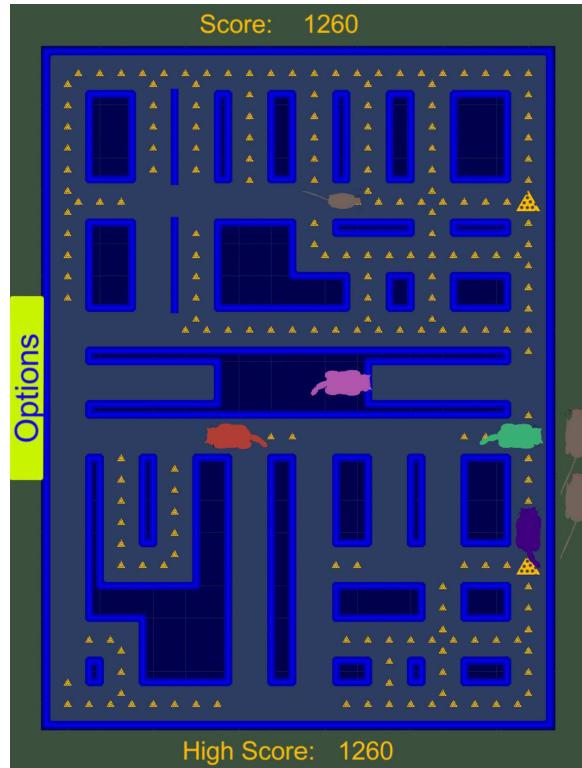


# Rat Race: Creating a 2D Classic Arcade Style Maze Chase Game in UNITY 3D

Todd D. Vance

DEPLORABLE MOUNTAINEER  
*E-mail address:* SUPPORT@TODDVANCE.TECH

©2017 Todd D. Vance



May 4, 2017



## Contents

Chapter 1.	Let's Create a Game!	1
1.1.	Elements of a Classic Arcade Game	3
1.2.	Spiral Approach to Learning and Game Creation	5
1.3.	Duties of the Reader/Student	8
1.4.	What This Book Covers	9
1.5.	Software Needed	9
 Chapter 2. Introduction to UNITY		 11
2.1.	Install UNITY	11
2.1.1.	Close Other Programs	11
2.1.2.	Download Unity	12
2.1.3.	Install Unity	12
2.1.4.	Run Unity	14
2.2.	Install VISUAL STUDIO If Needed	14
2.3.	Install GIMP	15
2.4.	Game Objects and Prototype Assets	15
2.4.1.	Running GIMP	16
2.4.2.	Configuring GIMP	16
2.4.3.	Draw a Circle in GIMP	17
2.4.4.	Draw a Square in GIMP	18
2.4.5.	Draw a Diamond in GIMP	19
2.4.6.	Rectangle 1:2	20
2.4.7.	Rectangle Golden	21
2.4.8.	Equilateral Triangle	22
2.4.9.	Isosceles Triangle	23
2.4.10.	Right Triangle	23
2.4.11.	Regular Hexagon	23
2.4.12.	Regular Octagon	24
2.4.13.	Hexagon 1:1 and Octagon 1:1	26
2.5.	Initial UNITY Scene	27
2.5.1.	Brief Tour of UNITY	28
2.5.2.	Set Up a New Scene	30
2.5.3.	Orthographic Camera and 3D Coordinates	31
2.5.4.	Game Coordinates	32



2.5.5. Importing Assets	33
2.5.6. Creating a Prototype Maze	34
Chapter 3. C Sharp Programming I	37
3.1. UNITY C#	37
3.1.1. A Pathfinder Project	38
3.2. Analysis of our C# Program	40
3.3. Compiling and Linking	42
3.4. A First Refactor	42
Chapter 4. Non-Game Prototype	47
4.1. Adding Motion to the Game	47
4.1.1. The GetInput Method	50
4.1.2. Using the Input Manager Window	52
4.1.3. Adding Physics Components	53
4.1.4. Making the Sprite Move	54
4.1.5. Speeding Up the Motion	55
4.1.6. Adjusting the Speed	56
4.1.7. Collide with the Walls	58
4.2. Design Decisions for the Grid	58
4.3. Our Prototype So Far	59
Chapter 5. Source Control	61
5.1. Git	61
5.1.1. Setting up Github	62
5.1.2. Installing SourceTree	62
5.1.3. Putting Rat Race under Source Control	62
5.1.4. Publishing Rat Race to Github	73
Chapter 6. C# Programming II	75
6.1. Types	75
6.1.1. Numeric Types, Casts, and Rounding	76
6.1.2. Boolean Types	79
6.1.3. Structs	79
6.2. Collections	79
6.2.1. Arrays	79
6.2.2. Multi-Dimensional Arrays	80
6.2.3. Constructors	80
6.2.4. Sets	81
6.2.5. Queues	81
6.3. Scope and Visibility	82
6.4. Control	82
6.4.1. If Statements	82



6.4.2. For Loops	82
6.5. Random Number Generation	82
6.6. Annotations	82
6.7. UNITY Constructs	82
6.7.1. Vectors	82
6.7.2. UNITY Annotations	82
<b>Chapter 7. Procedural Content Generation I: Automatic Maze Generation</b>	<b>83</b>
7.1. Playable Maze Design	84
7.2. Starting the MazeBuilder Class	85
7.3. Testing the Result by Drawing It	88
7.3.1. Prefabs	88
7.3.2. Instantiation of Game Objects in Code	89
7.3.3. Making the MazeBuilder Draw the Maze	89
7.3.4. Fixing Glitches in Motion	90
<b>Chapter 8. Artificial Intelligence: The A* Algorithm</b>	<b>95</b>
8.1. A* on a Grid	97
8.2. The Algorithm	99
8.3. Designing an Implementation of the A* Search	102
8.4. Implementing A* in UNITY C#	104
8.4.1. Write the Program	104
8.4.2. The AStar Class	105
8.4.3. Testing the A* class	108
<b>Chapter 9. Game Mechanics Prototype</b>	<b>113</b>
9.1. Controlling the Enemy	113
9.2. Dots to Eat	117
9.3. Clean up the Assets	119
<b>Chapter 10. Designing a Game</b>	<b>121</b>
10.1. Game Design Document I	121
10.1.1. Marketing Statement	121
10.1.2. Elevator Pitch	122
10.1.3. Technology	122
10.1.4. Features	122
10.1.5. Mechanics	122
10.1.6. Attract Mode	123
10.1.7. Assets	123
10.1.8. Ideas for Future Development	123
10.2. Rapid Prototyping	124
10.3. Game Architecture	124



10.4. Game Engine Architecture	124
10.5. Level of Detail in a Design	124
<b>Chapter 11. Creating Game Art I: The Mouse</b>	<b>125</b>
11.1. Creating the Mouse	125
11.1.1. Google for Mouse Images	125
11.1.2. How does a Mouse Walk or Run?	125
11.1.3. Draw a Concept Mouse	125
11.1.4. Creating a Sprite Sheet	127
11.1.5. Importing the Sprite Sheet	128
11.1.6. Putting the Mouse in the Scene	128
<b>Chapter 12. UNITY Animation Framework I</b>	<b>131</b>
12.1. Animating the Mouse Bones	132
<b>Chapter 13. Creating Game Art II: The Cat and the Cheese</b>	<b>135</b>
13.1. Creating the Cat	135
13.1.1. Google for Cat Images	135
13.1.2. Draw a Concept Cat	135
13.1.3. Creating a Sprite Sheet	136
13.2. Cheese for Dots	137
<b>Chapter 14. Improving the Maze Walls I: Wall Segments</b>	<b>141</b>
14.1. Encoding a Wall Segment	142
14.2. Creating Wall Segments in GIMP	143
14.3. Build a Wall	144
14.4. Corner Walls	147
14.5. Remaining Wall Segments	148
<b>Chapter 15. Improving the Maze Walls II: Building the Maze</b>	<b>151</b>
15.1. Modifying the MazeBuilder Script	152
15.2. Fixing the Wall Sprites	156
<b>Chapter 16. UNITY Animation Framework II: Improving the Animation</b>	<b>159</b>
16.1. Improving the Mouse Animation	159
16.1.1. Add a Run Animation	159
16.1.2. Set Up the Animator	160
16.1.3. Set up the Transitions	161
16.1.4. Modify the PlayerController Script	161
16.2. Improving the Cat Animation	163
<b>Chapter 17. Procedural Content Generation II: Improving the Walls</b>	<b>165</b>
17.1. Merging Islands	165
17.2. Eliminating Dead Ends	167



17.3. Refactoring <b>MazeBuilder</b> and Fixing Concave Corners	170
Chapter 18. Game States I: Win and Lose	175
18.1. States	175
18.2. Discrete Finite Automata	176
18.3. Scenes in UNITY	177
18.4. Creating the Game Over Scene	178
18.5. UI Canvas in Unity	178
18.6. Scene Changes	180
18.6.1. The GameOver Scene	180
18.6.2. The Win Scene	181
18.7. Game Design Document II: Game States	182
18.8. Improving Game State Management	186
Chapter 19. Game State II: Attract Mode	189
Chapter 20. Game State III: Improving Game Behavior	193
20.1. Leveling Up	195
Chapter 21. Cat Box, More Cats and Teleportation Tunnels	199
21.1. Refactor Maze Builder	200
21.2. Cat Box and Tunnels	206
21.3. Help Cats Escape from Box	207
21.4. Teleporters on Tunnels	209
Chapter 22. AI II: Cat Unherding and Flee from Player	211
22.1. Make Cats Avoid Each Other	211
22.2. Flee Player Mode	212
22.3. Growth Hormone Cheese	213
22.4. Make Cats Edible	216
Chapter 23. UNITY Sound Framework	219
23.1. Installing Audacity	219
23.2. Make a Game Tone	219
23.3. Beginning a Music Player Script	222
23.4. Music Theory in a Minute (or 30)	224
23.5. Music Parser	226
23.5.1. Play Tune	229
23.5.2. Remove Initial Whitespace	229
23.5.3. Get And Execute Command	230
23.5.4. Get And Play Note	230
23.5.5. Get And Set Octave	232
23.5.6. Get And Set Ratio	233
23.5.7. Get Number	233



23.6. Playing Tunes in the Game	235
23.7. Other Game Sounds	235
23.7.1. Sound Player Singleton	235
23.7.2. Meow	236
23.7.3. Squeak	238
23.7.4. Gulp	238
23.7.5. Finishing the Sound Player	239
23.7.6. Making the Cat Meow	240
23.7.7. Making the Mouse Squeak	240
23.7.8. Playing Gulp Sounds	241
 Chapter 24. Options Menus and Saving Values	243
24.1. Options Scene	243
24.2. Preferences Game Object	245
 Chapter 25. Review and Refactor	251
25.1. State of the Game	251
25.2. Refactoring Plan	252
25.3. Changing Our Tunes: move this!!!	253
25.4. Eliminating the Game Class	256
25.4.1. Attract Mode	257
25.4.2. Start of Game	263
25.4.3. Reset Player and Death	265
25.4.4. Remove Win scene	265
25.4.5. Remove Old Game Manager	267
 Chapter 26. Game State IV: Making Every State Work	271
26.1. Fixing the States	271
26.1.1. Init	271
26.1.2. Attract Mode: Title	273
26.1.3. Attract Mode: Description	274
26.2. Scores	275
26.2.1. Scoring Points	278
26.2.2. Keeping Score Between Levels	280
26.2.3. Attract mode: High Scores	281
 Chapter 27. The Demo State	287
27.1. Game Recording Framework	288
27.1.1. Prefabs Singleton	288
27.1.2. Mini Language Specification	289
27.1.3. Recorder Singleton	292
27.1.4. The Play Method	299
27.1.5. Demo Scene Adjustments	302



