

pseudocode edition

2D Game Development: From Zero to Hero

**A compendium of the community
knowledge on game design and development**

2D Game Development: From Zero To Hero (pseudocode edition, version v0.5.6) is licensed under the CC-BY-NC 4.0 license.

This book can be found in the following official repositories:

- https://github.com/Penaz91/2DGD_F0TH/
- https://gitlab.com/Penaz/2dgd_f0th

Perseverance is the backbone of success.

Anonymous

To my family
To my friends, both international and not
To whom never gives up
Daniele Penazzo

Contents

1	Foreword	1
2	Introduction	2
2.1	Why another game development book?	2
2.2	Conventions used in this book	2
2.2.1	Logic Conventions	2
2.2.2	Code Listings	3
2.2.3	Block Quotes	3
2.3	Structure of this Book	3
3	The Maths Behind Game Development	5
3.1	The modulo operator	5
3.2	Vectors	5
3.2.1	Adding and Subtracting Vectors	5
3.2.2	Scaling Vectors	6
3.2.3	Dot Product	7
3.2.4	Vector Length and Normalization	7
3.3	Matrices	8
3.3.1	What is a matrix	8
3.3.2	Matrix sum and subtraction	8
3.3.3	Multiplication by a scalar	8
3.3.4	Transposition	8
3.3.5	Multiplication between matrices	9
3.3.6	Other uses for matrices	10
3.4	Trigonometry	10
3.4.1	Radians vs Degrees	11
3.4.2	Sine, Cosine and Tangent	11
3.4.3	Pythagorean Trigonometric Identity	12
3.4.4	Reflections	12
3.4.5	Shifts	12
3.4.6	Trigonometric Addition and subtraction	13
3.4.7	Double-Angle Formulae	13
3.4.8	Inverse Formulas	13
3.5	Coordinate Systems on computers	14
3.6	Transformation Matrices	15
3.6.1	Stretching	15
3.6.2	Rotation	16
3.6.2.1	Choosing the direction of the rotation	16
3.6.2.2	Rotating referred to an arbitrary point	16
3.6.3	Shearing	16
4	Some Computer Science Fundamentals	18
4.1	De Morgan's Laws and Conditional Expressions	18
4.2	Estimating the order of algorithms	18
4.2.1	$O(1)$	19
4.2.2	$O(\log(n))$	19
4.2.3	$O(n)$	19
4.2.4	$O(n \cdot \log(n))$	20
4.2.5	$O(n^2)$	20
4.2.6	$O(2^n)$	20
4.3	A primer on calculating the order of your algorithms	20

4.3.1	Some basics	20
4.3.2	What happens when we have more than one big-O?	21
4.3.3	What do we do with recursive algorithms?	22
4.3.4	How do big-O estimates compare to each other?	23
4.4	Simplifying your conditionals with Karnaugh Maps	24
4.4.1	“Don’t care”s	24
4.4.2	A more complex map	26
4.4.3	Guided Exercise	27
4.5	Object Oriented Programming	28
4.5.1	Introduction	28
4.5.2	Objects	28
4.5.3	Abstraction and Interfaces	29
4.5.4	Inheritance and Polymorphism	29
4.5.5	The Diamond Problem	29
4.5.6	Composition	30
4.5.7	Coupling	30
4.5.8	The DRY Principle	30
4.5.9	SOLID Principles	31
4.5.10	“Composition over Inheritance” design	31
4.6	Designing entities as data	31
4.7	Reading UML diagrams	32
4.7.1	Use Case Diagrams	32
4.7.1.1	Actors	33
4.7.1.2	Use Cases	33
4.7.1.3	Notes	35
4.7.1.4	Sub-Use Cases	35
4.7.2	Class Diagrams	35
4.7.2.1	Classes	35
4.7.2.2	Relationships between classes	36
4.7.2.3	Notes	38
4.7.2.4	Interfaces	38
4.7.3	Activity Diagrams	39
4.7.3.1	Start and End Nodes	39
4.7.3.2	Actions	39
4.7.3.3	Decisions (Conditionals) and loops	40
4.7.3.4	Synchronization	41
4.7.3.5	Signals	41
4.7.3.6	Swimlanes	42
4.7.3.7	Notes	42
4.7.3.8	A note on activity diagrams	43
4.8	Generic Programming	43
4.9	Advanced Containers	43
4.9.1	Dynamic Arrays	44
4.9.1.1	Performance Analysis	44
4.9.2	Linked Lists	46
4.9.2.1	Performance Analysis	46
4.9.3	Doubly-Linked Lists	48
4.9.4	Hash Tables	48
4.9.5	Binary Search Trees (BST)	50
4.9.6	Heaps	51
4.9.7	Stacks	51
4.9.8	Queues	52

4.9.9	Circular Queues	53
4.10	Introduction to MultiTasking	54
4.10.1	Co-Routines	54
4.11	Introduction to MultiThreading	54
4.11.1	What is MultiThreading	54
4.11.2	Why MultiThreading?	55
4.11.3	Thread Safety	55
4.11.3.1	Race conditions	55
4.11.3.2	Critical Regions	57
4.11.4	Ensuring determinism	57
4.11.4.1	Immutable Objects	57
4.11.4.2	Mutex	58
4.11.4.3	Atomic Operations	58
5	Project Management Basics and tips	59
5.1	The figures of game design and development	59
5.1.1	Producer/Project Manager	59
5.1.2	Game Designer	59
5.1.3	Writer	59
5.1.4	Developer	59
5.1.5	Visual Artist	60
5.1.6	Sound Artist	60
5.1.7	Tester	60
5.2	Some generic tips	60
5.2.1	Be careful of feature creep	60
5.2.2	On project duration	60
5.2.3	Brainstorming: the good, the bad and the ugly	61
5.2.4	On Sequels	61
5.3	Common Errors and Pitfalls	61
5.3.1	Losing motivation	61
5.3.2	The “Side Project” pitfall	61
5.3.3	Making a game “in isolation”	62
5.3.4	Mishandling Criticism	62
5.3.5	Not letting others test your game	62
5.3.6	Being perfectionist	62
5.3.7	Using the wrong engine	63
5.4	Software Life Cycle Models	63
5.4.1	Iteration versus Increment	63
5.4.2	Waterfall Model	63
5.4.3	Incremental Model	64
5.4.4	Evolutionary Model	64
5.4.5	Agile Software Development	65
5.4.5.1	User Stories	66
5.4.5.2	Scrum	66
5.4.5.3	Kanban	66
5.4.5.4	ScrumBan	67
5.4.6	Where to go from here	67
5.5	Version Control	67
5.6	Metrics and dashboards	68
5.6.1	Cyclomatic Complexity	68
5.6.2	SLOC	68
5.6.3	Code Coverage	68

5.6.4	Code Smells	68
5.6.5	Coding Style breaches	68
5.7	Continuous Integration and Delivery	68
6	Introduction to Game Design	69
6.1	Platforms	69
6.1.1	Arcade	69
6.1.2	Console	69
6.1.3	Personal Computer	69
6.1.4	Mobile	70
6.1.5	Web	70
6.2	Input Devices	70
6.2.1	Mouse and Keyboard	70
6.2.2	Gamepad	70
6.2.3	Touch Screen	70
6.2.4	Dedicated Hardware	71
6.2.5	Other Input Devices	71
6.3	Game Genres	71
6.3.1	Shooters	71
6.3.2	Strategy	71
6.3.3	Platformer	71
6.3.4	RPG	72
6.3.5	MMO	72
6.3.6	Simulation	72
6.3.7	Rhythm Games	72
6.3.8	Visual novels	72
7	Writing a Game Design Document	73
7.1	What is a Game Design Document	73
7.2	Possible sections of a Game Design Document	73
7.2.1	Project Description	73
7.2.2	Characters	73
7.2.3	Storyline	74
7.2.3.1	The theme	74
7.2.3.2	Progression	74
7.2.4	Levels and Environments	75
7.2.5	Gameplay	75
7.2.5.1	Goals	75
7.2.5.2	Game Mechanics	75
7.2.5.3	Skills	76
7.2.5.4	Items/Powerups	76
7.2.5.5	Difficulty Management and Progression	76
7.2.5.6	Losing Conditions	77
7.2.6	Graphic Style and Art	77
7.2.7	Sound and Music	77
7.2.8	User Interface	78
7.2.9	Game Controls	78
7.2.10	Accessibility Options	78
7.2.11	Tools	79
7.2.12	Marketing	79
7.2.12.1	Target Audience	79
7.2.12.2	Available Platforms	79

7.2.12.3	Monetization	80
7.2.12.4	Internationalization and Localization	80
7.2.13	Other/Random Ideas	81
7.3	Where to go from here	81
8	The Game Loop	82
8.1	The Input-Update-Draw Abstraction	82
8.2	Input	82
8.2.1	Events vs Real Time Input	82
8.3	Timing your loop	83
8.3.1	What is a time step	83
8.3.2	Fixed Time Steps	83
8.3.3	Variable Time Steps	83
8.3.4	Semi-fixed Time Steps	84
8.3.5	Frame Limiting	84
8.3.6	Frame Skipping/Dropping	85
8.3.7	Multithreaded Loops	85
8.4	Issues and possible solutions	85
8.4.1	Frame/Screen Tearing	85
8.5	Drawing to screen	86
8.5.1	Clearing the screen	86
9	Collision Detection and Reaction	88
9.1	Collision Detection: Did it really collide?	88
9.1.1	Collision Between Two Points	88
9.1.2	Collision Between A Point and a Circle	88
9.1.3	Collision Between Two Circles	89
9.1.4	Collision Between Two Axis-Aligned Rectangles (AABB)	90
9.1.5	Line/Point Collision	92
9.1.6	Line/Circle Collision	93
9.1.7	Point/Rectangle Collision	95
9.1.8	Point/Triangle Collision	95
9.1.9	Circle/Rectangle Collision	96
9.1.10	Line/Line Collision	97
9.1.11	Line/Rectangle Collision	99
9.1.12	Point/Polygon Collision	99
9.1.13	Circle/Polygon Collision	99
9.1.14	Rectangle/Polygon Collision	100
9.1.15	Line/Polygon Collision	100
9.1.16	Polygon/Polygon Collision	100
9.1.17	Pixel-Perfect collision	100
9.1.18	Multi-pass collision detection	101
9.2	Finding out who hit what	101
9.2.1	The Brute Force Method	101
9.2.2	Building Quad Trees	102
9.2.3	Calculating the position of tiles	103
9.3	Collision Reaction/Correction	106
9.3.1	HitBoxes vs HurtBoxes	106
9.3.2	Collision Reaction Methods	107
9.3.2.1	The Direction + Velocity Method	107
9.3.2.2	Shallow-axis based reaction method	109
9.3.2.3	The “Snapshot” Method	109

9.3.2.4	The “Tile + Offset” Method	110
9.4	Common Issues with Collision Detection	110
9.4.1	The “Bullet Through Paper” problem	110
9.4.2	Precision Issues	110
10	Cameras	112
10.1	Screen Space vs. Game Space	112
10.2	Most used camera types	113
10.2.1	Static Camera	113
10.2.2	Grid Camera	113
10.2.3	Position-Tracking Camera	113
10.2.3.1	Horizontal-Tracking Camera	113
10.2.3.2	Full-Tracking Camera	113
10.2.4	Camera Trap	114
10.2.5	Look-Ahead Camera	115
10.2.6	Hybrid Approaches	115
10.3	Clamping your camera position	115
11	Game Design	117
11.1	Tutorials	117
11.1.1	Do not pad tutorials	117
11.1.2	Integrate tutorials in the lore	117
11.1.3	Let the player explore the controls	117
11.2	Consolidating and refreshing the game mechanics	118
11.2.1	Remind the player about the mechanics they learned	118
11.2.2	Introduce new ways to use old mechanics	118
11.3	Rewarding the player	118
11.3.1	Reward the player for their “lateral thinking”	118
11.3.2	Reward the player for their tenacity	119
11.3.3	Reward the player for exploring	120
11.3.4	Reward the player for not immediately following the given direction	120
11.3.5	Reward the player for not trusting you entirely	120
11.4	Tips and Tricks	121
11.4.1	General Purpose	121
11.4.1.1	Make that last Health Point count	121
11.4.1.2	Avoiding a decision can be a decision itself	121
11.4.1.3	Telegraphing	121
11.4.2	Shooters	122
11.4.2.1	Make the bullets stand out	122
12	Creating your resources	123
12.1	Graphics	123
12.1.1	Some computer graphics basics	123
12.1.1.1	Color Depth	123
12.1.1.2	True Color vs. Indexed Color	123
12.1.1.3	Lossless Formats	123
12.1.1.4	Lossy Formats	123
12.1.1.5	Transparency	124
12.1.2	General Tips	124
12.1.2.1	Practice, Practice, Practice...	124
12.1.2.2	References are your friends	124
12.1.2.3	Don’t compare your style to others’	124
12.1.2.4	Study other styles	125

12.1.2.5	Learn to deconstruct objects into shapes	125
12.1.3	Sprite sheets	125
12.1.4	Virtual Resolution	126
12.1.5	Using limited color palettes	128
12.1.6	Dithering	128
12.1.7	Layering and graphics	129
12.1.7.1	Detail attracts attention	129
12.1.7.2	Use saturation to separate layers further	129
12.1.7.3	Movement is yet another distraction	129
12.1.7.4	Use contrast to your advantage	130
12.1.7.5	Find exceptions	130
12.1.8	Palette Swapping	130
12.1.9	Pixel Art	130
12.1.9.1	What pixel art is and what it is not	130
12.1.9.2	Sub-pixel animation	130
12.1.10	Tips and Tricks	131
12.1.10.1	Creating “Inside rooms” tilesets	131
12.2	Sounds And Music	132
12.2.1	Some audio basics	132
12.2.1.1	Sample Rate	132
12.2.1.2	Bit Depth	133
12.2.1.3	Lossless Formats	133
12.2.1.4	Lossy Formats	133
12.2.1.5	Clipping	133
12.2.2	Digital Sound Processing (DSP)	134
12.2.2.1	Reverb	134
12.2.2.2	Pitch Shift	134
12.2.2.3	Filtering	134
12.2.2.4	Doppler Effect	135
12.2.3	“Swappable” sound effects	135
12.2.4	Some audio processing tips	135
12.2.4.1	Prefer cutting over boosting	135
12.3	Fonts	135
12.3.1	Font Categories	135
12.3.1.1	Serif and Sans-Serif fonts	136
12.3.2	Proportional and Monospaced fonts	136
12.3.3	Using textures to make text	137
12.3.4	Using Fonts to make text	137
12.4	Shaders	137
12.4.1	What are shaders	137
12.4.2	Shader Programming Languages	137
12.4.3	The GLSL Programming Language	137
12.4.4	Some GLSL Shaders examples	137
13	Procedural Content Generation	138
13.1	What is procedural generation (and what it isn’t)	138
13.2	Advantages and disadvantages	139
13.2.1	Advantages	139
13.2.1.1	Less disk space needed	139
13.2.1.2	Larger games can be created with less effort	139
13.2.1.3	Lower budgets needed	139
13.2.1.4	More variety and replayability	139

13.2.2	Disadvantages	139
13.2.2.1	Requires more powerful hardware	139
13.2.2.2	Less Quality Control	140
13.2.2.3	Worlds can feel repetitive or “lacking artistic direction”	140
13.2.2.4	You may generate something unusable	140
13.2.2.5	Story and set game events are harder to script	140
13.3	Where it can be used	140
14	Useful Patterns, Containers and Classes	142
14.1	Design Patterns	142
14.1.1	Singleton Pattern	142
14.1.2	Command Pattern	143
14.1.3	Flyweight	144
14.1.4	Observer Pattern	145
14.1.5	Strategy	147
14.1.6	Chain of Responsibility	148
14.1.7	Component/Composite Pattern	149
14.2	Resource Manager	150
14.3	Animator	151
14.4	Finite State Machine	151
14.5	Menu Stack	152
14.6	Particle Systems	152
14.6.1	Particles	152
14.6.2	Generators	152
14.6.3	Emitters	153
14.6.4	Updaters	153
14.6.5	Force Application	153
14.7	Timers	153
14.8	Inbetweening	153
14.9	Chaining	154
15	Artificial Intelligence in Videogames	155
15.1	Path Finding	155
15.1.1	Representing our world	155
15.1.1.1	2D Grids	155
15.1.1.2	Path nodes	156
15.1.1.3	Navigation meshes	156
15.1.2	Heuristics	156
15.1.2.1	Manhattan Distance heuristic	157
15.1.2.2	Euclidean Distance heuristic	157
15.1.3	Algorithms	158
15.1.3.1	The Greedy “Best First” Algorithm	159
15.1.3.2	The Dijkstra Algorithm	160
15.1.3.3	The A* Algorithm	161
15.2	Finite state machines	163
15.3	Decision Trees	163
15.4	Behaviour Trees	163
16	Other Useful Algorithms	164
16.1	World Generation	164
16.1.1	Maze Generation	164
16.1.1.1	Randomized Depth-First Search (Recursive Backtracker)	164
16.1.1.2	Randomized Kruskal’s Algorithm	166

16.1.2	Recursive Division Algorithm	168
16.1.3	Binary Tree Algorithm	171
16.1.4	Eller's Algorithm	173
16.2	Noise Generation	175
16.2.1	Randomized Noise (Static)	175
16.2.2	Perlin Noise	175
17	Developing Game Mechanics	176
17.1	General Purpose	176
17.1.1	I-Frames	176
17.1.2	Scrolling Backgrounds and Parallax Scrolling	176
17.1.2.1	Infinitely Scrolling Backgrounds	176
17.1.2.2	Parallax Scrolling	178
17.1.3	Avoid interactions between different input systems	178
17.1.4	Sprite Stenciling/Masking	178
17.1.5	Loading screens	179
17.2	2D Platformers	179
17.2.1	Simulating Gravity	179
17.2.2	Avoiding "Floaty Jumps"	179
17.2.3	Ladders	181
17.2.4	Walking on slanted ground	181
17.2.5	Stairs	181
17.2.6	Jump Buffering	181
17.2.7	Coyote Time	182
17.2.8	Timed Jumps	183
17.3	Top-view RPG-Like Games	183
17.3.1	Managing height	183
17.4	Rhythm Games	184
17.4.1	The world of lag	184
17.4.1.1	Video Lag	184
17.4.1.2	Audio Lag	184
17.4.1.3	Input Lag	184
17.4.1.4	Lag Tests	184
17.4.2	Synchronizing with the Music	184
17.4.2.1	Time domain vs. Frequency Domain	184
17.4.2.2	The Fast Fourier Transform	184
17.4.2.3	Beat Detection	184
17.5	Match-x Games	184
17.5.1	Managing and drawing the grid	184
17.5.2	Finding and removing Matches	185
17.5.2.1	Why don't we delete the matches immediately?	187
17.5.3	Replacing the removed tiles and applying gravity	187
17.6	Cutscenes	187
17.6.1	Videos	187
17.6.2	Scripted Cutscenes	188
18	Testing your game	189
18.1	When to test	189
18.1.1	Testing "as an afterthought"	189
18.1.2	Test-Driven Development	189
18.1.3	The "Design to test" approach	189
18.1.4	You won't be able to test EVERYTHING	189

18.2	Mocking	189
18.3	Types of testing	190
18.3.1	Automated Testing	190
18.3.2	Manual Testing	190
18.4	Unit Testing	190
18.5	Integration Testing	191
18.6	Regression Testing	191
18.7	Playtesting	191
18.7.1	In-House Testing	191
18.7.2	Closed Beta Testing	191
18.7.3	Open Beta Testing	191
19	Profiling and Optimization	192
19.1	Profiling your game	192
19.1.1	Does your application really need profiling?	192
19.1.1.1	Does your FPS counter roam around a certain “special” value?	192
19.1.1.2	Is the animation of your game stuttering but the FPS counter is fine?	192
19.1.2	First investigations	192
19.1.2.1	Is your game using 100% of the CPU?	192
19.2	Optimizing your game	193
19.2.1	Working on references vs. returning values	193
19.2.2	Optimizing Drawing	193
20	Balancing Your Game	194
20.1	The “No BS” principle	194
20.2	Always favour the player	194
20.3	A primer on Cheating	194
20.3.1	Information-based cheating	195
20.3.2	Mechanics-based cheating	195
20.3.3	Man-in-the-middle	195
20.4	How cheating influences gameplay and enjoyability	195
20.4.1	Single Player	195
20.4.2	Multiplayer	196
20.4.2.1	P2P	196
20.4.2.2	Dedicated Servers	197
21	Marketing your game	199
21.1	The importance of being consumer-friendly	199
21.2	Pricing	199
21.2.1	Penetrating the market with a low price	199
21.2.2	Giving off a “premium” vibe with a higher price	200
21.2.3	The magic of “9”	200
21.2.4	Launch Prices	200
21.2.5	Bundling	200
21.2.6	Nothing beats the price of “Free” (Kinda)	200
21.3	Managing Hype	201
21.4	Downloadable Content	201
21.4.1	DLC: what to avoid	201
21.4.2	DLC: what to do	202
21.5	Digital Rights Management (DRM)	202
21.5.1	How DRM can break a game down the line	203
21.6	Free-to-Play Economies and LootBoxes	203
21.6.1	Microtransactions	203

21.6.1.1	The human and social consequences of Lootboxes	203
21.6.2	Free-to-Play gaming and Mobile Games	204
21.7	Assets and asset Flips	205
21.8	Crowdfunding	205
21.8.1	Communication Is Key	205
21.8.2	Do not betray your backers	206
21.8.3	Don't be greedy	206
21.8.4	Stay on the "safe side" of planning	206
21.8.5	Keep your promises, or exceed them	206
21.8.6	A striking case: Mighty No. 9	206
22	Game Jams	208
22.1	Have Fun	208
22.2	Stay Healthy	208
22.3	Stick to what you know	208
22.4	Hacking is better than planning (But still plan ahead!)	208
22.5	Graphics? Sounds? Music? FOCUS!	209
22.6	Find creativity in limitations	209
22.7	Involve Your Friends!	209
22.8	Write a Post-Mortem (and read some too!)	210
22.9	Most common pitfalls in Game Jams	210
23	Dissecting games: two study cases	211
23.1	A bad game: Hoshi wo miru hito	211
23.1.1	Introduction	211
23.1.2	Balance Issues	211
23.1.2.1	You can't beat starter enemies	211
23.1.2.2	The Damage Sponge	211
23.1.2.3	Can't run away from battles, but enemies can	211
23.1.2.4	Statistics	212
23.1.3	Bad design choices	212
23.1.3.1	You get dropped in the middle of nowhere	212
23.1.3.2	The starting town is invisible	212
23.1.3.3	The Jump Ability	212
23.1.3.4	Items are invisible	212
23.1.3.5	Item management	212
23.1.3.6	Buying Weapons makes you weaker	213
23.1.3.7	Enemy Abilities	213
23.1.3.8	You can soft lock yourself	213
23.1.4	Confusing Choices	213
23.1.4.1	Starting level for characters	213
23.1.4.2	Slow overworld movement	213
23.1.4.3	Exiting a dungeon or a town	214
23.1.4.4	The Health Points UI	214
23.1.5	Inconveniencing the player	214
23.1.5.1	The battle menu order	214
23.1.5.2	Every menu is a committal	214
23.1.5.3	Password saves	214
23.1.5.4	Each character has their own money stash	215
23.1.6	Bugs and glitches	215
23.1.6.1	Moonwalking and save warping	215
23.1.6.2	The final maze	215

23.1.6.3 The endings	215
23.1.7 Conclusions	216
23.2 A good game: Dark Souls	216
24 Where To Go From Here	217
24.1 Collections of different topics	217
24.1.1 Books	217
24.1.2 Videos	217
24.1.3 Multiple formats	217
24.2 Pixel Art	217
24.2.1 Multiple Formats	217
24.3 Sound Design	217
24.3.1 Multiple Formats	217
24.4 Game Design	218
24.4.1 Books	218
24.5 Game Development	218
24.5.0.1 Videos	218
Glossary	219
Engines, Libraries And Frameworks	221
Some other useful tools	229
Free assets and resources	231
Contributors	234

List of Tables

1	Conversion between degrees and Radians	11
2	Some reflection formulas for trigonometry	12
3	Some Shift Formulas for Trigonometry	12
4	Some addition and difference identities in trigonometry	13
5	Some double-angle formulae used in trigonometry	13
6	The first truth table we'll simplify with Karnaugh Maps	24
7	Karnaugh Map for A XOR B	24
8	Karnaugh Map where the elements of the two "rectangles" have been marked green and red	24
9	Truth table with a "don't care" value	25
10	Karnaugh Map with a "don't care" value	25
11	Karnaugh Map where we pretend the "don't care" value is equal to 1	25
14	Performance table for Dynamic Arrays	45
15	Summary Table for Dynamic Arrays	46
16	Performance table for Linked Lists	47
17	Summary Table for Linked Lists	47
18	Performance table for Doubly-Linked Lists	48
19	Summary Table for Linked Lists	48
20	Performance table for Hash Tables	49
21	Summary Table for Hash Tables	49
22	Performance table for Binary Search Trees	50
23	Summary Table for Binary Search Trees	50
24	Performance table for Heaps	51
25	Summary Table for Heaps	51
27	Summary table for the Singleton Pattern	143
29	Summary table for the Flyweight Pattern	145
30	Summary table for the Observer Pattern	147
31	Summary table for the Strategy Pattern	148
32	Summary table for the Chain of Responsibility Pattern	149

List of Figures

1	Image of a vector	5
2	Graphical representation of a sum of vectors	6
3	Example of a vector multiplied by a value of 3	6
4	Example of a vector multiplied by a value of 0.5	6
5	Example of a vector multiplied by a value of -2	7
6	Unit Circle definition of sine and cosine	11
7	Graphical plotting of the angle of a vector	14
8	Image of a coordinate plane	14
9	Image of a screen coordinate plane	14
10	$O(n)$ growth rate, compared to $O(n^2)$	22
11	Big-O Estimates, plotted	23
12	How $O(2^n)$ overpowers lower complexities	23
13	Example of a diamond problem	30
14	Example of a use case diagram	33
15	Example of an actor hierarchy	33
16	Example of a use case	34
17	Example of a use case hierarchy	34
18	Example of a use case extension	34
19	Example of a use case inclusion	35
20	Example of a sub-use case	35

21	Example of classes in UML	36
22	Relationships between classes in an UML Diagram	36
23	Example of inheritance in UML class diagrams	36
24	Example of association in UML class diagrams	37
25	Example of aggregation and composition in UML class diagrams	37
26	Example of dependency in UML class diagrams	38
27	Defining an interface in UML	38
28	Interface Realization in UML	39
29	Example of an activity diagram	39
30	Example of activity diagrams start and end nodes	39
31	Example of Action in activity diagrams	40
32	Example of decision, using hexagons to represent the condition	40
33	Example of loops, using guards to represent the condition	40
34	Example of how nested loops and conditions are performed	41
35	Example of concurrent processes in activity diagrams	41
36	Example of signals in activity diagrams	42
37	Example of swimlanes in activity diagrams	42
38	Example of a note inside of an activity diagram	43
39	Dynamic Arrays Reference Image	44
40	Adding an element at the beginning of a Dynamic Array	44
41	Adding an element at the end of a Dynamic Array	45
42	Adding an element at an arbitrary position of a Dynamic Array	45
43	Linked List Reference Image	46
44	Double-Ended Linked List Reference Image	46
45	Inserting a new node at the beginning of a linked list	47
46	Inserting a new node at the end of a (double-ended) linked list	47
47	Inserting a new node at an arbitrary position in a (double-ended) linked list	47
48	Doubly Linked List Reference Image	48
49	Hash Table Reference Image (Hash Table with Buckets)	49
50	Binary Search Tree Reference	50
51	Heap Reference Image (Min-Heap)	51
52	How a stack works	52
53	Array and linked list implementations of a stack	52
54	How a queue works	52
55	Array and linked list implementation of a queue	53
56	How a circular queue works	53
57	Array and linked list implementation of a circular queue	54
58	Two threads and a shared variable	55
59	Thread 1 reads the variable	56
60	While Thread 1 is working, Thread 2 reads the variable	56
61	Thread 1 writes the variable	56
62	Thread 2 writes the variable	57
63	Both Threads Terminated	57
64	Diagram of the waterfall life cycle model	63
65	Diagram of the incremental life cycle model	64
66	High-level diagram of the evolutionary life cycle model	65
67	Diagram of the evolutionary life cycle model	65
68	Example of a Kanban Board	67
69	An example screen from Git, a version control system	68
70	An example of screen tearing	86
71	How not clearing the screen can create glitches	87
72	Example used in the AABB collision detection	90

73	Top-Bottom Check	90
74	Top-Bottom Check is not enough	91
75	An example of a left-right check	91
76	Example of the triangle inequality theorem	92
77	Example of a degenerate triangle	92
78	Point/Triangle Collision Detection: division into sub-triangles	95
79	Example image for line/line collision	97
80	Example for collision detection	101
81	Graphical example of a quad tree, overlaid on the reference image	102
82	Example tile-based level	103
83	Tile-based example: falling	104
84	Example tile-based level with a bigger object	105
85	Tile-based example with a bigger object: falling	106
86	Example of a hitbox (red) and a hurtbox (blue)	106
87	Images used as a reference for collision reaction	107
88	How the direction + velocity method reacts to collisions on a horizontal plane	108
89	How velocity changing direction can teleport you	108
90	Example of shallow-axis based reaction	109
91	Example of the “Bullet through paper” problem	110
92	Reference Image for Screen Space and Game Space	112
93	Example of an horizontally-tracking camera	113
94	Example of a full-tracking camera	114
95	Example of camera trap-based system	114
96	Example of look-ahead camera	115
97	How the camera may end up showing off-map areas	116
98	Example of how to induce lateral thinking with environmental damage	119
99	Example of how to induce lateral thinking by “breaking the fourth wall”	119
100	Example of secret-in-secret	120
101	Example of a telegraphed screen-filling attack in a shooter	122
102	Example sprite that gets padded to match hardware constraints	125
103	Example spritesheet that gets padded to match hardware constraints	126
104	Windowed Game Example - A 640x480 game in a 1920x1080 Window	127
105	Fullscreen Game Example - Recalculating items positions according to the window size	127
106	Fullscreen Game Example - Virtual Resolution	127
107	Dithering example	128
108	Dithering Table Example	128
109	Some more dithering examples	129
110	Example of black and transparent tileset used in “inside rooms”	131
111	Example of incomplete “inside room”	131
112	Example of “inside room” with the black/transparent overlay	132
113	Graphical Representation of Sample Rate (44.1KHz)	132
114	Graphical Representation of Sample Rate (8KHz)	133
115	Example of a serif font (DejaVu Serif)	136
116	Example of a sans-serif font (DejaVu Sans)	136
117	Example of a proportional font (DejaVu Serif)	136
118	Example of a monospaced font (Inconsolata)	136
119	The UML diagram for a singleton pattern	142
120	UML diagram for the Command Pattern	144
121	UML Diagram of the Flyweight pattern	145
122	The UML diagram of the observer pattern	146
123	The UML diagram of the strategy pattern	147
124	UML Diagram of the Chain of Responsibility Pattern	148

125	Diagram of the Component Design Pattern	149
126	Diagram of a character's state machine	151
127	Diagram of a menu system's state machine	152
128	Example of Manhattan distance	157
129	Example of Euclidean Distance	158
130	Pathfinding Algorithms Reference Image	158
131	Pathfinding Algorithms Heuristics Reference Image	159
132	The path taken by the greedy "Best First" algorithm	160
133	The path taken by the Dijkstra Algorithm	161
134	Finite state machine representing an enemy AI	163
135	How the recursive backtracker algorithm works (1)	164
136	How the recursive backtracker algorithm works (2)	165
137	How the recursive backtracker algorithm works (3)	165
138	How the recursive backtracker algorithm works (4)	165
139	How the Randomized Kruskal's Algorithm Works (1/6)	166
140	How the Randomized Kruskal's Algorithm Works (2/6)	166
141	How the Randomized Kruskal's Algorithm Works (3/6)	167
142	How the Randomized Kruskal's Algorithm Works (4/6)	167
143	How the Randomized Kruskal's Algorithm Works (5/6)	167
144	How the Randomized Kruskal's Algorithm Works (6/6)	168
145	How the Recursive Division Algorithm Works (1/6)	168
146	How the Recursive Division Algorithm Works (2/6)	169
147	How the Recursive Division Algorithm Works (3/6)	169
148	How the Recursive Division Algorithm Works (4/6)	169
149	How the Recursive Division Algorithm Works (5/6)	170
150	How the Recursive Division Algorithm Works (6/6)	170
151	The bias of Recursive Division Algorithm	171
152	How the Binary Tree Maze generation works (1/6)	171
153	How the Binary Tree Maze generation works (2/6)	172
154	How the Binary Tree Maze generation works (3/6)	172
155	How the Binary Tree Maze generation works (4/6)	172
156	How the Binary Tree Maze generation works (5/6)	173
157	How the Binary Tree Maze generation works (6/6)	173
158	How Eller's Maze Generation Algorithm Works (1/7)	174
159	How Eller's Maze Generation Algorithm Works (2/7)	174
160	How Eller's Maze Generation Algorithm Works (3/7)	174
161	How Eller's Maze Generation Algorithm Works (4/7)	174
162	How Eller's Maze Generation Algorithm Works (5/7)	174
163	How Eller's Maze Generation Algorithm Works (6/7)	174
164	How Eller's Maze Generation Algorithm Works (7/7)	174
165	Example of Random Noise	175
166	Demonstration of an image with loop points	177
167	Plotting a physics-accurate jump	180
168	Plotting a jump with enhanced gravity	180
169	Example of how jump buffering would work	181
170	Example of how coyote time would work	182
171	Example of how timed jumps would work	183
172	Example of a matrix, saved as "array of arrays"	185
173	What happens when deleting a match immediately	187
174	Example of a P2P connection	196
175	Two cheaters meet in P2P	197
176	What would happen if one of the Peers had the authoritative game state	197

177	Example of a dedicated server	198
178	Two cheaters vs. Dedicated server	198

List of Code Listings

1	Example code listing	3
2	Example of an $O(1)$ algorithm	19
3	Example of an $O(\log(n))$ algorithm (Binary Search)	19
4	Example of an $O(n)$ algorithm (printing of a list)	19
5	Example of an $O(n^2)$ algorithm (bubble sort)	20
6	A simple $O(1)$ algorithm	21
7	A simple $o(n)$ algorithm	21
8	The bubble sort algorithm, an $O(n^2)$ algorithm	21
9	A more complex algorithm to estimate	21
10	Example of an entity declared as data	32
11	Game Loop example	82
12	Game loop with fixed timesteps	83
13	Game loop with variable time steps	83
14	Game loop with Semi-Fixed time steps	84
15	Game loop with Frame Limiting	84
16	Point to point collision detection	88
17	Shortened version of a point to point collision detection	88
18	Point to circle collision detection	89
19	Shorter version of a point to circle collision detection	89
20	Circle to Circle Collision Detection	89
21	Shorter Version of a Circle to Circle Collision Detection	90
22	Axis-Aligned Bounding Box Collision Detection	91
23	Line to Point Collision detection	93
24	Partial Implementation of a Line to Circle Collision Detection	93
25	Line to circle collision detection	94
26	Point/Rectangle collision detection	95
27	Point/Triangle Collision Detection	96
28	Rectangle to Circle Collision Detection	96
29	Implementation of the line/line collision detection	98
30	Implementation of the line/rectangle collision detection	99
31	Example of a possible implementation of pixel perfect collision detection	100
32	Brute Force Method of collision search	102
33	Converting player coordinates into tile coordinates	104
34	Tile-based collision detection	104
35	Code for the direction + velocity collision reaction	107
36	Example of the "snapshot" collision reaction method	109
37	Example of swappable sound effects	135
38	Example procedural weapon creation	138
39	Example Randomized weapon creation	138
40	Example of a singleton pattern	142
41	Example of a singleton pattern with lazy loading	143
42	Example code for the Command Pattern	144
43	Code for a flyweight pattern	145
44	Code for an observer pattern	146
45	Code for a strategy pattern	147
46	Code for a chain of responsibility pattern	148
47	Example Implementation Of the Component Pattern	150

48	Linear Tweening	153
49	Example of a simple easing function	153
50	Ease-in	154
51	Ease-out	154
52	Possible representation of a 2D grid	155
53	Example code calculating the Manhattan distance on a 2D grid	157
54	Example code calculating the Euclidean distance on a 2D grid	158
55	The node structure used in the greedy "Best First" algorithm	159
56	The greedy "Best First" algorithm	159
57	The node structure used in the Dijkstra Algorithm	160
58	The Dijkstra Algorithm	161
59	The A* Algorithm	162
60	Example implementation of randomized noise	175
61	Example of I-Frames Implementation	176
62	Example of an infinitely scrolling background implementation	177
63	Code for applying gravity to an object	179
64	Code for jump with enhanced gravity while falling	180
65	Code for jumping without buffering	181
66	Jump buffering example	182
67	Coyote time code example	182
68	Example code of how timed jumps work	183
69	Finding horizontal matches in a match-3 game	185
70	Finding vertical matches in a match-3 game	186

1 Foreword

Every time we start a new learning experience, we may be showered by a sleuth of doubts and fears. The task, however small, may seem daunting. And considering how large the field of Game Development can be, these fears are easily understandable.

This book is meant to be a reference for game developers, oriented at 2D, as well as being a collection of “best practices” that you should follow when developing a game by yourself (or with some friends).

But you shouldn’t let these “best practices” jail you into a way of thinking that is not yours, actually my first tip in this book is **do not follow this book. Yet.**

Do it wrong.

Learn why these best practices exist by experience, make code so convoluted that you cannot maintain it anymore, don’t check for overflows in your numbers, **allow yourself to do it wrong.**

Your toolbox is there to aid you, your tools don’t have feelings that can be hurt (although they will grumble at you many times) in the same way that you cannot hurt a hammer when missing the nail head. You cannot break a computer by getting things wrong (at least 99.9999% of the time). Breaking the rules will just help you understand them better.

Write your own code, keep it as simple as you can, and practice.

Don’t let people around you tell you that “you shouldn’t do it that way”, if you allow that to happen you’re depriving yourself of a great opportunity to learn. Don’t let others’ “lion tamer syndrome” get to you, avoid complex structures as much as possible; cutting and pasting code will get you nowhere.

But most of all, never give up and try to keep it fun.

There will be times where you feel like giving up, because something doesn’t work exactly as you want it to, or because you feel you’re not ready to put out some code. When you don’t feel ready, just try making something simple, something that will teach you how to manipulate data structures and that gives you a result in just a couple days of work. Just having a rectangle moving on the screen, reacting to your key presses can be that small confidence boost that can get you farther and farther into this world.

And when all else fails, take a pen, some paper and your favourite [Rubber Duck](#) (make sure it is impact-proof) and **think**.

Coding is hard, but at the same time, it can give you lots of satisfaction.

I really hope that this book will give you tips, tricks and structures that one day will make you say “Oh yeah, I can use that!”. So that one day you are able to craft an experience that someone else will enjoy, while you enjoy the journey that brings to such experience.

2 Introduction

A journey of a thousand miles begins with a single step

Laozi - Tao Te Ching

Welcome to the book! This book aims to be an organized collection of the community's knowledge on game development techniques, algorithms and experience with the objective of being as comprehensive as possible.

2.1 Why another game development book?

It's really common in today's game development scene to approach game development through tools that abstract and guide our efforts, without exposing us to the nitty-gritty details of how things work on low-level and speeding up and easing our development process. This approach is great when things work well, but it can be seriously detrimental when we are facing against issues: we are tied to what the library/framework creators decided was the best (read "applicable in the widest range of problems") approach to solving a problem.

Games normally run at 30fps, more modern games run at 60fps, some even more, leaving us with between 33ms to 16ms or less to process a frame, which includes:

- Process the user input;
- Update the player movement according to the input;
- Update the state of any AI that is used in the level;
- Move the NPCs according to their AI;
- Identify Collisions between all game objects;
- React to said Collisions;
- Update the Camera (if present);
- Update the HUD (if present);
- Draw the scene to the screen.

These are only some basic things that can be subject to change in a game, **every single frame**.

When things don't go well, the game lags, slows down or even locks up. In that case we will be forced to take the matter in our hands and get dirty handling things exactly as we want them (instead of trying to solve a generic problem).

When you are coding a game for any device that doesn't really have "infinite memory", like a mobile phone, consoles or older computers, this "technical low-level know-how" becomes all the more important.

This book wants to open the box that contains everything related to 2D game development, plus some small tips and tricks to make your game more enjoyable. This way, if your game encounters some issues, you won't fear diving into low-level details and fix it yourself.

