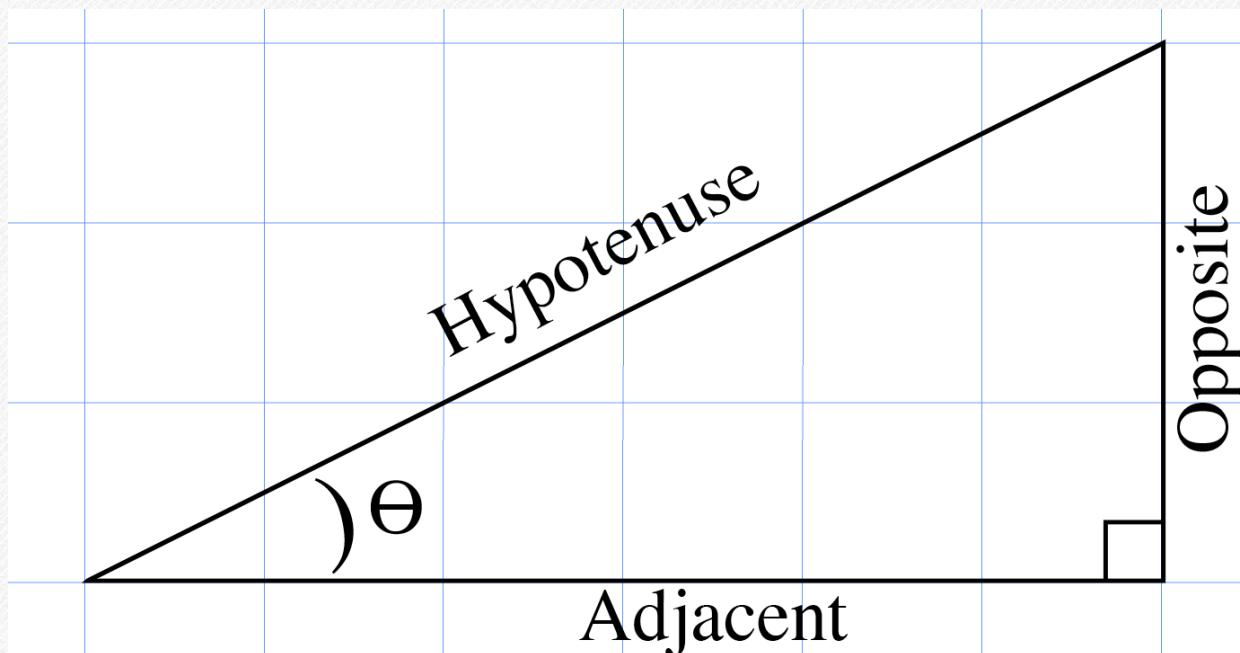
vertical measure as one argument and a horizontal measure as the other), that'll be sufficient for many common situations.

If, on the other hand, you're curious to better understand the math behind why atan2() works and what it is, here's a quick refresher. (If you've never taken or seen any trigonometry, a few pages may not be enough to make this totally clear.)

First, it'll be necessary to define what the tangent function is. It's one of the main trigonometry functions, alongside sine and cosine (those show up in code as `sin()` and `cos()` respectively). The tangent in a right triangle – any triangle with a 90 degree corner – can be written as a function of an angle Θ (theta):

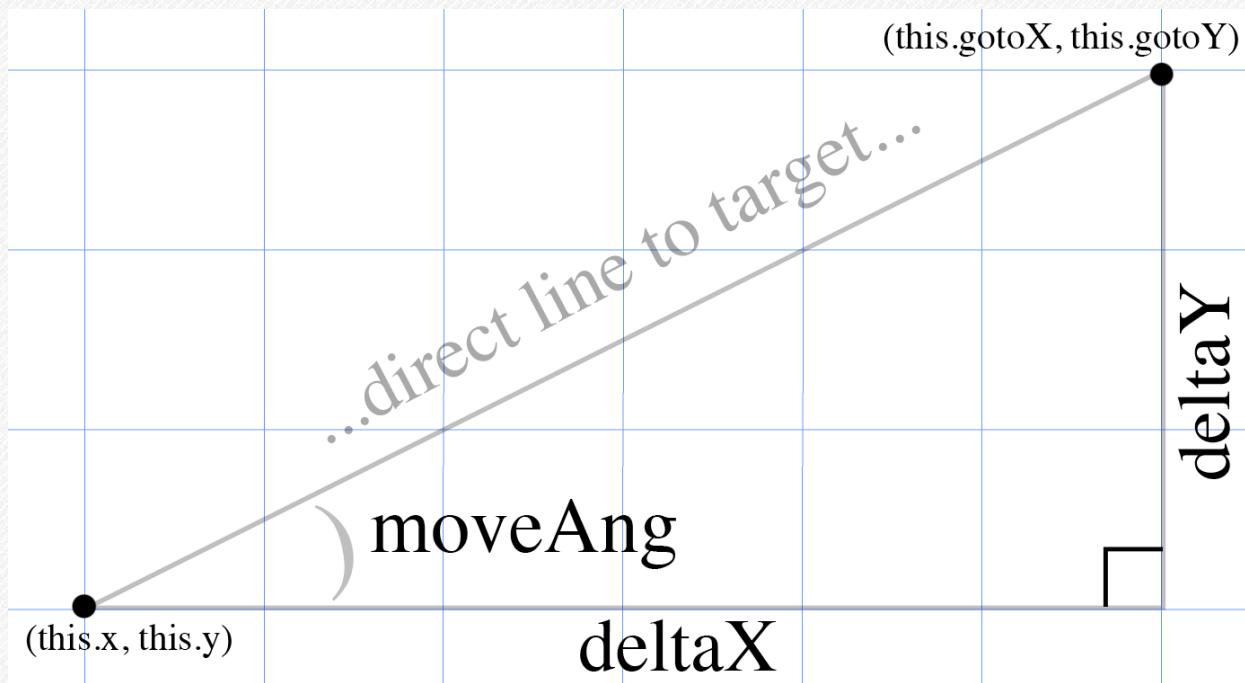
$$\tan(\Theta) = \text{opposite}/\text{adjacent}$$

There's a mnemonic SOH CAH TOA used to remember which sides each main trigonometry operation uses. It's short for Sine = Opposite/Hypotenuse, Cosine = Adjacent/Hypotenuse, Tangent = Opposite/Adjacent. The trigonometry labels:





The tangent is of special interest because the sides of the triangle it needs, opposite and adjacent, are simply the difference between the two points y ($\text{deltaY} = \text{this.gotoY}-\text{this.y}$) and x ($\text{deltaX} = \text{this.gotoX}-\text{this.x}$) values:



Substituting the variable names, `tan()` finds this:

$$\tan(\text{moveAng}) = \frac{\text{deltaY}}{\text{deltaX}}$$

You can calculate `deltaY` and `deltaX`. The `moveAng` value is the part you're trying to find. Then how does tangent help? It doesn't! However, arctangent, the opposite of tangent, does.

Arctangent, which also goes by inverse tangent, `arctan`, `atan`, or \tan^{-1} (they're all 100% the same thing) cancels out `tan()`. Since you can find the leg lengths on the right triangle, `arctan()` can be used to find the angle. First, apply it to both sides of the equals sign:

$$\arctan(\tan(\text{moveAng})) = \arctan\left(\frac{\text{deltaY}}{\text{deltaX}}\right)$$



...the arctan() and tan() cancel out, leaving behind...

$$\text{moveAng} = \arctan(\frac{\text{deltaY}}{\text{deltaX}})$$

In programming, the function for arctangent generally goes by atan() for short, and could be written **atan(deltaY/deltaX)**

The atan2 version shown in the code is a minor variant that wants 2 arguments, y as the first then x as the second, rather than a single argument expecting y divided by x. Programming **atan2(deltaY,deltaX)** in the code is similar to – though not exactly the same as – calling the one argument form as **atan(deltaY/deltaX)**. The comma in atan2() sort of takes the spot of the division in atan(), which is why *atan2() has the weird and unconventional order of the y value coming before the x value* in the pair.

There is one major difference though: atan2() also takes an important step to preserve directionality in relation to the negatives on deltaY or deltaX. Note that deltaY/deltaX, when divided, would not be able to distinguish both being positive from both being negative since the signs would cancel out.

(Special thanks to Harold Bowman-Trayford, on Twitter as @RenegadeOwner, for pointing out to me this major difference between using atan(y/x) as opposed to atan2(y,x) in code!)

Once you have the angle, you can use cos() and sin(), with moveAng as the argument in each, to convert the overall direct speed into separate speeds in the horizontal and vertical directions. This step is similar to how the trigonometry functions are used in the Racing and Space Battle games.



END OF MATH DETOUR (KIND OF), AND PUTTING A STOP TO THE RAPID SHAKING

Using just that approach, the moving dot will shake rapidly when nears the target coordinate due to eternally overstepping the target first in one direction and then in the opposite.

As a physical illustration: if in real life you can step up to one meter at a time, and walk toward a coin on the floor, when you get within one meter you shouldn't step a full meter over to the other side of it. If you did, you'd next turn around and step right back over it again. As soon as you're standing closer to the goal than your maximum stepping range, you ought to step less than your maximum. You ought to step directly to your goal and end in a stable, final position.

To do that in code, you'll need to compute the distance left to go, or `distToGo`. Distance can be found by taking the square root of the summed squares of the difference along each axis. That's a mouthful in words, but it's the standard distance formula, or the same as solving the pythagorean theorem for the hypotenuse's length:

```
var deltaX = this.gotoX - this.x;
var deltaY = this.gotoY - this.y;
var moveAng = Math.atan2(deltaY, deltaX);
var moveX = UNIT_PIXELS_MOVE_RATE * Math.cos(moveAng);
var moveY = UNIT_PIXELS_MOVE_RATE * Math.sin(moveAng);

var distToGo = Math.sqrt(deltaX*deltaX + deltaY*deltaY);

if(distToGo > UNIT_PIXELS_MOVE_RATE) {
    this.x += moveX;
    this.y += moveY;
} else {
    this.x = this.gotoX;
    this.y = this.gotoY;
}
```

That will work. No more shaking. However...



ANOTHER, SLIGHTLY MORE STRAIGHTFORWARD WAY

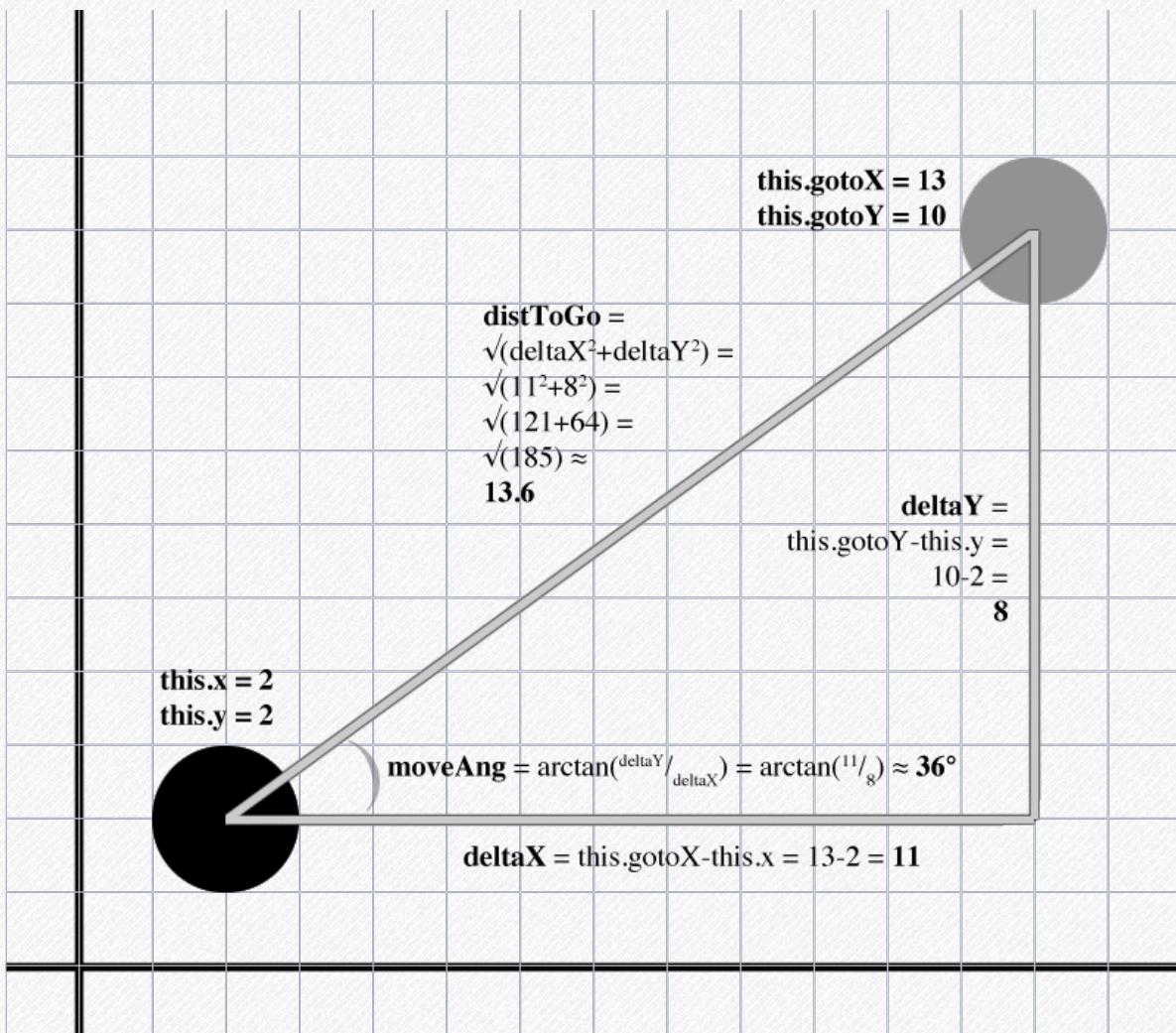
There's a way to do this that skips the angle entirely. It's especially fitting here since you're finding the distance anyway. Plus, since a walking army Unit doesn't need to rotate its bitmap like an overhead car or ship, the angle isn't needed:

```
var deltaX = this.gotoX-this.x;
var deltaY = this.gotoY-this.y;
var distToGo = Math.sqrt(deltaX*deltaX + deltaY*deltaY);
var moveX = UNIT_PIXELS_MOVE_RATE * deltaX/distToGo;
var moveY = UNIT_PIXELS_MOVE_RATE * deltaY/distToGo;

if(distToGo > UNIT_PIXELS_MOVE_RATE) {
    this.x += moveX;
    this.y += moveY;
} else {
    this.x = this.gotoX;
    this.y = this.gotoY;
}
```

Where'd the atan2(), cos(), and sin() go? Where's moveAng?

The main change to look at is how the moveX and moveY variables are computed. Either way, you want to multiply the total speed by the percentage of the total speed to be applied along the horizontal or vertical axis. In the first approach, speed was multiplied by cosine and sine of moveAng. Despite not calling any trigonometry functions in the second approach, it's also multiplying by the cosine and sine of the angle.



Note: the vertical axis is upward positive here, since that's how math classes and textbooks tend to graph points, but remember that for on-screen graphics like in most game programming environments the vertical axis is downward positive. The math works out the same. Likewise code trig functions tends to deal with angles in radians rather than degrees.

As per SOH CAH TOA mentioned earlier: Sine of an angle is Opposite over Hypotenuse, and Cosine of an angle is Adjacent over Hypotenuse. Written as variables used in our code:

$$\sin(moveAng) = \frac{\Delta Y}{distToGo}$$

$$\cos(moveAng) = \frac{\Delta X}{distToGo}$$



That's the definition of sin and cos for the angle. Because the values for `deltaX`, `deltaY`, and `distToGo` are known, the angle and trigonometry function calls aren't needed.

In vector geometry, dividing the `x` and `y` vector components by their magnitude (aka length, here it's `distToGo` in code), is called “normalizing” the vector. Normalizing scales the vector to a length of 1. That's consistent with trigonometry's unit circle lengths returned by sine and cosine.

SOME SUMMARY RULES OF THUMB

As long as you can identify when and how to apply these functions, that will be enough to work with many situations. The code doesn't care whether you can explain *why* it works. You won't need to derive these or work them out by hand.

- If you need to find the angle between two points, for example to rotate an image of a vehicle or missile, you can use `atan2(deltaY,deltaX)` to find that angle (in radians, probably).
- If you don't need the angle, and only need to move a point closer to a destination in a straight line, you can skip `atan2()` and instead divide the horizontal and vertical distance by the direct diagonal distance. Multiply that by the desired speed.

EXPECTED RESULT

After all that explaining, the only code that changed is the `move()` function in `Unit.js`, leaving this as the entire definition:



```
var deltaX = this.gotoX-this.x;
var deltaY = this.gotoY-this.y;
var distToGo = Math.sqrt(deltaX*deltaX + deltaY*deltaY);
var moveX = UNIT_PIXELS_MOVE_RATE * deltaX/distToGo;
var moveY = UNIT_PIXELS_MOVE_RATE * deltaY/distToGo;

if(distToGo > UNIT_PIXELS_MOVE_RATE) {
    this.x += moveX;
    this.y += moveY;
} else {
    this.x = this.gotoX;
    this.y = this.gotoY;
}
```

Functionally speaking, clicking anywhere should cause the white dot to move in a straight line to that point then come to a standstill until you click elsewhere.



r.t.s. game step 5

swarm of units

Still here? Made it through the trigonometry detour? Even better, enjoyed it, maybe weren't even phased by it? Or skipped it entirely, instead copy and pasting the new code?

Either and any way, you're still moving forward. Rock on.

This marks a return to general gameplay tasks. There won't be need for a separate math explanation again for a while.

Time to upgrade from a single test Unit to supporting a swarm of units to command. This functionality is at the heart of any real-time strategy game.

The Unit is already implemented as a class. Making a swarm is mainly a matter of populating an array with Unit instances and going through that list to move and draw each of them.

Especially nice about this is, the same code will work no matter whether you're dealing with a list of four units or 400.

All changes at this phase will be Main.js. Code in Unit.js won't need to be altered yet. Remove the declaration for the lone testUnit from the top of Main.js. In its place put a const value indicating how many Units you want to start with (I'll make mine 8, but you can use two, 20, or 200). After it, declare an empty array that will contain all of those units as a list:

```
const PLAYER_START_UNITS = 8;  
var playerUnits = [];
```

With the singular testUnit replaced by the list of many playerUnits, look in Main.js for functions which are called on testUnit and in those places instead call the same function for



each element in the list. Previously testUnit.move() was in the moveEverything() function, but the upgraded moveEverything() function will now apply this approach:

```
function moveEverything() {  
    for(var i=0;i<playerUnits.length;i++) {  
        playerUnits[i].move();  
    }  
}
```

This can be read as “for every Unit in the array (or list) playerUnits[], call each one’s move() function.” If instead of using an array you had declared testUnit1, testUnit2, testUnit3, and testUnit4... then this would be the same as typing out testUnit1.move(); testUnit2.move(); and so on.

This pattern can be repeated in drawEverything() to call draw() on each element of playerUnits[].

As a slight exception, where you previously called reset() on testUnit in the window.onload setup code, you’ll now need to inject a couple extra lines. In addition to resetting each Unit, this is also a good place to populate the playerUnits array with however many separates instances of unitClass you need:

```
for(var i=0;i<PLAYER_START_UNITS;i++) {  
    var spawnUnit = new unitClass();  
    spawnUnit.reset();  
    playerUnits.push(spawnUnit);  
}
```

This example is a bit different since rather than being able to iterate based on the length of the array – since it starts empty – the loop is iterating a fixed number of times based on the PLAYER_START_UNITS constant defined at the top of the file. The for-loop is counting up to eight (because I set



PLAY...UNITS to 8), running the code inside its braces once each time through.

Rather than accessing entries in the array by index, every lap a new instance of unitClass is made and saved as a temporary local variable, spawnUnit. That way reset() can be called on the instance, then on the next line the array function push() adds the newly reset() instance to the playerUnits array.

The first pass through the for-loop is creating and calling reset() on what will later be accessed directly as playerUnits[0]. The second lap creates and calls reset() on the instance playerUnits[1]... and so on, up to playerUnits[7]. Since eight units are made, and array addresses start at 0 in JavaScript (and most programming languages), there's no playerUnits[8] instance, just instances in positions 0,1,2,3,4,5,6, and 7.

The other place testUnit is referenced is where the mouse click event is handled, instructing the Unit where it should move to next. This applies to multiple units. Because this code interacts with the same Unit instance twice, once to set gotoX and once to set gotoY, there's a way to get the instance by index only once per lap and then set the values on that Unit for gotoX and gotoY:

```
for(var i=0;i<playerUnits.length;i++) {  
    var eachUnit = playerUnits[i];  
    eachUnit.gotoX = mousePos.x;  
    eachUnit.gotoY = mousePos.y;  
}
```

You could use playerUnits[i].gotoX and playerUnits[i].gotoY instead of making and using the local variable eachUnit. It would accomplish exactly the same thing with no difference in



performance or functionality. This is a common way of dealing with array elements though, since it reduces the chance for error. It's useful to be acquainted with this use of an array.

Of special importance is that when the local variable is set to `playerUnits[i]`, you can then change or call functions on the local variable and it will directly affect whichever playerUnit instance it was set to. Changing `gotoX` and `gotoY` on `eachUnit` is the same as changing those values on `playerUnits[i]`, it's not making and changing a copy of it.

The equals sign is creating a reference to that same instance on the list. If it were creating a copy instead, you'd need to end the block of code in the for-loop by setting `playerUnits[i] = eachUnit`; to duplicate the changed `gotoX` and `gotoY` variables back over to it. To be clear: you don't need to do that. The variable and the array instance are both referring to the exact same set of values and functions in memory.

This approach makes the code a tad easier to read. It can technically also provide a very, very slight performance improvement, since a tiny amount of work happens for the computer to access an array element by its index. The difference in performance is completely negligible in this case, so that isn't a motivation or practical consideration here.

I've contrived this example a bit to ensure you'll know how to make sense of it if you see it in someone else's code. The issue could have been avoided had I suggested adding a helper function, through which you'd pass updated `gotoX` and



gotoY values as two parameters in a single call. That might look like: playerUnits[i].setGoto(mousePos.x,mousePos.y);

EXPECTED RESULT

Opening the HTML file in browser should now show a small swarm of dots scattered around the top-left area of the playfield. Clicking anywhere will direct them all to move to the position clicked. There's no way to select only a fraction of the Units yet – all Units are, for now, selected all of the time.

Because they're all sharing the exact same destination coordinate from each move instruction, once they call catch up, they'll overlap and appear to be acting as one dot. There's still eight, but when they all move from the same second position to the same third position and so on, all at the same speed, there's no way to tell there's more than one.

Spreading them out will address this problem.



r.t.s. game step 6 spread destinations to see group

Ensuring the units do not exactly overlap upon completing their movement instruction can be done in much the same way that you're making sure they don't all overlap when they are created: add a little randomization to the coordinates.

The actual math and programming here are straightforward. Add a randomized offset to the horizontal and vertical positions of a unit's goto destination when giving it an order.

The slight challenge here comes down to one of how to design and think about the game code. Does it make more sense for that randomized offset to be in Main.js, where the mouse click is ordering units to relocate, or should it be in Unit.js someplace, and left to each Unit to be responsible for not standing on the toes of another?

Put another way: is it up to the Commanding Officer (mouse code) or individual Troops (Unit.js code) to be responsible for avoiding collisions near the ordered position?

If I tell 10 people to go take cover behind a wall, is it on me to give each one precise instructions for where to stand? Can't I entrust the people to all go approximately where I asked, then spread out on their own? I find the latter seems more natural. While this sort of thinking about physical reality doesn't always lead to an optimal choice in every situation, it's useful. If it makes justifiable real-world sense then it's more likely to be what another programmer reading my code (or, equivalently,



me reading my own code a few months later!) would expect in terms of where to look to find the scatter code in question.

This minor randomization won't be the final answer to how to organize a group of commanded troops. This is a placeholder approach, a first pass that's good enough so you can move on to deal with issues like troop selection/deselection. That functionality would be far harder to test if troops were converging on the exact same spot after each order.

As is often the case with these kinds of details, it's best to not get too caught up on worrying about the perfect answer. Do something that works well enough to unlock further progress. It can be revisited and reworked if needed for a more complex approach is discovered later.

Since it will be the unit's responsibility to randomize its own offset, there's now a good reason to send its movement position through a helper function instead of setting gotoX and gotoY directly from Main.js's mouse handler. If the function were going to set gotoX and gotoY exactly, you could do something like setGoto(mousePos.x,mousePos.y); Since each Unit will have leeway for how far from the clicked point it will move, a more descriptive name for the function is one which accurately reflects its ambiguity. In Unit.js:

```
this.gotoNear = function(aroundX, aroundY) {
  this.gotoX = aroundX + Math.random()*UNIT_MAX RAND DIST FROM WALK TARGET;
  this.gotoY = aroundY + Math.random()*UNIT_MAX RAND DIST FROM WALK TARGET;
}
```

This uses a new cons value; add it to the top of Unit.js:
UNIT_MAX RAND DIST FROM WALK TARGET. I'd suggest 50 for it will work well enough as a starting point. By sending



the mouse click coordinates to every Unit through this function, each will pick a random position up to 50 pixels east of the mouse and up to 50 pixels south of the mouse. This solution isn't ideal, and a better one will be covered later. For the moment it at least prevents most units from overlapping when they're commanded as a group. A selected swarm will remain a swarm even after relocating. This also makes it easier to tell how crowded and numerous a group of Units is.

In Main.js, the click function no longer needs the temporary variable introduced in the previous step. By using a function to relay both coordinates at once, the goto command is simplified to a single line:

```
canvas.addEventListener('click', function(evt) {
    var mousePos = calculateMousePos(evt);
    for(var i=0;i<playerUnits.length;i++) {
        playerUnits[i].gotoNear(mousePos.x, mousePos.y);
    }
});
```

EXPECTED RESULT

Directing the Units will no longer cause them all to converge onto the exact same location. Wherever you click, the mob of units will head generally toward that area, but will each decide on its own how far from the command point it'll move to. Each time the group is relocated, it will shuffle which units are where within the squad. Units may still occasionally overlap, often at least partly. In general though, the units will now be spread out enough to work on selecting and commanding specific subgroups of the units.



r.t.s. game step 7

click and drag selection lasso

Changing from a single Unit under control to a whole group of them is a major improvement to the gameplay mechanics. In order to select specific groupings of the troops it will be helpful to be able to trace a box (or “lasso”) with the mouse. This type of input is an expected convention for this genre.

Conceptually making the box involves two chunks of work:

1. Being able to click and drag the mouse to create a “lasso” rectangular outline which, when released, will select those Units that are located within its boundaries.
2. APopulating a separate array which lists only those units which are currently selected by the player. Units that fall within the most recent selection boundaries are put on this list. When you click to command units, the code iterates over the list selected units instead of the full list used for drawing and updating units.

This is sort of an extension to the second chunk of work, but adding some additional visual indication of which Units are on the selected list will aid in both usability and debugging.

As the functionality for the mouse grows, it may make sense to migrate much of it into a separate JS file. For now, it may be a little easier to keep it in one place to reduce flipping back and forth.

On to the lasso.



Near the top of Main.js, set four new variables to 0: lassoX1, lassoY1, lassoX2, and lassoY2. Use lassoX1 and lassoY1 to track the spot where the mouse drag begins. Use lassoX2 and lassoY2 to store the mouse's current position. When the mouse button is released, you'll be able to compare troop positions to these two corner points to determine which are standing in the selection area. Near these four new variables, set up an additional one, isMouseDragging, and set it to false. You'll change this to true between mouse press and mouse release events.

```
var lassoX1 = 0;
var lassoY1 = 0;
var lassoX2 = 0;
var lassoY2 = 0;
var isMouseDragging = false;
```

While you work on getting the box drag functionality going, it'll be a bit easier if you don't have the swarm of Units constantly shuffling around each time the mouse clicks. Comment out the 'click' event handling from the initialization code. It'll be back soon, in a way, for distributing group commands.

Much as the freshly commented-out initialization code attached functionality for a 'click' event, there are several other kinds of mouse button events that JavaScript can connect functions to. Copy and paste to duplicate your 'click' event handler twice, setting up one for the 'mousedown' event and the other for the 'mouseup' event.

The 'mousedown' and 'mouseup' events are different than the 'click' event in they don't presume or filter by any duration or movement. Click assumes the mouse goes down then comes



back up very quickly with minimal movement, or it won't fire. The down and up events on the other hand, are called any time the mouse button goes down or up on the canvas. This leaves control over how to interpret the gesture in your hands.

In the 'mousedown' handling function, set all four lasso coordinates to match the current mouse position. Set the drag variable to true there.

In the 'mouseup' handling, simply set that dragging variable back to false – soon this will check which units fit inside the lasso coordinates at the time of release.

As for the actual mouse drag updates: you already have a 'mousemove' event that's updating the debug text with the mouse position every time the mouse moves. In the 'mousemove' function, whenever the isMouseDragging value is true (meaning the mouse button is down), update the lasso's second corner to the current mouse position. Here are the mouse handlers in window.onload's setup code:

```
canvas.addEventListener('mousemove', function(evt) {
    var mousePos = calculateMousePos(evt);
    document.getElementById("debugText").innerHTML = "("+mousePos.x+","+mousePos.y+")";
    if(isMouseDragging) {
        lassoX2 = mousePos.x;
        lassoY2 = mousePos.y;
    }
});

canvas.addEventListener('mousedown', function(evt) {
    var mousePos = calculateMousePos(evt);
    lassoX1 = mousePos.x;
    lassoY1 = mousePos.y;
    lassoX2 = lassoX1;
    lassoY2 = lassoY1;
    isMouseDragging = true;
});
```



```
canvas.addEventListener('mouseup', function(evt) {
    isMouseDragging = false;
} );

/*canvas.addEventListener('click', function(evt) {
    var mousePos = calculateMousePos(evt);
    for(var i=0;i<playerUnits.length;i++) {
        playerUnits[i].gotoNear(mousePos.x, mousePos.y);
    }
} ); */
```

Near the end of drawEverything() and while the mouse button is held, draw a yellow rectangular outline between the lasso corner coordinates:

```
if(isMouseDragging) {
    coloredOutlineRectCornerToCorner(lassoX1, lassoY1, lassoX2, lassoY2, 'yellow');
}
```

Putting that near the end of the program's draw code is important because the selection box should overlap the Units, not the other way around. The selection box is part of the GUI (Graphical User Interface) layer and should always be shown in front of whatever objects are in the scene.

The draw function called there doesn't exist yet. Head over to GraphicsCommon.js to create it. It needs to draw a rectangular outline of a given color when given corner points:

```
function coloredOutlineRectCornerToCorner(corner1X, corner1Y, corner2X, corner2Y, lineColor) {
    canvasContext.strokeStyle = lineColor;
    canvasContext.beginPath();
    canvasContext.rect(corner1X, corner1Y, corner2X - corner1X, corner2Y - corner1Y);
    canvasContext.stroke();
}
```

These graphics context calls are different than the ones used for filling a shape. Instead of passing the color to fillStyle, it is passed to strokeStyle since the stroke (the line portion of the shape) is what we're drawing. Instead of fillRect() we're calling



the plain `rect()` function. The `stroke()` call traces the rectangular shape. The `beginPath()` command prevents the `stroke()` function from affecting other shapes, otherwise it could affect the circles drawn to show Units.

The `rect()` function, like `fillRect()`, doesn't want a start and end corner coordinate. Instead, the second two arguments are the width and height of the rectangle. By subtracting the second corner's x and y values from the first corner's, you're able to translate them into width and height values. This works for negatives too, in case the second point is leftward or upward from the first corner rather than down and to the right.

The other rectangle draw function in `graphicsCommon.js`, the one for a filled shape, expects width and height as its third and fourth arguments. Breaking that convention in this new function risks making a confusing bug later due to mixing up whether those third and fourth values ought to be width and height or a second corner coordinate. The “CornerToCorner” part of the function name is added to emphasize in a visually noticeable way that this function expects a corner for not only the first pair but also the second pair of parameters.

EXPECTED RESULT

Clicking and dragging should stretch a yellow rectangle. One corner remains where the mouse button was pressed down, and the other follows the mouse until the mouse button goes back up. When the mouse button is released, the rectangle should promptly vanish. In this phase, there's no way to command or select the units.



r.t.s. game step 8

use selection box to choose units

Create another array in Main.js named selectedUnits below the playerUnits array:

```
var selectedUnits = [];
```

In the ‘mousemove’ handler there’s an old debug output line which is sending the mouse coordinates to the output text every time the mouse moves. Go ahead and remove that line, the one that begins `document.getElementById("debugText")`, so that you’ll have an easier time setting up a debug output related to the new selection functionality.

Having a function in Unit.js that can determine whether it’s currently standing within a box will reduce the complexity of code in Main.js for checking each Unit against the lasso:

```
this.isInBox = function(leftX,topY,rightX,bottomY) {
  if(this.x < leftX) {
    return false;
  }
  if(this.y < topY) {
    return false;
  }
  if(this.x > rightX) {
    return false;
  }
  if(this.y > bottomY) {
    return false;
  }
  return true;
}
```

Because they are global, the lasso variables could be used directly, however, passing the corner coordinates into this function as parameters makes it more versatile for other uses, like determining whether a Unit is currently within the canvas.



Additionally, there are many different ways to determine whether a point is within a rectangle. Some are more compact, more versatile, or slightly higher performance.

For first getting it to work at all, I'm favoring one of the most straightforward and uncomplicated ways to write the comparison: if the point is left of the left side, above the top edge, right of the right side, or below the bottom edge, then the point isn't inside the box. If it passes all of those conditions, then by the process of elimination, the dot must be inside the boundaries.

Another helpful function for the unitClass to have is one that draws a colored rectangle around the unit's location. That can be used to show at a glance which Units are selected. Without something like this, it could be much more difficult to check with certainty when and whether the selection box and deselection actions are working as expected.

Adding an extra const value near the top of Unit.js, on the line after the UNIT_PLACEHOLDER_RADIUS, gives us a way to easily tweak how large a margin the selection box will have around each Unit ("dim" is for dimension):

```
const UNIT_SELECT_DIM_HALF = UNIT_PLACEHOLDER_RADIUS+3;
```

Note how this const value is based on an earlier one. It's defining the margin for the box as 3. Even if you later adjust the size of the placeholder circle graphic, the selection box will still be three pixels out along each edge.

Before the Unit.js draw() function, create another function just for drawing the box outline around a Unit:



```
this.drawSelectionBox = function() {
    coloredOutlineRectCornerToCorner( this.x-UNIT_SELECT_DIM_HALF,
        this.y-UNIT_SELECT_DIM_HALF,
        this.x+UNIT_SELECT_DIM_HALF,
        this.y+UNIT_SELECT_DIM_HALF, 'green' );
}
```

This deals only with the outline. The actual character image is left to be drawn by the existing draw() function. As per its name, the coloredOutlineRectCornerToCorner() draw function expects its arguments to be x and y coordinates of each corner. Here, they are set as offsets from the center.

Each frame, all Unit graphics (circles) are drawn first. On a second pass, boxes are drawn around the selected units. This prevents Unit graphics from ever overlapping selection boxes.

Main.js should loop through, and call drawSelectionBox() on, units in the selectedUnits array, not on all units in playerUnits:

```
function drawEverything() {
    // clear the game view by filling it with black
    colorRect(0, 0, canvas.width, canvas.height, 'black');

    for(var i=0;i<playerUnits.length;i++) {
        playerUnits[i].draw();
    }

    for(var i=0;i<selectedUnits.length;i++) {
        selectedUnits[i].drawSelectionBox();
    }

    if(isMouseDragging) {
        coloredOutlineRectCornerToCorner(lassoX1, lassoY1, lassoX2, lassoY2, 'yellow');
    }
}
```

Lastly, back in Main.js, the ‘mouseup’ handler function can be updated to check the isInBox() function for each Unit to determine which belong in the selectedUnits array:



```
canvas.addEventListener('mouseup', function(evt) {
    isMouseDragging = false;

    selectedUnits = []; // clear the selection array

    for(var i=0;i<playerUnits.length;i++) {
        if( playerUnits[i].isInBox(lassoX1, lassoY1, lassoX2, lassoY2) ) {
            selectedUnits.push(playerUnits[i]);
        }
    }
    document.getElementById("debugText").innerHTML = "Selected " +
        selectedUnits.length + " units";
} );
```

The drag variable is still set to false, since the mouse button is released at this time. The code after that is new. It starts by resetting the selection list to empty, then checking through all playerUnit instances for any that are within the lasso box. Any that pass that test get a push() into the selectedUnits array. Afterward, the debug text below the playfield updates showing how many units were netted by the most recent drag.

EXPECTED RESULT

At this time, the lasso will only work if the initial anchor point is up and to the left of the end coordinate. In other words, you have to use a down-right sweeping drag motion for the box to work, otherwise Units won't be found. There's also not yet a way to issue commands to the selected Units.

What should be working though: You can selectively box in Units, clicking and dragging from above-left of them to below-right of them. Units currently selected should have a box drawn around them, and each time you release the mouse button, a text number below the playfield should clearly state how many units you've currently selected.



r.t.s. game step 9 lasso box working in any direction

The selection box currently only works if the anchor point (start coordinate for the drag) is upward and leftward of the final release point (end coordinate for the drag). This is terribly counterintuitive, contrary to conventions established not only within the real-time strategy genre but even within people's common interactions with files on computers. Dragging any direction needs to work equally well.

The current version of `isInBox()` expects the first x value to be left of the second, and the first y value to be above the second. This is hinted at by the names of the parameters (`leftX`, `topY`, `rightX`, and `bottomY`), but it's the comparison logic within the function that is written in a way that relies on `leftX` being left of `rightX` and `topY` being above `bottomY`.