27.1.6. Recording the Maze Generation	303
27.1.7. Starting and Stopping Recording	303
27.1.8. Recording the Dots	305
27.1.9. Recording the Mouse	306
27.1.10. Recording the Cats	307
27.1.11. Recording and Timing the Demo	307
<b>Chapter 28. Game State V: Continuing Where We Left Off</b>	<b>311</b>
28.1. Attract Mode: Options	311
28.2. Game Mode	312
28.2.1. Game Mode: Intro	312
28.2.2. Game Mode: Play	313
28.2.3. Game Mode: Start Level	314
28.2.4. Game Mode: Death	314
28.2.5. Lives Display Widget	316
28.2.6. Game Mode: Game Over	317
28.2.7. High Score Saving	319
28.2.8. Game Mode: Reset Player	320
28.2.9. Game Mode: End Level	320
28.3. Cutscene 1: A Game of Cat and Mouse	321
28.3.1. Sine Curve Zig-Zag	322
28.3.2. Make Mouse Face Direction of Motion	324
28.3.3. Make the Cat Chase the Mouse	325
28.3.4. Face the Mouse	325
28.3.5. Move Toward the Mouse	325
28.3.6. Adjust the Speed	325
28.3.7. Stop, Eat, Grow, and Eat	326
28.3.8. Game Mode Options	328
<b>Chapter 29. Game State VI: Post Game Mode</b>	<b>329</b>
29.1. Post Game Mode: New High Score	329
29.2. Post Game Mode: High Scores	331
29.2.1. Post Game Mode: Options	332
<b>Chapter 30. Improving the Game</b>	<b>335</b>
30.1. Make Attract Mode Scenes Comparable to Title Screen	336
30.1.1. The Description Scene	336
30.1.2. The High Scores Scene	336
30.1.3. The Demo Scene	336
30.2. Navigation Through Attract Mode	336
30.3. Keyboard Control	337
30.4. Details in Description Screen	338
30.5. Eaten Cat Animation	338



Chapter 31. Additional edibles and hazards	341
31.1. Sprite for First Pickup	341
31.2. Blender Crash Course	341
31.2.1. Make a Peanut Butter Label	343
31.2.2. Adding the Label to the Jar	343
31.3. Pickup Object	345
31.3.1. Spawning the Pickup	350
Chapter 32. Fixing the Remaining Issues	351
32.1. Fix Cat Clumping	351
32.2. Death Issues	354
32.2.1. Fix Error Messages on Death	354
32.2.2. Fix Music Player Dying	354
32.3. Fix Movement Issues	354
32.3.1. Fix Mouse Getting Stuck	354
32.3.2. Fix Random AI	354
32.4. Cutscene 2: When the Mouse is Away, the Cats will Play	355
32.4.1. More Graphics in Cutscenes	357
32.4.2. Unload Options on Quit	357
32.4.3. Set High Score Value on Level Up	357
Chapter 33. Extensive Play Testing	359
33.0.1. Cheat Codes	359
Chapter 34. Building for Deployment	363
Chapter 35. Touch Screen Controls	365
Chapter 36. More Features	367
Chapter 37. Publishing	369
Appendix A. Selected Solutions to Exercises	371
Appendix B. Installing Firefox and Acrobat Reader	373
B.1. Installing Firefox	373
B.2. Installing Acrobat Reader	373
Bibliography	375



## CHAPTER 1

### **Let's Create a Game!**

Remember the Golden Age of the arcade in the 1980s? It was what could have been called a PAC-MAN-eat-dot industry; any game that didn't earn a quarter every three minutes got removed and replaced. Thus the games were built to have lights and sounds to attract the players, and then to challenge the players enough that they were likely to lose the game, and fast. Still, from time to time, a skillful player would do unusually well and achieve a spot on the high score list.

We can build a game of this type. We have the technology: faster, stronger, better. And it will cost much less than six million dollars. We shall use the UNITY game engine, whose game editor and corresponding integrated development environment (IDE) runs on the PC and Macintosh. UNITY can deploy games than run on the PC and Mac as well as on Linux, in a web browser and on mobile devices.

Figure 1.1 shows a screenshot of the game we shall create in this book. The player controls a mouse sprite which must eat all the cheese in the maze without being eaten by some very aggressive AI-controlled cats. Elements of this game and the process for creating it include:

- Installation and overview of UNITY
- Introduction to software tools for game making
- Object-oriented C# programming
- Introduction to Artificial Intelligence (AI) programming
- Motion control for both the cats and the mouse
- Grid-based automated level building and Procedural Content Generation (PCG)
- Persistence in UNITY: Score keeping with high-score saving
- Multiple lives and a widget that automatically updates when lives are lost
- Teleportation tunnels
- Sound and music and animated sprite assets, including a music player that plays compositions written as text
- A game state system managing the title and high score screens, as well as states in the game like dying, leveling up, and high score submission



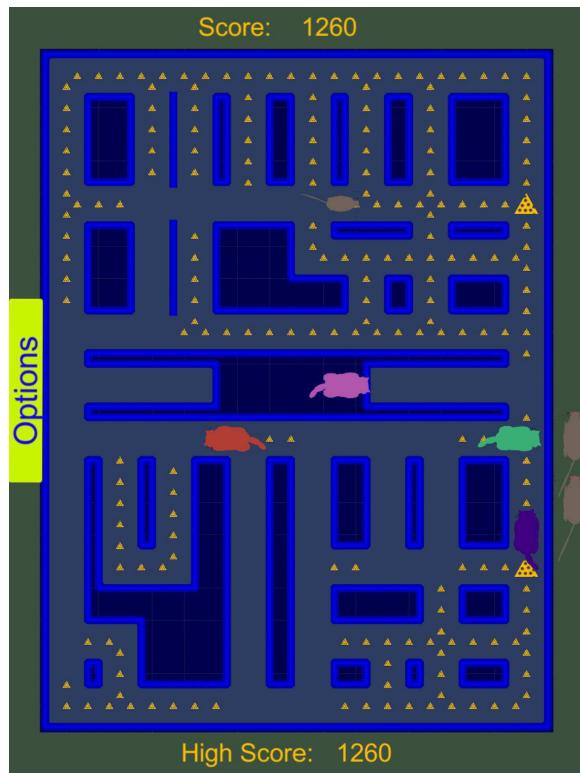


FIGURE 1.1. The *Rat-RACE* Game!

- Scripted cutscenes
- Source control with Git
- Refactoring
- Design patterns
- Pseudocoding algorithms and algorithm design
- Making the game work on multiple platforms
- Publishing your game for money or just for fame
- And more....

This book can be treated as a course that covers many aspects of game programming. It is suitable as a first course in game programming to be taken after an introduction to programming (in any language).

The first reading, carefully following all the steps, walks the reader through building the *Rat-RACE* arcade game. After that, it can be used as a reference for designing and building other 2D arcade-style games (and many of the concepts transfer to 3D game programming). One possibility is to go through the book a second time, but build a completely different game.



After this introduction, the course will properly begin with downloading and installing the main tool, UNITY 3D (currently at version 2017.1, but the code has been tested in UNITY 5.6 as well). Other tools are either optional or can be replaced with alternatives. Tools will be needed to create sound and sprite assets, in particular. An alternative is to find and download assets from the web. There are plenty of free and paid assets. However, check licensing information carefully. Some assets cannot be used commercially and must be avoided if you intend to sell the game. Some cannot even be used in a game published for free.

Next, the game is prototyped, and then an overall design of the game is considered along with implementation notes. This includes the game mechanics (rules and how input is interpreted by the game) as well as the assets (artwork, including sound, images, and structure of levels), game state (such as Attract Mode and leveling up), and architecture (how the software is structured to build the game).

Then, the game building continues in a spiral fashion, from prototype to bad game to better game to polished game.

### 1.1. Elements of a Classic Arcade Game

Think back to a 1980s style stand-up arcade game. When you entered the arcade, many games would be in what is called “Attract Mode,” intended to entice you to spend a quarter. Typically they made no, or maybe a little bit of sound (the real sound began as a reward for spending the quarter) and the controls would be non-functional. There would be a title screen with logos and graphics to announce the game. After a few seconds, it would change to some other screen, perhaps a description screen giving the story of the game or maybe a quick how-to-play, perhaps showing various pickups and their point values.

After several seconds, a high-score list would be shown. The top ten scores with the initials of players promised fame (at least in the local community) and one’s name on a TV screen (in those days before YOUTUBE, a person’s initials appearing on something that looked like a TV for others to see was magical, particularly for children). This generated competition and caused people to play again and again, always trying to beat their own and their friends’ high scores. Children also had fantasies of “real world glory” from scoring well in video games too: *Ender’s Game* [Car94] and *The Last Starfighter* [Fos84] (both of which are also movies) illustrated such fantasies.

Ultimately came the demo: essentially a video playback (without all the game sounds, or at least with a reduced number of sounds) of a game in progress to show what game play looks like. The playback did not



usually last long before either the player died or it was just interrupted at an arbitrary point and the game returned to the title screen and cycled again. (Later, when arcade games came to home computers, some games let users record their own demos to show to friends).

The game would continue cycling between these states while in Attract Mode, until someone inserted a quarter. When that happened, there was typically some kind of sound (as well as a change to the text and/or graphics on the screen) to acknowledge the quarter, and a text field somewhere on the screen would say “1 credit” or “Credits: 1” or similar. Adding more quarters would add more credits. Each credit represents one play, or for games that let two or more players play, each credit allows one additional player to play. Some later games (toward the end of the Golden Age) let you trade credits for additional lives (or powerups) in a game. This was controversial (though it is more common today—if it can be monetized, it will be), since a person with more quarters to spend could boast a higher score without really having more skill.

If the player pressed the “Start Game” button and the number of credits was positive, the number of credits would decrement and the game would begin. One could also press the “Start Two Player Game” button and if there were enough credits, the appropriate number of credits would be subtracted and a multiplayer game would start. The way a multiplayer game worked was that either there were two or more sets of controls, or there was one set of controls and players took turns between lives.

Then the game would start. Typically there would be some kind of introductory music and animation, and then the game would begin for real. In these games, the player had to hit the ground running. If the player looked away for even a short time, the player would lose. Remember, it was all about getting people to spend quarters.

What happened afterward depended on the game being played, but typically, the player would have some small number (three was common) of lives. A player “lost a life” upon dying in the game, and the game was over when there were no more lives left. If instead of dying the player completed a level, the game would advance to the next level. There might be a cutscene (typically an animation and sound sequence that tells the next part of the story, or perhaps is just there for comic relief and a break) or something as simple as the screen flashing or some level-up sound, something to reward the player for a job well done. The next level would begin, and it would be harder than the last level.

Sometimes there would be bonus lives given, typically for reaching a certain score.

With most arcade games, the game could theoretically go on forever: either harder and harder levels were generated automatically, or



levels were just repeated (maybe with the game play moving faster). The *Guinness Book of World Records* [McW83] would publish records for high scores on some games. Some of these records can be found at <http://www.classicarcadegaming.com/wr/guinness/index.htm>.

What usually happened is a person would run out of lives, and the “Game Over” message would show over top the game. The player would stop or disappear in some kind of “death” animation, but the enemies would often continue animating to gloat. After a few seconds of this, if the player made a score that was in the top ten, that person would be allowed to enter their initials. After this (or if it timed out while still showing the default “AAA” initials) the high score screen would appear with the new high score highlighted. After several seconds, it would go back to the “Attract Mode” and title screen.