Or why not, make everything from scratch using some pure-multimedia interfaces (like SDL or SFML) instead of fully fledged game engines (like Unity).

This book aims to be a free (as in price) teaching and reference resource for anyone who wants to learn 2D game development, including the nitty-gritty details.

Enjoy!

2.2 Conventions used in this book

2.2.1 Logic Conventions

When talking about logic theory, the variables will be represented with a single uppercase letter, written in math mode: A

The following symbol will be used to represent a logical "AND": \wedge

The following symbol will be used to represent a logical “OR”: \vee

The logical negation of a variable will be represented with a straight line on top of the variable, so the negation of the variable A will be \bar{A}

2.2.2 Code Listings

Listings, algorithms and anything that is code will be shown in monotype fonts, using syntax highlighting where possible, inside of a dedicated frame:

Listing 1: Example code listing

```
function Example(phrase):  
2     print phrase
```

2.2.3 Block Quotes

There will be times when it’s needed to write down something from another source verbatim, for that we will use block quotes, which are styled as follows:

```
Hi, I’m a block quote! You will see me when something is... quoted!  
I am another row of the block quote! Have a nice day!
```

2.3 Structure of this Book

This book is structured in many chapters, here you will find a small description of each and every one of them.

- **Foreword:** You didn’t skip it, right?
- **Introduction:** Here we present the structure of the book and the reasons why it came to exist. You are reading it now, hold tight, you’re almost there!
- **The Maths Behind Game Development:** Here we will learn the basic maths that are behind any game, like vectors, matrices and screen coordinates.
- **Some Computer Science Fundamentals:** Here we will learn (or revise) some known computer science fundamentals (and some less-known too!) and rules that will help us managing the development of our game.
- **Project Management Tips:** Project management is hard! Here we will take a look at some common pitfalls and tips that will help us deliver our own project and deliver it in time.
- **Introduction to game design:** In this section we will talk about platforms games can run on, input methods as well as some game genres.
- **Writing a Game Design Document:** In this section we will take a look at one of the first documents that comes to exist when we want to make a game, and how to write one.
- **The Game Loop:** Here we will learn the basics of the “game loop”, the very base of any videogame.
- **Collision Detection and Reaction:** In this section we will talk about one of the most complex and computationally expensive operations in a videogame: collision detection.
- **Cameras:** In this section we will talk about the different types of cameras you can implement in a 2D game, with in-depth analysis and explanation;
- **Game Design:** In this chapter we will talk about level design and how to walk your player through the learning and reinforcement of game mechanics, dipping our toes into the huge topic that is game design.
- **Creating your resources:** Small or solo game developers may need to create their own resources, in this section we will take a look at how to create our own graphics, sounds and music.
- **Procedural Content Generation:** In this chapters we will see the difference between procedural and random content generation and how procedural generation can apply to more things than we think.

- **Useful Patterns and Containers:** A head-first dive into the software engineering side of game development, in this section we will check many software design patterns and containers used to make your game more maintainable and better performing.
- **Artificial Intelligence in Videogames:** In this section we will talk about algorithms that will help you coding your enemy AI, as well as anything that must have a “semblance of intelligence” in your videogame;
- **Other Useful Algorithms:** In this section we will see some algorithms that are commonly used in game, including path finding, world generation and more.
- **Developing Game Mechanics:** Here we will dive into the game development’s darkest and dirtiest secrets, how games fool us into strong emotions but also how some of the most used mechanics are implemented.
- **Testing your game:** This section is all about hunting bugs, without a can of bug spray. A deep dive into the world of testing, both automated and manual.
- **Optimizing and Profiling your game:** When things don’t go right, like the game is stuttering or too slow, we have to rely on profiling and optimization. In this section we will learn tips and tricks and procedures to see how to make our games perform better.
- **Balancing Your Game:** A very idealistic vision on game balance, in this chapter we will take a look inside the player’s mind and look at how something that may seem “a nice challenge” to us can translate into a “terrible balance issue” to our players.
- **Marketing Your Game:** Here we will take a look at mistakes the industry has done when marketing and maintaining their own products, from the point of view of a small indie developer. We will also check some of the more conflictual topics like lootboxes, microtransactions and season passes.
- **Game Jams:** A small section dedicated on Game Jams and how to participate to one without losing your mind in the process, and still deliver a prototype.
- **Dissecting Games:** A small section dedicated to dissecting the characteristics of one (very) bad game, and one (very) good game, to give us more perspective on what makes a good game “good” and what instead makes a bad one.
- **Where to go from here:** We’re at the home stretch, you learned a lot so far, here you will find pointers to other resources that may be useful to learn even more.
- **Glossary:** Any word that has a **g** symbol will find a definition here.
- **Engines and Frameworks:** A collection of frameworks and engines you can choose from to begin your game development.
- **Tools:** Some software and toolkits you can use to create your own resources, maps and overall make your development process easier and more manageable.
- **Premade Assets and resources:** In this appendix we will find links to many websites and resource for graphics, sounds, music or learning.
- **Contributors:** Last but not least, the names of the people who contributed in making this book.

Have a nice stay and let’s go!

3 The Maths Behind Game Development

Do not worry about your difficulties in Mathematics. I can assure you mine are still greater.

Albert Einstein

This book assumes you already have some minimal knowledge of maths, including but not limited to:

- Logarithms
- Exponentials
- Roots
- Equations
- Limits

In this chapter we'll take a quick look (or if you already know them, a refresher) on the basic maths needed to make a 2D game.

3.1 The modulo operator

Very basic, but sometimes overlooked, function in mathematics is the “modulo” function (or “modulo operator”). Modulo is a function that takes 2 arguments, let's call them “a” and “b”, and returns the remainder of the division represented by a/b .

So we have examples like $mod(3, 2) = 1$ or $mod(4, 5) = 4$ and $mod(8, 4) = 0$.

In most programming languages the modulo function is hidden behind the operator “%”, which means that the function $mod(3, 2)$ is represented with $3\%2$.

The modulo operator is very useful when we need to loop an ever-growing value between two values (as will be shown in [infinitely scrolling backgrounds](#)).

3.2 Vectors

For our objective, we will simplify the complex matter that is vectors as much as possible.

In the case of 2D game development, a vector is just a pair of values (x, y) .

Vectors usually represent a force applied to a body, its velocity or acceleration and are graphically represented with an arrow.

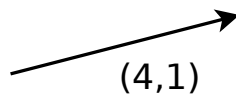


Figure 1: Image of a vector

The pain of learning about vectors is paid off by their capacity of being added and subtracted among themselves, as well as being multiplied by a number (called a “scalar”) and between themselves.

3.2.1 Adding and Subtracting Vectors

Adding vectors is as easy as adding its “members”. Let's consider the following vectors:

$$v = (2, 4)$$

$$u = (1, 5)$$

The sum vector s will then be:

$$s = u + v = (2 + 1, 4 + 5) = (3, 9)$$

Graphically it can be represented by placing the tail of the arrow v on the head of the arrow u , or vice-versa:

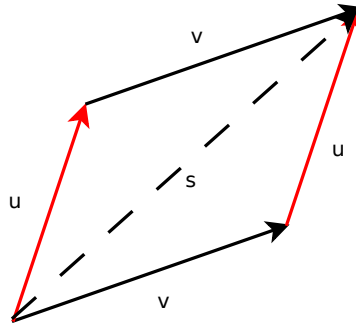


Figure 2: Graphical representation of a sum of vectors

3.2.2 Scaling Vectors

There may be situations where you need to make a vector x times longer. This operation is called “scalar multiplication” and it is performed as follows:

$$v = (2, 4)$$

$$3 \cdot v = (2 \cdot 3, 4 \cdot 3) = (6, 12)$$

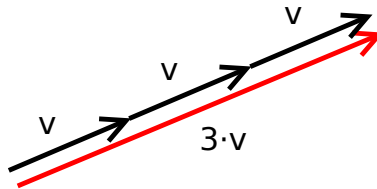


Figure 3: Example of a vector multiplied by a value of 3

Obviously this works with scalars with values between 0 and 1:

$$v = (2, 4)$$

$$\frac{1}{2} \cdot v = (\frac{1}{2} \cdot 2, \frac{1}{2} \cdot 4) = (1, 2)$$

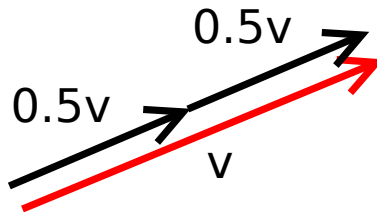


Figure 4: Example of a vector multiplied by a value of 0.5

When you multiply the vector by a value less than 0, the vector will rotate by 180°.

$$v = (2, 4)$$

$$-2 \cdot v = (-2 \cdot 2, -2 \cdot 4) = (-4, -8)$$

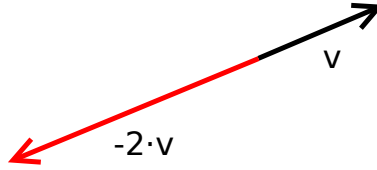


Figure 5: Example of a vector multiplied by a value of -2

3.2.3 Dot Product

The dot product (or scalar product, projection product or inner product) is defined as follows:

Given two n-dimensional vectors $v = [v_1, v_2, \dots, v_n]$ and $u = [u_1, u_2, \dots, u_n]$ the dot product is defined as:

$$v \cdot u = \sum_{i=1}^n (v_i \cdot u_i) = (v_1 \cdot u_1) + \dots + (v_n \cdot u_n)$$

So in our case, we can easily calculate the dot product of two two-dimensional vectors $v = [v_1, v_2]$ and $u = [u_1, u_2]$ as:

$$v \cdot u = (v_1 \cdot u_1) + (v_2 \cdot u_2)$$

Let's make an example:

Given the vectors $v = [1, 2]$ and $u = [4, 3]$, the dot vector is:

$$v \cdot u = (1 \cdot 4) + (2 \cdot 3) = 4 + 6 = 10$$

3.2.4 Vector Length and Normalization

Given a vector $a = [a_1, a_2, \dots, a_n]$, you can define the length of the vector as:

$$||a|| = \sqrt{a_1^2 + a_2^2 + \dots + a_n^2}$$

Or alternatively

$$||a|| = \sqrt{a \cdot a}$$

We can get a 1-unit long vector by “normalizing” it, getting a vector that is useful to affect (or indicate) direction without affecting magnitude. A normalized vector is usually indicated with a “hat”, so the normalized vector of $a = [a_1, a_2, \dots, a_n]$ is

$$\hat{a} = \frac{a}{||a||}$$

Knowing that the length of a vector is a scalar (a number, not a vector), normal scalar multiplication rules apply. (See [Scaling Vectors](#))

3.3 Matrices

3.3.1 What is a matrix

Matrices are essentially an $m \times n$ array of numbers, which are used to represent linear transformations.

Here is an example of a 2×3 matrix.

$$A_{2,3} = \begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 0 \end{bmatrix}$$

3.3.2 Matrix sum and subtraction

Summing and subtracting $m \times n$ matrices is done by summing or subtracting each element, here is a simple example. Given the following matrices:

$$A_{2,3} = \begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 0 \end{bmatrix} \quad B_{2,3} = \begin{bmatrix} 1 & 3 & 0 \\ 4 & 2 & 4 \end{bmatrix}$$

We have that:

$$A_{2,3} + B_{2,3} = \begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 3 & 0 \\ 4 & 2 & 4 \end{bmatrix} = \begin{bmatrix} 2+1 & 1+3 & 4+0 \\ 3+4 & 2+2 & 0+4 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 4 \\ 7 & 4 & 4 \end{bmatrix}$$

3.3.3 Multiplication by a scalar

Multiplication by a scalar works in a similar fashion to vectors, given the matrix:

$$A_{2,3} = \begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 0 \end{bmatrix}$$

Multiplication by a scalar is performed by multiplying each member of the matrix by the scalar, like the following example:

$$3 \cdot A_{2,3} = 3 \cdot \begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 3 \cdot 2 & 3 \cdot 1 & 3 \cdot 4 \\ 3 \cdot 3 & 3 \cdot 2 & 3 \cdot 0 \end{bmatrix} = \begin{bmatrix} 6 & 3 & 12 \\ 9 & 6 & 0 \end{bmatrix}$$

3.3.4 Transposition

Given an $m \times n$ matrix A , its transposition is an $n \times m$ matrix A^T constructed by turning rows into columns and columns into rows.

Given the matrix:

$$A_{2,3} = \begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 0 \end{bmatrix}$$

The transpose matrix is:

$$A_{2,3}^T = \begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix}$$

3.3.5 Multiplication between matrices

Given 2 matrices with sizes $m \times n$ and $n \times p$ (mind how the number of rows of the first matrix is the same of the number of columns of the second matrix):

$$A_{3,2} = \begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} B_{2,3} = \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix}$$

We can calculate the multiplication between these two matrices, in the following way.

First of all let's get the size of the resulting matrix, which will be always $m \times p$.

Now we have the following situation:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

Matrix multiplication is called a “rows by columns” multiplication, so to calculate the first row - first column value we'll need the first row of one matrix and the first column of the other.

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

The values in the example will be combines as follows:

$$2 \cdot 2 + 3 \cdot 0 = 4$$

Obtaining the following:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

Let's try the next value:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

The values will be combined as follows:

$$2 \cdot 3 + 3 \cdot 1 = 9$$

Obtaining:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 9 & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

Same goes for the last value, when we are done with the first row, we keep going similarly with the second row:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 9 & 8 \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

Which leads to the following calculation:

$$1 \cdot 2 + 2 \cdot 0 = 2$$

Which we will insert in the result matrix:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 9 & 8 \\ 2 & ? & ? \\ ? & ? & ? \end{bmatrix}$$

You can try completing this calculation yourself, the final result is as follows:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 4 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 9 & 8 \\ 2 & 5 & 4 \\ 8 & 12 & 16 \end{bmatrix}$$

Multiplication between matrices is **non commutative**, which means that the result of $A \times B$ is not equal to the result of $B \times A$, actually one of the results may not even be possible to calculate.

3.3.6 Other uses for matrices

Matrices can be used to quickly represent equation systems, with equation that depend on each other. For instance:

$$\begin{bmatrix} 2 & 3 & 6 \\ 1 & 4 & 9 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$$

Can be used to represent the following system of equations:

$$\begin{cases} 2x + 3y + 6z = 4 \\ 1x + 4y + 9z = 5 \end{cases}$$

Or, as we'll see, matrices can be used to represent transformations in the world of game development.

3.4 Trigonometry

When you want to develop a game, you will probably find yourself needing to rotate items relative to a certain point or relative to each other. To do so, you need to know a bit of trigonometry, so here we go!

3.4.1 Radians vs Degrees

In everyday life, angles are measured in degrees, from 0 to 360 degrees. In some situations in math, it is more comfortable to measure angles using radians, from 0 to 2π .

You can covert back and forth between radians and degrees with the following formulas:

$$angle\ in\ degrees = angle\ in\ radians \cdot \frac{180}{\pi}$$

$$angle\ in\ radians = angle\ in\ degrees \cdot \frac{\pi}{180}$$

This book will always refer to angles in radians, so here are some useful conversions, ready for use:

Table 1: Conversion between degrees and Radians

Degrees	Radians
0°	0
30°	$\frac{\pi}{6}$
45°	$\frac{\pi}{4}$
60°	$\frac{\pi}{3}$
90°	$\frac{\pi}{2}$
180°	π
360°	2π

3.4.2 Sine, Cosine and Tangent

The most important trigonometric functions are sine and cosine. They are usually defined in reference to a “unit circle” (a circle with radius 1).

Given the unit circle, let a line through the origin with an angle θ with the positive side of the x-axis intersect such unit circle. The x coordinate of the intersection point is defined by the measure $\cos(\theta)$, while the y coordinate is defined by the measure $\sin(\theta)$.

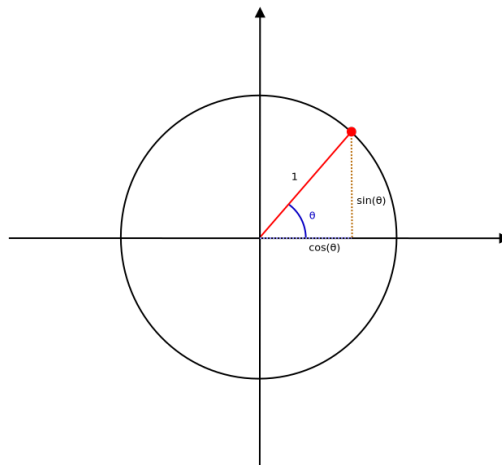


Figure 6: Unit Circle definition of sine and cosine

For the purposes of this book, we will just avoid the complete definition of the tangent function, and just leave it as a formula of sine and cosine:

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)}$$

3.4.3 Pythagorean Trigonometric Identity

One of the most important identities in Trigonometry is the “Pythagorean Trigonometric Identity”, which is expressed as follows, valid for each angle θ :

$$\sin^2(\theta) + \cos^2(\theta) = 1$$

Using this identity, you can express functions in different ways:

$$\cos^2(\theta) = 1 - \sin^2(\theta)$$

$$\sin^2(\theta) = 1 - \cos^2(\theta)$$

Also remember that $\sin^2(\theta) = (\sin(\theta))^2$ and $\cos^2(\theta) = (\cos(\theta))^2$.

3.4.4 Reflections

Sometimes we may need to reflect an angle to express it in an easier way, and their trigonometric formulas will be affected, so the following formulas may come of use:

Table 2: Some reflection formulas for trigonometry

Reflection Formulas
$\sin(-\theta) = -\sin(\theta)$
$\cos(-\theta) = \cos(\theta)$
$\sin(\frac{\pi}{2} - \theta) = \cos(\theta)$
$\cos(\frac{\pi}{2} - \theta) = \sin(\theta)$
$\sin(\pi - \theta) = \sin(\theta)$
$\cos(\pi - \theta) = -\cos(\theta)$
$\sin(2\pi - \theta) = -\sin(\theta) = \sin(-\theta)$
$\cos(2\pi - \theta) = \cos(\theta) = \cos(-\theta)$

3.4.5 Shifts

Trigonometric functions are periodic, so you may have an easier time calculating them when their arguments are shifted by a certain amount. Here we can see some of the shift formulas:

Table 3: Some Shift Formulas for Trigonometry

Shift Formulas
$\sin(\theta \pm \frac{\pi}{2}) = \pm \cos(\theta)$
$\cos(\theta \pm \frac{\pi}{2}) = \mp \sin(\theta)$
$\sin(\theta + \pi) = -\sin(\theta)$

Shift Formulas

$$\begin{aligned}\cos(\theta + \pi) &= -\cos(\theta) \\ \sin(\theta + k \cdot 2\pi) &= \sin(\theta) \\ \cos(\theta + k \cdot 2\pi) &= \cos(\theta)\end{aligned}$$

3.4.6 Trigonometric Addition and subtraction

Sometimes you may need to express a trigonometric formula with a complex argument by splitting such argument into different trigonometric formulas. If such argument is a sum or subtraction of angles, you can use the following formulas:

Table 4: Some addition and difference identities in trigonometry

Addition/Difference Identities
$\sin(\alpha \pm \beta) = \sin(\alpha)\cos(\beta) \pm \cos(\alpha)\sin(\beta)$
$\cos(\alpha \pm \beta) = \cos(\alpha)\cos(\beta) \mp \sin(\alpha)\sin(\beta)$

3.4.7 Double-Angle Formulae

Other times (mostly on paper) you may have an argument that is a multiple of a known angle, in that case you can use double-angle formulae to calculate them.

Table 5: Some double-angle formulae used in trigonometry

Double-Angle Formulae
$\sin(2\theta) = 2\sin(\theta)\cos(\theta)$
$\cos(2\theta) = \cos^2(\theta) - \sin^2(\theta)$

3.4.8 Inverse Formulas

As with practically all math formulas, there are inverse formulas for sine and cosine, called *arcsin* and *arccos*, which allow to find an angle, given its sine and cosine.

In this book we won't specify more, besides what could be the most useful: the 2-argument arctangent.

This formula allows you to find the angle of a vector, relative to the coordinate system, given the *x* and *y* coordinates of its “tip”, such angle θ is defined as:

$$\theta = \arctan\left(\frac{y}{x}\right)$$

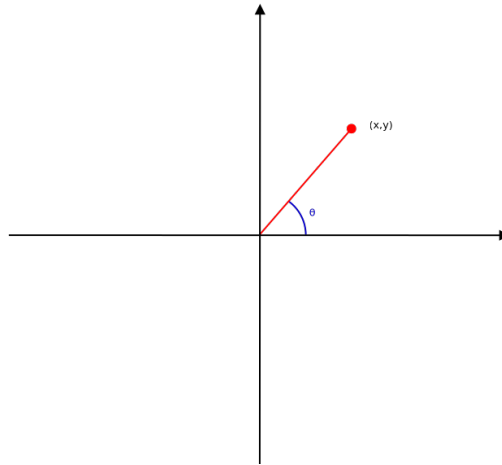


Figure 7: Graphical plotting of the angle of a vector

3.5 Coordinate Systems on computers

When it comes to 2D graphics on computers, our world gets quite literally turned upside down.

In our math courses we learned about the Coordinate Plane, with an x axis going from left to right and a y axis going from bottom to top, where said axis cross it's called the "Origin".

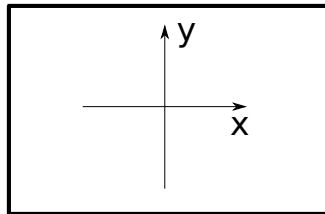


Figure 8: Image of a coordinate plane

When it comes to 2D graphics on computers and game development, the coordinate plane looks like this:

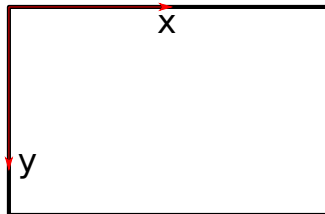


Figure 9: Image of a screen coordinate plane

The origin is placed on the top left of the screen (at coordinates $(0,0)$) and the y axis is going from top to bottom. It's a little weird at the beginning, but it's not hard to get used to it.

3.6 Transformation Matrices

There will be a time, in our game development journey where we need to rotate an object, and that's bound to be pretty easy because rotation is something that practically all engines and toolkits do natively. But also there will be times where we need to do transformations by hand.

An instance where it may happen is rotating an item relative to a certain point or another item: imagine a squadron of war planes flying in formation, where all the planes will move (and thus rotate) relative to the "team leader".

In this chapter we'll talk about the 3 most used transformations:

- Stretching/Squeezing/Scaling;
- Rotation;
- Shearing.

3.6.1 Stretching

Stretching is a transformation that enlarges all distances in a certain direction by a defined constant factor. In 2D graphics you can stretch (or squeeze) along the x-axis, the y-axis or both.

If you want to stretch something along the x-axis by a factor of k , you will have the following system of equations:

$$\begin{cases} x' = k \cdot x \\ y' = y \end{cases}$$

which is translated in the following matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Likewise, you can stretch something along the y-axis by a factor of k by using the following matrices:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

You can mix and match the factors and obtain different kinds of stretching, if the same factor k is used both on the x and y-axis, we are performing a *scaling* operation.

In instead of stretching you want to squeeze something by a factor of k , you just need to use the following matrices for the x-axis:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \frac{1}{k} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

and respectively, the y-axis:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{k} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

3.6.2 Rotation

If you want to rotate an object by a certain angle θ , you need to decide upon two things (besides the angle of rotation):

- Direction of rotation (clockwise or counterclockwise);
- The point of reference for the rotation.

3.6.2.1 Choosing the direction of the rotation

We will call T_R the transformation matrix for the “rotation” functionality.

Similarly to stretching, rotating something of a certain angle θ leads to the following matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = T_R \begin{bmatrix} x \\ y \end{bmatrix}$$

If we want to rotate something **clockwise**, relative to its reference point, we will have the following transformation matrix:

$$T_R = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

If instead we want our rotation to be **counterclockwise**, we will instead use the following matrix:

$$T_R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

These formulas assume that the x-axis points right and the y-axis points up, if the y-axis points down in your implementation, you need to swap the matrices.

3.6.2.2 Rotating referred to an arbitrary point

The biggest problem in rotation is rotating an object relative to a certain point: you need to know the origin of the coordinate system as well and modify the matrices as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = T_R \begin{bmatrix} x - x_{origin} \\ y - y_{origin} \end{bmatrix} + \begin{bmatrix} x_{origin} \\ y_{origin} \end{bmatrix}$$

In short, you need to rotate the item by first “bringing it centered to the origin”, rotating it and then bring it back into its original position.

3.6.3 Shearing

During stretching, we used the elements that are in the “main diagonal” to stretch our objects. If we modify the elements in the “antidiagonal”, we will obtain shear mapping (or shearing).

Shearing will move points along a certain axis with a “strength” defined by the distance along the other axis: if we shear a rectangle, we will obtain a parallelogram.

A shear parallel to the x-axis will have the following matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

While a shear parallel to the y-axis will instead have the following matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

4 Some Computer Science Fundamentals

The computing scientist's main challenge is not to get confused by the complexities of his own making.

Edsger W. Dijkstra

In order to understand some of the language that is coming up, it is necessary to learn a bit of the computer science language and fundamentals.

This chapter will briefly explain some of the language and terms used, their meaning and how they contribute to your activity of developing games.

In this chapter we'll assume you already know what the following terms mean:

- Truth Table
- Algorithm

4.1 De Morgan's Laws and Conditional Expressions

De Morgan's laws are fundamental in computer science as well as in any subject that involves propositional logic. We will take a quick look at the strictly coding-related meaning.

De Morgan's laws can be written as:

not (A and B) = not A or not B

not (A or B) = not A and not B

In symbols:

$$\overline{(A \wedge B)} = \bar{A} \vee \bar{B}$$

$$\overline{(A \vee B)} = \bar{A} \wedge \bar{B}$$

These laws allow us to express our own conditionals in different ways, allowing for more readability and maybe avoid some boolean manipulation that can hinder the performance of our game.

4.2 Estimating the order of algorithms

Now more than ever, you need to be able to be efficient. How do you know how “efficient” some piece of algorithm is?

Seeing how much time it takes is not an option, computer specifications change from system to system, so we need something that could be considered “cross-platform”.

This is where notations come into play.

There are 3 types of Asymptotic notation you should know: Ω , Θ and O .

$\Omega()$ represents a **lower bound**: this means that the algorithm will take **at least** as many cycles as specified.

$O()$ represents an **upper bound**: it's the most used notation and means that the algorithm will take **at most** as many cycles as specified.

$\Theta()$ is a **tight bound**, used when the big- O notation and the big- Ω notation have the same value, which can help define the behaviour of the algorithm better.

We will now talk about the most common Big- O notations, from “most efficient” to “least efficient”.

4.2.1 $O(1)$

An algorithm that executes in $O(1)$ is said to execute “in constant time”, which means that no matter how much data is input in the algorithm, said algorithm will execute in the same time.

An example of a simple $O(1)$ algorithm is an algorithm that, given a list of elements (with at least one element), returns `True` if the first element is `null`.

Listing 2: Example of an $O(1)$ algorithm

```
function isFirstElementNull(elements):  
2   return true if elements[0] is null
```

To be precise, this algorithm will perform both in $O(1)$ and $\Omega(1)$, so it will perform in $\Theta(1)$.

4.2.2 $O(\log(n))$

An algorithm that executes in $O(\log(n))$ is said to execute in “logarithmic time”, which means that given an input of n items, the algorithm will execute $\log(n)$ cycles at most.

An example of a $O(\log(n))$ algorithm is the so-called “binary search” on a ordered list of items.

Listing 3: Example of an $O(\log(n))$ algorithm (Binary Search)

```
binarySearch(elements, element_to_find):  
2   get middle element  
   is it the element_to_find?  
4   yes: return the middle element position  
   else:  
6       is the element to find bigger than the "middle element"?  
         yes: perform binarySearch on the half of the list bigger than "middle element"  
         "  
8         no: perform binarySearch on the half of the list smaller than "middle element"  
         "
```

The best case is the time when you get the element to find to be the “middle element” of the list, in that case the algorithm will execute in linear time: $\Theta(1)$ - You need **at least one lookup** ($\Omega(1)$) and **at most one lookup** ($O(1)$).

In the worst case, the element is not present in the list, so you have to split the list and find the middle element until you realize that you don’t have any more elements to iterate - this translates into a **tight bound** of $\Theta(\log_2 n)$

4.2.3 $O(n)$

An algorithm that executes in $O(n)$ is said to execute in “linear time”, which means that given an input of n items, the algorithm will execute at most n cycles.

An example of a simple $O(n)$ algorithm is the one that prints a list, element by element.

Listing 4: Example of an $O(n)$ algorithm (printing of a list)

```
printList(list):  
2   for each element in list:  
       print element
```

It’s evident that this algorithm will call the `print` function n times, where n is the size of the list. This translates in a $\Theta(n)$ complexity, which is both $O(n)$ and $\Omega(n)$.

There is no “best” or “worst” case here, the algorithm prints n elements, no matter their order, the alignment of planets and stars or the permission of its parents.

4.2.4 $O(n \cdot \log(n))$

An algorithm that executes in $O(n \cdot \log(n))$ executes in a time slightly longer than a linear algorithm, but it’s still considered “ideal”. These algorithms are said to execute in “quasi-linear”, “log-linear”, “super-linear” or “linearithmic” time.

Given an input of n elements, these algorithms execute $n \cdot \log(n)$ steps, or cycles.

Some algorithms that run in $O(n \cdot \log(n))$ are:

- Quick Sort
- Heap Sort
- Fast Fourier Transforms (FFT)

These algorithms are more complex than a simple example and would require a chapter on their own, so we’ll leave examples aside for now.

4.2.5 $O(n^2)$

Quadratic algorithms, as the algorithms that execute in $O(n^2)$ are called, are the door to the “danger zone”.

These algorithms can eat your CPU time quite quickly, although they can still be used for small computations somewhat efficiently.

Given an input of n elements, these algorithms execute n^2 cycles, which means that given an input of **20** elements, we’d find ourselves executing **400** cycles.

A simple example of a quadratic algorithm is “bubble sort”. A pseudo-code implementation is written here.

Listing 5: Example of an $O(n^2)$ algorithm (bubble sort)

```
bubbleSort(A : list of sortable items )
2   n = length(A)
   repeat
4       swapped = false
       for i = 1 to n-1 inclusive do
6           if A[i-1] > A[i] then
               swap( A[i-1], A[i] )
8               swapped = true
           end if
10      end for
   until not swapped
```

Anything with complexity higher than $O(n^2)$ is usually considered unusable.

4.2.6 $O(2^n)$

Algorithms that execute in exponential time are considered a major code red, and will usually be replaced with heuristic algorithms (which trade some precision for a lower complexity).

Given an input of 20 elements, an algorithm that executes in $O(2^n)$ will execute $2^{20} = 1\,048\,576$ cycles!

4.3 A primer on calculating the order of your algorithms

4.3.1 Some basics

When you estimate an algorithm, you usually want to calculate how it functions “in the worst case”, which usually means that all loops get to their end (of the list or the counter) and everything takes the longest time

possible.

Let's start with an example:

Listing 6: A simple $O(1)$ algorithm

```
// A simple O(1) algorithm: assigning to a variable
2 my_variable = 1
```

This is a simple assignment operation, we are considering this instantaneous. So its complexity is $O(1)$.

Now let's see another algorithm:

Listing 7: A simple $o(n)$ algorithm

```
// A simple O(n) algorithm: iterating through a list
2 for item in my_list:
    print(item)
```

In this case we are iterating through a list, we can see that as the list grows, the number of times we print an element on our screen grows too. So if the list is n items long, we will have n calls to the output statement. This is an $O(n)$ complexity algorithm.

Now let's take something we already saw and analyze it: the bubble sort algorithm:

Listing 8: The bubble sort algorithm, an $O(n^2)$ algorithm

```
bubbleSort(A : list of sortable items )
2   n = length(A)
   repeat
4       swapped = false
       for i = 1 to n-1 inclusive do
6           if A[i-1] > A[i] then
               swap( A[i-1], A[i] )
8               swapped = true
           end if
       end for
10      until not swapped
```

This will require a small effort on our part: we can see that there are 2 nested loops in this code. What's our worst case? The answer is "The items are in the reverse order".

When the items are in the reverse order, we will need to loop through the whole list to get the biggest item at the end of the list, then another time to get the second-biggest item on the second-to-last place on the list... and so on.

So every time we bring an item to its place, we iterate through all the list once. This happens for each item.

So, in a list of length "n", we bring the biggest item to its place "n times" and each "time" requires scanning "n" elements: the result is $n \cdot n = n^2$.

The algorithm has time complexity of $O(n^2)$.

4.3.2 What happens when we have more than one big-O?

There are times when we have code that looks like the following:

Listing 9: A more complex algorithm to estimate

```
// -----
2 n = length(A)
```

```
repeat
4   swapped = false
   for i = 1 to n-1 inclusive do
6       if A[i-1] > A[i] then
           swap( A[i-1], A[i] )
8           swapped = true
       end if
   end for
10  until not swapped
12  // -----
14  for item in A:
16      print(item)
```

As we can see the first part is the bubble sort algorithm, followed by iterating through the (now ordered) list, to print its values.

We can calculate the total estimate as $O(n^2) + O(n)$ and that would be absolutely correct, but as the list grows, the growth rate of $O(n)$ is very minor if compared to $O(n^2)$, as can be seen from the following figure:

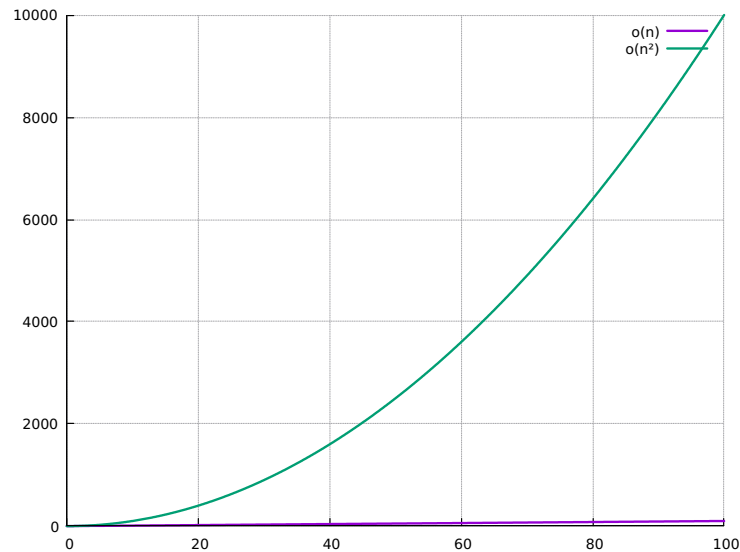


Figure 10: $O(n)$ growth rate, compared to $O(n^2)$

So we can drop the $O(n)$ and consider the entire algorithm as an $O(n^2)$ algorithm in its entirety: this means that when dealing with complexity estimates, you always keep the terms that have the largest “growth rate” (check the [Big-O estimates comparison](#) section for more details).

4.3.3 What do we do with recursive algorithms?

When recursive algorithms are involved, things get a lot more complex, and they involve building recursion trees and sometimes you’ll have to use the so-called “master theorem for divide-and-conquer recurrences”.

Such methods are outside the scope of this book as of now.

4.3.4 How do big-O estimates compare to each other?

Here we can see how big-O estimates compare to each other, graphically and how important it is to write not-inefficient algorithms.

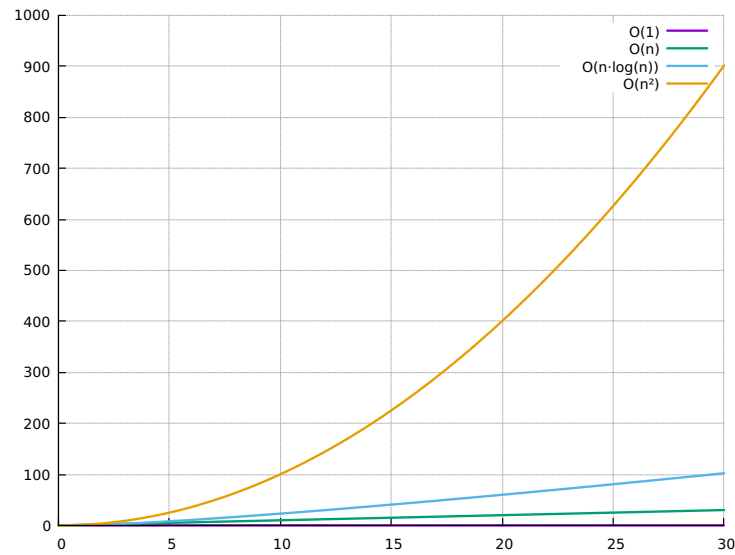


Figure 11: Big-O Estimates, plotted

There is a very specific reason why the $O(2^n)$ estimate is missing from the previous plot: we wouldn't be able to see anything worthwhile if it was included, as seen from the following plot:

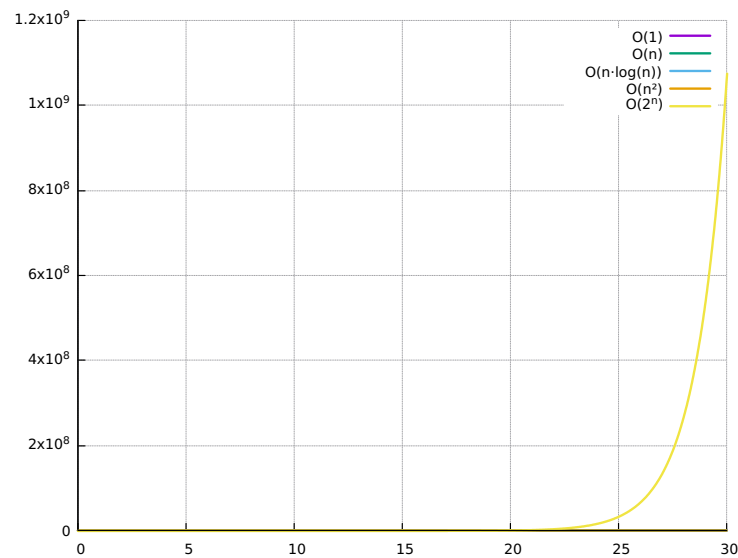


Figure 12: How $O(2^n)$ overpowers lower complexities

[This section is a work in progress and it will be completed as soon as possible]

4.4 Simplifying your conditionals with Karnaugh Maps

Karnaugh maps are a useful tool to simplify boolean algebra expressions, as well as identifying and potentially solving race conditions.

The output of a Karnaugh Map will always be an “OR of ANDs”.

The best way to explain them is to give an example.

Let’s take the following truth table:

Table 6: The first truth table we’ll simplify with Karnaugh Maps

A	B	f
0	0	0
0	1	1
1	0	1
1	1	0

Said table can contain any number of variables (we’ll see how to implement those). To be precise, this table represents the formula $A \text{ XOR } B$ (XOR means ‘exclusive or’).

Let’s arrange it into a double-entry table, like this (Values of A are on top, values of B are on the left):

Table 7: Karnaugh Map for $A \text{ XOR } B$

		A	
		0	1
B	0	0	1
	1	1	0

Now we have to identify the biggest squares or rectangles that contain 2^n elements equal to 1 so that we can cover all the “1” values we have (they can overlap). In this case we’re unlucky as we have only two small rectangles that contain one element each:

Table 8: Karnaugh Map where the elements of the two “rectangles” have been marked green and red

		A	
		0	1
B	0	0	1
	1	1	0

In this case, we have the result we want with the following formula: $f = (A \wedge \bar{B}) \vee (\bar{A} \wedge B)$

Not an improvement at all, but that’s because the example is a really simple one.

4.4.1 “Don’t care”s

Karnaugh Maps show more usefulness when we have the so-called “don’t care”s, situations where we don’t care (wow!) about the result. Here’s an example.

Table 9: Truth table with a “don’t care” value

A	B	f
0	0	0
0	1	1
1	0	1
1	1	x

Putting this truth table into a Karnaugh map we get something a bit more interesting:

Table 10: Karnaugh Map with a "don't care" value

		A	
		0	1
B	0	0	1
	1	1	x

Now we have a value that behaves a bit like a “wild card”, that means we can pretend it’s either a 0 or 1, depending on the situation. In this example we’ll pretend it’s a 1, because it’s the value that will give us the biggest “rectangles”.

Table 11: Karnaugh Map where we pretend the "don't care" value is equal to 1

		A	
		0	1
B	0	0	1
	1	1	1

Now we can find two two-elements rectangles in this map.

The first is the following one:

		A	
		0	1
B	0	0	1
	1	1	1

In this case, we can see that the result is 1 when $B = 1$, no matter the value of A. We’ll keep this in mind.