One approach you could take, which would yield a more compact function at the expense of readability, would be to think up some math trick that could be used to check whether one point is between two others independent of their order. Here's one such math trick:

```
this.isInBox = function(x1,y1,x2,y2) {  
    return (this.x-x1)*(this.x-x2) < 0 &&  
        (this.y-y1)*(this.y-y2) < 0;  
}
```

If `this.x` is in between two numbers, then subtracting both numbers from it should yield one positive (left of it) and one negative (right of it). Multiplying a negative times a positive yields a negative (`<0`). If both differences are either positive or negative, then their product will be greater than 0. If the same



test passes for y (the `&&` can be read as “AND also” meaning both comparisons must be true or it all yields false), then the Unit has been determined to be within the horizontal and vertical boundaries and therefore is within the rectangle.

That has a problem though: why it works is hard to visualize. If the solution you code is hard to visualize, that can make it harder to troubleshoot, extend, or use with confidence later.

With that in mind, here’s a more readable alternative solution:

```
this.isInBox = function(x1,y1,x2,y2) {
  var leftX, rightX;
  if(x1 < x2) {
    leftX = x1;
    rightX = x2;
  } else {
    leftX = x2;
    rightX = x1;
  }

  var topY, bottomY;
  if(y1 < y2) {
    topY = y1;
    bottomY = y2;
  } else {
    topY = y2;
    bottomY = y1;
  }

  if(this.x < leftX) {
    return false;
  }
  if(this.y < topY) {
    return false;
  }
  if(this.x > rightX) {
    return false;
  }
  if(this.y > bottomY) {
    return false;
  }
  return true;
}
```

Though that’s slightly less efficient, the usual optimization disclaimer applies here too. Unless you’re calling this function



thousands of times per frame, the infinitesimal difference in that function's run time doesn't justify making code harder to read. In the second form it's much more clear what is being done and why it works. It manually checks to determine which coordinate value is left of the other, uses that as the left value, and so on. It then performs a logical series of comparisons to assess whether the point lies within the selection boundaries.

The other code showed first is compact and efficient, but too cryptic. Readability of code matters. To reiterate a point worth frequent reiteration: readability is not only for when other programmers need to make sense of what you wrote. It's also helpful for you to be able to come back to your own code later and easily make sense of (or extend upon) what you were thinking and doing at the time.

EXPECTED RESULT

Dragging the lasso from any direction to any direction will now select Units. Successive drags will change the selection to whichever have been looped most recently. If you drag a box where there are no Units then the selection will be cleared.

There's still not yet a way to command the selected Units.



r.t.s. game step 10 command selected units

In order to command the selected troops via mouse click, you'll need to program a test for distinguishing whether a mouse down then up action was meant by the user to perform a click-drag selection or an in-place click. Various criteria could be used for making this distinction. How long the mouse button was held is one valid type of consideration. How far the mouse moved from mouse up until mouse down is another.

The latter is appealing since you're already tracking the mouse press and release coordinates. If that distance is below a threshold, interpret it as a click, otherwise let the selection box action takeover. Using only this test, and doing it only when the mouse is released, causes odd behavior in certain cases. For example, you can drag to form a big selection area, and then move the mouse back to where it started, which would be interpreted as a move order. Such cases could be detected and dealt with later as a polish detail. Measuring the anchor to release distance is a reliable enough way to tell a drag from a click.

You may be wondering why the built-in 'click' event shouldn't be trusted and have HTML5 dictate whether input counts as a click. For that matter, a quick web search shows there are also mouse events in HTML5 for 'drag' related actions. That sure sounds applicable. Why go to the trouble of hijacking the 'mousemove' and 'mouseup' functions to manually do what it looks like built-in code could be used to accomplish?



There are a few reasons.

1. When using the HTML5 events, the ‘click’, ‘mousedown’, and ‘mouseup’ events could all fire for a single click event. This makes it difficult to ensure the game only does one behavior or the other (either selecting or commanding units).
2. The ‘drag’ mouse event features in HTML5 are specifically designed for draggable objects. That solution might fit if you wanted to drag an icon, or reposition a shape. It’s not a good fit for using drag to form a loop between two abstract coordinates in empty space.
3. Although we’re using HTML5/JS as the programming target, these steps, exercises, and approaches illustrate patterns for handling similar challenges for a variety of platforms and languages. Since my aim is to teach game development basics first, and HTML5 particulars second, I’m using a manual method that can be reliably reproduced in a number of other common platforms and programming languages.

Begin by defining a tuning const at the top of Main.js which specifies how much the mouse needs to move to count as a drag action rather than a click:

```
const MIN_DIST_TO_COUNT_DRAG = 10;
```

Those Units, as with most distance and positional numbers in our code, are in pixels. For perspective on how large or small that is, the Unit.js const for UNIT_PLACEHOLDER_RADIUS is 5. That means the circles used to represent each Unit have a diameter of 10 pixels. This means any motion larger than a Unit’s size will count as a drag for performing selection.



Create a helper function in Main.js that will check whether the distance from the first lasso corner to the second exceeds this measure. I'm putting mine below the calculateMousePos() definition simply to keep mouse-related functions together:

```
function mouseMovedEnoughToTreatAsDrag() {  
    var deltaX = lassoX1 - lassoX2;  
    var deltaY = lassoY1 - lassoY2;  
    var dragDist = Math.sqrt(deltaX*deltaX + deltaY*deltaY);  
    return ( dragDist > MIN_DIST_TO_COUNT_DRAG );  
}
```

In the ‘mouseup’ handling, use this new helper function as a filter for when to execute the previously coded selection-box behavior. If, instead, the test finds the mouse didn’t move enough, use the command behavior of the sort previously assigned to the ‘click’ handler:

```
canvas.addEventListener('mouseup', function(evt) {  
    isMouseDragging = false;  
  
    if(mouseMovedEnoughToTreatAsDrag()) {  
        selectedUnits = []; // clear the selection array  
  
        for(var i=0;i<playerUnits.length;i++) {  
            if( playerUnits[i].isInBox(lassoX1 ,lassoY1, lassoX2, lassoY2) ) {  
                selectedUnits.push(playerUnits[i]);  
            }  
        }  
        document.getElementById("debugText").innerHTML = "Selected " +  
            selectedUnits.length + " units";  
    } else { // mouse didn't move far, treat as click for move command  
        var mousePos = calculateMousePos(evt);  
        for(var i=0;i<selectedUnits.length;i++) {  
            selectedUnits[i].gotoNear(mousePos.x, mousePos.y);  
        }  
        document.getElementById("debugText").innerHTML =  
            "Moving to ("+mousePos.x+","+mousePos.y+")";  
    }  
});
```

No matter how far the mouse moved, still set the dragging flag to false. This prevents the selection box from being drawn



after the mouse button is released. Then the behavior splits based on the helper function's check on the lasso size.

The previously commented out 'click' case can now be removed. It was left temporarily just to borrow from when working on the command case in this code. In addition to the debugText output, the adaptation here switched the control loop to apply to the selectedUnits rather than all playerUnits.

EXPECTED RESULT

The core experience (gameplay and player input) still lacks action, but is beginning to resemble a real-time strategy game. Clicking and dragging the mouse should select Units, no matter which direction the selection box is dragged in. Clicking elsewhere without dragging the mouse should then direct any selected units to move near the point clicked.

Because each Unit has its own target destination it's possible to issue commands to multiple groups of units which will be carried out at the same time. With quick mouse inputs you can send one group to the northeast, another to the corner in the southeast, and the remainder to the middle of the map. The orders should be carried out without any Units waiting for the others to finish their own instructions.



r.t.s. game step II rows formation for moved group

Earlier a quick kludge was added to keep a group of ordered Units from winding up atop one another. A random horizontal and vertical offset was added to each Unit commanded. This makes it possible to select Units and groups separately even after they've been commanded to move to the same point.

There are clear drawbacks to this random approach. Because the maximum offset is the same regardless of how many Units are moved, when giving move instructions to a lone Unit, it seems to have an especially broad interpretation of the position you ordered it to. On the other hand, moving a full group with many units has high odds of a few clusters overlapping. The area covered by a directed group should be a function of how many Units are in the group, so that a lone Unit gets very specific move orders, but a large squad will take up an amount of room appropriate for its numbers.

The solution could stop there. A new argument to the Unit class's `gotoNear()` function could accept a maximum scatter range based on how many Units are in the commanded group. Done in that way, there would still be a problem with random clumping of Units.

The solution is to set up the commanded Units to each find a distinct position in a rank-and-file formation. This will make the number of Units in the group easier to estimate, and provides a high level of control for choosing a subset to command. This



approach also keeps Units as close together as they can be to their destination without being overly crowded.

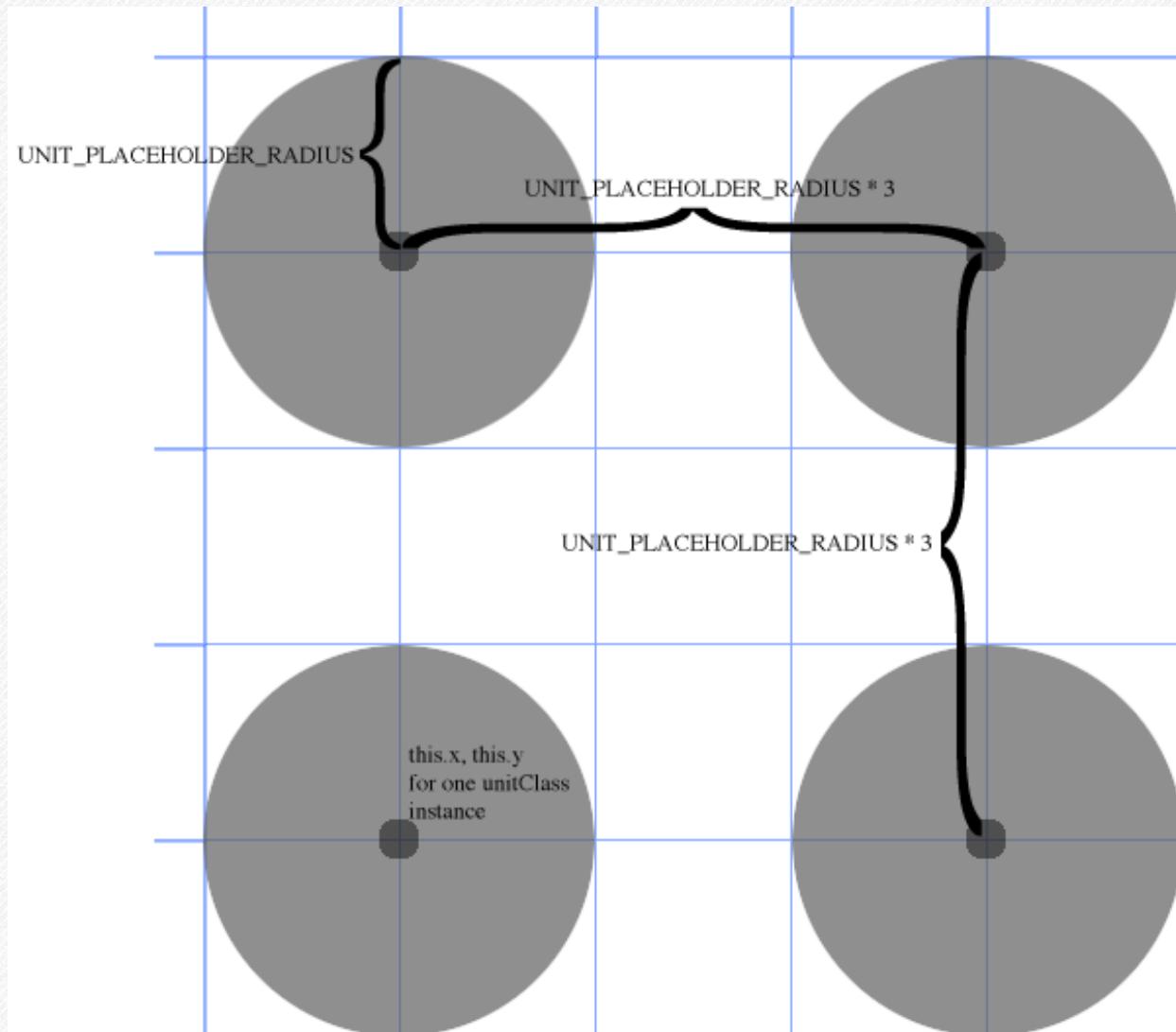
A solution this rigid and mathematically clean may not be appropriate for all situations in which you're coordinating a swarm of grouped entities for videogame programming. Given that the Units for a real-time strategy game are often trained soldiers, self-organizing into ranks is pretty appropriate.

For testing, it will be helpful to have more than eight units to experiment with. Increase the const value in Main.js for PLAYER_START_UNITS to 20. While updating and adjusting tuning constants, in Unit.js there will no longer be a need for the const UNIT_MAX RAND DIST FROM WALK TARGET, since Units will no longer be randomly spaced. Create a new const named UNIT_RANKS_SPACING and set it to 3 times the UNIT_PLACEHOLDER_RADIUS:

```
const UNIT_RANKS_SPACING = UNIT_PLACEHOLDER_RADIUS*3;
```

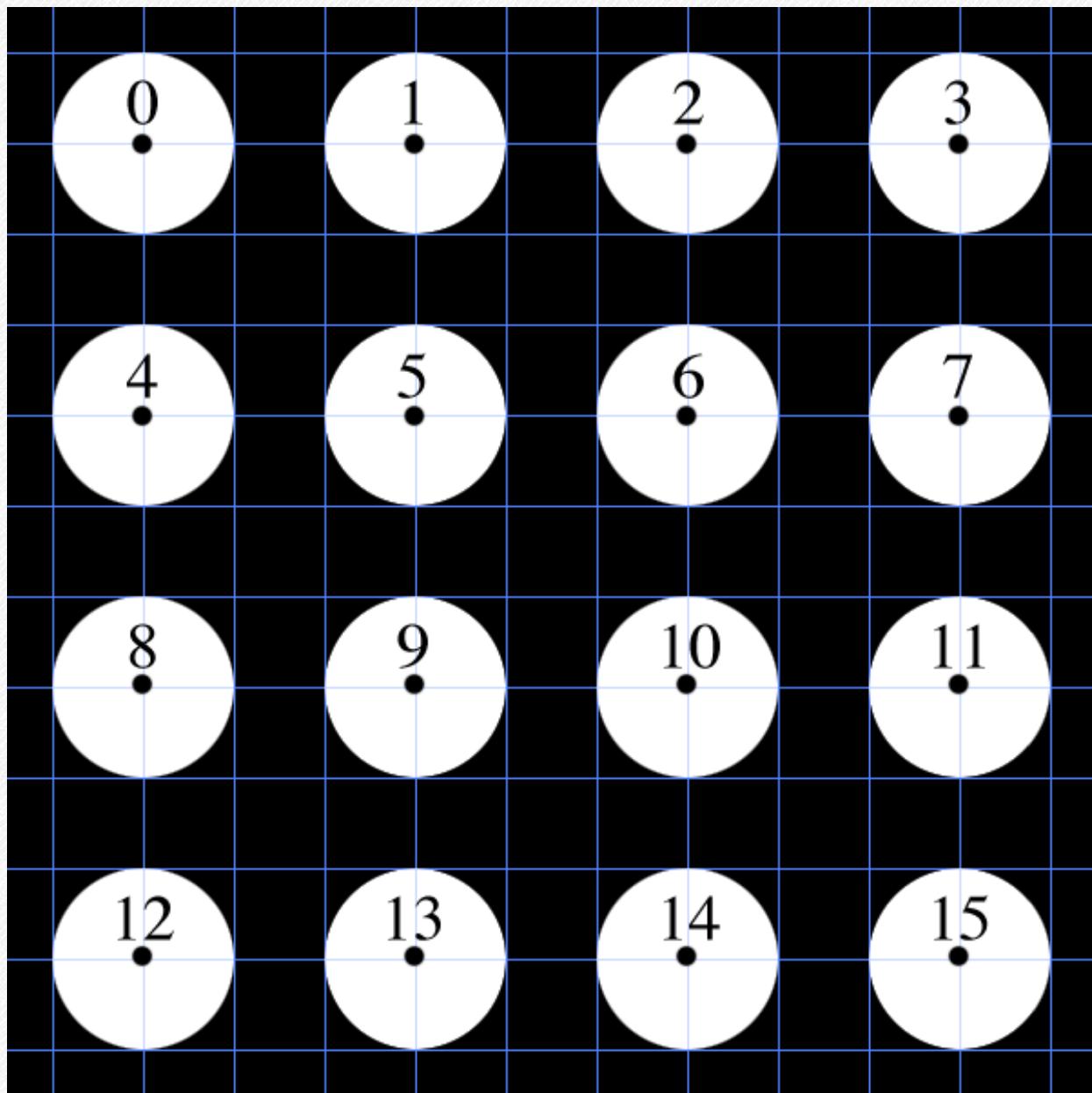
Spacing Unit centers three half-units apart, leaves a margin of half a Unit between each. Two of those halves are inside the units and the third creates empty space between them.

The next page shows why the margin works the way it does.



The `gotoNear()` function in `Unit.js` needs two new arguments which can be used by the Unit to find its place within the group ranks.

The first of those new arguments is an index specifying which position the Unit will take in formation. In a 4x4 group, the positions can be thought of as being indexed in the way shown here:



Position 0 will always be the top-left position in the group. The highest number in the formation will always be the lowest right position. A formation offset expression ensures each index maps to a specific position in the formation.

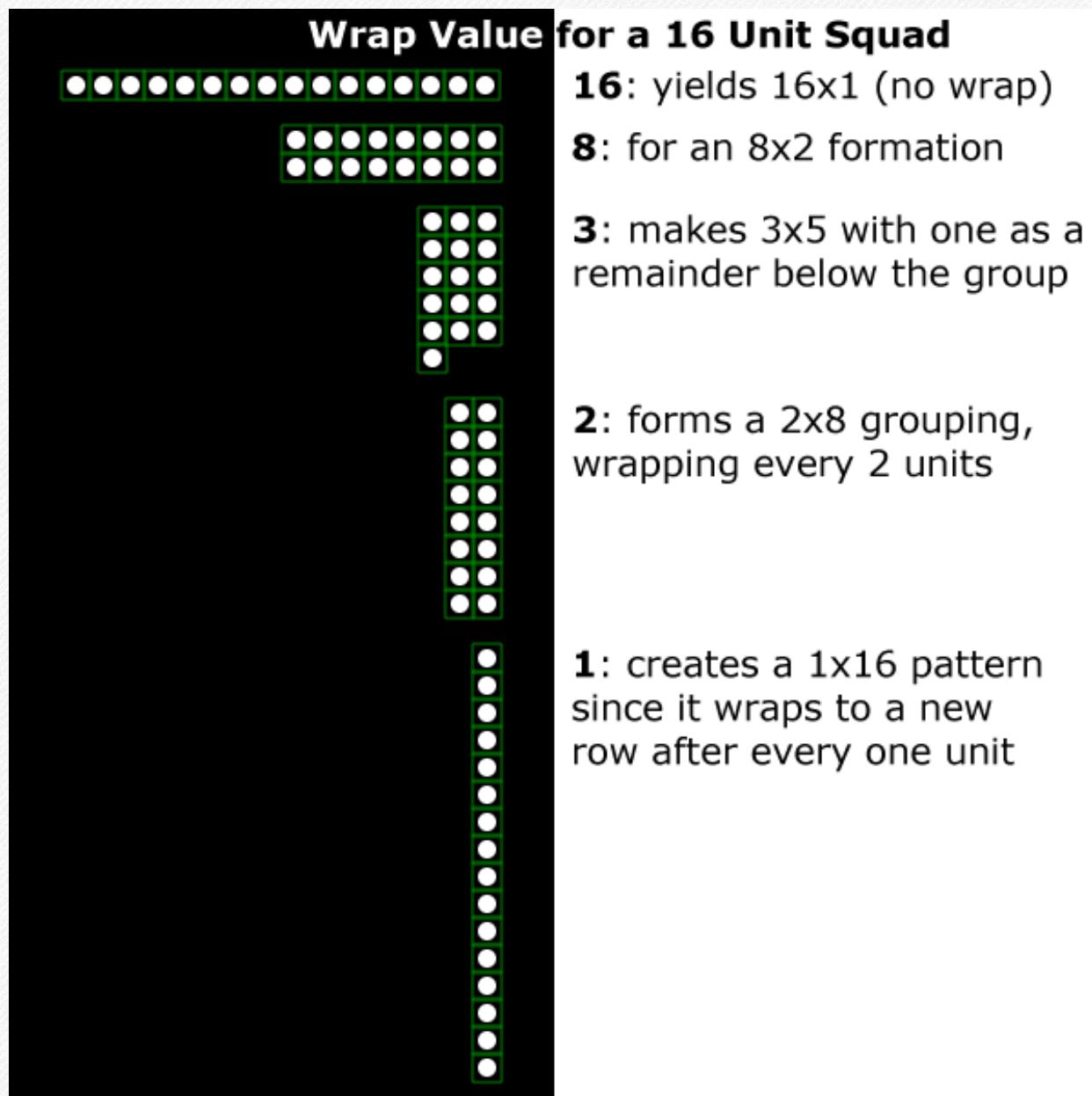
From a strictly numbers standpoint, the approach used to achieve this effect is closely related to how tile position index values in Racing and Warrior Legend were transformed into row and column tile coordinates for the map.



The second new value that each Unit will need to know in order to turn its formation slot number into an x and y pixel offset is the dimension of the group.

Put another way: how many Units do you want in each row before wrapping the line down to start the next? This is needed because the dimensions of the group will vary based on how many Units are being moved.

Here's how various wrap values can affect formation shape:

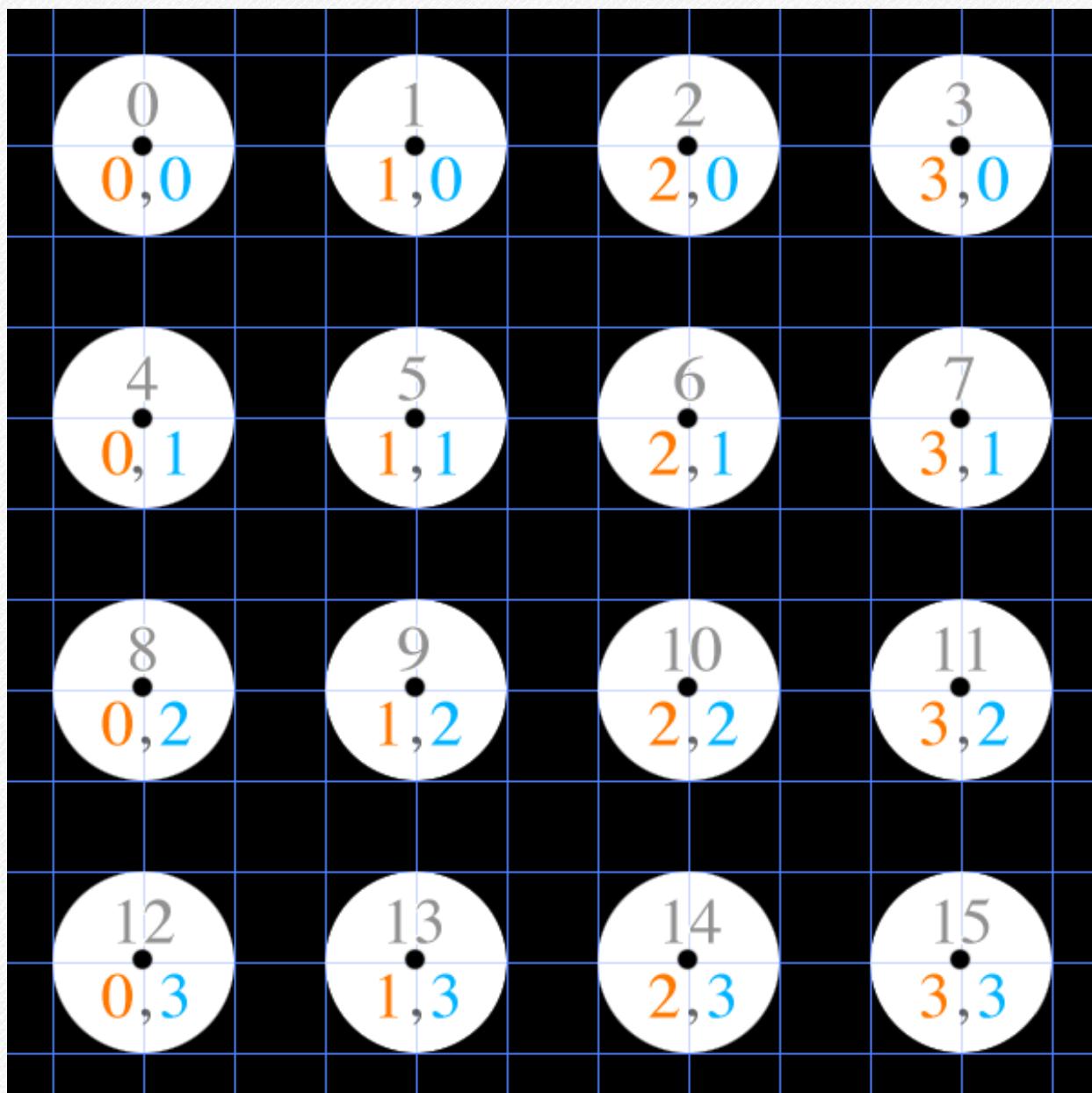




The updated gotoNear() function in Unit.js looks like so:

```
this.gotoNear = function(aroundX, aroundY, formationPos, formationDim) {  
    var colNum = formationPos % formationDim;  
    var rowNum = Math.floor(formationPos / formationDim);  
    this.gotoX = aroundX + colNum*UNIT_RANKS_SPACING;  
    this.gotoY = aroundY + rowNum*UNIT_RANKS_SPACING;  
}
```

As with the tile grids, the modulus gives the column (orange in the diagram below), and division gives the row (blue). For a 4x4 group:





The grayed out top center number for each circle in the example above represents the index for that position within the formation. Since the layout shown has 4 Units per row, its formationDim value is 4.

To briefly reiterate Math.floor()'s effect in all this, it's merely chopping off the decimal part after division so that instead of 11 divided by 4 giving 2.75 it cuts away the 0.75, leaving 2 as a usable whole row number. This way each increase of 4 in the Unit formation index will bump its row number down 1 further.

The multiplications by UNIT_RANKS_SPACING translate the row and column into pixel offsets for the Unit position.

How to ensure that each Unit in the group gets a unique formationPos value? Pass in the for-loop's counter as it counts through the array of selected Units. It's guaranteed to go from 0 up to one less than the number of Units in the group.

The only question remaining is how to decide on a wrap value suitable for a given number of Units. Row wrap numbers that are too small or too large produce awkwardly elongated formations. Compact ranks not only look better, they can also have an impact on gameplay. If the group is more spread out vertically or horizontally than it ought to be, the Units can't defend and cover one another as well from approaching enemies.

Since the goal is to form square or nearly square formations, it makes sense to base the wrap value on the square root of the number of Units in the selected group. That yields an ideal 2x2 square for 4 Units, a 3x3 square for 9 Units, and upward to



5x5 for 25 Units or 6x6 for 36 Units. Here's the new else case in 'mouseup' within Main.js:

```
var mousePos = calculateMousePos(evt);
var unitsAlongSide = Math.floor(Math.sqrt(selectedUnits.length+2));
for(var i=0;i<selectedUnits.length;i++) {
    selectedUnits[i].gotoNear(mousePos.x, mousePos.y, i, unitsAlongSide);
}
document.getElementById("debugText").innerHTML =
    "Moving to ("+mousePos.x+","+mousePos.y+");
```

The `Math.floor()` is once again used to cut a potentially messy number, here the result of a square root, down to a whole number. It's useful to know that 4 Units should be in each row, or 5 units in each row, but far less helpful to know there should be 4.25 or 5.66 units in each row of the formation.

As for the +2 fudge factor in the square root: using only the square root gives a good answer only if the number of Units in a formation happens to be a perfect square. The square root works nicely for 4 Units (2^2), 9 Units (3^2), 16 Units (4^2), 81 Units (9^2), etc. If there are just a few Units less than a perfect square, when `Math.floor()` trims off the decimal the formation winds up with one whole less row than if the group had only one or two more Units. By adding two, a small group which is almost big enough to form a square (7 Units, or 14 Units...) will form an incomplete square rather than a more unbalanced rectangle.

Here's the practical effect: without that +2, a group of 3 Units will form a vertical 1x3 line, rather than filling in $\frac{3}{4}$ of a 2x2 square. For a set of 15 Units, you'd get an elongated 3x5 formation, rather than a nearly filled in 4x4 formation.

The +2 serves as a tipping point so that relatively small numbers of Units can fill in most of the next biggest square.



The adjustment doesn't need to be proportional to the total group size because the difference is most significant only for very small groups of around 20 Units or less. The difference between a 3x5 and a 4x4 pattern for 15 Units is very noticeable since one is obviously oblong. The difference between a 4x5 compared to 6x6 becomes proportionally less important, since either looks reasonably square-ish.

No reason to simply trust me on this though. You've got the code. If you're not sold on why that +2 is there, try selecting and moving small subgroups of Units with it removed from the code, then try it with the number inserted back in. Experiment with other numbers in its place. Maybe you'll find a value that you feel is more natural in terms of the proportions for the groups that it produces.

EXPECTED RESULT

Clicking and dragging to select works the same as before, and clicking still directs selected Units to the spot clicked.

The difference now is that rather than bunching up randomly, the dots should form orderly ranks, more like soldiers. No matter whether you're directing 3 Units, 1 Unit, or all 20 Units, they should form an evenly spaced group that's roughly even in its number of rows and columns.



r.t.s. game step 12

enemy units on map

So far all Units have been on the player's side. For a battle to take place you'll need enemy Units to target.

Both the player and enemy Units could be kept mixed in the same array. Keeping units for each side in a different array simplifies any case which only needs to scan one team's units.

Having team arrays separated will help, for example, when checking player lasso selection. That code only needs to consider player Units. Either team scanning for targets will also benefit from a per-team list. If all Units were kept on a common list then in these situations it would be necessary to scan every Unit, filtering each on-the-fly for which team it's on.

Even with each team on a separate list, it will also be useful for a Unit to be able to instantly check whether it's on one team or the other without needing to scan the lists to see which it's on. To do this, update the `reset()` function in `Unit.js` to store a true or false argument that indicates whether or not the Unit is on the player's side.

While making this distinction in the `reset()` function, use the unit's team to position enemy units in the opposite corner and give members of each team a specific color:



```
this.resetAndSetPlayerTeam = function(playerTeam) {
    this.playerControlled = playerTeam;
    this.x = Math.random()*canvas.width/4;
    this.y = Math.random()*canvas.height/4;

    // flip all non-player units to opposite corner
    if(this.playerControlled == false) {
        this.x = canvas.width - this.x;
        this.y = canvas.height - this.y;
        this.unitColor = 'red';
    } else {
        this.unitColor = 'white';
    }

    this.gotoX = this.x;
    this.gotoY = this.y;
    this.isDead = false;
}
```

The only other update needed in Unit.js is to put that newly set Unit color to use:

```
this.draw = function() {
    if(this.isDead == false) {
        colorCircle( this.x, this.y, UNIT_PLACEHOLDER_RADIUS, this.unitColor );
    }
}
```

The flip side of the convenience to having each team on their own list will be a bit of repetition in places that deal with all Units. If the game had more than three or more teams in-play it might make sense to have an array of all teams to iterate through them in these cases. With only two, it'll work fine to give each team an array and loop through both teams in Main.js's resetAndSetPlayerTeam(), move(), and draw() calls.

Set up the enemy team similarly to the player's team, placed near the corresponding definitions used for the player:

```
const ENEMY_START_UNITS = 15;
var enemyUnits = [];
```

In the window.onload function, the reset() calls on the player units need to be updated to the longer name and passed a



'true' argument to signal that they belong to the player's side.
The enemy team gets a similar treatment right after:

```
for(var i=0;i<PLAYER_START_UNITS;i++) {  
    var spawnUnit = new unitClass();  
    spawnUnit.resetAndSetPlayerTeam(true);  
    playerUnits.push(spawnUnit);  
}  
  
for(var i=0;i<ENEMY_START_UNITS;i++) {  
    var spawnUnit = new unitClass();  
    spawnUnit.resetAndSetPlayerTeam(false);  
    enemyUnits.push(spawnUnit);  
}
```

Make parallel changes to the move() function:

```
function moveEverything() {  
    for(var i=0;i<playerUnits.length;i++) {  
        playerUnits[i].move();  
    }  
    for(var i=0;i<enemyUnits.length;i++) {  
        enemyUnits[i].move();  
    }  
}
```

Likewise with the unit.draw() portion of drawEverything():

```
for(var i=0;i<playerUnits.length;i++) {  
    playerUnits[i].draw();  
}  
  
for(var i=0;i<enemyUnits.length;i++) {  
    enemyUnits[i].draw();  
}
```

What's equally critical is paying attention to which code *isn't* copied for the new team array. The drawSelectionBox() calls, for example, should only be made for the selectedUnits list. The lasso should only scan through the player team's array.

EXPECTED RESULT

White Units in the top-left should look and work the same. Red Units in the bottom right should not respond to player input.

Those Red Units do not yet move, and cannot be attacked.



r.t.s. game step 13

click on enemy for attack command

The red enemy dots made in the previous step aren't very exciting. They lack any interactivity or movement. One thing at a time, as always. Before they move and fight back, first give the player a way to target enemies. This functionality will be easier to test while they're motionless.

The player will input combat commands in the same way as move orders. After selecting the group to command, the next click will be on the enemy to target instead of open ground.

The game will interpret which action to take based on whether the player clicks near enough to an enemy Unit (setting that one as the new target) – otherwise it will fall back to the current goto position functionality.

First, you'll need to detect which action to issue when the mouse click finishes. In Main.js's 'mouseup' handler, check out the else case which directs all selected Units to move. Loop through the enemy units here to check whether any of them are close enough to the mouse location for the order to count as an attack instruction.

Detecting whether a Unit is near the mouse is a pretty generalizable bit of functionality and a conceptually distinct task. Wrapping it in a helper function is a better idea than spilling the for-loop and distance checks right into the middle of the 'mouseup' if-else handling. Writing a `getUnitUnderMouse()` function to contain this reasoning in one



centralized place will simplify the code that uses it similarly much as mouseMovedEnoughToTreatAsDrag() did.

Define this helper below mouseMovedEnoughToTreatAsDrag() just to keep all mouse code in a group. Mouse functions can later be migrated into a separate file, and having them already grouped will make that reorganization faster. The new getUnitUnderMouse() function needs to iterate over all Units, returning a reference to which is beneath the mouse:

```
function getUnitUnderMouse(currentMousePos) {
  var closestDistanceFoundToMouse = MIN_DIST_FOR_MOUSE_CLICK_SELECTABLE;
  var closestUnit = null; // using null instead of undefined, to mean 'none found'

  for(var i=0;i<playerUnits.length;i++) {
    var pDist = playerUnits[i].distFrom(currentMousePos.x, currentMousePos.y);
    if(pDist < closestDistanceFoundToMouse) {
      closestUnit = playerUnits[i];
      closestDistanceFoundToMouse = pDist;
    }
  }
  for(var i=0;i<enemyUnits.length;i++) {
    var eDist = enemyUnits[i].distFrom(currentMousePos.x, currentMousePos.y);
    if(eDist < closestDistanceFoundToMouse) {
      closestUnit = enemyUnits[i];
      closestDistanceFoundToMouse = eDist;
    }
  }

  return closestUnit;
}
```

This is incorporating a couple of outside references that aren't set up yet. There's a new const and a Unit distFrom() function.

NEW CONST AND HELPER FUNCTION

The const MIN_DIST_FOR_MOUSE_CLICK_SELECTABLE's meaning is probably a no-brainer. To be clear though, it's an indication of how close the mouse has to be to a Unit to qualify as on top of it when clicking. I defined this near the top of Main.js (it will move to another file with the rest of the



mouse code soon). I set mine to 12, giving the player margin for error when clicking since the Units are small.

The new helper function is `distFrom()`, add it to the unit's class. It finds and returns how far the Unit is from the x and y values passed in as the arguments:

```
this.distFrom = function(otherX, otherY) {  
    var deltaX = otherX - this.x;  
    var deltaY = otherY - this.y;  
    return Math.sqrt(deltaX*deltaX + deltaY*deltaY);  
}
```

It's tempting to try to use this new helper function to reduce redundancy where there was already a distance check being calculated at the top of `move()` in `Unit.js`, namely, here:

```
this.move = function() {  
    var distToGo = this.distFrom(this.gotoX, this.gotoY); // no!  
    var moveX = UNIT_PIXELS_MOVE_RATE * deltaX/distToGo;
```

However that will not work – hence the red text! The lines after the distance check use the `deltaX` and `deltaY` values, which in that rearrangement are concealed as local variables in the `distFrom()` function. This leaves a tiny bit of code duplication, but as compromises of this kind go, at least the distance formula is relatively common, short, and easy to recognize.

WHAT GETUNITUNDERMOUSE() DOES

Returning to the `getUnitUnderMouse()` function's overall logic, the function uses two local variables. One is a reference to the closest Unit found so far, the other saves the distance to that Unit as the nearest to be outdone in a later comparison.

The `closestUnit` variable begins as null, which means if it's still null when the function returns, no Unit was in range of the mouse.



By initializing the local closestDistanceFoundToMouse variable to MIN_DIST_FOR_MOUSE_CLICK_SELECTABLE, the function will skip over any Unit beyond that range.

The other variable in use is the currentMousePos argument to the function. That will come in from the mouse event handler.

ABOUT USING NULL

Depending on whether you've done some other programming, you may be wondering what *null* is. Or, if you know what null is, then you may wonder why I'm opting to use that rather than letting the variable remain *undefined* by just not initializing it (in JavaScript null and undefined are different).

For the first part: null is used to indicate that a variable which is meant for use as a reference to an object does not have any object to reference yet. When the question captured by a variable is "which one of these?" null can be used to indicate "I don't know," whether that's "I don't know yet" or "I tried to find an appropriate reference, but couldn't."

When a program throws an error relating to a null pointer error or a null reference, that means the code tried to do something with an instance (call a function on it, access or set a variable on it, etc.) but the variable isn't pointing to a valid object. To prevent those kinds of errors, if there's a chance the object won't be set by a certain point in code, you can check whether the reference is null before doing anything more with it.