## 1.2. Spiral Approach to Learning and Game Creation

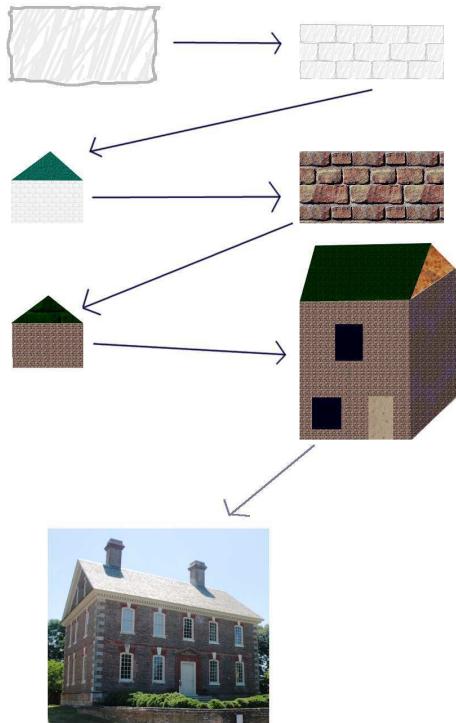
When learning to build something complex, even creating a single piece of it might be beyond the ability of the beginner. Thus, one has to build not the first building block, but a modest approximation to the first building block (Figure 1.2). More of these modest approximations of building blocks are made, and a modest approximation to the whole game can then be made from them. Next, we go back around to the beginning and improve those modest building blocks, one at a time. Thus we have gone into a circle and the second time through, we are on a higher level. Hence we have the “spiral” metaphor. This metaphor applies both to building a game and learning anything complex. See Figure 1.3.

Think of building a house by first making a basic shelter from tree branches. Later, time can be spent building a log cabin. Later still, glass windows could be added. Eventually logs will be supplemented with exterior siding and interior walls, and extras like plumbing and electricity can be retrofitted. (Some buildings have been made following this pattern. A historic building in my hometown started as a log cabin but was turned into a modern house. In London, Westminster Abbey is a thousand-year-old stone building that has been retrofitted with plumbing and electricity. If you can ever visit it, it’s worth a tour.)

In the case of *Rat-RACE*, we start building a rough prototype using basic sprite assets that are relatively easy to create in minutes in GIMP. It starts with a “maze” (really, just a few walls) with a player in it that can’t move.

Then, we add motion to the player, getting into C# programming early. Next we do something rather large, but necessary to continue: we automatically generate a very basic maze. Then we add dots that cannot





**FIGURE 1.2.** We Start Simple and Add Complexity a Little at a Time (image from National Park Service)

be eaten and an enemy that cannot move. However, to place the dots automatically, a simple AI framework is built. This same AI framework then is reused to make the enemy move aggressively toward the player. Then the dots are made edible.

At this time, the game is a very rough prototype, but has much of the mechanics expected in the game. The sprites are ugly, there is no score or leveling up or anything nice like that, but you can move the player, eat dots, and the enemy chases you. This I call the “mechanic prototype” as it is useful for showing off the game mechanic to the boss or whoever you might be making the game for. If your boss is an artist, he will not be happy with that, because it is ugly. A programmer boss would like it better.

The next cycle in the spiral is making all that look better, with sprites closer to what the game will ship with, better looking mazes, and so on. Though the focus of this cycle is artistic in nature, programming will still be necessary. Before actually doing this, we shall consider design goals

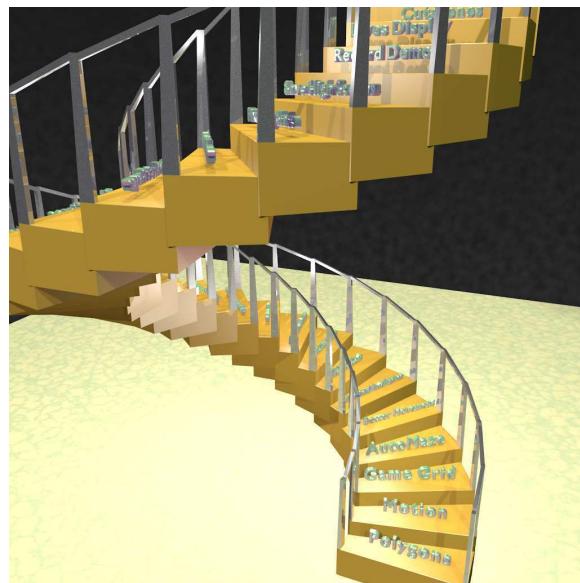


FIGURE 1.3. Spiral Design of *Rat·RACE*

for the game and write a rough draft of a Game Design Document (GDD). Then we make the mouse look like a mouse, for example, and create a basic animation for it. Then we do the same for a cat, cheese, and maze walls (though neither the cheese nor maze walls are animated; not that they cannot be).

Then we improve the animation, so the cat and mouse have a run mode and an idle mode, and their animations reflect that. This will be the first of many encounters with what is called a finite state machine (FSM), also called a discrete finite automaton (DFA).

Next we make the maze generation software work better, so the mazes look more maze-like.

The next cycle is then to actually add game features to it: you can die, you can win the level. That is the minimum to make it a game. Here is where we do a “deep dive” into finite state machines. We also add the beginnings of an “Attract Mode”.

Then, we add some features: a box for the cats to deploy from, and tunnels for the player to escape from the cats. Also, there are now four cats, not one. We give each cat a “personality” by using different AI strategies for each one. We create special “Growth Hormone” cheese that temporarily turns the mouse into...a rat? something bigger. Something that can eat the cats. We add to the cats’ AI the ability to flee this super mouse.

At this stage, the game prototype actually looks like something to show! In fact, the author made a You-Tube video for this stage of production: <https://www.youtube.com/watch?v=mr7ory8QMt0> which is one of many ways to show off your work to a boss or potential customers.

From this point on, it is a matter of doing the “last 10 percent” which takes at least as much time and effort as the first 90 percent, paradoxically. We add sound, demo recording, scoring and level management, bonus pickups, and other features, as well as fix bugs, play test (not as fun as it sounds, considering how much play testing needs to be done), cheat codes to make play-testing easier, and finally deploy on multiple platforms, publish, and maybe, make money.

### 1.3. Duties of the Reader/Student

Plan to put quite a few hours into this course. An hour each weekday is better than 10 hours every Friday, or you will lose your momentum. Two hours each weekday is even better still. However, five minutes every hour or continually while multitasking doesn’t really do the job (The author even recalls a recent study that multitasking causes permanent changes in the brain that lowers a person’s IQ....). A continuous block of an hour a day most days of the week dedicated to focusing on just this project is close to optimal.

What you get out of this course is a direct function of what you put in. Try to make various concepts an integral part of you rather than gaining a cursory understanding. If your document reader or ebook reader allows you to do so (ACROBAT READER does: <https://get.adobe.com/reader/enterprise/>, and so does the AMAZON KINDLE device), highlight and add notes so you can quickly review sections of this document later. At worst, you may have to kill some trees for the benefit of self-education (which is a funny way of saying, if nothing else works, take notes in a paper notebook).

Sections include theory as well as practice. The theory is important for understanding. Just as mankind did not understand how to put a man on the moon until people like Newton and Einstein wrote the theory, we cannot be effective game programmers without understanding the theory of state machines, AI, and so on.

In addition, references are given. It is strongly encouraged to at least look at the free online references. Some references requiring purchase, or at least a visit to a good university library or an interlibrary loan, are also helpful but are treated as optional. These references give more depth to the topics covered.



#### 1.4. What This Book Covers

By following this text, you will accomplish the following:

- Install Unity and other software.
- Create polygon sprites for prototypes.
- Set up the beginnings of a maze with the sprites.
- Learn some C# programming.
- Make a sprite move under keyboard control.
- Make the sprite collide with walls without passing through.
- Create a grid to build the game on.
- Check your project into source control.
- Use Procedural Content Generation (PCG) to build a playable but basic maze.
- Learn the basics of Artificial Intelligence (AI) programming to control the prototype enemy.
- Create pickups (the dots) the player can consume.
- Write a draft Game Design Document (GDD) for the game.
- Create sprites for the game: mouse, cat, cheese, wall segments.
- Animate the mouse and cat sprites.
- Improve the maze-generation code for better mazes.
- Add a game state machine to control in-game states, win and lose, and attract mode.
- Add teleportation tunnels.
- Create and add sound effects and music, including a crash course in music theory.
- Play-test, find, and fix bugs.
- Save options between games.
- Learn to refactor code.
- Maintain a high score list between games.
- Write software to record and play back a demo.
- Add scripted cutscenes.
- Add better graphics to make the game look good.
- Walk through a crash course in Blender 3D modeling software to make the graphics.
- Add cheat codes.
- Build, deploy, and publish.

#### 1.5. Software Needed

The following software is recommended. UNITY 3D is the only absolutely required one. The other software is very useful and strongly recommended, but replacements can be found. All recommended software is available for free (with no time limits or other “demo” limitations). This



text purposely avoids limited “free demos”. Some software listed has paid versions with extra features, but none of these features are important for this text.

- UNITY 3D (Personal Edition is fine) Version 5.6 (slightly later versions will probably work) <https://unity3d.com/get-unity/download>
- GIMP (GNU Image Manipulation Program) <https://www.gimp.org/downloads/>
- AUDACITY (audio software) <http://www.audacityteam.org/download/>
- VISUAL STUDIO Community Version 2017 <https://www.visualstudio.com/downloads/> (or MONODEVELOP, included with UNITY, if not on Windows).
- GITHUB DESKTOP App <https://desktop.github.com/>
- SOURCETREE <https://www.sourcetreeapp.com/>
- A web browser (for example, FIREFOX: <https://www.mozilla.org/en-US/firefox/products/>)
- A .pdf reader (to read this document). Recommended is ACROBAT READER: <https://get.adobe.com/reader/enterprise/>
- Optionally for more advanced users: BLENDER (3D modeling software) <https://www.blender.org/download/>

**EXERCISE 1.1 (Install Needed Software).** Using the urls provided, install as much of the above software on your system as you can. If you get stuck, walkthroughs are given throughout the text for much of the software when it is first used, starting with UNITY itself in the next chapter.



## CHAPTER 2

## Introduction to UNITY

In this chapter, we install Unity, create some basic prototype assets (polygons), and mock-up a prototype maze with a polygon player and enemy.

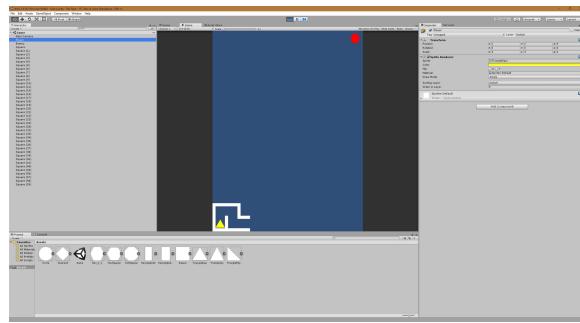


FIGURE 2.1. Basic Prototype with Non-Moveable Sprites

At the end of this chapter, we will have a scene with sprites in it. It's not much. Figure 2.1 shows the really basic "game". The purpose of the prototype is to have something, anything, that is visible progress in the development of the game. This is what a programmer can show his boss while saying, "I did this today." In the next chapter, we introduce C# programming, and in the chapter after that, we use C# to make the sprite move.

### 2.1. Install UNITY

Before doing anything else, make sure you have time to install UNITY. There is a lot to install, so you might want to wait till just before bedtime and let it run overnight, for example. The initial steps take about 15 minutes, then you can let it run for perhaps hours (though checking during the middle for a popup that needs to be OK'd may be necessary).

**2.1.1. Close Other Programs.** Before installing UNITY, just to be safe, close any open instance of UNITY or VISUAL STUDIO or MONODEVELOP. Running instances can interfere with the installation process. (This has happened with the author, who installed VISUAL STUDIO Tools for UNITY



while VISUAL STUDIO was running. No error messages were shown and the installation was declared successful, but VISUAL STUDIO crashed when opened from then on. It took some Googling to find a fix.) If you want to be really safe, close other programs too. It is rare in modern days (with computers, the last few years is “modern”), but still not unheard of, for software installers to crash a machine and cause you to lose data in open programs.

**2.1.2. Download Unity.** Go to the UNITY download page: <https://unity3d.com/get-unity/download>. Click the green **Choose your Unity Download** button. Look for the Personal edition (this is the free version) and click the green **Download Now** button.