The second rectangle is:

		A	
		0	1
B	0	0	1
	1	1	1

In this case, we can see that the result is 1 when $A = 1$, no matter the value of B.

This translates into a formula of: $f = (A) \vee (B)$, considering that we don’t care about the result that comes out when $A = 1$ and $B = 1$.

4.4.2 A more complex map

When we have more variables, like the following truth table:

A	B	C	D	<i>f</i>
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	<i>x</i>

Now we'll have to group up our variables and put them in a Karnaugh Map using Gray Code, practically each row or column differs from the adjacent ones by only one bit.

The resulting Karnaugh map is the following (AB on columns, CD on rows):

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	x	1
	10	0	1	1	1

We can see two rectangles that contain 2^n items, one with 2 items, the other with 8, considering the only “don't care” value as 1.

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	x	1
	10	0	1	1	1

In this first rectangle, we can see that the values of C and D don't matter towards the result, as well as the value of B. The only variable that gives the result on this rectangle is $A = 1$. We'll keep that in mind

Let's see the second rectangle:

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	x	1
	10	0	1	1	1

In this case A doesn't give any contribution to the result, but at the same time we need $B = 1$, $C = 1$ and $D = 0$ to get the wanted result.

$D = 0$ translates into $\bar{D} = 1$, which brings the formula to: $f = A \vee (B \wedge C \wedge \bar{D})$.

If we didn't have that "don't care" value, everything would have been more complex.

4.4.3 Guided Exercise

Let's remove the "don't care" value and have the following truth table:

A	B	C	D	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Let's put it into a Karnaugh Map:

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

Find the biggest rectangles:

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

Extract the result: $f = (A \wedge \bar{C}) \vee (A \wedge \bar{B}) \vee (B \wedge C \wedge \bar{D})$

4.5 Object Oriented Programming

4.5.1 Introduction

One of the biggest programming paradigms in use is surely the “Object Oriented Programming” (from now on: “OOP”) paradigm. The fundamental unit of a program, in this paradigm is the *Object*. This paradigm allows to structure your code in a more modular and re-usable way, as well as implementing abstractions, allowing for more solid code and making it possible for other code to make use of your own code without needing to know any details besides its *Interface*.

4.5.2 Objects

Objects are the fundamental unit in OOP, objects are essentially a collection of data and functions. Objects are actually the physical instantiation of what is called a “Class”.

To simplify the concept: a “Class” is a house blueprint, an “Object” is the house itself.

Objects contain data and functions, for the sake of precision, we will use their technical names:

- Functions that are part of an object are called **methods** and they can be classified as:
 - *Instance Methods* when they act on a single object instance;
 - *Static Methods* when they don’t (usually they’re utility functions), that also means that these methods belong to the Class itself and not to its instance.
- Each piece of data contained in the class is called a **Field** and they can be classified as:
 - *Instance Fields* when they’re part of the instance and can change from instance to instance;
 - *Static Fields* when they’re part of the class but don’t change between instances (**Caution:** it does not mean they cannot change, in that case the change will snowball into all the instances).

4.5.3 Abstraction and Interfaces

Abstraction is a fundamental point in OOP, and it is usually taken care of via so-called **Interfaces**.

Interfaces are the front-end that an object offers to other objects so they can interact.

As an example: the interface to your PC is given by Keyboard, Mouse and Screen - you don't need to know how the single electron travels through the circuits to be able to use a computer; same goes for a class that offers a well-abstracted interface.

Being able to abstract a concept, removing the necessity to know the internal workings of the code that is used, is fundamental to be able to write solid and maintainable code, because implementations change, but interfaces rarely do.

Making classes work together with interfaces allows you to modify and optimize your code without having each edit snowball into a flurry of compiler (or interpreter) errors. For instance: a rectangle class exposes in its interface a method `getArea()` - you don't need to know how to calculate the area, you just call that method and know it will return the area of the rectangle.

The concept of keeping the internal workings of a class is called *Information Hiding*.

4.5.4 Inheritance and Polymorphism

One of the biggest aspects of OOP is **inheritance**: you can create other classes based on a so-called “base class”, allowing for extendability of your software.

You can create a “Person” class, with a name, surname and age as fields, and by inheriting from the “Person” class you can create a “Student” class, which has all the fields from Person, plus the “clubs” and “grade” fields.

This allows to create a “tree of classes” that represents part of your software.

From inheritance, OOP presents a concept called **Polymorphism** (From “Poly” - Many, “Morph” - Shape), where you can use the base class to represent the entire class tree, allowing for substitution.

In our “Person-Student” example, you could use a pointer to either a Person or a Student for the sake of getting their first name.

In some languages it is possible for an object to inherit from multiple other objects, this is called “Multiple Inheritance”

4.5.5 The Diamond Problem

Usually when you call a method that is not present in the object itself, the program will look through the object's parents for the method to execute. This usually works well when there is no ambiguity, but what if it happens?

When multiple inheritance is involved, there is a serious possibility of a situation similar to the following

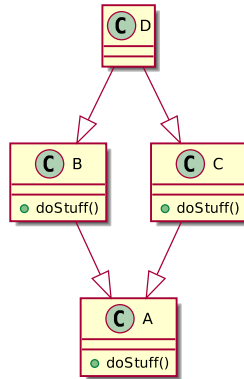


Figure 13: Example of a diamond problem

In this example, class A implements a method `doStuff()` that is overrode by both classes B and C (which inherit from A): now class D inherits from both B and C but does not override `doStuff()`, which one will be chosen?

This is the reason many languages do not implement multiple inheritance between classes (like Java, which allows multiple inheritance only between interfaces), other implement the so-called “virtual inheritance” (C++) and others again use an ordered list to solve the problem (Python).

This is not something you usually need to worry about, but you may want to be careful when you structure your classes to avoid “diamond problems”, so to avoid headaches.

4.5.6 Composition

As opposed to inheritance’s “IS-A” relationship, composition makes use of a “HAS-A” type of relationship.

Composition allows to define objects by declaring which properties they have: a player character can be a sprite with a “Movable” component, or a box could have a “RigidBody” component.

This way we can create new objects by reusing basic components, making maintenance easier as well as saving lines of code, avoiding “the diamond problem” and reducing coupling.

[This section is a work in progress and it will be completed as soon as possible]

4.5.7 Coupling

Coupling is a term used to define the phenomenon where an edit to some part of a software snowballs into a bunch of edits in parts of the software that depend on the modified part, and the part that depend on the previously edited dependency, etc. . .

Introducing unnecessary coupling in our software will come back to bite us in the future, affecting maintainability in a very negative way, since any edit we make (for instance, to fix a bug) can potentially lead to edit the rest of the software (or game) we are writing.

This means that it’s in our best interest to reduce code coupling as much as possible, following the good principles of “nutshell programming” and following the SOLID principles, shown next.

[This section is a work in progress and it will be completed as soon as possible]

4.5.8 The DRY Principle

DRY is a mnemonic acronym that stands for “Don’t Repeat Yourself” and condenses in itself the principle of reducing repetition inside a software, replacing it with *abstractions* and by *normalizing data*.

This allows for each piece of code (and knowledge, since the DRY principle applies to documentation too) to be unambiguous, centralizing its responsibilities and avoiding repetition.

Violations of the DRY principle are called “WET” (write everything twice) solutions, which base themselves on repetition and give higher chances of mistakes and inconsistency.

4.5.9 SOLID Principles

SOLID is a mnemonic acronym that condenses five principles of good design, to make code and software that is understandable, flexible and maintainable.

- **Single Responsibility:** Each class should have a single responsibility, it should take care of one part of the software specification and each change to said specification should affect only said class. This means you should avoid the so-called “God Classes”, classes that take care of too much, know too much about the system and in a nutshell: have too much responsibility in your software.
- **Open-closed Principle:** Each software entity should be open to extension, but closed for modification. This means that each class (for instance) should be extensible, either via inheritance or composition, but it should not be possible to modify the class’s code. This is practically enforcing *Information Hiding*.
- **Liskov Substitution Principle:** Objects in a program should be replaceable with instances of their subtypes and the correctness of the program should not be affected. This is the base of inheritance and polymorphism, if by substituting a base class with one of its child (which should have a Child-is-a-Base relationship, for instance “Circle is a shape”) the program is not correct anymore, either something is wrong with the program, or the classes should not be in a “IS-A” relationship.
- **Interface Segregation:** Classes should provide many specific interfaces instead of one general-purpose interface, this means that no client should depend on methods that it doesn’t use. This makes the software easier to refactor and maintain, and reduces coupling.
- **Dependency Inversion:** Software components should depend on abstractions and not concretions. This is another staple of nutshell programming and OOP - Each class should make use of some other class’s interface, not its inner workings. This allows for maintainability and easier update and change of code, without having the changes snowball into an Armageddon of errors.

4.5.10 “Composition over Inheritance” design

[This section is a work in progress and it will be completed as soon as possible]

4.6 Designing entities as data

Sometimes it can be useful to design your entities as data, instead of making them into static objects that possibly require a new release of your product.

Designing your objects as data allows you to use configuration files to create, configure, tinker and extend your product, as well as allow for modifications by people who are fans of your game.

For instance, in a fantasy RPG you could have 3 types of enemies all defined as classes:

- Skeleton
- Zombie
- Ghost Swordsman

Which all have the same behaviour but different animations and sprites.

These classes can inherit from an “entity” abstract class which defines the base behaviour and then can be extended to create each unique enemy.

Another idea could be designing an “entity” class that can be instantiated, and have a configuration file that defines what each entity is and what its properties are.

An idea could be the following

Listing 10: Example of an entity declared as data

```
entity:
2  name: skeleton
   health: 10
4  damage_on_hit: 2.5
   spritesheet: "./skelly.png"
6  animations:
   walking:
8     start_sprite: 4
     frame_no: 4
10    duration: 0.2
   attacking:
12    start_sprite: 9
     frame_no: 2
14    duration: 0.1
```

With more complex building algorithms, it is possible to change behaviours and much more with just a configuration file, and this gives itself well to roguelike games, which random selection of enemies can benefit from an extension of the enemy pool. In fact, it's really easy to configure a new type of enemy and have it work inside the game without recompiling anything.

This allows for more readable code and a higher extensibility.

4.7 Reading UML diagrams

UML (Universal Modeling Language) is a set of graphical tools that allow a team to better organize and plan a software product. Diagrams are drawn in such a way to give the reader an overall assessment of the situation described while being easy to read and understand.

In this chapter we will take a look at 3 diagrams used in UML:

- Use Case Diagrams
- Class Diagrams
- Activity Diagrams

4.7.1 Use Case Diagrams

Use Case Diagrams are usually used in software engineering to gather requirements for the software that will come to exist. In the world of game development, use case diagrams can prove useful to have an “outside view” of our game, and understand how an user can interact with our game.

Here is an example of a use case diagram for a game:

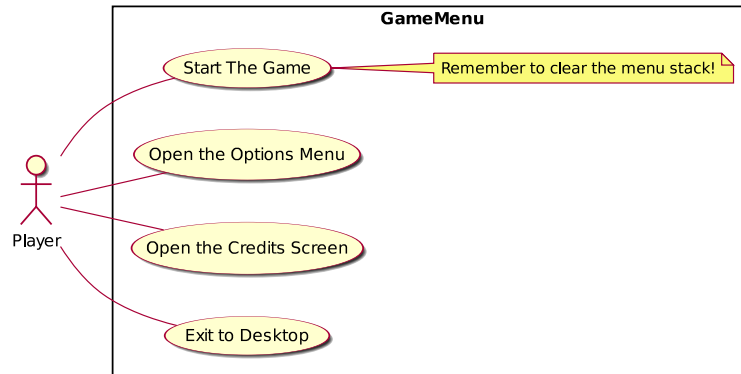


Figure 14: Example of a use case diagram

4.7.1.1 Actors

Actors are any entity that can interface with our system (in this case, our game) without being part of it. Actors can both be human, machines or even other systems.

Actors are represented with a stick figure and can inherit from each other: this will create an “IS-A” relationship between actors.

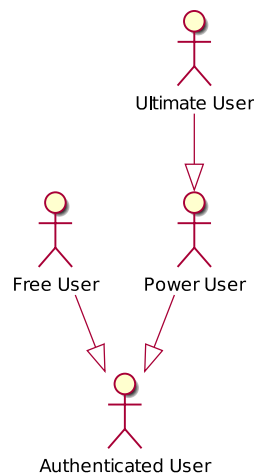


Figure 15: Example of an actor hierarchy

In the previous example, we can see that a “Free User” is an “Authenticated User”, as well as a “Power User” (which could be a paying user) is itself an “Authenticated User” while an “Ultimate User” (which could be a higher tier of paying user) is a “Power User” (thus has all the “Power User” capabilities, plus some unique) and by transitive property an “Authenticated User”.

As seen, inheritance between actors is represented with a solid line with a hollow closed arrow. Such arrow points towards the “supertype” or “parent” from which the subject (or “subtype”, or “child”) inherits.

This representation will come back in the UML language for other diagrams too.

4.7.1.2 Use Cases

Use cases represent the functionalities that our system offers, and the relationships between them.

Use cases are represented with an ellipse with the name of the use case inside. Choosing the right name for a use case is extremely important, since they will represent the functionality that will be developed in our game.

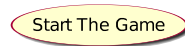


Figure 16: Example of a use case

4.7.1.2.1 Inheritance

As with many other elements used in UML, use cases can inherit from each other. Inheritance (also called “Generalization”) is represented with a closed hollow arrow that points towards the parent use case.

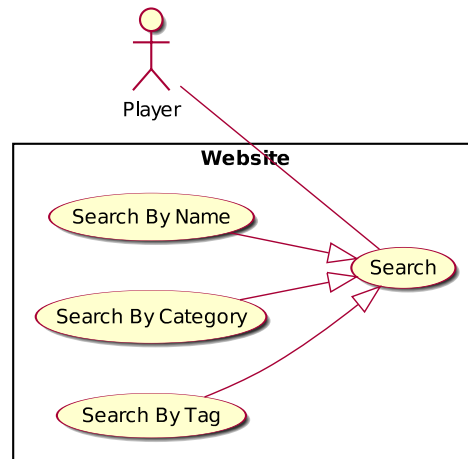


Figure 17: Example of a use case hierarchy

4.7.1.2.2 Extensions

Use case extensions specify how and when optional behaviour takes place. Extended use cases are meaningful on their own and are independent from the extending use case, while the extending use case define the optional behaviour that may not have much sense by itself.

Extensions are represented via a dashed line with an open arrow on the end, labeled with the `<<extend>>` keyword, pointing towards the extending use case.

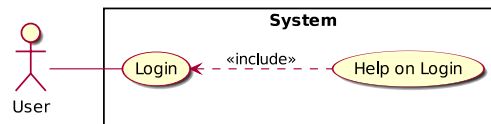


Figure 18: Example of a use case extension

4.7.1.2.3 Inclusions

Inclusions specify how the behaviour of the included use case is inserted in the behaviour of the including use case. Inclusions are usually used to simplify large use cases by splitting them or extract common behaviours of two or more use cases.

In this situation, the including use case **is not** complete by itself.

Inclusions are represented via a dashed line with an open arrow on the end, labeled with the `<<include>>` pointing towards the included use case.

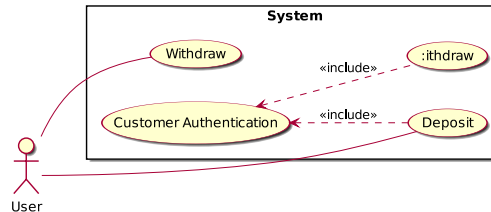


Figure 19: Example of a use case inclusion

4.7.1.3 Notes

In use case diagrams, as well as in many other UML diagrams, notes are used to jot down conditions, comments and everything useful to better understanding the diagram that cannot be conveyed through a well definite structure inside of UML.

Notes are shaped like a sheet of paper with a folded corner and are usually connected to the diagram with a dashed line. Each note can be connected to more than one piece of the diagram.

You can see a note at the beginning of this chapter, in the use case diagram explanation.

4.7.1.4 Sub-Use Cases

Use cases can be further detailed by creating sub-use cases, like the following example.

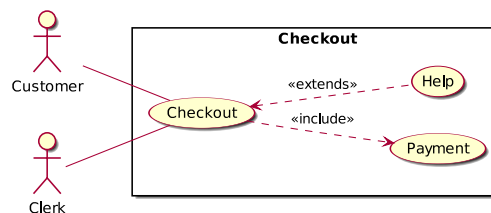


Figure 20: Example of a sub-use case

4.7.2 Class Diagrams

4.7.2.1 Classes

Class diagrams are used a step after analyzing your game, since they are used for planning classes. The central element of a class diagram is the “class”, which is represented as follows:

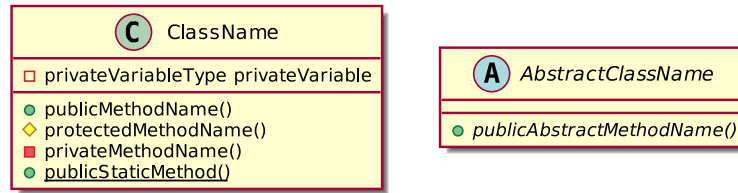


Figure 21: Example of classes in UML

Classes are made up by a class name, which is shown on top of the class; abstract classes are shown with a name in *italics*.

Public members are highlighted by a “+” symbol (or in our case, a green symbol) before their name, protected members use a “#” symbol (or a yellow symbol) and private members use a “-” symbol.

Static members are shown with an underlined name, while abstract members are shown in *italics*.

4.7.2.2 Relationships between classes

Expressing only single classes on their own doesn’t give UML a lot of expressive power when it comes to planning your games. Here we’ll take a quick look at the most used relationships between classes.

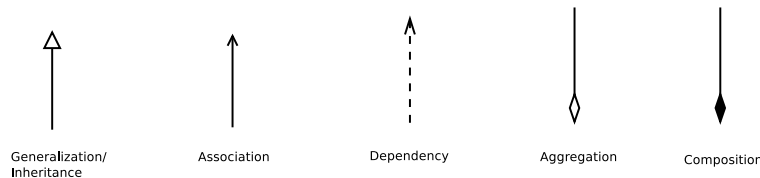


Figure 22: Relationships between classes in an UML Diagram

4.7.2.2.1 Inheritance

Inheritance is represented via a hollow closed arrow head that points towards the base class (exactly like in **Actor inheritance**), this means that the classes are in a “supertype and subtype” relationship.

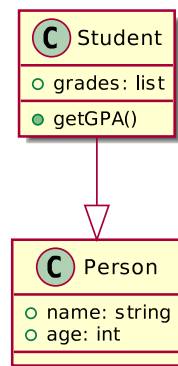


Figure 23: Example of inheritance in UML class diagrams

In this example we say that “Student IS-A Person” and inherits all Person’s methods and fields.

4.7.2.2.2 Association

Association represents a static relationship between two classes. This is usually represented with a solid line with an arrow. The arrow usually shows the reading order of the association, so if you see an “Author” class and a “Book” class, the “wrote” association will be pointing from the “Author” to the “Book” class.

In case the relationship is bi-directional, the arrow points are omitted, leaving only a solid line between the two classes.

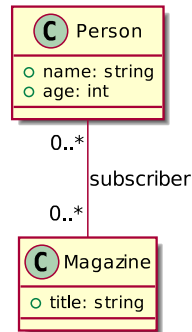


Figure 24: Example of association in UML class diagrams

An example of an association is the relationship between a “Person” and a “Magazine”, such relationship is the “Subscription”. In this case the relationship is bi-directional, since a “Magazine” can be subscribed by many people, but a single “Person” can subscribe to many “Magazine”s.

4.7.2.2.3 Aggregation and Composition

Aggregation is a special case of the association relationship, and represents a more specific case of it. Aggregation is a variant of a “has-a” relationship and represents a part-whole relationship.

Aggregation is represented with a hollow diamond and a line that points to the *contained* class, classes involved in an aggregation relationships *do not* have their life cycles dependant one another, that means that if the container is destroyed, the contained objects will keep on living. An example could be a teacher and their students, if the teacher dies the students will keep on living.

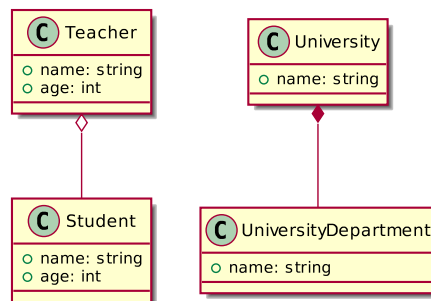


Figure 25: Example of aggregation and composition in UML class diagrams

Composition is represented with a filled diamond instead than a hollow one, in this case there is a life cycle dependency, so when the container is destroyed the contents are destroyed too. Like when a university is dissolved, its departments will cease to exist.

4.7.2.2.4 Dependency

The dependency relationship is the one that gives us the least amount of coupling, it represents a “supplier-client” relationships, where the supplier supplies its functions (methods) to the client. The association is represented in a dashed line with an open arrow that points towards the supplier.

This means that the client class requires, needs or depends on the supplier.

There are many categories of dependency, like `<<create>>` or `<<call>>` that explain further the type of dependency exists between the supplier and the client.

An example could be between a “Car Factory” and a class “Car”: the “CarFactory” class depends on the “Car” class, and such dependency is an instantiation dependency.

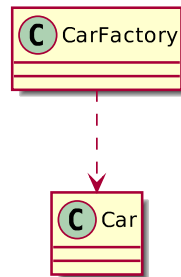


Figure 26: Example of dependency in UML class diagrams

4.7.2.3 Notes

As with Use Case diagrams, class diagrams can make use of notes too, and the graphical language used to represent them is exactly the same one used in the Use Case Diagrams.

4.7.2.4 Interfaces

Sometimes there is a need to convey the concept of “interface” inside a UML class diagram, that can easily be done in 2 ways:

- By using the class construct, with the keyword (called “stereotype”) `<<interface>>` written on top of it;
- By using the “lollipop notation” (called “interface realization”).

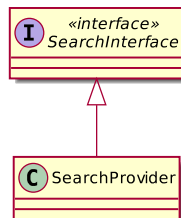


Figure 27: Defining an interface in UML

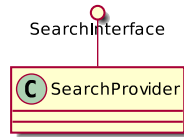


Figure 28: Interface Realization in UML

4.7.3 Activity Diagrams

Activity diagrams are the more powerful version of flow charts: they represent the flux of an activity in detail, allowing to have a better understanding of a process or algorithm.

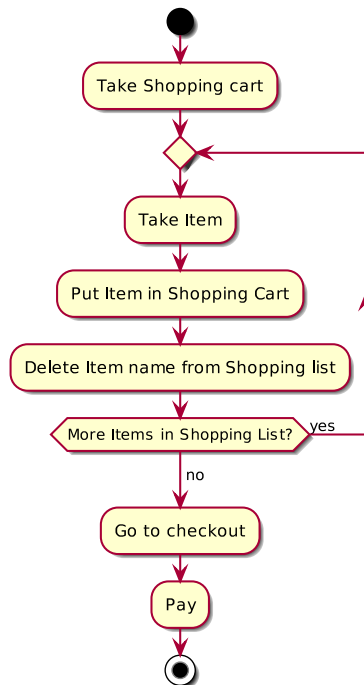


Figure 29: Example of an activity diagram

4.7.3.1 Start and End Nodes

Each diagram begins with a “start node”, represented with a filled black circle, and they end with an “end node” which is represented with a filled black circle inside of a hollow circle.



Figure 30: Example of activity diagrams start and end nodes

4.7.3.2 Actions

Each action taken by the software is represented in the diagram via a rounded rectangle, a very short description of the action is written inside the rounded rectangle space.

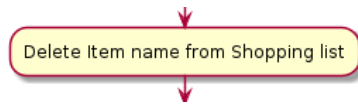


Figure 31: Example of Action in activity diagrams

4.7.3.3 Decisions (Conditionals) and loops

Decisions and loops are enclosed in diamonds. If a condition needs to be written, the diamond can become an hexagon, to make space for the condition to be written or guards can be used to express the condition.

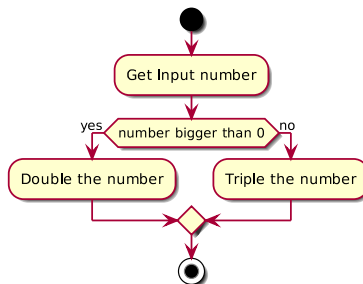


Figure 32: Example of decision, using hexagons to represent the condition

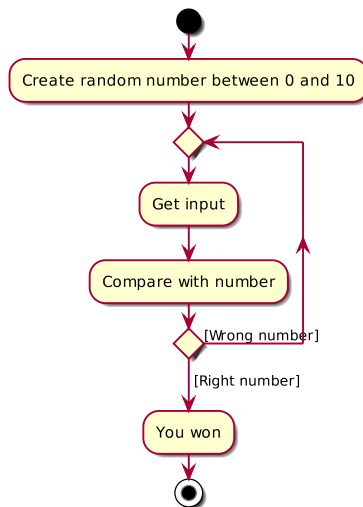


Figure 33: Example of loops, using guards to represent the condition

All the branches that depend on a condition or are part of a loop start and end on a diamond, as shown below.

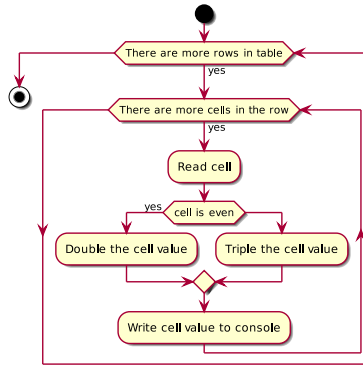


Figure 34: Example of how nested loops and conditions are performed

4.7.3.4 Synchronization

Synchronization (or parallel processing) is represented in activity diagrams by using filled black bars that enclose the concurrent processes: the bars are called “synchronization points” or “forks” and “joins”

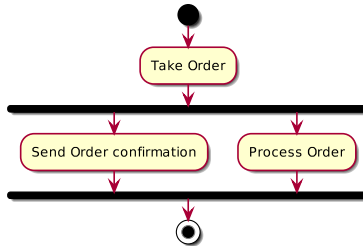


Figure 35: Example of concurrent processes in activity diagrams

In the previous example, the activities “Send Order Confirmation” and “Process Order” are processed in parallel, independently from each other, the first activity that finishes will wait until the other activity finishes before entering the end node.

4.7.3.5 Signals

Signals are used to represent how activities can be influenced or modified from outside the system. There are two symbols used to represent signals.

The “Sent Signal” symbol is represented with a convex pentagon (which reminds an arrow going away from our system), while the “Received Signal” is represented by a concave pentagon (which reminds a “slot” where the “sent signal” symbol can connect to).

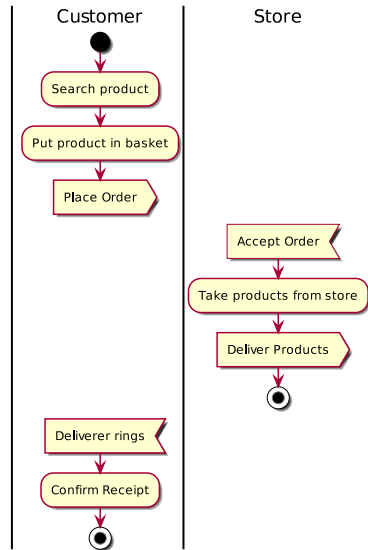


Figure 36: Example of signals in activity diagrams

[This section is a work in progress and it will be completed as soon as possible]

4.7.3.6 Swimlanes

Swimlanes are a way to organize and group related activities in columns. For instance a shopping activity diagram can have the “Customer”, “Order”, “Accounting” and “Shipping” swimlanes, each of which contains activities related to their own categories.

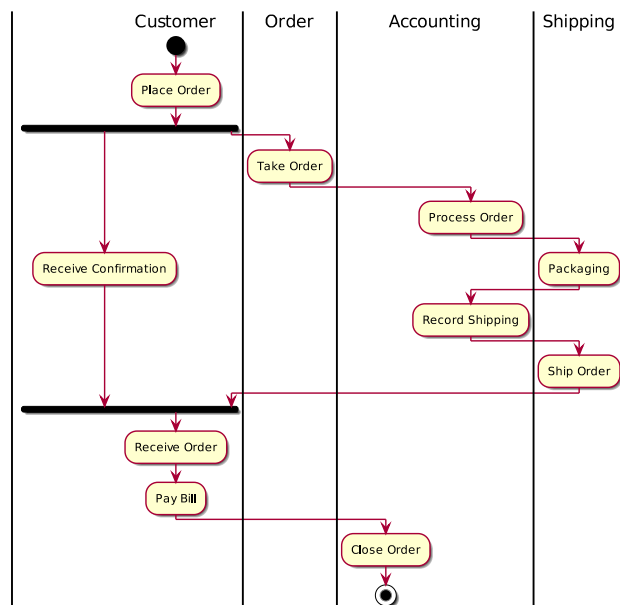


Figure 37: Example of swimlanes in activity diagrams

4.7.3.7 Notes

As with Use Case and Class diagrams, Activity Diagrams can make use of notes, in the same way as the other two diagrams we presented in this book do.

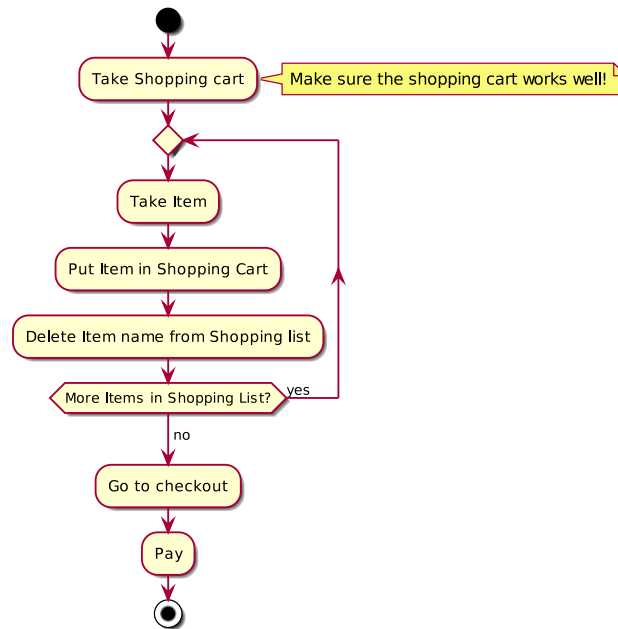


Figure 38: Example of a note inside of an activity diagram

4.7.3.8 A note on activity diagrams

The components of activity diagrams shown here are just a small part of the used components, but they should be enough to get you started designing and reading most of the activity diagrams that exist.

4.8 Generic Programming

Sometimes it may be necessary (mostly in the case of containers) to have the same kind of code to work on different data types, which means that we need to **abstract types into variables** and be able to code accounting for such types.

Generic Programming is a blanket-term that defines a style of computer programming where algorithms are written in terms of “to be specified later” data types, this usually applies to languages that make use of *static typing*_g.

4.9 Advanced Containers

This section is dedicated to give some basic explanation of some advanced containers that are used in computer science, allowing us to make an informed choice when we want to implement some even more advanced containers in the future.

We will include big-O performance counters for the basic functions of: adding/removing an item at the beginning, adding/removing an item at the end, adding/removing an item in an arbitrary position and indexing at a certain position.

This section is in no way exhaustive, but should be enough to make an informed decision on what containers to use for our components, according to necessities.

4.9.1 Dynamic Arrays

In many languages, arrays are sized statically, with a size decided at compile time. This severely limits the array's usefulness.

Dynamic Arrays are a wrapper around arrays, allowing it to extend its size when needed. This usually entails some additional operations when inserting or deleting an item.

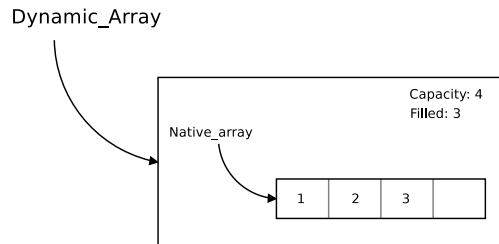


Figure 39: Dynamic Arrays Reference Image

4.9.1.1 Performance Analysis

Indexing an item is immediate, since arrays allow to natively index themselves.

Inserting an item at the beginning is a heavy task, since it requires either moving all the present items or rebuilding the internal native array. Such operations require copying or moving each element, giving us a time complexity averaging on $O(n)$.

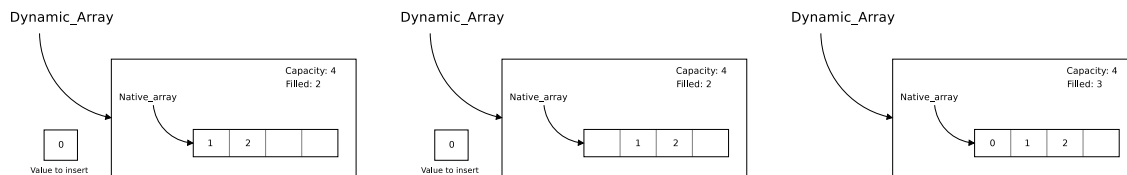


Figure 40: Adding an element at the beginning of a Dynamic Array

Inserting an item at the end, if we keep a pointer to the last item inserted, is an operation that usually happens immediately (time complexity $O(1)$), but when the array is full, we need to instantiate a new native array (usually double the size of the current one) and copy all elements inside the new array (operation that has time complexity of $O(n)$). Since the number of $O(1)$ operations outweighs by a long shot the number of $O(n)$ operations, it's possible to demonstrate that in the long run appending an item at the end of a dynamic array has a time complexity averaging around $O(1)$.

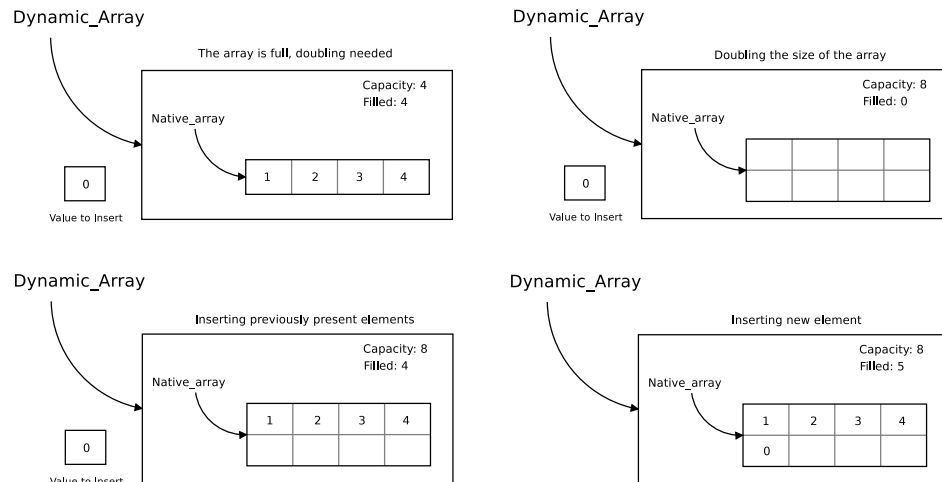


Figure 41: Adding an element at the end of a Dynamic Array

Inserting an item in an arbitrary position, much like inserting an item at the beginning requires moving some items further into the array, potentially all of them (when the arbitrary position is the beginning of the array), thus giving us a time complexity of $O(n)$. Such operation could trigger an array resize, which has no real influence on the estimate.

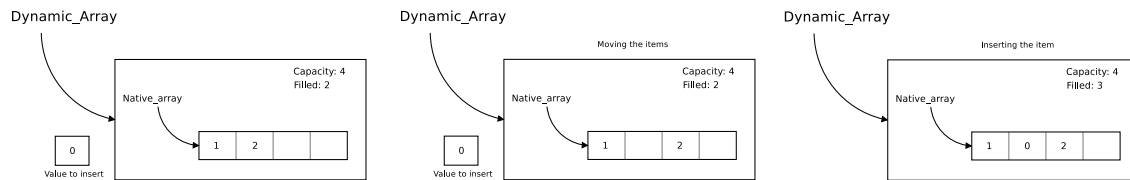


Figure 42: Adding an element at an arbitrary position of a Dynamic Array

Some implementations of the Dynamic Arrays try to save space when the number of items goes lower than $\frac{1}{4}$ of the array capacity during a deletion, the internal array is rebuilt with half the size. Such operation has a time complexity of $O(n)$.

Some other implementations use a $\frac{1}{4}/\frac{3}{4}$ rule, halving the array capacity when the item deletion brings the number of items lower than $\frac{1}{4}$ of the array and doubling it when an insertion makes the number of elements higher than $\frac{3}{4}$ of the array capacity.

Note: Not all programming languages have native support for arrays, for instance Python uses lists.

[This section is a work in progress and it will be completed as soon as possible]

Table 14: Performance table for Dynamic Arrays

Operation	Average Cost
Indexing	$O(1)$
Insert/Delete At Beginning	$O(n)$
Insert/Delete At End	$O(1)$ amortized
Insert/Delete at position	$O(n)$

Table 15: Summary Table for Dynamic Arrays

Container Name	Dynamic Array
When To Use it	All situations that require direct indexing of a container, but insertions and removals are not extremely common, and usually take the form of “push back” (insertion at the end)
Advantages	Direct Indexing, Fast iteration through all the elements, given by the fact that arrays are stored compact in memory, fast appending.
Disadvantages	Slow insertions in arbitrary positions and at the head of the array.

4.9.2 Linked Lists

Linked Lists are a data structure composed by “nodes”, each node contains data and a reference to the next node in the linked list. Differently from arrays, nodes may not be contiguous in memory, which makes indexing problematic.

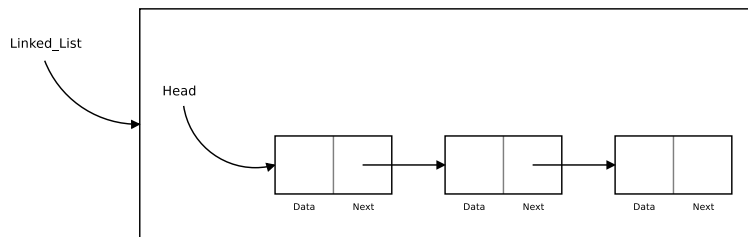


Figure 43: Linked List Reference Image

Some implementations feature a pointer to the last element of the list, to make appending items at the end easier and quicker.

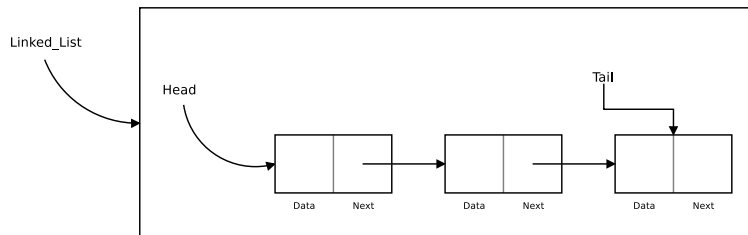


Figure 44: Double-Ended Linked List Reference Image

4.9.2.1 Performance Analysis

Since we only have a handler on the first node, indexing requires us to scan all the elements until we reach the one that was asked for. This operation has a potential time complexity of $O(n)$.

Inserting an item at the beginning is immediate, we just need to create a new node, make it point at the current head of the list and then update our “handle” to point at the newly created node. The number of operations is independent of how many data we already have, so the time complexity is $O(1)$.

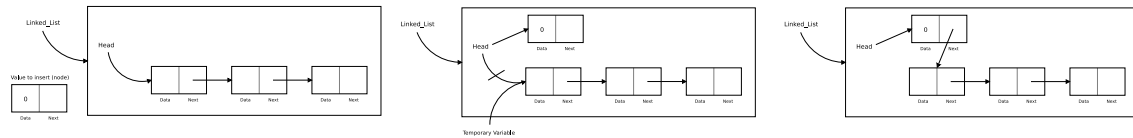


Figure 45: Inserting a new node at the beginning of a linked list

Appending an item at the end has a time complexity that varies depending on the chosen implementation: if the list has a reference to the final node, we just need to create a new node, update the final node's reference (usually called “next”) to point at the new node and then update the reference to the final node to point at the newly created node (time complexity $O(1)$). If our queue doesn't have such reference, we will need to scan the whole list to find the final node (time complexity $O(n)$).