The second part of that question is a subjective matter of coding style. When a variable in JavaScript hasn't been set yet, it does not default to null, it is undefined. To signal that no



Unit was close enough to the mouse, you could choose to report back the undefined value, checking for that rather than null.

My reason for preferring and recommending null in this case is because I tend to assume undefined indicates an accident, like a misspelled variable. By setting it to null it's a more clear signal of intention.

Note too that in JavaScript (null == undefined) will yield true, whereas (null === undefined) is false. I've avoided using the three equals sign comparison elsewhere in this book since it's a bit of a more advanced topic, but I wanted to at least briefly acknowledge here that if you need to differentiate null from undefined in code you'll need an extra equal sign in order to do so.

CHANGES TO MOUSEUP HANDLER

The ‘mouseup’ event handler function still needs to put all that helper machinery to good use.

This work starts in the else case around line 101, in the code which is only executed if the mouse is not dragged far enough for selection box behavior.

After getting the mousePos from the calculateMousePos() function, on the next line create a local variable named clickedUnit and set it from getUnitUnderMouse(mousePos).

If the clickedUnit variable comes back not null, and the playerControlled variable on clickedUnit is false (not player controlled, i.e. is an enemy), then print out the number of units being sent to attack in the debug text below the canvas.



Otherwise fall back to the previous movement command behavior.

This is a case where the actual code may be easier to follow than the written out description:

```
canvas.addEventListener('mouseup', function(evt) {
    isMouseDragging = false;

    if(mouseMovedEnoughToTreatAsDrag()) {
        selectedUnits = []; // clear the selection array

        for(var i=0;i<playerUnits.length;i++) {
            if( playerUnits[i].isInBox(lassoX1, lassoY1, lassoX2, lassoY2) ) {
                selectedUnits.push(playerUnits[i]);
            }
        }
        document.getElementById("debugText").innerHTML = "Selected " +
                                                       selectedUnits.length + " units";
    } else { // mouse didn't move far, treat as click for move or attack command
        var mousePos = calculateMousePos(evt);
        var clickedUnit = getUnitUnderMouse(mousePos);

        if(clickedUnit != null && clickedUnit.playerControlled == false) { // enemy?
            // then command units to attack it!
            document.getElementById("debugText").innerHTML =
                "Player commands "+selectedUnits.length+" units to attack!";
        } else {
            // didn't click enemy unit, direct any currently selected units to move
            var unitsAlongSide = Math.floor(Math.sqrt(selectedUnits.length+2));
            for(var i=0;i<selectedUnits.length;i++) {
                selectedUnits[i].gotoNear(mousePos.x, mousePos.y, i, unitsAlongSide);
            }
            document.getElementById("debugText").innerHTML =
                "Moving to ("+mousePos.x+","+mousePos.y+")";
        }
    }
});
```

After all that, the units aren't actually being sent over to attack yet. Thanks to that set up work that'll be much easier to do.

EXPECTED RESULT

The game should act as it did in the previous step, except now if you click on a red enemy Unit, the debug text below the game will show text like, “Player commands 7 units to attack!”



r.t.s. game step 14 separate file for input

Main.js is rapidly expanding, and a ton of that code is for mouse handling. Time to move it into its own file so it'll no longer clutters Main.js.

Begin by duplicating Main.js, since it has all the mouse code. Rename the copied file Input.js and open it in your editor of choice. Add Input.js to the list of imported JS files in the main HTML code. Its order in the list won't really matter, but since Main.js uses its functions it's logical to load Input.js earlier.

Keep the lasso, drag true/false, selectedUnits, and MIN_DIST values in Input.js, and remove them from Main.js.

calculateMousePos, mouseMovedEnoughToTreatAsDrag, and getUnitUnderMouse can go in Input.js, removing the Main.js versions. The window.onload initialization, moveEverything(), and drawEverything() should stay in Main.js, and those functions ought to be cleaned out of Input.js.

Even after all this chopping, a huge chunk of Main.js is still mouse code. That's because the mouse event handlers are written in-line in the window.onload initialization code. The way to fix this is to define those event handlers as separate functions, feeding them by name to the addEventListener calls. Currently, the code looks like this...

```
canvas.addEventListener('mousemove', function(evt) {
  var mousePos = calculateMousePos(evt);
  if(isMouseDragging) {
    lassoX2 = mousePos.x;
    lassoY2 = mousePos.y;
  }
});
```



...instead, cut it into two parts. Define the function:

```
function mousemoveHandler(evt) {  
    var mousePos = calculateMousePos(evt);  
    if(isMouseDragging) {  
        lassoX2 = mousePos.x;  
        lassoY2 = mousePos.y;  
    }  
}
```

Then connect to it by name:

```
canvas.addEventListener('mousemove', mousemoveHandler);
```

This does the same thing as when it's defined in-line. The function still needs that single parameter, which has the same role as function(evt) did in the in-line style. The word "Handler" in the function name isn't necessary, but it's an easy way to mark the function's purpose, i.e. to show that it's intended to be connected as an event listener, never called directly.

Likewise, there's no requirement for the first part of the function's name to match the event it's connected to. This naming makes it easy to see whether each function is matched to the correct event. This is also why I'm violating the capitalization convention – instead of mouseMove, starting each word with a capital letter, I've opted to name the function mousemoveHandler for consistency with the event it's meant for. The listener code in window.onload is now:

```
canvas.addEventListener('mousemove', mousemoveHandler);  
canvas.addEventListener('mousedown', mousedownHandler);  
canvas.addEventListener('mouseup', mouseupHandler);
```

This is made possible thanks to transplanting each of those longer function definitions to the bottom of the Input.js file like so:



```
function mousemoveHandler(evt) {
    var mousePos = calculateMousePos(evt);
    if(isMouseDragging) {
        lassoX2 = mousePos.x;
        lassoY2 = mousePos.y;
    }
}

function mousedownHandler(evt) {
    var mousePos = calculateMousePos(evt);
    lassoX1 = mousePos.x;
    lassoY1 = mousePos.y;
    lassoX2 = lassoX1;
    lassoY2 = lassoY1;
    isMouseDragging = true;
}

function mouseupHandler(evt) {
    isMouseDragging = false;

    if(mouseMovedEnoughToTreatAsDrag()) {
        selectedUnits = []; // clear the selection array

        for(var i=0;i<playerUnits.length;i++) {
            if( playerUnits[i].isInBox(lassoX1,lassoY1,lassoX2,lassoY2) ) {
                selectedUnits.push(playerUnits[i]);
            }
        }
        document.getElementById("debugText").innerHTML = "Selected " +
                                                       selectedUnits.length + " units";
    } else { // mouse didn't move far, treat as click for move or attack command
        var mousePos = calculateMousePos(evt);
        var clickedUnit = getUnitUnderMouse(mousePos);

        if(clickedUnit != null && clickedUnit.playerControlled == false) { // enemy?
            // then command units to attack it!
            document.getElementById("debugText").innerHTML =
                "Player commands "+selectedUnits.length+" units to attack!";
        } else {
            // didn't click an enemy unit, so direct selected units to this location
            var unitsAlongSide = Math.floor(Math.sqrt(selectedUnits.length+2));
            for(var i=0;i<selectedUnits.length;i++) {
                selectedUnits[i].gotoNear(mousePos.x, mousePos.y, i, unitsAlongSide);
            }
            document.getElementById("debugText").innerHTML =
                "Moving to ("+mousePos.x+","+mousePos.y+")";
        }
    }
}
```

Why are there no parenthesis at the end of those function names where they're set up as event handlers? There are no



parentheses there because the functions are not supposed to be evaluated at that instant in the code. You're not passing the return value of mousedownHandler to addEventListener, you're passing the actual function. If given parenthesis it would run its code then and there, instead of handing over a handle on the actual function to call later whenever the related event gets detected.

The (evt) argument in the mouse handler functions is short for event, and contains information about the event that called the function. In this case, it allows the mouse's x and y position to be calculated as part of the data sent in with the mouse events. The single event argument is assumed/default for functions that are intended for use as event handlers, even though depending on your desired functionality or which event is connected the evt information is not always needed or used.

EXPECTED RESULT

No change to functionality, but this refactoring has carved down the Main.js file from over 160 lines down to around 70.

Next: support for acting on those attack commands!



r.t.s. game step 15 units attack when ordered

In an earlier step we made changes to determine whether a mouse click happens near an enemy Unit.

In the Unit.js file's reset() function, establish a this.myTarget reference and initialize it to null, to mean no target is picked. Add a function, setTarget(), to the unitClass that accepts a parameter, newTarget, and sets this.myTarget to that value. I'd defined it near this.gotoNear() since it will serve a similar purpose. As usual, the order of the definitions won't have any importance on whether or how the code works.

In Input.js, head to where the Unit attack debug text is set, and add code to iterate through all selectedUnits, calling setTarget() on each. Pass the clickedUnit variable as setTarget()'s argument.

Define another const value, UNIT_ATTACK_RANGE, at the top of Unit.js, set it to 55 for now. This will be the radius within which units will stop to attack their targets.

At the top of this.move() in unitClass, check whether there's a non-null target to attack. If so, check whether the target is within UNIT_ATTACK_RANGE pixels. If it's beyond that range, set the goto destination to the target's position to approach it. Otherwise, if it's in-range, overwrite the move destination with the unit's present position so it will stand still to attack. Then set the isDead flag on the target to true so it will no longer be drawn or updated.

You'll also need a Unit to notice if its target has been defeated.



Here's the new top of Unit.js's `this.move()` function, the rest of which remains unchanged:

```
this.move = function() {
  if(this.myTarget != null) {
    if(this.myTarget.isDead) {
      this.myTarget = null;
      this.gotoX = this.x;
      this.gotoY = this.y;
    } else if(this.distFrom(this.myTarget.x, this.myTarget.y) > UNIT_ATTACK_RANGE) {
      this.gotoX = this.myTarget.x;
      this.gotoY = this.myTarget.y;
    } else {
      this.myTarget.isDead = true;
      this.gotoX = this.x;
      this.gotoY = this.y;
    }
  }
}
```

EXPECTED RESULT

Commanding units to attack a red Unit should now cause them to move within range of the targeted enemy, at which point the target enemy should be turned invisible. Though the Unit is removed from display it is not yet being removed from the team's array in memory.



r.t.s. game step 1b enemies that wander automatically

There's a lot of potential for complexity in artificial intelligence to direct an army in a real-time strategy game. First, to get things going, have the enemy soldiers all migrate mindlessly.

In Unit.js check out the move() function. If the AI has a target assigned it will do the same pursue and attack as player units directed to attack do. The case to be worked out: what to do if the enemy doesn't have a target.

If the Unit has no target and is computer controlled, semi-randomly update the unit's move target some amount west and some amount north:

```
if(this.myTarget != null) {  
    // (same stuff in here as before, just showing which "if" this new else goes to)  
} else if(this.playerControlled == false) {  
    if(Math.random() < 0.02) {  
        this.gotoX = this.x - Math.random()*70;  
        this.gotoY = this.y - Math.random()*70;  
    }  
}
```

It's simple, but it works. Enemies now move!

Each frame the game independently checks whether a random number from 0.0-1.0 is less than 0.02, meaning that about once every 50 frames (at 30 frames per second, every ~1.7 seconds) each independent AI Unit can pick a new walk destination north and west of its current position.

EXPECTED RESULT

Enemies will move upward and leftward until they eventually each leave the left or top of the canvas. They can be defeated by the player, but otherwise won't attack.



r.t.s. game step 17 enemies that attack when close

In the previous step you wrote the computer team's move logic. If there's a target assigned for the Unit, the automatic pursuit and attack behavior can take over in the same way they do if the player clicks on an enemy. All you need to get the enemies fighting is, decide on a reasonable way for the enemy units to choose targets.

When the player's units are all the way across the battlefield from the enemy units, it's sensible to let the navigation logic control the enemy units, rather than combat code. On the other hand if an enemy Unit is near a player target it would seem oblivious to not approach and attack, or maybe as a more advanced behavior retreat if certain conditions are met. Either way, ignoring player units that are approximately in attack range wouldn't look right.

Since the deciding difference in this case is distance to the nearest opposing Unit, add code in the computer unit's move update to check for distance to player units. If the nearest Unit is roughly within attack range (maybe a bit further, so it seems awake and paying attention), set that player Unit as the target. Otherwise, as a fall back, decide on the next semi-random westward and northward migration.

It's important the computer is not too 'perfect' in terms of its reaction and decision time. Humans are limited by motor reflexes and have limited attention to focus usefully on one place and task at a time. By comparison, the computer can



act nearly instantly, with incredible precision, and it can control each Unit independently of every other one in the fight.

For fairness, keep the distance check in the random frequency limiter. That inconsistent gap in computer reaction leaves a small, unpredictable window of time in which an attentive player has an opportunity to act first and win.

Though performance optimization isn't the main rationale for making the distance checks infrequent, as the number of units in play increases on both sides the artificial intelligence could be a real performance bottleneck. Processing load can be kept low by distributing it across many frames instead of all getting crunched and updated every frame.

As for how to find the closest player Unit, the `distFrom()` helper will come in handy. What we need isn't the distance to one Unit though, rather we need to find the closest opposing Unit. To do this, each enemy Unit will need to consider every player Unit.

If the AI needed to work for either team, or both teams, this search would need to be written in a way which passed in the team to search as an argument to the `move()` function. That won't be necessary at the moment since only the computer team needs this functionality.

Using the reference to `playerUnits` as a shortcut, the else-if for computer control within `unitClass` now looks like this:



```
    } else if(this.playerControlled == false) {
        if(Math.random() < 0.02) {
            var nearestOpponentDist = UNIT_AI_ATTACK_INITIATE;
            var nearestOpponentFound = null;
            for(var i=0;i<playerUnits.length;i++) {
                var distTo = playerUnits[i].distFrom(this.x,this.y);
                if(distTo < nearestOpponentDist) {
                    nearestOpponentDist = distTo;
                    nearestOpponentFound = playerUnits[i];
                }
            }
            if(nearestOpponentFound != null) {
                this.myTarget = nearestOpponentFound;
            } else {
                this.gotoX = this.x - Math.random()*70;
                this.gotoY = this.y - Math.random()*70;
            } // end of else, no target found in attack radius
        } // end of randomized ai response lag check
    } // end of playerControlled == false (i.e. code block for computer control)
```

This involves a new tuning const to indicate how far away the player Unit can be for the computer Unit to see and target it:

```
const UNIT_AI_ATTACK_INITIATE = UNIT_ATTACK_RANGE + 10;
```

This new const is based on the UNIT_ATTACK_RANGE so the enemy Unit will seem aware of its surroundings. If it weren't for the randomization comparison check against 0.02 causing the computer's AI to skip updating most frames the sight range would cause the computer to shred player units. With the random lag in there, an attentive player can win, but will tend to lose if not actively fighting back.

The process for finding the nearest enemy Unit is exactly the same as is used in our mouse click code. Here though, it's over a wider range and for a different reference point. What we have at this point works fine in terms of our desired behavior, but it's beginning to involve some duplicated code.

In both situations, the process outputs exactly one value, a Unit if one is found in range, or null otherwise. The cases differ



by a few inputs: the x and y coordinates for the center of the search range, the maximum range to consider, and which list of units to scan through. Here's a fine situation to wrap up that algorithm into a new helper function:

```
function findClosestUnitInRange(fromX,fromY,maxRange,inUnitList) {  
    var nearestUnitDist = maxRange;  
    var nearestUnitFound = null;  
    for(var i=0;i<inUnitList.length;i++) {  
        var distTo = inUnitList[i].distFrom(fromX,fromY);  
        if(distTo < nearestUnitDist) {  
            nearestUnitDist = distTo;  
            nearestUnitFound = inUnitList[i];  
        }  
    }  
    return nearestUnitFound;  
}
```

Where to put that definition? Since it will be needed by both Input.js and Unit.js it's not a great fit for either. If there were more helpers like this one, a separate file could hold them, but with only one so far, that seems a bit premature. It's also not fully abstracted from this specific program as, for example, the graphicsCommon.js functions are. It uses Unit semantics and expects the distFrom() helper function to exist.

For now, I'll make mine the first function in Main.js, which is as good a place as any, for a function used throughout the project. It can easily be moved to another file later if it's getting in the way. No need to decide on everything all at once.

With that new helper function, the else-if for computer Unit logic is now a lot more straightforward and easy to follow:

```
    } else if(this.playerControlled == false) {
        if(Math.random() < 0.02) {
            var nearestOpponentFound =
                findClosestUnitInRange(this.x, this.y, UNIT_AI_ATTACK_INITIATE, playerUnits);

            if(nearestOpponentFound != null) {
                this.myTarget = nearestOpponentFound;
            } else {
                this.gotoX = this.x - Math.random()*70;
                this.gotoY = this.y - Math.random()*70;
            } // end of else, no target found in attack radius
        } // end of randomized ai response lag check
    } // end of playerControlled == false (i.e. code block for computer control)
```

Using this new helper in `getUnitUnderMouse()` in the `Input.js` file exposes a small snag. The `getUnitUnderMouse()` function wants the nearest Unit regardless of team, not from just one or the other.

Each team could be checked separately for a match off either list, and if both lists yielded candidates the nearer of the two could be returned. The programming to handle that would be a mess of its own, but the goal is to make things cleaner.

Here's how that would look (I'm *not* suggesting this, I'll show a tidier way to accomplish the same thing):



```
function getUnitUnderMouse(currentMousePos) {
    var nearestPlayerUnit =
        findClosestUnitInRange(currentMousePos.x, currentMousePos.y,
                               MIN_DIST_FOR_MOUSE_CLICK_SELECTABLE,playerUnits);

    var nearestEnemyUnit =
        findClosestUnitInRange(currentMousePos.x, currentMousePos.y,
                               MIN_DIST_FOR_MOUSE_CLICK_SELECTABLE,enemyUnits);

    if(nearestPlayerUnit == null) { // if no player unit found
        return nearestEnemyUnit; // return nearest enemy (may be null too)

    } else if(nearestEnemyUnit == null) { // if no enemy unit found either
        return null; // then just output null meaning no unit of either team worked

    } else { // if we have a unit from both teams, test which is closer
        var pDist = nearestPlayerUnit.distFrom(currentMousePos.x, currentMousePos.y);
        var eDist = nearestEnemyUnit.distFrom(currentMousePos.x, currentMousePos.y);
        if(pDist < eDist) {
            return nearestPlayerUnit;
        } else {
            return nearestEnemyUnit;
        }
    }
}
```

If the code kept a single list with all Units from both teams, that whole function could be reduced to:

```
function getUnitUnderMouse(currentMousePos) {
    return findClosestUnitInRange(currentMousePos.x, currentMousePos.y,
                                  MIN_DIST_FOR_MOUSE_CLICK_SELECTABLE,allUnits);
}
```

That one's certainly easier to understand.

Earlier on, having separate lists for the player Units and the enemy Units seemed preferable. That streamlined situations where one team needed to scan the other, or for the mouse lasso to search just one team's Units.

There are other places in the code though, where having on a single list would be nice. The move() and draw() functions in Main.js, both do separate but very similar loops over each team's list.



It doesn't have to be one way or the other. If you want a list with all Units in some cases, and a list of just one team or the other for other situations, you can have both.

Remember how the selectionList references a list of Units but doesn't rip them from the playerUnits list? Unit's aren't really "in" one group or the other. The lists work like lists of names – Units can be named on more than one.

Keep and manage a list for allUnits in addition to playerUnits and enemyUnits. Wherever Units are added or removed from any list, a bit of extra effort will be needed to keep them all in sync. That bit of work can be hidden away in helper functions.

Near the top of Main.js, after the playerUnits and enemyUnits list definitions, create an empty array for allUnits. Write a function below it that can add a unit to either team in addition to automatically adding that same unit to the allUnits list:

```
var allUnits = [];

function addNewUnitToTeam(spawnedUnit,fightsForTeam) {
    fightsForTeam.push(spawnedUnit);
    allUnits.push(spawnedUnit);
}
```

By using addNewUnitToTeam() any time you put a new unit on either team, the allUnits list will be populated too. If you need to remove a Unit from either team array, a similar helper that can cut it from both the team-specific and allUnits arrays, will come in handy.

The window.onload set up for the teams now looks like this:



```
for(var i=0;i<PLAYER_START_UNITS;i++) {
  var spawnUnit = new unitClass();
  spawnUnit.resetAndSetPlayerTeam(true);
  addNewUnitToTeam(spawnUnit, playerUnits);
}

for(var i=0;i<ENEMY_START_UNITS;i++) {
  var spawnUnit = new unitClass();
  spawnUnit.resetAndSetPlayerTeam(false);
  addNewUnitToTeam(spawnUnit, enemyUnits);
}
```

While already on a roll with refactoring and cleaning up the team array handling, here's another spot with duplication. Pull that redundant code into a helper function:

```
function populateTeam(whichTeam,howMany,isPlayerControlled) {
  for(var i=0;i<howMany;i++) {
    var spawnUnit = new unitClass();
    spawnUnit.resetAndSetPlayerTeam(isPlayerControlled);
    addNewUnitToTeam(spawnUnit, whichTeam);
  }
}
```

This reduces the code in window.onload for team set up to:

```
populateTeam(playerUnits,PLAYER_START_UNITS,true);
populateTeam(enemyUnits,ENEMY_START_UNITS,false);
```

With the allUnits array populated, the Main.js functions which iterate over both team arrays can now use allUnits instead. This is now the whole definition for moveEverything():

```
function moveEverything() {
  for(var i=0;i<allUnits.length;i++) {
    allUnits[i].move();
  }
}
```

In drawEverything() there were two team loops, but now:

```
for(var i=0;i<allUnits.length;i++) {
  allUnits[i].draw();
}
```

When we first wrote findClosestUnitInRange(), there weren't enough related helper functions to easily categorize it. With



these additional clean up and helper functions, it's easy to see a category it fits into.

Create a new file named UnitTeam.js and add it to the HTML. Where it goes in the .js listing order won't make a difference, but right after (next to) Unit.js seems like a reasonable spot.

Toss into UnitTeam.js – and remove from Main.js – the three helper function that deal with whole teams, the empty team and allUnit array declarations, and the tuning const values that determine each team's starting size. This restores Main.js to a fairly streamlined, high level description of what's going on with the game. Sweet.

EXPECTED RESULT

Enemy Units will drift leftward and upward, as in previous versions. The player will be able to select and command his or her Units to attack. What's new is, enemy Units will now stop when they get within attack range of the player's Units and will then periodically remove Units from the player's group.

There will be a little brokenness in the game's current behavior. In particular, the player will still be able to select and command defeated (invisible!) Units. Any of the enemy's defeated, invisible Units will still attack the player's group. There are glitch ghosts on the battlefield.



r.t.s. game step 18

remove ghosts from both teams

Defeated Units stay selected. You can deselect them again, or even give the invisible Units orders. You may catch your Units disappearing without a visible cause, even though all enemies have been defeated.

The issue here is that defeated Units aren't going away. There's a conditional in the unit's draw() function that prevents defeated Units from showing, but that's it. The code elsewhere doesn't care whether a Unit is alive for lasso selection, mouse commands, or automatic attacking.

This kind of bug is very common among programmers new to dealing with entity lists. It's fortunate that the effects in this case are so noticeable. Until enemy Units had a way to move and attack, it wasn't even visible that this error has been buried in both teams.

Another, more subtle side effect of this issue: it's making the computer-controlled Units far less dangerous than they should be. Enemy Units focus on the nearest target, even after its defeat, rather than picking a different, live Unit to attack. (There's even code checking if their current target is defeated, and if so forgetting that target. Then on the very next frame it will find and pick that exact same target again! Oops.)

Sometimes it makes sense to completely remove an object from memory when it is destroyed in game. Other times it makes sense to conceal and recycle it, through a practice called memory or object pooling.



Creating a new object in memory can be a bit more computationally expensive than shuffling it between lists. In addition to the cost of the memory change, creating and destroying many elements can accumulate as fragmentation in memory. Fragmentation occurs when data that you'd assume are adjacent (like, say, sequential elements in a list) wind up scattered in memory, decreasing the usefulness of the cache. In a cutting edge videogame, scattered access for a huge number of elements could degrade performance, or eventually lead to bigger memory operations failing.

With a few dozen Units and simple graphics, the computer will be fine creating and destroying some Units during battle.

If, however, you wanted massive armies of hundreds of Units on each side, and those armies were frequently firing missiles and grenades both ways, memory pooling would be helpful for those projectiles, and possibly for that many Units, too.

Another thing to weigh when deciding on a solution is whether you want to show defeated Units. If a defeated unit should change its appearance to a pile of robot rubble, you could keep the Unit on the list and display it as destroyed. In that case, the full Unit could be swapped for a simpler placeholder, or it could be juggled to a different list that could be skipped for processes like target search and mouse lasso selection.

Whenever you're working on an issue that involves invisible issues figure out a way to make the problem more visible. Dead Units are not being drawn. That can change by adding an else case in the `unitClass draw()` function:

```
this.draw = function() {
  if(this.isDead == false) {
    colorCircle( this.x, this.y, UNIT_PLACEHOLDER_RADIUS, this.unitColor );
  } else {
    colorCircle( this.x, this.y, UNIT_PLACEHOLDER_RADIUS, 'yellow' );
  }
}
```

Now defeated Units turn yellow instead of becoming invisible.

It's now plain to see that defeated Units on both sides continue to function in terms of their movement and attack capabilities.

Add code to scan through the allUnits array for any that have been recently defeated. Upon finding any, remove them from all relevant lists. In addition to the team lists and the allUnits array, also scan through the selectedUnits array to remove any defeated Units.

Removing an element from the middle of an array adds the complication of altering the numbering of other elements after it on the list. In a 10 element array, removing an element from the list will shift down all the ones after it, cascading to the end with the 10th becoming the 9th. That shuffle can lead to skipping over an element. When the for-loop is done with the 4th element and removes it, next it will check the 5th element, but the one that was previously 5th just slipped past undetected into the newly vacated 4th spot.

A simple trick to solve this is to loop backward over an array if removing any elements from it. Instead of starting with index 0 and counting up to the last element, start at the last element (1 less than the list length, since there's an element 0) and count down with i-- through each entry until you've dealt with each,



including element 0. If you remove an element from the middle of the list the ones higher that get shuffled down were already dealt with earlier in the pass. Here's how that looks in a new helper function for UnitTeam.js, plus a function that calls it on all of the game's Unit lists:

```
function removeDeadUnitsFromList(fromArray) {
  for(var i=fromArray.length-1; i>=0; i--) {
    if(fromArray[i].isDead) {
      fromArray.splice(i,1);
    }
  }
}

function removeDeadUnits() {
  removeDeadUnitsFromList(allUnits);
  removeDeadUnitsFromList(playerUnits);
  removeDeadUnitsFromList(enemyUnits);
  removeDeadUnitsFromList(selectedUnits);
}
```

Why does each array need to be scanned separately? Why not just scan through allUnits, then remove the offending index from the matching team array?

Even though allUnits has the same Units as the team arrays, a given object's address in each array can be different. When the game starts up, playerList[19] may be the same unit designated by allUnits[19], but enemyList[0] will map to the allUnits[20] spot, enemyList[1] to allUnits[21], and so on. If Units are added and removed during play, the relationship between lists will quickly be completely scrambled.

Over in Main.js, at the end of moveEverything(), add a call to the new helper function to scrub the lists of any Units that which been defeated during play:



```
function moveEverything() {
    for(var i=0;i<allUnits.length;i++) {
        allUnits[i].move();
    }

    removeDeadUnits();
}
```

After testing this functionality during play, you can remove the `isDead` check in the `unitClass draw()` function. The recently added debug case can also be removed leaving only one line:

```
this.draw = function() {
    colorCircle( this.x, this.y, UNIT_PLACEHOLDER_RADIUS, this.unitColor );
}
```

EXPECTED RESULT

The game will play mostly the same. Importantly, destroyed Units will now be promptly removed from action.

There should be no more invisible Units with selection boxes. Player Units should no longer be lost after nearby enemy Units have been defeated. The game will also be much harder than it was previously, since the enemy Units will no longer be distracted by constantly re-targeting a player Unit that they've already defeated.



r.t.s. game step 19 only remove units when destroyed

It's not really necessary to call `removeDeadUnits()` every cycle. Most frames it will make no changes, since most frames no units will be newly defeated.

There are currently 35 units maximum (15 enemies, 20 player). It's clear from this that the `removeDeadUnits()` function can only have an effect at most 35 times after the game starts. However even on cycles when it's having no effect, it still loops through and checks up to 90 `isDead` values, since the same units appear on multiple lists. If performing those 90 comparisons (plus a little loop overhead) every frame at 30 frames every second, there's another 2,700+ operations per second serving no purpose. Far more will occur when the game has more units and a larger play area.

As I often warn, stressing about optimization before finding an actual performance issue is a potential distraction. However the optimization in this case doesn't need to be complex at all. You know exactly where in the code a Unit dies. You can have the `removeDeadUnits()` function do its work only when that part of the code is reached. Adding a few extra lines can avoid thousands of unnecessary operations per second.

It's tempting to directly insert the removal function in `Unit.js` where we set `isDead` to true during combat. By doing so, you'd change the lists while looping through one of them which, as explained earlier, can be a problem. Also, there are situations in an RTS game where, due to an explosion or area



of effect spell, isDead my change for many units at the same time. There's no reason for every Unit defeated in that situation to call removeDeadUnits() though. Calling it once at the end can clean all of them up.

It will work best to wait until after all units are updated for the frame. Call removeDeadUnits() at the end of every frame, but have it do its work only if isDead was set to true for at least one unit this frame.

Add a new true/false value (I'll call mine anyNewUnitsToClear). Set it true in the code next to where isDead is changed for any Unit. In removeDeadUnits(), do the removal checks only if anyNewUnitsToClear is true, at which time set it to false so the cleanup functions will be skipped on subsequent frames.

The new true/false value can be setup in UnitTeam.js, as well as a function to toggle it to true. The toggle function reduces the need for Unit.js to know exactly how UnitTeam.js works:

```
var anyNewUnitsToClear = false;
function soonCheckUnitsToClear() {
    anyNewUnitsToClear = true;
}
```

Back in the removal function add an if-conditional to skip all the clean up helper functions unless they're needed. Also within that if-statement change the anyNewUnitsToClear variable back to false once the work has been completed:



```
function removeDeadUnits() {
    if(anyNewUnitsToClear) {
        removeDeadUnitsFromList(allUnits);
        removeDeadUnitsFromList(playerUnits);
        removeDeadUnitsFromList(enemyUnits);
        removeDeadUnitsFromList(selectedUnits);

        anyNewUnitsToClear = false;
    }
}
```

Both the if and the function will need closing braces.

Lastly, over in Unit.js, on the line after setting isDead = true call the function from UnitTeam.js to toggle the cleanup flag:

```
soonCheckUnitsToClear();
```

EXPECTED RESULT

No change. The difference in performance at this scale will not be noticeable. If you'd like to test to verify that this is working as expected, you could add a counter that increments within the removeDeadUnits()'s if-conditional and prints its latest value as the HTML page's debug text or using a temporary call to console.log().



r.t.s. game step 20 declare a winning team

When Brick Breaker needed to check how many bricks were left, there were a couple of ways to perform that check. A scan over all bricks could determine whether any remained. That's a tad inefficient. Alternatively, you could keep a counter of how many were in the game, decreasing it by one during each brick removal event.

Either of those approaches could be applied here. However, because the units are being removed from their team lists, and not just flagged as gone, there's an even better option here: You can see whether a team's list is empty.

Since this matter deals with units at a whole team level it is a good fit for the UnitTeam.js file. Above the definition for AddNewUnitToTeam(), write a new function to check whether one or both teams have zero units left. If so, update the debug text to report the winning side:

```
function checkAndHandleVictory() {
  if(playerUnits.length == 0 && enemyUnits.length == 0) {
    document.getElementById("debugText").innerHTML = "IT'S... A... DRAW?";
  } else if(playerUnits.length == 0) {
    document.getElementById("debugText").innerHTML = "ENEMY TEAM WON";
  } else if(enemyUnits.length == 0) {
    document.getElementById("debugText").innerHTML = "PLAYER TEAM WON";
  }
}
```

Call this function at the end of moveEverything() in Main.js, right after the removeDeadUnits() function has its chance.

EXPECTED RESULT

When all remaining units from either team are removed during play, the text below the canvas should state the winning side.



r.t.s. game step 21

keep units on the battlefield

Computer controlled units wander beyond the playable area. This isn't so much an issue for the player since the mouse can only click on the canvas, but moving a squad near the south or east edge can put a few out of bounds.

Create a new tuning const for how near the edge units can be:

```
const UNIT_PLAYABLE_AREA_MARGIN = 20;
```

Now write a function that constrains the gotoX and gotoY unit variables to within the margins of the space from all edges:

```
this.keepInPlayableArea = function() {
  if(this.gotoX < UNIT_PLAYABLE_AREA_MARGIN) {
    this.gotoX = UNIT_PLAYABLE_AREA_MARGIN;
  } else if(this.gotoX > canvas.width-UNIT_PLAYABLE_AREA_MARGIN) {
    this.gotoX = canvas.width-UNIT_PLAYABLE_AREA_MARGIN;
  }

  if(this.gotoY < UNIT_PLAYABLE_AREA_MARGIN) {
    this.gotoY = UNIT_PLAYABLE_AREA_MARGIN;
  } else if(this.gotoY > canvas.height-UNIT_PLAYABLE_AREA_MARGIN) {
    this.gotoY = canvas.height-UNIT_PLAYABLE_AREA_MARGIN;
  }
}
```

Be careful to not make an error in leaving a gotoX in the bottom cases, or in forgetting to replace the `canvas.width` in those vertical comparison cases with `canvas.height`. Such errors are common in this kind of situation since the similarity tempts copy and pasting then making changes.

Call this function in the middle of the unit's `this.move()`. Place it after `gotoX` and `gotoY` have been set by the target chasing or computer control code, but before those values have a chance to affect a unit's movement direction.



EXPECTED RESULT

Units from both sides should now avoid getting closer than 20 pixels from any canvas edge. If the player moves his or her units out of the way or loses, this can cause enemy units to bunch up into what looks like a single point in the top left. That won't happen when the computer controlled team's strategy is improved. The Units are at least now always visible and in play.



r.t.s. game step 22

removal of update marks from code

As always, this step is the same as the previous example solution, except with all the markers for recently changed lines cleaned out.

EXPECTED RESULT

No change to the functionality.

Congratulations on getting the last of your starter games set up! You've really accomplished quite a lot, implementing many common code concepts and dealing with a bunch of recurring gameplay design challenges.

Before getting into the Section 2 exercises, which will begin by returning all the way back to Tennis Game, a brief intermission is in order to give us a break from these specific games.

INTERMISSION

SOUND EXERCISES USEFUL IN EVERY GAME!

GAME SOUND BONUS ROUND

Something that you may or may not have noticed missing in all game examples so far: **audio**. Sound effects and music are an important part of nearly any game experience. When they're done right, they help a game catch and hold attention, act as additional memory triggers about the experience, can form part of the game's reward system, and add to the tone or atmosphere. Audio is so expected in games that a lack of it can be distracting, making it difficult for any game to feel finished until sounds and music are added.

I've avoided audio for the specific games for a few reasons:

1. A good solution can serve all of the projects equally well.
2. Setting up audio in HTML5 has enough quirks and gotchas, I didn't want to add this complexity to that of the existing game logic.
3. I didn't want you sidetracked mid-programming by hunting for, editing, or recording custom audio.

Over the next series of steps, you'll make a small audio test project, the core functionality of which you'll be able to integrate back into each of the earlier example games.

game audio phase I make a short sound, find a song

In order to test different sound situations, it will be useful to have a handful of audio files available. For creating simple retro sound effects I suggest bfxr, a free online tool based on DrPetter's 2007 program sfxr, with some additional features added by increpare: <http://www.bfxr.net>

As for music, Kevin MacLeod is a prolific composer who has generously made his hundreds and hundreds of instrumental tracks in dozens of styles all available as Creative Commons Attribution. That means there's a specific way the work must be credited – full details at https://wiki.creativecommons.org/Best_practices_for_attribution – but if the crediting is done right, it's 100% free to use: <http://incompetech.com/music/>

Here are other common sources of Creative Commons and/or Public Domain audio (suggested by Harold Bowman-Trayford):

- [http://opengameart.org/art-search-advanced?
field_art_type_tid%5B%5D=12](http://opengameart.org/art-search-advanced?field_art_type_tid%5B%5D=12)
- <https://soundcloud.com/groups/creative-commons>
- <https://www.freesound.org>

If you'd like to record sound effects with a microphone, or edit sounds that you find on the internet (be sure it's something that you're specifically allowed to use!), then Audacity is a great free tool: <http://audacity.sourceforge.net>



Audacity is also an important program to have around since, as you'll see very soon, making sound effects for web games is going to require having all sounds in at least two formats.

Find or make two sound effects (one short, one long) and one music track. Use Audacity to export all files as both mp3 and ogg formats. For the short sound, also make a .wav version.

As examples to work with going forward, I've made and picked out a couple in the gameAudio-effects folder. Other steps will load the files directly from this shared folder, to avoid the need to have multiple copies of them copied for step.