Look at the green **Download Installer** button carefully. If it says Version 5.6.something, you are good to go. If not, you might still do ok if it is not far off, but if you really want to be safe (in particular, if you are a beginner), you would have to find the link near the bottom of the screen that says "Older versions of UNITY" and click that, then find some version of 5.6.something in the list and download that.

When you find UNITY 5.6, click the appropriate download button to download the installer. What happens next depends on your browser, but you could either just run it, or save it first, find where it was saved, and then run it. You may have to give the computer or browser permission to install.

**2.1.3. Install Unity.** When this is done, the UNITY **Download Assistant** window comes up. Read and accept the license agreement. (I know, nobody reads it. A software company proved that once by offering a cash prize; it took 3000 sales before it was claimed: <http://techtalk.pcpitstop.com/2012/06/12/it-pays-to-read-license-agreements-7-years-later/>. But, legal disclaimer, read it or suffer the fate of Dilbert: <http://dilbert.com/strip/1997-01-14>.)

Check the “Accept” box after you understand all that legalese (you do understand it all, right?). Click **Next** or **Continue**. Select “64 bits” if appropriate for your system (a 64-bit system is strongly recommended) and click **Next**. Any other screens that might have been added since this was written, read carefully and try to make the right decision and click **Next**, till you get to the **Choose Components** screen. Now some decisions need to be made.

The defaults are probably OK for most people. You definitely want to install UNITY, and install the documentation and standard assets. The example project is completely optional. If you are on Windows, you want



to install VISUAL STUDIO along with VISUAL STUDIO Tools for UNITY. Otherwise, you can use the built-in MONODEVELOP (which you can use on Windows too if you prefer). Then what remains is optional depending on what you want and where you want to deploy games. If you have the disk space and don't mind waiting a while to install, it's ok to just check everything. If you leave something out, it is possible to add it later from within UNITY.

Click **Next**. Downloading to a temporary location is recommended. Most people can keep the default install folder, but if you want multiple copies of UNITY, you have to make sure they are in different folders.

Follow any remaining instructions, if any. Note that if you want to have multiple versions of UNITY at the same time, you will have to change the directory it installs to, because by default, it writes over whatever version you have already. Otherwise, you can leave the default. For Mac, you may have to move another version somewhere else first, then install.

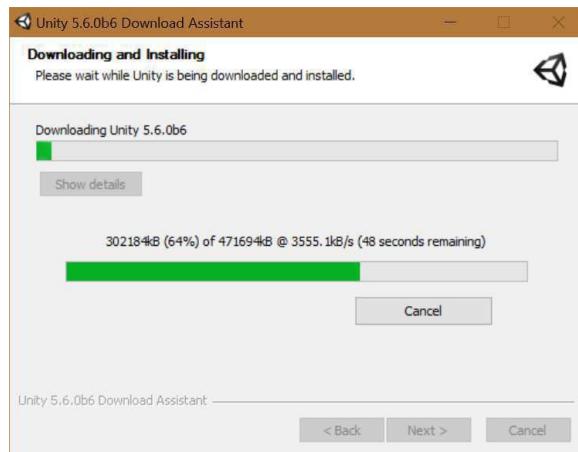


FIGURE 2.2. UNITY Installer Has a Lot of Work Left to Do

Click **Next** and wait (Figure 2.2 shows 48 seconds remaining from one component. The upper bar shows there are many, many more components left to install.) It takes time. If you checked a lot of optional components to install, it takes a lot of time.

Now is a good time to walk away for maybe even a few hours. But note that you may have to clear a popup or two during the install (e.g. VISUAL STUDIO already up to date—the author got this popup installing the beta version, but not the final version, so it “should” not be an issue if not using a beta version). To be safe, visit the computer from time to time to OK/Continue any popups.

**2.1.4. Run Unity.** When done, test that it works by running UNITY 3D. You may be asked to give it access to private networks. Do this, but not to public networks, whose safety cannot be guaranteed. Feel free to watch any videos or browse any new user introduction documentation. If you have not done so already, UNITY will require you to register. There are no serious side effects to doing this, and they do not send very many e-mails (if you correctly check/uncheck boxes offering to send you e-mails). (If you later register for other optional Unity services, some of them send more e-mails).

## 2.2. Install VISUAL STUDIO If Needed

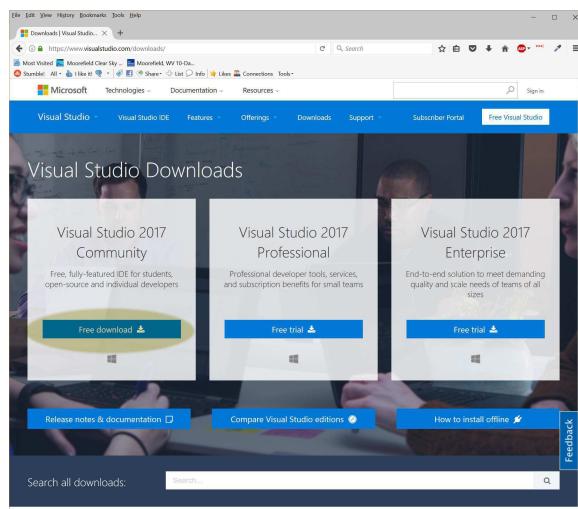


FIGURE 2.3. Install VS2017 Community Edition if Needed

This step should not be necessary because the UNITY installer should do it for you (if on Windows; Mac uses MONODEVELOP instead and this section does not apply). However, if the UNITY installer did not do this already for some reason, and you do not already have Microsoft VISUAL STUDIO 2017, you need to install it. The free community edition is fine. Go to <https://www.visualstudio.com/downloads/> (Figure 2.3) to get to the download page.

Under the VISUAL STUDIO Community section, click the blue **Free Download** button. What happens next depends on your browser, but you could either just run it, or save it first, find where it was saved, and then run it. You may have to give the computer or browser permission to install.

When this is done, the installer window comes up. Just follow the instructions. The default settings should be ok, except be sure that C# is included with the installation.

### 2.3. Install GIMP

While optional (if you have a preferred graphics editor already), this is strongly recommended. Instructions in this book will assume GIMP, and unless your graphics editor is sophisticated like the expensive full version of PHOTOSHOP, it might not have all the features needed to do the job.

Go to the page at <https://www.gimp.org/downloads/> and click the appropriate installer button. What happens next depends on your browser, but you could either just run it, or save it first, find where it was saved, and then run it. You may have to give the computer or browser permission to install. Follow the instructions, accepting the license agreement.

### 2.4. Game Objects and Prototype Assets

Let us create some reusable prototype sprite assets. To do this, we use GIMP, though if you already have and know another full-featured graphics editor like PHOTOSHOP, go ahead and use that.

We shall create several prototype sprite images that will be useful for not just Rat Race, but other games you may wish to build later. These sprites are placeholders until we create something better. They are just for getting started and moving forward quickly.

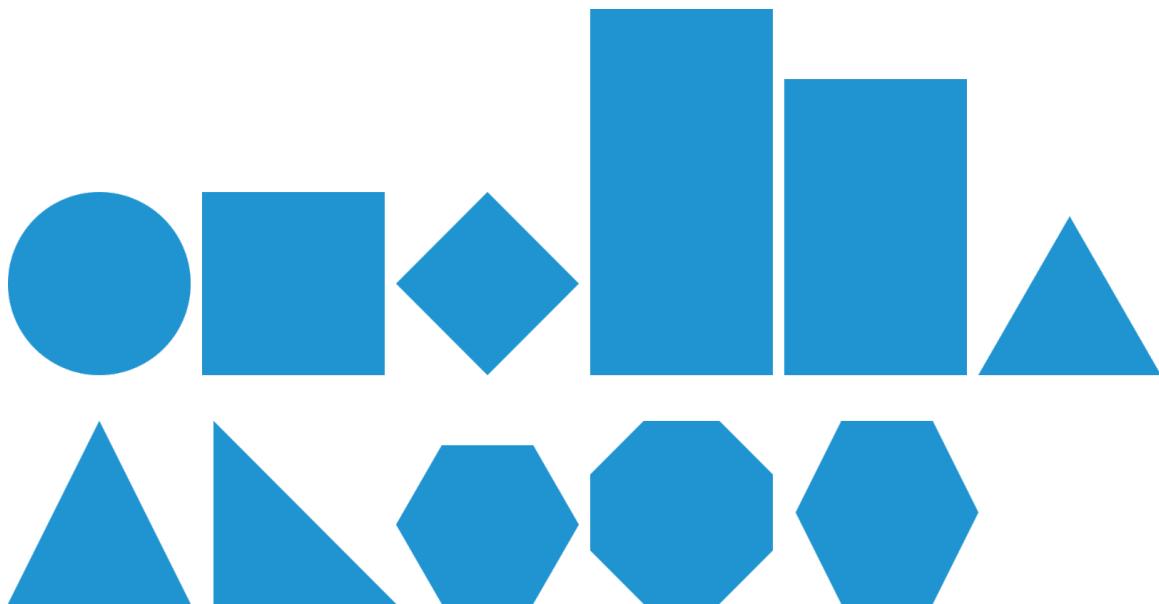


FIGURE 2.4. We Shall Create These Shapes

The prototype shapes we shall make are:

<http://toddvance.tech>



- (1) A circle
- (2) A square
- (3) A diamond
- (4) A rectangle twice as high as wide
- (5) A rectangle with “golden” aspect ratio
- (6) An equilateral triangle
- (7) An isosceles triangle that fits in a square
- (8) A right triangle that fits in a square
- (9) A regular hexagon
- (10) A regular octagon
- (11) A hexagon that fits in a square
- (12) An octagon that fits in a square (We won’t need to make this, because unlike a regular hexagon, a regular octagon already fits in a square)

These shapes will be 256 units wide, a reasonable size, particularly given that some graphics cards can optimize shapes whose dimensions are a power of two. They will be white on a transparent background. They are white so that UNITY can easily recolor them to any color (UNITY uses a “multiplicative” model to recolor sprites). We show all the shapes (in blue rather than white because the paper it is printed on is white) that we shall create in Figure 2.4.

#### 2.4.1. Running GIMP.

**TASK 2.1.** First, start up GIMP (or your favorite graphics editor, but the instructions here assume GIMP). It may take a few minutes the first time it is run because it loads and caches fonts and scripts.

**2.4.2. Configuring GIMP.** The author prefers the GIMP layout to be a little different from the default. This is a user preference, but if you do not already have a preference, consider the following changes.

**TASK 2.2.** First, let us select “Single Window Mode”. When GIMP opens the first time, there are three separate windows. Worse, if you have multiple monitors, they are spread around among the monitors. That is the first thing to fix.

Find the main Image window (with “GNU Image Manipulation Program” in its titlebar, and the “Gimp” creature with eyes and ears in dark gray over a light-gray background, not the Layers-Gradients or the Toolbox windows) and select  Windows  Single Window Mode (this notation means, select the Windows menu, then the Single Window Mode item from the menu). This will simplify GIMP usage greatly. The



■ Toolbox and ■Layers windows are now docked to the main ■Image window.

The remaining preferences are set through **Edit** ➤ **Preferences**. A popup window labeled “Preferences” now lets you change almost anything about GIMP. If you mess up too badly, you can always hit the **RESET** button, which will also reset to the default multiple window mode, but you know how to fix that now.

Here are the author’s recommendations for preferences:

- (1) Under **Environment** (this notation means “Environment” appears in a popup), change the  minimum number of undo levels to 100
- (2) Under **Help System**, check if there is a warning that “the user manual is not installed locally”. If you see this, you have two options: either return to the GIMP webpage and download and install the user manual and set the  User Manual option to “Use a locally installed copy”, or make sure the  User Manual option is set to “Use the online version”.
- (3) Under **Tool Options**, check the  Save tool options on exit box. You can reset a tool option at anytime if you do not like what it was saved to. Also, check the  Set layer or path as active box, which ensures if you click on something with the move tool, it becomes the active layer to be moved.
- (4) Under **Toolbox**, check all three boxes at the top:  Show foreground and background color,  Show active brush, pattern, and gradient, and  Show active image.
- (5) Under **Default New Image**, change the  Image Size value to 256x256. Change the  Fill with option to “Transparency”.
- (6) Under **Default Grid**, the black default color is too often invisible on images used in games. Click the black box following  Foreground color to bring up a color chooser dialog. Under the HTML notation, type: “1ecbb5” to make it a medium blue-green. Change the default grid spacing from 10x10 to 32x32.