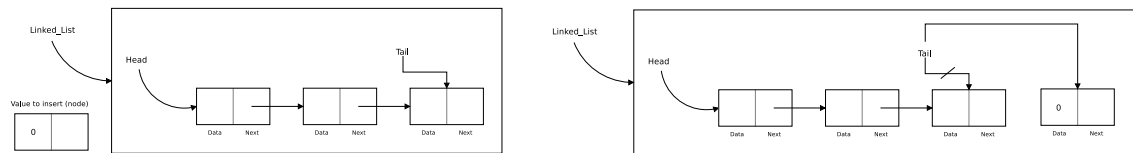


Figure 46: Inserting a new node at the end of a (double-ended) linked list

Inserting at an arbitrary position requires us to scan the list until we find the position that we want, after that we just need to split and rebuild the references correctly, which is a fast operation.

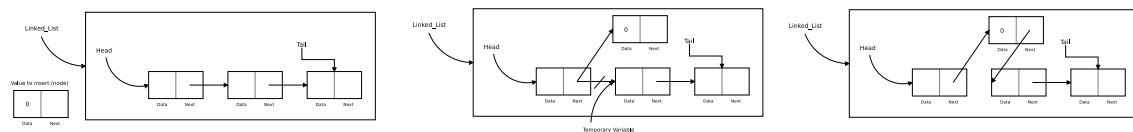


Figure 47: Inserting a new node at an arbitrary position in a (double-ended) linked list

[This section is a work in progress and it will be completed as soon as possible]

Table 16: Performance table for Linked Lists

Operation	Average Cost
Indexing	$O(n)$
Insert/Delete At Beginning	$O(1)$
Insert/Delete At End	$O(1)$ for double-ended, $O(n)$ otherwise
Insert/Delete at position	time to search + $O(1)$

Table 17: Summary Table for Linked Lists

Container Name	Linked List
When To Use it	All situations that require quick insertions/removals, either on the head or the tail (used as stacks or queues).
Advantages	Very fast insertions/removals, quite fast iteration through all the elements.
Disadvantages	Slow indexing at an arbitrary position. Sorting can be complex.

4.9.3 Doubly-Linked Lists

A doubly-linked list is a variation of a linked list where each node not only has a reference to its successor, but also a reference to its predecessor. This allows for easy processing of the list in reverse, without having to create algorithms that entail a huge overhead.

All the operations of insertion, indexing and deletion are performed in a similar fashion to the classic singly-linked list we saw earlier, just with an additional pointer to account for.

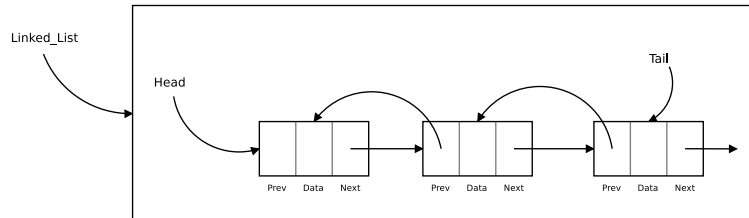


Figure 48: Doubly Linked List Reference Image

[This section is a work in progress and it will be completed as soon as possible]

Table 18: Performance table for Doubly-Linked Lists

Operation	Average Cost
Indexing	$O(n)$
Insert/Delete At Beginning	$O(1)$
Insert/Delete At End	$O(1)$
Insert/Delete at position	time to search + $O(1)$

Table 19: Summary Table for Linked Lists

Container Name	Doubly-Linked List
When To Use it	All situations that require quick insertions/removals, either on the head or the tail (stacks or queues) or iterating through an entire list, forwards or backwards.
Advantages	Very fast insertions/removals, quite fast iteration through all the elements. Possibility of easily iterating the list in reverse order.
Disadvantages	Slow indexing at an arbitrary position. Sorting can be complex.

4.9.4 Hash Tables

Hash Tables are a good way to store **unordered data** that can be referred by a “key”. These structures have different names, like “maps”, “dictionaries” or “hash maps”.

The idea behind a hash map is having a key subject to a *hash function*_g that will decide where the item will be positioned in the internal structure.

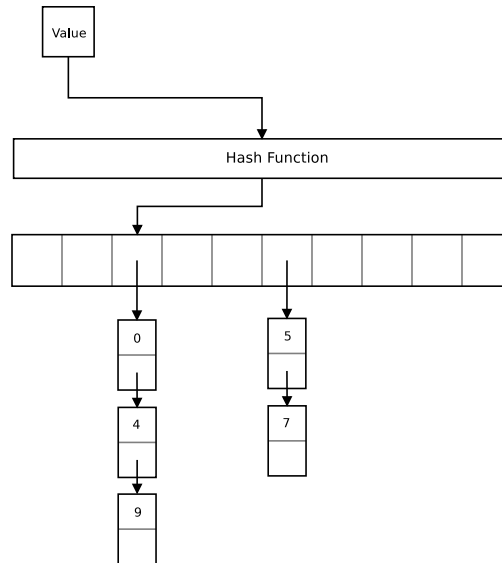


Figure 49: Hash Table Reference Image (Hash Table with Buckets)

The simplest way to implement a hash table is using an “array with buckets”: an array where each cell has a reference to a linked list.

On average, finding an item requires passing the key through the hash function, such hash function will tell us where the item is in our internal structure immediately. Thus giving a time complexity of $O(1)$.

Inserting has more or less the same performance, the key gets worked through the hash function, deciding which linked list will be used to store the item.

Deletion works in the same fashion, passing the key through the hash function and then deleting the value; giving a time complexity of $O(1)$

Table 20: Performance table for Hash Tables

Operation	Average Cost
Searching	$O(1)$
Insert	$O(1)$
Delete	$O(1)$

Table 21: Summary Table for Hash Tables

Container Name	Hash Table
When To Use it	All situations that require accessing an element by a well-defined key quickly. Building unordered data sets.
Advantages	Fast insertions/removals, direct indexing (in absence of hash collisions) by key.
Disadvantages	In case of a bad hashing function, it reverts to the performance of a linked list, cannot be ordered.

4.9.5 Binary Search Trees (BST)

Binary search trees, sometimes called “ordered trees” are a container that have an “order relation” between their own elements.

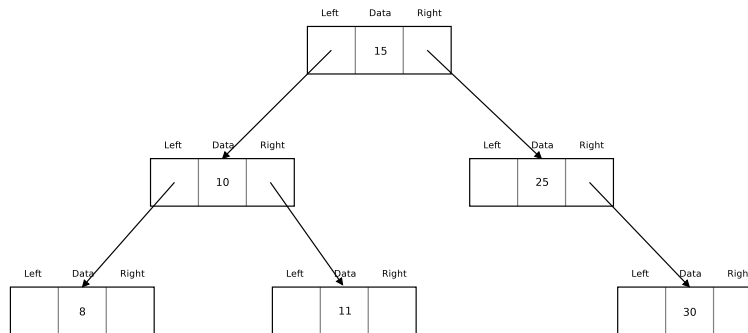


Figure 50: Binary Search Tree Reference

The order relation allows us to have a tree that is able to distinguish between “bigger” and “smaller” values, thus making search really fast at the price of a tiny slowdown in insertion and deletion.

Searching in a BST is easy, starting from the root, we check if the current node is the searched value; if it isn't we compare the current node's value with the searched value.

If the searched value is greater, we search on the right child. If it is smaller, we continue our search on the left child.

Recursively executing this algorithm will lead us to find the node, if present. Such algorithm has a $O(\log(n))$ time complexity.

In a similar fashion, insertion will recursively check subtrees until the right spot of the value is found. The insertion operation has the same time complexity as searching: $O(\log(n))$.

Deletion is a bit more conceptually complex, since it's necessary to maintain the ordering of the nodes. Such operation has a time complexity of $O(\log(n))$.

Table 22: Performance table for Binary Search Trees

Operation	Average Cost
Searching	$O(\log(n))$
Insert	$O(\log(n))$
Delete	$O(\log(n))$

Table 23: Summary Table for Binary Search Trees

Container Name	Binary Search Tree
When To Use it	Situations that require good overall performance and requires fast search times.
Advantages	Good insertion and removal times, searching on this structure is fast.
Disadvantages	Given the nature of the data structure, there is no direct indexing, nor ordering.

4.9.6 Heaps

Heaps are a tree-based data structure where we struggle to keep a so-called “heap property”. The heap property defines the type of heap that we are using:

- **Max-Heap:** For each node N and its parent node P , we’ll always have that the value of P is always greater or equal than the value of N ;
- **Min-Heap:** For each node N and its parent node P , we’ll always have that the value of P is always less or equal than the value of N ;

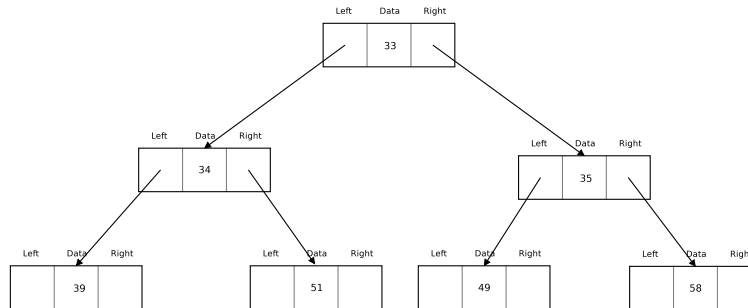


Figure 51: Heap Reference Image (Min-Heap)

Heaps are one of the maximally efficient implementation of priority queues, since the highest (or lowest) priority item is stored in the root and can be found in constant time.

Table 24: Performance table for Heaps

Operation	Average Cost
Find Minimum	$O(1)$ to $O(\log(n))$, depending on the implementation
Remove Minimum	$O(\log(n))$
Insert	$\Theta(1)$ to $O(\log(n))$ depending on the implementation

Table 25: Summary Table for Heaps

Container Name	Heap
When To Use it	All situations where you require to find and/or extract the minimum or maximum value in a container quickly; like priority queues.
Advantages	Good general time complexity, maximum performance when used as priority queues.
Disadvantages	No inherent ordering, there are better solutions for general use.

[This section is a work in progress and it will be completed as soon as possible]

4.9.7 Stacks

Stacks are a particular data structure, they have a limited way of working: you can only put or remove items on top of the stack, plus being able to “peek” on top of the stack.

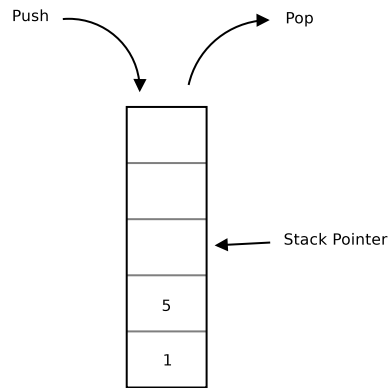


Figure 52: How a stack works

Stacks are LIFO (Last in - First Out) data structures, and can be implemented with both a linked list or a cleverly-indexed array.

Depending on the single implementation, the operation used to “pop” an item from the stack will also return the element, ready to be used in an upcoming computation.

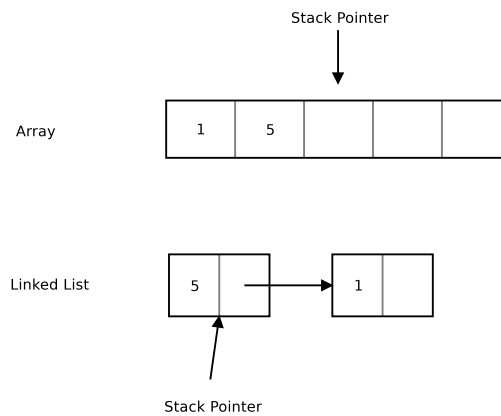


Figure 53: Array and linked list implementations of a stack

[This section is a work in progress and it will be completed as soon as possible]

4.9.8 Queues

Queues are the exact opposite of stacks, they are FIFO (First in - First Out) data structures: you can put items on the back of the queue, while you can remove from the head of the queue.



Figure 54: How a queue works

Depending on the single implementation, the operation used to “dequeue” an item from the queue will also return the element just removed, ready to be used in an upcoming computation.

As with stacks, queues leverage limitations in their way of working for greater control over the structure itself. Usually queues are implemented via linked lists, but can also be implemented via arrays, using multiple indexes and index-wrapping when iterating.

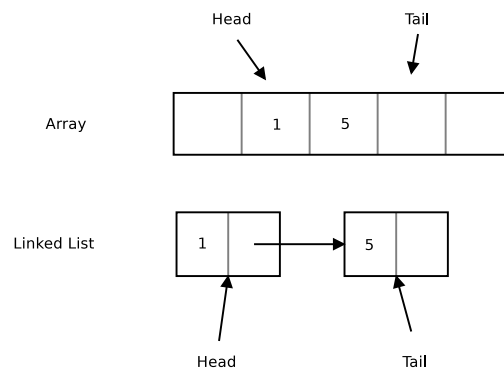


Figure 55: Array and linked list implementation of a queue

[This section is a work in progress and it will be completed as soon as possible]

4.9.9 Circular Queues

Circular Queues are a particular kind of queues that are infinitely iterable, every time an iterator goes after the last element in the queue, it will wrap around to the beginning.

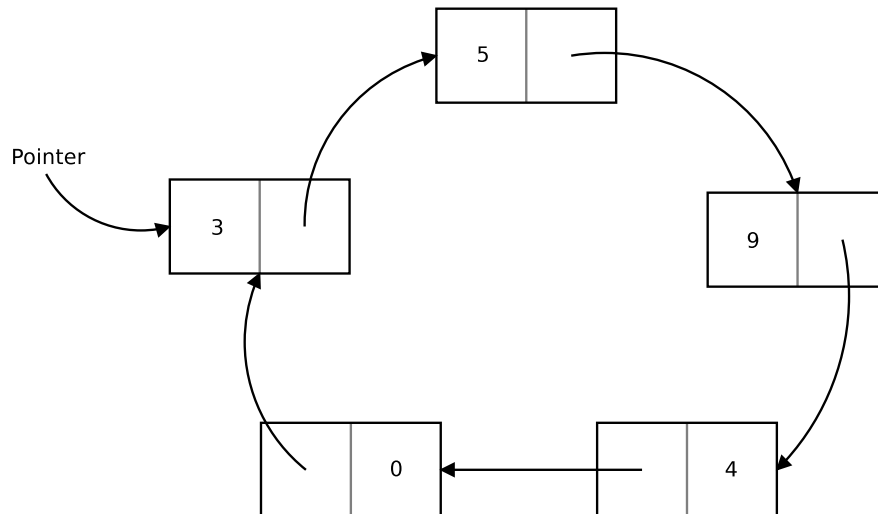


Figure 56: How a circular queue works

Circular Queues can be implemented via linked lists or cleverly indexed arrays, with all the advantages and disadvantages that such structures entail.

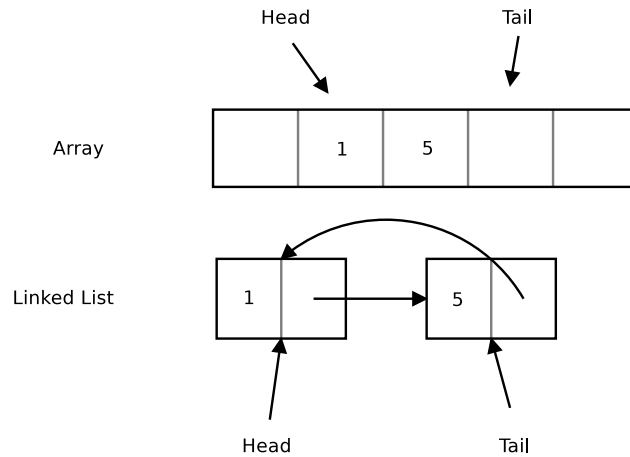


Figure 57: Array and linked list implementation of a circular queue

[This section is a work in progress and it will be completed as soon as possible]

4.10 Introduction to MultiTasking

When it comes to humans, we are used to have everything at our disposal immediately, but when it comes to computers, each processing unit (CPU) is usually able to perform only one task at a time.

To allow for multi-tasking (doing many activities at once), the CPU switches between tasks at high speeds, giving us the illusion that many things are happening at once. There are many methods to ensure multi-tasking without *process starvation*_g, the most used is **pre-emption** where there are forced context switches between processes, to avoid one hogging the CPU.

4.10.1 Co-Routines

[This section is a work in progress and it will be completed as soon as possible]

4.11 Introduction to MultiThreading

When it comes to games and software, we usually think of it as a single line of execution, branching to (not really) infinite possibilities; but when it comes to games, we may need to dip our toes into the world of multi-threaded applications.

4.11.1 What is MultiThreading

MultiThreading means that multiple threads exist in the context of a single process, each thread has an independent line of execution but all the threads share the process resources.

In a game, we have the “Game Process”, which can contain different threads, like:

- World Update Thread
- Rendering Thread
- Loading Thread
- ...

MultiThreading is also useful when we want to perform concurrent execution of activities.

4.11.2 Why MultiThreading?

Many people think of MultiThreading as “parallel execution” of tasks that leads to faster performance. That is not always the case. Sometimes MultiThreading is used to simplify data sharing between flows of execution, other times threads guarantee lower latency, other times again we may *need* threads to get things working at all.

For instance let’s take a loading screen: in a single-threaded application, we are endlessly looping in the input-update-draw cycle, but what if the “update” part of the cycle is used to load resources from a slow storage media like a Hard Disk or even worse, a disk drive?

The update function will keep running until all the resources are loaded, the game loop is stuck and no drawing will be executed until the loading has finished. The game is essentially hung, frozen and your operating system may even ask you to terminate it. In this case we need the main game loop to keep going, while something else takes care of loading the resources.

4.11.3 Thread Safety

Threads are concurrent execution are powerful tools in our “programmer’s toolbox”, but as with all powers, it has its own drawbacks.

4.11.3.1 Race conditions

Imagine a simple situation like the following: we have two threads and one shared variable.

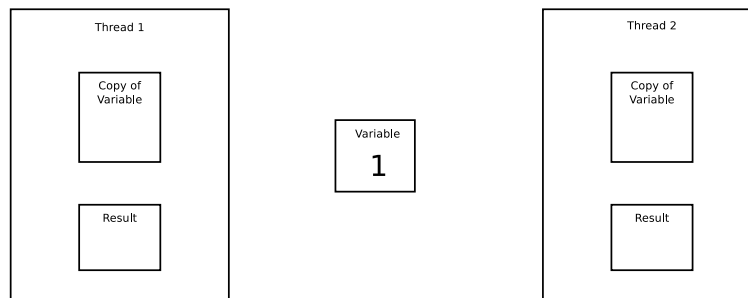


Figure 58: Two threads and a shared variable

Both threads are very simple in their execution: they read the value of our variable, add 1 and then write the result in the same variable.

This seems simple enough for us humans, but there is a situation that can be really harmful: let’s see, in the following example each thread will be executed only once. So the final result, given the example, should be “3”.

First of all, let’s say Thread 1 starts its execution and reads the variable value.

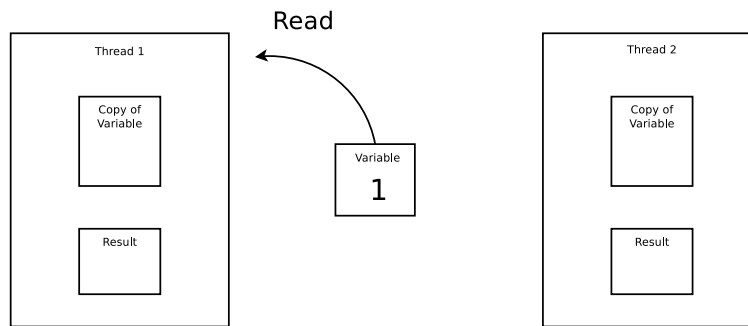


Figure 59: Thread 1 reads the variable

Now, while Thread 1 is calculating the result, Thread 2 (which is totally unrelated to Thread 1) starts its execution and reads the variable.

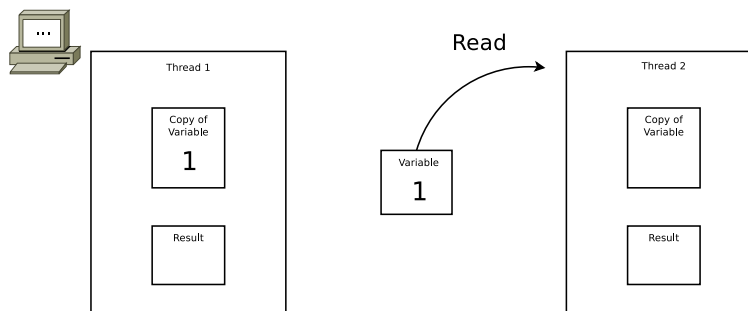


Figure 60: While Thread 1 is working, Thread 2 reads the variable

Now Thread 1 is finishing its calculation and writes the result into the variable.

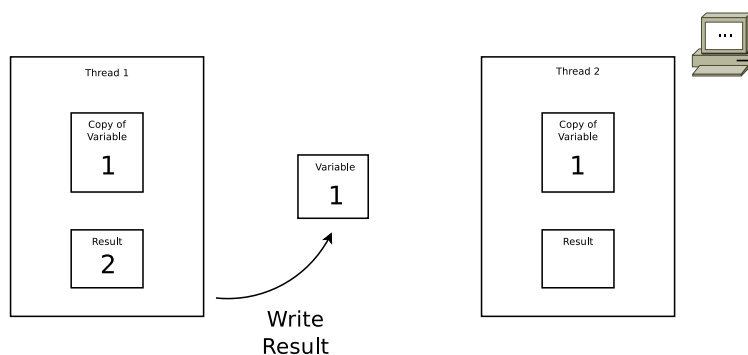


Figure 61: Thread 1 writes the variable

After That, Thread 2 finishes its calculation too, and writes the result into the variable too.

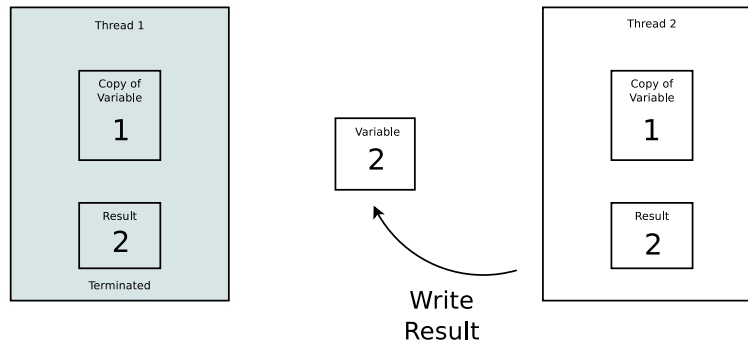


Figure 62: Thread 2 writes the variable

Something is not right, the result should be “3”, but it’s “2” instead.



Figure 63: Both Threads Terminated

We just experienced what is called a **“race condition”**: there is no real order in accessing the shared variable, so things get messy and the result is not deterministic. We don’t have any guarantee that the result will be right all the time (or wrong all the time either).

4.11.3.2 Critical Regions

Critical Regions (sometimes called “Critical Sections”) are those pieces of code where a shared resource is used, and as such it can lead to erroneous or unexpected behaviours. Such sections must be protected from concurrent access, which means only one process or thread can access them at one given time.

4.11.4 Ensuring determinism

Let’s take a look at how to implement multithreading in a safe way, allowing our game to perform better without non-deterministic behaviours. There are other implementation approaches (like thread-local storage and re-entrancy) but we will take a look at the most common here.

4.11.4.1 Immutable Objects

The easiest way to implement thread-safety is to make the shared data immutable. This way the data can only be read (and not changed) and we completely remove the risk of having it changed by another thread. This is an approach used in many languages (like Python and Java) when it comes to strings. In those languages strings are immutable, and “mutable operations” only return a *new string* instead of modifying the existent one.

4.11.4.2 Mutex

Mutex (Short for **mutual exclusion**) means that the access to the shared data is serialized in a way that only one thread can read or write to such data at any given time. Mutual exclusion can be achieved via algorithms (be careful of *out of order execution*_g), via hardware or using “software mutex devices” like:

- Locks (known also as *mutexes*)
- Semaphores
- Monitors
- Readers-Writer locks
- Recursive Locks
- ...

Usually these multithreaded functionalities are part of the programming language used, or available via libraries.

4.11.4.3 Atomic Operations

[This section is a work in progress and it will be completed as soon as possible]

5 Project Management Basics and tips

Those who plan do better than those who do not plan even though they rarely stick to their plan.

Winston Churchill

Project management is a very broad topic but I feel that some basics and tips should be covered in this book, as knowing some project management can save you a lot of headaches and can make the difference between success and a colossal failure.

5.1 The figures of game design and development

Before delving into the topic at hand, we need to familiarize ourselves with the main figures that are involved in the process of game design and development, since you'll probably (if you are the only developer of your game) have to take upon all their tasks.

5.1.1 Producer/Project Manager

The producer is a figure that has experience in many fields and has an overall view of the project. They essentially keep the project together.

Their duties are:

- Team Building (and its maintenance too);
- Distributing duties and responsibilities;
- Relations with the media.

5.1.2 Game Designer

The game designer takes care of the game concept, usually working with really specific software, usually provided by the programmers in the team (like specific level editors).

They design balanced game mechanics, manage the learning curve and take care of level design too.

5.1.3 Writer

Writers are the ones who can help you give your game its own story, but also help with things that are outside the mere game itself.

Some of their jobs include:

- Writing tutorial prompts;
- Writing narration;
- Writing dialogue;
- Writing pieces for the marketing of your game (sometimes known as "Copywriting").

5.1.4 Developer

Logic and mathematics are the strong suit of programmers, the people who take care of making the game tick, they can also have many specializations like:

- Problem Solver
- Game mechanics programmer;
- Controls programmer;
- AI developer;
- Visuals Programmer;

- Networking programmer;
- Physics programmer;
- ...

5.1.5 Visual Artist

In 2D games visual art is as important as in 3D games and good graphics can really boost the game's quality greatly, as bad graphics can break a game easily.

Among visual artists we can find:

Both in 2D and 3D games:

- 2D Artists;
- Animators;
- Conceptual Artists.

In 3D games:

- 3D Modelers;
- Texture Artists;
- Environmental Artists.

5.1.6 Sound Artist

As with graphics, sound and music can make or break a game. Sound artists may also be musicians, and their task is to create audio that can be used in a video game, like sound effects, atmospheres or background music.

5.1.7 Tester

Probably the most important job in a game development team, testing needs people with high attention to detail, as well as the ability to handle stress well.

Testers are able to find, describe and help you reproduce bugs and misbehaviours of your game.

5.2 Some generic tips

5.2.1 Be careful of feature creep

The “it would be cool to” trap, formally called “feature creep”, is a huge problem in all projects that involve any amount of passion in them.

Saying “it would be cool to do xxxx: let's implement it!” can spiral out of control and make us implement new features forever, keeping us from taking care of the basics that make a good game (or make a game at all).

Try to stick to the basics first, and then eventually expand when your game is already released, if it's worth it: First make it work, only then make it work well.

5.2.2 On project duration

When it comes to project management, it's always tough to gauge the project duration, so it can prove useful to remember the following phrase:

“If you think a project would last a month, you should add a month of time for unforeseen events. After that, you should add another month for events that you really cannot foresee.”

This means that projects will last at least 3 times the time you foresee.

5.2.3 Brainstorming: the good, the bad and the ugly

Brainstorming is an activity that involves the design team writing down all the ideas they possibly can (without caring about their quality yet).

This is a productive activity to perform at the beginning of the game development and design process, but it can be a huge source of feature creep if done further down the line.

After the initial phase of brainstorming, the team analyzes the ideas and discards the impossible ones, followed by the ones that are not “as good as they sounded at first”. The remaining ideas can come together to either form a concept of a videogame or some secondary component of it.

In short: brainstorming is a great activity for innovation, but since it’s essentially “throwing stuff at a wall and see what sticks” it can also be unproductive or even “excessively productive” and in both cases we end up with nothing in our hands.

5.2.4 On Sequels

In case your game becomes a hit, you will probably think about making a sequel: this is not inherently a bad thing, but you need to remember some things.

When developing a sequel, you will have to live up to your previous game, as well as the expectations of the players, and this becomes more and more difficult as the “successful sequels” go on.

Not only a sequel must be “as good or better” than its predecessor, but also it should add something to the original game, as well as the established lore (if there is one).

Your time and resource management must be top-notch to be able to “bring more with less”, since your resource need cannot skyrocket without a reason.

Also don’t get caught in the some kind of “sequel disease” where you end up making a sequel just to “milk the intellectual property”: you will end up ruining the whole series.

5.3 Common Errors and Pitfalls

When you are making a new game, it’s easy to feel lost and “out of your comfort zone”, and that’s okay! It’s also easy to fall into traps and pitfalls that can ruin your experience, here we take a look at the most common ones.

5.3.1 Losing motivation

Sometimes it can happen to lose motivation, usually due to having “too much ambition”: make sure you can develop the kind of game you want to make, and also leave multiplayer out of the question. It will just suck up development time, and it isn’t that much of an important feature anyway (and it can still be implemented later, see *Stardew Valley*).

Like in music, many people prefer “mediocrity” to “something great”, so don’t force yourself to innovate: do things well enough and if the innovative idea comes, welcome it.

If you get tired, take a break, you’re your own boss, and no one is behind you zapping you with a cattle prod: just focus on making a good overall product and things will go well.

5.3.2 The “Side Project” pitfall

It happens: you have a ton of ideas for games of all kinds, and probably you’ll start thinking:

■ what is bad about a small “side project”, to change things up a bit

You will end up having lots of “started projects” and nothing finished, your energy will deplete, things will become confusing and you won’t know what game you’re working on anymore.

Instead, make a small concept for the new mechanic and try to implement it in your current game, you may find a new mix that hasn’t been tried before, making your game that much more unique.

5.3.3 Making a game “in isolation”

While making a game you will need to gather some public for it, as well as create some hype around it: making a game on your own without involving the public is a mistake that deprives you of a huge source of suggestions and (constructive) criticism.

Make your game public, on platforms like itch.io or [IndieDB](https://indiedb.com), get feedback and encouragement. Create trailers towards the end of development, put them on [YouTube](https://www.youtube.com) or [Vimeo](https://vimeo.com) and if you want to go all out, get in touch with the press (locally first) and create hype around your product.

5.3.4 Mishandling Criticism

Among all the other things going on, we also need to handle feedback from our “potential players”, and this requires quite the mental effort, since we can’t make it “automatic”.

Not all criticism can be classified as “trolling”, and forging our game without listening to any feedback will only mean that such game won’t be liked by as many people as we would like, maybe for a very simple problem that could have been solved if only we listened to the public.

At the same time, not all criticism is “useful” either, not classifying criticism as “trolling” does not mean that trolling doesn’t exist, some people will take pride in ruining other people’s mood by being annoying and uselessly critic, or even finding issues that don’t exist.

The question you should ask yourself is simple:

Is this criticism I’m receiving constructive? Can it make my game better?

If the answer is no, then you may want to ignore such criticism, but if it is constructive, maybe you want to keep it in consideration.

5.3.5 Not letting others test your game

This is a common mistake when you are focused on making the game: using your own skill as a “universal measure” for the world’s skill level. You may be an unknown master at 2D platformers, and as such what can be “mildly difficult” for you may be “utterly impossible” for the average player. Or the opposite.

Try to keep the challenge constant through the levels, applying the usual slight upwards curve in difficulty that most games have, and let others test your game.

A beta version with feedback capabilities (or just a beta version and a form or email address can do the trick too) is pure gold when it comes to understanding what your players think about the game’s challenge level.

Remember: when a level is (perceived as unfairly) too hard, players will stop playing the game.

5.3.6 Being perfectionist

If you are called “perfectionist” by your friends, that should be a red flag in your game development process since the beginning.

Finding yourself honing the game over and over, allocating countless hours (that always feel as “not enough”) into making the game “better”, will end up just sabotaging the development process itself.

When you have:

- Good Visuals and Good Audio
- Working Gameplay
- A challenge that lasts the test of time
- The testing phase completed

You have a complete product. **Release it.** Updating it is very easy these days, and maybe that will give you the mental energy to undertake a new game. Maybe a sequel even?

5.3.7 Using the wrong engine

The game engine is one of the most important decisions you can take at the beginning of your game development journey. Realizing that you used the wrong engine after months of development can be a huge setback, as well as a “black hole” for your motivation.

Don’t trust market hype over an engine, and don’t trust the vendor’s promises either.

Does the game engine have the features you will need **already**? No? Then your money should stay where it is, and you should look somewhere else.

If such engine’s producer is promising the feature you want in future, don’t trust it, that version may come, or it may never come at all. If you bought the engine and such feature won’t ever be there, your money won’t come back.

5.4 Software Life Cycle Models

When talking about project management (in itself or in the broader field of Software Engineering) it is really useful to talk about some guideline models that can be used to manage your project.

5.4.1 Iteration versus Increment

Before getting to the models, we need to discuss the difference between two terms that are often used interchangeably: “iteration” and “increment”.

Iteration is a non-deterministic process, during an iteration you are revisiting what you have already done, and such revisiting can include an advancement or a regression. While iterating, you have no idea when you will finish your job.

Increment is deterministic instead, with increments you are proceeding by additions over a base. Every increment creates a “new base” for the next increments, and increments are numbered and limited, giving you an idea of when you have to finish your job.

5.4.2 Waterfall Model

The Waterfall model, also known as “sequential model” is the simplest one to understand, easily repeatable (in different projects) and is composed by phases that are **strictly sequential**, which means:

- There is no parallelism;
- There is no overlap between phases;
- When a phase is completed, you cannot go back to it.

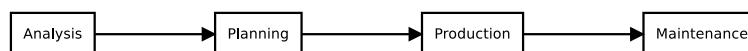


Figure 64: Diagram of the waterfall life cycle model

This makes the Waterfall life cycle model *extremely rigid*, everything needs to be carefully analyzed and documented (sometimes people define this model “document-driven”) and the coding is done only in its final

phases.

In order to have a good result, this model requires quantifying some metrics (time spent, costs, ...) and such quantification heavily relies on the experience of the project manager and the administrators.

5.4.3 Incremental Model

When a project of a certain size is involved, it's a bad idea to perform the so-called "big-bang integration" (integrating all the components together). Such approach would make troubleshooting a nightmare, so it's advisable to *incrementally integrate* the components.

The Incremental Model allows to have a "high-level analysis and planning", after that the team decides which features should be implemented first. This way the most important features are ready as soon as possible and have more time to become stable and integrate with the rest of the software.

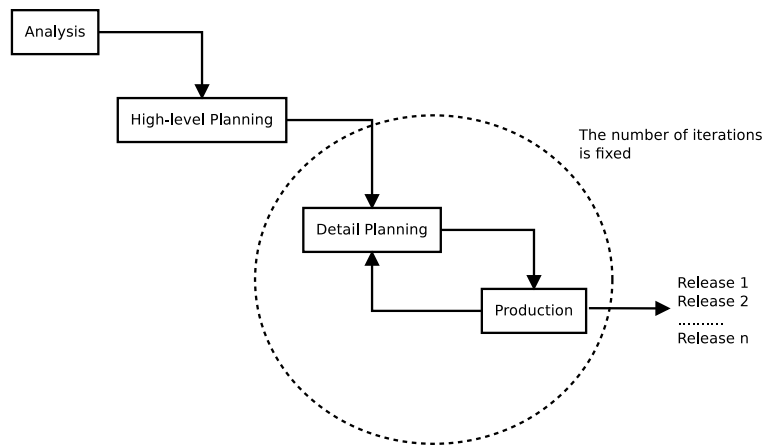


Figure 65: Diagram of the incremental life cycle model

This model can make use of strictly sequential phases (detail planning -> release -> detail planning -> release ...) or introduce some parallelism (for instance planning and developing frontend and backend at the same time).

As seen from the diagram, the high-level analysis and planning are not repeated, instead the detail planning and release cycle for a well-defined number of iterations, and on each iteration we will have a working release or prototype.

5.4.4 Evolutionary Model

It's not always possible to perfectly know the outline of a problem in advance, that's why the evolutionary model was invented. Since needs tend to change with time, it's a good idea to maintain life cycles on different versions of your software at the same time.

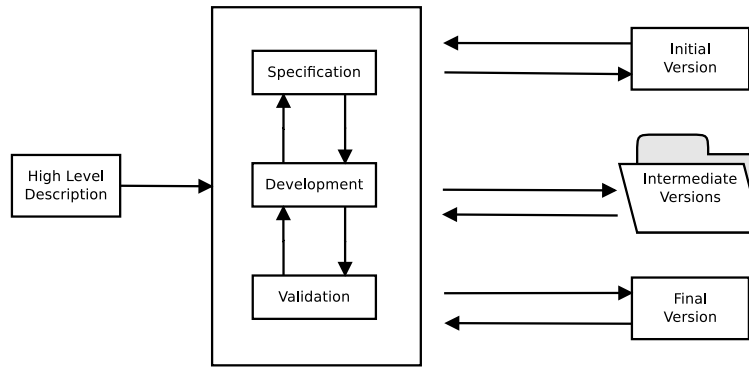


Figure 66: High-level diagram of the evolutionary life cycle model

Adding a way to implement the feedback you get from your customers and stakeholders completes the micro-managed part of the life cycle model, each time feedback and updates are implemented, a new version is released.

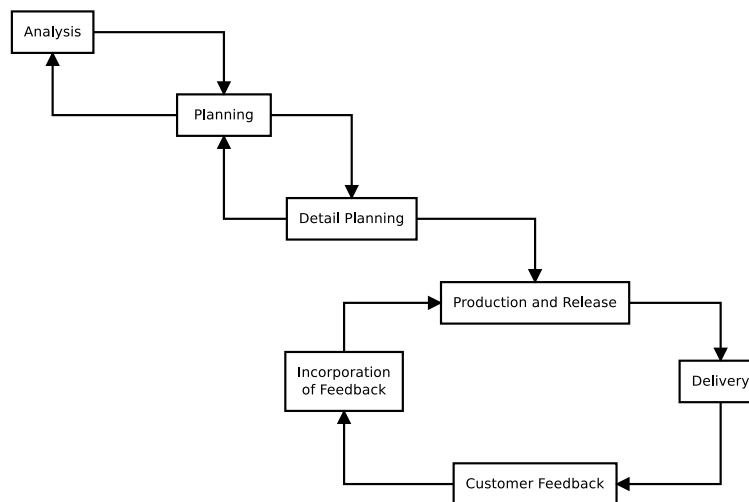


Figure 67: Diagram of the evolutionary life cycle model

5.4.5 Agile Software Development

Agile Software Development was born as a reaction to the excessive rigidity of the models we've seen so far. The basic principles of Agile Software Development are presented at the <http://agilemanifesto.org> website, but we will shortly discuss them below.

- Rigid rules are not good;
- A working software is more important than a comprehensive documentation;
- Seek collaboration with the stakeholder instead of trying to negotiate with them;
- Responding to change is better than following a plan
- Interactions and individuals are more important than processes and tools.

Obviously not everything that shines is actually gold, there are many detractors of the Agile model, bringing on the table some criticism that should be noted:

- The agile way of working entails a really high degree of discipline from the team: the line between “flexibility” and “complete lack of rules” is a thin one;

- Software without documentation is a liability more than an asset: commenting code is not enough - you need to know (and let others know) the reason behind a certain choice;
- Without a plan, you can't estimate risks and measure how the project is coming along;
- Responding to change can be good, but you need to be aware of costs and benefits such change and your response entail.

5.4.5.1 User Stories

Agile models are based on “User Stories”, which are documents that describe the problem at hand.

Such documents are written by talking with the stakeholder/customer, listening to them, actively participating in the discussion with them, proposing solutions and improvements actively.

A User Story also defines how we want to check that the software we are producing actually satisfies our customer.

5.4.5.2 Scrum

The term “scrum” is taken from the sport of American Football, where you have an action that is seemingly product of chaos but that instead hides a strategy, rules and organization.

Let's see some Scrum terminology:

- **Product Backlog:** This is essentially a “todo list” that keeps requirements and features our product must have;
- **Sprint:** Iteration, where we choose what to do to create a so-called “useful increment” to our product. Each Sprint lasts around 2 to 4 weeks and at the end of each sprint you obtain a version of your software that can be potentially sold to the consumer;
- **Sprint Backlog:** Essentially another “todo list” that keeps the set of user stories that will be used for the next sprint.

As seen from the terminology, the Scrum method is based on well-defined iterations (Sprints) and each sprint is composed by the following phases:

- **Sprint Planning:** You gather the product backlog and eventually the previous sprint backlogs and decide what to implement in the upcoming sprint;
- **Daily Scrum:** A daily stand-up meeting that lasts around 15 minutes where a check on the daily progress is done;
- **Sprint Review:** After the sprint is completed, we have the verification and validation of the products of the sprint (both software and documents);
- **Sprint Retrospective:** A quality control on the sprint itself is done, allowing for continuous improvement over the way of working.

5.4.5.2.1 Criticisms to the Scrum approach

The Scrum approach can quickly become chaotic if User Stories and Backlogs are not well kept and clear. Also, no matter how short it can be, the Daily Scrum is still an invasive practice that interrupts the workflow and requires everyone to be present and ready.