EXPECTED RESULT

No expected result, besides having picked or copied over the sound files that you wish to use. The gameAudio-effects folder for the example solution only has the sound files. There's no code covered yet by this step.

game audio phase 2

the simplest (unreliable) approach

Create a new HTML file. Its contents will just be the most generic, minimum requirements for a JavaScript program to run in browser without throwing errors. The actual JavaScript will only require two lines:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta content="text/html;charset=utf-8" http-equiv="Content-Type">
<meta content="utf-8" http-equiv="encoding">
</head>
<body>
<script>
  var someSound = new Audio("../gameAudio-effects/bloop.wav");
  someSound.play();
</script>
</body>
</html>
```

The filename for the sound needs to match the name of your chosen .wav sound file. The common sound folder is only being used to avoid needlessly duplicating the sound files each step. Sounds do not have to be in a separate directory; they could be sitting in the same folder, so long as any paths in code were adjusted, too, removing: “../gameAudio-effects/”

EXPECTED RESULT

If you drag that HTML file into a browser to test it, you’ll probably hear a beep sound. Refreshing the page should play the beep again. I say *probably* because not all browsers, not even all modern browsers, agree on which sound formats to support. The normal .wav sound format is not valid across all of them. Try testing this tiny page in Firefox, Chrome, and either Safari or Internet Explorer. Some should beep.

game audio phase 3 detecting format compatibility

Fortunately, having your sound files in two different formats can cover all major modern browsers. An additional benefit of the two file types we can generally count on is that they are compressed to be much smaller than the .wav sound format, which makes the files and their download times anywhere from half to a tenth that of the .wav file.

Download time hasn't been an issue yet since the "download" time is non-existent when testing files locally. When sharing the game on the web, the difference between a short and a long load time can be the difference between players trying the game or quitting before it opens. It's worth keeping an eye on file sizes – anything more than even a few MB total is fairly big for a web-based game, even though that wouldn't seem very large for a downloadable game.

This will be our new JavaScript between the <script> tags:

```
var audioFormat;

function setFormat() {
    var audio = new Audio();
    if (audio.canPlayType("audio/mp3")) {
        audioFormat = ".mp3";
    } else {
        audioFormat = ".ogg";
    }
}

setFormat(); // we've still got to call the function for the code in it to run

var someSound = new Audio("../gameAudio-effects/bloop"+audioFormat);
someSound.play();
```



The browser is checking which format it can play, and then using that as the extension part of sound file names to ensure it only uses compatible sounds.

EXPECTED RESULT

Now when testing this file across a variety of modern browsers, you should hear a beep each time the page loads.

This works fine if all you need is to have a sound that plays when the program starts. In a game context you'll want to connect the sound to play on collisions or button presses, which has to happen repeatedly and may take place at short, rapid intervals. Next you'll set up a click event handler to play the sound and test what happens when you attempt to play the sound multiple times in a row.

game audio phase 4

play sound on click event

To get the program to play a sound when you click, it will be handy to have a canvas visible which indicates where to click. Also add a window.onload function to connect the mouse handler as soon as the canvas is ready. Here's the JavaScript:

```
var audioFormat;
var someSound;
var canvas, canvasContext;

function setFormat() {
    var audio = new Audio();
    if (audio.canPlayType("audio/mp3")) {
        audioFormat = ".mp3";
    } else {
        audioFormat = ".ogg";
    }
}

function mouseupHandler() {
    someSound.play();
}

function loadSounds() {
    setFormat();
    someSound = new Audio("../gameAudio-effects/bloop"+audioFormat);
}

window.onload = function() {
    loadSounds();

    canvas = document.getElementById('gameCanvas');
    canvasContext = canvas.getContext('2d');
    canvas = document.getElementById('gameCanvas');
    canvas.addEventListener('mouseup',mouseupHandler);
    canvasContext.fillRect(0, 0, canvas.width, canvas.height);
}
```

This may look like a lot of changes from the previous step, but it's just duplicating a bit of the common structure from the game projects. You'll also need to add the canvas after the <body> tag and before the <script> tag:

```
<canvas id="gameCanvas" width="800" height="600">
```



EXPECTED RESULT

The window will now display an 800x600 pixel black rectangle. Clicking on that rectangle should cause a beep sound to play.

On the surface this seems fine.

Click the mouse very rapidly to hear the issue with how the sound is being played. Two or three quick taps are likely to cause at least one less beep than the number of mouse clicks. The sound is blocked from playing again until its previous time being played has a chance to finish.



game audio phase 5 switch to the longer sound file

Rapid clicks in the previous step raised suspicion that sound played in rapid succession might not be handled well. With the very short beep this was no big deal. For sound effects that include a little trailing off at the end, this issue will be much more noticeable. The quiet trailing of the sound will crowd out the loud first part. By swapping in a longer sound file it'll be easier to test the sound set up.

By changing the asset to one that exaggerates the issue you'll have an easier time confirming how the changes ahead affect the sound playing behavior.

This will require only a single line change:

```
someSound = new Audio("../gameAudio-effects/longer"+audioFormat);
```

That's the longer sound effect that I included as my example if you're using my provided sound files. If you've made or found one of your own, change the filename in code accordingly.

EXPECTED RESULT

Clicking the mouse once on the black canvas area will play the sound. While the sound is playing, subsequent clicks seem to be ignored. After the sound ends, clicking on the canvas will start the sound again. As another way to observe this effect, try clicking as rapidly as possible for several seconds, and instead of hearing the sound trigger often you will instead hear it playing steadily back-to-back each time promptly after the previous one finishes.



game audio phase 6 the hiccup workaround

One clumsy but extremely simple and therefore surprisingly common solution to this problem is to restart the sound as part of trying to trigger it to play. To do this, add a single line before the `someSound.play()` in your mouse handler:

```
someSound.currentTime = 0;
```

If you think of the sound as a music player (which is basically what HTML5's sound support seems intended for), that line of code is moving the tracking position back to the start of the tune. If the sound isn't already playing, then that restart command has no practical effect, after which the `.play()` call will initiate the sound as it has been doing. Otherwise, if the sound is already playing, this will have the effect of abruptly restarting it.

On the plus side, this change does make the sound much more responsive to user input, time of collision, or whatever other event you may use to trigger it from code. For certain types of sound effects, as when much of the sound's time is a quiet trail off of the initial sound, this might work fine. For sounds that have distinct parts or are expected to echo and overlap, the abruptness of this shortcut can be easily heard.

EXPECTED RESULT

Each click should cause the sound effect to start or restart immediately. The sound will still only play once at a time, but instead of the previous play blocking a subsequent attempt the new sound will seem to instantly cancel the previous one.

game audio phase 7

the redundant buffer workaround

The machine that you're using probably has no trouble playing more than one sound at a time. This limitation isn't due to the hardware, it's due to how the sound is handled by the browser.

You could load and play the short sound effect alongside the longer sound effect to prove this point. Instead, why not load the longer sound a second time and have it play over itself?

Make a second sound buffer that also uses the long sound. Name it altSound. Make a variable named altSoundTurn and give it a default value of false:

```
var someSound, altSound;  
var altSoundTurn = false;
```

In loadSounds(), prepare it in the same way as someSound:

```
function loadSounds() {  
    setFormat();  
    someSound = new Audio("../gameAudio-effects/longer"+audioFormat);  
    altSound = new Audio("../gameAudio-effects/longer"+audioFormat);  
}
```

The biggest change will be to the mouse click handler, which will now be written to alternate between which buffer it uses:

```
function mouseupHandler() {  
    if(altSoundTurn) {  
        altSound.currentTime = 0;  
        altSound.play();  
    } else {  
        someSound.currentTime = 0;  
        someSound.play();  
    }  
    altSoundTurn = !altSoundTurn; // toggle between true and false  
}
```



One minor trick shown here not used in earlier examples is the ! – in programming we refer to an exclamation mark as either a “bang” or “not” symbol – which inverts any true/false value.

Now playing the sound will alternate between the two buffers, allowing us to hear the sound overlapping itself. By keeping the sound reset hack, you’ll also ensure that each time the user clicks, an instance of the sound will start anew, while the second most recent instance will continue to play.

EXPECTED RESULT

Clicking repeatedly should now sound convincingly like each click is creating a fresh sound, even though the program is technically only juggling between two instances of the sound. A click should no longer abruptly terminate the most recent sound that started. If you’re unclear how it’s still helping us to set the currentTime values to 0, run the program with those lines commented out. It should be fairly noticeable that once two overlapping sounds have started, additional mouse clicks have no discernible effect until one of the sounds completes playing.



game audio phase 8 wrap the extra buffer in a class

The approach shown so far gets results but would be a pain to deal with as-is for each sound in a game. You shouldn't have to manually set up two instances of each sound and keep an extra true/false flag for every sound to track which buffer will play next. This may be manageable while only using one sound, but for even a few sounds this would quickly get silly.

Crafting a class in a separate file to hold this pattern can save trouble and headaches later.

Before making the split, think about what this needs to work. What will you need to do with every sound?

You'll need to load it and play it. That's all there is to it. Those will be the functions called on the sound class from outside.

There's no reason for code outside the class to know anything about the alternator true/false value or the extra sound buffer. More information is coming up soon on a way that access restriction can be enforced.

Create a new text file, using SoundOverlaps.js for the name. Include it in the HTML file above where the `<script>` block opens, after the `<canvas>` element:

```
<canvas id="gameCanvas" width="800" height="600">

<script src="SoundOverlaps.js"></script>
<script>
```

As for what goes in SoundOverlaps.js:

```

var audioFormat;

function setFormat() {
  var audio = new Audio();
  if (audio.canPlayType("audio/mp3")) {
    audioFormat = ".mp3";
  } else {
    audioFormat = ".ogg";
  }
}

function SoundOverlapsClass() {
  this.load = function(filenameWithPath) {
    setFormat(); // calling this to ensure that audioFormat is set before needed

    this.altSoundTurn = false;
    this.mainSound = new Audio(filenameWithPath+audioFormat);
    this.altSound = new Audio(filenameWithPath+audioFormat);
  }

  this.play = function() {
    if(this.altSoundTurn) {
      this.altSound.currentTime = 0;
      this.altSound.play();
    } else {
      this.mainSound.currentTime = 0;
      this.mainSound.play();
    }
  }

  this.altSoundTurn = !this.altSoundTurn; // toggle between true and false
}
}

```

The `audioFormat` variable and `setFormat()` function have been moved to this file out of the main HTML file, to ensure they're available to be used in another project by dropping this file in. The `setFormat()` function gets called at the start of `load()` for a new sound. Although there are several ways that you could block it from being called on subsequent sound loads it's harmless enough to keep it simple for now.

The `SoundOverlapsClass` is a container for the sound loading, sound playing (alternating the buffer automatically), and related set of variables as required per sound.



Returning to the HTML file, remove the audioFormat variable and setFormat() function since they now live in the separate sound-dedicated external JavaScript file. Also remove the someSound, altSound, and altSoundTurn variables along with any references to them since they're now captured in the new class. Below where the canvas and canvasContext variables are declared add two sound variables:

```
var longerSound = new SoundOverlapsClass();
var shorterSound = new SoundOverlapsClass();
```

Both sounds are included this time to illustrate how little code is needed to support additional sounds, each with its own extra buffer.

The mouseupHandler() function can play the long sound:

```
function mouseupHandler() {
    longerSound.play();
}
```

Write a keyboard handler. Hook it up in the window.onload set up function so that a key press will trigger the other sound:

```
function keyPressed(evt) {
    shorterSound.play();
    evt.preventDefault(); // keeps arrows and spacebar from scrolling page
}
```

Cut out the loadSounds() function, since it is now taken care of by the SoundOverlapClass load() function. At the top of window.onload, instead of calling loadSounds() specify the filename with path for each of the two sounds:

```
longerSound.load("../gameAudio-effects/longer");
shorterSound.load("../gameAudio-effects/bloop");
```



Recall that you don't want to specify the file type extension in the load() function. That will be determined and filled in based on which type of audio format the browser supports.

Lastly, add an event handler connecting the 'keydown' event to the keyPressed() function defined above. Where you put it in window.onload doesn't matter, but as a logical grouping I'd suggest adding it on a line near mouseupHandler(), to keep the input handlers together:

```
canvas.addEventListener('mouseup',mouseupHandler);
document.addEventListener("keydown",keyPressed);
canvasContext.fillRect(0, 0, canvas.width, canvas.height);
```

Seeing the mouse and keyboard handlers side-by-side like highlights a distinction that I've previously glossed over. The mouse handler is connected to the canvas... but the keyboard handler is connected to the document. Why? What gives?

The mouse listener only applies when the mouse clicks or moves within the game's canvas. For the keyboard, however, it cannot be connected to the canvas. The canvas can't hold keyboard focus. The document level for events refers to the whole webpage, which is what normally catches keystrokes to scroll with the arrows and spacebar. (This is also why there's a line added in keyPressed() handlers to block those key effects from happening.)

EXPECTED RESULT

Click for long sounds, press keys for short sounds.

game audio phase 9 encapsulate the sound class

As mentioned in the introduction, in this book I've specifically avoided going too deep into object-oriented programming. It's not that I'm in any way opposed to it, goodness no, but I find that the topic is most beneficial for programmers that have first run into the obstacles of trying to write large programs or reusable parts of programs without it.

Furthermore, while first working out an idea – whether as a beginner creating practice games, or a more experienced programmer trying out a quick prototype – there can be advantages in speed and flexibility to keeping things exposed at first. Control can be tightened up based on the emerging usage patterns as the game begins to solidify.

All that said, this sound playing class is a great candidate for demonstrating one of the central concepts underlying object-oriented programming: encapsulation. Encapsulation refers to hiding the implementation details so other code can utilize a class without knowing how (and not breaking how) it works.

No code outside of the sound class should have access to changing or playing the two sound buffer files or the true/false value used to toggle between them. Other code in the game shouldn't even know or care that there are two sound buffers.

In object-oriented programming “private” variables are the ones inaccessible to outside code. Private is also the keyword used in many programming languages to mark this distinction. In JavaScript, rather than using a private keyword, the



difference is marked in a more subtle way: class variables are declared using “var ” instead of “this.”

The “var ” form can be used in a class definition to keep the variable’s access hidden beyond the class. This book has treated classes as though they are somehow different from functions, in JavaScript they’re actually functions being used differently. The “var ” keyword can be used to make the variable local to the class in much the same way as it would make a variable local within any function.

Speaking of classes being functions, you can pass arguments to a class’s definition just as you do for other functions. In JavaScript this is done in combination with *initialization code which is not inside any function definition*, creating what in most other programming languages would be a “constructor” – meaning set up code that’s run when the object gets made:

```
function SoundOverlapsClass(filenameWithPath) { // accept argument for constructor

    setFormat(); // calling this to ensure that audioFormat is set before needed

    // variables as "private", hidden to outside. Using "var " instead of "this."
    var mainSound = new Audio(filenameWithPath+audioFormat);
    var altSound = new Audio(filenameWithPath+audioFormat);

    var altSoundTurn = false;

    this.play = function() { // not "var ", keeping "this.", as we need it exposed!
        if(altSoundTurn) { // note: no "this." since it's "var" - local/private
            altSound.currentTime = 0;
            altSound.play();
        } else {
            mainSound.currentTime = 0;
            mainSound.play();
        }
        altSoundTurn = !altSoundTurn; // toggle between true and false
    }

}
```



As the comment after `this.play()` suggests, functions can also be made private. Since `play()` needs to be called from outside the class the “`this.`” part of its definition is what’s needed.

If you’re coming from a programming language that supports traditional classes and constructor functions, seeing the call to `setFormat()` floating alone in the class definition like that may look jarring, offensive, and wrong. You’re actually totally free in JavaScript to intermix any and all sorts of code in-between the definitions of functions within the class. I’ve made a point of avoiding that throughout as a way to stick to a pattern more familiar and universal across languages, but used properly this is one of JavaScript’s features.

Back in the HTML file you can then drop the `.load()` calls for the sounds from the top of `window.onload`, instead passing in the filename string directly where the sounds are declared:

```
var longerSound = new SoundOverlapsClass("../gameAudio-effects/longer");
var shorterSound = new SoundOverlapsClass("../gameAudio-effects/bloop");
```

EXPECTED RESULT

Same functionality as before. Click to play long sounds, press keys to play short sounds. The `SoundOverlaps.js` file is nearly in a condition to be dropped directly into another project and used merely by knowing how to interface with it: loading sounds by relative path and filename root, calling `.play()` on the sounds, and having `.ogg` and `.mp3` versions of each file. Any game you drop it into won’t need to care about what else is going on inside the class to make that possible.

If this is the first time you’ve seen these features in JavaScript, you may be feeling tempted to jump back into the previous



game projects to do some refactoring, burying as private as many of the currently publicly exposed “this.” variables and functions as possible. If you’d like to do so for practice, then feel free! Remember though, in addition to changing each variable where it is declared, you’ll also need to change each place it’s referenced to drop its “this.” prefix.

game audio phase 10 looping music track

Fortunately, music won't be nearly as complicated to deal with as the sound effects files. Unlike sound effects, unless you're doing very advanced dynamic game scoring, you don't really expect the music to ever overlap itself. You also don't need more than one music track playing at a time.

On the other hand, whereas sounds tend to end rather quickly, music tends to be longer and looping. Because of that difference you'll want to build in a way to stop or switch the song as it plays. Sound effects don't need that functionality.

First, rename the `SoundOverlaps.js` file to `SoundAndMusic.js`, since we will handle sounds and music in the same file to simplify their inclusion in other projects. This also ensures the `setFormat()` function is available for both. No matter whether you're dealing with sound or music, in both cases you're up against the same problem of there not being a single audio format currently supported by all major browsers.

After renaming the JS file, remember to change its import statement near the top of the HTML file too:

```
<script src="SoundOverlaps.js"></script>
```

If the game only needs a single song to play on loop the entire time, it could simply load and play the audio with the `loop` attribute set true. It would then repeat indefinitely. However it will be handy to save a reference to our music file after it's started. That way you can stop, resume, or replace it with another music track during play.



Here's the code to add in the SoundAndMusic.js file:

```
function BackgroundMusicClass() {  
  
    var musicSound = null;  
  
    this.loopSong = function(filenameWithPath) {  
        setFormat(); // calling this to ensure that audioFormat is set before needed  
  
        if(musicSound != null) {  
            musicSound.pause();  
            musicSound = null;  
        }  
        musicSound = new Audio(filenameWithPath+audioFormat);  
        musicSound.loop = true;  
        musicSound.play();  
    }  
  
    this.startOrStopMusic = function() {  
        if(musicSound.paused) {  
            musicSound.play();  
        } else {  
            musicSound.pause();  
        }  
    }  
}
```

Whereas SoundOverlapsClass() requires a different instance per sound, *for BackgroundMusicClass()* you'll only need a single music player for the whole game. In addition to looping sound, the class can end one song to swap in another in. It can also pause or resume music. Near where the canvas variables are declared, add an instance of the music player:

```
var backgroundMusic = new BackgroundMusicClass();
```

Then change the mouse and keyboard handlers to test it:

```
function mouseupHandler() {  
    backgroundMusic.startOrStopMusic();  
}  
  
function keyPressed(evt) {  
    backgroundMusic.loopSong("../gameAudio-effects/Walk_Right_In_1929");  
    evt.preventDefault(); // without this, arrow keys scroll the browser!  
}
```



Lastly, at the end of window.onload, trigger the music player to loop the longer sound effect:

```
backgroundMusic.loopSong("../gameAudio-effects/longer");
```

By using the long sound effect initially, rather than the longer music track, it will be quicker to determine whether the sound is looping as expected. Otherwise, you'd be waiting minutes to make sure that it restarts each time it finishes, which should also be checked after the music is paused then resumed.

EXPECTED RESULT

Upon opening the page the long sound effect will begin looping. Clicking the mouse on the canvas should cause that loop to pause and resume. Pressing any key will switch from the long sound effect to looping the full-length music file. Once the song has been switched, clicking to pause and resume will keep the song the same, it shouldn't change back to the looped sound effect. Note that pressing a key is not set up here to switch the loop back to the long sound effect, although you can modify it to do so if you'd like to test that.



game audio phase II getting the file ready for game use

Remember how for images the games blocked starting until all images finished loading? The ImageLoading.js load() functions connect a function to each image's onload event to count off how many files have been loaded. It's possible to do a similar trick with audio, except instead of using an onload function for audio variables, you would add an event listener to count off assets when the 'canplaythrough' event happens.

The 'canplaythrough' event doesn't work exactly the same as an onload function. Rather than meaning that the audio is fully loaded, it actually indicates that, given the rate at which it has loaded so far, it can now be played through without pausing. Part of the audio can still be loading as the first part plays.

Even though you *can* do that, for simple games like the type covered here it's not really necessary. It may not even be all that desirable. With images, it was very important to have them all loaded before the game's logic started – some logic may be based on image dimensions, and the player needs to see what they are doing. The sounds and music, however, will be harmlessly silent, or just a bit delayed, if they get played before they finish loading.

It's probably better to get the player into the action sooner (even if all audio isn't ready yet) than to block them from starting while it completes downloading the audio. Even short, low fidelity music often outweighs the total file sizes of all



relatively simple sprite images for a game, so I'd suggest not bothering to hold up the player's experience for it.

If, on the other hand, you're building a rhythm game that needs to keep action in sync with music, then it would be important to preload all the sounds and music before starting. In that case the Web Audio API, which provides finer control over sounds, may be a better fit. (As of this writing, only about 2/3 of web users have a browser that supports it. The simpler approach shown here is more widely supported at this time.)

For our purposes – playing simple sound effects and looping background music – SoundAndMusic.js file is fine to use as-is.

For this step's example solution, I'll just clean out the //// marks that were added to reflect previous changes.

EXPECTED RESULT

There won't be any change in functionality from the previous step. The sound class is now ready to drop into any previous game project to easily add sound and music support.

Remember when doing so to add it in the listing of JS files within the game's main HTML.

Speaking of HTML, the actual project file made for this section is not important. That HTML won't be necessary for integrating audio into the other game projects, except perhaps as an example of how to use the audio functionality. The HTML file can otherwise be scrapped entirely. Its only purpose was to demonstrate the SoundAndMusic.js file while working on it. It's that JavaScript file which will be useful to include a copy of when adding sounds to other projects.

SECTION 2

COMPLETION EXERCISES

THE “LAST 10%” IS HALF OF DEVELOPMENT

GAME 1: TENNIS MANIA

COMPLETION EXERCISES

What looks like the last 10 percent of a game's development frequently takes up the last half of the total development time.

In the first section covering Tennis Game's foundations you got the core gameplay working. It's important to not confuse the game being functionally playable with the game being fully finished and presentable. What you have at this point are pieces that can interact with the player and with one another. There's more to a game than its mechanics.

Aside from adding images and sounds, most of the following steps will still be a challenge in programming and gameplay design. The difference now is, rather than working to get a new feature to exist at all, the effort will be spent on making incremental refinements and improvements to what's there.

To return for a moment to the recipe analogy: The cake has been baked, and it might even taste right, but it doesn't have any icing, decoration, or candles. An undecorated cake isn't a cake at all. It's just bread. If you give plain bread to someone who is asking – and expecting – to get cake, they're going to be pretty upset with you.

It's time to add the frosting and candles.

tennis game exercise 1 triple the height of both paddles

Task: Find where the paddle's vertical dimension is located in code. Multiply it times three and try playing that way. Before moving forward to the next step, undo this change.

Discussion: This is about getting practice with the tuning aspect of game development. That means that rather than trying to establish completely new features or functionality, you're instead revisiting earlier assumptions about size, speed, quantities, and so on.

This step is also a reminder that not all tuning values work equally well. Too far in one direction or the other can totally break your core gameplay, even if the rest of the logic for object relationships is otherwise completely the same.

Consider, too, that for a matter like paddle size we have the benefit of having seen before some images of finished games that had answers to the question. We come into this project with a rough idea of what “reasonable” or normal ought to look like. The very first designers of these kinds of games had no such answer key in mind. When working your own original gameplay concepts in the future it’s important to keep a skeptical view on whether your game’s issues might not be in the logic, but instead be hidden in not having good tuning values to make the best of how your game works.

The difference between playable and unplayable, boring and exciting, or fair and unfair can often be a single tuning value that’s too large or too small.

tennis game exercise 2 tune target score and balance it

Task: Change the number of points needed to win to 11. Tweak other tuning values so that a full play session takes around 90 seconds, and doesn't always come out with the same winner. You, as the player, should be winning about half of the rounds, and the computer player should be winning the other half. Accomplish this only by changing existing tuning values, not by changing or adding new program features or functionality. In particular, revisit the values for paddle heights (smaller), horizontal ball speed (higher), and the right paddle's automated movement speed (higher).

Discussion: In the previous exercise, it may have seemed as though there was a roughly 'right' size for the player paddle. However, it's important to realize that for game design what makes a value right can change based on how other values are tuned. If we only change one at a time, at best we'll find the optimal value in relation to the rest of the constants.

Another approach, and the one tried here, is to choose a value or two to set first and then pivot on, adjusting any the other values in turn to make the best of that first setting.

This is *not* the same as saying that all possible tuning values are equally right as long as they're in balance. That we can make the game play as well as possible for a target goal of 2 or for a target goal of 200 doesn't mean that the two ways of tuning the game are equally appealing experiences.

You don't, for example, have numerical control over how long the game will take to play out. That's an indirect result of the values that you can tune. You likely have a sense that playing the game for 2 seconds total might not feel substantial, whereas 20 minutes per round would feel very drawn out. It's hard say specifically what feels right aside from experimenting a bit and testing out various possible values.

Tuning for game design isn't about finding the 'right' value for each number though. It requires making choices about what you want to pivot on, and why.

Consider, for example, that you can decide about how long you want matches to take, say 1-2 minutes. Then you can selectively pick and pivot on tuning variables. If you've picked a target score you want ("10 is a nice, round number"), you can adjust the paddle length and ball speed to change the pacing until matches last that long. Alternatively, if you decide on a paddle height that looks right, you could adjust the score goal to make matches generally your desired time frame.

Much as you have a sense for how long the game should take to play, you probably also have a feel for how often the player ought to miss or return the ball. If making the game last 2 minutes with a target score of 200 requires that the player virtually never hits the ball moving at light speed, that's not fair.

Game design requires experimenting with and balancing out these numbers until none feel too obviously out of whack. There are many possible combinations that would do that, though. Which tuning values will you choose to pivot on?

tennis game exercise 3 invisible player

Task: Find the section of code that draws the player's paddle. Comment out that line, to try playing with an invisible paddle. This change is only temporary. We'll need to make the code active again by removing the comment mark to go forward. Even with the paddle not displayed the game can be played!

Discussion: This exercise is a reminder that the connection between functionality and appearances is something that we have to actively go out of our way to create. The image of the paddle and its behavior in connection to the ball or mouse input are wholly separate parts of code. In a sense, this is likely obvious at a technical level. However, it feels unnatural because it's far removed from how we experience objects in real space. We're adapted to trusting our eyes. Consequently, many of the most difficult bugs to identify and resolve have to do with when gameplay functionality and the visual representations for it aren't properly in sync as expected.

We can also approach debugging from the other direction. Here we made visual representation of functionality invisible, but for many bugs we can take what was invisible or abstract and find ways to create a visual representation for it.

The disconnect between visuals and functionality isn't all or nothing. Sometimes what appeared to be happening and what technically has been happening can be subtly different in ways that affect gameplay situations. Keep an eye out for these, especially in relation to how collisions are handled.

tennis game exercise 4 easier to return serve

Task: Set a minimum and maximum vertical speed for the ball when it is reset each round. A minimum vertical speed ensures the ball won't be too easy to control when returning it. A maximum vertical speed ensures the ball isn't too likely to score before either player has a chance to touch it.

Discussion: Right now the ball sometimes goes straight toward the corners to begin the round, which feels like too much of the challenge and outcome of the match is coming from that auto ball launch rather than interaction between the players. Randomize the ball's vertical speed between a minimum and maximum range (5 to 9 works pretty well) each time the ball resets. 50% of the time flip whether the ball is going on an upward or downward diagonal to reduce the predictability a bit more.

tennis game exercise 5 add sound effects

Task: Copy in the SoundAndMusic.js file from the section on audio, as well as the bloop sound files. Include the added JS file in the HTML as well. Use <http://www.bfxr.net> to generate a low pitched longer tone, and use Audacity to convert the exported wav file to mp3 and ogg formats. Play the bloop sounds for ball collisions (with walls or paddles) and play the lower tone when either player misses the ball.

Discussion: Sound effects are critically important for games. Even a classic game in this style benefits from having them. It's better to have low-fidelity audio, mere beeps and bleeps, than to have no sound effects whatsoever. They can give the game a sense of satisfying rhythm, and also play a role in the sense of reward or penalty during play.

With the details of playing sounds figured out and buried away in a separate class, it's relatively easy to add sound effects. With the benefit of free modern tools we're able to create, edit, and convert sound effects without much hassle. Then, it's simply a matter of inserting a few lines of code indicating when we want those sound effects triggered.

While working on a game, it's easy to get used to not having sound effects in it. Another player will definitely notice! It's not an option to consider your game finished without sounds. If there aren't sounds in your game, there just aren't sounds yet.

tennis game exercise b ball speed increase during play

Task: Have the ball's horizontal speed increase the longer it remains in play since the last goal scored. There are many ways to do this, the simplest of which could increase speed by some percentage after each hit, restoring it back to the minimum speed upon each ball reset. The classical way of dealing with this is to keep a counter that increments each time the ball hits either paddle, resetting that counter to zero when either player scores, then increasing the ball speed on the 4th and 12th hits since the last point was scored.

Discussion: Difficulty came up earlier in addressing tuning. A subtle source of difficulty in tuning the speeds of the ball and paddles is that what may be fun for some players could feel overwhelming, or boring, to others. Finding a speed slow enough to be reasonable to all players might make the game so easy that a skilled player might never give up a point.

The difference could be comfort with the computer mouse for fast reflex games, practice playing games of this kind, or how seriously someone takes the play experience. Whatever the reasons, as game designers we need to acknowledge and address variance in player ability. Whereas a longer and larger game experience might give the player options of difficulty modes, for a game this simple that's unnecessary.

Instead, we can let the ball start each round slow for the first few hits, let it speed up for the next half dozen or so hits, then reach a very challenging pace if it's still in play after that.

This pattern has the effect that even very new players can likely keep the ball in play for at least a few hits, during which they can familiarize themselves with how the game works. Players who are already comfortable with the game then have quite a few chances to score at a ‘normal’ medium speed. If the round is dragging on too long, meaning neither side has scored for the past dozen hits, then the ball can speed up drastically to raise the chances of either side making an error.

Changing the ball speed in three discrete increments during three specific hit count ranges, rather than increasing it little by little each time it’s returned, has the desirable effect of making the speed change more noticeable. If the change is gradual, it may accomplish the same purpose of bringing an otherwise overly long run to a more swift conclusion, but it would lack the dramatic escalation from noticing the change taking place in a single bigger step.

(Note about the example solution I included for this step: I replaced the score values with the ball’s horizontal speed and the hit counter, to make it very easy to verify that those numbers are changing in the manner expected. That isolated change will be undone going forward, but I left it in there as a demonstration for how I tested that the feature is working.)

tennis game exercise 7

refactor into separate js files

Task: The logic required to alter the ball's speed as described in the previous step increased the program's length. To keep the project manageable as it continues to grow, chop up the Tennis Game project into separate JavaScript files. Place the files in a js folder and pull them in from lines in the HTML.

While you're doing this bit of reorganization, go ahead and create ballMove() and ballDraw() functions to contain freely floating code for those tasks. I recommend splitting the program into: ball.js, graphicsCommon.js, input.js, main.js, and paddle.js. No need to rush into creating classes at this point. Merely grouping the code into files will already be a major improvement on how easy the project will be to continue developing. Move the sound effect files in an audio folder, too.

Discussion: When this step came up for the more complex games in Section 1, I provided you with more direction on how to make these changes. This time the division is largely up to you! If you'd like to see how I chose to do it, the example solution is available for this exercise as well.

By dividing the program into separate, well-labeled files, you won't need to scroll past intermixed ball code to make a change to something about the paddle, or the input, and so on. Keeping code readable and navigable isn't a binary good or bad, but exists on a continuum. This improvement can also be seen as a step toward ordering the program code into classes further down the road.

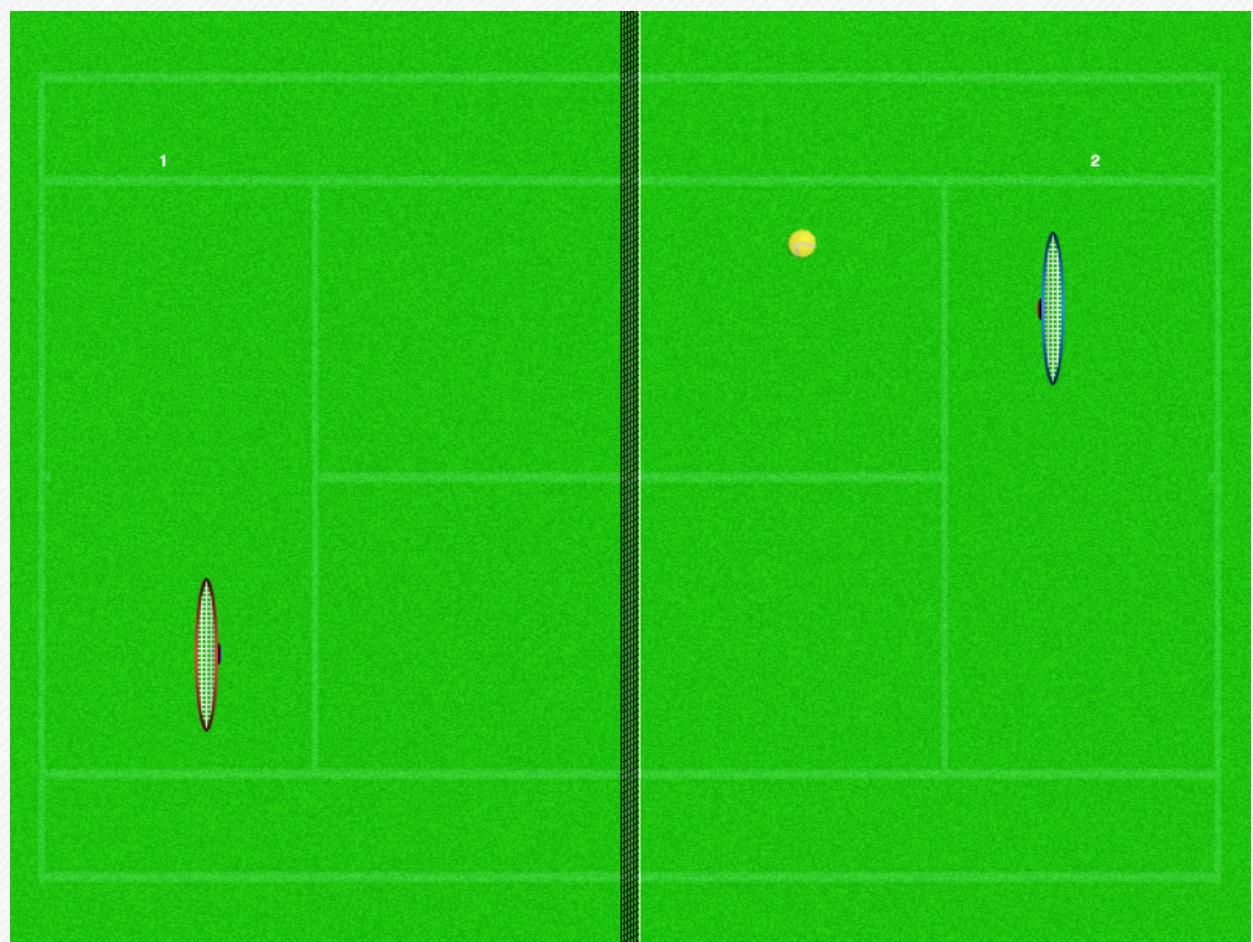
tennis game exercise 8 add image graphics

Task: Create graphics for the paddles, ball, and background, then connect them so they're displayed instead of the black and white shapes. I suggest borrowing and adapting ImageLoading.js from a previous project – perhaps a later step in Space Battle since that project also didn't need to load a tilemap array in the way that Racing and Warrior Legend do. Modify the window.onload function to initiate image loading, moving much of the other initialization code into another function called after images are loaded.

Create one or two graphics functions in GraphicsCommon.js to display the bitmaps. This game doesn't need to rotate images, so the more complex version of the bitmap display function which accounts for rotation isn't necessary.

Since the images give the paddles and ball a more physical representation, this is an appropriate time to provide a different sound when the ball hits a paddle as opposed to when it hits the top or bottom walls. I'm sticking to a tennis theme for my example solution, but you're free to take it another direction if you'd like. If the net or something similar is part of your background image, drawNet() can be removed too.

Discussion: Hopefully by this point it's becoming fairly straightforward to get images into your games. That's partly from having seen and done it before, partly from having past code to borrow from and adapt, and partly from practice.



tennis game exercise 9

MOVE PADDLES AWAY FROM SIDES

Task: Since the paddles are currently thin and right next to the canvas edge, collision with the ball only needs to be checked as it hits the score line. This makes for simpler programming. The downside is that this means there's no time after players miss in which they can still see the ball moving behind them.

That doesn't sound like it would affect gameplay, since at that point it's too late to stop it. However that can be a potentially meaningful moment. Watching the ball draining behind the paddle gives players a chance to register and make sense of what just happened. It adds a moment of minor celebration or regret, respectively, during which the consequence of missing is still visually apparent even though it's too late to prevent it.

For this exercise, modify the code so the paddles are noticeably separated from the edge, say about 15% of the playfield width from each edge, leaving only the middle 70% of the field playable. To simplify finding an ideal position, you should connect the paddle's distance from the edge to a new tuning constant then make adjustments to that new value.

Discussion: Drawing the paddles further from each edge isn't the difficult part. For that, 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 collision code so the ball can move while behind the paddles, yet still respond properly if it overlaps the paddles.

Currently, the paddle collision check happens with the ball's edge check. To support paddles further from the sides, these two collision checks will need to be decoupled, as was done for moving the Brick Breaker paddle off the bottom of the canvas.

Beware: depending on your paddle's thickness and the ball's maximum speed, it's possible for the ball to pass entirely through the paddle in a single logic update. If the ball's center is moving 20 pixels per frame and the paddle is only 10 pixels thick the ball could jump from one side of the paddle to the other, never overlapping the paddle to cause a collision response. There are fancy ways that this issue can be detected and handled, but the easiest solution – which works fine – is to simply ensure that the paddle's collision thickness is greater than the maximum speed of the ball.

The other complication at this point is that because the ball's horizontal speed is flipped horizontally upon detecting paddle collision, the ball can briefly get stuck zigzagging within the paddle. To see what I mean, slide down on the ball while it's moving past the paddle at its slowest speed. To avoid this issue, verify the ball's horizontal direction as part of the collision detection. In other words, only bounce off the left paddle if the ball is currently moving left, and vice versa. This approach also avoids playing hit sounds repeatedly.

tennis game exercise 10

two player mode with keyboard

Task: Classic tennis-style games emphasized head-to-head play between humans. With only one mouse for analog input, keyboard control makes more sense for a two player game. Trying to control the paddle with the keyboard will inevitably be slower and less precise than the mouse. When in two player mode, giving both players keyboard key assignments will be more fair than a mouse versus keyboard match. The left side can move with the W and S keys to go up and down, while the right side can use the up and down arrows.

Add boundary checks to keep the players from moving the paddles off the top or bottom edges of the canvas. The mouse didn't need this because the mouse coordinate can't be given outside the canvas area.

Discussion: Borrowing code from Warrior Legend's starter steps may be useful for catching and using key hold states. Because the paddles aren't in classes and there are so few keys to track, global variables for the hold states will be fine.

Since we'll also want to be able to revert to mouse-vs-computer mode, create a global true/false value to indicate the game is in two player mode. If it's true, allow the keyboard hold states to control both paddles. If it's false, let the computer control the paddle on the right.

(Historical note: the original tennis-inspired games in arcades and home consoles were not played with a mouse or button inputs, but instead each player has a rotating dial to spin.)

tennis game exercise II

title menu

Task: When the program starts, don't show the paddles or the ball, only show the background image with text on top that explains how the game is played. Indicate on that layout that the player can press the 1 key to play alone using the mouse, or the 2 key to play multiplayer with W/S versus arrow keys.

Have the same layout and menu appear when the round pauses due to one player or the other reaching the target number of points. Also include a short message making it clear which player won the most recent round.

Discussion: There are two reasons why a title menu improves the player experience: they'll be able to select how many players to support, and they can get ready before play starts.

About half of what we're aiming to set up here is basically already connected to how the game behaves when the round is over: paddles and ball aren't drawn and extra text goes on the canvas. Set that state to be true when the program begins. Instead of clicking to bypass it, wait for the player to press the 1 or 2 keys to then set two player mode true/false in addition to clearing scores and taking the game out of its menu state.

Since the frozen state will now be a crude menu, rename that variable to reflect "menu" in place of "win." It's in several different files, be sure to replace all uses. The number keys atop the keyboard each have different keycodes than the numbers on the keypad. Be sure to check for both keycodes when responding to the player pressing either 1 or 2.

tennis game exercise 12 programmatic giant score numbers

Note: You're on your own now. No more example solutions to refer to. But I'm confident that by this point you've got this!

Good luck!

Task: The scores are barely readable. The numbers ought to be made bigger so both players can easily see the score during the match. Instead of using a big font, create a set of functions that together can draw out the score for each player as big blocky numbers using filled rectangle calls.

Discussion: While there are a lot of good fonts available, and HTML5 can easily load and display fonts, one of the ways that a program can set itself apart in polish is by programming custom numbers and text. This also makes it possible, should you be interested in doing so, to implement effects like outlines, per-letter movement, animated changes between values, and so on.

I recommend creating a separate file for this since the draw code per letter will quickly pile up. Then make a function that switches on a whole digit by drawing the required filled rectangles based on the value passed to it. You'll need another function that can accept the whole value to be printed which carves it into a sequence of single digits to pass to the first function, accumulating horizontal offsets between each digit for spacing.

tennis game exercise 13 ball trail effect using array

Task: 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. Store some previous number of prior ball positions (the total number should be an easily changed tuning value) and use those to draw the ball not at a single point but as a trail of positions. For starters, track the past five locations.

Draw something at each of those coordinates to visualize the trail data. You might use, for example, circles of decreasing brightness, or images of the ball that get increasingly transparent further from the actual position. For a challenge (these are the challenge exercises, after all), figure out how to draw a polygonal trail that connects the point chain together with thickness that decreases toward the last position tracked.

Discussion: Using an array of coordinate positions, in which each entry in the array has both x and y values, will be the easiest way to make this code flexible with respect to tuning how many past positions to store for use.

One way or another the game will need to deal smoothly with the initial condition, before the ball has any motion history. A common way to do this, which works for many types of trail representations, is to set all trail points directly under the ball's start position whenever you reset the ball.

Each time the ball's position gets updated, remove the oldest coordinate in the array, then add as the top position (for the leading spot) the ball's current location. To draw the trail points, prior to displaying the ball's current location loop through the trail positions from oldest to newest, to ensure the older trail will not overlap the actual ball or newer trail parts. Draw any representation you'd like at each location.

The algorithm described above involves touching each trail point every frame to cascade the coordinates, even though really only two points change: the top point is added and the bottom point is removed. All others in the middle of the list are consistent one frame to the next.

There are a number of clever optimizations that can be applied to make minimal calculations or changes per frame. For example: By tracking a head position and cycling that number you could keep other list elements in place. Or, you could choose a data structure for which adding to one side and removing from the other is especially efficient.

With so few points total, and doing this operation for a single object, these optimizations will have no major impact on game performance. If, however, you were implementing trails for a fireworks-like or fluid particle system, in which trails could be much longer and involve hundreds or even many thousands of objects simultaneously, such improvements would have to be considered. Except purely for the sake of practice, here it probably makes the most sense to instead optimize for the code's readability and your time as the developer, rather than trying to squeeze out a handful of calculations each update.

tennis game exercise 14 computer player sees bounces

Task: The computer-controlled player is very basic. If the ball is lower than the paddle position, the paddle moves down, and if the ball is higher, the paddle moves up. If the ball is bounced off the wall on its way to the other side, the computer player will waste time going the wrong direction relative to where the ball is headed.



Good artificial intelligence for games – world-class chess competitions aside – should make errors that look like errors a human would make, such as going too far and missing the ball, or barely misjudging the destination. However wasting time chasing the wrong direction at a constant speed is a very robotic, artificial-looking mistake. The only reason the player can score points is that the computer player's speed is limited to make it artificially slow. The player's paddle can instantly move from any position to any other.

A more sophisticated and human-like AI should anticipate where the ball will cross its side, accounting for a wall bounce when that will happen first. Human players try to do this, but miss for different reasons, like misgauging how far to move.

Discussion: Calculating the point of reflection and its later intercept with the paddle's line of movement will involve a little algebra or geometry. It's perfectly alright to do some thinking on scratch paper and google searching if it helps. (When finding non-programming math on the web, positive y will be

drawn as up instead of down, but the math won't be any different since the change is purely in how it's represented.)

You can break the math code into steps. No need to arrange it all into one single tangle.

Remember slope, the m value in slope-intercept equations? It's rise over run. The line we want to know the slope of is that described by run of ballSpeedX and rise of ballSpeedY. Combined with our knowledge of the ball's current position, the slope value can be useful for determining future intercepts.

First, whenever the ball is moving toward the right side of the canvas, determine whether the ball is on course to cross the right paddle's line of motion between the top and bottom of the playable area. If it is not, the ball will bounce on its way over, making further calculations necessary. Otherwise, the paddle can simply move toward the first calculated intercept.

Compute where the ball will cross the paddle's line, not where it will cross the canvas edge. Recall that centering the paddle perfectly where the ball will hit causes a straight, easily returned shot back, which is a bad move in this style of game. After identifying the ball's exact point of intercept, scoot aside a little to avoid sending the ball back in a horizontal line.

When the ball is moving away from the AI player moving roughly back toward the center will help it appear more human. The AI player can't calculate its next intercept position until after the ball is struck by the player's paddle, since doing so changes the ball's angle and therefore its destination.

tennis game exercise 15 compute precise paddle hit point

Task: Tunneling happens when the ball moves in a single frame from one side of the paddle to the other. At low speeds this problem doesn't come up, but as the ball speeds up it can become an issue if you're using thin paddles and a small ball.

The solution covered earlier, making the paddle collision area thicker than the ball's fastest speed along that axis, works well but can break down when the ball is moving quickly at a steep angle near the top or bottom edge. The current workaround can also result in unpredictable return angles, due to a difference between where the ball's path crosses the paddle's line and where the ball happens to enter the paddle's collision rectangle on a specific frame.

Discussion: Review the path-line intercept used for the computer-controlled paddle in the previous step. This solution is similar. Divide the velocity components to determine the slope of the ball's motion. Use that slope to find the ball's height where it crosses the paddle's edge between the current and next frame. Determine hit or miss by the position between frames, so that it's independent of which positions the ball gets redrawn. Also set ball return velocity based on that spot.

This may seem like a silly inconvenience for such a simple game, but whether the ball hits or misses a paddle is the central action of this game. It determines Wins and losses. If it's done in a sloppy manner that fails to handle steep shots properly, players may notice and feel the game is cheating!

tennis game exercise 1b unblockable corners (optional)

Task: Prevent the paddles from reaching the very top or very bottom of the ball's moveable area. If the ball goes to those corners the paddles will be unable to block it from scoring. Make sure that these holes are small enough that getting the ball through them is very difficult to do consistently.

Discussion: Although this detail is rarely remembered or retained in more modern variations, in one of the most popular original games in this genre the paddles were unable to reach the very top and very bottom of the canvas. This made it so those corners were impossible to block or return the ball from. This feature originated as a bug during development but was kept rather than fixed since it meant gameplay would never go on indefinitely.

In the games old enough to still have that exposed gap, ball control was much less granular than the method used here. Rather than calculating the ball's distance from the center of the paddle and using that in a continuous way to determine a vertical ball speed, the original segmented the paddle into about 7 separate chunks, each of which gives the ball a different vertical speed. This relates to the corner gaps because it means that in the original many defensive positions could not shoot toward the gap no matter what part of the paddle touched the ball. In the version you have here, though, technically any hit could aim the ball to the corner area.

Fortunately, the same fidelity that makes that kind of precise aim possible also makes it very unlikely. Even a pixel or two up or down from the right defensive position can cause a different enough trajectory that the ball will miss the gap.

If you choose not to include this feature, surely no one would blame you. Even though it was an important piece of the classic game's mechanics, it's the sort of design decision that a more modern audience likely doesn't have a taste for. If the specific goal of your game is to partially recreate the classic gameplay as a historical simulation experience there may be some audience for that as such, but without some explanation the corner gutters now seem unfair to many players.

What I do recommend, though, is at least trying it, and testing the game with people. See if people notice. Maybe players will get fixated on trying to get the ball into that gap, and maybe that plays into the experience in a way that adds more than it takes away. It's easy to imagine how we think players will or won't respond to things, to talk ourselves into this or that, but testing very often proves our reasoning faulty or incomplete.

This feature doesn't take very long to implement, nor to tear back out if you find it's something you feel makes the game worse. Excluding a classic, simple feature like this ought to be a conscious decision, not done on a whim. Creative work sometimes involves throwing away what felt at the time like forward progress, but that kind of experimentation is a normal part of the process.

GAME 2: BRICK BREAKANOID COMPLETION EXERCISES

Keep in mind when working on completing Brick Breaker: new games in this genre are still developed and played to this day.

It's rare to see Tennis Game variants anymore except the ones made for practice by people starting out. On the other hand, if you wanted to play a game like Brick Breaker, you could easily turn up dozens of quite playable, sophisticated, maybe even innovative modern takes on that classic gameplay pattern.

While the games in this style are not necessarily in the top sales charts, going to get licensed for their own Saturday morning cartoon, or winning many awards, they are still being played by millions of players.

You're crossing the threshold from making games purely for your own practice into the realm of working on games that, sufficiently polished, could be released for strangers to play.

Awesome.

brick breaker exercise | organize code into separate files

Task: Currently all the code in this project exists in the same HTML source file. Divide it into multiple JS files, each containing only code and variables for the paddle, ball, or bricks, respectively. For functions or variables that relate to multiple objects (for example: collision handling) keep it in the main file, or with either of the objects involved. Refactor as needed – so since moveEverything() is mainly ball movement, move the code to a ballMove() helper to label that functionality.

Discussion: You’re hopefully getting comfortable with making this kind of split by now, having done it a few times in project foundations and now having done it on your own for Tennis Game. Perhaps this is so much so that now diving back into a game where the code hasn’t yet been split up into meaningful separate files feels *wrong* somehow? That’s the spirit!

You’re welcome to divide up the project files in whatever way will help you best. As before, having separate files for abstract groupings like input or graphicsCommon may be handy, too.

brick breaker exercise 2

keep and display score

Task: 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 start. Reset the score back to zero whenever the ball resets.

Discussion: One natural idea is that score might be based on bricksLeft. The problem with that is then the score would be reset when the bricks reset after being cleared. Instead, create another variable to keep score. Increase that value by 100 each time a brick is removed. Reset the score in ballReset(), and display it in a corner of the playfield.

Although this feature doesn't take much to get it working, it's a case where a little code can significantly affect 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, the more important it becomes to keep the ball going. This may make the player less willing to take risks when using the paddle's edges to achieve angled shots.

brick breaker exercise 3 life limit and full reset

Task: 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 also reset all the bricks then, too. This signals a full game reset. Create a variable for lives remaining, and display it during play. Decrease it whenever the player misses the ball. When lives drop below zero, reset the score and bricks.

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

brick breaker exercise 4

click to serve the ball

Task: With lives in the game, it's now a problem that the first shot of each round is usually missed. We used to be able to ignore that, and start playing when ready after a miss or two. Now that means starting the game with fewer lives than we're supposed to have. Instead begin each round with the ball right above the paddle and a little to one side, so that it shoots upward at an angle upon the player's next click.

Discussion: Create another true/false variable. When it's true that can signify that the ball is held by the player and ready for launch. When it's false the ball will behave as it normally does.

Add a 'mouseup' event handler function, as seen in the RTS Game project. Inside the handler flip your ball hold variable to false.

Where ball movement happens in code, first check whether the ball held variable is true. If it is, set the ball's position to just above and a little to the right of the paddle's center to give the ball an angle upon launch. If the ball is held skip past or bail out (return) before other ball motion or logic gets handled.

Lastly, set the hold boolean value to true any time the ball gets lost. Doing this will set up the player to serve in the new round.

brick breaker exercise 5 fix ball stuck along boundary

Task: With the new serve feature outlined in the previous exercise, it's possible to release the ball off the edge of the canvas 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 can get stuck along the edge, zigzagging tightly up the side then all the way back down.

Modify the code in a way that the anomaly doesn't happen.

Discussion: Unlike many other exercises that are about adding new features, this is about addressing a bug in the code. We didn't have to explicitly program the unintended behavior, instead it emerged at the intersection of two or more features that each work fine for most typical situations.

The issue is in the ball movement code, in how the ball bounces off the left and right canvas edges. There are a pair of if-condition checks for whether the ball is left or right of the canvas boundaries, and if so the ball's horizontal speed gets flipped through multiplication by -1. If the ball is more than one frame of motion beyond the edge, then the next logic cycle the ball will still be beyond the same boundary, making it reverse direction. This makes it wiggle to and fro.

There are several different ways to deal with this. Here are three approaches that can work:

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 if 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 canvas but on its way back then the game won't flip its motion back outward.

Option B) Incorporate the Math.abs() function, for absolute value, around ballSpeedX when setting speed after collision, with a negative in front of it 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 in-bounds position along the edge, and reverse the ball's lateral speed. This way no matter how far off the canvas the ball is, it won't matter since it will promptly be brought back into the play area. The next frame its movement will be away from the wall which it's already no longer overlapping.

Why choose one approach over the others? One thing to keep in mind is that we're going to be adding sound effects to this game, and when we do so we'll want a spot in code that only triggers once for an out of bounds ball. Option B fails that test as it could trigger twice in a stutter of the ball gets placed two horizontal movement spans past the wall due to releasing from the paddle moved all the way to an edge. Option C might be used to get more consistent shot control when serving by bumping a ball to the paddle's center upon release. My preference then is Option A.

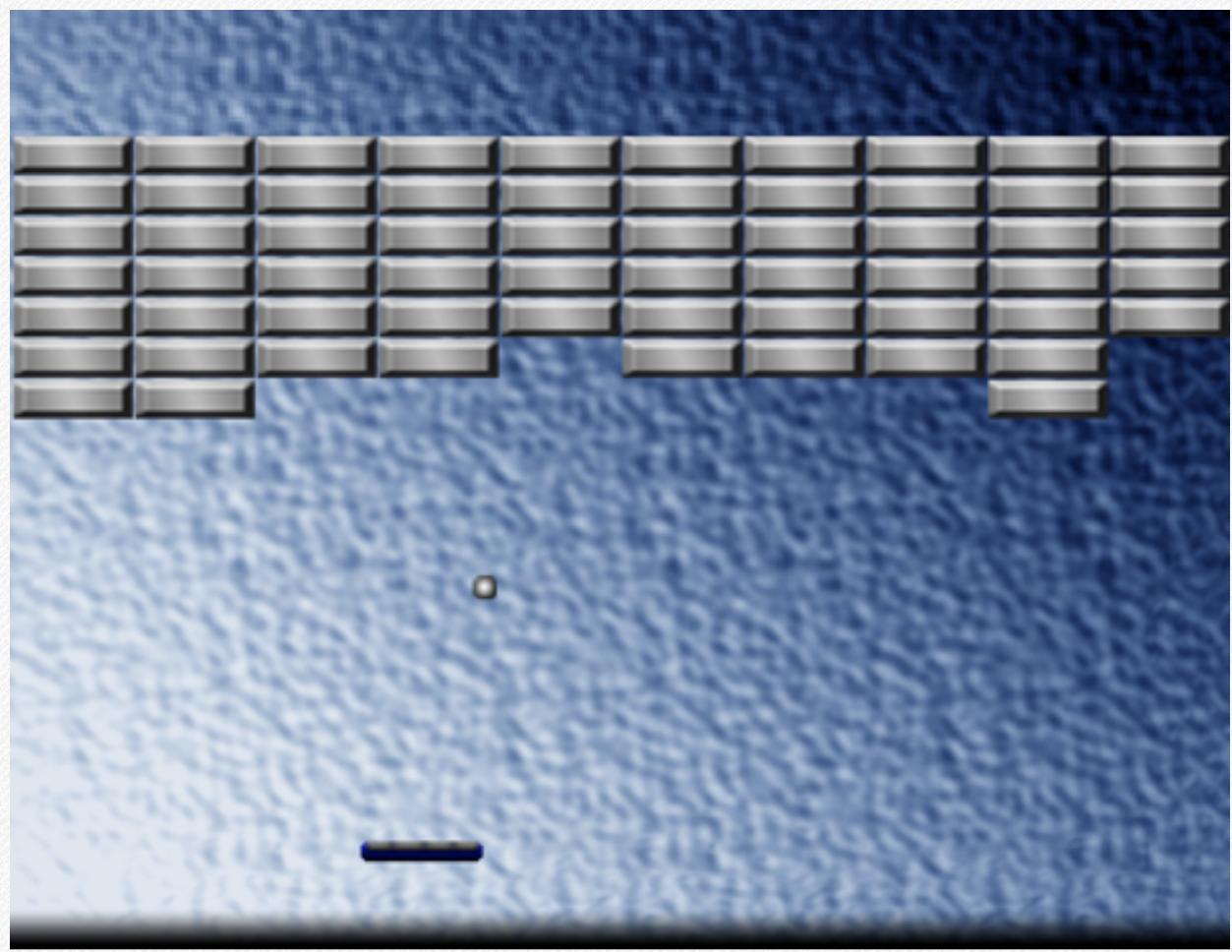
brick breaker exercise b replace coded art with images

Task: All graphics are currently made in code. Instead create some image files and use those in place of the programmed graphics. Borrow and adapt ImageLoading.js from the Tennis Game's exercises or from Space Battle's foundational steps.

Discussion: One way to get dimensions, proportions, and the overall look right is to first take a screenshot 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 over the programmed shapes. To take a screenshot in Windows, try the Snipping Tool. On Mac, Shift+Command+4 followed by Spacebar can be used to save the next window you click on as an image on your desktop.

Next slice those layers apart into the respective images to be loaded and displayed. Only a single file and image variable is needed for all bricks, since 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 of the sort.

As always, remember that rectangle draw functions tend to specify top-left coordinates by its (x,y) arguments, whereas the ball's circle uses the (x,y) as its center point. The difference in these offsets needs to be considered when placing images. When in doubt, it helps to leave the originally programmed graphics in the code to be partly drawn over by the image files. This way you can visually check alignment, easily identifying if the new graphic is centered on the old one.



brick breaker exercise 7

display lives as row of icons

Task: Currently how many lives the player has left is indicated by a number written out in text. Many games instead show a series of icons to show the number of tries remaining in a more visually appealing manner. Figure out an icon you'd like to use (perhaps a heart, ball, star, or 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.

Discussion: Only one image variable should be needed, along with a for-loop to draw each of the icons. To achieve equal spacing of the icons either increment a local horizontal position variable by a fixed amount per each icon, or multiply some spacing constant times the loop's iterator variable, similar to how the bricks are positioned to be drawn.

brick breaker exercise 8

title screen

Task: Give the game a title screen. Create an image in your image-editing program, or construct one in code. Display that layout until a mouse click starts gameplay. Include on your title menu a clear message indicating that the player must click to begin. Return to this message when the player runs out of lives. Display on the title the player's score from the finished round, since the player might be too focused on trying to stop the ball to see their score before the end.

Discussion: Add another true/false variable, set to true by default, to indicate whether the title is being displayed. In `moveEverything()` and `drawEverything()` if that variable is true then handle or display title information. You can use a `return` call to bail out of either function immediately, causing it to skip any of the normal gameplay update or draw code after it.

Set the title menu's signal variable to false in the 'mouseup' handler that you added for ball release. Set that variable to true in the reset code that gets called when your player loses their last ball.

Here's a question: Should the last score variable shown on the title layout be connected to a different stored value than the variable which holds the score during play? One answer could be to not reset the game score until the player starts a new round. By leaving it untouched after the previous match ends you could avoid having a second value.



I'll suggest, however, that it's slightly more straightforward and conceptually clearer to instead create a separate variable specifically for storing and displaying the most recent round's score on the title screen. This reduces the chance of a bug later occurring which could break the last score displayed due to resetting score when a match ends. That kind of change to one aspect of the game should not cause a disruption to a seemingly unrelated feature on the title menu.

At the end of each round set the separate variable to whatever the score was when the round ended. Now it won't matter whether the gameplay score value gets reset at the end of play or the start of play. Reducing the ways your game could wind up broken during later changes is often a wise use of time.

When the title menu is drawn, if the last round score value is some number other than 0 – using that as the stored score's default – display that variable with a few words that make clear what the number is.

brick breaker exercise 9 ball speed increase

Task: The game's intensity is “flat” and unchanging. Increase the ball speed during play to engage the player. By elevating challenge you prevent the experience from feeling too routine.

Ball speed could be increased based on the number of times the ball has hits the paddle since the round started, based on the number of bricks cleared, or time. You could also make it so that some form of user input affects ball speed, for example to enable more skilled players to finish the round in less time.

One of the original ways to increase the ball speed is to make the ball’s speed based on the highest brick row hit since the ball reset. By giving higher bricks a greater point value than lower bricks, this poses a strategic tradeoff for the player. Getting the ball bouncing behind high value bricks means tons of points, but trying to do so makes it more difficult to keep the ball in play. Careful players can choose to instead juggle the ball against the lowest bricks, advancing row by row.

Figure out on what basis you'd like to increase ball speed during play, and make it happen. Feel free to experiment!

Discussion: No matter what basis you’re using to increase ball speed, creating a new variable to track it separately may be helpful. The distance formula applied to the ball’s current velocity can give you its magnitude, or current speed.

Normalizing that vector by dividing the x and y components by the magnitude gives a unit vector in the ball’s current direction, which you can then scale by the new higher speed.

brick breaker exercise 10 extra lives for score milestones

Task: 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, consider the total number of bricks, and the potential effect of a chain bonus from the earlier exercise. Update the game's code to award a new life when the player crosses milestones. Signal the milestone information on the title screen and/or in-game.

Discussion: You'll need to prevent the game from rewarding the same milestone multiple times. One easy way to achieve this is to store a new variable as the number the player needs to exceed to earn the next life, increasing up that number by some interval each time the player's score exceeds it. This approach has the added benefit of making it easy to show the next scoring goal. When the player loses reset any counters or other values used by the milestones to track progress.

The other problem that arises isn't technical, but is instead a design issue. How can you prevent a very capable player from playing the game indefinitely? Solutions include increasing the distance between milestones (so that each additional extra life is harder to earn than the previous one), limiting the number of extra lives that can be earned from milestones before no further extra lives can be won, or both.

brick breaker exercise II

add audio

Task: Copy in and integrate SoundAndMusic.js from the sound section. Make a list of which events you'll want sounds for in your game, and either generate or record and edit sounds to fill each of those purposes.

Discussion: Whenever something in your game changes in a discrete, momentary way, rather than a continuous way over time, you'll want to have a sound effect to highlight the event.

Obvious moments for a sound are physical events, including the ball colliding with each type of object that can affect its motion. Other less physical examples include losing a try by missing the ball, clicking past the title screen, and earning an extra life from a scoring milestone. There are at least a few more events that could benefit from having sound cues. You'll again need to use Audacity to convert the files into both the ogg and mp3 formats to ensure cross-browser compatibility.