After setting all the preferences, click **OK** to finish. Then, to make the changes take effect, exit GIMP and restart it.

### 2.4.3. Draw a Circle in GIMP.

TASK 2.3. Open GIMP if it is not already open. Select **File** ➤ **New...** and make sure the width and height in the popup is set to 256x256. Also be sure, under **Advanced Options**, we use “RGB” colorspace and fill with



“Transparency”. Click **OK**. If the image is too big or too small, you can use **ctrl**-Scroll Wheel (Mac: **⌘**-Scroll Wheel) to zoom in and out.

Let us get used to using the grid immediately: Activate (making sure there is a check next to the menu item): **View** **> Snap To Grid**, then **View** **> Show Grid**, and finally open the **Image** **> Configure Grid** popup. Make sure the **Foreground** color is reasonable (such as Hex 1ecbb5) and that the Spacing width and height is 32x32 pixels.

Let us now outline a circle selection. In the **Toolbox** (by default, to the left of the main **Image** panel), select the second icon in the top row, which looks like a dotted ellipse filled with gray. This is the Ellipse Select tool.

Since **Snap To Grid** is in effect, you can click the top-left corner of the image, and drag to the bottom-right corner and release, and you will get a circle filling the entire image. Just to be sure, look at the **Tool Options** section of the **Toolbox** panel and check that the position is (0,0) and the size is 256x256.

By default, the background color is white, and we want a white circle, so select **Edit** **> Fill with BG Color**. The image should now be a white disk.

Save this as a PNG (Portable Network Graphics) image: select **File** **> Export As**. A file dialog pops up. At the bottom, you should see **Select File Type (By Extension)**. This means, saving as something.png automatically creates a PNG file, which is what we want. So, change the Name (at the top) to “Circle.png”. Make sure the **Save in folder** points to a location you can find again when you need to import this into UNITY. Click **Export** and accept the defaults of any popups that appear. Check that the new file was created.

You could now exit GIMP and restart it for the next shape, or you could look at the tab bar above the graphics area of the main window, and notice there is one tab with a thumbnail image of the circle in it. Click the “x” in the box to close this image. When a popup appears, you can click **Discard Changes** because it was already exported. You also have the option of saving the file in GIMP’s internal format. This is useful if you need to edit the file later.

#### 2.4.4. Draw a Square in GIMP.

**TASK 2.4.** Open GIMP, if it is not already open. Select **File** **> New...** (note the shortcut key in the menu—learning these can save time in the future) and make sure the width and height in the popup is set to 256x256. Also be sure, under **Advanced Options**, we use “RGB” colorspace and

fill with “Transparency”. Click **OK**. Use **[ctrl]-Scroll Wheel** (Mac: **[⌘]-Scroll Wheel**) to zoom to a convenient size if necessary.

We do not have to use a grid this time. Not only that, we do not have to select anything, since the image is already square. Ensure the background color is still white and select **Edit > Fill with BG Color**. The image should now be filled with a white square. (If you had accidentally selected something, **Select > None** will clear it. Then you can **Edit > Fill with BG Color** again to fill the whole image.)

Execute **File > Export As**. Confirm that at the bottom, you see “Select File Type (By Extension)”. Change the Name (at the top) to “Square.png”. Make sure the “Save in folder” points to a location you can find again (it is recommended to be the same folder that the Circle was saved into). Click **Export** and accept the defaults of any popups that appear. Check that the new file was created.

Click the “x” in the box of the tab with the thumbnail image of the square in it to close this image.

**2.4.5. Draw a Diamond in GIMP.** A diamond is really like a square, but rotated 45 degrees. Again, we want the diamond to fill the square image region.

TASK 2.5. Open GIMP if necessary and create a new image, 256x256 with transparent background, just as before. Show, snap-to and configure the grid on 32x32 squares. We will actually need the grid this time.

To draw a diamond, we need to use the “Free Select” (Lasso) tool. It is the third icon on the top row of the toolbox, and it looks like a lasso: click this.

There are several ways the Free Select tool can be used. You can use it to draw like a pencil, or you can use it to make a polygon: click, then click the next vertex, and so on. It will automatically put straight lines between the vertices. You can close the polygon by clicking on the first vertex already created. A small circle will appear when your mouse pointer is close enough to the vertex to close the polygon when clicked.

It is the polygon selection that we shall use. We now select the four corners of the diamond, starting at the top and going clockwise: top, right, bottom, and left. Click on the top of the image, at the middle. Because **View > Snap To Grid** is on, it will land exactly at the middle of the top edge (at coordinates (128,0)).

Click right edge, right in the middle of the edge, to make the next vertex. A diagonal line will appear from the middle of the top edge to the middle of the right edge.



Continuing, click the middle of the bottom edge, then the middle of the left edge, creating three-fourths of the outline of the diamond.

Finally, click the middle of the top edge again, ensuring a disk shows indicating that the polygon is closed. The lines shown should show you have selected a perfect diamond.

Now that the diamond-shaped region is selected, fill it with white, the background color, as before, and export it to a file called “Diamond.png” in the same directory as the other shapes.

If you make a mistake in selecting the diamond, the `ESC` key will cancel the selection so you can start over. The `Backspace` key will just cancel the last vertex so you can try again. If you have finished making the selection and it does not look right, `Select > None` will clear it so you can try again.

**2.4.6. Rectangle 1:2.** The only difference between this rectangle and the square already made in GIMP is the rectangle is 512 pixels high by 256 pixels wide. So, one can follow the same instructions as for the square with only a few changes, which are highlighted:

Open GIMP, if it is not already open. Select `File > New...` (note the shortcut key in the menu—learning these can save time in the future) and make sure the width and height in the popup is set to 256x**512**. Also be sure, under `File > Advanced Options`, we use “RGB” colorspace and fill with “Transparency”. Click `OK`. Use `ctrl`-Scroll Wheel (Mac: `⌘`-Scroll Wheel) to zoom to a convenient size if necessary.

We do not have to use a grid. But it is harmless, so if you want to practice, show and snap-to the grid: `View > Snap To Grid`, `View > Show Grid`, and then `Image > Configure Grid` and make sure the “Foreground” color is reasonable (such as Hex `1ecbb5`) and that the Spacing width and height is 32x32 pixels.

This time we do not have to select anything, since the image is already square. Ensure the background color is still white and select `Edit > Fill with BG Color`. The image should now be filled with a white square. (If you had accidentally selected something, `Select > None` will clear it. Then you can `Edit > Fill with BG Color` again to fill the whole image.)

Execute `File > Export As`. Confirm that at the bottom, you see “Select File Type (By Extension)”. Change the Name (at the top) to “**Rectangle2h.png**”. Make sure the “Save in folder” points to a location you can find again (it is recommended to be the same folder that the Circle was saved into).

Click **Export** and accept the defaults of any popups that appear. Check that the new file was created.

Click the “x” in the box of the tab with the thumbnail image of the square in it to close this image.

**2.4.7. Rectangle Golden.** As you might have surmised already, to make the golden rectangle, the same instructions as for the square and the double-high rectangle hold, but the dimensions change. The only question is, changed to what? We show how to compute this, but non-mathematically-inclined readers might want to skip to the exercise below, where the final answer is given.

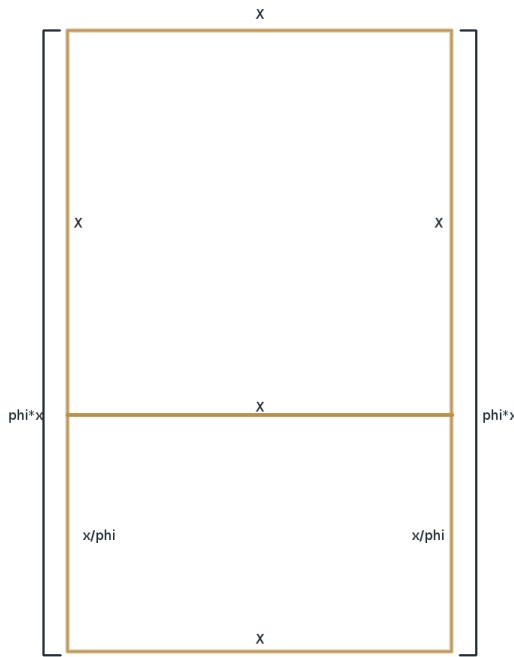


FIGURE 2.5. A Golden Rectangle,  $x=256$ ,  $\phi=\text{phi}=1.61803398875$

A golden rectangle is defined to be one such that, if it is cut into two by cutting perpendicular to the longer dimension, in such a way that one of the two pieces is a square, then the remaining piece is a golden rectangle as well. See figure 2.5. Thus, the larger rectangle is cut into a square and a smaller rectangle, and the smaller rectangle's side ratio is the same as the larger rectangle's (namely  $\phi$  or phi, which is  $\frac{1+\sqrt{5}}{2}$ , or about 1.61803398875). One can show using algebra (by solving  $x + x/\phi = \phi * x$  for  $\phi$ ) that the value

given is the only positive number that makes this work. Thus, since we want  $x$  to be 256,  $\phi x$  has to be 414 (since it has to be an integer).

Why the golden ratio? Artists believe it to be a “pleasing shape” to the eye.

**EXERCISE 2.6.** Follow the instructions for creating a square or a double-height rectangle, except create a 256x414 rectangle. Export it as “RectangleGolden.png”.

**2.4.8. Equilateral Triangle.** See Figure 2.6. One can determine the correct dimensions for an equilateral triangle using algebra, just as with the golden rectangle. Non-mathematically-inclined readers might want to skip to the task below.

If each side has length  $x$ , then dropping a vertical altitude line from the top vertex to the bottom side makes a pair of right triangles. The hypotenuse is  $x$  and one side is  $x/2$ . From the Pythagorean Theorem, the remaining side must be  $h$  such that  $h^2 + (x/2)^2 = x^2$ , and this can be solved using the techniques of algebra to show  $h = x\frac{\sqrt{3}}{2}$ , or about 0.86602540378 $x$ .

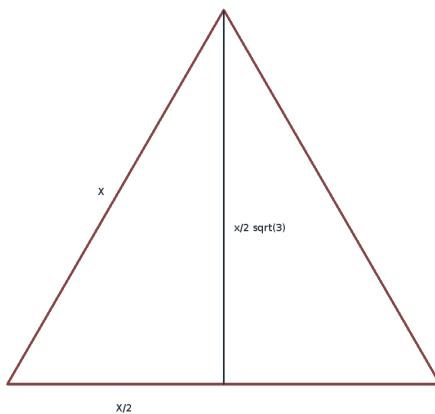


FIGURE 2.6. Constructing an Equilateral Triangle

**TASK 2.7.** Create a new image whose dimensions are 256x222 (222 is about  $256 \times 0.86602540378$ ). In addition to snapping to grid, we need another option turned on for this one: `View > Snap to Canvas Edges`.

Just as we did with the diamond, select the Lasso tool and make a path from the bottom left corner to the middle of the top edge to the bottom right corner, and back to the bottom left corner, and fill it with white. Export it as “TriangleEqui.png”.

**2.4.9. Isosceles Triangle.** The isosceles triangle is like the equilateral triangle, but easier.

TASK 2.8. Follow the instructions for an equilateral triangle, but instead of a 256x222 image, use a 256x256 image. Export it as “TriangleIso.png”.