5.4.5.3 Kanban

Kanban is an Agile Development approach taken by the scheduling system used for lean and just-in-time manufacturing implemented at Toyota.

The base of Kanban is the “Kanban Board” (sometimes shortened as “Kanboard”), where plates (also called “cards” or “tickets”) are moved through swimlanes that can represent:

- The status of the card (To Do, Doing, Testing, Done)
- The Kind of Work (Frontend, Backend, Database, ...)
- The team that is taking care of the work

The board helps with organization and gives a high-level view of the work status.

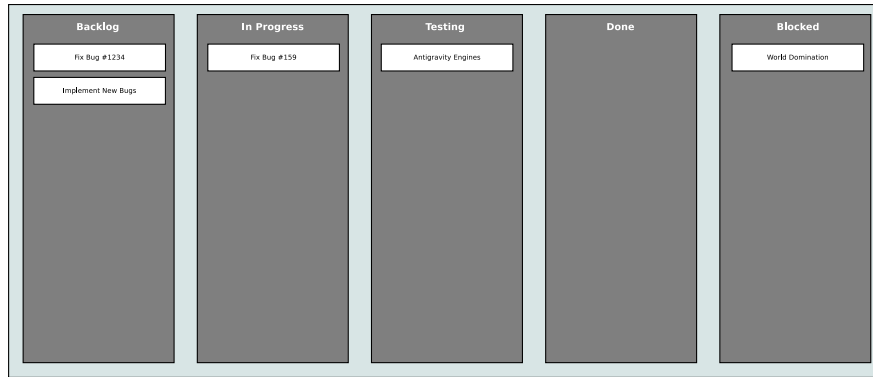


Figure 68: Example of a Kanban Board

5.4.5.4 ScrumBan

ScrumBan is a hybrid approach between Scrum and Kanban, mixing the Daily Scrum and Sprint Approach with the Kanban Board.

This approach is usually used during migration from a Scrum-Based approach to a purely Kanban-based approach.

5.4.6 Where to go from here

Obviously the models presented are not set in stone, but are “best practices” that have been proven to help with project management, and not even all of them.

Nothing stops you from taking elements of a model and implement them into another model. For example you could use an Evolutionary Model with a Kanban board used to manage the single increment.

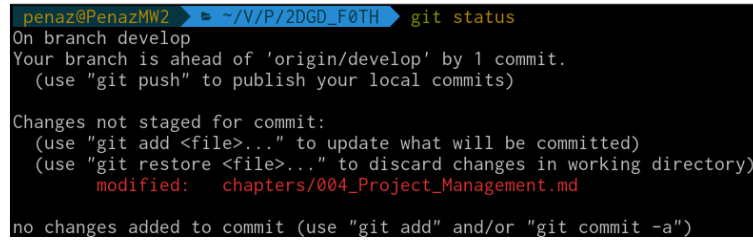
5.5 Version Control

When it comes to managing any resource that is important to the development process of a software, it is vitally important that a version control system is put in place to manage such resources.

Code is not the only thing that we may want to keep under versioning, but also documentation can be subject to it.

Version Control Systems (VCS) allow you to keep track of edits in your code and documents, know (and blame) users for certain changes and eventually revert such changes when necessary. They also help saving on bandwidth by uploading only the differences between commits and make your development environment more robust (for instance, by decentralizing the code repositories).

The most used Version Control system used in coding is Git, it’s decentralized and works extremely well for tracking text-based files, like code or documentation, but thanks to the LFS extension it is possible for it to handle large files efficiently.



```
penaz@PenazMW2 ~/V/P/2DGD_F0TH$ git status
On branch develop
Your branch is ahead of 'origin/develop' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   chapters/004_Project_Management.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Figure 69: An example screen from Git, a version control system

Other used version control systems are Mercurial and SVN (subversion).

Another useful feature of many version control systems are remote sources, which allow you to upload and synchronize your repositories with a remote location (like GitHub, GitLab or BitBucket for instance) and have it safe on the cloud, where safety by redundancy is most surely ensured.

5.6 Metrics and dashboards

During development you need to keep an eye on the quality of your project, that's when you need a **project dashboard**: but before that, you need to decide what your **quality metrics** are, that means the measurements that define if your project is “up to par” with what you expect or not.

5.6.1 Cyclomatic Complexity

[This section is a work in progress and it will be completed as soon as possible]

5.6.2 SLOC

[This section is a work in progress and it will be completed as soon as possible]

5.6.3 Code Coverage

[This section is a work in progress and it will be completed as soon as possible]

5.6.4 Code Smells

[This section is a work in progress and it will be completed as soon as possible]

5.6.5 Coding Style breaches

[This section is a work in progress and it will be completed as soon as possible]

5.7 Continuous Integration and Delivery

[This section is a work in progress and it will be completed as soon as possible]

6 Introduction to Game Design

Why should you make games? Do it to give players joy from your unique perspective and to have fun expressing yourself. You win and the players win.

Duane Alan Hahn

In this section we will talk about platforms, input systems and game genres, in a quick fashion. This chapter will introduce you to the language and terms used in game design, this way the following chapters will be easier to comprehend.

We will talk about the differences and challenges deriving from each decision and the basic way game genres work.

6.1 Platforms

There are several different platforms a game can be developed for, and each one has its own advantages and drawbacks. Here we will discuss the most notable ones.

6.1.1 Arcade

Arcade cabinets have been around for decades, and have still a huge part in the heart of gaming aficionados with classic series going on like “Metal Slug”. The main objective of these machines is to make you have fun, while forcing you to put quarters in to continue your game.

These cabinets’ software is known to be very challenging, having some nice graphics and sound. Arcade games are usually presented in the form of an “arcade board”, which is the equivalent of a fully-fledged console, with its own processing chips and read-only memory.

In the case of arcades, the hardware is usually tailored to support the software; with some exceptions added later (like the Capcom Play System, also known as CPS), where the hardware is more stable between arcades, while the software changes.

6.1.2 Console

Consoles are a huge (if not the biggest) part in the video game industry. Their Hardware is dedicated solely to gaming (and some very marginal “multimedia functionalities”) and it evolves in “generations”: this means that each “generation” has a stable hardware programmers can study and exploit.

This hardware stability is a double-edged sword: the hardware can be really hard to master at the beginning, resulting in some poor-performing games at the beginning of the generation, but when mastered the results are incredible. This feeds into a cycle that looks like the following:

1. New Generation is introduced
2. Initial confusion, with poor performance and graphics
3. Hardware is mastered and games have great performance/graphics
4. The games become “too big” for the current generation and a new generation must be introduced.

6.1.3 Personal Computer

Personal Computers are another huge part of the video game industry. They are extremely flexible (being general-purpose machines) but have a huge drawback: their hardware is not the same from one unit to the other. This means that the programmer needs to use “abstraction layers” to be able to communicate with all the different hardware.

This can have performance costs, as well as forcing the programmer to add options to lower graphic settings, resolution and more.

All of this just to be able to run on as many computers as possible. The upside is that when the computer is really powerful, you can get great performance and amazing quality, but that's a rare occasion.

6.1.4 Mobile

One of the most recent platforms game developers work on is right in your pocket: your smartphone.

Today's smartphones have enough power to run fully-fledged videogames, on the go. Sadly the touch screen can prove to be really uncomfortable to use, unless the game is specially tailored for it.

6.1.5 Web

Another platform that has seen a massive rise in recent times is the Web: with WebGL and WebAssembly, fully-fledged games (including 3D games) can run on our browser, allowing for massively-multiplayer experiences (like Agar.io) without the hassle of manual installation or making sure the game is compatible with your platform.

A drawback of the “web approach” is the limited performance that web browsers, WebGL and WebAssembly can give, as well as the need to download the game before being able to play (and sometimes you may need to re-download the game if you cleared your browser's cache).

6.2 Input Devices

A game needs a way to be interacted with: this “way” is given by input devices. In this section we will take a brief look at the input devices available in a game.

6.2.1 Mouse and Keyboard

One of the most common input devices, most of the currently available frameworks and engine have support for input via mouse and keyboard. These input methods are great for visual novels, point and click adventures, FPS/TPS games and anything that is considered to be “made for PCs”.

6.2.2 Gamepad

One of the classics of input devices, works well with the majority of games: FPS/TPS games may need some aim assist mechanic in your game. Point and click adventures feel clunky with this input method.

As with Mouse and Keyboard, most of the currently available engines and frameworks support gamepads.

6.2.3 Touch Screen

With the coming of smartphones, touch screen is a new input device that we have to account for. Touch screens emulate computer mice well enough, although they lack precision.

The nature of being a mix between an input device and a screen brings a lot of new ways to experience a game if well done. Many times touch screens are used to simulate game pads: the lack of the tactile feedback given by buttons makes this simulation clunky and uncomfortable.

Some of the most recent framework and engines support touch screens, although there's an additional layer of complexity given by the specific operating system of the smartphone you're building for.

6.2.4 Dedicated Hardware

Some games require dedicated hardware to work at their best, if at all. Guitars (guitar hero), wheels for racing games, joysticks for flying simulators, arcade sticks for arcade ports. . .

Dedicated hardware requires precise programming, and is usually an advanced topic. On PCs many “dedicated input devices” are recognized as “game pads” and use an “axis” and “buttons” abstraction that makes coding easier.

6.2.5 Other Input Devices

A special mention is deserved for all the input devices that are “general purpose” (as in not “dedicated”) but are still in a group outside what we saw so far.

In this group we see gyroscopes, accelerometers (like the Nintendo Wii/Switch JoyCons), sensors, IR as well as other exotic hardware that can still be exploited in a videogame.

6.3 Game Genres

Let’s analyze some game genres to understand them better and introduce some technical language that may be useful in **writing a Game Design Document**.

These genres are quite broad, so a videogame is usually a mix of these “classes” (like a strategy+simulation game).

6.3.1 Shooters

Shooters are games that involve. . . shooting. They can include any kind of projectile (bullets, magic from a fairy, arrows from a hunter) and can be crossed with any other genre (creating sub-genres in a way), like 2D platformers.

Some of the most known shooter genres are:

- **FPS** (first person shooters), 3D games where the game is shown from the point of view of the protagonist. This involves only seeing a HUD and the weapon, instead of the whole character;
- **TPS** (third person shooters), 3D games where the game is shown from a behind-the-character perspective. Some show the whole protagonist, while others adopt an over-the-shoulder perspective;
- **Top Down Shooters**, usually 2D games where you may be piloting a vehicle (space ship, plane, etc. . .) and shoot down waves of enemies, in this category we fit arena shooters (like *Crimsonland*) and space shooters (like *Galaga*);
- **Side scroller shooters**, usually 2D games and platformers, where you control the protagonist and shoot enemies on a 2D plane, in this category we find games like *Metal Slug*.

6.3.2 Strategy

Strategy games involve long-term planning and resource control, they are slower games, but can be really intense when played in competition with other players.

Some of the most popular strategy genres are:

- **RTS** (real time strategy), where units are controlled in real time;
- **Turn-based strategy**, where units and resources are managed in turns;

6.3.3 Platformer

Platformer games involve difficult jumps and precise movement, they can both be 2D and 3D games. A prime example of platformer games is the Mario series: Mario 1,2,3 for 2D games and Mario 64 for 3D.

6.3.4 RPG

RPGs or “Role Playing Games” are games where you assume the role of a character in a fictional setting. In RPGs the world is well-defined and usually have some level or class system and quite advanced item management.

RPGs can be either action/adventure, with real-time actions, turn-based or hybrid, where the movement is done in real time but battles happen in turns. Some prime examples of RPG games are the Legend of Zelda series, as well as the Final Fantasy series.

6.3.5 MMO

MMO (Massively Multiplayer Online) is a term used for games that have a heavy multiplayer component via the internet. The most known MMO genre is MMORPGs (Massively Multiplayer Online Role-Playing Games).

6.3.6 Simulation

Simulation games cover a huge variety of games that are create to “simulate reality”, in more or less precise ways. Among simulation games we can find:

- **Racing Games:** sometimes more simulative others more arcade-like, racing games simulate the experience of driving a vehicle, more or less realistic (from modern cars to futuristic nitro-fueled bikes);
- **Social Simulation:** simulating the interaction between characters, a pioneer on the genre is surely “The Sims”;
- **Farming simulation:** simulating the quietude and work in the fields;
- **Business simulation:** like “game dev tycoon” or “rollercoaster tycoon”;

But there are also other kinds of simulations, like Sim City, where you manage an entire city.

6.3.7 Rhythm Games

Rhythm games are based on the concept of following a music beat as precisely as possible, this can be also used as a “mechanic” in other types of games.

Some examples of Rhythm games are “Dance-Dance Revolution” (also known as DDR), as well as more innovative games like “Crypt of the Necrodancer” (a mix between rhythm game and dungeon crawler).

6.3.8 Visual novels

Visual novels are graphical adventures whose primary objective is “telling a story”, they can be linear or have a “choose your own path” component. They usually feature multiple endings and hand-crafted still images as artwork.

7 Writing a Game Design Document

If you don't know where you are going. How can you expect to get there?

Basil S. Walsh

One of the most discussed things in the world of Game Development is the so-called “GDD” or “Game Design Document”. Some say it's a thing of the past, others swear by it, others are not really swayed by its existence. Being an important piece of any software development process, in this book we will talk about the GDD in a more flexible way.

7.1 What is a Game Design Document

The Game Design Document is a Body Of Knowledge that contains everything that is your game, and it can take many forms, such as:

- A formal design document;
- A Wiki^g;
- A Kanboard^g.

The most important thing about the GDD is that it contains all the details about your game in a centralized and possibly easy-to-access place.

It is not a technical document, but mostly a design document, technical matters should be moved to a dedicated “Technical Design Document”.

7.2 Possible sections of a Game Design Document

Each game can have its own attributes, so each Game Design Document can be different, here we will present some of the most common sections you can include in your own Game Design Document.

7.2.1 Project Description

This section is used to give the reader a quick description of the game, its genre (RPG, FPS, Puzzle,...), the type of demographic it covers (casual, hardcore, ...). Additional information that is believed to be important to have a basic understanding of the game can be put here.

This section should not be longer than a couple paragraphs.

A possible excerpt of a description could be the following:

This game design document describes the details for a 2D side scrolling platformer game where the player makes use of mechanics based on using arrows as platforms to get to the end of the level.

The game will feature a story based on the central America ancient culture (Mayan, Aztec, ...).

The name is not defined yet but the candidate names are:

7.2.2 Characters

If your game involves a story, you need to introduce your characters first, so that everything that follows will be clear.

A possible excerpt of a characters list can be the following:

Ohm is the main character, part of the group called “The Resistance” and fights for restoring the electrical order in the circuit world.

Fad is the main side character, last survivor and heir of the whole knowledge of “The Capacitance” group. Its main job is giving technical assistance to Ohm.

Gen. E. Rator is the main antagonist, general of “The Reactance” movement, which wants to conquer the circuit world.

This can be a nice place where to put some character artwork.

If your game does not include a story, you can just avoid inserting this section altogether.

7.2.3 Storyline

After introducing the characters, it’s time to talk about the events that will happen in the game.

An example of story excerpt can be the one below:

It has been 500 mega-ticks that the evil **Rator** and the reactance has come to power, bringing a new era of darkness into the circuit world.

After countless antics by the evil reactance members, part of the circuit world’s population united into what is called “The Resistance”.

Strong of thousands of members and the collaboration of *the Capacitance*, the resistance launched an attack against the evil reactance empire, but the empire stroke back with a carpet surcharge attack, decimating the resistance and leaving only few survivors that will be tasked to rebuild the resistance and free the world from the reactance’s evil influence.

This is when a small child, and their parents were found. The child’s name, **Ohm**, sounded prophetic of a better future of the resistance.

And this is where our story begins.

As with the Characters section, if your game does not include a story, you can just skip this section.

7.2.3.1 The theme

When people read the design document, it is fundamental that the game’s theme is quickly understood: it can be a comedy-based story, or a game about hardships and fighting for a better future, or maybe it is a purely fantastic game based on ancient history. . .

Here is a quick example:

This is a game about fighting for a better future, dealing with hardships and the deep sadness you face when you are living in a world on the brink of ruin.

This game should still underline the happiness of small victories, and give a sense of “coziness” in such small things, even though the world can feel cold.

If you feel that this section is not relevant for your game, you can skip it.

7.2.3.2 Progression

After defining the story, you should take care of describing how the story progresses as the player furthers their experience in a high-level fashion.

An example:

The game starts with an intro where the ruined city is shown to the player and the protagonist receives their magic staff that will accompany them through the game.

The first levels are a basic tutorial on movement, where the shaman teaches the player the basic movement patterns as well as the first mechanic: *staff boosting*. Combat mechanics are taught as well.

After the tutorial has been completed, the player advances to the first real game area: **The stone jungle**.

...

7.2.4 Levels and Environments

In this section we will define how levels are constructed and what mechanics they will entail, in detail.

We can see a possible example here:

The First Level (Tutorial) is based in a medieval-like (but adapted to the center-America theme) training camp, outside, where the player needs to learn jumping, movement and fight straw puppets. At the end of the basic fighting and movement training, the player is introduced to *staff boosting* which is used to first jump to a ledge that is too high for a normal jump, and then the mechanic is used to boost towards an area too far forward to reach without boosting.

...

Some level artwork can be included in this section, to further define how the levels will look and feel.

7.2.5 Gameplay

This section will be used to describe your gameplay. This section can become really long, but do not fear, as you can split it in meaningful sections to help with organization and searching.

7.2.5.1 Goals

Why is the player playing your game?

This question should be answered in this section. Here you insert the goals of your game, both long and short term.

An example could be the following:

Long Term Goal: Stop the great circuit world war

Optional Long Term Goal: Restore the circuit world to its former glory.

Short Term Goals:

- Find the key to the exit
- Neutralize Enemies
- Get to the next level

7.2.5.2 Game Mechanics

In this section, you describe the core game mechanics that characterize the game, extensively. There are countless resources on how to describe game mechanics, but we'll try to add an example here below.

The game will play in the style of the well-known match-3 games. Each match of 3 items will add some points to the score, and new items will “fall” from a randomly chosen direction every time.

Every time an “L” or a “T” match is performed, a special item of a random colour will be generated, when a match including this item is made, all the items in the same row and column will be deleted and bonuses will be awarded.

Every time a match with 4 items in a row is performed, a special item of a random colour will be generated, when a match including such item is made, all items in a 3x3 grid centered on the item will be deleted and bonuses will be awarded.

Every time a match with 5 items in a row is performed, a special uncolored item will be generated, this can be used as a “wildcard” for any kind of match.

In case the 5-item special is matched with any other special item, the whole game board will be wiped and a bonus will be awarded.

...

7.2.5.3 Skills

Here you will describe the skills that are needed by the users in order to be able to play (and master) your game.

This will be useful to assess your game design and eventually find if there are some requirements that are too high for your target audience; for instance asking a small child to do advanced resource management could be a problem.

This will also help deciding what the best hardware to use your game on could be, for instance if your game requires precise inputs for platforming then touch screens may not be the best option.

Here’s an example of such section:

The user will need the following skills to be able to play the game effectively:

- Pressing Keyboard Buttons or Joypad Buttons
- Puzzle Solving (for the “good ending” overarching puzzle)
- Timing inputs well (for the sections with many obstacles)

...

7.2.5.4 Items/Powerups

After describing the basic game mechanics and the skills the user needs to master to be able to play the game effectively, you can use this section to describe the items and powerups that can be used to alter the core gameplay.

For example:

The player can touch a globular light powerup to gain invincibility, every enemy that will touch the player will get automatically killed. The powerup duration is 15 seconds.

Red (incendiary) arrows can be collected through the levels, they can get shot and as soon as they touch the ground or an enemy, they burst into flames, similarly to a match.

...

In this section you describe all items that can be either found or bought from an in-game store or also items derived from micro-transactions. In-game currency acquisition should be mentioned here too, but further detailed in the monetization section.

7.2.5.5 Difficulty Management and Progression

This section can be used to manage how the game gets harder and how the player can react to it. This will expand on game mechanics like leveling and gear.

This section is by its own nature quite subjective, but describing how the game progresses helps a lot during the tighter parts of development.

Below a possible example of this section:

The game will become harder by presenting tougher enemies, with more armor, Health Points and attack. To overcome this difficulty shift, the player will have to create defense strategy and improve their dodging, as well as leveling up their statistics and buy better gear from the towns' shops.

In the later levels, enemies will start dodging too, and will also be faster. The player will need to improve their own speed statistic to avoid being left behind or “kited” by fast enemies.

As the game progresses, the player will need to acquire heavy weapons to deal with bigger bosses, as well as some more efficient ranged weapons to counteract ranged enemies.

...

This section is good if you want to talk about unlocking new missions/maps/levels too.

7.2.5.6 Losing Conditions

Many times we focus so much on how the player will get to the end of the game that we absolutely forget how the player can *not* get to the end of the game.

Losing conditions must be listed and have the same importance of the winning conditions, since they add to the challenge of the game itself.

A possible example of how a “losing conditions” section could be written is the following:

The game can be lost in the following ways:

- Losing all the lives and not “continuing” (Game Over)
- Not finding all the Crystal Oscillators (Bad Ending)

A possible variation on the theme could be having an “endings” section, where all (both good, bad and neutral) endings are listed.

7.2.6 Graphic Style and Art

Here we describe the ideas on how the game will look like. Describing the graphic style and medium.

Here is a possible example of the game:

This is a 2D side scroller with a dark theme, the graphics should look gloomy and very reminiscing of a circuit board.

The graphical medium should be medium-resolution pixel art, allowing the player's imagination to “fill in” the graphics and allowing to maintain a “classic” and “arcade” feeling.

...

7.2.7 Sound and Music

Sadly, in way too many games, music and sound is an afterthought. A good soundtrack and sound effect can really improve the immersion, even in the simplest of games.

In this section we can describe in detail everything about Music and Sound Effects, and if the section becomes hard to manage, splitting it in different sub-sections could help organization.

Music should be based on the glitch-hop style, to complement the electronic theme. 8 or 16-bit style sounds inside the score are preferable to modern high-quality samples.

Sound effects should appeal to the 8 or 16-bit era.

Lots of sound effects should be used to give the user positive feedback when using a lever to open a new part of the level, and Extra Lives/1UP should have a jingle that overrides the main music.

7.2.8 User Interface

In this section we will describe everything that concerns the User Interface: menus, HUD, inventories and everything that will contribute to build the user experience that is not strictly tied to the gameplay.

This is especially important in games that make heavy use of menus, like turn-based strategy games or survival games where inventory management can be fundamental.

Let's see an example of how this section can be written:

The game will feature a cyberpunk-style main menu, looking a lot like an old green-phosphor terminal but with a touch of futurism involved. The game logo should be visible on the left side, after a careful conversion into pixel-art. On the right, we see a list of buttons that remind old terminal-based GUIs. On the bottom of the screen, there should be an animated terminal input, for added effect.

Every time a menu item is highlighted or hovered by the mouse, the terminal input will animate and write a command that will tie to the selected menu voice, such as:

- Continue Game: `./initiate_mission.bin -r`
- Start Game: `./initiate_mission.bin --new`
- Options: `rlkernel_comm.bin --show_settings`
- Exit: `systemcontrol.bin --shutdown`

The HUD display should remind a terminal, but in a more portable fashion, to better go with the “portability” of a wrist-based device.

It's a good idea to add some mock designs of the menu in this section too.

7.2.9 Game Controls

In this section you insert everything that concerns the way the game controls, eventually including special peripherals that may be used.

This will help you focusing on better implementing the input system and limit your choices to what is feasible and useful for your project, instead of just going by instinct.

Below, a possible way to write such section

The game will control mainly via mouse and keyboard, using the mouse to aim the weapon and shoot and keyboard for moving the character.

Alternatively, it's possible to connect a twin-stick gamepad, where the right stick moves the weapon crosshair, while the left stick is used to move the character, one of the back triggers of the gamepad can be configured to shoot.

If the gamepad is used, there will be a form of aim assistance can be enabled to make the game more accessible to gamepad users.

7.2.10 Accessibility Options

Here you can add all the options that are used to allow more people to access your game, in more ways than you think.

Below, we can see an example of many accessibility options in a possible game.

The game will include a “colorblind mode”, allowing the colours to be colorblind-friendly: such mode will include 3 options: Deuteranopia, Tritanopia and Monochromacy.

Additionally, the game will include an option to disable flashing lights, making the game a bit more friendly for people with photosensitivity.

The game will support “aim assistance”, making the crosshair snap onto the enemy found within a certain distance from the crosshair.

In order to assist people who have issues with the tough platforming and reaction times involved, we will include the possibility to play the game at 75%, 50% and 25% speed.

7.2.11 Tools

This section is very useful for team coordination, as having the same toolkit prevents most of the “works for me” situations, where the game works well for a tester/developer while it either crashes or doesn’t work correctly for others.

This section is very useful in case we want to include new people in our team and quickly integrate them into the project.

In this section we should describe our toolkit, possibly with version numbers included (which help reducing incompatibilities), as well as libraries and frameworks. The section should follow the trace below:

The tools and frameworks used to develop the game are the following:

Pixel Art Drawing: Aseprite 1.2.13

IDE: Eclipse 2019-09

Music Composition: Linux MultiMedia Studio (LMMS) 1.2.1

Map and level design: Tiled 1.3.1

Framework: SFML 2.5.1

Version Control: Git 2.24.0 and GitLab

7.2.12 Marketing

This section allows you to decide how to market the game and have a better long-term plan on how to market your game to your players.

Carefully selecting and writing down your target platforms and audience allows you to avoid going off topic when it comes to your game.

7.2.12.1 Target Audience

Knowing who is your target audience helps you better suit the game towards the audience that you are actually targeting.

Here is an example of this section:

The target audience is the following:

Age: 15 years and older

Gender: Everyone

Target players: Hardcore 2D platformer fans

7.2.12.2 Available Platforms

Here you describe the launch platforms, as well as the platforms that will come into the picture after the game launched. This will help long term organization.

Here is an example of how this section could look:

Initially the game will be released on the following platforms:

- PC
- Playstation 4

After launch, we will work on the following ports:

- Nintendo Switch
- XBox 360

After working on all the ports, we may consider porting the game to mobile platforms like:

- Android 9.0 +
- iOS 11.0 +

...

7.2.12.3 Monetization

In this optional section you can define your plans for the ways you will approach releasing the game as well as additional monetization strategies for your game.

For example:

The game will not feature in-game purchases.

Monetization efforts will be focused on selling the game itself at a full “indie price” and further monetization will take place via substantial Downloadable Content Expansions (DLC)

The eventual mobile versions will be given away for free, with advertisements integrated between levels. It is possible for the user to buy a low-price paid version to avoid seeing the advertisements.

7.2.12.4 Internationalization and Localization

Internationalization and Localization are a matter that can make or break your game, when it comes to marketing your game in foreign countries.

Due to political and cultural reasons, for instance you shouldn't use flags to identify languages. People from territories inside a certain country may not be well accepting of seeing their language represented by the flag of their political adversaries.

Another example could be the following: if your main character is represented by a cup of coffee, your game could be banned somewhere as a “drug advertisement”.

This brings home the difference between “Internationalization” and “Localization”:

Internationalization Making something accessible across different countries without major changes to its content

Localization Making something accessible across different countries, considering the target country's culture.

We can see a possible example of this section below:

The game will initially be distributed in the following languages:

- English
- Italian

After the first release, there will be an update to include:

- Spanish
- German

- • French

7.2.13 Other/Random Ideas

This is another optional section where you can use as a “idea bin”, where you can put everything that you’re not sure will ever make its way in the game. This will help keeping your ideas on paper, so you won’t ever forget them.

We can see a small example here:

Some random ideas:

- User-made levels
- Achievements
- Multiplayer Cooperative Mode
- Multiplayer Competitive Mode

7.3 Where to go from here

This chapter represents only a guideline on what a Game Design Document can be, feel free to remove any sections that don’t apply to your current project as well as adding new ones that are pertinent to it.

A Game Design Document is a Body of Knowledge that will accompany you throughout the whole game development process and it will be the most helpful if you are comfortable with it and it is shaped to serve you.

8 The Game Loop

All loops are infinite ones for faulty RAM modules.

Anonymous

8.1 The Input-Update-Draw Abstraction

As animations and movies are an illusion, so are games. Games and movies show still images tens of times per second, giving us the illusion of movement.

Any game and its menus can be abstracted into 3 main operations that are performed one after the other, in a loop:

- 1) Process the user input
- 2) Update the world (or menu) status
- 3) Display (Draw) the updated world (or again, menu) to the screen

So a pseudocode implementation of such loop would be something like the following:

Listing 11: Game Loop example

```
function game():  
2   game_is_running=True  
   while game_is_running:  
4       process_user_input()  
       update_world()  
6       draw()
```

This abstraction will become really useful when dealing with many rows of code and keeping it neatly organized.

8.2 Input

8.2.1 Events vs Real Time Input

Some frameworks may be able to further abstract how they process input by giving an *API_g* that allows to make use of **events**.

Most of the time, events will be put in a queue that will be processed separately. This way it's easier to program how to react to each event and keep our code neatly organized. The downside is that the performance of an event-driven input processing is directly tied to how many events are triggered: the more events are triggered, the longer the wait may be before we get to our processed input. Also in case of conflicting inputs (for instance pressing left and right on the keyboard), the one that gets processed later in the queue might take over.

This usually depends on the implementation of the event queue: usually an event queue is less wasteful in terms of resources and allows for less coupled code, but the queue could be cluttered with events we're not interested in (for instance mouse movement events in a game that uses only keyboard for controls) so we need to take the time to configure our event handler to ignore certain events when not necessary.

On the opposite side, we have so-called “real-time input”, where at a certain point of our update routine, we check for the instantaneous status of the input peripherals and process it immediately. This allows for a faster, more reactive code and to apply some different logic (for instance pressing left and right on the keyboard can be coded to make the character stop). Besides being more immediate, this system shares a lot of traits with “polling” which can be performance-heavy, as well as inducing some undesired code coupling.

A well-implemented and well-configured event-based system should feel no different from real-time input, with the advantage of having better performance and having less code coupling.

8.3 Timing your loop

When it comes to anything that remotely relates to physics (that includes videogames), we need to set the relation to time in our loop. There are many ways to set our delta time (or time steps), we'll see some of the most common.

8.3.1 What is a time step

A time step (or delta time) is a number that will define “how much time passed” between two “snapshots” of our world (remember, the world is updating and showing in discrete intervals, giving the illusion of movement). This number will allow us to make our loop more flexible and react better to the changes of load and machines.

8.3.2 Fixed Time Steps

The first and simplest way is to use a fixed time step, our delta time is fixed to a certain number, which makes the simulation easier to calculate but also makes some heavy assumptions:

- Vertical Synchronization is active in the game
- The PC is powerful enough to make our game work well, 100% of the time

An example of fixed time step loop can be the following (assuming 60 frames per second or $dt = \frac{1}{60}$):

Listing 12: Game loop with fixed timesteps

```
dt = 1.0/60.0
2 game_is_running = True

4 while game_is_running:
    process_user_input()
6     update_world(dt)
    draw()
```

Everything is great, until our computer starts slowing down (high load or just not enough horsepower), in that case the game will slow down.

This means that every time the computer slows down, even for a microsecond, the game will slow down too, which can be annoying.

8.3.3 Variable Time Steps

A way to limit the issues given by a fixed time step approach is to make use of variable time steps, which are simple in theory, but can prove hard to manage.

The secret is measuring how much time passed between the last frame and the current frame, and use that value to update our world.

An example in pseudocode could be the following:

Listing 13: Game loop with variable time steps

```
game_is_running = True
2
while game_is_running:
4     dt = measure_time_from_last_frame()
    process_user_input()
6     update_world(dt)
```

```
draw()
```

This allows to smooth the possible lag spikes, even allowing us to disable Vertical Sync and have a bit less input lag, but this approach has some drawbacks too.

Since the delta time now depends on the speed of the game, the game can “catch up” in case of slowdowns; that can result in a slightly different feeling, depending on the framerate, but if there is a really bad slowdown Δt can become really big and break our simulation, and collision detection will probably be the first victim.

Also this method can be a bit harder to manage, since every movement will have to be scaled with Δt .

8.3.4 Semi-fixed Time Steps

This is a special case, where we set an upper limit for our time steps and let the update loop execute as fast as possible. This way we can still simulate the world in a somewhat reliable way, avoiding the dangers of higher spikes.

A semi-fixed time step approach is the following (assuming 60 fps or $dt = \frac{1}{60}$):

Listing 14: Game loop with Semi-Fixed time steps

```
dt = 1.0/60.0
2 game_is_running = True

4 while game_is_running:
    frametime = measure_time_from_last_frame()

    6     while frametime > 0.0:
        deltaTime = min(dt, frametime)
        process_user_input()
        10     update_world(dt)
        frametime = frametime - deltaTime
    12     draw()
```

This way, if the loop is running too slow, the game will slow down and the simulation won’t blow up. The main disadvantage of this approach is that we’re taking more update steps for each draw step, which is fine if drawing takes more than updating the world. If instead the update phase of the loop takes more than drawing it, we will spiral into a terrible situation.

We can call it a “spiral of death”, where the simulation will take Y seconds (real time) to simulate X seconds (of game time), with $Y > X$, being behind in your simulation makes the simulation take more steps, which will make the simulation fall behind even more, thus making the simulation lag behind more and more.

8.3.5 Frame Limiting

Frame limiting is a technique where we aim for a certain duration of our game loop. If an iteration of the game loop is faster than intended, such iteration will wait until we get to our target loop duration.

Let’s again consider a loop running at 60fps (or $dt = \frac{1}{60}$):

Listing 15: Game loop with Frame Limiting

```
targetTime = 1.0/60.0
2 game_is_running = True

4 while game_is_running:
    dt = measure_time_from_last_frame()
    6     process_user_input()
    update_world(dt)
```

```
8      draw()
      wait(targetTime - time_spent_this_frame())
```

Even if the frame is limited, it's necessary that all updates are tied to our delta time to work correctly. With this loop the game will run **at most** at 60 frames per second, if there is a slowdown the game will slow down under 60 fps, if the game runs faster it won't go over 60fps.

8.3.6 Frame Skipping/Dropping

A common solution used when a frame takes longer to update and render than the target time is using the so-called “frame dropping”. The game won't render the next frame, in an effort to “catch up” to the desired frame rate.

This will obviously cause a perceptible visual stutter.

8.3.7 Multithreaded Loops

Higher budget (AAA) games don't usually use a variation of the “classic” game loop, but instead make use of the capabilities of newer hardware. Using multiple threads (lines of execution) executing at the same time, making everything quicker and the framerate higher.

Multithreaded loops are created in a way that separates the input-update part of the game loop from the drawing part of it. This way the update thread can take care of updating our simulation, while the drawing/rendering loop can take care of drawing the result to screen.

The catch is that we can't just wait for the input-update thread to finish before rendering, that wouldn't make it quicker than just using a one-threaded game loop: instead we make the rendering thread “lag behind” the input-update thread by *1 frame* - this way while the input-update thread takes care of the frame number n , the drawing thread will be rendering the prepared frame number $n - 1$.

Thread						
Updating	1	2	3	4	5	6
Rendering		1	2	3	4	5

This 1-frame difference between updating and rendering introduces lag that can be quantified between *16.67ms* (at 60fps) and *33.3ms* (at 30fps), which needs to be added with the 2-5 ms of the LCD refresh rate, and other factors that can contribute to lag. In some games where extreme precision is needed, this could be considered unacceptable, so a single-threaded loop could be considered more fitting.

8.4 Issues and possible solutions

In this section we have a little talk about some common issues related to the game loop and its timing, and some possible solutions

8.4.1 Frame/Screen Tearing

Screen tearing is a phenomenon that happens when the “generate output” stage of the game loop happens in the middle of the screen drawing a frame.

This makes it so that a part of the drawn frame shows the result of an output stage, while another part shows a more updated version of the frame, given by a more recent game loop iteration.

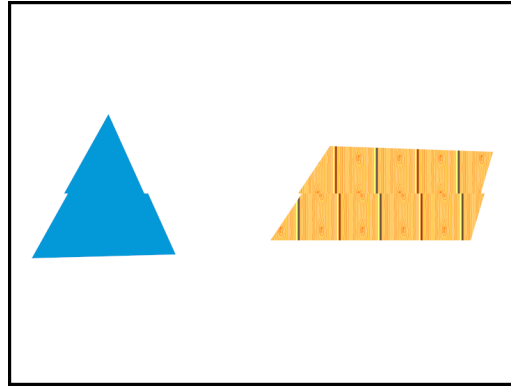


Figure 70: An example of screen tearing

A very used fix for this phenomenon is **double buffering**, where two color buffers are used. While the first is shown on screen, the game loop updates and draws on the second color buffer.

When comes the time to draw the color buffer on screen, an operation called “flipping” is performed, where the second color buffer is shown on screen, so that the game loop can draw on the first color buffer.

To smooth the game, a technique called “triple buffering” can be used, which adds a third color buffer is used to make the animation smoother at the cost of a higher input lag.

8.5 Drawing to screen

When drawing to screen, the greatest majority of games make use of what is called the “painter’s algorithm”, which looks something like the following:

1. Clear the screen
2. Draw The Farthest Background
3. Draw The Second Farthest Background
4. Draw The Tile Map
5. Draw The enemies and obstacles
6. Draw The Player
7. Display everything on screen

As a painter, we draw the background items before the foreground ones, layering each one on top of the other. Sometimes games make use of priority queues to decide which items to draw first, other times game developers (usually under the time constraints of a game jam) just hard-code the draw order.

8.5.1 Clearing the screen

Special note to clearing the screen: this is an operation that sometimes may look useless but, like changing the canvas for a painter, clearing the screen (or actually the “buffer” we’re drawing on) avoids a good deal of graphical glitches.

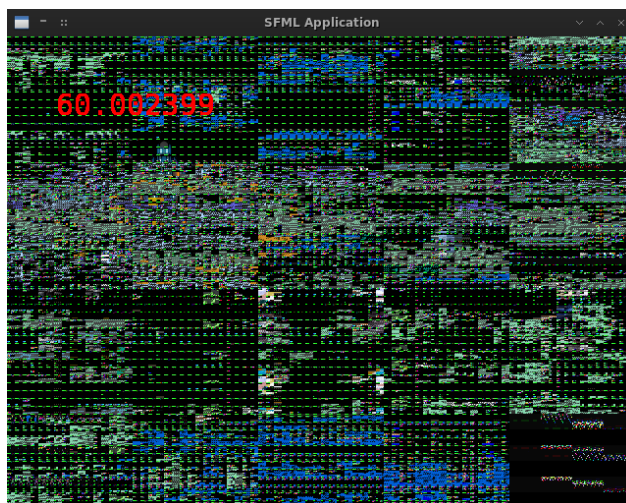


Figure 71: How not clearing the screen can create glitches

In the previous image, we can see how a black screen with only a FPS counter can end up drawing all kinds of glitches when the screen buffer is not cleared: we can clearly see the FPS counter, but the rest of the screen should be empty, instead the GPU is trying to represent residual data from its memory, causing the glitches.

9 Collision Detection and Reaction

Every detection of what is false directs us towards what is true:
every trial exhausts some tempting form of error.

William Whewell

When it comes to collision management, there are two main phases:

- **Collision Detection:** you find out which game objects collided with each other;
- **Collision Reaction:** you handle the physics behind the collision detected, making the game objects react to such collision.

Collisions don't only happen between game objects (two fighters hitting each other), but also between a character and the world (or they would end up just going through the ground).

In this section we'll talk about some ways you can detect and react to collisions.

9.1 Collision Detection: Did it really collide?

First of all, we need to see how we can make sure that two objects really collide with each other.

9.1.1 Collision Between Two Points

This is the simplest case: points are mono-dimensional objects, and the only way two points can collide is when they have the same coordinates.

An example algorithm would be the following:

Listing 16: Point to point collision detection

```
function point_collision(point A, point B):  
2   if A.x == B.x AND A.y == B.y:  
       return True  
4   else  
       return False
```

A possible lazy/shorter version could be:

Listing 17: Shortened version of a point to point collision detection

```
function point_collision(point A, point B):  
2   return A.x == B.x AND A.y == B.y
```

This algorithm consists in a constant number of operations, so it runs in $O(1)$.

Since numbers in computers can be **really** precise, a collision between two points may be a bit too precise, so it could prove useful to have a “buffer” around the point, so that we can say that the two points collided when they're **around the same place**.

9.1.2 Collision Between A Point and a Circle

Now a circle comes into the mix, a circle has two major characteristics: a **center** and a **radius**.

A point is considered inside of a circle when the distance between the point and the center of the circle is *less than or equal* to the radius.

So we need a function that calculates the distance between two points, and then use it to define if a point is inside a circle.

An example could be the following:

Listing 18: Point to circle collision detection