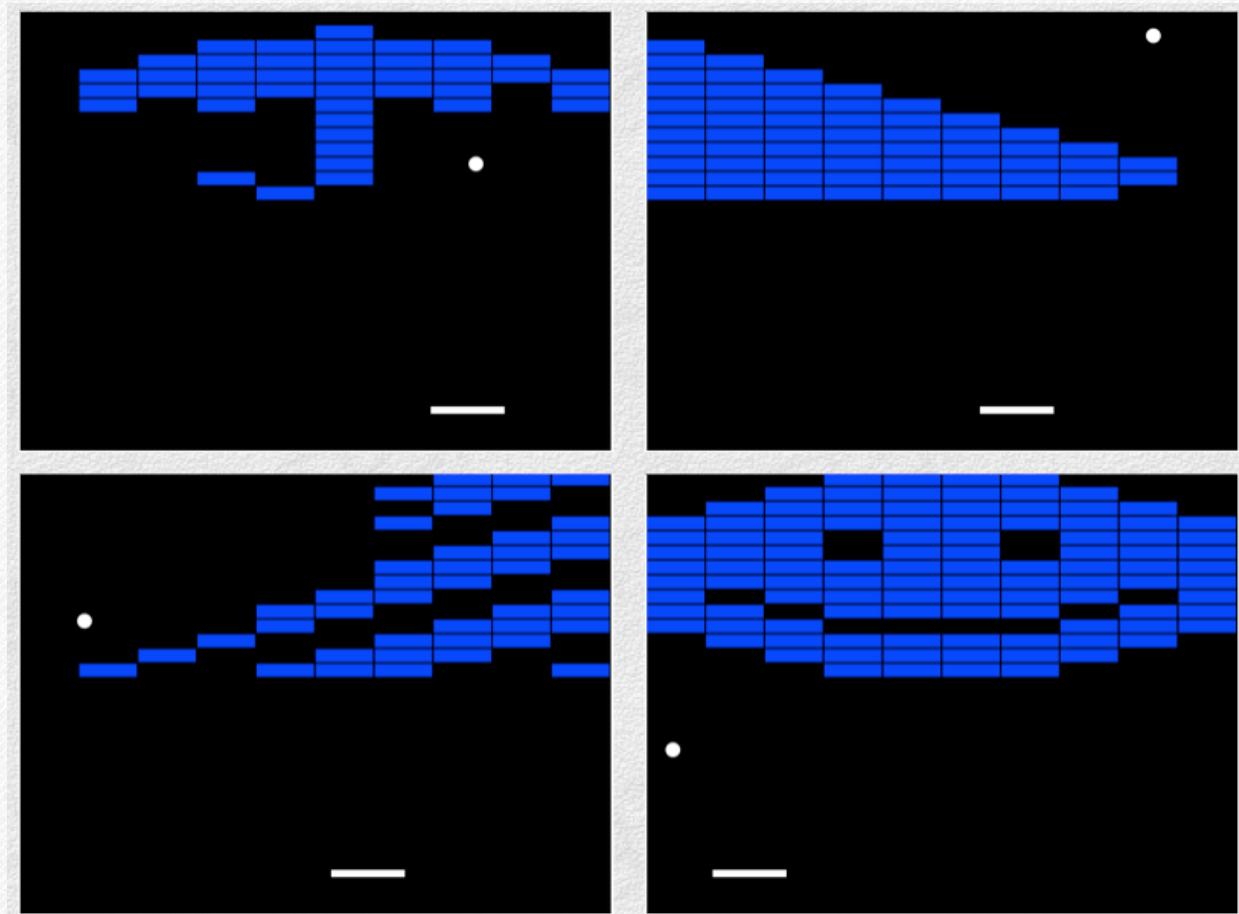
If you'd like to find some music to play for the game, browse for one at <http://incompetech.com/music/royalty-free/> (if you release the game you'll need to include a proper Creative Commons Attribution mention of Kevin MacLeod) or a similarly confirmed source of music that you have the rights to include. Finding suitable songs and narrowing down to ones you want to use for menus and gameplay can be fun, and a great way to help set the game's tone and atmosphere.

brick breaker exercise 12 bricks layout from array in code

Task: The brick grid currently contains only one type of brick, and all bricks always begin turned on. To support different types of bricks, and different layouts for different levels, you'll need to have control to lay out where bricks start. Instead of the code assuming all bricks should be filled in, create an array in the code in which 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. Upon adding different kinds of bricks, you'll be able to use other numbers to represent them.

Discussion: Refer to the Racing game's Track.js or Warrior Legend World.js for examples of how to specify your tile positions from an array. You'll want to make, display, and modify a copy of the level grid so that you can easily restore the layout when all the bricks get cleared or the game resets.

This will also involve changing how bricks get counted. Detecting how many bricks the player needs to remove cannot be determined by multiplication, which only works for a filled grid. Every brick will need to be counted separately, since many will start already missing for hand-designed layouts.



brick breaker exercise 13 different kinds of bricks

Task: In the 2D array the game now treats 0 as meaning no brick and 1 as indicating a plain brick should be there when the round begins. Add support for three more brick types: number 2, meaning a brick that can survive two hits before breaking, number 3 for a brick that has to be hit three times to break, and number 4 for an unbreakable brick. Create a new image for each of these bricks, and add a few of each type to the brick layout to test their functionality in-game.

Discussion: 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. The unbreakable brick will have different collision code, since it won't degrade when hit. The game will also need to disregard unbreakable bricks when counting the number of bricks left, since they'll still be present when all breakable bricks have been cleared.

When testing these different brick types don't worry about making levels that are especially well-designed with aesthetic balance or gameplay pacing. Set up the whole bottom row as the brick type you're currently working on to make it easier to confirm whether it's working or not. Level design can be done more thoughtfully later after these each work.

brick breaker exercise 14 one brick collision per trip (retro)

Task: In the original games that started this genre, ball to brick physics behaved in a way that many modern players find buggy and unnatural. The ball in this style will never bounce off the side of any brick, so besides when it hits the canvas edge it can only bounce vertically. More importantly: the ball in this style only hits one brick per each trip from either the back wall or paddle, otherwise it passes right through any other bricks on its way back after bouncing. Remake this behavior.

Discussion: This style of ball-brick collision changes how the game feels. It makes hitting the back wall far more difficult, since hitting any brick will stop 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 stubbornly trapped back there, bouncing upward upon each collision until it can get down to the paddle without bumping any bricks on the way.

In the classic games additional design features affected 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). This also made the player's paddle cut to half its normal width until the ball was next lost.

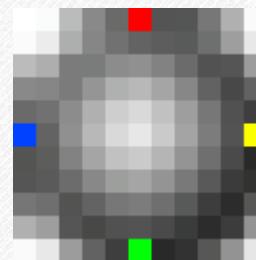
Most modern players do not tolerate this type of “physics” even though it has some advantages for gameplay. I suggest creating a true/false toggle in code to easily turn it back to the modern solution explored in the exercise before this one.

brick breaker exercise 15 give the ball thickness for collision

Task: Up to this point the ball has been treated as a point for the sake of collisions. For a tiny ball this is forgivable as an approximation. The ball in this game is borderline too big for that to look right, since the ball sometimes skims through bricks as it passes. Improve the ball's collision detection so that the ball behaves as though it has thickness according to its visual size on-screen.

Discussion: There are ways to use math to precisely calculate the collision between a circle of arbitrary size and a rectangle. Those general cases are likely more complex than is needed here though. You don't have variable-dimension rectangles at random angles being struck by super high speed balls of different sizes. All the bricks are identical and upright, the ball has a maximum speed, and the ball is always the same size.

Remember: you have an efficient way – grid division – to look up whether a point is over a brick. There's a rough and very simple solution that works as long as the ball is big enough relative to its movement speed and small enough relative to the bricks. Check out this ball, zoomed to show its edges:



The colored dots aren't meant as part of the ball's appearance, but instead show imaginary points for the top, right, bottom,

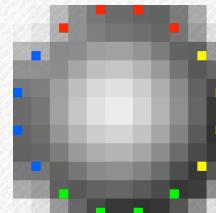
and left edges of the ball. These can be found during play by adding or subtracting the ball's radius measurement from the ball's coordinate. The left point is (ballX-BALL_RADIUS, ballY) and the bottom point is (ballX, ballY+BALL_RADIUS). Instead of checking the ball's center point for brick collisions, make a helper function that checks whether a given pixel location is over a brick, then do the following tests:

- If the ball is moving leftward and its left center point is in a brick, bounce horizontally
- If the ball is moving rightward and its right center point is in a brick, bounce horizontally
- If the ball is moving upward and its top center point is in a brick, bounce vertically
- If the ball is moving downward and its bottom center point is in a brick, bounce vertically

If the ball is too small relative to its movement speed, it can go from having all four points outside of bricks to next having all four points inside a brick. That produces unwanted behavior. A special case could be written to check if all four points are inside a brick, to then fall back to using the type of calculation which handled the ball earlier based only on its center point.

If the bricks are too thin relative to the ball, the ball could still squeeze past them. This can be fixed by checking more than just the four points. A loop could cycle around the ball's edge in a short angular hop, handling a collision with any of the top several points as top hitting a brick, any of the right side points as triggering a right side collision for the ball, and so on. The

diagram for the ball's collision would then look more like this (pixelated to show relative scale):



Try fewer points before going crazy on point density along the edge though.

Getting the ball to play nicely with this method for collision thickness may involve a bit of experimentation with maximum ball speed and the ball's size.

If you're wondering whether all this trouble is worth it, keep in mind that the ball is the only moving object in the game, and in a way it's the closest thing the player has to an avatar in the playfield above the paddle. All attention will be on it!

Also despite the number of calculations involved per frame, as I often like to point out a modern processor really won't break a sweat over this kind of stuff. You could probably have dozens, even hundreds of balls or more bouncing around the screen using these kinds of approaches, and if there's slowdown it'd be more likely due to all the ball images being drawn than the collision calculations being performed. For a single ball on the playfield don't be too worried about the efficiency of its collision code, go for what gets you the behavior that you feel looks and feels right in play.

brick breaker exercise 1b temporary power-up drop downs

Task: When bricks get removed there should be a small random chance of a pill-shaped power-up falling straight down from it. If the player paddle touches the power-up before it goes off screen, add a temporary special power for either the ball or paddle. Powers should only last a limited time, a certain number of uses, or until the ball is lost, depending on what you feel best fits each kind of power-up.

Discussion: Besides adding some variety and excitement to the play experience, by making these power-ups falling items to catch they serve to distract and pull the player from their primary responsibility of blocking and aiming the ball. Some ideas for power-ups commonly found in this genre include:

- Fireball – the ball glows red and for some time cuts clean through bricks one right after another without bouncing
- Cannon – When the player gets this, a small cannon, rocket, or laser gets added to the player's paddle, probably with limited ammo. Clicking fires a shot straight up 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 and a life isn't lost until the last of the multiple balls exits the screen.
- 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 – Give a sizable gob of points, good toward reaching the next extra-life milestone.

There are a few different steps involved in getting this type of feature working. First get power-up abilities working when different number keys are pressed. Press 1 for fireball, press 2 for cannon, press 3 for multiball, press 4 to enable sticky ball, etc. That will make it quick and easy to test functionality, rather than needing to spend time playing hoping a certain one you want to test will appear at a time when you can catch it. Work on functionality for only one power-up at a time.

Once they're each working with a key press, only then add the new class for the power-up capsule, giving it a constant downward movement and its own graphic. Manage an array of power-up capsules so that you can have more than one on screen at a time. Check for collision between the paddle and the capsule. When the two overlap remove the capsule from the array and trigger the corresponding power.

Depending upon how flexible you make your multiball implementation, you may wish to enforce a maximum of three balls for simplicity and balance. You can build this feature in a way that all three balls in play could split into three more, and so on, rapidly filling up the playfield if the player can keep them from falling off the bottom. Alternatively, you could just disallow this item being spawned if one is already falling or if there is currently more than one ball in play.

brick breaker exercise 17 make and support multiple levels

Task: The ability to define the level layout in code, complete with several types of bricks for variety, was covered earlier. To make full use of that though, the game really ought to have at least several different level layouts. What good is having a way to design levels for the game if no levels get designed?

Discussion: Create multiple level arrays in code, arranging the brick types and spaces to form patterns and challenges. Decide how you would like to let the player access these.

The simplest way is to have the levels play in sequence, so that to progress to the next stage the player has to clear the current stage, resetting to the first upon running out of lives.

Another approach would be to randomize which levels load each time the player clears a stage. If doing that find an approach that avoids showing the same level twice, such as putting all stages in a list and randomizing its order.

The third option is to let the player pick which stage they want to play, for example pressing a different key from the title screen for each level, or cycling between stages with special keys in-game. Even if you don't leave this type of access in the final game for release, level skip of this fashion can be very useful for testing your designs as you work.

The level arrays each be coded as separate variables, or set up as an array of arrays. The latter is convenient in that it allows you to set or switch levels based on a single index into the master level array.

GAME 3: TURBO RACING COMPLETION EXERCISES

Unlike the abstract context and gameplay of Brick Breaker the Racing game takes place in a more literal space. Racing has cars and streets, not rectangles and circles.

This creates all kinds of creative opportunities. You can make cartoon settings, select between drivers with different stats, or upgrade upgrades. You could even travel to different real-world locations, made recognizable by landmarks and art for settings. Weather and time of day can be added, with an effect not only on the visuals but also on the vehicle steering and visibility. Other cars can be made computer controlled and given different personalities in how they drive.

For Brick Breaker good art meant fancier decorative abstract rectangles in one form or another. For Racing, you could even pre-render 3D models, form large decorative pieces involving multiple adjacent tiles, or modify the movement and draw code to show the racing from a three-quarters or isometric angle instead of directly overhead.

With this project you're not merely making a game, you're developing a little virtual world that has vehicles in it.

racing exercise 1

new tile types: grass and oil slick

Task: Create two new types of level tiles: grass and oil slick.

Drawing their tile art, add them to the track map, and program special functionality for when the cars drive over them. When either car drives over grass make it move significantly slower, and while driving over an oil slick don't let the car steer.

Discussion: An easy way to ensure that your new tile art for grass and oil slick squares are the correct size and format is to begin work on them by duplicating (or copy and pasting) any existing tile image, renaming the copy, then drawing over it. Alternatively, dimensions can also be found in the source.

The new track tile types should be defined in World.js as 6 and 7. Load the images via `loadImageForTrackCode()` in the `ImageLoading.js` file. To add test squares to your track insert a 6 and 7 into the numerical grid in `Track.js`'s `trackGrid` array. .

As for the intended consequences, look in `carMove()` in `Car.js`. How you make the player slow down on grass or lose its ability to steer while driving over oil are up to you.

racing exercise 2 day/night or theme tile sets

Task: Merely by switching which graphics are used to draw the track, the same layout can be easily 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 edit or redraw each tile to fit its new theme.

In-game, let the players toggle between the themed art sets by key presses. Explain those clearly with an on-screen textual hint which shows up before the cars start moving. When we later add support for multiple tracks we can make specific themes load for specific layouts by making that part of the track data, but for now key toggles work for testing purposes.

Discussion: Create a new variable in the World.js file that reflects your current theme. Update ImageLoading.js so that instead of loading a single array or tile images it will form an array of arrays. The first index in the arrays corresponds to the theme, the second index still indicates which sort of tile the graphic there is for. Back in World.js use the theme value as an extra index into the tile art being displayed. In Input.js check for when a certain key is detected – any key that you may like for this purpose – and there in the code cycle the number for which theme art is used to display the track.

racing exercise 3

basic car-to-car collision

Task: Currently cars drive harmlessly through one another. As a first take on getting collision working, detect when the center of each car comes within some distance of one another. The distance can be based on the average of the car's width and length. If the cars are within that range, nudge them apart.

Discussion: Besides looking a little funny when cars ghost through one another, this also has real gameplay implications since lack of collisions prevents players from aggressively blocking one another or otherwise interrupting the other player's movement. With the cars unable to bump, practically speaking they may as well be taking turns to race.

Real cars generally shouldn't touch, but since these are videogame cars they're even safer to slam together than toys. If you're interested in making a more realistic racing game, without bumper cars action, you could just add more severe consequences for collisions. For example, you could cause a car to lose when it hits a wall, or declare a draw if cars hit one another. In either case, real vehicle(s) might well be wrecked and unable to complete the race anyhow.

racing exercise 4

terrible computer controlled car

Task: Though being able to support two players is a great feature to have, one of the advantages of a computer game tends to be that you can play it even when no one else is around to play with. Programming a way for the computer controlled car able to drive the track all on its own without bashing against walls is a harder problem that we'll take a look at solving in a later exercise. For now don't stress over that whole "without bashing against walls" part.

Set up a new true/false value in your code that lets you set the second player's car to be controlled by the computer instead of accepting keyboard input. When the car is computer driven, have it semi-randomly toggle its hold keys on and off every fraction of a second or so, so the car will fidget and roam.

Discussion: Though the computer driver seems confused about what it's doing, with a bit of tuning this small change can already make the track feel less lonely during testing. Even after the car has a way to drive the whole track, this semi-random wiggling mode may come in handy to help it recover and find its way back if it gets nudged off course while racing.

This exercise lays the groundwork so you can toggle between two player mode or single player versus computer. After this, when or if you wish to, you can devote more attention to giving the computer controlled car improved behavior.

racing exercise 5

keep and show true race time

Task: Most racing games display a clock, and freeze the time when the player finishes, to show how long it took to complete the course. Find a way to calculate, update, and display in the game how long the race has been going. Ideally, do this in a way that reflects actual time passage, measured in seconds, rather than counting on stable frame rate.

Show the time in a stopwatch format. Pad leading zeros to show 2 digits of minutes, 2 digits of seconds, and a tenths place for precision.

Discussion: Because of slight inconsistency in how many frames per second the game will run, the timer should not be doing its counting based on an assumption of how much time passes between each frame. Rather, store a start value based on the system's actual time of day information when the race begins, and during the race update the time display based on the subtraction of the current system time from the time at which the race started.

As for how to add leading zeros to your time display (ex. showing “00:02.5” for 2.5 seconds), there are many different ways to do that in JavaScript and it’s a very common thing for programmers to do in JavaScript. If you find yourself feeling at all stuck on it, feel free to search the web for other solutions to compare and adapt to your situation.

racing exercise b nitro boost

Task: Give each car a new key that when pressed gives the vehicle a short but significant and noticeable boost in its speed, along with temporarily decreased maneuverability. Only let each car do this at most one time per race.

Discussion: As an element for the gameplay design of a racing game, the temporary nitro boost is a great way to add a strategic choice to game about when in the track the car can get the most benefit out of using it. This has the added benefit of dramatically exaggerating who is in the lead and by how much during the race if one car has used their nitro while the other has not.

The inclusion of this feature also creates more opportunities for last minute turnarounds by either player, adding to the excitement.

When later implementing support for multiple laps, nitro can be recharged or accumulated upon completing laps. For now give each car a way to track whether its nitro is available or already spent. Even though the effect of nitro should be short, it will likely work best if done over multiple frames, and so a countdown value in the car code for how many cycles of boosted drive physics are left will likely be useful.

Note that changing the speed alone likely won't be enough. When the car is boosting use different tuning constants, too.

racing exercise 7

support multiple tracks

Task: As with the Brick Breaker project, it would be a shame for us to have a versatile set of level features but only a single track in the game. Set up some keys to switch easily between multiple track layouts, modify the code to be able to read one out of multiple track arrays, and create another track or two.

Discussion: When creating the maps it will be easiest to take on the challenge in layers, or incremental iterations.

Before worrying about the placement of decorative elements like flags or trees, before placing lap checkpoints, and before filling in the area beyond the drivable area, get the path figured out with walls placed on an otherwise open road. Alternatively, copy and paste some lines filled with walls as your starting point, then carve roads out of it with 0's. Try driving the map to see where corners are too sharp, roads feel wide enough (or too wide) in places, and how you feel about the overall length of the track.

Once you've made a track shape that you find plays well and has an okay visual appeal, then experiment with introduction of grass and oil slicks to the stage in various ways. Merely because these are working in the game does not mean that every level necessarily needs to use them. It's a sign of amateur level design when a stage has every feature and asset in the game forced into it. Pick what the level is about – straightaways, oil slicks, narrow roads, big curvy turns, etc. then design the map around that.

Loading the level layout grid irreversibly removes some of the information from it, such as where the player car begins. A good way to preserve map information for future loads is to not access it directly, but instead play on a copy of the data. This way, the active level grid can just be copied back over again from the original level data if the stage needs to reset.

The cars currently can only start facing one direction. Ideally, to give the newly designed level layouts more versatility, there should be another map layout number which can be placed adjacent to the player start position as a signifier of how the cars should be oriented when that level begins.

racing exercise 8

ramp tiles and airborne cars

Task: Jump ramp blocks can add an element of verticality and danger to the game. Whereas real ramps would only work from one direction, for simplicity let these work from any angle. An overhead depiction like a pyramid might make the most sense. When a car drives into this tile send it flying.

Discussion: While the car is airborne:

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 while the car is low the actual object will overlap its shadow instead of vice versa.
3. Don't check for or handle collision response with any walls, grass, oil slicks, or other obstacles while it's in the air. Checkpoints and goal line should probably still count.
4. While airborne the car shouldn't be able to gain or lose speed, and it shouldn't be able to turn.
5. The car's vertical height value should gradually increase from ground level, then decreasing back toward zero.
6. The faster the car is going when it hits a ramp, the higher it should reach in 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, assuming you'd like those shortcuts to be valid.

racing exercise 9

sound effects (advanced!)

Task: Give the car sound effects for the motor, steering, brakes, collisions, ramping, and landing.

Discussion: Adding sound effects to previous projects was a pretty straightforward application of the sound code prepared in the audio chapter. For those sort of games sound effects are a simple matter (within the code, at least) of adding a call to `play()` within the block of logic that you'd like to trigger the sound. The sound would then play through once in its entirety. The exception was the music, which would be set to loop.

Car sound effects are more complex from the code side. You'll need to devise a new way to specify programmatically things like having an engine loop start or stop while the gas is held, possibly switching between different engine tones based on the car's total speed. How long brake sounds screech or which brake sound effect should play depend on what the car is doing when the brakes are applied. Collision sounds should likewise also depend on the conditions of the collision.

The relatively simple and widely supported Audio way of playing sound effects has ways to approximate the above in a decent way, although if you'd like to investigate programming for more complex sound features look into the Web Audio API. Unfortunately at the time of this writing Web Audio is not supported by all major browsers, and due to the lag of people updating their installed browsers that may still be the case for quite some time even after it gets added to new editions.

racing exercise 10

in-game track editor gui

Task: By this time in the game's development there are quite a few tile numbers to keep track of when making a map layout. Between walls, decorations, ground surfaces, player start positions, and invisible checkpoints that's a lot to memorize. Even if you know them all by heart it can be hard to visualize the level by glancing at the layout grid.

Since your game already loads the track art and displays map data it could work as a great starting point for point-and-click map making functionality. Create a level editor mode within the game that allows you to change to a particular track layout, use mouse interactions to change track elements, then display the level's layout information in a way that you can copy into your source code to drive later.

Discussion: Besides making it vastly easier to design a level with aesthetic considerations in mind, this also avoids pesky problems that inevitably happen while hand editing map files. You no longer have to worry over missing a comma, leaving out a number in the grid, or otherwise accidentally breaking the format needed by the program. The editor keeps it right.

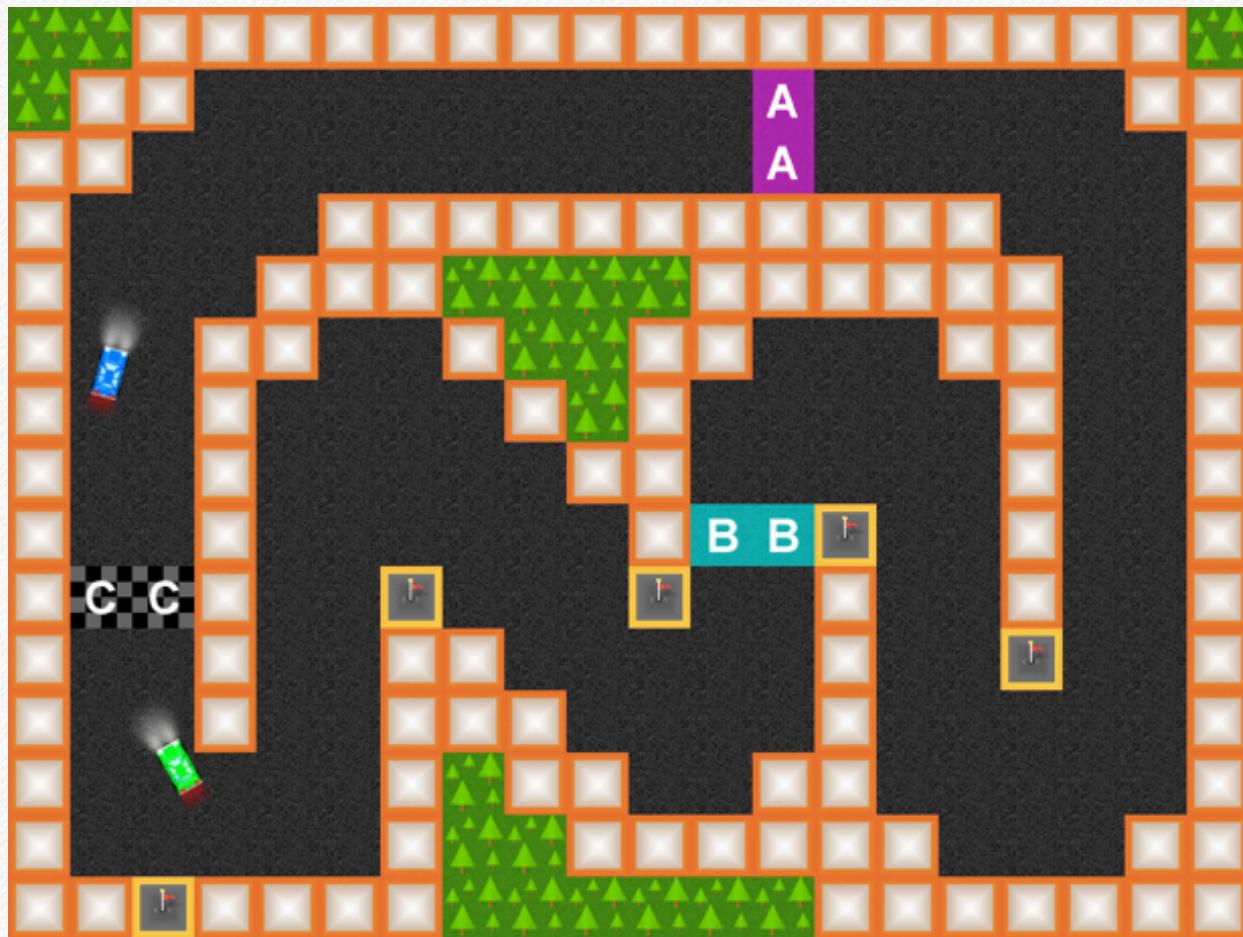
To get this feature working at all first, having a set of keys mapped to the various map control functions would be fine. Eventually it would be nice to have a track 'paint palette' that can be used to change which type of tile the mouse places. To avoid the special keys being pressed during play, have a single key to toggle 'editor mode' and ignore the keys if it's off.

racing exercise II laps with checkpoints

Task: The way the goal line is currently implemented it acts as a wall, except that the player who first crashes into it wins. An issue with this is that it means a track cannot be a looped circuit, it can only be designed as a meandering path to be driven from start to finish once as the whole race. To give each race more longevity, change how the finish line works so that cars can drive over it and increase their lap counter. When either car's lap counter reaches three, declare that car the winner. Redesign the test track a bit so that the finish line connects back into the track, sealing in the bottom left where the finish used to be with a few wall segments.

Additionally, you'll need to implement a way for the game to track with certainty whether the player has actually driven the full course for each lap, rather than simply driving on and off the finish line to rack up laps.

Discussion: One good way to track whether the player is completing laps and driving in the proper direction is to make two or more strips of the track invisible checkpoints, using the finish line as a third but visible checkpoint strip. Have each car keep track of its most recent checkpoint crossed, for comparison to the one next driven over. The whole strip across the track has to be marked so that the player can't accidentally drive past it without going over it. Though these strips would look and behave in-game indistinguishably from normal road sections, in code they serve as useful markers for which way the cars are driving. An illustration may help:



The A and B sections in this image would be indicated in the track data with unique numbers, but would look like plain road during play. The C tiles would look like the finish line. Each car stores which checkpoint it last crossed, counting laps only if the car crosses C having last crossed B. Besides only counting complete laps this will also prevent the cars from earning laps by driving the entire track backwards.

Even though the checkpoints are labelled in the picture with letters, in the car code they can be easily dealt with as numbers. I.e. the cars could have a `lapProgress` value that gets set to 1 for driving over A, 2 for driving over B, and back to 0 for crossing the finish line at C. A lap gets counted if and only if that number is 2 when being set back to 0.

racing exercise 12

collision at front and rear of car

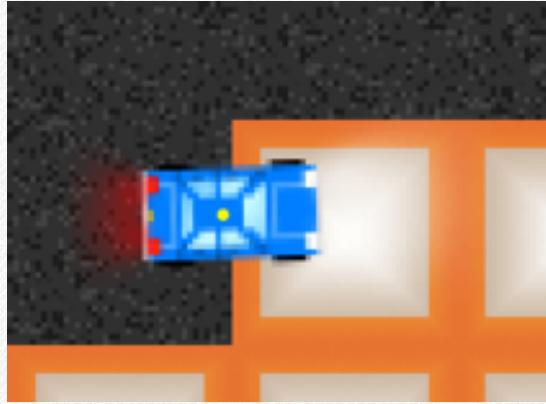
Task: The cars currently can drive halfway into a barricade. This happens because for collision purposes the car gets treated as a single point located at the center of the image. Your objective in this exercise is to improve the collision logic between the car and track obstacles so that the car can no longer overlap the barriers, or at least not nearly as severely (a little bit of partial overlap is less of a big deal).

Discussion: There are a few ways to approach this problem. Many solutions come down to checking multiple points around the car instead of only one. If any point is found over the wall, the car gets blocked or gently pushed away. For example, with the use of `sin()` and `cos()` to account for vehicle rotation, you could determine the approximate location of each headlight and tail light, detecting if any of those points overlap a wall.

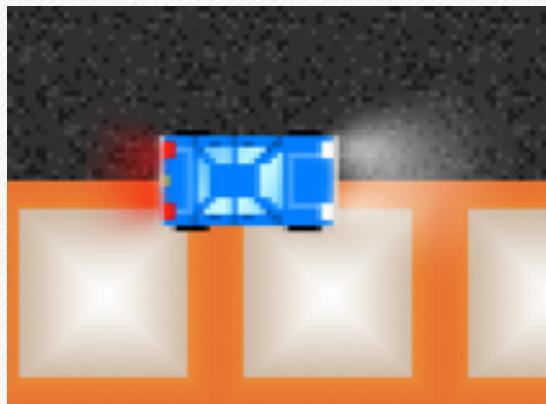
A slightly simpler check would use one point at the front center of the car, and one at the rear center. That wouldn't be as thorough, but could at least eliminate the most distracting and common causes of overlap which happen when the car runs directly into a wall or backs into one.

Generally a decently working solution that does what we want for most common gameplay situations frees time to spend on other features and aspects that the player is more likely to notice and appreciate.

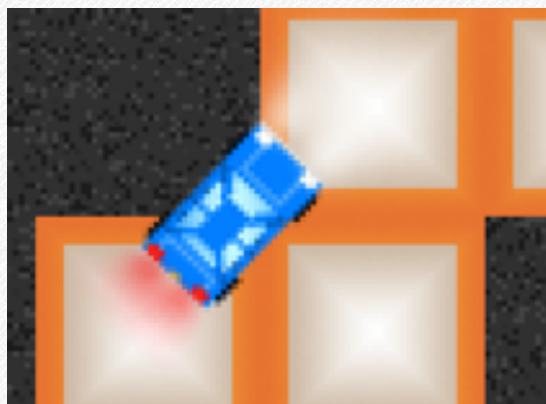
Here are tradeoffs of the potential solutions:



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.



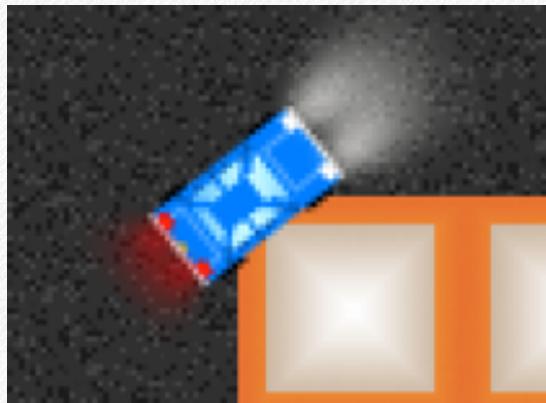
Another issue is that the car can drive with its left or right half atop barriers. The shortcut method of checking for collision only at the front center and back center of the car would not help prevent this problem.



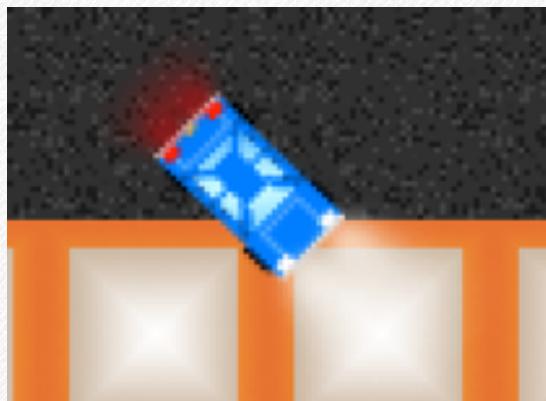
When someone is playing they are unlikely to wind up in a jam like this, but trying to get the car into this position can be a good way to check a new collision solution. Does it prevent you from getting here? Can you somehow get the car totally stuck?



Checking for collision at the car's four corners, roughly under each of the lights, can yield pretty good results. With that solution the car cannot drive farther forward from this position.



But all shortcut solutions inevitably still have some special cases that don't work perfectly. Checking the four corners would still let the car get here. Checks at the center of the sides could prevent and correct for this.



With the car now using multiple points away from the center, you may need to catch low speed steering that slides the collision points onto a wall. Not moving forward was enough when the car was a point. Now it may not be able to turn.

These improvements and issues are important to keep the car from overlapping walls and other obstacles. But, for knowing if the car is over grass, oil slick, or some other ground indicator, it may still be fine to check the center point, treating that as the value for the whole car. There's no need, in other words, 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. Conceptually the center point will work fine as an average position for which zone the car is or isn't in.

(For the goal line, however, checking the front part of the car might be nice. This could make a subtle time difference, or even affect who wins.)

racing exercise 13

larger scrolling track (one player)

Task: The track currently 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, but not so much of a change that we'll have our hands full trying to fill in that space with test layout. Have the screen pan to stay roughly centered over Player one's car. Prevent the camera scroll from showing beyond the defined track tiles (or let it happen momentarily then correct it).

For now ignore Player 2's car. We'll work on a solution that can account for that other vehicle later.

Discussion:

At a high level, you need to do is create values to track the camera's horizontal and vertical scroll amounts, adjust those to keep the view over the car, and change the draw code to offset all images by those camera pan variables. Getting that last part working first will make it easier to test the other steps.

We could manually subtract the scroll values from the x and y draw position of every image displayed. By doing that when we increase the camera's x-axis slide all objects would shift their draw positions left, creating the impression of a camera moving right. There's a better way to do this, though. Before any draw code happens each frame a call can be made to `translate()` later draw calls made during the frame. The `save()` and `restore()` functions can be used to record and bring back

the default positioning, which will be important for example to keep any user interface imagery from scrolling too as the camera position changes.

The remaining piece of this exercise is updating the camera offsets every frame. To keep the player's car in the center, the camera scroll values need to be based on the car's location, except offset by half the screen's width and half the screen's height. To avoid showing map data (or lack thereof) from out of bounds, when the player nears an edge of the track the camera needs to be held back above the valid map area.

(An example of this camera scrolling functionality can be found in the Section Bonus - Mini-Extras\grid-scrolling\ directory.)

racing exercise 14 better computer driver

Task: Much as the invisible checkpoints can be added to the track for the game to have additional information that the player is unaware of, invisible data points can also be added to the map to give the computer driver hints about where to drive next to complete proper laps. Set up the map to accept new numbers, in a range (ex. from 50-75), loading the center of those positions into an array of waypoint (AI path notes) information while replacing them with road tiles for play.

Get the computer driver to navigate by a “connect the dots” approach: driving from each waypoint to the next higher until no higher waypoint is available, then repeat the loop. If this change is done right the computer car should be able to complete laps on its own, or even win the race if left alone.

Set up a key to toggle one or two player mode. Adding a proper clickable option on a brief title screen will be nice to allow the player to choose whether they wish to play alone versus the computer or against another human player. That presentable button is no substitute for a toggle key to use while testing out the computer’s driving and recovery behavior though, since with the key you can get the car into tough spots to assess its ability to recover and get back on track.

Discussion: Unlike the checkpoint strips on the course, this data isn’t needed as part of the track grid during play. These can safely be replaced with a plain ground’s value during level

initialization, so long as a copy of the original layout is kept in memory to fully reload the level from.

The mathematical **dot product** operation (web search for it) will be useful in determining whether the computer car needs to turn left or right to steer more toward its next waypoint. Specifically, here the property of dot products that you'll be able to exploit is that it can be used to figure out which side of a line a point is on. Used in relation to the car's facing, this can yield a useful answer to questions about spatial relationships such as, "is the car currently facing toward or away from the target waypoint?" and, if it is already in front of the car, "is the waypoint to the right or to the left in front of me?"

While troubleshooting your code to get the car to follow the waypoint chain it will be very useful to surface, or visualize, your waypoint information. Draw circles on the nodes, color the current or next one differently, and connect the nodes in order with lines.

Color coding and lines can also be used to help visualize whether the dot product math is being used to get answers to the questions you're using it for. By connecting that visualization to the car you can steer as the player you'll be able to fluidly test it at various angles and positions far better than trying to test by only watching the automatic driver.

If the car gets stuck, switch to a recovery mode that wiggles and turns semi-randomly in reverse until it's back on track.

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.

It's possible to write functions to determine whether the AI car has direct, unobstructed line of sight to its next waypoint. For that matter, there are also (admittedly rather complex) ways to try to compute waypoint information automatically to navigate around any obstacles that are in the way. Alternatively, if you're careful to place the waypoint nodes plentifully and in such a way that they maintain line of sight to one another those additional features may be largely unnecessary.

Since the cars are able to bump or block one another there is some definite possibility of a curious player shoving or guiding the computer controlled car off its planned track. Finding a way to make it look like the car is at least trying to respond appropriately to this event can help your drive seem alive.

As a compromise, you could give your car AI a "panic" mode if it struggles too long without reaching its next node. In that state the car could wander randomly until it finds a node, or drop its focus back to the previous node. Be careful about just picking the nearest node, as it could be opposite a barrier.

One other very different and potentially humorous way to handle the player pushing the computer car into a corner could be to have the computer-controlled car temporarily switch to aggressively trying to chase and ram the player's car. Tracking for the player's center point is really no different than trying to steer toward a node.

racing exercise 15 weather effects (visual + steering)

Task: Having weather effects in a racing game is a great way to get more mileage out of the tracks that you've already defined. When it's raining darken the whole screen, draw constantly moving rain drops, and affect car behavior such that it moves in a way more like driving in wet conditions. Feel free to exaggerate the effects of rain, realism might be less fun than making it more noticeable how the driving is different!

Discussion: When it's raining change to a different set of the game's tuning values or program logic to negatively affect acceleration, braking, ground friction, and turning precision (maybe with oversteering, or even sliding while cornering?).

To make it rain only some percentage of the time, simply set the rain true/false value at track start time by making 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, get a random value from 0-100, making the rain setting true if that value is under 30.

For darkening the whole screen, drawing a mostly transparent black rectangle over the screen may work well enough. For rain drawing many tiny short angled lines randomly around the screen 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.

racing exercise 1b

zoom camera for two player big map

Task: When both cars are controlled by keyboard input have the camera zoom out or in however's needed to keep both player cars within the frame plus some reasonable margin.

Discussion: The scale() function on the document's context is the key to doing this. Getting the view to zoom in or out by scaling isn't the hard part though. Where this can get a bit messy is in trying to ensure that the world is never drawn outside the boundaries of the defined area, even as the view scales and relocates to keep both players in-frame. Another aspect to remember is that rather than having the player camera track one car or the other, it should likely center approximately on the midpoint between both cars.

GAME 4: WARRIOR ADVENTURES

COMPLETION EXERCISES

Expansive RPGs and action-adventure both classically share this same type of starting point.

Either could be given villagers with dialog, a vast and varied world to explore, as well as a plethora of magic spells, armor types, and forms of combat.

The hard part of designing these types of games isn't coming up with ideas of what can be done. The hard part is making tough decisions about what isn't going to be done, picking a realistic scope. It's all too easy to get stuck deep into a project far too ambitious for any individual or small team to reasonably finish and release. I've seen smart and talented people lose morale due to getting stuck 20% into making their dream RPGs, leading to a long set back in their game making.

I made a conscious choice to pick a project scope for these exercises such that it won't leave you spending the next year making art and planning town dialog. The exercises to follow revolve around designing a classic dungeon or cave interior for an action-adventure game, complete with multiple rooms, inventory items that empower the player, simple enemies, and two forms of combat. This is a bite-sized way to try it out. If you wish to take it further, know that there be dragons.

warrior legend exercise I four facings for player sprite

Task: Draw four different facings of the player sprite, one for each of the cardinal directions (i.e. North, East, South, West). You should have one of the character's back, one facing right, one facing forward, and one showing the character's left side. Program the player's movement code to keep track of the last direction moved in, and in the player's draw code display the directional sprite that corresponds to the last direction moved. Put simply, walking left should show the character facing that direction, and it should remain facing that direction until a different direction gets pressed.

There's no need to animate the walking or anything else about the sprite image yet. That can be accounted for later.

Discussion: Even though the environment is drawn from overhead, and both movement and attack are handled as though seen from an overhead, traditionally the characters in these types of games get depicted from a side view. This helps to better show their outfits and equipment, making the characters more distinct from one another. It's quite hard to make a completely overhead view interesting as it mostly shows the top of the head, shoulders, and toes.

When working on a side, front, or rear version this character, don't worry about getting the proportions realistic. For both navigation and combat it will work nicely for the character to fill up most of a whole tile's dimensions. This will likely yield short legs with a wide body. In this genre that's quite alright.

warrior legend exercise 2 player health and harmful obstacle

Task: There's no way yet for the player to be harmed. Put a health bar on the screen showing 4 whole units, and create a tile type for spikes that will harm (but not instantly defeat) the player when walked over. Add some spikes to the level layout to test them. When the player's health reaches or dips below zero reset the game.

Discussion: When drawing the spikes attention should be paid to somehow conveying that these can be walked over, even if they shouldn't be. Factors to consider in this include not depicting it as a deep pit or hole of spikes, and to not put a thick border around it. To fit this criteria these will likely wind up looking like short vertical metal sticks or small pyramids on the floor. The same way floor tiles get drawn behind the key and door can be used here so that the spike image can be only the spikes, the rest as transparent to show the floor.

Subtracting health for this purpose isn't as straightforward as it may sound. If health is removed immediately any frame that the player is on spikes then an instant later the player will be out of health. A safety timer of brief invulnerability for a fraction of a second will make it possible for the player to rush over a short strip of spikes incurring only minor harm.

Some games in this genre support health in half-increments. It really isn't much more complex to do. Give the player twice as many units of health as the whole units, and draw every other one as the other half either filled or emptied.

warrior legend exercise 3 flying bat enemy

Task: The bat's movement works like the UFO from Space Battle, except with pauses to rest between flying from one place to another. Let it wander a bit as it flies to help it seem lifelike. When paused use an image with its wings folded up, and when flying cycle between two or three flap poses. When the bat touches the player take away one health unit.

Place bats in the layout the same way you place the player.

Discussion: What makes this enemy type an excellent choice to start with is that its movement does not need to be affected by walls. It can start on, stop on, and fly directly over either the floor tiles or wall tiles. It's also an easy starting choice because although its wings animate while it moves, unlike the player or some ground creatures it doesn't face different directions. It can stay facing the camera regardless of how it's moving.

For the animations, build in a cycle countdown to give each picture multiple frames on screen before skipping to the next one. The animation advancement timer needs to be a separate variable than which frame is currently being used. If the picture changes on every logic update it look like a flickering mess. More advanced and general solutions are possible, and can be considered as part of an animation class later. At first the bare minimum to get it working will look the same to the player.

As with other things that we've worked on adding to the game it's probably easiest to first work on getting one bat in play, then building an array to support multiple of them.

warrior legend exercise 4 melee attack in faced direction

Task: The player needs a way to defeat the bats. Give the player a sword to use which flashes a swipe in whichever direction the player is facing (i.e. in the last direction the player has moved in). If the flash overlaps the bat, remove the bat.

Discussion: The sword swipe could be handled as part of the player class. A slightly more general solution though, since other attack types will be added to the game, is to have the flash basically a projectile that spawns directly in front of the player, briefly freezes the player's movement when fired, and vanishes on its own shortly after. It's not as roundabout as it seems to handle the sword in that way, since by doing so it can help reduce the work to later add ranged attacks.

The other part of this exercise is putting a reload timer of sorts on the player, to prevent unreasonably rapid-fire attacking. This can be done similar to every other countdown delay.

warrior legend exercise 5 color coded keys and doors

Task: Make different colored keys and doors (or locks). The red key opens the red door, the blue key opens the blue door, and the yellow key opens the yellow door. Redesign the level to use these to enforce the same order, removing some of the excess door and key craziness.

Discussion: There's a pretty contrived trick being used that was already enforcing an order to the exploration. In effect, even though all keys could open all doors, the player had to enter the door first that would give enough keys to get past the double doors, to then get enough keys to then get through three doors.

The level should be a lot less cluttered after the color coded keys are added. This is great because it will leave more room for enemies and items. It's also worthwhile because having an increasing number of doors appear between areas really doesn't scale nicely to four or more, soon they'd be taking over the whole room.

warrior legend exercise b recorded sound effects

Task: For previous game sound effects many have been made from bfxr or sfxr, but for sounds in this game try to make all the sounds using a low cost computer microphone and a program like Audacity to edit them into a usable state.

Also part of this task: make a list for which sounds your game will need, and after making the sounds, add them in the code.

Discussion: The easiest way to do this is to make sounds with your mouth. That can be sufficient for certain types of sounds, especially for things like character vocals when your warrior gets hurt or swings the sword. For other mechanical sounds like the sword hitting the enemy, the sword swinging and missing, sword hitting the wall, opening doors, or picking up keys, it may work better to record banging objects together near the microphone.

You don't have to use the actual objects involved. Please do not hit a bat with a sword! Many times the real source of a sound doesn't come across nearly as well recorded as some other exaggerated equivalent anyway. A powerful punch sound effect might be made by breaking celery, even though punching doesn't (really, really shouldn't) actually make that sound. Likewise the sounds don't need to be literal. Even if some environment and action sounds are physical, picking up a key might play a short musical sting, or a chime, rather than making the awful sound of picking up keys.

warrior legend exercise 7 tile-wandering enemy

Task: Make an enemy that walks freely among the walkable floor area. It shouldn't pick up keys, open doors, or move through walls. Whether it gets blocked by spikes or can move across spikes unharmed is up to you. Its only attack can be bumping into the player, so no need to fuss with enemy projectiles yet. This one might be a zombie, a skeleton, a robot, a slime, a mummy, or a Frankenstein's monster-like monster. Whatever it is, pick something that the player won't expect should behave too intelligently. This enemy will simply wander randomly, totally oblivious to the player's position.

Whether you want to let the monsters move through one another or bump into one another is up to you. When they wander have them commit to a direction for a random interval or until they bump into something, whichever happens first.