**2.4.10. Right Triangle.** A right triangle is easier than the equilateral or the isosceles triangle.

TASK 2.9. Make a 256x256 image and use the lasso tool to make a polygon from the upper left corner to the lower right corner to the lower left corner and back to the upper left corner. Export it as “TriangleRight.png”

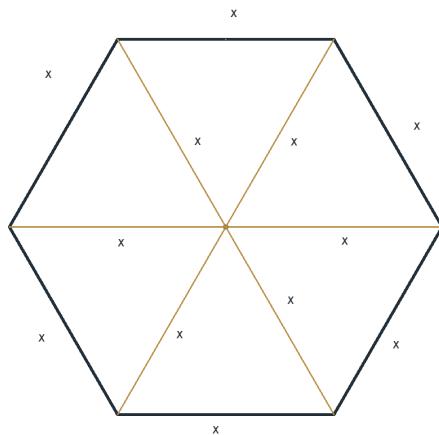


FIGURE 2.7. A Regular Hexagon Can Be Cut into 6 Equilateral Triangles

**2.4.11. Regular Hexagon.** This is the hardest shape so far, but still not that hard once the computations for the equilateral triangle have been done. Non-mathematically-inclined readers might want to skip to the task below.

From Euclidean geometry, we know a regular hexagon can be decomposed into six equilateral triangles: the radius (from center to a vertex) of a regular hexagon is equal to the length of a side. See Figure 2.7.

Therefore, if we want the width of the hexagon to be  $2x = 256$ , each side is seen to be of width  $x = 128$ . The height would then be twice the height of each equilateral triangle, or  $2x\sqrt{3} \approx 222$ . Therefore, we start with an image that is 256x222.

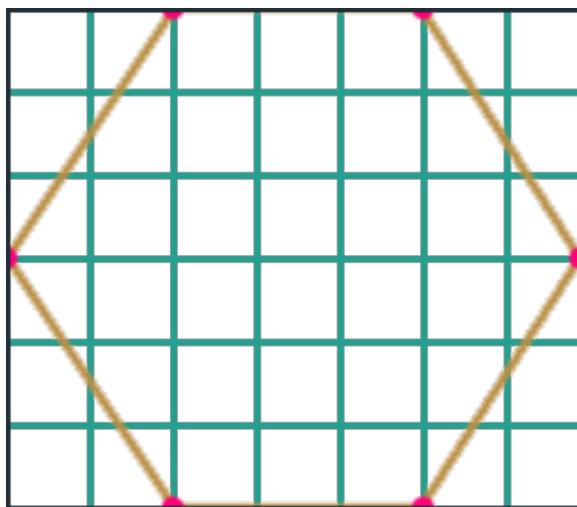


FIGURE 2.8. Selecting the Hexagon

**TASK 2.10.** Start with an image that is 256x222. Our grid size of 32 does not divide 222 evenly, so open **Image > Configure Grid**. Below the boxes where you enter the spacing is a “chain link” icon. Click that to “break” the link. This lets us set the width and height to different values. Then, keep the width at 32, but set the height to 37, which divides 222 evenly. Use the lasso tool to select the hexagon as shown in Figure 2.8 and fill the selection with white. Export the image as “HexRegular.png”

**2.4.12. Regular Octagon.** We wish to create an octagon with equal sides. Its width must be 256, and because of the symmetry of an octagon, its height will also be 256.

Think of a line connecting the center of the octagon to the midpoint of some side. This will be an altitude of one of the eight (non-equilateral) triangles that make up the octagon. This altitude would have to have length 128 so that the octagon would be 256 units wide.

How do we find the length of a side of this octagon? This side is the base of one of the eight triangles, whose altitude is known to be 128. It takes

trigonometry to determine this. Non-mathematically-inclined readers might want to skip to the task below.

We need to consider the angles of the eight triangles making up the octagon. Because there are 360 degrees in a circle and there are 8 triangles, the apex of each triangle (at the center of the octagon) is 45 degrees. Since the sum of the angles in a triangle add to 180 and since the other two angles are equal, they must be  $\frac{180-45}{2} = 67.5$  degrees each. By dividing any one of these triangles in half, we have two right triangles, with the non-right angles being 67.5 and  $\frac{45}{2} = 22.5$ . See Figure 2.9.

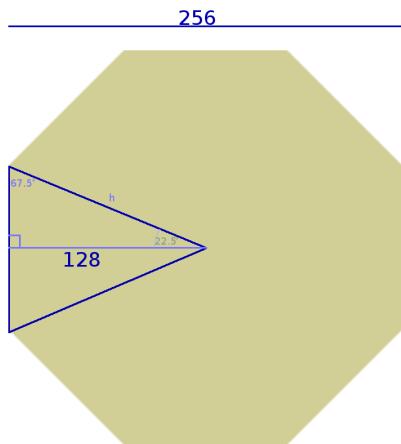


FIGURE 2.9. Computations for the Octagon

The one length we know is the distance from the center of the octagon to the midpoint of a side, corresponding to one side of the right triangle, and the length is 128. Because the cosine of an angle is the adjacent side divided by the hypotenuse, we have  $\cos 22.5^\circ = 128/h$ . Thus,  $h$ , the radius of the octagon, is  $\frac{128}{\cos 22.5^\circ} \approx 138.546201638$ . From the Pythagorean theorem, we can compute the length of the remaining side, which is half the length of a side of the octagon: about 53.0193359853. Thus, each side of the octagon is 106 pixels.

How do we size the grid to be able to select the proper region? The image is 256 pixels wide, and 106 pixels from the center make the length of a side. That leaves 150 pixels to the left and right, or 75 to the left, and 75 to the right. The easiest way to do this is to divide the octagon into quarters, and make a quarter of it. Then, duplicate it twice to fill out the whole octagon. We shortly show how to do this.

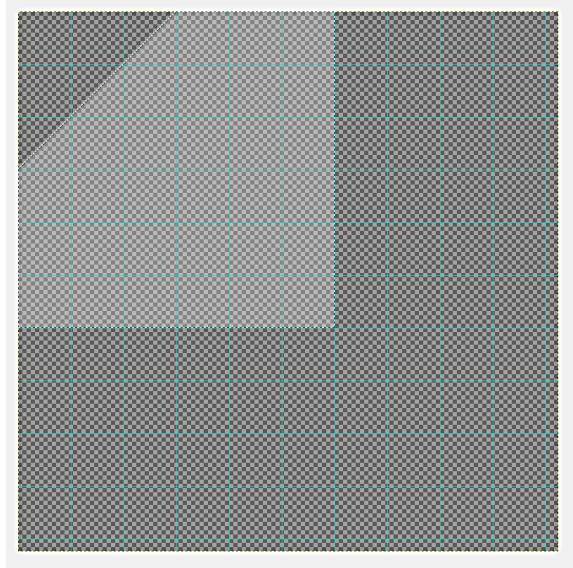


FIGURE 2.10. Selecting a Quarter of the Octagon

**TASK 2.11.** Start with a 256x256 image.

Set the grid size to be 25x25, since 25 divides 75 evenly. We will select a little more than a quarter of the octagon, knowing that the over-selection will be written over. See Figure 2.10 and use that as a guide to select the upper-left quarter. Fill it with white.

Now do the following: select `Layer > Duplicate Layer`. You will see an additional layer appear in the `Layers` panel to the right, but the image will not change (since it is a duplicate right on top of the original). Next, select `Layer > Transform > Flip horizontally`. Now, you will see the upper portion of the octagon in the image window. Finally, select `Layer > Merge down` to recombine the two layers into one.

That was half the job. We now finish the job and make a complete octagon. Execute `Layer > Duplicate Layer`. Again, you will see an additional layer appear in the `Layers` panel to the right, but the image will not change. Execute `Layer > Transform > Flip vertically`. Now, you will see the whole octagon in the image window. Finally, `Layer > Merge down` to recombine the two layers into one.

Export this image as “OctRegular.png”.

**2.4.13. Hexagon 1:1 and Octagon 1:1.** We are going to “cheat” and use our existing hexagon and octagon to make the versions that fit into a square. Actually we do not need to do this for the octagon: the regular octagon already fits into a square!

We now modify the hexagon.

**TASK 2.12.** From GIMP, instead of creating a new image, execute **File > Open**. Find the “HexRegular.png” file (you may have to change directories) and select it and click **Open**.

We now make it fit a perfect square by adjusting the image size. Open **Image > Scale Image**. To the right of the **Image Size** width and height boxes, is a chain link. Click that link to “break” the link. This lets us scale the width and height separately. Leave the width at 256 and change the height to 256. Click the **Scale** button and you now see a hexagon that is no longer regular, but its height and width are the same. Export it as “Hex\_1\_1.png”.

**CHALLENGE 2.13.** This is for those comfortable with doing trigonometry computations. Create a regular pentagon (five-sided polygon) as a sprite. The computation method (but not the results) for the octagon should help. The trick of making one-fourth of it and duplicating will not work anymore, so the reader needs to find another way. Note that we will not need the pentagon in this text, so this challenge is optional.

## 2.5. Initial UNITY Scene

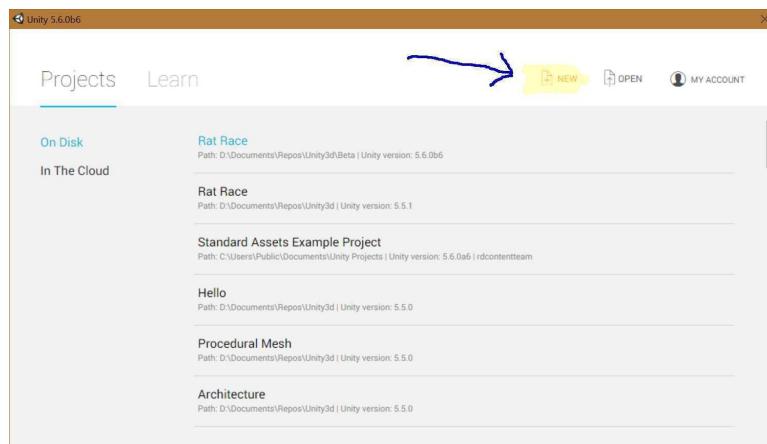


FIGURE 2.11. Create a New Project in UNITY

**TASK 2.14.** Open UNITY. You may be asked to sign in. If you have a registered UNITY account, use the e-mail address and password for that. If not, click **Create New** and follow the instructions to create a new account with UNITY.

After all this is done and you are signed in, the **Projects** window will appear. Click the **New** button, with the icon that looks like a dog-eared sheet of paper with a plus sign on it (Figure 2.11).

Enter the name “Rat Race” for the project name. Choose a location that is convenient and that you can find again easily. You are probably a member of just one organization unless you set up more, so either accept the default organization, or if you set up more, choose the one you want.

Change the 3D to 2D, since this is a 2D game. We will not add any asset packages yet. This can be done later anyway. Go ahead and leave UNITY Analytics enabled—that seems harmless at worst. Finally, click the blue **Create Project** button and wait a minute for the project to be created and the UNITY Editor, opens. The screen will look something like Figure 2.12.