```
structure Circle:
2  // Let's define a circle class/structure
  Point center;
4  Integer radius

6 function distance(Point A, Point B):
  // Calculates the distance between two points
8  return square_root((A.x + B.x)^2 + (A.y + B.y)^2)

10 function circle_point_collision(Circle A, Point B):
  if distance(A.center, B) <= A.radius:
12    return True
  else:
14    return False
```

Again, the lazier version:

Listing 19: Shorter version of a point to circle collision detection

```
structure Circle:
2  // Let's define a circle class/structure
  Point center;
4  Integer radius

6 function distance(Point A, Point B):
  // Calculates the distance between two points
8  return square_root((A.x + B.x)^2 + (A.y + B.y)^2)

10 function circle_point_collision(Circle A, Point B):
  return distance(A.center, B) <= A.radius:
```

Although slightly more heavy, computation-wise, this algorithm still runs in $O(1)$.

9.1.3 Collision Between Two Circles

Let's add another circle into the mix now, we can declare:

Two circles are colliding when the distance between their centers is less or equal the sum of their radii

In pseudo code this would be:

Listing 20: Circle to Circle Collision Detection

```
structure Circle:
2  // Let's define a circle class/structure
  Point center;
4  Integer radius

6 function distance(Point A, Point B):
  // Calculates the distance between two points
8  return square_root((A.x + B.x)^2 + (A.y + B.y)^2)

10 function circle_circle_collision(Circle A, Circle B):
  if distance(A.center, B.center) <= A.radius + B.radius:
12    return True
  else:
14    return False
```

The shorter version would be:

Listing 21: Shorter Version of a Circle to Circle Collision Detection

```
structure Circle:
2  // Let's define a circle class/structure
  Point center;
4  Integer radius

6 function distance(Point A, Point B):
  // Calculates the distance between two points
8  return square_root((A.x + B.x)^2 + (A.y + B.y)^2)

10 function circle_circle_collision(Circle A, Circle B):
  return distance(A.center, B.center) <= A.radius + B.radius:
```

Again, this algorithm performs a number of operations that is constant, so it runs in $O(1)$.

9.1.4 Collision Between Two Axis-Aligned Rectangles (AABB)

This is one of the most used types of collision detection used in games: it's a bit more involved than other types of collision detection, but it's still computationally easy to perform. This is usually called the “Axis Aligned Bounding Box” collision detection, or AABB.

Let's start with a bit of theory. We have two squares:

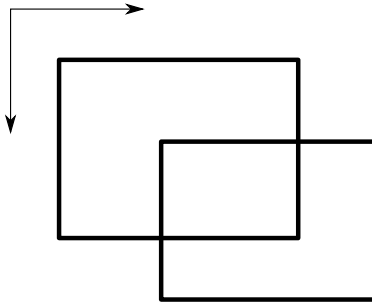


Figure 72: Example used in the AABB collision detection

To know if we may have a collision, we need to check if one of the sides is “inside” (that means between the top and bottom sides) of another rectangle:

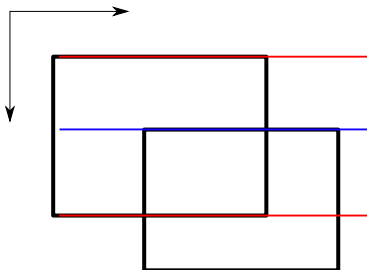


Figure 73: Top-Bottom Check

In this case we know that the “top side” of the second rectangle (highlighted in blue) has a y coordinate between the first rectangle's top and bottom sides' y coordinates (highlighted in red).

Though this is a necessary condition, this is not sufficient, since we may have a situation where this condition is satisfied, but the rectangles don't collide:

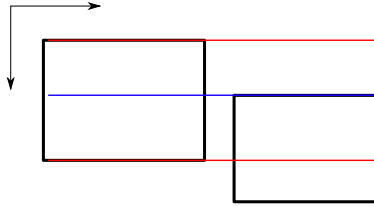


Figure 74: Top-Bottom Check is not enough

So we need to check the other sides also, in a similar fashion:

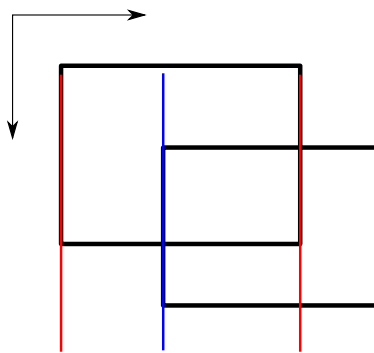


Figure 75: An example of a left-right check

This has to happen for all four sides of one of the rectangle.

Now we can try putting down a bit of code, we'll assume that rectangles are defined by their top-left corner (as usually happens) and their width and height:

Listing 22: Axis-Aligned Bounding Box Collision Detection

```
structure Point:
2   // Rewritten as a memo
   Integer x
4   Integer y

6 structure Rectangle:
   Point corner
8   Integer width
   Integer height

10 function rect_rect_collision(Rectangle A, Rectangle B):
12   if (A.corner.x < B.corner.x + B.width) AND
      (A.corner.x + A.width > B.corner.x) AND
14     (A.corner.y < B.corner.y + B.height) AND
      (A.corner.y + A.height > B.corner.y):
16     return True
   else:
18     return False
```

This complex conditional checks 4 things:

- The left side of rectangle A is **at the left** of the right side of rectangle B;
- The right side of rectangle A is **at the right** of the left side of rectangle B;
- The top side of rectangle A is **over** the bottom side of rectangle B;
- The bottom side of rectangle A is **underneath** the top side of rectangle B.

If all four checks are true, then a collision happened.

The best way to understand this algorithm properly is to test it by hand and convince yourself that it works.

This is a very light algorithm but can quickly become heavy on the CPU when there are many objects to check for collision. We'll see later how to limit the number of checks and make collision detection an operation that is not as heavy on our precious CPU cycles.

9.1.5 Line/Point Collision

We can represent a segment by using its two extreme points, which proves to be a quite inexpensive way to represent a line (it's just two points). Now how do we know if a point is colliding with a line?

To know if a point is colliding with a line we need... Triangles!

Every triangle can be represented with 3 points, and there is a really useful theorem that we can make use of:

The sum of the lengths of any two sides must be greater than, or equal, to the length of the remaining side.

So, given a triangle ABC:

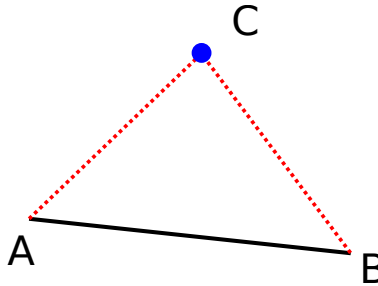


Figure 76: Example of the triangle inequality theorem

We get the following 3 inequalities:

$$\overline{AB} + \overline{BC} \leq \overline{AC}$$

$$\overline{AC} + \overline{BC} \leq \overline{AB}$$

$$\overline{AB} + \overline{AC} \leq \overline{BC}$$

What is more interesting to us is that when the one of the vertices of the triangle is **on** its opposite side, the triangle degenerates:

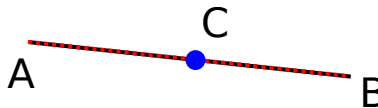


Figure 77: Example of a degenerate triangle

And the theorem degenerates too, to the following:

$$\overline{AC} + \overline{BC} = \overline{AB}$$

So we can calculate the distance between the point and each of the two extremes of the line and we know that when the sum of such distances is equal to the length of the line, the point will be colliding with the line.

In code, it would look something like the following:

Listing 23: Line to Point Collision detection

```
structure Point:
2   Integer x
   Integer y
4
structure Line:
6   Point A
   Point B
8
function distance(Point A, Point B):
10  // Calculates the distance between two points
   return square_root((A.x + B.x)^2 + (A.y + B.y)^2)
12
function line_point_collision(Point pt, Line ln):
14  // First, let's calculate the length of the line
   length = distance(ln.A, ln.B)
16  // Now let's calculate the distance between the point pt
   // and the point "A" of the line
   pt_a = distance(ln.A, pt)
18  // Same Goes for the distance between pt and "B"
   pt_b = distance(ln.B, pt)
20  // Now for the detection
   if (pt_a + pt_b == length):
22     return True
24  else:
   return False
```

It could prove useful to put a “buffer zone” in here too, so that the collision detection doesn’t result too jerky and precise.

9.1.6 Line/Circle Collision

As in the previous paragraph, we memorize a line as a pair of Points, so checking if the circle collides with either end of the line is easy, using the Point/Circle collision algorithm.

Listing 24: Partial Implementation of a Line to Circle Collision Detection

```
structure Point:
2   Integer x
   Integer y
4
structure Line:
6   Point A
   Point B
8
structure Circle:
10  Point center
   Integer radius
12
...
14
function line_circle_collision(Circle circle, Line line):
```

```
16     collides_A = circle_point_collision(circle, line.A)
    collides_B = circle_point_collision(circle, line.B)
18     if (collides_A OR collides_B):
        return True
20     ...
```

Now our next objective is finding the closest point **on the line** to the center of our circle. The details and demonstrations on the math behind this will be spared, just know the following:

Given a line \overline{AB} between points $A = (x_1, y_1)$ and $B = (x_2, y_2)$ and a point $P = (x_k, y_k)$, the point on the line closest to P has coordinates:

$$x = x_1 + u \cdot (x_2 - x_1)$$

$$y = y_1 + u \cdot (y_2 - y_1)$$

With:

$$u = \frac{(x_k - x_1) \cdot (x_2 - x_1) + (y_k - y_1) \cdot (y_2 - y_1)}{||B - A||^2}$$

That's a lot of math!

We need to be careful though, cause this formula gives us the point for an *infinite* line, so the point we find could be outside of our line. We will use the line/point algorithm to check for that.

After we made sure the point is on the line, we can measure the distance between such point and the center of our circle, if such distance is less than the radius, we have a hit! (Or just apply the circle/point collision algorithm again).

The final algorithm should look something like this:

Listing 25: Line to circle collision detection

```
structure Point:
2     Integer x
    Integer y
4
structure Line:
6     Point A
    Point B
8
structure Circle:
10    Point center
    Integer radius
12
function distance(Point A, Point B):
14    // Calculates the distance between two points
    return square_root((A.x + B.x)^2 + (A.y + B.y)^2)
16
function line_point_collision:
18    ...
20
function circle_point_collision:
    ...
22
function line_circle_collision(Circle circle, Line line):
24    // We check the ends first
    collides_A = circle_point_collision(circle, line.A)
26    collides_B = circle_point_collision(circle, line.B)
    if (collides_A OR collides_B):
```

```
28     return True
    // We pre-calculate "u", we'll use some variables for readability
30     x1 = line.A.x
    x2 = line.B.x
32     xk = circle.center.x
    y1 = line.A.y
34     y2 = line.B.y
    yk = circle.center.y
36     u = ((xk - x1) * (x2 - x1) + (yk - y1) * (y2 - y1)) / (distance(line.A, line.B))^2
    // Now let's calculate the x and y coordinates
38     x = x1 + u * (x2 - x1)
    y = y1 + u * (y2 - y1)
40     // "Reuse", we'll use some older functions, let's create a point, with the
    coordinates we found
    P = Point(x,y)
42     // Let's check if the "closest point" we found is on the line
    if (line_point_collision(line, P)) == False:
44         // If the point is outside the line, we return false, because the ends have
        already been checked against collisions
        return False
46     else:
        // Let's Reuse the Point/Circle Algorithm
48     return circle_point_collision(circle, P)
```

9.1.7 Point/Rectangle Collision

If we want to see if a point collides with a rectangle is really easy, we just need to check if the point's coordinates are inside the rectangle.

Listing 26: Point/Rectangle collision detection

```
function pointRectCollision(float x1, float y1, float rectx, float recty, float rectwidth
, float rectheight):
2    // We check if the point is inside the rectangle
    return x1 >= rectx AND x1 <= rectx + rectwidth AND y1 >= recty AND y1 <= recty +
    rectheight
```

9.1.8 Point/Triangle Collision

A possible way to define if a point is inside a triangle, we can use a bit of geometry.

We can use *Heron's formula* to calculate the area of the original triangle, and compare it with the sum of the areas created by the 3 triangles made from 2 points of the original triangle and the point we are testing.

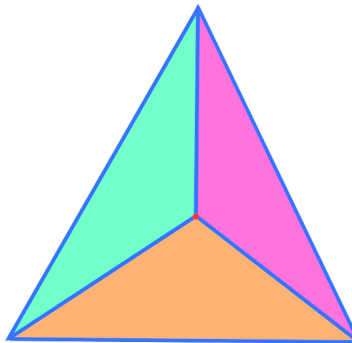


Figure 78: Point/Triangle Collision Detection: division into sub-triangles

If the sum of the 3 areas (represented in different colors in the figure) equals to the original calculated area, then we know that the point is inside the triangle.

Let's see the code:

Listing 27: Point/Triangle Collision Detection

```
function point_triangle_collision(float px, float py, float x1, float y1, float x2, float
y2, float x3, float y3):
2   float original_area = abs((x2 - x1) * (y3 - y1) - (x3 - x1) * (y2 - y1))
   float area1 = abs((x1-px)*(y2-py) - (x2-px)*(y1-py))
4   float area2 = abs((x2-px)*(y3-py) - (x3-px)*(y2-py))
   float area3 = abs((x3-px)*(y1-py) - (x1-px)*(y3-py))
6   if (area1 + area2 + area3 == original_area):
       return True
8   else:
       return False
```

9.1.9 Circle/Rectangle Collision

First of all we need to identify which side of the rectangle we should test against, so if the centre of the circle is to the right of the rectangle, we will test against the right edge of the rectangle, if it's above we'll test against the top edge and so on...

After that, we just perform some math on the distances and calculated values to detect if the circle collides with the rectangle.

Listing 28: Rectangle to Circle Collision Detection

```
structure Point:
2   // Rewritten as a memo
   Integer x
4   Integer y

6 structure Rectangle:
   // Let's define a rectangle class/structure
8   Point corner
   Integer width
10  Integer height

12 structure Circle:
   // Let's define a circle class/structure
14  Point center;
   Integer radius

16 function circle_rectangle_collision(Circle circ, Rectangle rect):
18  // Detects a collision between a circle and a rectangle

20  // These variables are used as the coordinates we should test against
   // They are temporarily set to the circle center's coordinates for a reason we'll see
   soon
22  Integer tx = circ.center.x
   Integer ty = circ.center.y

24
   // Let's detect which edge to test against on the x axis
26  if (circ.center.x < rect.corner.x):
       // We're at the left of the rectangle, test against the left side
28     tx = rect.corner.x
   else if (circ.center.x > rect.corner.x + rect.width):
       // We're at the right of the rectangle, test against the right side
30     tx = rect.corner.x + rect.width

32
   // Same thing on the vertical axis
```

```
34     if (circ.center.y < rect.corner.y):
35         // We're above the rectangle, test against the top side
36         ty = rect.corner.y
37     else if (circ.center.y > rect.corner.y + rect.height):
38         // We're below the rectangle, test against the bottom side
39         ty = rect.corner.y + rect.height
40
41     // Let's get the distance between the testing coordinates and the circle center
42     Integer distanceX = circ.center.x - tx
43     Integer distanceY = circ.center.y - ty
44     Float distance = square_root(distanceX^2 + distanceY^2)
45
46     // Note that if the center of the circle is inside the rectangle, the testing
47     // coordinates will be the circle's center itself, thus the next conditional will always
48     // return true
49
50     if (distance <= circ.radius):
51         return True
52
53     // Default to false in case no collision occurs
54     return False
```

9.1.10 Line/Line Collision

Line/Line collision is quite simple to implement once you know the inner workings of geometry, but first we need to explain the thought behind this algorithm, so... **math warning!!**

Let's look at the following image:

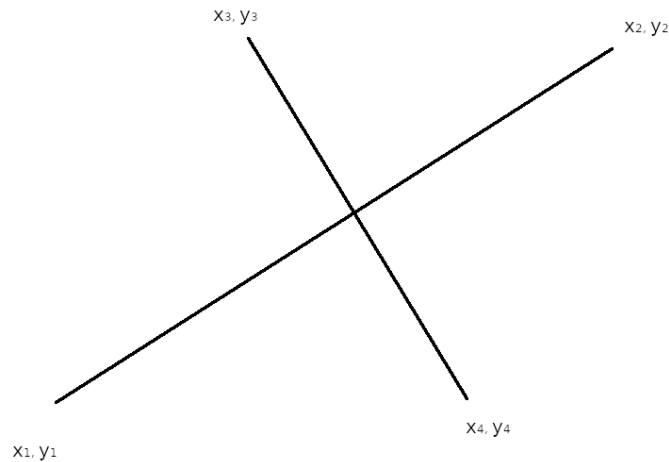


Figure 79: Example image for line/line collision

A generic point P_a of line A can be represented with the following formula:

$$P_a = P_1 + u_a \cdot (P_2 - P_1)$$

which translates into the coordinate-based equations:

$$\begin{cases} x_a = x_1 + u_a \cdot (x_2 - x_1) \\ y_a = y_1 + u_a \cdot (y_2 - y_1) \end{cases}$$

This makes us understand that any point of line A can be represented by its starting point P_1 , plus a certain fraction (represented by u_a) of the vector represented by $P_2 - P_1$.

This also means that $0 \leq u_a \leq 1$, else the point won't be on the segment.

In the same way, a generic point P_b of line B can be represented with:

$$P_b = P_3 + u_b \cdot (P_4 - P_3)$$

which becomes:

$$\begin{cases} x_b = x_3 + u_b \cdot (x_4 - x_3) \\ y_b = y_3 + u_b \cdot (y_4 - y_3) \end{cases}$$

The two lines will collide when $P_a = P_b$, so we get the following equations:

$$\begin{cases} x_1 + u_a \cdot (x_2 - x_1) = x_3 + u_b \cdot (x_4 - x_3) \\ y_1 + u_a \cdot (y_2 - y_1) = y_3 + u_b \cdot (y_4 - y_3) \end{cases}$$

That need to be solved in the u_a and u_b variables.

The result is:

$$\begin{cases} u_a = \frac{(x_4 - x_3) \cdot (y_1 - y_3) - (y_4 - y_3) \cdot (x_1 - x_3)}{(y_4 - y_3) \cdot (x_2 - x_1) - (x_4 - x_3) \cdot (y_2 - y_1)} \\ u_b = \frac{(x_2 - x_1) \cdot (y_1 - y_3) - (y_2 - y_1) \cdot (x_1 - x_3)}{(y_4 - y_3) \cdot (x_2 - x_1) - (x_4 - x_3) \cdot (y_2 - y_1)} \end{cases}$$

Substituting either of the results in the corresponding equation for the line will give us the intersection point (which may be useful for some particle effects).

Now some notes on our solution:

- If the denominator for the equations for u_a and u_b equals to zero, the two lines are parallel
- If both the numerator and denominator for u_a and u_b are equal to zero, the two lines are coincident
- If both $0 \leq u_a \leq 1$ and $0 \leq u_b \leq 1$ then the two segments collide.

Now we can translate all this math into code:

Listing 29: Implementation of the line/line collision detection

```
function lineLineCollision(float x1, float y1, float x2, float y2, float x3, float y3,
    float x4, float y4):
2    // Let's calculate the denominator, this will allow us to avoid a
    // "divide by zero" error
4    float den = ((y4 - y3) * (x2 - x1) - (x4 - x3) * (y2 - y1))

6    if den == 0:
        // The lines are parallel
8        return false

10    float uA = ((x4 - x3) * (y1 - y3) - (y4 - y3) * (x1 - x3)) / den
    float uB = ((x2 - x1) * (y1 - y3) - (y2 - y1) * (x1 - x3)) / den
12
```



```
14     // Let's see if uA and uB tell us the lines are colliding
    if (uA >= 0 AND uA <= 1) AND (uB >= 0 AND uB <= 1):
        return true
16
    // If not, they don't collide
18    return false
```

This collision detection algorithm can be useful for line-based puzzle games, like the untangle puzzle.

9.1.11 Line/Rectangle Collision

Given the previous explanation about the Line/Line collision detection, it's quite easy to build a Line/Rectangle algorithm; distinguishing the cases where we want to account for a segment being completely inside of a rectangle or not.

Listing 30: Implementation of the line/rectangle collision detection

```
function lineLineCollision(float x1, float y1, float x2, float y2, float x3, float y3,
    float x4, float y4):
2    // our previous implementation of the line/line collision detection

4    function pointRectCollision(float x1, float y1, float rectx, float recty, float rectwidth
        , float rectheight):
        // our previous implementation of a point/rectangle collision detection
6
    function lineRectangleCollision(float x1, float y1, float x2, float y2, float rectx,
        float recty, float rectwidth, float rectheight):
8        // If we want to test if a line is completely inside of a rect, we just need
        // to see if any of its endpoints is inside the rectangle
10        if (pointRectCollision(x1, y1, rectx, recty, rectwidth, rectheight) OR
            pointRectCollision(x2, y2, rectx, recty, rectwidth, rectheight)):
            // At least one of the ends of the segment is inside the rectangle
12            return True

14        // Now to test the rectangle against the line, if it's not completely inside
        left = lineLineCollision(x1, y1, x2, y2, rectx, recty, rectx, recty + rectheight)
16        right = lineLineCollision(x1, y1, x2, y2, rectx + rectwidth, recty, rectx + rectwidth
            , recty + rectheight)
        top = lineLineCollision(x1, y1, x2, y2, rectx, recty, rectx + rectwidth, recty)
18        bottom = lineLineCollision(x1, y1, x2, y2, rectx, recty + rectheight, rectx +
            rectwidth, recty + rectheight)

20        if left OR right OR top OR bottom:
            // We hit one of the sides, we are colliding
22            return True

24        // In any other case, return false
        return False
```

This can prove useful to test for “line of sight” inside an AI algorithm.

9.1.12 Point/Polygon Collision

[This section is a work in progress and it will be completed as soon as possible]

9.1.13 Circle/Polygon Collision

[This section is a work in progress and it will be completed as soon as possible]

9.1.14 Rectangle/Polygon Collision

[This section is a work in progress and it will be completed as soon as possible]

9.1.15 Line/Polygon Collision

[This section is a work in progress and it will be completed as soon as possible]

9.1.16 Polygon/Polygon Collision

[This section is a work in progress and it will be completed as soon as possible]

9.1.17 Pixel-Perfect collision

Pixel perfect collision is the most precise type of collision detection, but it's also by far the slowest.

The usual way to perform collision detection is using **bitmasks** which are 1-bit per pixel representation of the sprites (white is usually considered a “1” while black is considered a “0”).

A logic “AND” operation is performed, pixel-by-pixel, on the bitmasks; with the sprite position taken in consideration, as soon as the first AND returns a “True” a collision occurred.

Listing 31: Example of a possible implementation of pixel perfect collision detection

```
1  structure Color:
2      Integer colorData
3      Boolean isWhite()
4
5      structure Bitmask:
6          Color[] data
7          Color getColor(x, y)
8
9      structure Sprite:
10         Bitmask bitmask
11         Integer x
12         Integer y
13         Integer width
14         Integer height
15
16 function pixel_perfect_collision(Sprite A, Sprite B):
17     // Calculate the intersecting rectangle to limit checks
18     x1 = max(A.x, B.x)
19     x2 = min((A.x + A.width), (B.x + B.width))
20
21     y1 = max(A.y, B.y)
22     y2 = min((A.y + A.height), (B.y + B.height))
23
24     // For each pixels in the intersecting rectangle, let's check
25     for each y from y1 to y2:
26         for each x from x1 to x2:
27             a = A.bitmask.getColor(x - A.x, y - A.y)
28             b = B.bitmask.getColor(x - B.x, y - B.y)
29
30             if (a.isWhite() AND b.isWhite()):
31                 return True
32
33     // If no collision is occurred by the end of the checking, we're safe
34     return False
```

This algorithm has a time complexity of $O(n \cdot m)$ where n is the total number of pixels of the first bitmask, while m is the total number of pixels in the second bitmask.

9.1.18 Multi-pass collision detection

It's possible to have a more precise collision detection at a lower cost by combining different collision detection algorithms.

The most common way to apply a multi-pass collision detection is by dividing the process in a “broad” and a “fine” pass.

The broad pass can use a very simple algorithm to check for the possibility of a collision, the algorithms used are usually computationally cheap, such as:

- Circle/Circle Collision Detection
- AABB Collision detection

When the simpler algorithm detects the possibility of a collision, a more precise algorithm is used to check if a collision really happened, usually such finer algorithms are computationally expensive and will benefit from the first “broad pass” filter, thus avoiding useless heavy calculations.

9.2 Finding out who hit what

Now we need to find which game objects collided, and this can be easily one of the most expensive parts of our game, if not handled correctly.

This section will show how knowing which items will surely **not** collide can help us optimize our algorithms.

We need to remember that each object (as good practices suggest) know only about themselves, they don't have “eyes” like us, that can see when another object is approaching them and thinking “I'm gonna collide”. The only thing we can do is having “someone else” take care of checking for collisions.

As an example, we'll take the following situation:

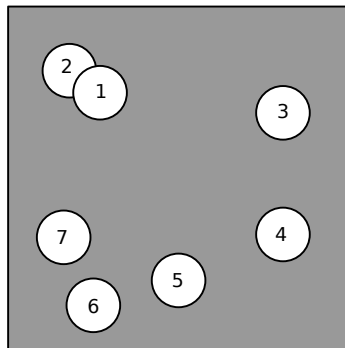


Figure 80: Example for collision detection

We can evidently see how circles 1 and 2 are colliding, but obviously our game won't just “know” without giving it a way to think about how two objects collide.

9.2.1 The Brute Force Method

The simplest method is the so-called “brute force” method: you don't know which items may collide? Just try them all.

So if we consider a list of 7 game objects, we'll need to see if 1 collides with 2, 1 collides with 3, ..., 2 collides with 1, ...

An algorithm of this type could be the following:

Listing 32: Brute Force Method of collision search

```
function is_collision(item A, item B):  
2    // Defines how two items collide (being circles, this could be a difference of radii)  
  
4    items = [1, 2, 3, 4, 5, 6, 7]  
    colliding_items = []  
6  
    for A in items:  
8        for B in items:  
            if A is not B:  
10           // We avoid checking if an item collides with itself, for obvious reasons  
            if is_collision(A, B):  
12                colliding_items.add((A, B))
```

This algorithm runs in $O(n^2)$, because it checks every item with every other, even with itself.

In this example, the algorithm completes in 49 steps, but you can imagine how a game could slow down when there is an entire world to update (remember the collision detection, among with other updates and rendering/drawing, must happen in less than 16.67 and 33.33ms, so if you can save time, you totally should).

9.2.2 Building Quad Trees

A nice idea would be being able to limit the number of tests we perform, since the brute force method can get really expensive really quickly.

When building quad-trees, we are essentially dividing the screen in “quadrants” (and if necessary, such quadrants will be divided into sub-quadrants), detect which objects are in such quadrants and test collisions between objects that are inside of the same quadrant.

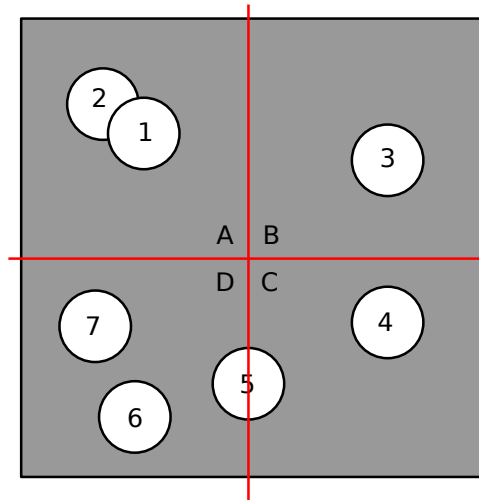


Figure 81: Graphical example of a quad tree, overlaid on the reference image

With the original brute force method, we will make at most 49 tests for 7 items (although it can be optimized), while with quad trees we will perform:

- 2 tests on quadrant A (1 against 2, and 2 against 1);
- No tests on quadrant B (an object alone cannot collide against other objects);

- 2 tests on quadrant C (4 against 5, and 5 against 4);
- 6 tests on quadrant D (5-6, 5-7, 6-7, 6-5, 7-6, 7-5)

For a total of 10 tests, which can be further optimized (by 50%) by avoiding testing objects that have already been tested.

[This section is a work in progress and it will be completed as soon as possible]

9.2.3 Calculating the position of tiles

When you are using tiles to build a level, it can prove hard to be able to use quad trees or brute force methods to limit the number of collision checks inside your game.

Using a bit of math is probably the easiest and most efficient method to find out which collisions happened.

Let's take an example level:

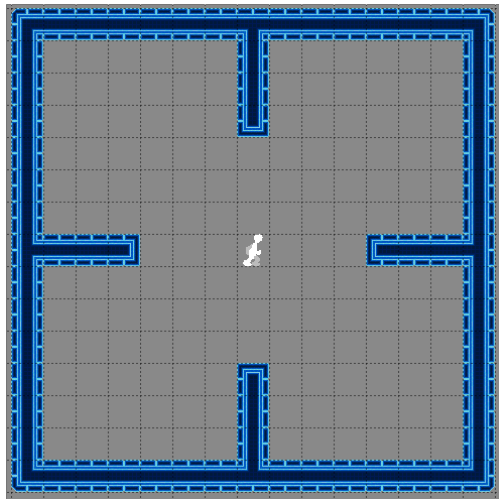


Figure 82: Example tile-based level

If a game entity is falling, like in the following example:

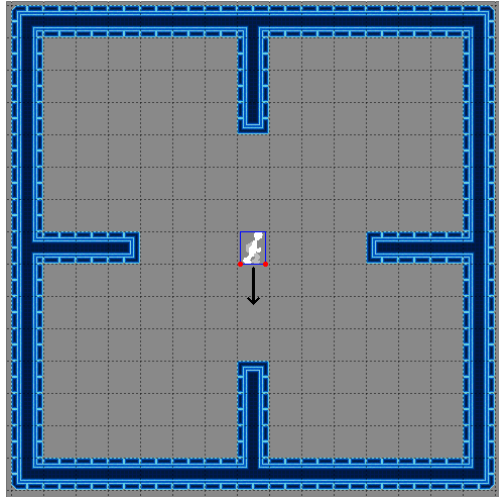


Figure 83: Tile-based example: falling

Using the simple AABB collision detection, we will need to check only if the two lowest points of the sprite have collided with any tile in the level.

First of all let's consider a level as a 2-dimensional array of tiles and all the tiles have the same size, it is evident that we have two game entities that work with different measures: the character moves pixel-by-pixel, the ground instead uses tiles. We need something to make a conversion.

Assuming `TILE_WIDTH` and `TILE_HEIGHT` as the sizes of the single tiles, we'll have the following function:

Listing 33: Converting player coordinates into tile coordinates

```

constant TILE_WIDTH = 32
2 constant TILE_HEIGHT = 32

4 function convert_pixels_to_tile(Integer x, Integer y):
    // Converts a point into tile coordinates
6     tile_x = floor(x / TILE_WIDTH)
    tile_y = floor(y / TILE_HEIGHT)
8     return (tile_x, tile_y)

```

To know which tiles we need to check for collision, we just have to check the two red points (see the previous image), use the conversion function and then do a simple AABB check on them.

Listing 34: Tile-based collision detection

```

constant TILE_WIDTH = 32
2 constant TILE_HEIGHT = 32

4 struct Rectangle{
    // A rectangle will represent the player
6     Point corner
    Integer width
8     Integer height
}

10
12 function convert_pixels_to_tile(Integer x, Integer y):
    // Converts a point into tile coordinates
    tile_x = floor(x / TILE_WIDTH)
14     tile_y = floor(y / TILE_HEIGHT)
    return (tile_x, tile_y)

```

```
16 // We assume the player is falling, so no check will be shown here
18 points_to_check = [
20     Point(player.corner.x, player.corner.y + player.height),
21     Point(player.corner.x + player.width, player.corner.y + player.height),
22 ]
23 for each point in points_to_check:
24     detected_tile_coordinates = convert_pixels_to_tile(point.x, point.y)
25     detected_tile = get_tile(detected_tile_coordinates)
26     if AABB(player, detected_tile.rectangle):
27         // React to the collision
```

Considering that this algorithm calculates its own colliding tiles, we can state that its complexity is $O(n)$ with n equal to the number of possibly colliding tiles calculated.

If an object is bigger than a single tile, like the following example:

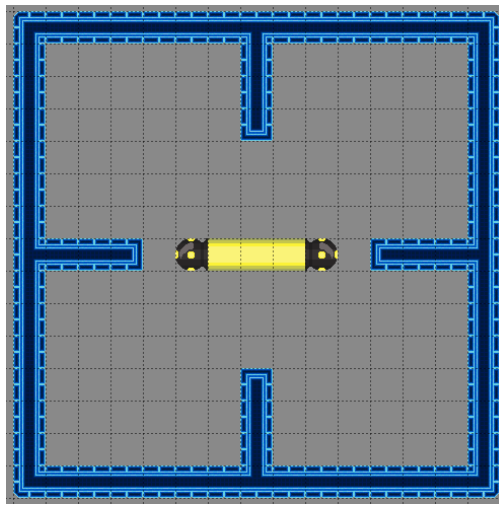


Figure 84: Example tile-based level with a bigger object

We will need to calculate a series of intermediate points (using the `TILE_WIDTH` and `TILE_HEIGHT` measures) that will be used for the test

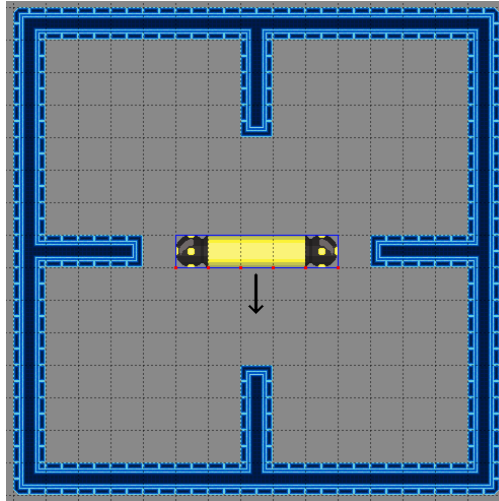


Figure 85: Tile-based example with a bigger object: falling

And using the same method the colliding tiles can be found without much more calculations than the previous algorithm, actually we can use exactly the same algorithm with a different list of points to test.

9.3 Collision Reaction/Correction

When you are sure, via any algorithm, that a collision has occurred, you now have to decide how to react to such collision. You may want to destroy the player or the target, or you may want to correct the behaviour, thus avoiding items getting inside walls.

9.3.1 HitBoxes vs HurtBoxes

First of all, we need to explain the difference between a “HurtBox” and a “HitBox”.

Such difference can be more or less important, depending on the game that is coded, and sometimes the two concepts can be confused.

A **HitBox** is a shape (usually a rectangle, see [Collision Between Two Axis-Aligned Rectangles \(AABB\)](#)) that is used to identify where a certain entity can *hit* another entity. For the player a “hitbox” could encase their sword while attacking.

A **HurtBox** is instead a shape that is used to identify where a certain entity can *get hurt* by another entity. For the player a “hurtbox” could be their body.

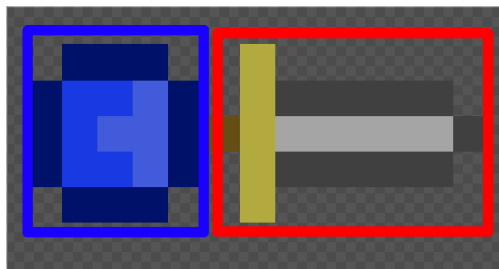


Figure 86: Example of a hitbox (red) and a hurtbox (blue)

9.3.2 Collision Reaction Methods

It has happened: a collision occurred and now the two objects are overlapping.

How do we react to this event in a convincing (not necessarily “realistic”) and efficient manner? There are a lot of methods to react to collisions and below we will show some of the most used, along with some interesting ones.

We will use the following image as reference for each collision reaction:

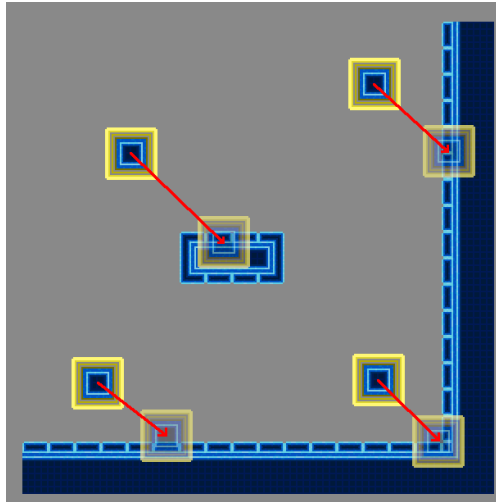


Figure 87: Images used as a reference for collision reaction

We will study each case separately, and each case will be a piece of this reference image.

9.3.2.1 The Direction + Velocity Method

This is the simplest method, computationally speaking: as soon as the objects gets inside of a wall, you push it back according to the direction its velocity has or just the direction of the character itself.

9.3.2.1.1 How it works

This works when you treat the x and y axis separately, updating one, checking the collisions that come up from it, update the other axis and check for new collisions.

Listing 35: Code for the direction + velocity collision reaction

```
// Direction+Velocity collision reaction with rectangles
2 function update(dt):
    // ...
4     player.position.x = player.position.x + player.x_speed * dt
    // Refer to your favourite collision detection and broad/fine passes
6     if collision(player, object):
        if player.x_speed > 0: // going right
8             player.position.x = object.rectangle.left // reset position
            player.x_speed = 0 // stop the player
10        if player.x_speed < 0: // going left
            player.position.x = object.rectangle.right // reset position
12            player.x_speed = 0 // stop the player
        player.position.y = player.position.y + player.y_speed * dt
14    // Again, refer to your favourite collision detection and broad/fine passes
    if collision(player, object):
```

```
16     if player.y_speed > 0: // going down
17         player.position.y = object.rectangle.top // reset position
18         player.y_speed = 0 // stop the player
19     if player.y_speed < 0: // going up
20         player.position.y = object.rectangle.bottom // reset position
21         player.y_speed = 0 // stop the player
22     // ...
```

9.3.2.1.2 Analysis

Let's see how this method reacts in each situation.

When we are trying to fall on the ground, this method works as follows:

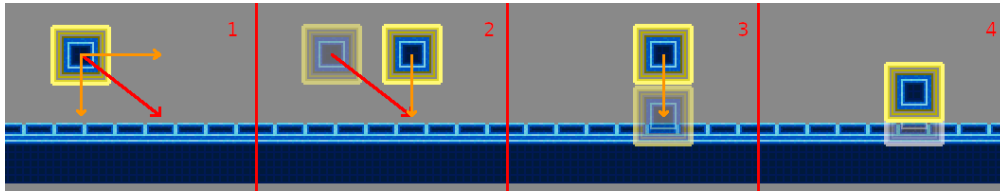


Figure 88: How the direction + velocity method reacts to collisions on a horizontal plane

1. We divide the movement vector in its x and y components.
2. We move along the x axis and check for collisions, in this case there are none (the ghost represents our previous position).
3. We move along the y axis, after checking for collisions we find that we are colliding on the ground (the ghost represents our next position).
4. We react to the collision by moving the sprite on top of the ground.

9.3.2.1.3 Quirks and issues

This method can be used only with completely solid platforms. If you want to make use of platforms that you can cross one-way, since you may get teleported around when your velocity changes direction.

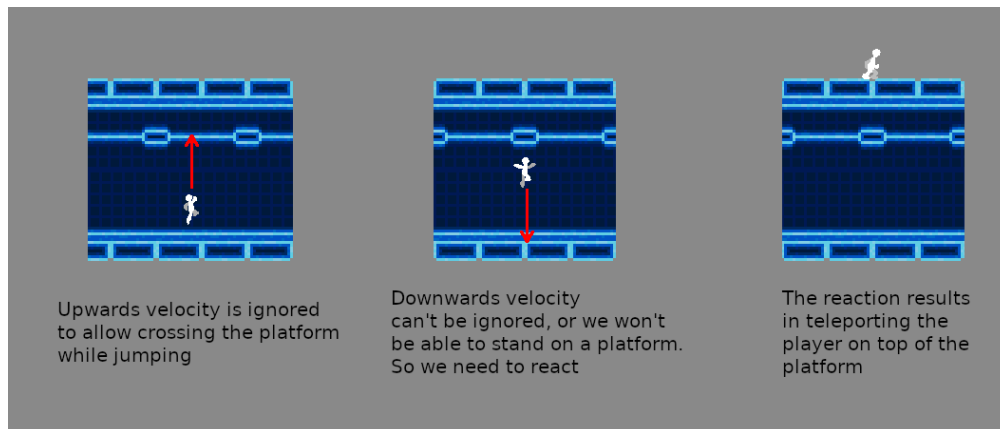


Figure 89: How velocity changing direction can teleport you

In the previous example we try to jump on a platform by going through it, but our jump quite doesn't make it. Since velocity has changed direction, we end up being teleported over the platform, which is considered a

glitch.

9.3.2.2 Shallow-axis based reaction method

This method works in a similar fashion to the direction and velocity method, but prioritizes reactions on the axis that shows the shallowest overlap.

This requires measuring how much the objects overlap on each axis, which can be a little more involved, but not really expensive.

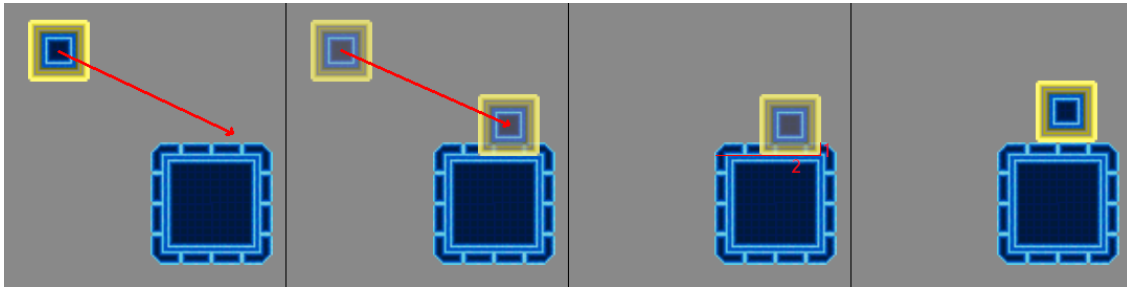


Figure 90: Example of shallow-axis based reaction

In the previous picture, we can see how the algorithm chooses to solve the collision on the y axis first and only on the x axis after; but since solving the y axis solves the collision, no reaction is performed on the x axis.

[This section is a work in progress and it will be completed as soon as possible]

9.3.2.3 The “Snapshot” Method

This method is a bit more involved, but allows for a finer control over how you go through or collide with certain obstacles.

The secret to this method is taking a snapshot of the object’s position before its update phase and do a series of comparisons with the position after the update.

Listing 36: Example of the "snapshot" collision reaction method