Discussion: Choosing an enemy type that the player can expect should act unintelligently is a classic trope in games. Robots, giant insects, zombies, snakes, and so on don't give any impression that they'll strategize, coordinate, retreat, find their way around a modest obstacle, or may not even directly respond to the player being nearby. As a developer that makes these kinds of monsters incredibly appealing.

Despite being a sort of lazy trope, these can actually be pretty fun as assorted room filler enemies too. You can even reuse this same behavior for a number of different enemy types, giving them different graphics, speed, and health values.

warrior legend exercise 8

player ranged attack with ammo

Task: Give the player a projectile – arrow, throwing javelin, kunai – and a limited quantity, perhaps 3 or 5. The attack should go in the direction the player is facing. You can hook up the attack to a separate key, or use the same key as the sword swipe but use a separate key to instead cycle between available weapons. The current number of shots remaining, as well as which weapon is currently selected if using the weapon cycle approach, need to be surfaced in the interface as well.

Discussion: There are a few separate pieces to this exercise. Before dealing with ammo limits, I suggest getting the shot working with unlimited ammo, so there's one less concern to consider for what's broken if the projectile isn't firing. As a first pass on projectile movement allow it to pass right through walls and enemies, again to reduce the likelihood of the shot not appearing due to some bug in its collisions get handled.

Being thoughtful about the order of development can make for simpler subtasks, each one being not only faster to make but also easier to test in isolation. This reduces the risk for tricky bugs that can arise from trying to do too many things at once across too many areas of the code. Test your work often!

The ammo type next to the limit will ideally be indicated with an icon, but for a placeholder when displaying it as text on the screen I'd encourage using a label ("Kunai: 3") instead of only the number. This will make for less to explain if sitting a friend in front to test the game before completion.

warrior legend exercise 9 enemy ammo drops, limit max ammo

Task: Having only a few shots keeps the player from overusing them, but without a way to replenish them they'll never get used. Create an icon image and ammo drop that roughly one in every 5 enemies defeated will drop. After a few seconds have the item flicker for its last remaining second then vanish if it hasn't been collected yet. The amount per pickup is up to you: singles, triple batch, etc.

Put a cap on the maximum number of shots that the player can carry at once.

Discussion: Once these are working the player can start with zero ammo, scrounging these up on the go.

The number of shots in each batch will depend on how often you want the player to use that rather than the sword attack, as well as how much damage you make the ranged projectiles do. If multiple shots are needed to bring down a common enemy, more per batch will be useful.

If the limit is too low, for example three, no matter how often drops happen the player may feel compelled to hold on to them in case they're needed for a boss, puzzle trigger, or some other purpose that the sword may not work for. If the limit is too high the player can wind up with practically unlimited arrows by a bit of grinding, after enemy respawns get added. Having a reasonable limit (10?) plus frequent drops will give the player reason to use them, since they know they'll still have a handful saved after use and can find more soon.

warrior legend exercise 10 projectile enemy with facing

Task: Now that the player has a ranged attack the enemies need a way to keep up. Create a monster that's able to fire projectiles. Like the player it should have different graphics for each of the four cardinal directions.

Unlike the other enemies which don't pay attention to the player's position, these enemies should only fire when the player steps into their row or column, at which time on the next AI update occasion have the enemy turn to face the player and fire. Make an exception for if the player is behind the enemy, so that the player can sneak up on it for each hit.

Discussion: These enemies will be tough. In addition to giving them the ranged attack and crude awareness of the player, these would be good enemies to take multiple hits to bring down. An approach that can make it harder for the player to shred through tougher enemies too easily is to push the enemy back a tile or two when struck, forcing the player to reposition between hits.

Tune the projectiles from the enemies to be slow enough that the player has a chance to step out of the way before being hit, especially if all the way across the screen when the opponent fires. Being hit by a projectile that moves too fast to be dodged can be quite frustrating. Damage from projectiles in this case should often be indirect, as when hastily dodging the shot puts the player over spikes or in the way of an otherwise easily avoided wandering enemy.

warrior legend exercise II adjacent rooms

Task: Add support for more than one screen-sized room. If the player walks off any edge of the canvas switch to another layout. Upon doing so the player should get repositioned to the opposite edge of the screen. In other words, walking off the right side of the screen will put the player one room right, and on the left side of the new layout. The player should be able to then turn around and return to the room they left.

Cut some holes into the wall to be able to test this feature. Each time the game reloads a room respawn any enemies in the room, but which keys have been picked up and which doors have been opened should persist between rooms.

Discussion: The game has so many enemy types, but not really enough room to space them out. Worse still, with all the keys and doors on the same screen there's no need to explore. This feature fixes that. With many rooms you have the option to spread out encounters and gating in any way that you wish.

The trickiest part here will be in thinking about how you'd like to connect the room layouts in code. One decent solution for a small test building is to have each room as a separate array, then another simulated 2D array within which those rooms are arranged. For a larger space making a custom tilemap editor, covered in a later exercise, will be helpful.

In general the game should only be moving, drawing, and updating the room that's currently on screen. Empty the enemy array when changing rooms, refilling it in the next.

warrior legend exercise 12 items to increase total hearts, heal

Task: The player has up to a limit of four whole units of health, or possibly broken into eight halves if you've opted to do that. Add two new kinds of items to pick up: one that restores some lost health (2 or 3 units), the other which adds another whole unit of health to the maximum and fully refills health.

Enemies should randomly drop health hearts, but the health maximum increase pickup should only be placed as part of the map data. Like the door and key changes, a heart once picked up shouldn't reappear in a room upon re-entry.

Discussion: Now that total health can be increased, it may also be nice to decrease the player's starting health maximum to leave even more room for upgrades.

Total health increases can be used as either a soft or a strict way of directing the player's journey.

By a soft constraint, I'm referring to having tight rooms with tough enemies which the player is unlikely to survive with only the starting amount of health. A skilled or lucky player may be able to push their way through that, but generally players will figure out from trial and error or from it looking intimidating that they should probably explore a different direction first. That different direction can help prepare them by having some additional health pickups.

The way to use hearts as a strict barrier is to have a long strip of spikes that must be walked across to proceed. Until the player has more health, nothing they do can get them across.

warrior legend exercise 13

In-game tilemap editor

Task: Back when the game had only a handful of different tiles, and only one room to lay out, it was reasonably easy to look at the grid and visualize the room layout, maybe looking up or guessing from context what an isolated number was. Now that there's various kinds of locks and keys, various kinds of monsters, and multiple kinds of items to place, you've probably noticed that it's quickly becoming unmanageable as text. With multiple rooms to lay out, this is a big problem.

So that you can design levels without typing out grids in code, build-in an editor mode for your game. The editor can be as crude and simple as you'd like – the intention isn't to keep it in the final game as a feature for players to make use of, it's only to improve your work flow.

There are two ways that you can export the updated map code. The first way is to have a key that updates the debug text below your program with the JavaScript array definitions to copy into your source code. The second option would download a file that you'd reference from your code. The second sounds smoother experience, but local file security restrictions of the browser can make this a bit complicated.

Discussion: Bind numbers to enemies and items, and bind other keys along your keyboard's top row to room tiles. Mouse over tiles, press the key, and see the change represented. When changing to map editing mode skip spawning the player or enemies, instead drawing their spawn tiles as a still image.

Keep an eye out for ways to automate in your editor anything that can help reduce bugs or save you time. If you find yourself frequently putting down long stretches of walls, or wiping out big regions with open floor, program some code and keys to simplify doing those operations.

To save the player from getting inside a wall while walking between rooms, it's important for the walls, openings, and doors along each edge to match up in adjacent rooms. Trying to keep that up by hand could be difficult, but the program could rapidly and reliably enforce that for you by copying wall changes to the next room over. Note, too, that unlocking a door on either side of a room should also remove the door on the opposite side of the adjacent room.

A word of caution: trying to make an infinitely flexible and user friendly editor is a massive undertaking quite apart from making a game. Work on the editor to save yourself time, to get practice with designing your own tool(s), and maybe so that you can set up functionality that's unique to your game. Beyond that, there are already some impressive, fairly general tilemap editors out there. If your aim is to match or outdo those, I'd suggest taking that on as a different project later specifically with that aim, instead of letting it become distraction from completing the current game project.

warrior legend exercise 14 from key colors to gating powers

Task: Having colored keys that correspond to colored doors is kind of lame. Far more interesting for the player (and you as the game designer!), even though it's really not much different in terms of the programming needed, is to replace the colored key system with a variety of other gating items.

Discussion: There are a ton of ways to do gating. Examples-

- Rocks that must be busted with a pickaxe to advance.
- Water that must be frozen to walk over.
- Raging flames that must be doused to walk through.
- Cracks that you have to change forms to slip through.
- Sleeping beast that must be woken with sneezing powder.
- Boarded up doorway that must be busted open with an axe.
- Deep pit that can only be crossed by use of a magic carpet.

In some cases these don't work exactly the same as one of the color coded keys. The rocks should leave behind rubble. Flames should respawn on re-entry and frozen water should unfreeze upon returning to the room. The sleeping beast might need to walk away once disturbed.

Several of these items, the axe or freeze spell for example, suggest that the player may expect to be to use those as weapons in place of the sword or standard projectiles. You can explore that if you'd like. It could add a lot to the game to have some additional weapon and attacks.

warrior legend exercise 15

Inventory menu

Task: Some gating types might happen automatically from context, like how the locks automatically use up a key. For others though it may make better sense for the player to change to the item and use it manually. Now that there are several types of items in the game a simple inventory menu would be a lot better way for the player to pick between them, instead of cycling through them with a single second. This has the added benefit of being able to have space on the screen to give the player additional descriptive information about what the item is or how it's to be used, in case it isn't obvious from its appearance or name alone.

Discussion: I'm going to go out on a limb here and guess that you probably don't have a ton of items implemented yet. That's fine! Even if you only have only a few, it will still be well worth the practice to work through the challenges involved in popping up a working menu screen over your game. You'll need to pause all the action in the background, and let the player's controls instead select an item from within the menu.

Something to be careful about in game design is to be aware of when you may be subconsciously designing around the limitations (or gaps!) in your existing interface. When the way to change items was cycling through them, more than a few items could feel like too many. With a mouse-based inventory, your gears may begin turning on ways to fill up all that area. In either case remember: you're in total control of the game! You can move your design boundaries in or out however you wish.

warrior legend exercise 1b room change scrolling effect

Task: Having the room instantly switch when walking across an edge is jarring and disorienting. Whenever you cross an edge and change rooms pause the action while one room slides into view and pushes the previous room out of view. This scroll effect will give the space much better continuity.

Discussion: This sounds so easy to do, but in actuality it can be sort of a pain to implement. We want to pan the camera smoothly from one room over to the next, which would be quite easy to figure out (even if hard to really rig up) for physical stage. Here however the room draw code is only set up to render one room at a time, always from the top-left corner of the screen. The position of enemies and shots are not stored in relation to which room they're in. Based on an earlier optimization to workaround enemy positions being screen relative, we're removing the previous room's enemies from the enemy array before instantiating the enemies for the next room.

Yikes!

The thing to remember, and what's nice about starting with retro-inspired projects when new to game programming: this effect was done on a sub-2-MHz processor with RAM that's measured in the single or low double digits of kilobytes. You can figure this out on a modern computer for sure. This does not mean it will be easy, but even if the way that you find to do it is a little brute forced and inelegant, you can get it working.

warrior legend exercise 17 up/down stairs with fade

Task: The world is flat. You can fix that with stairs tiles! Make a form that shows them going up and another form that shows them going down into the ground. Add these tiles, along with changes to the level data connectivity information to allow you to switch between separately laid out floors each with their own adjacent rooms. In editor mode create the ability to add or remove stairs tiles, automatically placing or removing the connecting other type of stairs on the next floor up or down respectively. Since screen scroll won't make sense for going up or down stairs instead have the screen fade to black, then back from black on the next floor.

Discussion: When checking for collision against the stairs it will be necessary to distinguish stepping into the tile from another versus simply standing within the tile. This is because the only guaranteed safe place to place the player after going up stairs is on the stair tile itself, but without a distinction made for stepping onto the tile the player would get switched up and down the stairs repeatedly. To go back down stairs after going up the player should need to step off the stairs then back onto them.

Having even two dimensions of adjacent rooms packed with so many types of tiles was a lot to mess with directly in the arrays, now with a third dimension added I'd give up on touching or reading the level data without using your visual editing mode. With that in mind, you may wish to experiment with more concise ways to format and save the level data.

warrior legend exercise 18 boss fight

Task: Add a boss fight in a room that can only be reached after the player has obtained all gating items and most health increase pickups.

Discussion: Holy smokes boss fights are awesome. They can also be a lot of trouble to implement. They require special art, and the art is often considerably bigger than the graphics used for tiles or normal enemies. Special programming specific to the boss fight can mean new kinds of attacks and new kinds of puzzle elements.

There's a term programmers use for code that only gets used in one place or for one situation: a special case. It's usually used as a derogatory label with negative connotation, like, "this code is terrible, it's a bunch of special cases," because there's a strong and healthy desire to write reusable code when possible, and to find elegant solutions that can satisfy a variety of problems.

Here's the thing: a boss is literally a special case. It's special. And if you try too hard to make the boss work with bits and pieces of code that you're using to power a bunch of other things in the game, it might not wind up as special. It needs to stand out in the player's mind as memorable. It needs to end the game on a note of delightful surprise, which it can't do if it's merely another enemy that happens to soak up a lot more damage before being defeated.

Make the boss special. If that takes special cases, so be it.

warrior legend exercise 19 find some playtesters and playtest

Task: Get people who haven't played your game, haven't seen you play it, and haven't heard you talk about the game. Put them in front of your game with as little explanation or coaching as possible. Observe them playing. Only speak up if they get super stuck or specifically ask for help. Fix what they get stuck on or complain about. Then find a different tester or testers and run the game by them in the same way. Repeat until people don't get stuck or feel the need to ask questions.

Once the game gets released, you won't be there in the room with the player to answer their questions or to help them get unstuck. *The game has to be self-explanatory to be released.* Note that adding a bunch of informational manual-like text to the game probably won't help, as most players skip that and trust the game to explain itself as-needed while they play.

Discussion: If you've never done this before, brace yourself. Testing is humbling. A game that you felt was nearly done always turns out to have more left to do and improve. Pay attention, especially, to things that you feel yourself wanting to apologize over, or explain away, to the players you show the game to. Between sessions find ways to make those go away.

This isn't about the tester knowing more than you about game design. It's valuable because they have a fresh set of eyes. No matter how hard you work, no matter how skilled you are, no matter how much you care for your project, one thing you can never give your own game is a fresh set of eyes.

GAME 5: SPACE BATTLE DELUXE

COMPLETION EXERCISES

Though Warrior Legend clearly requires more content in the form of art, audio, and hand-designed levels, Space Battle deals with a very different group of concepts and challenges.

The gameplay ideas covered by Space Battle will revolve around difficulty tuning, handling with multiple projectiles, spectacle of various kinds, and even a little bit of persistent user information to remember high scores. By keeping the scope classic arcade-sized this project can also be a good one to really finish well, as opposed to getting stretched yourself too thin to get it together and working at all.

That said: as always, this is your game to design at this point. The questions and discussion details I provide are for making this into a more complete action gameplay experience, but you have the power and freedom to make anything you'd like. Space doesn't have to be about lasers and slippery dodging. You could make the game instead about exploring planets and space stations across a galaxy, jumping to light speed, finding clues to solve a mystery between mineral mining and resupplying at trading posts. Such a game might even have an economy of ship and equipment upgrades, and a choice between which crew members you wish to bring on board.



space battle exercise I

multiple shots handled as an array

Task: Currently the player's ship only has a single shot at a time that can be used. Update the code so that the player ship can fire before the previous shot has run its course. To do this, rather than giving the player's ship a single, reused shot, modify the code so that the ship instead has an array of shots.

Discussion: Start the array empty. Whenever the player presses the fire button create a new instance of the shot class and add that instance into the shot array. Every logic frame iterate through the shots to draw them to screen, move them, count down their lifetime limiters, and check shot collisions.

The easiest way to connect a fire key to shot firing is to have one shot fired per key press. However, to prevent the player from spiraling out a near-solid stream of dots while spinning, a reload interval should be added to block the player from firing too quickly again until some time has passed.

Classically, any games in this genre limited rate of fire quite differently than how we ordinarily think about reload rates. In games like *Centipede*, *Space Invaders*, *Gravitar*, and *Galaga*, the player was limited by the number of shots on screen at once. As touched upon earlier, this has the effect of rewarding accuracy by giving faster reload when targets aren't missed.

To simulate that effect when using a variable size array: add fresh shots to the shot array only if its current number of elements (assuming here that you remove expired shots) is less than three, four, or some other preferred cap.



space battle exercise 2

using score to incentivize behavior

Task: Arcade games virtually always have scores. In an earlier exercise we added scoring to Brick Breaker, although that approach was as simple as it gets: 100 points per hit. Decide on some more complex way that you'd like to score this game, based on whichever player behaviors you'd like to reward, and display that score on screen during play.

Discussion: What makes this scoring exercise different is that instead of treating points like a consistent price tag on each object separately, you can incorporate other aspects of the player's performance and choices into the score, depending upon how you want your game played.

Here are some concrete forms this sort of scoring might take:

- Rewarding chain combos with a multiplier – For this, in addition to displaying the score, show a multiplier that starts at 1X. Each time the player defeats a UFO within, say, 5 seconds of the previous one, increases that to 2X, 3X, and so on. If 5 seconds pass and the player hasn't destroyed another alien, drop it back to 1X. Multiply the score per ship by that number, so that destroying UFOs rapidly enough yields massive score increases. This encourages the player to rush, take risks, and have an incentive to improve at the ship's handling.
- Rewarding how close the player is to the enemy – If the player's ship gets very close to the enemy ship, award an extra bonus for a shot that makes contact, maybe 250 rather than the usual 100 per ship. This encourages the player to play in a



more risky way, putting themselves in greater danger and likely increasing the excitement of the game for advanced players, compared to being able to pick enemies off from long range with no penalty.

- Penalizing shots that miss, or costing ammo per bullet – This one is so simple, but really can have a noticeable effect on how people play. Either deduct 50 points for each shot that vanishes without hitting an enemy, or simply deduct 10-25 points per each shot fired. Both approaches encourage the player to be more careful about aiming, discouraging sloppy play in the form of spraying wildly in the enemy's direction. This encourages the player to get better at aiming.

It's good to pick only one of these options though, or decide on a different one of your own. Mix these will be confusing. Scoring logic has to "read" if the player is going to notice and care about playing to maximize it. The more convoluted the scoring system, the less likely the player is to understand it. If the player doesn't understand it, it won't affect their strategy.



space battle exercise 3

custom sci-fi art

Task: Science fiction is a genre rich with iconic craft. While you probably shouldn't rip off the look of a ship from an existing franchise, doing a quick web search for reference may help lead you to some interested ideas of your own about wing shapes, engine placement, cockpit style, weapons appearance, and more. Give your player ship, enemy ship, and projectile a custom and original appearance.

Discussion: Unless you have prior experience in computer art it probably won't look like something out of Hollywood, but that's fine – rather than worrying about photorealism or artistic style, strive for structural recognizability. Someone should be able to tell which part is the cockpit, which part is the engine, and so on.

As with other moving graphics it will be important to use an art program like Photoshop or GIMP.org that supports image transparency, otherwise you'll have a rectangle behind every graphic. Using the PNG format ensures transparency data is preserved and usable. When checking on these graphics in-game if unsure whether you have transparency working right I suggest changing the background color to bright green or cyan, to be able to definitely see whether the graphics have their silhouette, not a black square drawn behind them.

When replacing the player ship graphic, remember to keep its front end pointing right, as that will be the 0 degree rotation in code. A radially symmetric design for the player ship, like an



overhead saucer, should be avoided since how the player controls requires clear feedback of which direction the ship's thruster is pointed in. On the other hand, since the enemy ship does not reorient while moving, an image should be used that's appropriate for sliding around without changes to the way that it's facing.

The simplest interpretation of this exercise could be replacing the image files. However there are many ways that a little code could add more flair to your visuals. Flames could flicker and scale behind the ship while thrust is applied. You could give projectiles an oriented graphic, like a rocket or a laser bolt. The angle for that shot could be borrowed from the ship's at the time it's fired, or computed from its movement by the built-in trigonometry operation `Math.atan2(y,x)` – where y and x in this case would just be the shot's y velocity and x velocity.



space battle exercise 4

title screen

Task: The game currently drops right into the action. Besides not giving the player time to prepare, this also means we lack a good place to post information like the controls, instructions, high score, credits, or title logo. To fix this, add a proper title screen to the game that the player has to click or press a key to get past.

Discussion: You've made a title screen before by this point. This time around take the exercise further. Instead of thinking of this as a placeholder title screen, really work design a layout that you find appealing and feel is functionally complete.

Name your game. Create a logo image of it for the title screen.

If you're not feeling up to logo design, at least pick out a font for the game's title to be displayed in and build a decorative image from it. Ideally locate a font that you are certain you have rights to use for this sort of purpose. Even if no one is likely to come after you for the logo of a practice project, finding a source you're free to legally use can be helpful for future projects, too.



space battle exercise 5 drifting destroyable space rocks

Task: Alongside that enemy UFO that's moving erratically from point to point, a drifting space rock which drifts in a straight line (wrapping at screen edges) until it gets destroyed can add some variety and dangerous decoration to the play space. Make these space rocks larger than the player ship or the UFO, perhaps about twice as wide and twice as tall.

Discussion: One of the landmark games that came to define this family of arcade games, Atari's *Asteroids*, included drifting space rocks as the main challenge, with the occasional UFO serving to mix things up. I'm suggesting a different balance here: the gameplay will be mainly about the UFOs, with the drifting space rocks here to mix things up.

Rocks in that famous classic broke up into smaller bits each time a large one got destroyed, adding more and faster hazards to the environment. That's not the focus of this game. I'll revisit a variation of that though as a later exercise to ensure that it's something that you know how to do. By having the rocks in there they're serving their purpose alongside the UFO as environmental dangers to keep the player moving.

To avoid the sizable complication of trying to get the UFO to reliably dodge the boulders let the UFO pass through the rocks by not checking for collision between them. Or, if that sounds enjoyable to you to figure out, it's something that with time and experimentation you can definitely figure out. (It helps that the UFO moves in such a steady and controlled way!)



space battle exercise b enemies that can shoot

Task: Speaking of making the UFO the focal point of the action, so far the only way it harms the player is by winning in a collision if the player accidentally wrecks into it. For this exercise give the UFO an ability to shoot that's comparable to the player ship's range and projectile speed. Whether you want to limit the enemy shots to one per ship by using a reusable instance or give them rapid fire capability by using an array of shots is entirely up to you.

Discussion: One aspect not addressed in the task description is how or where the enemy ought to aim their shots. The atan2 trigonometry function covered in RTS Game Step 4, used there to get soldiers to walk toward their instructed goal point, can also be used here to find the angle needed for the shot to be fired from the UFO toward the player ship. Or, as covered right after the math detour in that step as the simpler way, you could normalize the x and y offsets then by dividing them by the distance from UFO to player, then multiply that by the shot speed to get the shot's x velocity and y velocity.

When those two methods were proposed for getting the units to walk in RTS Game, the two options were, practically speaking, equivalent. Here that isn't quite true because the shot accuracy (or more importantly, its inaccuracy) is an important value to tune for the gameplay experience.

The UFO probably shouldn't fire directly at the player ship each time it shoots, or the player won't stand a chance. The



most natural way to incorporate inaccuracy, because it's how real inaccuracy works, is to deviate the shot's angle randomly plus or minus some small random angle. In other words, the UFO's could fire within “ $+- 5$ degrees” relative to the line directly to the player. That angular randomization has the desirable and believable effect of making the enemy far more dangerous up close than when farther away.

There's also another way to simulate inaccuracy: Rather than shooting directly at the player, aim for a spot plus or minus some range from that spot. Think of this as “aim for a spot $+- 50$ pixels in any direction from the player's center.” That may seem similar in principle, but in terms of gameplay and visual feedback it's not nearly the same as using a randomized angle offset. When doing things this way, the farther the player is from the UFO the *narrower* the enemy's firing cone will be, whereas the nearer the player gets to the UFO the less accurate the enemy's attacks will be. That won't look right.

The difference isn't quite as dramatic as I'm stating it here, since the player ship's total hit area still varies by distance in a meaningful way, but it's one of those subtle details that a game designer should be able to recognize. This is also why it's helpful for designers to have familiarity with games at a code level. Here the two math approaches seem equivalent, but whereas one's believable and matches player expectation, the other looks sloppy and makes the enemy appear a bit silly.

space battle exercise 7

multiple enemies in array

Task: Continuing in the direction of giving UFOs prominent in the game experience, it's time to support multiple enemies in play at a time. Much as we could put player shots in an array to support an arbitrary number of them, for this exercise the enemy UFOs can be created and updated in an array to play against however many you wish to have in the game at once.

Discussion: Besides replacing the lone UFO instance variable with an array, and iterating over that array's entries to draw and update each, this change will surface some other tangles as well that we've been able to ignore up to this point without much consequence. In particular, this may expose some slight mess with regard to deciding how to deal with having all enemy shots checked against the player, and all player shots checked against all enemies. Should the enemy shots be data within the UFO, with each UFO being responsible for updating its own shot or shots, or should there be a common pool of shots entirely separate from which UFOs created them?

When handling the player ship's shots it seemed sensible to have the ship contain, update, and display its own shot. Now that the game will support more UFOs in play at once, when a UFO gets hit we'll likely want to remove it from the list rather than resetting it. Wouldn't it be weird though if destroying an enemy ship made the shot it had already fired instantly vanish, too? For this reason, I suggest separating the enemy shots into an array separate from the individual enemies. All enemies will then add their shots to the same collection with them all.

space battle exercise 8 fragment rocks

Task: Having the space rocks fragment into new smaller rocks when shot made sense for the original game *Asteroids*. As per its name, it was focused largely on interaction with those rocks. When adding rock breaking here it's important to find a way to do so that it will not fill the play area and take over the main action in this game, too.

Why bother making fragmenting rocks at all, if it's not as good a fit for this game's direction? Besides my wanting to make sure that you know how to do it (and given this is literally a learning exercise designed for practice, that seems legit), the player is likely to expect it in this kind of context. People come into games with expectations and assumptions that were set by other games that they've played or seen. If this feature isn't implemented in any form it may appear to be overlooked, a lazy omission, or a bug when shooting a large rock doesn't do something along the lines of spawning smaller rocks.

Discussion: If you want to do the traditional style of large rocks that divide into medium rocks which divide into smaller rocks, go for it. All I'm suggesting here though will satisfy the expectation for something *like this* to happen while being different enough that the rocks won't accumulate and change the gameplay experience into being mainly about the rocks.

Creating smaller rocks when destroying a bigger one can be thought of as the large rock "shooting" out smaller rock(s) in one or more directions while removing the big rock. Even



though the space rock isn't armed, in the way that the player and enemy ships are, whether shooting out rock fragments or laser shots, either way one object creates another kind at its current location.

This is a very common pattern in game programming. As other examples: When paratroopers or bombs fall from a plane they are essentially “shot from” the plane. When a military factory in real-time strategy games creates a new tank, that too is like being “shot out” of the factory.

To separate the concept from specific games or case particulars the words “spawner” / “spawned” / “spawning” are often used instead of a weapon term like shooting. The ship spawns a missile, the UFO spawns a laser bolt, the large space rock spawns two medium space rocks, the medium space rock spawns two small space rocks. For later reuse this pattern is one which it might be handy to find ways to arrange as a reusable class, rather than copy/pasting or remaking it again each time and every case for when it's needed.

Having made the connection that creating new smaller rocks is really about getting the object to spawn other objects as part of when it gets destroyed. One idea is to have the space rock blast out tiny, temporary space rocks that otherwise work much like the player and UFO shots. These will require their own small rock image. These could check for collision against both the player ship and the enemy UFOs. Just like shots the rocks shards should expire shortly after they're created.

This approach satisfies the player's expectation that hitting a rock will decompose it into smaller rocks, but with a twist that affect how and when the player interacts with them. Now the space rocks have come another way to interact indirectly with the UFOs, breaking them up as the drift by where enemies are located. This could be used to extend the player's practical range, or to defeat multiple nearby enemies very rapidly (ideal if you opted to score based on consecutive hit combos). Conversely this design also increases their risk as a hazard, giving the player reason to be wary about flying near them even if otherwise not directly in its path.

space battle exercise 9

player weapon types

Task: In the previous exercise the space rocks were given the power to fire in multiple directions at once when destroyed. By comparison the actual ship's offensive capabilities seem a bit tame. Design and implement other ways for the player's shots to behave. While testing these different weapon types let the player change between them by pressing the number keys, although after they're working as desired we'll set up a nicer and more automatic way for players to change weapon types.

Discussion: Many of these changes can be made in the player's firing code, or to the code for the shot itself.

Examples of modifications to the player's firing code might be spawning two parallel shots from each wing, three forking out at different angles from the ship's nose, or firing a ring of shots in all directions. Shot differences could simply be numerical, making it possible to support shots that have different lifetimes/ranges and speed values. Or, you could add totally new functionality to shots to support homing missiles, wave beams, or exploding shells that send normal player shots out in all directions. Enemy ships could be frozen, pushed, magnetized, or switched to fighting for the player's side.

How you indicate in code which shot type is which and how their behavior varies is up to you. A simple number assigned to each kind, like we used for track tile types, will work fine.

space battle exercise 10 vanishing weapon item drops

Task: Instead of switching weapons by pressing number keys, have enemies sometimes spawn weapon pickups when they get destroyed. If the weapon pickup isn't collected quickly enough have it flash then vanish. If the weapon does get touched by the player switch to a weapon type suggested by a letter or two on the weapon's item art.

Discussion: When designing an action game we often think mainly about the dangers, or repellers in the environment, that the player is compelled to avoid. Equally important though is giving the player something to desire in the play space, to serve as an attractor. This escalates excitement and tension during play by giving people meaningful tradeoffs, rushing into risks not because they feel like being challenged but because near that danger is something the player has practical use for.

For decades, games accomplished that by having enemies semi-randomly drop items when defeated. When an item spawns that will vanish, the player is pushed to either charge dangerously into where other enemies may still be, or else to become skilled enough to remove the nearby enemies quickly.



space battle exercise II difficulty waves progression

Task: We've set up enemies in an array, and hinted at the suggestion that we ought to remove defeated enemies from the list. We haven't yet dealt with what to do when that array gets emptied and there are no enemies left for the player.

When all the tiles in Brick Breaker got cleared we promptly provided the player with more to do, first refreshing the layout but later loading in a different level. In lieu of being able to load different spatial layouts, the equivalent to stages for this game will be changing the number of enemies ships, number of space rocks, and some tuning values connected to each. Set the game up so after the initial set of enemies gets defeated the game progresses on to Wave 2, and so on, in which each wave gets harder and more intense than the earlier ones.

Discussion: First, get the enemy ships removed from their array when defeated, similar to the RTS Game's function `removeDeadUnitsFromList()` in the `UnitTeam.js` file. When the last enemy gets removed, increase a global counter for which wave the player is on. Display the wave number on the screen, or at least in the debug output. Thereafter, what you do with that wave number is simply a subjective design matter.

Which gameplay variables will you decide to alter based on which wave the player has reached?

space battle exercise 12

local multiplayer co-op/vs support

Task: Space Battle can have two players sharing the keyboard just as we did for Racing. Create a true/false variable in code that when set to true sets up a second playable ship steered with WASD rather than the arrows. Choice of firing button for that second player is up to you. Make it possible to play either versus mode, in which the only ships present are the player ones and they can destroy one another, or in co-op mode, with the two players versus all the enemy UFOs.

Discussion: Dropping in a second player ship seems easy for co-op, but remember that a bunch of additional collision handling will be needed for not only its new shots but also all enemy shots and UFOs against it. Whether you want to keep the drifting space rocks that fragment in for versus mode is a judgment call, but I like that it would encourage players to move around and make for something to aim at besides directly at one another. In co-op the ability for players to pick up different weapon drops may create enjoyable mixtures, like one ship having what amounts to a temporary ring shield while the other sits nearby firing long-range projectiles.

space battle exercise 13 generated twinkling star field

Task: Add stars to the background and get them twinkling.

Discussion: A solid black background looks unfinished on a modern machine. A still image could be used, however the game will look and feel much more complete if there are stars in the background. Given we're out in space there are not many options for what we'd might see happening nearby in the background. Stars should be there though, and we can make stars twinkle. Fortunately that's a straightforward sort of thing to generate and update.

Randomly generate a bunch of positions within the canvas area. Save that information into one or more arrays. Each draw frame after erasing the previous frame draw all of those dots with some independently flickering shades of gray.

space battle exercise 14

particle effects explosions

Task: Spawn a bunch of flashy but harmless flame and smoke whenever projectiles destroy rocks, player ships, or UFOs.

Discussion: People sometimes get worried about particle effects as being some complex thing. Aside from trying to optimize it, which isn't what we're here to focus on, it's actually not very different from things you've already done.

All those rock bullets spraying from the space rock when destroyed are a lot like particles, except that they check for collisions and therefore can do harm. Those shots even expire, marking themselves for removal from the array then getting cleaned up when needed (you are doing that, right?).

The main differences between how the shot array is handled, as opposed to an array meant for particles, is that:

1. Particles one should not perform any collision checking
2. Particles one should have no consequences on gameplay
3. Particles should support a more dynamic variation, such as randomization in rotational speed, speed decay, and possibly slight variations in scale
4. Particles may change in appearance over time (ex. fading)

Particle effects are also a prime subject for optimizations. More efficient ways to process and display them can keep the game running smooth even with hundreds or thousands of particles in use. Because there are often so many of them,



even a fairly subtle improvement or issue in how much time they need for each one's processing is amplified hundreds of times or more per frame.

One popular approach to improve particle performance is a pattern known as object pooling. In object pooling, instead of creating and destroying each particle when it's needed or no longer needed, you'd instead have a large number of them available at all times, even though at any given time most might be inactive and hidden. When 'new' particles are needed, an unused one from the pool would just be recycled. The same particles get repurposed again and again, saving the computer from the overhead involved in creating or destroying objects in memory.

space battle exercise 15 enemies aim considering wrap

Task: The player can easily see where a shot will wrap across the screen edge, or adapt iteratively to line up shots after the first attempts miss, and the UFOs on the other side won't do a thing about it. They'll actually fire in the wrong direction. For this exercise modify how the UFOs handle their targeting and aiming so that whenever it makes more sense to attack in a way wrapped over the screen edge it will take that shot.

Discussion: Once you get this working the previously smart strategy of hanging out near the screen edges will promptly become a disastrous plan. The computer opponents will go from having not been able to consider the wrap angles at all to instead considering them with as much ease and precision as shots anywhere else in-bounds. These are made even more dangerous because the player will usually be focused on wherever their ship is flying, making it tough to notice an enemy firing from the other side of the canvas.

If you find that giving the computer this ability makes it unreasonably hard, consider diluting their accuracy or frequency of fire when they're near edges. Bring them closer to the human player's level of inconsistency when trying to land those trick shots. Giving the computer a contrived disadvantage like this may feel like cheating somehow. The truth is that the computer already enjoys a massive advantage. It can see everywhere at once, and calculate angles perfectly. More effective AI, especially if it seems unrealistically talented, often isn't as fun as AI the player might win against.

space battle exercise 1b save and clear local high score

Task: Given all that this game has going on in weapon types, scoring complexity, artificial intelligence and difficulty waves, this seems like a game in which we might actually care to save and then later try to outdo a best high score. There's no need to save all high scores for worldwide leaderboards, although with php and a simple database that could be setup.

Currently, the score is lost every time the game is opened. Save the player's own local high scores in a way that allows the value to persist between game sessions. On the title screen given the player a clearly worded instruction for how they can clear that best score if they wish to start again fresh.

Discussion: There are different ways to accomplish this purpose, but one reliable method to look into is the LocalStorage feature of HTML5. When testing whether you can save a value and reload it in a persistent way between game sessions there's no reason to have that number connected to score yet. Get it working in a way that you can quickly test, such as having keys affect the number in question, and the saved number being displayed on the title screen. Once it's working for those arbitrary values switch it over to being used in the same way for the high score.

space battle exercise 17

player ship selection

Task: It actually takes surprisingly little technical effort to make a few different ship types for the player: For each one make a different image, vary its performance tuning variables, and maybe change its starting weapon type.

In addition to the default craft also create a nimble scout ship, a heavy attack ship, and a stealth fighter.

Discussion: Obviously turn rate, acceleration and deceleration are the main ways ships can 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.

How to determine which ship the player uses is a design choice totally up to you. Perhaps every two waves 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 ship by clicking on a different ship icon from 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 unlocked, making them a reward for improved play that can extend the game's replay value.

Because the high scores are programmed to be remembered, as long as the ship unlock is connected to that number the game should also remember and restore the ships earned.



If you do decide to implement the ships in a way that will be locked by progress, it will be useful for testing purposes to add cheat buttons to instantly unlock or switch the player's ship by affecting score or level number. There's no sense in burning time trying to reach a certain part of the game, only to spot that something small needs to be adjusted in the code, followed by replaying that time to verify that fix.

space battle exercise 18 bot player for co-op and vs play

Task: Having already built in the necessary support for multiple players on the human's team, and/or player ship versus player ship, it would be nice to be able to play that way even when no other humans are available to play. Program an option to have the computer control the second player ship.

Discussion: There are a lot of detailed AI challenges involved in getting this working well, but getting it to work at all should not be very complicated. At the most basic form it can amount to simulating the events of pressing player 2's keys at semi-random intervals. For it to become more sophisticated you'll need to program it to periodically reassess what it should be doing (ex. aiming at an enemy, firing, dodging or shooting to deal with a space rock, wandering for something to do), then coding each of those behaviors.

While playing in single player, pay close attention to what you look for in the game that prompts you to change what you're doing. Being fired at by an enemy probably prompts retreat or counterattack. Being within a certain somewhat larger range may be all that needs to happen for you to turn and go after a UFO. If a rock collision seems likely from its proximity, speed, and heading, you likely either get out of its way or turn to shoot it depending on how nearly you're facing it or already perpendicular to its incoming path. Find ways to program systematic object searches, prioritize objectives, and commit for some short interval to a certain course of action.