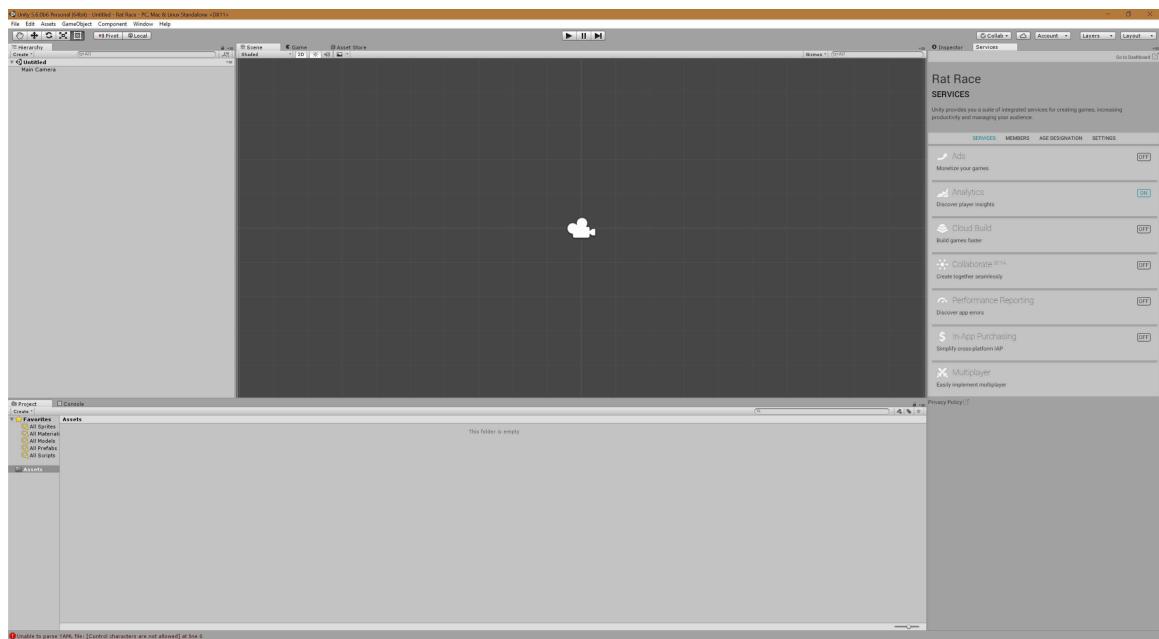


FIGURE 2.12. Screenshot of Initial UNITY Scene.

**2.5.1. Brief Tour of UNITY.** Let us look at the default layout. We describe everything briefly, but some of these things we shall cover again as they are needed, and with more detail. At the top is the menu bar with the **File** menu, **Edit** menu, **Assets** menu, and so on. Just below that is the header bar. On the left side of the header bar are a hand icon, four arrows icon, two curved arrows icon, four diagonal arrows icon, and a box

icon. These are the manipulators. The box icon is the 2D rectangle manipulator for 2D and User Interface (UI) elements, while the four arrows (up, down, right, and left) icon is the position manipulator for moving objects around. The hand is the “pan” manipulator for panning the screen around. The curved arrows are for rotating objects, and the diagonal arrows are for scaling (resizing) objects.

Just to the right you see **Pivot** and **Local** buttons (by default). The first button says which part of an object is “focused” for rotation, and so on. If you click it, it changes to **Center**. Click it again to set it back to **Pivot**. Every object has a pivot point, defined when the object was created or imported, but it might be different from the center point, which is defined by its geometry. The **Local** button tells which space coordinate axes are in: local (centered on the object) or global (centered at the world origin point which may or may not be near the object).

Toward the middle is a right-pointing triangle (**Play**), a double vertical bar (**Pause**), and a right-pointing triangle against a vertical bar (**Step**). The **Play** button runs the game. You can press it now, but the “game” won’t do much yet. It may take a few seconds the first time to set things up, then an empty blue game screen will appear, and the **Play** button will turn blue to show that the game is playing. Press the **Play** button again to stop. The **Pause** button will pause or resume a running game without stopping it. If the game is paused, the **Step** button steps forward one frame. This is sometimes useful for debugging a game.

All the way right are more buttons: **Collab**, a cloud icon, **Account**, **Layers**, and **Layout**. We do not use these much.

Then, the remainder of the editor window is divided into panels. The top left panel is the **Hierarchy** panel, showing every object in our current scene. The top middle panel is the one we use most, showing the scene. Currently, it only has the **Camera** (the box symbol represents a game object) in it. Notice it has tabs at the top, so you can select a tab to switch from the **Scene** tab to either the **Game** tab or the **Asset Store** tab.

The top right panel currently shows the **Services** tab, but there is also an **Inspector** tab which we use more. Go ahead and click the **Inspector** tab to bring that up. Whenever an object is selected in the **Hierarchy**, the **Inspector** panel shows details for that object.

Then, at the bottom is the **Project** window, which will show all assets used in the game (currently, there are none). There are tabs at the top here as well, and the other tab is the **Console**, which is useful when testing a game. Log entries from a game are printed onto the console, and error messages show up there too.



It is possible to change the layout by dragging the lines between panels around to resize panels, or right-click the tab of a panel to close the tab or add a new tab. If a tab is closed by accident, it can be reopened through the **Window** menu in the menu bar, and selecting the name of the panel to open.

It is also possible to drag tabs around, to move a panel to another location, or even undock it completely to make a new window. If you do this accidentally and want to undo it, just click on the tab at the top of the window and drag it back into the main window.

**EXERCISE 2.15.** First take a screenshot of the current layout of Unity and save it. It is probably similar to what is shown in this book, so this step is really optional. On a PC, a screenshot of the selected window is taken with **ctrl+ Alt+ PrtScr**, and then opening GIMP and creating a new image from the clipboard using **Create > From Clipboard...**. Alternatively, use **Win + PrtScr** to automatically save the whole screen to a file. On a Mac, use **↑ + ⌘ + 4** and drag-select a rectangle bounding the image you want. It will be saved as a .png file on your desktop.

Then, rearrange the layout at random: close some tabs, resize some panels, add some new panels, close and reopen some in different locations, undock some even, and make a real mess of things.

Finally, restore the layout to how it looks in the screenshot. If you get stuck, remember you can click the **Layouts** button and select “reset”, but it is better to learn to do this yourself, because you will often need to change layouts on purpose, and sometimes you will do so accidentally and need to fix it.

**2.5.2. Set Up a New Scene.** With UNITY open, switch from the **Scene** tab to the **Game** tab (with a PAC-MAN icon) in the top middle window (in the default layout). There are changeable options at the top of the **Game** panel: **Display 1**, **Free Aspect** (by default), and **Scale**, to the left, and **Maximize on Play**, **Mute Audio**, **Stats**, and **Gizmos**, and a downward-pointing triangle on the right. We are interested in the **Free Aspect**. We wish to change this into a portrait mode standard definition game screen, which has ratio 3 to 4. If you click, you see **4:3** is an option but (by default) “3:4” is not. But you can click the plus sign on a gray disk to add a new option.

**TASK 2.16.** In the **Game** tab, click the button that currently says **Free Aspect**. Click the plus sign to create a new setting. Make its label “3:4”. Change the type from Fixed Resolution to Aspect Ratio. Finally, make the width 3 and the height 4.



There is now a new option in the **Aspect** button called “3:4”. Select the  aspect, and the blue portion of the Game window will change shape to be tall but narrow. Now, optionally, you can adjust the **Hierarchy** and **Inspector** to make them a little wider: you have the space to do that now. Click the **Scene** tab (with the grid icon) to leave the **Game** panel and return to the **Scene** panel.

Let us now save the scene before continuing, just to make that a good habit: **File**  **Save Scenes**. The first time you save a scene, it will ask you for a name. Call it “Game” for now (we shall create other scenes later). Also for now, keep the default folder (the “Assets” folder top level). We shall consider organization of assets later.

**2.5.3. Orthographic Camera and 3D Coordinates.** In the **Hierarchy** panel to the left, select the  **Main Camera** that was put into the scene by default. At least two things will be noticeable when you do this: there is a Camera Preview box in the **Scene** panel (currently showing nothing but a field of blue), and the **Inspector** panel is now populated with the camera’s properties. It is this **Inspector** we are interested in now.

The objects in the **Hierarchy** (except the “Game” scene object with a little UNITY Cube icon next to it) are “GameObjects” (UNITY uses just one word for it: **GameObject**). When you selected the only game object in the scene the **Inspector** panel showed the “Components” of the object: the **Transform** component, the **Camera** component (every  **Camera** game object has a **Camera** component, a reuse of terminology that takes getting used to), a **GUI Layer**, a **Flare Layer**, and an **Audio Listener**. We shall be mainly interested in the **Transform** and **Camera** components for now.

Each component then has several “Properties”, some of which have values that the user can change. For example, the **Transform** component has **Position**, **Rotation**, and **Scale** properties, each of which has an **X**, a **Y**, and a **Z** field. The values are floating point numbers shown in the boxes. These can be edited.

By default, the camera is at position  $x = 0$ ,  $y = 0$ , and  $z = -10$ , or more succinctly,  $(0, 0, -10)$ . Note the world origin (mentioned before) is at  $(0, 0, 0)$ . Thus, the camera is ten units away from the world origin, in the negative  $z$  direction. What does this mean in terms of the geometry of the scene?

The first thing to note is that there are three coordinates even though we selected a 2D game when creating the project. This is because UNITY is a 3D program. 2D work is essentially fit into a 3D framework.

UNITY uses the same left handed coordinate standard found on most graphics cards. The  $x$  direction is from left to right (in particular,  $(-2, 0, 0)$  is



two units to the left of the origin, and (3,0,0) is three units to the right of the origin). The  $y$  direction is from bottom to top (so (0,5,0) is five units above the origin, for example). The  $z$  coordinate is front to back, that is from the position of the user sitting in front of the monitor toward a point behind the monitor. One can imagine the monitor screen itself being at  $z = 0$ . The camera, as we saw already, is at  $z = -10$ , so it can be imagined sitting 10 units in front of the monitor, perhaps where the user is sitting. (How big is a unit? In the game in 3D mode, a unit is often taken as a meter, but in 2D mode it is all “shrunk” (the word is actually “projected”) onto a flat monitor, so if you try to map units on the  $z$  axis out of the screen into the real world, it really depends on the zoom level of the camera lens.)

The rotation values show the amount of rotation, in degrees, about the  $x$  axis, the  $y$  axis, and the  $z$  axis. (These are called Euler Angles. Remembering that will help one search for methods in C# to work with them.) This is (0,0,0) for the camera, so the camera is unrotated, in its default position pointing toward the back of the monitor (in the same direction as the user’s gaze).

The scale values show how much the camera has been scaled along each axis. These are all 1, so the camera is unscaled (scale being a multiplicative factor, not an additive quantity, has 1 for its identity, not 0). Camera scale does not actually mean much, because one never sees the camera in the game, so it is usually left unchanged from the default.

Is this really where we want the camera? It turns out, no. We discuss this next.

**2.5.4. Game Coordinates.** We shall build the game on a grid, 48 squares wide by 64 squares tall (which gives the 3:4 ratio). We discuss why later. We would like the (0,0) square (these will be game coordinates) to be at the bottom left of the camera’s field of view, and therefore the (47,63) square will be at the top right. We actually want (0,0, $z$ ), in world-space coordinates to be the *center* of the lower-left square, and (47,63, $z$ ) to be the *center* of the upper-right square. Thus, the center of the camera should be aimed at the arithmetic mean (average):  $(\frac{47}{2}, \frac{63}{2}, z)$ . We shall keep  $z$  at  $-10$  so the camera is 10 units in front of the grid.

Therefore, we need to set the Transform component’s Position property to  $x = 23.5$ ,  $y = 31.5$ , and  $z = -10$ .

**TASK 2.17.** With the `Main Camera` game object selected in the `Hierarchy` panel, adjust the `X`, `Y`, and `Z` fields of the `Position` property of the `Transform` component to have these values,  $x = 23.5$ ,  $y = 31.5$ , and  $z = -10$ . Leave the other values unchanged. The camera may disappear off the `Scene` view when you do this. You can find it



again by panning (right-click and drag) the view, or just double-click the `Main Camera` game object in the `Hierarchy` to center the view on the `Main Camera`. Optionally, also use the scroll wheel on the mouse to zoom in and out.

We are not done. The camera is positioned right, but we need to set the field of view. In a real camera, this would involve choosing a lens and adjusting the focus and zoom. In **UNITY**, we adjust the properties of the `Camera` component.

**TASK 2.18.** First, the `Projection` property should already be set to “Orthographic”. If not, make the change. The choices are Perspective or Orthographic. The former is for 3D games (though some older top-down games used an orthographic camera) and shows the view in proper perspective with far objects appearing at a smaller scale than near objects. Orthographic cameras don’t actually exist in real life (but can be approximated with a very expensive lens having a very, very long focal length while keeping a wide field of view) but with an orthographic camera, objects are not scaled by distance. A one meter cube fills the same amount of screen space whether it is 1 meter away or 1000 meters away. This is what we want with a 2D game.