```
// Snapshot collision reaction
2 // All the sprite origins are on the top-left corner of the entity
  snapshot = player_instance.copy() // The "snapshot"
4
  // Update the player_instance here
6 player_instance = player_instance + (velocity * dt)

8 // Now check for collisions
  ...
10 for each block colliding with player_instance:
    if (snapshot.y >= block.y + block.height) AND (player_instance.y < block.y + block.
    height):
12     // We are coming on the block from below, react accordingly
    // Ignoring this reaction will allow players to phase through blocks when coming
    from below
14     player_instance.y = block.y + block.height

16 if (snapshot.y + snapshot.height <= block.y) AND (player_instance.y + snapshot.height
    > block.y):
    // We are coming on the block from above
18     player_instance.y = block.y
    player_instance.on_ground = true
```

```
20     if (snapshot.y + snapshot.width <= block.x) AND (player_instance.x > block.x):
21         // We are coming on the block from left
22         player_instance.x = block.x - player_instance.width
23
24     if (snapshot.y >= block.x + block.width) AND (player_instance.x < block.x + block.
25         width):
26         // We are coming on the block from right
27         player_instance.x = block.x + block.width
```

This method solves the problem given by platforms that can be crossed one-way.

9.3.2.4 The “Tile + Offset” Method

[This section is a work in progress and it will be completed as soon as possible]

9.4 Common Issues with Collision Detection

9.4.1 The “Bullet Through Paper” problem

The “bullet through paper” is a common problem with collision detection, when an obstacle is really thin (our “paper”), and the object is really fast and small (the “bullet”) it can happen that collision is not detected.

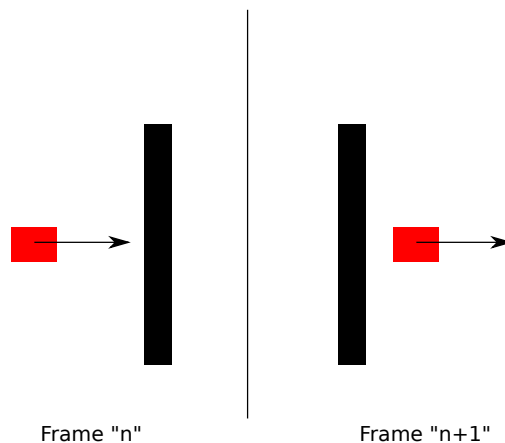


Figure 91: Example of the “Bullet through paper” problem

The object is going so fast that it manages to go through the entirety of the obstacle in a single frame.

Possible solutions to this problems are various, some even going out of the realm of the so-called “time-stepping techniques” (like speculative contacts or ray casting) that can be very expensive from a computational standpoint.

Such solutions should therefore be enabled (or implemented) only for fast-moving objects and only if necessary, since resources and time are at a premium in most cases.

9.4.2 Precision Issues

Sometimes it can happen that the position is reset incorrectly due to machine precision or wrong rounding, this can lead to the character that looks spazzy or just going through the floor at random times. The solution to these issues is making sure that the position and state are set correctly so that there are no useless state changes between frames.

Sometimes the “spazziness” of the character derives from the fact that collision reaction sets the character one pixel over the floor, triggering the “falling” state, the next frame the state would be changed to “idle” and then in the frame “n+2” the cycle would restart with collision reaction putting the character one pixel over the floor.

10 Cameras

The great majority of games don't limit the size of their own maps to the screen size only, but instead they make maps way bigger than the screen.

To be able to manage and display such maps to the screen, we need to talk about cameras (sometimes called “viewports”): they will allow us to show only a portion of our world to the user, making our game much more expansive and involving.

10.1 Screen Space vs. Game Space

Before starting with the most used type of cameras, we need to distinguish between what could be called “screen space” and what is instead “game space” (sometimes called “map space”).

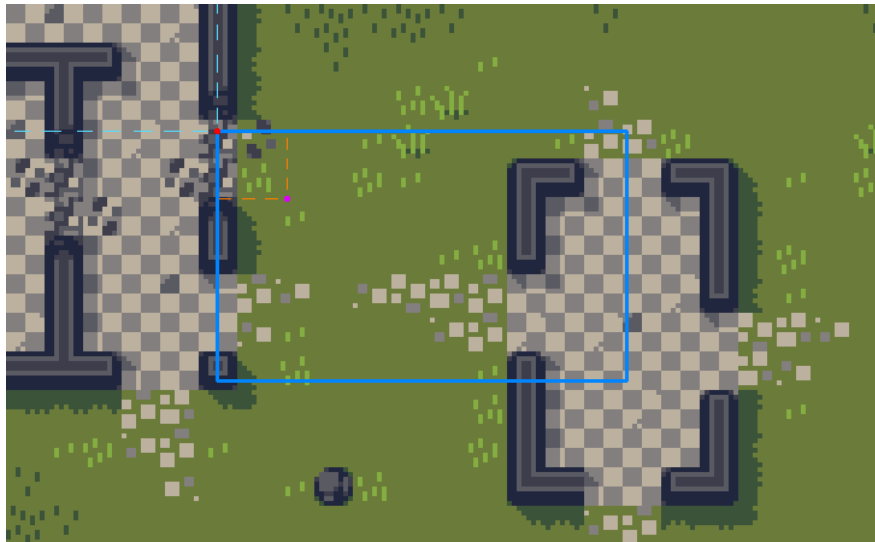


Figure 92: Reference Image for Screen Space and Game Space¹

We talk about “game space” when the coordinates of a point we are talking about are referred to the top-left corner of the entire game (or level) map.

Instead we talk about “screen space” when the coordinate of such point are referred to the top-left corner of the screen.

Looking at our reference image, we can see how different the coordinates of the magenta dot are in screen space and in map space.

It is possible to convert screen space to map space and vice-versa by accounting for the viewport offset (represented by the red dot in the reference image), like follows:

$$\text{screen coordinates} = \text{map coordinates} - \text{viewport coordinates}$$

$$\text{map coordinates} = \text{screen coordinates} + \text{viewport coordinates}$$

In a more friendly way, we can see our viewport as a “window” that moves around the map. Alternatively, we can see it as a viewport that is still all the time but has the map scrolling under it.

¹Jawbreaker tileset, listed as public domain at <https://adamatomic.itch.io/jawbreaker>

10.2 Most used camera types

10.2.1 Static Camera

This is the simplest camera we can implement: each level has the size of the screen (or of the virtual resolution we decided, see [Virtual Resolution](#)), and every time we go out of the map, the screen fades to black and back to the new “room”.

[This section is a work in progress and it will be completed as soon as possible]

10.2.2 Grid Camera

This is an improvement on the static camera formula, each level (or room) has the size of the screen (or virtual resolution we chose), every time we go out of the map, the screen scrolls into the new section. This camera is used by the first Legend Of Zelda game for the Nintendo Entertainment System.

[This section is a work in progress and it will be completed as soon as possible]

10.2.3 Position-Tracking Camera

This camera is a bit more involved: the viewport tracks the position of the player and moves accordingly, so to keep the character centered on the screen. There are two types of position tracking cameras that are used in videogames: horizontal-tracking and full-tracking cameras.

This type of camera can has some serious drawbacks when sudden and very quick changes of direction are involved: since the camera tracks the player all the time, the camera can feel twitchy and over-reactive; this could cause uneasiness or even nausea.

10.2.3.1 Horizontal-Tracking Camera

Horizontal-tracking cameras keep the player in the center of the screen horizontally, while jumps don't influence the camera position. This is ideal for games that span horizontally, since we won't have the camera moving when jumping and temporarily hiding enemies we may fall on.

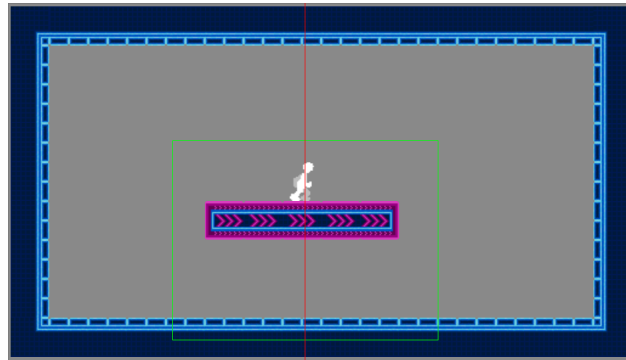


Figure 93: Example of an horizontally-tracking camera

This is the camera used in the classic Super Mario Bros. for the Nintendo Entertainment System.

[This section is a work in progress and it will be completed as soon as possible]

10.2.3.2 Full-Tracking Camera

Sometimes our levels don't span only horizontally, so we need to track the player in both axes, keeping it in the center of the screen at all times. This is good for platformers that don't require extremely precise maneuvering, since precise maneuvering could result in way too much movement from the camera.

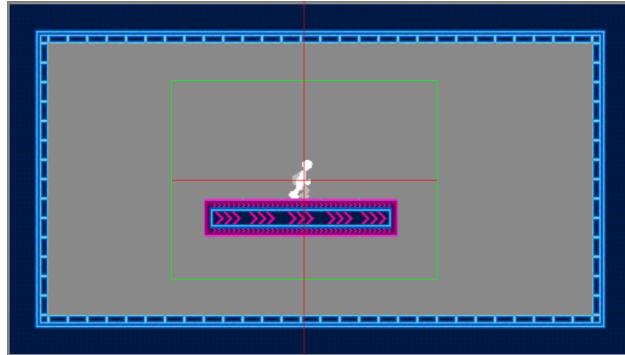


Figure 94: Example of a full-tracking camera

[This section is a work in progress and it will be completed as soon as possible]

10.2.4 Camera Trap

The “Camera Trap” system was invented to eliminate, or at least mitigate, the issues given by the position tracking camera. The playable character is encased in a “trap” that, when “escaped” makes the camera catch up in an effort to put the player back in such “trap”.

The trap is represented by an invisible rectangle which can be visualized on screen in case you need to debug your camera.

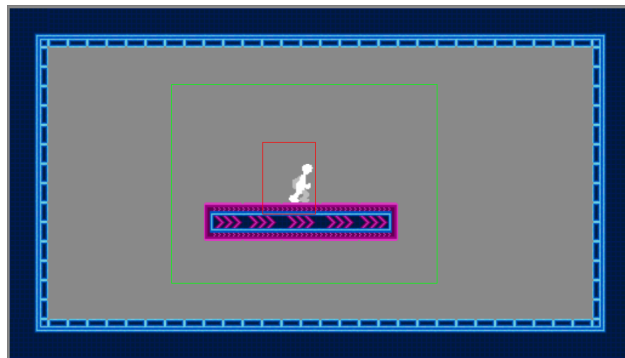


Figure 95: Example of camera trap-based system

This allows the camera to be less twitchy, giving a more natural sensation. Furthermore you can size the camera trap according to the type of game you are coding: slow-paced games can have a larger camera trap, allowing for the camera to rest more on the same screen, while faster paced games can have a smaller camera trap for faster reaction times.

[This section is a work in progress and it will be completed as soon as possible]

10.2.5 Look-Ahead Camera

This is a more complex camera that is implemented when the playable character moves towards a certain direction very quickly. The Look-Ahead camera is used to show more space in front of the player, giving more time to react to upcoming obstacles or enemies.

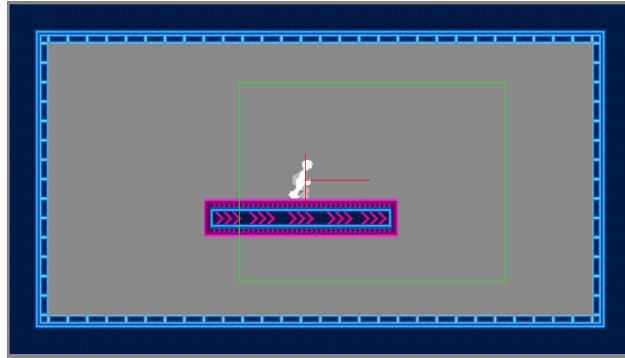


Figure 96: Example of look-ahead camera

This camera needs a good implementation when it comes to changing direction: having a sudden change of direction in the player character should have a slow panning response from the camera towards the new direction, or the game will feel nauseating.

So this camera is not ideal for games that require precision platforming, since the continuous “corrections” required to hit a tight platform would move the camera around too much, giving the player nausea.

[This section is a work in progress and it will be completed as soon as possible]

10.2.6 Hybrid Approaches

There are hybrid approaches to cameras too, mixing and matching different types of camera can give your game an additional touch of uniqueness. For instance in “Legend of Zelda: A link to the past”, the camera is a mix between a “camera trap” and a “grid camera”, where each zone is part of a grid, and inside each “grid cell” we have a tracking system based on the “camera trap”.

This allows the game to have a more dynamic feel, but also saves memory, since the SNES had to load only one “zone” at a time, instead of the whole map.

Feel free to experiment and invent!

10.3 Clamping your camera position

Whichever type of camera you decide to make use of (besides the static and grid cameras), there may be a side effect that could not be desirable: the camera tracking could follow the player so obediently that it ends up showing off-map areas.

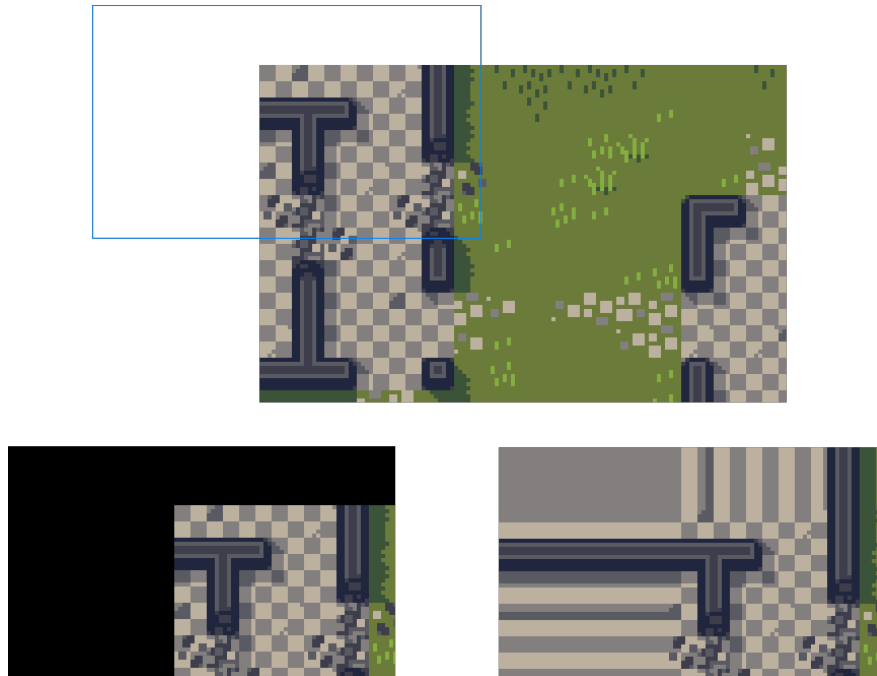


Figure 97: How the camera may end up showing off-map areas

Off-map areas may be shown as black in many cases, but in other cases (when the screen is not properly “cleared”) the off-map area can show glitchy versions of the current map.

In this case, it will be necessary to “clamp” the camera position, this way it will still follow the player, but won’t show off-map areas.

This usually just involves a check on the viewport boundaries against the map boundaries, followed by a reset of the coordinates to the map boundaries.

11 Game Design

There are three responses to a piece of design - yes, no, and WOW!
Wow is the one to aim for.

Milton Glaser

Game design is a huge topic, in this section we will just dip our toes into the argument, with some tips and tricks that can make the difference.

In this section we will also talk about level design tips, tricks and common pitfalls to avoid. We will talk about tutorials, entertaining the player and ways to reward them better.

11.1 Tutorials

11.1.1 Do not pad tutorials

Tutorials are meant to introduce the player to the game's mechanics, but a bad tutorial can ruin the experience. Tutorials should be comprehensive but also compact, padding tutorials should absolutely not be a thing.

Gloss over the simpler things (usually the ones that are common to the genre) and focus more on the unique mechanics of your game.

Avoid things like:

Use the “right arrow” button to move right, the “left arrow” button to move left, use “up arrow” to jump, use “down arrow” to crouch

Instead use:

Use the arrows to move.

And eventually present the more complex mechanics in an “obstacle course” fashion.

11.1.2 Integrate tutorials in the lore

Tutorials are better when well-integrated in the lore, for instance if your game features a high-tech suit maybe you should make a “training course” inside the structure where such suit was invented.

By integrating the tutorial into the game world, it will feel less of a tutorial for the player, but more like training for the game's protagonist.

11.1.3 Let the player explore the controls

Sometimes it's better to allow the player to explore the controls, by giving them a safe area to try: this area is usually a tutorial or a specific training area.

It can prove more effective to avoid spoon-feeding your player with all the moves, and just let them explore the core mechanics of the game by themselves, eventually assisted by an in-game manual of some sort.

So instead of doing something like (thinking about a 2D tournament fighter):

Do → ↘ ↓ + A to do a chop attack

Do → ↗ ↑ + A to do an uppercut

...

Try something like:

Do → ↗ ↑ + A to do an uppercut

Try more combination with your arrows and the attack buttons for more moves

Check the move list in the pause menu

11.2 Consolidating and refreshing the game mechanics

11.2.1 Remind the player about the mechanics they learned

There's a latin saying that goes "repetita juvant", which means "repeating does good".

A good idea is to sprinkle around different levels concepts that have been learned previously, so to remind and consolidate them. This is more effective when done shortly after learning a new mechanic.

11.2.2 Introduce new ways to use old mechanics

After a while, old mechanics tend to become stale, to rejuvenate them we can apply such mechanics to new problems. Changing their use slightly can make an old experience new again.

For instance, knowing that shooting our magic beam against something on the ceiling will make it drop (usually killing an enemy), we can make the player use such environmental interactivity to drop a suspended weight to open a door, or shoot a bell to "force" a change of guard so to sneak stealthily.

11.3 Rewarding the player

11.3.1 Reward the player for their "lateral thinking"

A good idea could be rewarding the player for not throwing themselves "head first" into the fight, but instead thinking out of the box and avoid the fight altogether, or just win it differently.

Putting a very powerful enemy in front of some treasure (for instance currency used in-game) can seem unfair, unless you place an unstable stalactite that can be shot with your magic beam.

Your magic beam won't deal enough damage to the enemy to kill it before such enemy takes your life, but a stalactite on their head will do the trick, and the reward for such lateral thinking will be a heap of coins (or gems, or whatever currency you invented).



Figure 98: Example of how to induce lateral thinking with environmental damage ^{2 3 4}

Giving tips to the player by breaking the fourth wall can be another idea, a rock or a patch of dead grass conveniently shaped like an arrow could point towards a secret room that has a fake wall.

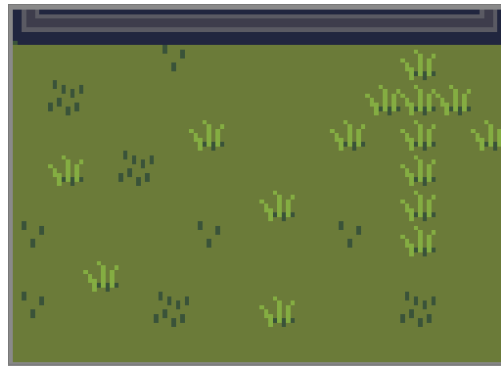


Figure 99: Example of how to induce lateral thinking by “breaking the fourth wall” ⁵

This last tip should be done very subtly, so not to ruin the immersion. Unless your game takes advantage from these kind of things (for instance games based on comedy).

11.3.2 Reward the player for their tenacity

After suggesting to reward players for not butting head-first into fights, now I’m going to suggest the exact opposite (in a way): reward your players for their tenacity.

Beating a tough boss with a certain (weak) weapon, or just the plain tenacity and skill that is needed to

²32x32 Chests attribution: Bonsaiheldin (<http://nora.la>), hosted at [opengameart](http://opengameart.org)

³Simple SVG ice platformer tiles, listed as “Public Domain (CC0)” at [OpenGameArt.org](http://opengameart.org)

⁴Fossil (Undead) RPG Enemy Sprites attribution: Stephen Challener (Redshrike), hosted by [OpenGameArt.org](http://opengameart.org)

⁵Jawbreaker tileset, listed as public domain at <https://adamatomic.itch.io/jawbreaker>

undertake a hard task, such feats should be rewarded: for instance with a powerful weapon that can be used after some level-ups.

11.3.3 Reward the player for exploring

Exploration can lead the player to discover secrets, which can range from simple gear, to pieces of unexplored environment, or even pieces of the game's lore.

World exploration should not be limited to simple secrets, a nice idea could be finding a path towards something that is usually considered “environmental damage” (like a catapult in the background) so that the player can deactivate it.

Thinking out of the box can lead to some really interesting results when it comes to this tip.

11.3.4 Reward the player for not immediately following the given direction

This is an extension of the previous point, the player should be rewarded for their exploratory efforts, even more when those efforts mean not immediately following the direction given by the designer.

“Thinking differently” should be rewarded with challenge and rewards up to said challenge. If the mission tells a player to climb up a tower, the more curious players could be led to hit the tower's underground dungeon before going on with the mission. A nice challenge in such dungeon with a fitting reward could expand on the game experience.

11.3.5 Reward the player for not trusting you entirely

Sometimes it can be fun, for both the game designer and the player, to play a bit of a trick to the player themselves.

Some famous games, like DOOM and Dark Souls, use secrets-in-secrets to trick players into thinking they found something valuable, while hiding something way more important. Let's see the example below.

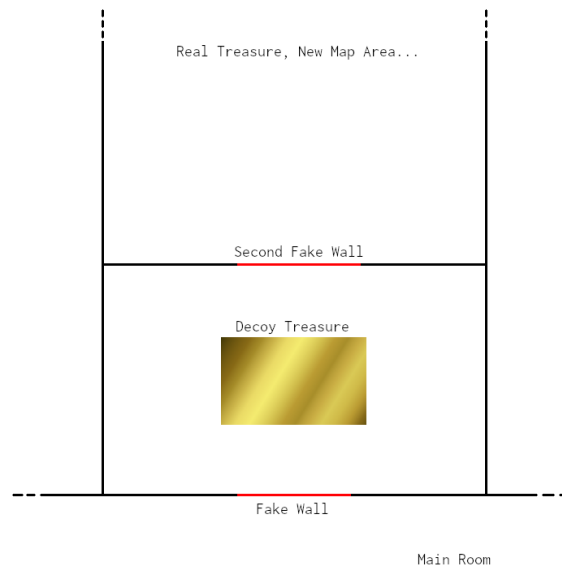


Figure 100: Example of secret-in-secret

We can see how we hid a secret inside of another secret and used a piece of valuable (but not *too valuable*)

treasure to make the player think they found the secret, while the real secret is hiding behind another fake wall.

[This section is a work in progress and it will be completed as soon as possible]

11.4 Tips and Tricks

11.4.1 General Purpose

11.4.1.1 Make that last Health Point count

Players love that rush of adrenaline they get when they escape a difficult situation with just one health point. That “just barely survived” situation can be “helped” by the game itself: some programmers decide to program the last HP in a special way.

Some prefer giving the last health point a value that is higher than the other health points (kind of like a “hidden health reserve”), others instead prefer giving a brief period of invincibility when that last “1HP” threshold is hit.

These small devices allow you to give players more of those “near death” experiences that can give players that confidence boost to keep them playing through a hard stage, while at the same time, reducing the chance that they will rage-quit.

11.4.1.2 Avoiding a decision can be a decision itself

An interesting way to make the characters from a game seem more real, is registering the “lack of response” or “lack of action” in the game’s AI or dialogue tree.

This means that “ignoring” has consequences, and inaction is in and itself an action of “doing nothing” which should be accounted for, just like ignoring someone in real life can have serious consequence or where someone may prefer to do nothing instead of taking one of many bad decisions.

This trick is used in the game “Firewatch”, where not responding to a dialogue prompt is a noted decision.

11.4.1.3 Telegraphing

Players hate the feeling of injustice that pops out when a boss pulls out a surprise attack, that’s why in many games where precise defense movement is required bosses give out signals on the nature of their attack.

This “telegraphing” technique, allows for that “impending danger” feel, while still giving the player the opportunity to take action to counteract such attack.

Telegraphing is a nice way to suggest the player how to avoid screen-filling attacks (which would give the highest amount of “impending danger”).

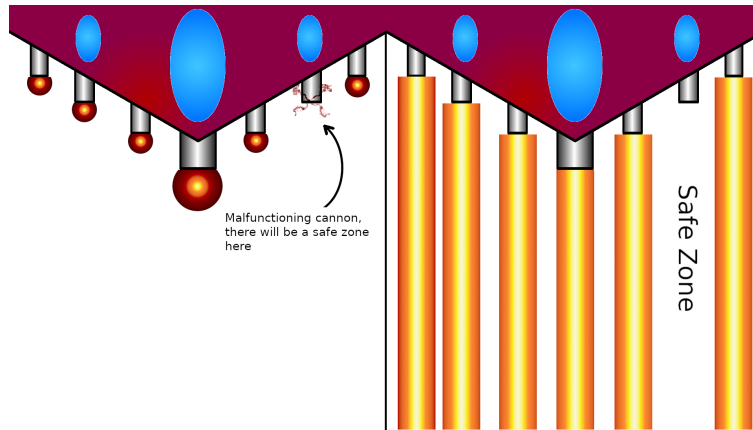


Figure 101: Example of a telegraphed screen-filling attack in a shooter

Another example of telegraphing was used in the Bioshock series: the first shots of an enemy against you always miss, that is used to avoid “out of the blue” situation, which can be seen as a form of “telegraphing” both the presence and position of enemies.

11.4.2 Shooters

11.4.2.1 Make the bullets stand out

One of the most annoying things that can happen when you’re running-and-gunning your way through a level is being hit by a not-so-visible bullet.

Your own bullets, as well as (and most importantly!) the enemies’ should stand out from the background and the other sprites, so that the player can see and avoid them.

Some people may want to ask why your own bullets should stand out too, the answer is: so you can easily aim for your targets.

12 Creating your resources

Art is not what you see, but what you make others see.

Edgar Degas

12.1 Graphics

12.1.1 Some computer graphics basics

[This section is a work in progress and it will be completed as soon as possible]

12.1.1.1 Color Depth

[This section is a work in progress and it will be completed as soon as possible]

12.1.1.2 True Color vs. Indexed Color

There are two main ways to represent color in digital images.

The first is “True Color”, which allows 256 values (from 0 to 255 included) for each color channel (red, green and blue), for a total of over 16 Million colors.

Each single color is identified by its value, which can be a waste of space and memory when the image has few well-defined colors.

The second way to store images is with “indexed color”: in this case a “palette” of colors is created and each pixel color refers to an index to such palette. This allows for smaller images, at the expense of the number of available colors. If you want to add a new color to the picture, first you need to add it to your palette if there is space.

12.1.1.3 Lossless Formats

There are a few ways to store information on a computer, either you store them raw, or you use some tricks to make such information occupy less space in your drive.

When it comes to storing information, lossless formats are usually uncompressed or make use of clever tricks to compress your images without losing information in the process.

In computer graphics, lossless formats include:

- JPEG 2000 (in its lossless form);
- Portable Network Graphics (PNG);
- Free Lossless Image Format (FLIF);
- PDF (in its lossless form);
- TARGA Files (Truevision TGA, with TGA file format);
- TIFF (in its lossless form).

12.1.1.4 Lossy Formats

When it comes to compressing information, the best way to store the least amount of information possible is to actually *not store them*. Lossy file formats get rid of some information present in the picture, in a more or less evident way (depending on the compression ratio) to lower the file size.

In computer graphics, lossy file formats include:

- JPEG (in its most common “lossy” form).

12.1.1.5 Transparency

Usually you need to have transparency in your artwork, for instance for your sprites. There are different ways to get transparency in your artwork, depending on the image format you're using and the support offered by the engine/framework you're using.

12.1.1.5.1 Alpha Transparency

This is the most common type of transparency available today: along with the usual Red-Green-Blue (RGB) channels, the image has an additional "Alpha" channel. Sometimes images with "Alpha Transparency" are also referred as "RGBA" images.

This allows to set the transparency precisely and allows for "partial transparency" too, which means that we are able to create shadows and semi-transparent surfaces.

The PNG format is one of the many image formats that supports alpha transparency.

12.1.1.5.2 Indexed Transparency

Normally used in GIF images, "Indexed Transparency" is the product of some limitation imposed in the format itself: you can only choose from a limited palette of colours to paint your picture.

If you want to have transparency in your picture, you will need to sacrifice a color and tell the format that such color is the "transparency color". In many images a very bright, evident color (like magenta) is used. Such color will not be painted, thus giving the transparency effect.

This also means that we cannot make semi-transparent surfaces, since only that specific color will be fully transparent, and that's it.

12.1.2 General Tips

In this section we will take a look at some basic art tips that will give you some indication on how to create your own art for your very own game. This will be pointers to keep you going.

12.1.2.1 Practice, Practice, Practice...

As with all forms of art, the secret to "getting good" is practice. There's no way to avoid it, your first piece may be nice or flat out terrible, your next one will be better, and the one after that will be even better... Hard work always beats "talent" in the long run.

12.1.2.2 References are your friends

References are a tricky topic, some artists swear by it, others oppose them fiercely.

In my opinion, looking at a real-life version of what you want to draw can be one of the most useful things you can do when drawing. Even when drawing something that involved a huge amount of fantasy, having a reference can give you indications on shapes and sizes.

It doesn't have to be a one-to-one reference either, you can get ideas for your dragon from crocodiles and lizards, or even snakes!

12.1.2.3 Don't compare your style to others'

One of the most frustrating things that can happen when learning something new, is comparing yourself to another artist and saying "I should be able to draw like them".

Everyone has their own unique style, and you should work on what makes it unique, instead of comparing your style to others.

12.1.2.4 Study other styles

This ties a bit to the previous point, you should not compare to others, but you should also take some time to look at other people's work, find what you like about it and implement it into your own art style.

Looking around you can help you grow as an artist and aid you in the difficult seeking of your own art style.

12.1.2.5 Learn to deconstruct objects into shapes

Every complex object can be deconstructed into simpler shapes: circles, ellipses, triangles, rectangles, squares...

You can use such simple shapes, overlapping them to create a skeleton of your subject (living or not), so that you can draw it in an easier way: with the layer system introduced by huge part of the current drawing applications, you don't have to do precision work when it comes to removing such skeleton.

12.1.3 Sprite sheets

Every time we create a sprite, we need some amount of memory to store its information, and to match the hardware constraints most of the time a sprite's image must be padded with unused pixels.

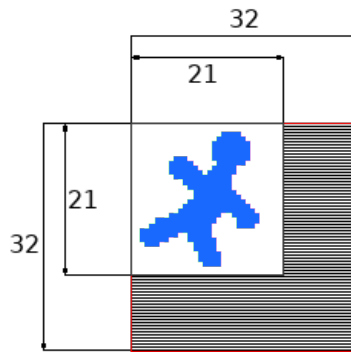


Figure 102: Example sprite that gets padded to match hardware constraints

Each sprite image that gets stored (and there could be potentially hundreds) wastes more and more memory, so we need a way to store sprites more efficiently. In the previous example we can see how a 21x21 sprite gets padded towards a 32x32 size, the sprite uses 52% more memory than it should!

Enter the humble Sprite Sheets.

We save our sprites (as well as animation frames) into a single drawing, called a “sprite sheet”. By composing a sprite sheet with several smaller images of the same size, we just need to adapt our rendering to draw a portion of such sprite sheet on our screen. The sprite sheet is the only thing that will need to be adapted to match our hardware constraints, saving memory.

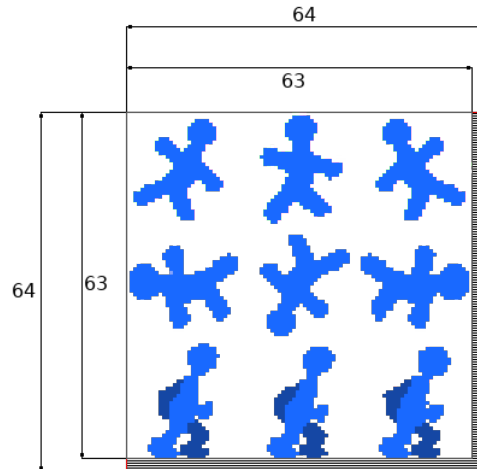


Figure 103: Example spritesheet that gets padded to match hardware constraints

In the previous example, the sprite sheet occupies only 1.5% more memory than it should. That's a great improvement

This way, instead of having a lot of references to sprites to draw, each one wasting its own memory, we just need the reference to the sprite sheet and a list of coordinates (rectangles, most probably) to draw.

Libraries like OpenGL support “sprite atlases” (or sprite batches), allowing for the graphics card to take care of drawing (after preparing the batch) while the CPU can use more of its cycles to take care of input, movement and collisions.

[This section is a work in progress and it will be completed as soon as possible]

12.1.4 Virtual Resolution

There are times where having the crispest graphics around is not a good idea, for instance in Pixel-Art games.

Times where it's a better idea to keep your game low-resolution, be it for a matter of performance (your first games won't be extremely optimized and will be slow even at low map sizes and resolutions) or to keep your pixel-art crisp and “pixelly”.

This clashes with the continuous push towards higher resolutions, which can mess up your game in a variety of ways, like the following ones.

If the game is forced in windowed mode, you'll have a problem like the following:

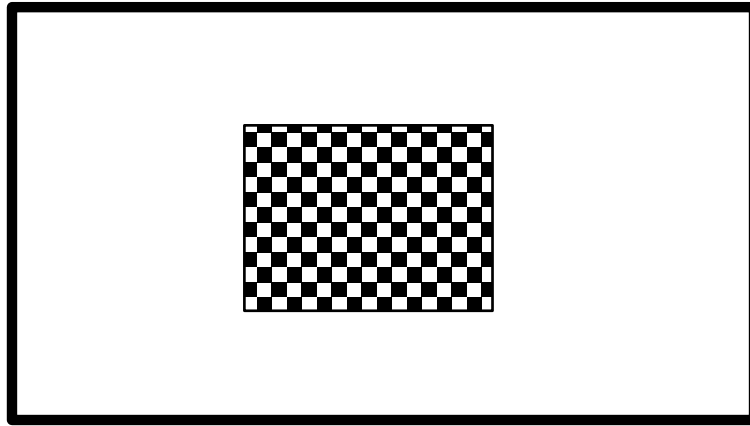


Figure 104: Windowed Game Example - A 640x480 game in a 1920x1080 Window

The game window is way too small and hard to see, and can get even smaller if the HUD_g takes even more space out of the window. This can be mitigated by calculating the position of each element in comparison to the window size, this although can result in items too small (or too big if downscaling), like the following HUD example.

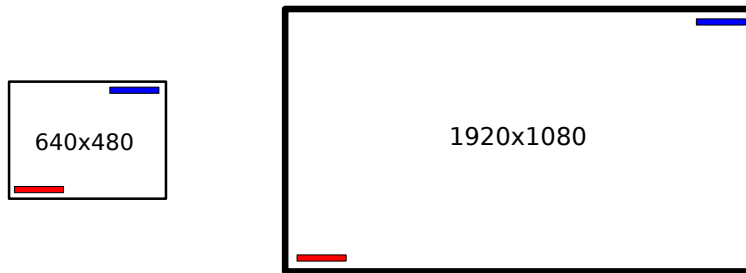


Figure 105: Fullscreen Game Example - Recalculating items positions according to the window size

This is where **virtual resolution** comes into play, the frame is rendered in a virtual surface which operates at a “virtual resolution” and the whole frame is drawn and upscaled (or downscaled) to the screen, without having to recalculate anything in real time (thus making the game faster and lighter).

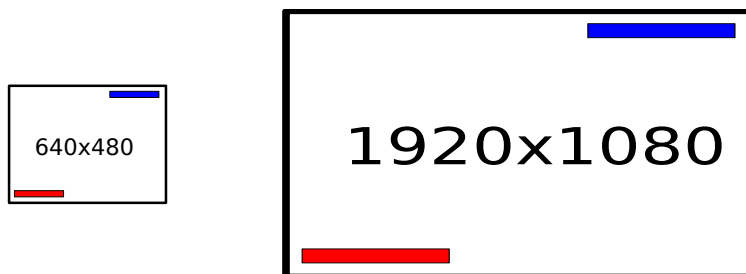


Figure 106: Fullscreen Game Example - Virtual Resolution

The items get scaled accordingly and there is no real need to do heavy calculations. Virtual Resolution allows for different kinds of upscaling, with or without filtering or interpolation, for instance:

- **No filtering** - Useful for keeping the “pixeliness” of your graphics, for instance in pixel-art-based games. It’s fast.;
- **Linear, Bilinear, Trilinear Filtering** - Gives a more soft look, slower;
- **Anisotropic Filtering** - Used in modern 3D games, highest quality but also among the slowest, it is usually used when rendering sloped surfaces (from the player’s point of view).

12.1.5 Using limited color palettes

[This section is a work in progress and it will be completed as soon as possible]

12.1.6 Dithering

Dithering (usually called “Color Quantization”) is a technique used to give the illusion of a higher “color depth” when you’re using a limited palette of colors.

The usage of dithering introduces patterns into the image when the pixels are visible, if instead the pixels are small enough the pattern will look like a new color, without actually introducing a new color into the palette.

Using two different levels of blue, we can use a dithering pattern to obtain a new tone of blue, like the following:

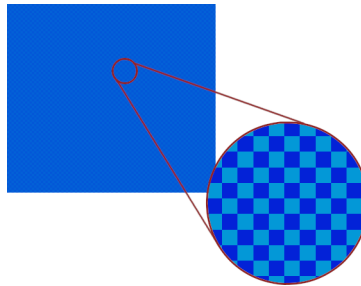


Figure 107: Dithering example

It’s possible to study how your palette reacts to dithering using a dithering table, this will give you an idea of the colors available via dithering.

You can see a simple example of a dithering table here:

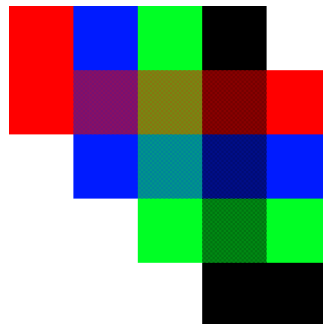


Figure 108: Dithering Table Example

You can see the palette colors on the top row and the left column, then you can see how dithering (in this case a simple checkerboard pattern) allows for different colors to pop out.

There are different dithering patterns, that allow for different type of colors, intensity and patterns:

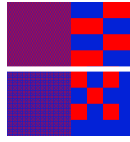


Figure 109: Some more dithering examples

12.1.7 Layering and graphics

There are some things that should be kept in mind when drawing layers for your game, here we talk about some key points about layering and graphics.

12.1.7.1 Detail attracts attention

When it comes to games, it's easy to get too excited and craft your work with the highest amount of detail possible, but there is a problem: detail tends to attract players' attention.

If you put too much detail in the background, you're going to distract them from the main gameplay that happens in the foreground, which can prove dangerous: the graphics can get messy and you can even get to the point of not being able to distinguish platforms from the background.

So a golden rule could be:

■ Use high detail in the foreground, gameplay-heavy elements - use less detail in the backgrounds

A good idea to make the background a bit less detailed is using blurring, which allows to keep the overall aesthetic but makes it look “less interesting” than what's in the foreground.

This doesn't mean the background should be devoid of detail, just don't overdo it.

12.1.7.2 Use saturation to separate layers further

Bright colors attract attention as much as detail does, so a good idea is making each background layer more “muted” (color-wise) than the ones in foreground.

The main technique to make backgrounds more mutes is lowering saturation, blending the colors with grey: this will make the background less distracting to the player.

So another rule can be written as:

■ Layers farther away should have lower color saturation than the layers closer to the camera

12.1.7.3 Movement is yet another distraction

As detail and saturation are due to attract attention from the player, movement is another one of those “eye-catchers” that can make the gameplay more chaotic and difficult for the player.

Small amounts of movement are ok, but fully-fledged animations in the background will prove distracting.

Let's take note of rule number 3 then:

■ Try to avoid movement in the background

12.1.7.4 Use contrast to your advantage

Contrasting (complementary) color pairs were used in impressionism for their “eye-catching” character, they are created starting from the 3 primary colors (in screens: Red, Green, Blue), choosing one and combining the other two in a “secondary color”.

Some complementary color pairs are:

- **Red and Cyan:** Choose red, then green+blue gives cyan
- **Green and Magenta:** Choose green, then red+blue gives magenta
- **Blue and Yellow:** Choose blue, then red+green gives yellow

*Remember that we’re talking about the RGB model of colors produced by **light**, not the traditional color wheel*

These colors tend to attract a lot of attention in the points of intersection of their hues, distracting the player from the main gameplay.

Our rule number four should then be:

■ Keep backgrounds low-contrast to avoid distracting players

12.1.7.5 Find exceptions

Nothing is ever set in stone, and no rules should keep you from playing a bit outside the box and fiddling with some exceptions to a couple rules.

12.1.8 Palette Swapping

Palette swapping is a technique mostly used in older videogames, where an already-existing graphic (like a player character’s sprite) is reused with a different palette (combination of colors).

This palette swap makes the new graphic quite recognizably distinct from the original graphic. This technique was normally used to tell apart first and second player.

A prime example of this is the videogame that (re)started it all: Super Mario Bros. Mario and Luigi are exactly the same graphic, but Luigi uses a different palette.

Some other videogames use palette swapping to indicate their status (like using a green or purple-based palette to indicate the “poisoned” status), or indicate a difference in their statistics (like a red-based palette to indicate an enhanced attack statistic), in other occasions different palettes are used to distinguish stronger versions of the same enemy.

Other franchises, like Pokémon, use palette swaps to introduce “special versions” of some entity (in the case of Pokémon, a shiny pokemon).

Palette Swapping can be used in more creative ways, though. Going back to Super Mario Bros. you can see that the clouds and the bushes in the levels are exactly the same graphic, just with a different palette. Same goes for the underground bricks and the overworld bricks: they just have a different color.

12.1.9 Pixel Art

[This section is a work in progress and it will be completed as soon as possible]

12.1.9.1 What pixel art is and what it is not

[This section is a work in progress and it will be completed as soon as possible]

12.1.9.2 Sub-pixel animation

[This section is a work in progress and it will be completed as soon as possible]

12.1.10 Tips and Tricks

This section contains various tips and tricks used by artists to create certain effects inside video games, demonstrating how sometimes something really simple can have a great effect on the game experience.

12.1.10.1 Creating “Inside rooms” tilesets

In many cases, when dealing with tile-based games, we need to create a tileset that is good to represent “inside” environments, like a basement, a cave or the inside of a building. A simple way to reach that goal is creating a set of black and transparent tiles that can be overlaid on another tileset, like the following:

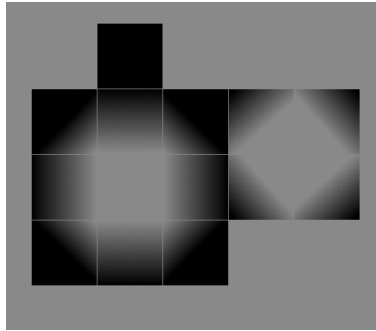


Figure 110: Example of black and transparent tileset used in “inside rooms”

Such tiles can then be overlaid onto something like the following:

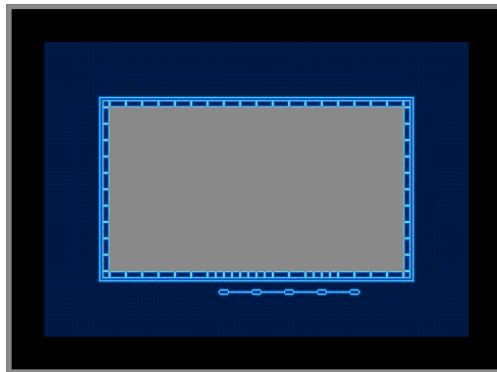


Figure 111: Example of incomplete “inside room”

And we obtain the following result:

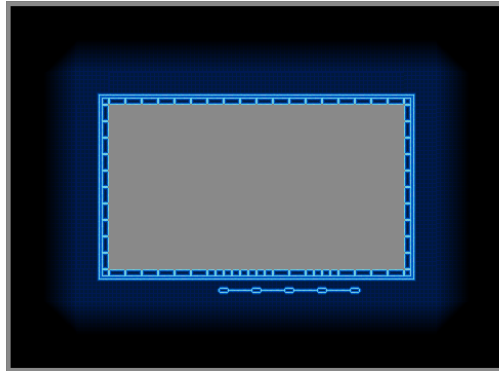


Figure 112: Example of “inside room” with the black/transparent overlay

[This section is a work in progress and it will be completed as soon as possible]

12.2 Sounds And Music

[This section is a work in progress and it will be completed as soon as possible]

12.2.1 Some audio basics

[This section is a work in progress and it will be completed as soon as possible]

12.2.1.1 Sample Rate

Differently from Analog Audio, which is continuous (as in has an infinite amount of detail), Digital Audio is a stream of numbers (ones and zeros) that is “discrete” in nature. That means that we blast these numbers thousands of times a second to be able to build a decent sounding sound.

The number of times we record such numbers from our digital microphone (as well as the number of times we blast such numbers back from our speakers) is called **sample rate** and it is measured in Hz .

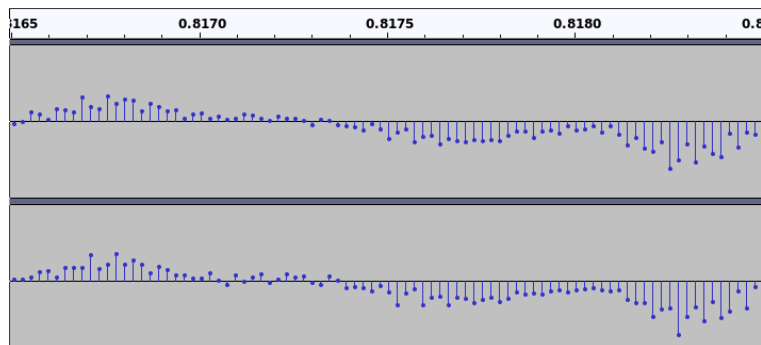


Figure 113: Graphical Representation of Sample Rate (44.1KHz)

In normal CD-Audio, we have a sample rate of 44100 Hz, which means that we recorded a sample 44100 times in a single second.

When making our game’s audio, we should always stay around such value, since going lower would make the audio sound worse, since we lower the amount of information the audio itself has.

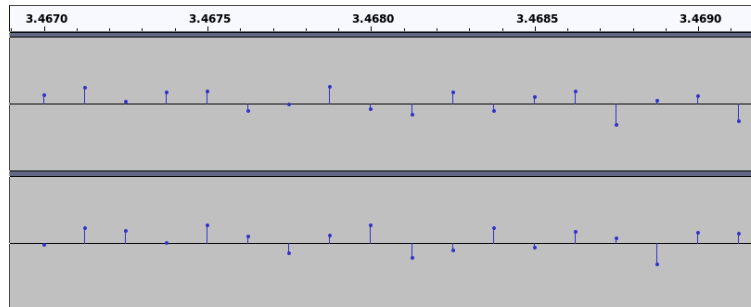


Figure 114: Graphical Representation of Sample Rate (8KHz)

Also we should avoid using weird sample rates, 44.1KHz (or 44100 Hz if you prefer) is a “magic value” that guarantees the most compatibility.

12.2.1.2 Bit Depth

Along with sample rate, there is another value in audio that expresses its quality: the bit depth.

The bit depth defines how many “volume values” we can have in a single sample, which shapes the quality of the sound in a different way than sample rate.

If our audio has a 1-bit sample rate, each sample will have only 2 values:

- **0:** Mute
- **1:** Blast at full volume

Which reduces the quality of the audio.

Usually audio has a 16 Bit depth, but more modern systems make use of 24 Bits or even 32 Bits.

12.2.1.3 Lossless Formats

As with graphics, there are audio formats that allow you to store uncompressed information, as well as compressed (but without losses) sounds. The most common lossless audio formats include:

- WAV;
- AIFF;
- FLAC.

12.2.1.4 Lossy Formats

As with graphics, there are also “lossy formats” that allow us to store information in even less space by getting rid of information that is well outside our hearing spectrum, for instance. Some of the most known are:

- Mpeg Layer 3 (MP3);
- OGG Vorbis;
- Windows Media Audio (WMA);
- Advanced Audio Codec (AAC).

12.2.1.5 Clipping

Clipping is a phenomenon when you’re trying to output (or record) a wave that exceeds the capacity of your equipment. As a result, the wave is “clipped” (cut) and there is a very audible distortion.

You surely heard clipping in audio before, usually when people scream on a low-quality microphone and the audio gets distorted.

The best way to repair clipping is to re-record the audio completely, although some tools can help in case you absolutely cannot re-record the audio.

Also you should be wary of clipping, because there may be cases where it damages your audio equipment.

12.2.2 Digital Sound Processing (DSP)

Let's think about a simple situation: we want to play a "walk" sound effect: every time our character's foot hits the ground, we play a "step" sound effect.

If we play the same sound over and over, it will become boring really quickly, breaking the immersion. An idea could be saving different sounds for a footstep and then every time the player takes a step, a random footstep sound will be played.

This solves the problem, at a cost: we need to use more memory to keep track of such sounds, multiply that for tens of sound effects and the game can easily run out of memory.

An alternative solution could be using DSP: editing the sound sample in real time to add more variety and depth while saving memory, the trade-off would be CPU time, but it's an acceptable deal.

12.2.2.1 Reverb

When you take a stroll on a sidewalk, you have a certain "openness" on the footstep sounds you hear, but that surely changes if you're walking with hard shoes on a hard floor inside of a small cave. You can hear a lot of reverb and echo at every single step.

Reverb is the first of the sound effects that we encounter in our journey: it allows to give more depth to our sound effects, making it sound like we're inside of a small cave or a very large room.

12.2.2.2 Pitch Shift

A way to give more variety to a sound effect without much work is using pitch shift to make our sound a bit higher or lower, randomly, so that each step is slightly different from the other: this way our ears will get less tired of hearing said sound effect.

Pitch shifting must be used with caution, since abusing it will distort the sound effect and break the immersion in our game.

Another example of pitch shift is used in racing games, where the car roar is pitch-shifted up or down according to the acceleration given to the car.

12.2.2.3 Filtering

Another sound effect we can use is filtering, which are divided in 3 main sections, according to the frequency they "allow to pass through":

- **Low Pass Filter:** This filter allows low frequencies to pass through unfiltered, while the frequencies higher than a defined threshold will be cut. This allows for effects where the bass is unaltered but higher frequencies are cut away;
- **High Pass Filter:** Opposite of the previous filter, this filter allows high frequencies to pass through unaltered while the frequencies lower than a defined threshold will be cut;
- **Band Pass Filter:** A combination of the two previous filters, this filter let's through all the frequencies between two defined threshold values. This allows for more interesting effects like a music sounding through an old radio.

An interesting example is when an explosion happens near the player, in that case the "stun" effect is given by using a low pass filter on an explosion sound (which makes it sound really low and muffled), eventually a

slowdown is applied and a secondary sound effect of a very high pitch sound is added (something similar to what you hear when your ears are ringing).

12.2.2.4 Doppler Effect

To give more depth to your sound effects, you can use pitch shift to create the “Doppler Effect” that we hear in real life when a police car passes by: when the car approaches us the pitch is higher, when the car is in front of us we hear the siren as it should be, and when the car passes us we hear a lower pitched version of the siren sound effect.

The Doppler effect can be really useful when applied to car racing games again, when we overtake one of our opponents (or one of our opponents overtakes us) using the doppler effect can help the player feel more “immersed” in the experience.

The doppler effect would actually apply to light too, but we would need to have something travelling at a really high speed or said object to be really far away (like a planet).

12.2.3 “Swappable” sound effects

Back to our walking example, an idea to increase the variety of sound effects at our disposal would be keeping a list of “swappable sounds”: sounds that are still part of the class we’re considering, but are radically different.

For instance we could have different walking sounds for different floors, so that walking on grass and walking on a stone pavement will be different. In this case it would be useful to make the sounds configurable and give the sound manager the chance to inspect what type of floor we’re walking on.

An example of “swappable sound effects” configuration is given in the following file, which is written in YAML:

Listing 37: Example of swappable sound effects