GAME 6: RTS ROBOT COMMANDER COMPLETION EXERCISES

In the first RTS Game section I mentioned that I chose this genre because it brings up different topics than the other games. This is every bit as true for the completion exercises as it was for the foundation steps. You'll be trying and getting practice with new parts of game development.

The technical and design concepts that you'll get experience with in the exercises that follow are applicable to many games and projects outside the genre, too.

As always, there are two things that you're building when working on this or any game project: the game itself, and your set of skills. The former is more tangible, easier to share and show, but its greater value is as proof that you passed a self-guided test to hone your abilities as a game developer. Those skills and abilities can last a lifetime.

You'll deal with drawing lines, displaying health bars, layering grids, checking for pixel color, calculating area of effect splash damage, detecting right click, interface buttons, and if you choose to take on the exercise for it, even basic pathfinding.

Congratulations on all of your progress and hard that have made it possible for you to get this far. Good luck with the final stretch. Finish strong!



r.t.s. game exercise I clear select on right click

Task: Whenever the right mouse button gets clicked deselect all units.

Discussion: Selecting an empty area to deselect all units is a clumsy workaround. As the person working on the game you may've gotten used to doing that while testing the features you're working on, but that's adapted comfort is dangerous because your players probably won't have the same level of patience. What you're working on you mentally forgive for being incomplete and at times slightly broken, but by the time your game gets released people expect the game to feel and look finished.

For cross platform support we typically try to make 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 games that originally required two-button mice, but it's good to ensure there are still ways to play the game if someone doesn't have two mouse buttons. As implemented, the game achieves that. That said there's no reason we can't also support the second button for the many players who do have it. PCs still vastly outnumber Macs.

(If you are using a Mac laptop and do not have easy access to a right mouse button, the right click functionality can be tested on your machine by holding the Control key when clicking.)





r.t.s. game exercise 2

draw targeting line

Task: Draw a colored line connecting each Unit to whichever Unit, if any, that it's targeting. Use line color to distinguish which team the Unit is doing the targeting.

Discussion: Currently it's tough to tell exactly which Unit any other Unit is targeting. They get near one another, stop, then units start disappearing – but who in the crowd is responsible? As we begin adding some automatic defensive action for the units on both armies it will be very helpful to visualize this information as we work on the game, even if we don't wind up leaving it in the final version.

When two units are targeting one another one's line might overlap the other. A slight offset for each team line's start and endpoint based on which team is doing the targeting will improve their visibility.





r.t.s. game exercise 3

find and edit sounds

Task: Find sound effects that you are certain you have the appropriate rights to use and modify for your game. Edit the sounds for use, and connect them to the gameplay events.

Discussion: Finding a good source or two for sound effects that you're sure you have the rights to use will be beneficial later for many future projects. Sometimes a sound can be generated electronically by parameters, sometimes a sound can be made with your mouth or hitting together things in your home, but often a source sound is needed that's quite different from what either of those will sound like.

Types of editing might include trimming silence off either side (careful to not clip too much off the trailing falloff at the end, which often goes longer than the waveform seems to show!), adjusting for relative volume, cropping out one piece from a longer sample, or mixing together multiple sounds to produce a desired effect.

If this feels like cheating somehow, to be finding recorded sounds to work with and from rather than generating and artificially remaking them all, know that preparing and mixing recorded sounds is a perfectly real and common part of the job that audio professionals do, too. Sometimes nothing that the computer generates or that can be made from available materials is going to match the sound needed, or sometimes for practical reasons a new sound can't be recorded from the actual source in suitable conditions.



r.t.s. game exercise 4

unit and background graphics

Task: Dots fighting dots was fine when getting core gameplay working, but as we move toward advanced features we'll soon want differentiated Unit types. Create some small robots for each army, as well as an image for the ground beneath the units. Also create versions of defeated units, small rubble piles, and update the code so that they'll appear and remain wherever units get removed from play.

Make the background image first, before units, since this way when drawing units you can experiment to determine what stands out well enough against the background of choice.

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 using a rectangle to overwrite the past frame. If necessary, change the coloration and/or brightness of the on-screen help and interface text to keep it legible.

Discussion: If you're not feeling up to creating the art for these, I've once again included some quick examples alongside the starter code for these exercises. The Heavy Unit graphic is for another Unit type covered in a later exercise. The "-off" version of the images is their rubble versions, which I kept the same overall dimensions to ensure that the piles line up automatically with where the robot's feet were. Otherwise if the rubble pile were cropped tightly it would appear to jump to the robot's waist upon appearing, centering on the Unit origin.



Most images authored for the games have used transparency, parts of the image where we see through to the background. For those images we used PNG format because it supports transparency. However the background image won't need to support transparency. The background is also far larger in dimensions than the Unit sprites. Between not needing to support transparency, and wanting to keep the file size down for a larger image, JPG is a good format for the background image.

JPG is especially well-suited, too, if using a texture like grass or dirt with many subtle variations. If you were using an art style that made the ground mostly large blocks of solid color, a format like PNG would actually be smaller and sharper.

A few things to keep in mind while drawing the ground terrain: First, at this time in the development, we don't have support for obstacle collision. 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, but for now 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, so don't spend too much time making something that you'll grow attached to.

I'm 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 usable placeholder) is as follows:

1. Create a new image the same dimensions as the playfield.
2. Fill the whole image with white using the paint bucket.
3. Use a noise filter to cover the image in many gray dots.
4. Tint the image dark green for a grassy look.
5. Decrease contrast a bit (to combine this with a little extra color control make a solid color layer above the background, and adjust its transparency slider).

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. HTML5/JS cannot directly open Photoshop or GIMP.org's native layer formats.

Make units very small, since there are many of them and they get treated as points for many collision purposes. 7 x 14 pixels will do, although you'll need to figure out how to zoom in with your image editing software while drawing at that resolution.



Blue here in place of transparency so it's visible on page.

There are techniques to tint image colors in code, to give each army different colored infantry with this same common image. Otherwise you could just tint it two different colors in your image editor to save as different files, though later if redoing the visual design you'd have to do two exports instead of one.

No animations are needed at this point, and with the units so small this is fairly forgivable. That said, if you'd like to have them toggle between a few animation frames when moving, if you've worked that out already for Warrior Legend this would be a good opportunity to adapt that code or practice working on an improved solution.



r.t.s. game exercise 5

unit health bars

Task: Give each Unit four health points. Display it as a four box bar above each Unit, where four white squares means at full health and one white square in a black rectangle means at lowest health.

Discussion: These will momentarily clutter up the battlefield considerably, but other improvements coming down the line will resolve that by only showing them for units in battle or currently selected.

Update the attack code so that units in a battle lose health units one at a time rather than being destroyed instantly. To make the battle outcomes less predictable randomize either the amount of damage per shot between 1-2 or give units some percent chance of missing when they attack.



r.t.s. game exercise b change health color for low or max

Task: 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.

Discussion: If a unit's health equals its starting value, draw it green. If it's above half the starting health value, draw it yellow. Otherwise draw it red.

Avoid hard coding the health value cutoffs from the current values, like 2 or 4. If the amount of health per unit gets adjusted later ideally this code should automatically still work correctly.

An approach that isn't based on hard coded values is also better for later supporting different types of units that start with other amounts of health. Having more than two health points remaining qualifies as medium level health for an infantry robot as it is for now, but for a giant heavy robot that value is probably much less than its half health value.





r.t.s. game exercise 7 bounded ai thinking counter

Task: Make a frame countdown variable on each Unit that gets set to a random range with a minimum and maximum each time a new action gets taken. Decrease that every frame, and rethink the unit's orders when that number reaches zero. Use this in place of the former approach which updating the unit's thinking whenever a random number happened to be below some percentage.

Discussion: Having AI update based on when a random value is below a certain threshold is a wonky way to do it. At times a Unit may go awkwardly long without updating, and other times it may seem terribly indecisive as it keeps changing its mind several times moment after moment.

This improved approach gives you full control over how far in either direction you think still looks and feels reasonable.

I should acknowledge here that using variables for cycle-based countdowns isn't really the preferred way to have something happen after a certain amount of time passes. An event system that will call a function after a certain amount of time can be a tad more efficient, a lot more elegant since it avoids distributing counter logic, and avoids frame rate dependence. That disclaimer aside, for projects at the scale covered it still certainly gets the job done. I've opted to show and explain this technique here because it works in virtually any programming language using only common elements, and it doesn't involve any complex framework under the hood.



r.t.s. game exercise 8

show health only on some units

Task: Right now all health bars show for all units all the time. This is far more information than the player can use at once. We've also been doing enough game programming by now to begin thinking ahead a little – when adding graphics, abstract bars over the action all the time won't look very good.

Instead of always showing health bars, only show bars for units that are currently selected or are within some range (for starters: perhaps 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 or battling allied units to rapidly size up their local crowd to spot Unit clusters with lower health.

Discussion: Drawing the health bars for selected units is as easy as calling the function to display health only on the elements in the array of currently selected units.

As for the distance check, where the Unit health bar function gets called wrap it in a condition that checks on whether the distance from the mouse to the Unit is within some range.

If the mouse is within range of the Unit, and it's selected, its health bar will be rendered twice. That's not really a big deal in the grand scheme of things, and it's not likely going to be the source of a performance hit. There are several ways to work around this, if you'd like to do so as an exercise. For example, you can skip showing the health bar for any units during the



selected units loop if it's found to be within range of the mouse. It's really not a big deal at this point though.

Alternatively, enemy health might only be visible when it has been recently attacked, instead of ever showing when near the mouse. That would increase challenge due to information hiding, since the player can only get updated information about the enemy by launching an attack.



r.t.s. game exercise 9

draw lasers that show missing

Task: Draw laser blaster lines that show which robots are actively attacking one another.

Discussion: This may sound redundant with the targeting lines, but besides only showing when in attack range these are a visual decoration we may wish to keep for release.

To make the lines more like laser beams:

1. Change the color of the laser to match the team firing it.
2. Randomly flicker the line by skipping its draw code 70% of the time, so it seems rapid fire. This also prevents a laser drawn later in the same frame from always being drawn over.
3. To further distinguish the robot that's firing from the side being fired at, randomly offset the target end of the line by up to 6 pixels in either direction along either axis. This way the source of the laser will be the point end of a cone that scatters on the side being attacked.



r.t.s. game exercise 10

heavy unit type

Task: To support a wide variety of units, you'd want to make a flexible system for updating the tuning of each type – such as an external JSON file that describes each Unit by its tuning. Well before trying to design and build that, find a way to hack in support for an extra Unit type. That can help surface which tuning values will be meaningful to have in an external file, while also giving a clear sense for ways that adding in new Unit types can affect the game's depth and strategy.

Discussion: 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. Indicate 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, with blue added to make its white form easier to see against the white page:





Make a destroyed version of the unit's graphic, too.

As a rough way to get this working make a true/false value in the Unit class for whether the Unit is of the heavy type. If this is set to true, handle this as our second Unit type in terms of its presentation and behaviors. To support many different kinds of units we might instead wish to use a number to keep track of which Unit type this is, to then act as an index into an array filled with tuning values loaded from external files, but one thing at a time.

Set up the unit's speed, attack range, and start health to vary. This will mean instead of having these as constants make them outputs of a function based on the Unit type, or variables that get set on the Unit when it's created as soon as its type is known. Alternatively, define new constants unique for the Heavy Unit to use instead. I suggest having it move half as fast as the infantry, able to attack at four times their range, and having four times as many hit points.

When and how should that heavy true/false type be set? As a testing shortcut for now randomly assign some percentage (perhaps ~10%) of units to be this special type. A low percentage keeps the battle largely focused on the default infantry but ensures there will be 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. Work out a health bar loop that scales box sizes based on the unit's maximum health.



r.t.s. game exercise II

color key collision affecting mobility

Task: Set up regions of the map to cause units that navigate through them to move faster or slower. This might map to things like roads or muddy terrain. The same concept can also be used to affect Unit visual range, attack range, or even add a shielding/dodge cover factor if for example standing by trees.

Discussion: A grid-based collision and level rendering system, like the one in the Racing and Warrior Legend projects, has several advantages. However the major drawback of the tile method is that the levels will look chunky and inorganic.

We could treat this as a rationale for placing 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. Many fun and successful games did that. Back then tile-based levels were not only a convenient way to make large landscapes, but were often the only way for games on old hardware to handle storing, showing, and colliding such 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.

To do this create two different versions the background. One should be a JPG, illustrating visual details, to show the player. The other image, of the same size, should be a PNG (to avoid



any minor color inaccuracies due to JPG's lossy compression artifacts) and it won't be shown to the player. The PNG will contain blobs of exact and specific color values corresponding to different terrain regions of the JPG. Different RGB colors can signify high grass, mud, shallow water, etc. To find what terrain type a Unit is standing in you can check and match against the color value in the PNG for its current position on the battlefield.

Depending on where the Unit is standing its attack range, movement speed, or other parameters may be affected. When coming up with ways to have the terrain type affect Unit mobility, consider how the changes could affect the infantry robots differently than the Heavy Unit. The Heavy Unit, for example, could effortlessly traverse mud that slows infantry, but the same Heavy Unit might be unable to cross water at all.

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 pathing intelligently around those barriers to follow their orders. Otherwise, telling the Unit to get to the other side of a pong and seeing it get stuck on the wrong side feels pretty maddening. A*, pronounced A-Star, is one very popular approach to pathfinding problems. It will be brought up again in a later exercise.



r.t.s. game exercise 12

fog of war grid or per-pixel

Task: Implement a “fog of war” system that conceals areas of the battlefield where the player has not yet reached. The area that the player’s units can’t see can be blocked by either black tiles (possibly with smooth edging tiles inserted all along the border to avoid blocky and straight boundaries) or by a black image that has individual pixel areas set transparent.

Discussion: There are two different ways for fog of war to work, and which you use is, as always, up to you.

Both begin with everything outside of player Unit sight range as black, revealing map information and enemy units as the player moves. One approach lets the player continue seeing action anyplace that has been uncovered since the round started. The other approach obscures the areas left behind, either completely or showing the terrain but not enemy units there. In the first way, walking everywhere in the map would thereafter be the same as not having fog of war implemented. In the second way, no matter how much you move you will still only see enemy units that are currently in sight range.

Given every Unit has sight range of its own to calculate, and most frames the fog of war isn’t pushed back by most units, updating the fog of war is often something that can be done somewhat infrequently, and only for units in motion.

A side effect of implementing fog of war is that you can make sight range another differentiator between units, for example making an otherwise weak Unit a valuable scout for recon.



r.t.s. game exercise 13

scrolling camera with minimap

Task: The battle is currently all limited to a single screen. Traditionally RTS games take place on a map large enough that the player needs to send out scouts to explore and figure out where opponents are located. While you don't need a massive area for this gameplay demo, it'll still be good practice with camera scrolling to extend the action beyond the size of one screen region.

To keep all that space manageable also implement a minimap that shows every Unit position compressed into a small part of the screen. Show a rectangle on it that corresponds to which part of the whole world is currently shown on the main view.

Discussion: Camera scrolling has been discussed elsewhere, for example with the racing game. It can be done with only two variables, one for the camera's offset horizontally and the other for its offset vertically. Those can then be subtracted from all object draw positions or, preferably, handled as an extra translation call prior to drawing all objects each frame.

This time though instead of computing the camera offsets based on a main character, the player should be able to pan around the map by hovering the mouse near the edges.

Alternatively or additionally, holding the keyboard arrow keys could move the screen view relative to the map. For all mouse functionality ensure that the camera's offset values are factored into the mouse's coordinates on the map, since the mouse itself knows nothing about the scrolling view.



Experiment with some test sizes before committing to drawing a ton of detail on your map or map layers. You may find while experimenting that your initially planned land mass is more ambitious than your processor or browser is comfortable dealing with.

As for the minimap, after all objects get drawn and the camera translation gets undone for interface elements draw a small black rectangle that is proportional to the world map.

Or, instead of a black rectangle, if you'd like to have terrain details on your radar you can scale down your terrain image in your image editor and darken it a bit to use as the radar view, in case you want terrain landmarks on the minimap. Those changes could even be done in code to ensure the minimap stays in sync with the full size map, and you can cache the outcome if you don't want to recalculate it every frame.

Next 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 voilà, 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 and size of the screen.



r.t.s. game exercise 14

projectile attack splash damage

Task: 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 made an artillery-style Unit. The only missing element to complete that is support for area of effect damage. That means that rather than only damaging the specific Unit targeted, any units within some radius of the point attacked take damage, potentially as a function of their distance from the location of the blast.

Discussion: The code for handling splash damage isn't bad. While a brute force solution of checking every possible Unit isn't very elegant, for something that happens occasionally and entirely within one frame that's a fine way to do it. Check each Unit to determine whether it's alive, on the other team (unless you'd like to support friendly fire for splash damage), and its distance to the attack point.

The hard part of this problem is how to visualize what it does. Particle effects can be a good fit for this, similar to the ones described for Space Battles. Additionally, the shot probably shouldn't have an instantaneous effect since it's long-range; war games tend to depict artillery attacks as slow, lobbed projectiles, visible while sailing through the air. That delay between the moment the shot is fired and when it bursts on the ground can help give it more dramatic impact.

To exist independently in space and time from whichever Unit fired the projectile, the shell or missile will need a class of its



own to track its state. On the plus side, that same class can be adapted and used for rockets, arrows, plasma balls, laser bolts, thrown grenades, and any other slow, visible, ranged attack. The only difference between those types of attacks comes down to different splash damage properties, projectile speeds, and visual elements.

Any Unit that fires projectiles can then simply add a new instance of that projectile class into a projectile list. All elements on that list would be updated and displayed either as part of that unit's update and draw or as a separate cycle which handles all projectiles regardless of the source. The latter approach, keeping the projectile list and its code outside of any particular Unit, has the advantage that when a Unit which fired it gets destroyed the projectile can continue to finish its trajectory. Otherwise the shot would most likely wind up destroyed when the Unit that fired it got wiped out.



r.t.s. game exercise 15 unit draw order from back to front

Task: Half the time, when two units stand close enough for their images to overlap, the one which should be drawn behind the other may instead be drawn on top. This happens because unit draw order is due entirely to the arbitrary order in which they were put in the array. The problem looks like this:



Whenever a unit's y value positions it higher on the screen than something else, we want that one drawn first, since the one drawn later will overlap it. In computer graphics that's called the Painter's Algorithm, since to handle overlapping objects in a view painters draw background elements before dealing with the foreground elements that overlap them. Even if the issue isn't especially glaring in this game due to the small graphics and solid color art it's worth knowing how to do. Besides, as long as your game handles draw order wrong you're going to be looking for it, and it's going to bug you.



Discussion: 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 basically hypothetical. It doesn't really come up, or when it does it doesn't happen long enough to be perceptible. What does happen, without this fix, quite frequently because defeated units are stationary, is this:



Since rubble piles are short and low to the ground it's rare for rubble piles to get in each other's way. It works well enough to first draw all the scrap heaps for dead robots each frame before doing a second pass to draw all the active units.

This situation is another reason why it can be beneficial to keep separate lists for active and inactive units, since it avoids



needing to scan through all units twice to achieve this draw order. That's not essential given the number of units involved, but it's simple enough to do that it's worth considering.

The more robust solutions ensure that you're actually drawing the units in order based on their y position, far to near. One way to do is to create a list for all objects to draw. The list could be cleared at the start of each frame, getting objects inserted into it during their logic updates based on their y position on screen. Or you could just fill that list and then sort it based on the object y positions. Since most frames most objects don't overlap one another, you may be able to get away with only selectively updating the list order for objects near one another.

As one complication: the center of the image has been the origin for bounding boxes and such, but for this sorted draw the character's feet should be the sort point. Otherwise tall characters could wind up sorted out of order in the proximity of shorter ones.

One upside to finding a thorough and proper solution to this issue 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 decorative, neutral Unit entities such as trees or abandoned buildings that sit still, get destroyed by collateral damage, or otherwise add a sense of setting to the battlefield. Scattering those around a region on the map that has a navigation key color to work as wooded territory could give a lot more depth to the scene, while also making the world feel dynamic due to destructible elements.



r.t.s. game exercise 1b player-like computer player

Task: The computer is kind of cheating by doing things the player cannot do. It can pay equal attention to all zones of the battlefield simultaneously, units can be rapidly selected and commanded independently for maximum effectiveness, and it's generally exploiting its full access to all battlefield data including its ability to interface with each Unit separately. In the interest of fairness, modify the computer team so that its constraints and controls work more like the player's.

Discussion: This is a weirdly challenging artificial intelligence problem, since the things that are intuitive for a human can be quite difficult to code systematically in a way for a computer to reproduce. We can spot a plan of attack being set up and think “Oh no,” or look at a cluster of units and quickly assess a relative weak point for the focus of an attack, but those are tough to calculate reliably.

Accepting the difficulty of getting this done in its ideal form, there are still some easier ways to approximate the human's limitations in the interest of fairness. Rather than dealing with all units as individuals, the computer team can divide troops into subgroupings of 4-25 and order those around as a whole. This is closer to how I play, although if you find a different range seems more fitting, base it on your activity pattern.

Right now each computer Unit has its own separate timer for when to next rethink its own commands, but this means subgroups within a giant army will receive orders as rapidly as



when there are few units on the field. A better abstraction would be a team-wide timer for when to issue a new command, and at the end of each interval the computer would only issue new orders to one or two of the subgroups regardless of how many subgroups there are in play. This better simulates the limitations of human attention.

A possibly fun but pretty involved and potentially futile piece of this puzzle would be giving the computer player only a single screen full of information that it can access or direct at a time. Instead of managing the whole army, it would keep a list of which units are currently on its screen, and periodically sweep its view around the map based on last-known group locations to issue orders in different areas. As a bit of a parlor trick the screen could even be shown to the player as a reassurance that the computer is playing fair.

At the end of the day, letting the computer cheat a little can be far more computationally efficient and can achieve practically the same results. Where is the line be drawn? Does the computer player also need a mouse-like cursor position, for selecting units only via hastily formed lasso rectangles? Should it have to infer Unit position from a visual assessment of the image data?

Focus energy and effort on what will most affect the player experience. Going too far into this problem space may be a case of focusing more on the fun you're having as a developer instead of the fun you're providing for the player. (If you find making the AI highly enjoyable, perhaps you can distill parts of that fun into what the player can do!)



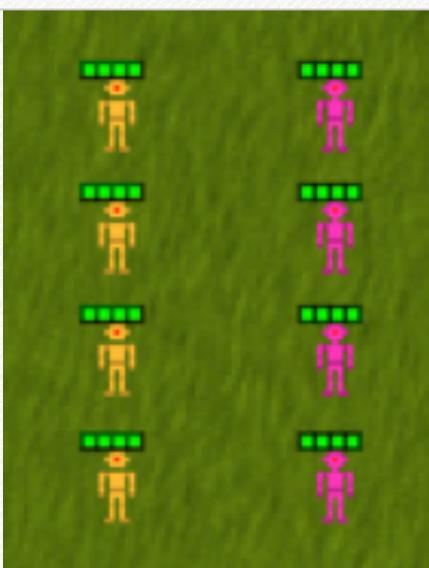
r.t.s. game exercise 17

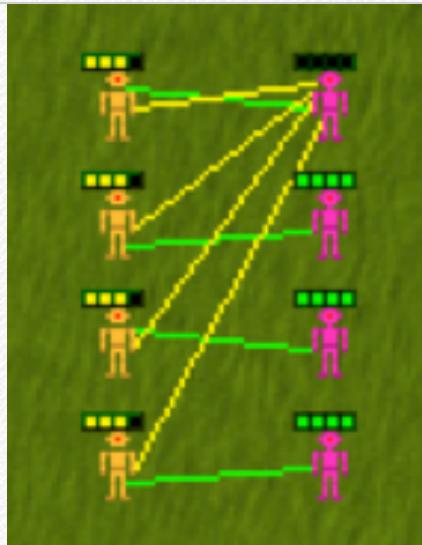
strategic computer army behaviors

Task: Give the computer player some strategy. Any strategy. Anything that looks and plays better than random will do.

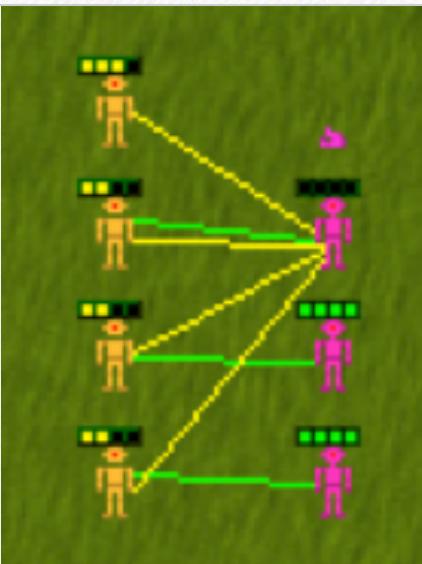
Discussion: Before the computer can be given strategies, you'll need to figure some out to know what you're trying to teach it to do. I'm not a great RTS player, but can illustrate a couple of pretty simple ones here as a starting point.

One basic strategies for classic real-time strategy games is based on the fact that a defeated Unit can't return fire but a damaged one can still deal full damage. 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?

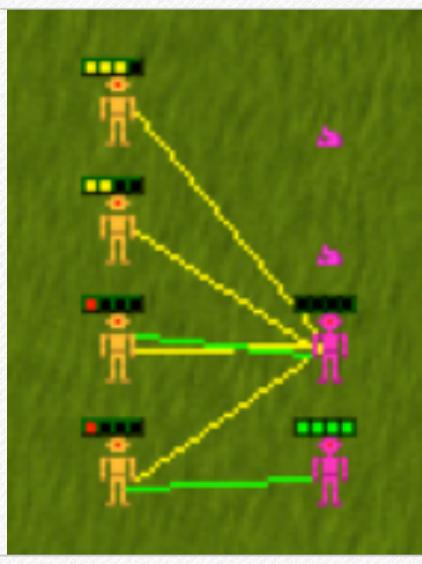
	<p>At the start of our example scenario, 4 units all with full health from each side are meeting in the battlefield. The orange team on the left is going to concentrate its fire on one target a time, whereas the purple robots on the right will each pick a different target.</p>
---	---



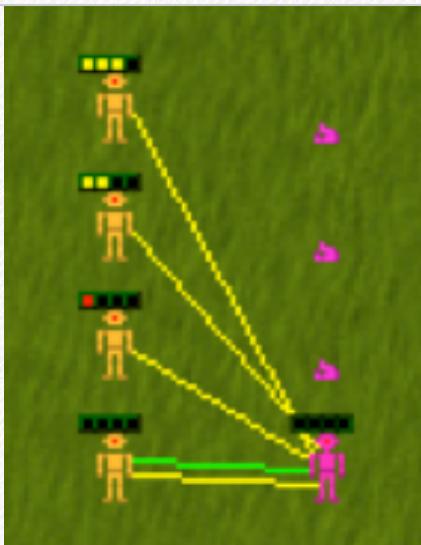
In the first round each Unit on the orange side loses 1 health, meanwhile one robot from the purple team is immediately eliminated.



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.



In round three, another purple enemy is toast. Now only two orange robots lose another hit point. The advantage for the orange side is probably pretty clear by now.



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.

The above case is admittedly idealized and a bit contrived for illustrating this point. Rarely do all units on both sides fire at exactly the same time, never miss, and line up so they're all nicely in one another's attack range. The number of units involved, how much damage each does when attacking, and how much health each Unit involved has 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 lone target half of those 8 shots would be wasted on an already defeated target that had its 4 health accounted for.



There it would be advantageous to break the groups into two sets of 4, each focusing firing on a separate target.

This type of focused attacking behavior and division into coordinated squads actually lines up nicely with the notion from the previous exercise of having the computer player operate subgroups rather than each Unit independently.

Even though the exact details inevitably vary during the chaos as dozens of robots fire lasers at one another, basic principles hold up:

1. Focusing fire with common orders is advantageous over simply moving your units near enemies then leaving each to randomly pick a target for itself.
2. Bringing a Unit with low health into a new firefight means you may lose a point of attack early in the exchange. Hide damaged robots back among the rear or main group where there's a lower probability that they will be targeted, bringing others that have full health forward to start fights.
3. Conversely: target damaged units and groups with damaged units before taking on strong ones, and you'll likely go into that second battle with more working lasers leftover.
4. When possible, send a full squad of four to attack a separated opposing Unit, rather than sending only a couple. 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 sending your crowd into their crowd and hoping for the best.

Notice, by the way, that these strategies outlined so far aren't programmed specifically into the game. At no point did code whether it's a better decision to focus fire, or distribute targets evenly. Nor are there equations in code for how many units should remain after a firefight between a certain number of units versus a certain number of other units, based on tactics used by either side. These strategies emerge from dynamic interplay of the gameplay systems. An RTS developer doesn't need to make deliberate effort to ensure these strategies work – or might not even be aware that these work in their game.

Now that you're going to try to get the computer player to be more effective, you'll need to notice such patterns. You may create some math expressions to evaluate risks or priorities.

Another basic gameplay strategy amounts to leading the enemy into an ambush. This works because characters auto-target and likely chase any enemy that gets in attack range. To exploit this a lone Unit or small group of units gets an enemy group's attention at maximum range then retreats into a bigger crowd of allies in waiting.

As explained earlier, a character walking into a group of enemies can sometimes be defeated before it even has an opportunity to return fire. All the ambushing units have only a



single, distracted target to attack. That leads to focused fire automatically, even without direct orders.

You could program the computer team to try this trick on the player, but maybe even more importantly you ought to find a way to safeguard against it being used on your computer player. Once a player figures out it works this can be done again and again to thin out the whole army until there's so few left that they can be stormed through. A fairly simple test to reduce the risk of this issue is giving units a maximum time or distance for which they'll fixate on an agitator before giving up and returning to their previous defensive position.

It's not necessary to become a competitive master at the game, and nor do you really want your artificial intelligence to be unstoppably optimal because that wouldn't be much fun either. You want your computer player to not look silly or negligent, as though it's careless or not paying attention. If the player gets that impression then the illusion that they're up against a challenging opponent similar to themselves is lost.



r.t.s. game exercise 18 grid obstacles and a* path finding

Task: Build a grid tile map for your level that includes obstacle zones on it, covering over and next to any boulders, pits, cliffs, lakes, shorelines, or other impassable areas. Implement A* ("A Star") pathfinding for your units so that if given an order to move someplace on the other side of an obstacle they can plot a course automatically that will go around it.

Discussion: A* is a tremendously well documented general algorithm for pathfinding, so I won't be trying to explain it in detail here. I will offer that Wikipedia has an excellent entry on it, as well as Dijkstra's algorithm which is a simpler relative of it. There is an excellent visualization of A* here:

<http://qiao.github.io/PathFinding.js/visual/>

You'll often read about these algorithms with relation to a node-based graph of line segments connected as waypoints, rather than grid tiles. The two are equivalent, as the grid is a special case in which the line segment weights and distances are consistent and there's a pattern in adjacent connectivity.

You can think of Dijkstra's as searching all possible paths from the start point, one step at a time in all directions, until one is found that reaches the end point. A* is different in that it applies a heuristic to bias its search in the general direction of the destination instead of checking in all directions at an equal pace. Here's an excellent article on heuristics:

<http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>



Pathfinding can be a difficult thing to get working, despite being a well-understood and well-documented problem. A lot about how it works is either a little abstract or involves so much data that it can be difficult to visualize well when working on it. This makes fixing it when it's broken can be very tricky.

One approach is to first get A* working in a separate program, without worrying about Unit and battlefield details. Then, port and integrate that solution into your game. This way it would be much simpler to visualize your data, stepping through the comparisons, and setting up test cases to ensure it works.

Because pathfinding involves a lot of comparisons and calculations, it's not something that you want every Unit to be checking on every frame that it's moving. It can be calculated entirely when the Unit receives its move instruction, at which time it can translate the A* results into a series of waypoints it should follow, thereafter trusting the waypoints rather than doing any more pathfinding calculations. When units are in a group nearby one another it may even be practical to do the search once for the whole squad and then share it among all of them, or give one waypoints and set others to follow it.

Getting pathfinding will make your units much more intelligent. This will especially be appreciated for the player units, so that they can follow player orders with less micromanagement. This also enables environments with greater gameplay variety.

r.t.s. game exercise 19 resources, build menu, & recyclers

Task: Give the armies a form of currency, starting both sides with a certain amount in their bank. Provide interface buttons that can order new infantry or Heavy units at costs you decide roughly at equal value (i.e. how many infantry does it take to be as valuable than one Heavy Unit?). Introduce a Recycler Unit as well that is like a heavy in its mobility and armor but has no attacks, instead being able to convert rubble piles into cash. Make sure the conversion is below original value, ex. maybe two infantry rubble piles to afford making one new infantry. Also let the player buy extra Recyclers if desired, but make them suitably expensive.

Discussion: This introduces all new types of strategy, from how the initial cash gets spent, to escorting the Recycler into territory where a skirmish recently took place, to targeting the enemy Recycler, to deciding what to build during play. Much of the challenge here won't be on getting this to work for the player, but in getting the computer player to use it sensibly.



r.t.s. game exercise 20 buildings and tech requirements

Task: Add several types of buildings to the build menu. Set up some building types as requirements to be able to produce new standard, Heavy, or harvester-style Units. Include at least one defensive building which has a lot of health and good range but no mobility. Only allow new buildings to be placed adjacent to existing buildings. Fit these onto a grid the same resolution as the pathfinding grid, but making each individual building's footprint at least a few tiles in size.

Discussion: Each time a building gets placed mark its tiles in the A* navigation grid as impassable so that Units will then find their ways around it.

Maybe even more so than buying infantry and using the Recyclers, this building a base business, even in very simple form, isn't something that a computer is very natural at doing. Old games would let the designers secretly hide clues in the map for where they as humans felt buildings of each type should go, and when the computer player had the money to build the next building in their series that's where they'd put it.

In order to avoid a stalemate, if an army loses its last Unit but still has buildings leftover, liquidate those buildings into small squads of standard infantry to give them a fighting chance. Alternatively, count the round as won when all mobile Units have been defeated.

EPILOGUE

Videogame development is a journey of endless technical, creative, and introspective challenges. It brings together technological wizardry and open-ended artistic expression.

Having worked through the exercises and concepts in this book you're now equipped with practical skills to strike out into creating original gameplay. Feel free to build upon your Hands-On projects, or mix elements of two or more of them into a hybrid, then run with it in a direction all your own. If you want to collaborate with other talented artists, designers, and programmers, you now have demonstrable capabilities and playable results to show others what you bring to the team.

One last thing I'll leave you with: challenge yourself. I don't mean that in the usual vague sense of "set your sights high" or "push your limits." I mean it literally. Write short, specific, incremental challenges for yourself when creating your own projects. The most important idea to take away from this book is that your game will be made not through tinkering idly, but instead by seeing how to slice your design into testable steps. Games are created one small feature after another. From here on out it's up to you to author your own series of challenges.

Happy game making!

SECTION BONUS - MINI-EXTRAS

Finished everything? Or: skimmed ahead to find something different to tinker with on a break from the main material?

Check out the “Section Bonus – Mini-Extras” folder contents:

- **grid-scrolling** Demonstrates how to do camera scrolling for a grid too large to show all at once on the screen. Also shows how to block the camera from showing beyond the world edge. Related to the camera exercises for Turbo Racing
- **many-balls-grid** Shows using an array of small, moving objects to handle many elements bouncing against a tile map. (Why? Because it's neat. Also: Look at how many it handles!)
- **sideview-platforming** Added to show how easily you can apply the same kinds of concepts learned in this book to make a side view game. The main differences: gravity, jump, and checking player collision at 4 different points (left, right, top, and bottom, each handles grid overlap a different way!).
- **space-conquerors** Core of a genre-defining arcade classic. I made this out of Brick Breaker's code for easier readability.
- **board-game-demo** Basic mouse-grid interaction for chess
- **racing-tilesheet** Sprite sheet for small files, fast load times

VERSION CHANGE HISTORY

Without specifically listing which or saying so, each update also includes accounting for any typos found or pointed out since the time of the previous update.

DECEMBER 25, 2014 - HOLIDAY EARLY ACCESS, V0

- Initial release. Complete, but still rough in many places.

JAN 5, 2015 - FIRST EDITION, V1

- Added illustrations to opening console log
- Added illustration early on of window coordinate space
- Added preview of Racing Game tile appearances to the end of a Brick Breaker section to help visualize where it's going
- Added syntax highlighting of comments in book's code
- Removed optional, previously not explained argument from eventlistener hookups

FEBRUARY 13, 2015 - SECOND EDITION, V2

- Added/started this Version Change History section
- Note added to intro about syntax highlighting in book
- Refactored Tennis Game paddle AI example solutions to be relative to paddle2CenterY instead of AITarget, to make the offsets and movement directions easier to visualize

- Added new diagrams to a few Tennis Game sections
- Changed all tile coordinate variables to be in column, row instead of labelled x, y (using x, y only for pixel positions)
- Removed some //// update marks that stayed between example code steps
- Corrected “nestled” loops to “nested” loops
- Removed margins from tile grid as used by Brick Breaker to simplify the math, instead showing margins by not putting bricks in the first 3 rows as a later step – this also involved changing the brick dimensions to 80x20 to fit snugly
- Made side brick collision a step in Brick Breaker’s Section 1 instead of in a Section 2 exercise – became Section 1’s Steps 10 and 11, eliminating Exercises 14 and 15 from Brick Breaker Section 2
- Added several illustrations and a more thorough explanation to the grid collision introduction in Brick Breaker
- Changed initial brick collision effect to be brick removal (without ball motion effect) instead of brick flicker
- Unindented some example solution files that were indented by one tab on all lines
- Changed how images are loaded to avoid a race condition. On very fast machines the game startup could trigger multiple times, causing the game’s speed to be too fast due to multiple overdraws. This problem didn’t usually happen,

but it would on certain machines. Now it uses an array of data pairs instead of incrementing a counter during load

- Added “Section Bonus - Mini-Extras” folder with many-balls-grid and sideview-platforming demonstration code
- Additional example step illustrations added to Brick Breaker, Racing, and Warrior Legend sections
- Scanned for many of the longest paragraphs and sentences in the book, whenever possible making those either more brief or splitting them into multiple distinct ideas
- Section 2 Tennis Game tuning discussion expanded
- Added other examples and discussion to the computer controlled AI exercise for Racing in Section 2
- Split up or shortened the largest 30 paragraphs in the book
- Split up or shortened the longest 20 sentences in the book
- Rearranged order of the book’s first few sections that appear before the games do (introduction, concepts covered page, table of contents, etc.)
- Named games for Section 2 instead of just putting “Pt. 2” after the same titles as appeared for Section 1

MARCH 31, 2015 - THIRD EDITION, V3

- Added Video Course Version for Section 1: Tennis Game, including the slightly different example solution code to go with each step as it’s covered in the video
- Added brief explanation of Section Bonus Mini Extras code

- Switched drawCar and initCar to carDraw and carInit for consistency with carMove function's noun-first naming
- Fixed typos and improved wording in many, many places throughout Section 1's games (*hundreds* of little touch ups). Special thanks to Harold Bowman-Trayford of Renegade Applications, LLC for his generous and detailed editing feedback which led to many of those wording adjustments. Check out Harold's game development work at @RenegadeOwner on Twitter!
- Added explanation near the image loading introduction in Racing to how a loading screen can be coded that updates as each new image finishes loading
- Renamed ROUNDSPEED_DECAY_MULT const in Racing to GROUNDSPEED_DECAY_MULT, and SPACESPEED_* for Space Battle once the player control becomes ship-like
- Renamed the keyboard event from e to evt in all locations for consistency with the naming of other event listeners
- Added a missing //// update marker or two in Tennis Steps
- Added Grid Scrolling mini-extra example to the bonus code
- Removed “Wall” part of filename for track_treeWall.png and track_flagWall.png in the Racing project's code and steps
- Fixed usage of incorrect imageLoading array in Space Battle step 1 example solution
- Explanation of why we're using window.onload() to kick off the program's code added to the early Tennis coding steps

APRIL 25, 2015 - FOURTH EDITION, V4

- Improved wording throughout Audio Intermission section, based on feedback from Harold Bowman-Trayford of Renegade Applications, LLC (@RenegadeOwner on Twitter)
- Changed lowercased filenames and lowercased class names in the Audio Intermission to start with capitals
- Replaced dash (-) with em-dash (–) where appropriate
- Added reference to the example grid scrolling bonus code to the related exercise for the Racing game in Section 2
- Split and reworded the 37 longest paragraphs in the book
- Split and reworded the 45 longest sentences in the book
- Added Space Conquerors to the Section Bonus Mini Extras based on reader request, is based heavily on the Brick Breaker code to maximize readability for new programmers

SEPTEMBER 1, 2015 - FIFTH EDITION, V5

- Improved wording throughout Tennis Game Exercises, also based on feedback from Harold Bowman-Trayford of Renegade Applications, LLC (@RenegadeOwner on Twitter)
- Added example solution to Space Battle Deluxe's Warm Up Exercise 1, Multiple Shots Handled as an Array
- Added Board Game and Racing Tilesheet to bonus source
- Updated Tennis Game video page to mention a new video course exists for Brick Breaker, Racing, and Warrior Legend
- Minor proofreading and wording fixes for Brick Breakanoid

Image credits

Page background by Angelica, from subtlepatterns.com

CC BY-SA 3.0 – Subtle Patterns © Atle Mo.

Cover page background pattern made by Caveman.

Unless or except where otherwise stated the illustrations and diagrams included in this ebook are created by and the intellectual property of the author (Chris DeLeon, for Gamkedo LLC).