The other property we need to look at is the Orthographic Size, just shown as `Size` in the `Inspector` panel. This is equal to the half-height of the camera in game coordinates (called Game Units). Since our grid is 48x64, the height of the grid is 64, and the half-height is 32. So, we set the `Size` property of the `Camera` component to 32. Now, the camera field of view (shown as a white rectangle in the `Scene` view; you may have to zoom out to see it) will exactly cover the grid.

The remaining properties can be left at their default values.

If we were to place a sprite at (0, 0) (and it doesn’t matter what the `Z` is as long as it is greater than -10 or it will not be visible because it would be behind the camera), it will be placed on the lower-left grid square. If you place multiple objects there, those with lower `z` values will be in front of those with higher `z` values and will either partly or completely obscure those behind them.

### 2.5.5. Importing Assets.

**TASK 2.19.** Open up a Windows EXPLORER or Macintosh FINDER window and navigate to the folder where the prototype shapes were stored: Circle.png, Diamond.png, Hex\_1\_1.png, HexRegular.png, OctRegular.png, Rectangle2h.png, RectangleGolden.png, Square.png, TriangleEqui.png, TriangleIso.png, and TriangleRight.png. Also make sure the



UNITY window is visible with the **Project** panel open to the **Assets** top level. Select these 11 files and drag them into the **Assets** area. All 11 should then automatically be imported into UNITY and get icons that look like thumbnails of the shapes.

There is still some work to do. Select each of the 11 new assets, one at a time, in the **Project** panel. As each is selected, import settings for the asset will appear in the **Inspector**. Find the **Pixels Per Unit** property under the **Sprite Mode**. It defaults to 100. Change it to 256. Do the same for all of the 11 new sprite assets. A shortcut is to select all of the 11 sprites (**ctrl**-click or on a Mac, **⌘**-click), but be sure nothing else is selected. Then when you change **Pixels Per Unit** in the **Inspector**, all 11 are changed. Click the **Apply** button at the bottom right of the properties shown in the **Inspector** to save the change. Then click a few of the shapes individually to confirm the change went through and that the **Apply** button is now grayed out.

This means, 256 pixels of the sprite will correspond to one game unit. All the sprites will be exactly one grid square wide. We can still scale the sprites individually as they are placed in the game, but we want a reasonable default to start with.

**EXERCISE 2.20.** UNITY now has the ability to create polygons built in. Google the documentation and use this to recreate some of the sprites already created, or see if you can find an easier way to make a regular pentagon, solid white, 256 units wide.

### 2.5.6. Creating a Prototype Maze.

**TASK 2.21.** Let us start building the prototype game. We shall use the isosceles triangle for the player and the hex\_1\_1 for the enemy. Go ahead and drag these into the **Hierarchy** panel. Then select each one in the **Hierarchy** panel and adjust their **Transform** component properties: the **X** and **Y** scale of each should be 3 (we want the player and enemy to be three grid squares wide), but the **Z** scale does not matter as long as it is nonzero. Set the position of the triangle to (2,2,0), and of the hexagon to (45,61,0), thus putting them at opposite corners.

Next, right-click on the **TriangleIso** sprite in the **Hierarchy** and select **Rename**. Change its name to “Player”. Then, right-click on the **Hex\_1\_1** sprite in the **Hierarchy** and select **Rename**. Change its name to “Enemy”.

For each of the two sprites in the **Hierarchy**, look at their **Sprite Renderer** components. For the triangle, click on the colored bar next to the **Color** property, and use the color chooser to change it to



yellow (Hex FFFF00FF). Close the color chooser when done. Also, follow the same steps to change the hexagon's color to red (Hex FF0000FF).

Note that in UNITY the hex values for colors have 8 digits, not 6 like GIMP. GIMP's hex value has two digits each for the hexadecimal values of red, green, and blue. UNITY has these three as well, as well as "alpha" which is the opacity: 00 means invisible, FF means completely opaque, and values in between are different levels of transparency. Thus, we set the triangle to all red and green with no blue and full alpha, which is yellow (red and green light mix to make yellow light), and the hexagon was full red, full alpha, but no green and blue. With practice, one can learn to visualize what hex values correspond to what colors. The color chooser lets you choose visually as well as by hex values, to make it easier.

**EXERCISE 2.22.** Try to guess what colors these (6 digit, ignoring alpha-opacity) hex values are. Confirm them with GIMP or UNITY: 000000, ffffff, 808080, ff0000, ff00ff, 00ffff, ff8000, and 80ff00. Try to guess a hex value that would make each of the following colors, and confirm with GIMP or UNITY: pink, reddish purple, sky blue, dark gray, and yellowish orange. You can also Google these colors and try to find their "official" hex values.

**TASK 2.23.** Next, put a square into the Hierarchy. Change its coordinates so that they are integers. Now, if you look at the manipulator buttons at the top left of the screen and select the four arrows pointing up, down, left, and right, the "position" manipulator, you can then move the sprite around by dragging the arrows (to drag horizontally or vertically), or the little square joining the arrows (to drag in anywhere on the plane). Hold down the **ctrl** (Mac: ) key while dragging it will keep it on integer values, which is what we want. Put the block anywhere you want that way, but make sure the coordinates are integers.

With the block still selected, hit **ctrl+D** ( +D on Mac) to duplicate it. **ctrl**-drag to position it. Do this multiple times and make part of a maze. Don't bother making a complete maze, this is just for illustration.

We shall eventually use a Procedural Content Generation (PCG) maze, that is, one generated automatically. Execute **File** **Save Scenes** to save your work.

We now know how to put objects into a scene to make games. See Figure 2.1 for an idea of what the screen should look like now (though the walls might be in different places). If you play the game now, you will see that prototype maze, but it will not do anything. We address this in a later section.



## CHAPTER 10

## Designing a Game

Before improving this prototype, we shall “go back to the beginning”, so to speak, and actually consider the design of this game. When working on a large project, especially if there is more than one person, it is necessary to have a design in mind to focus the coding, testing, and asset creation. The design is simpler than the actual project, so that it can be changed easily as needed. In particular, it is easier to change the design than to change the working code. Yet the design should be clear enough that different people working on different parts, or even the same person working on different parts at different times, can create things that fit together well.

Like with building the game and learning the techniques, a spiral approach is done, adding more detail to the design and improving the overall design in multiple iterations.

The first iteration of the design has already been done, sort of, in Chapter 1, “Let’s Create a Game!” Taking this chapter and “tightening it up” a bit and reorganizing it is a good first iteration of our design.

The next section is our first attempt at a game design document. The level of detail is not all that fine, but it should cover most of what the game entails. Except for possibly the first paragraph, which is a marketing-type paragraph, it is intended mainly for those building the game, not for those playing the game.

The “elevator pitch” is what you say to someone on the elevator who asks you what kind of game you are making. It is a very short description of what you are working on. Every employee should have an elevator pitch: if you end up on an elevator with the CEO who asks you what you are doing, it looks bad to say, “like, I don’t know, nothing?”

The format of the Game Design Document is not set in stone. This is but one way it can be done. Googling for “Game Design Document” will get you plenty of examples to look at. Some documents for big name studios producing big name games are very detailed.

### 10.1. Game Design Document I

**10.1.1. Marketing Statement.** Remember the Golden Age of the arcade, around the 1980s or so? It was a PacMan-eat-dot industry, where



a game that didn't earn a quarter every three minutes got removed and replaced. The games were built to have lights and sounds to attract the player, and then to challenge the player enough that he's likely to lose the game...fast. Still, from time to time, a player would do really well and get on the high score list.

We shall build a game of this type.

**10.1.2. Elevator Pitch.** The player controls a mouse who must eat all the cheese in the maze without being eaten by some very aggressive cats.

**10.1.3. Technology.** We shall use the UNITY game engine, whose software development editor and integrated development environment (IDE) runs on the PC and Macintosh, and can deploy games than run on the PC and Mac as well as the web and mobile devices and Linux.

We shall use Git for source control.

#### 10.1.4. Features.

- 2D classic arcade action
- Written with UNITY 5.6 and C#
- Artificial Intelligence (AI) for the cats
- Grid-based automated level building and Procedural Content Generation (PCG)
- Score keeping with high-score saving
- Multiple lives
- Teleportation tunnels
- Sound and music and animated sprite assets
- A game state system managing the title and high score screens
- Cutscenes
- Published to multiple locations for multiple platforms
- and more...

#### 10.1.5. Mechanics.

- Player is controled by a keyboard, joystick, game pad, or (for mobile devices) a touch screen, with inputs representing up, down, left, and right. An input will change the direction of the player sprite (the mouse) to that direction if it is possible to go that direction. Upon reaching an obstacle, the sprite will stop until a valid directional input is given.
- The enemy sprites (cats) will be controlled by AI and will try to catch the mouse. If the mouse is caught, a life is lost. Initially, the mouse has three lives but new lives are sometimes given as a bonus. When all lives are lost, the game is over.



- The mouse must eat all the dots (cheese) in the maze, at which time a newer, harder maze is given (levelling up).
- Special “growth hormone” cheese gives the mouse the ability to chase and eat the cats for a short period of time.
- Tunnels allow the mouse (or cats) to teleport, thus confusing the AI of the cats.
- Scoring is based on cheese eaten, cats eaten, bonus pickups eaten, and levels achieved. Top scores are saved.

#### **10.1.6. Attract Mode.**

- When the game is not in progress, it cycles through a title screen, a high score screen, a details screen, and a demo screen. The demo is a video clip of sample gameplay.
- An options button, leading to an options screen, is available. User-selectable options include music volume, sound effects volume, and perhaps other options.
- Attract mode ends when a “Play” button is pressed, and resumes on end of game (perhaps with a high score screen interlude).

**10.1.7. Assets.** Assets include an animated mouse, animated cats, cheese (dots), walls pieces, maze designs, sound effect clips, and level music.

#### **10.1.8. Ideas for Future Development.**

- Adrenaline: start with high reserve, low usage. If mouse is near cat, can use it to go faster, but reserves are depleted and mouse slows down. Need a “run key” (or maybe double-tap direction key) to signal “use the adrenaline now”
- Eating cats makes mouse full when growth hormone runs out and he walks slower for a time
- Cat seeing mouse directly ahead gets excited and runs faster, but tires out quickly and time needed to have speed burst again
- Mouse eating growth hormone cheese makes cats faster in running away in fear for a short interval of time
- Special animation frames: mouse sniffs nearby cat and looks afraid, cat sniffs nearby mouse and looks hungry, etc.
- Special pickups for special powers?
- Dangerous “pickups” like a marching mousetrap?



## Bibliography

- [AA15a] Joseph Albahari and Ben Albahari, *C# 6.0 in a nutshell: The definitive reference*, 6th ed., O'Reilly Media, Inc., 2015.
- [AA15b] \_\_\_\_\_, *C# 6.0 pocket reference: Instant help for c# 6.0 programmers*, 1st ed., O'Reilly Media, Inc., 2015.
- [Car94] O.S. Card, *Ender's game*, The Ender Quintet, Tom Doherty Associates, 1994.
- [CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson, *Introduction to algorithms*, 2nd ed., McGraw-Hill Higher Education, 2001.
- [Fos84] A.D. Foster, *The last starfighter*, Berkley Books, 1984.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [Gre14] Jason Gregory, *Game engine architecture, second edition*, 2nd ed., A. K. Peters, Ltd., Natick, MA, USA, 2014.
- [McW83] N. McWhirter, *Guinness 1983 book of world records*, Bantam Books, 1983.
- [RN03] Stuart J. Russell and Peter Norvig, *Artificial intelligence: A modern approach*, 2 ed., Pearson Education, 2003.