```
footsteps:
2  grass:
    - grasswalk1.wav
4    - grasswalk2.wav
    stone:
6    - stonewalk1.wav
    - stonewalk2.wav
8  metal:
    - metalstep1.wav
10 - metalstep2.wav
```

Making a configuration file instead of hard-coding the elements allows for easy extensibility and modding, which everyone loves (See [Designing entities as data](#)).

12.2.4 Some audio processing tips

[This section is a work in progress and it will be completed as soon as possible]

12.2.4.1 Prefer cutting over boosting

[This section is a work in progress and it will be completed as soon as possible]

12.3 Fonts

12.3.1 Font Categories

Before starting with fonts and the ways they can be integrated in your game, we should start with some definitions and categorizing fonts by their own characteristics.

12.3.1.1 Serif and Sans-Serif fonts

In typography, *serifs* are small strokes that get attached to larger strokes in a letter (or symbol) of certain fonts. The font families that make use of serifs are called **serif fonts** or **serif typefaces**.

Serif fonts look more elegant and give a “roman” feeling (in fact, serif fonts are also called *roman* typefaces) and are good for games that take place in historical settings or need a semblance of pretend “historical importance” (in their own world).

DejaVu Serif

Figure 115: Example of a serif font (DejaVu Serif)

Serif fonts look better on paper and could come out as a bit harder to read on screens. A famous serif font is Times New Roman.

On the opposite side, we have **sans-serif fonts**, where such small strokes are absent. Sans-Serif fonts seem easier to read on screens and look simpler, but they don’t look as good on paper, when long text bodies are involved.

DejaVu Sans

Figure 116: Example of a sans-serif font (DejaVu Sans)

A famous sans-serif font is Arial.

12.3.2 Proportional and Monospaced fonts

The majority of fonts used today are **proportional**, where each letter occupies its own space proportional to its own width. Examples of proportional fonts are Times New Roman and Arial.



Figure 117: Example of a proportional font (DejaVu Serif)

Notice the difference in width between certain pairs of letters, like “i” and “o” or “a” and “l”.

Proportional fonts are good for general text that don’t have any particular constraint.

On the opposite side, there are **monospaced** fonts, also called **fixed-width** fonts. In these font families, each letter occupies the same amount pre-defined width.



Figure 118: Example of a monospaced font (Inconsolata)

Again, notice how all letters occupy the same horizontal space.

Monospaced fonts are used for computer texts, coding and ascii-art. Examples of monospaced fonts are Courier New and Inconsolata.

12.3.3 Using textures to make text

[This section is a work in progress and it will be completed as soon as possible]

12.3.4 Using Fonts to make text

[This section is a work in progress and it will be completed as soon as possible]

12.4 Shaders

12.4.1 What are shaders

Shaders are technically small programs that (usually) run inside your Graphics Card (GPU). In gaming they are usually used for post-processing and special effect, allowing to free the CPU from a lot of workload, using the specialized GPU capabilities.

Shaders can be classified in different groups:

- **2D Shaders:** These shaders act on textures and modify the attributes of pixels.
 - **Pixel (Fragment) Shaders:** Used to compute color and other attributes relating to a single output pixel. They can be used for blur, edge detection or enhancement and cel/cartoon shading.
- **3D Shaders:** These shaders act on 3D models or other geometry.
 - **Vertex Shaders:** These shaders are run once per vertex given to the GPU, converting the 3D position in virtual space to the 2D coordinates of your screen. These shaders cannot create any new geometry.
 - **Geometry Shaders:** These shaders can create new primitives, like points or triangles.
 - **Tessellation Shaders:** These shaders allow to divide simple meshes into finer ones at runtime, this allows the meshes closest to the camera to have finer details, while the further ones will be less detailed.
 - **Primitive Shaders:** Akin to the computing shaders, but have access to data to process geometry.
- **Computing Shaders:** These shaders are not limited to graphics, but are related to the world of GPGPU (General Purpose computing on GPU), these can be used to further stages in animation or lighting.

12.4.2 Shader Programming Languages

There are numerous programming languages, depending on the platform and libraries you are using to program your game.

If you are using OpenGL, you should use the official OpenGL Shading Language, called **GLSL**.

If you are using Direct3D, you should instead use the “High Level Shader Language”, also called **HLSL**.

If instead you want to use Vulkan, you will need to use the **SPIR-V** (Standard Portable Intermediate Representation) format, but the good news is that (at the time of writing) you can convert your GLSL code into SPIR-V and use it with Vulkan.

Modern engines, like Unity and Unreal Engine also include GUI node-based editors that help you create new shaders by using directed graphs, without using any code.

12.4.3 The GLSL Programming Language

[This section is a work in progress and it will be completed as soon as possible]

12.4.4 Some GLSL Shaders examples

[This section is a work in progress and it will be completed as soon as possible]

13 Procedural Content Generation

Science is what we understand well enough to explain to a computer.
Art is everything else we do.

Donald Knuth

13.1 What is procedural generation (and what it isn't)

Sometimes you hear “procedural generation” being thrown around as a term describing that some part of a videogame is generated with an element of randomness to it.

This isn't entirely true, since “procedural generation” suggests the presence of a “procedure to generate the item”, in short: an algorithm.

A procedurally generated weapon is not statically created by an artist, but instead by an algorithm that puts together its characteristics. If the algorithm has the same data in its input, then the same item will be generated as an output.

When you introduce an element of randomness (or more precisely **pseudo-randomness**) you have what is called “random generation”.

Let's make a simple example: we want our Super-duper-shooter to make use of procedural/random generation to create your weapons. The following example will clarify the difference in algorithms between procedural and random generation, all weapons have a body, a scope, a barrel and an ammo magazine.

This is a possible algorithm for a procedural weapon:

Listing 38: Example procedural weapon creation

```
function createProceduralWeapon():
2   load body "body0001.png"
   load scope "scope0051.png"
4   load barrel "barrel0045.png"
   load ammo magazine "mag0009.png"
6   put together the loaded pieces
   set weapon damage to 45
8   set weapon range to 15
   set weapon spread to 23
10  return generated weapon
```

This instead is a possible algorithm for a random weapon, for simplicity we assume that the pieces are all compatible:

Listing 39: Example Randomized weapon creation

```
function createRandomizedWeapon():
2   load a random body from the folder "weaponBodies/shotguns"
   load a random scope from the folder "weaponScopes/shotguns"
4   load a random barrel from the folder "weaponBarrels/shotguns"
   load a random ammo magazine from the folder "weaponMagazines/shotguns"
6   put together the loaded pieces
   set weapon damage to a value between 35 and 50
8   set weapon range to a value between 13 and 18
   set weapon spread to a value between 20 and 30
10  return generated weapon
```

As you can see, the algorithms are very similar to each other, but the second one has an element of randomness added to it.

So, as a memorandum:

Procedural generation is **consistent**, even though something is generated in real time, given the same input the same output will be returned.

Random generation is usually **not consistent**, although it is possible to control the random generator (via its seed) to obtain deterministic results, given the same input.

Seeding a random number generator correctly can allow you to generate a huge universe without storing it into memory, for instance; although the edits to such universe will have to be saved in some other way.

13.2 Advantages and disadvantages

As with everything, procedural and random generation has its advantages and disadvantages, which will be explained below.

13.2.1 Advantages

13.2.1.1 Less disk space needed

Using algorithms to build worlds and items means generating them mostly in real-time, which means we don't have to save them to hard-disk, since if the algorithm is not randomized, you can always re-create the same worlds and items when requested. This was more pressing at the times of the NES, where game sizes were usually around a couple hundreds of KB.

13.2.1.2 Larger games can be created with less effort

When a world is handcrafted, everything has to be placed and textured manually, which takes time and money. This obviously puts a superior limit to how big these worlds can be.

When procedural (and randomized) generation comes into play, there is no theoretical limit to how big these worlds can be (considering an infinitely powerful hardware).

Same goes for items, each handcrafted item takes time and money, while using procedural generation you can re-use components of said items to generate a potentially infinite number of new items that have certain characteristics.

13.2.1.3 Lower budgets needed

Creating a videogame is expensive, in fact the so-called “AAA” games costs are in the order of millions of dollars. Using procedural and random generation you can create variations of your resources (textures, for instance), lowering costs.

13.2.1.4 More variety and replayability

When a world and its objects are handmade, the game experience is bound to be fixed: same items to collect, same world, same overall experience. Procedural and random generation can bring some sense of “unknown” to the game every time you play. This also enhances the replayability value of the game.

13.2.2 Disadvantages

13.2.2.1 Requires more powerful hardware

Procedural generation makes use of algorithms, and such algorithms can be really taxing on the computer hardware, so loading times might increase or users with less powerful computers might experience stutters as their computer cannot “keep up” with the game demands.

13.2.2.2 Less Quality Control

Computers are able to crunch numbers at an incredible rate, but they lack creativity. In a procedurally generated world you lose the “human touch” that can introduce subtleties and changes that can be brought by a good designer with experience.

At the same time, there is a variation in user experience, so you cannot guarantee the same gameplay quality to all players. Some players may find a really easy map to play in, while others might find a really hard map that prohibits such gameplay.

13.2.2.3 Worlds can feel repetitive or “lacking artistic direction”

Consequence of having less quality control, worlds and items might feel like they “lack artistry”, as well as being repetitive.

If you use procedural and randomized generation, you have the chance of generating incredibly large worlds with a huge variety of items with less resources and algorithms; that’s where our human nature of “recognizing patterns” crashes the party: repeating patterns are really easy to spot and can remove us from the game’s atmosphere and introduce us to one of our worst enemies: boredom.

13.2.2.4 You may generate something unusable

In extreme cases, there is a possibility that we end up generating an unplayable world, or useless items: terrain too high to climb, walls blocking a critically-necessary area, dungeon rooms with no exits, etc. . .

13.2.2.5 Story and set game events are harder to script

Being uncertain, procedural generation makes set events harder to script, if not impossible. In this case it’s more common to use a mix of procedural generation and pre-made game elements, where the fixed elements are used to drive the narrative and the procedurally generated elements are used to create an open world for the player to explore and vary its gameplay experience.

13.3 Where it can be used

Procedural (and random) generation can be used practically anywhere inside of a videogame, some examples could be the following:

- **World Generation:** Using an algorithm called “Perlin noise”, you can generate a so-called “noise map” that can be used to generate 3D terrain, using giving areas with higher concentration a higher height. For dungeon-crawling games you might want to use a variation of maze generation algorithms, and so on so forth;
- **Environment Population:** You can use an algorithm to position certain items in the world, and if an element of randomness is required, positioning items in a world is certainly a very easy task and can add a lot to your game, but be careful not to spawn items into walls!;
- **Item Creation:** As stated previously, you can use procedural generation to create unique and randomized items, with different “parts” or different “stats”, the possibilities are endless!;
- **Enemies and NPCs:** Even enemies and NPCs can be affected by procedural (and randomized) generation, giving every NPC a slightly different appearance, or scaling an enemy size to create a “behemoth” version of a slime, maybe by pumping its health points too, randomizing texture colors, again the possibilities are endless;
- **Textures:** It’s possible to colorize textures, giving environments different flavours, as well adding a layer of randomness to a procedurally generated texture can greatly enhance a game’s possibilities;
- **Animations:** An example of procedurally generated animations are the so-called “ragdoll physics”, where you calculate the forces impacting a certain body (and it’s “virtual skeleton”). A simpler way

could be making the program choose randomly between a set of pre-defined “jumping animations” to spice up the game;

- **Sounds:** You can use sound manipulation libraries to change the pitch of a sound, to give a bit of randomness to it, as well as using “sound spatialization” by changing the volume of a sound to make it come from a certain place, compared to the player’s position;
- **Story:** In case you want to put some missions behind a level-gate, you can use procedurally generated missions to allow the players to grind for experience and resources, so they are ready for the upcoming story checkpoints.

14 Useful Patterns, Containers and Classes

Eliminate effects between unrelated things. Design components that are self-contained, independent, and have a single, well-defined purpose.

Anonymous

In this chapter we will introduce some useful design patterns, data containers and classes that could help you solve some issues or increase your game's maintainability and flexibility.

14.1 Design Patterns

Design Patterns are essentially “pre-made solutions for known problems” and can help decoupling elements of your game, increasing maintainability.

Obviously design patterns are not a cure-all, they can introduce overhead and could lead to over-engineering: balance is key when it comes to creating a game (or any software in general).

14.1.1 Singleton Pattern

Sometimes it can be necessary to ensure that there is one and only instance of a certain object across the whole program, this is where the *singleton* design pattern comes into play.

To make a singleton, it is necessary to hide (make private) the class constructor, so that the class itself cannot be instantiated via its constructor.

After that, we need a static method that allows to get the singleton's instance, the method needs to be static to make it callable without an instance of the singleton class.

The UML diagram for a singleton is really simple.

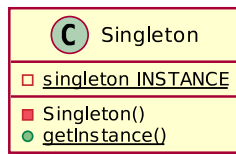


Figure 119: The UML diagram for a singleton pattern

Listing 40: Example of a singleton pattern

```
class Singleton {
2
    private static Singleton INSTANCE = new Singleton();
4
    // This makes the constructor impossible to use outside of the class
6    private Singleton() {}

8    public static Singleton getInstance() {
        return INSTANCE;
10    }
}
```

The previous singleton instantiates immediately, which may not always be necessary, in that case a good idea could be implementing the so-called “lazy loading”, where the instantiation happens the first time you ask the object for its own instance.

Listing 41: Example of a singleton pattern with lazy loading

```
class LazySingleton {  
2  
    private static LazySingleton instance = null;  
4  
    private LazySingleton() {}  
6  
    public static Singleton getInstance() {  
8        // Multi-threading: manage race conditions  
        // ----- Critical region start -----  
10        if (instance == null) {  
            instance = new Singleton();  
12        }  
        // ----- Critical region end -----  
14  
        return instance;  
16    }  
}
```

If multiple threads are involved in using a lazy-loading singleton, you may need to take care of preventing *race conditions*_g that could result in multiple instances of the singleton being created.

Many critics consider the singleton to be an “anti-pattern”, mostly because it is really overused and adds a possibly unwanted “global state” (see it as a global variable, but in an object-oriented sense) into the application.

Before applying the singleton pattern, ask yourself the following questions:

- Do I really need to **ensure** that only one instance of this object is present in the whole program?
- Would the presence of more than one instance of this object be detrimental to the functionality of this program? How?
- Does the instance of this object **need** to be accessible from everywhere in the program?

Table 27: Summary table for the Singleton Pattern

Pattern Name	Singleton
When to Use it	In all situations that strictly require one instance of an object, accessible globally.
Advantages	Allows the class to control its own instantiation, allows for easier access to the sole instance of a class.
Disadvantages	Introduces some restrictions that may be unnecessary, introduces a global state into the application.

14.1.2 Command Pattern

It may be necessary, during our software development, to abstract our functions into something that can be assigned and treated as an object.

Many programming languages now feature functions as “first class citizens”, allowing to treat functions as objects: assigning functions to variables, calling functions, lambdas, inline functions, functors, function pointers...

The command pattern allows us to abstract a function (or any executable line of code) into its own object that can be handled as such, allowing us to package a request into its own object for later use.

This pattern can be useful to code GUIs, making actions in our games that can be undone, macros, replays and much more.

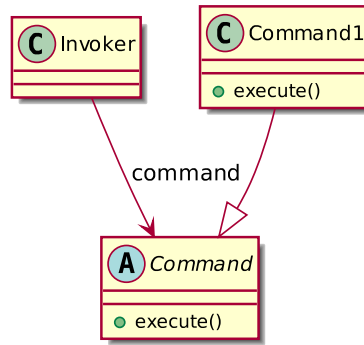


Figure 120: UML diagram for the Command Pattern

Listing 42: Example code for the Command Pattern

```

abstract class Command{
2   // This is the abstract class that will be used as interface
   public:
4       function execute();
}
6
class JumpCommand inherits from Command{
8   // This will implement the execute method
   public:
10      function execute(){
12          jump();
14      }
   private:
14      function jump(){
16          // DO STUFF
18      }
}

```

Pattern Name	Command
When to Use it	In all situations where you want to avoid coupling an invoker with a single request or when you want to configure an invoker to perform a request at runtime.
Advantages	Allows for encapsulation, less coupling, more flexibility and customization at runtime.
Disadvantages	Late binding and objects may introduce some overhead.

14.1.3 Flyweight

Sometimes it may be necessary to keep track of a large number of very similar objects.

Imagine a lot of sprites of trees that have the same texture and size, but have different positions: it could prove to be really resource-heavy to keep all the sprite objects in memory, each one with its own copy of the texture and size. This could prove to be performance-heavy too, since all those textures will have to be moved to the GPU.

Here comes the Flyweight pattern: we try to share as much of the so-called “intrinsic state” of the objects between the object that contain the so-called “extrinsic state”.

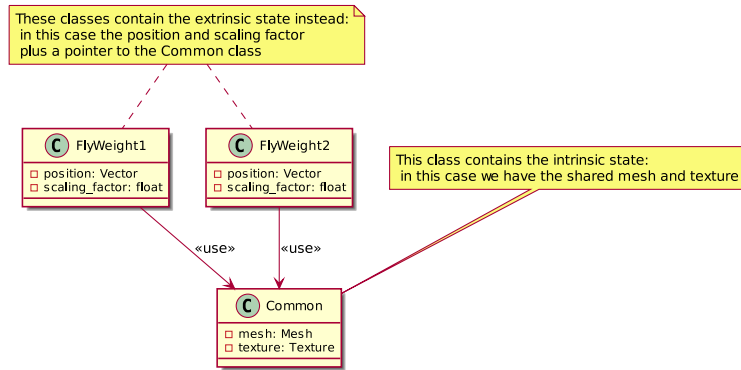


Figure 121: UML Diagram of the Flyweight pattern

Below is an example code for the flyweight pattern.

Listing 43: Code for a flyweight pattern

```

class Common{
2  // Contains the common data for a 3D Object to be replicated
  private:
4    Mesh mesh;
    Texture texture;
6 }

8 class FlyWeight{
  // Contains only the necessary data to create an instance of the item
10 private:
    Common common_pointer;
12    Vector position;
    float scale_factor;
14 }
  
```

Table 29: Summary table for the Flyweight Pattern

Pattern Name	Flyweight
When to Use it	When you need to support a large number of similar objects efficiently, when you need to avoid creating a large number of objects.
Advantages	Saves memory when a large number of similar objects is involved, avoids some of the overhead given by the creation of many objects.
Disadvantages	The intrinsic state must be “context independent”, so it cannot change (or all the flyweights that refer to that state will change too). Flyweight instantiation requires particular attention in multithreaded environments, due to the shared memory.

14.1.4 Observer Pattern

The observer pattern is used to implement custom event handling systems, where an object automatically reacts to events generated by another object.

There are 2 main objects used in an observer pattern:

- **The Subject:** sometimes called “Observed Object”
- **The observer:** sometimes called “Dependent Object”

The subject is the creator of a “stream of events” that is consumed by the observer objects.

The subject implements in its structure a list of observers that will be notified when a change occurs, as well as methods to register (add) a new observer as well as to unregister (remove) an existing observer, while the observers will implement a method that will be called by the subject, so that the observers can be notified of such change.

Here we can see an UML diagram of the observer pattern:

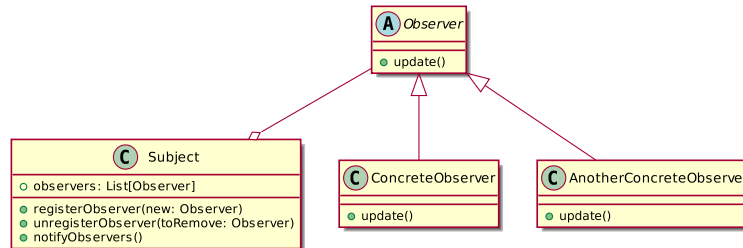


Figure 122: The UML diagram of the observer pattern

Here we can see the Observer abstract class (it can be an interface), a concrete subject and two Concrete Observers that implement what required by the Observer.

Here we can see an implementation of the observer pattern:

Listing 44: Code for an observer pattern

```
class Subject:
2   /* This is the observed class that contains the list of observers and
   * the notifyObservers method */
4
   observers = List()
6
   function register_observer(observer):
8       observers.append(observer)
10
   function notifyObservers():
12       for each observer in observers:
           observer.update()
14
class Observer:
16   /* This is the class that contains the update method, used to force
   * an update in the observer */
18
   function update():
20       print("I have been updated!")
22
subject = Subject()
24 observer = Observer()
   subject.register_observer(observer)
26 subject.notifyObservers()
```

If needed, you can pass information between the subject and the observers just by calling each `update()` method with the necessary arguments.

Table 30: Summary table for the Observer Pattern

Pattern Name	Observer
When to Use it	Event Handling systems, making objects react to other objects' actions
Advantages	Decoupling, added flexibility, more performant than if statements for conditions that happen rarely.
Disadvantages	Can be a bit hard to set up, makes the architecture more complex, if un-registration is not done well there could be serious memory leaks (even in garbage-collected languages).

14.1.5 Strategy

In some situations it may be necessary to select a single algorithm to use, from a family of algorithms, and that decision must happen at runtime.

In this case, the *strategy pattern* (also known as the “policy pattern”), allows the code to receive runtime instructions over what algorithm to execute. This allows for the algorithm to vary independently from the client that makes use of such algorithm.

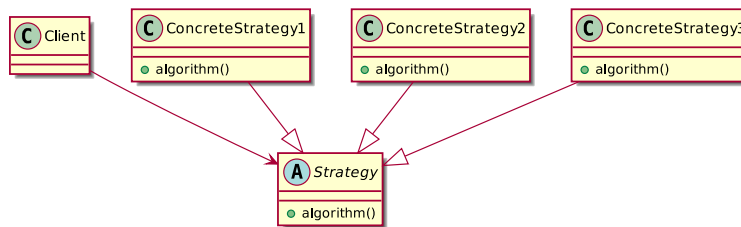


Figure 123: The UML diagram of the strategy pattern

Listing 45: Code for a strategy pattern

```

abstract class Strategy{
2    // This class defines the strategy interface the client will refer to

4    public function algorithm(){
        // This algorithm will be implemented by the subclasses
6    }
8
    class ConcreteStrategy1 inherits from Strategy{
10        public function algorithm() override{
            // Real implementation of the algorithm
12            // DO STUFF
        }
14    }

16    class ConcreteStrategy2 inherits from Strategy{
        public function algorithm() override{
18            // Real implementation of the algorithm
            // DO STUFF SLIGHTLY DIFFERENTLY
20        }
    }
22
    // Example Usage
24    function main(){
        Strategy to_execute;
26        if condition{

```

```

        to_execute = new ConcreteStrategy1();
28     }else{
        to_execute = new ConcreteStrategy2();
30     }
    to_execute.algorithm(); // This will execute 1 or 2 depending on "condition"
32 }

```

Table 31: Summary table for the Strategy Pattern

Pattern Name	Strategy
When to Use it	Every time you need to decide which algorithm to execute at runtime.
Advantages	Decoupling, added flexibility.
Disadvantages	Can cause proliferation of similarly-looking concrete strategies, late binding on functions and the object oriented nature of the pattern could create some overhead.

14.1.6 Chain of Responsibility

Sometimes we have the necessity of handling conditionals that are themselves connected to runtime conditions. This is where the *chain of responsibility pattern* comes into play, being essentially an object-oriented version of an if ... else if ... else statement.

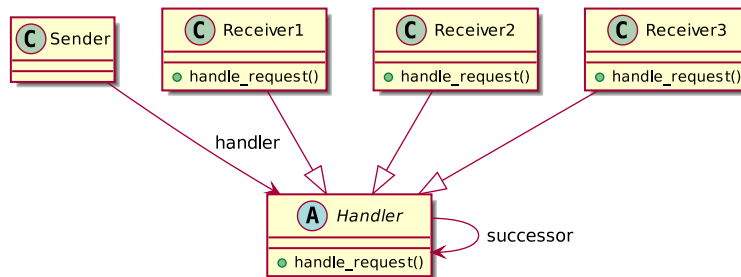


Figure 124: UML Diagram of the Chain of Responsibility Pattern

As can be seen from the diagram, the sender is not directly connected to the receiver, but instead it's connected to a “Handler” interface, making them independent.

As with a chain of responsibility in a company relays a task to “higher ups” if the task cannot be handled, the chain of responsibility pattern involves each received reviewing the request and if possible, process it, if not possible, relay it to the next receiver in the chain.

Listing 46: Code for a chain of responsibility pattern

```

abstract class Handler{
2     // This is the handler abstract/class interface that the sender connects to
    Handler next; // The next handler in the chain
4     public function handle_request(){
        if (condition){
6         // In case I can handle this request
            real_handler();
8     }

10     if next is not null{
        next.handle_request();
12     }
}

```

```
    }  
14  
    private function real_handler(){  
16        // This function gets implemented in the concrete classes  
    }  
18  
    public function add_handler(new_handler){  
20        next = new_handler;  
        return next; // Allows for chaining .add_handler().add_handler()...  
22    }  
}
```

Table 32: Summary table for the Chain of Responsibility Pattern

Pattern Name	Chain of Responsibility
When to Use it	When you need to implement flexible if...else if...else statements that change on runtime. When you want to decouple a sender from a receiver.
Advantages	Decoupling, added flexibility.
Disadvantages	Some overhead is added by the objects and late binding, could lead to proliferation of similar-looking handlers/receivers.

14.1.7 Component/Composite Pattern

When building any game entity, we find that the complexity of the game entity itself literally explodes: a monolithic class can include loads of different operations that should stay separate, such as:

- Input Handling
- Graphics and Animation
- Sound
- Physics
- ...

At this point our software engineering senses are tingling, something is dangerous here.

A better alternative in bigger projects is splitting the monolithic class and create different components and allow for their reuse later. Enter the Component pattern.

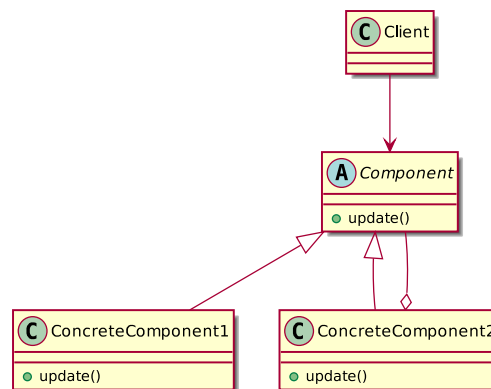


Figure 125: Diagram of the Component Design Pattern

The client is connected to a list of Components that have the same interface (in the previous case, the `update()` method), so each Game Entity can become a “container of components” that define its behaviour.

For instance, instead of having all the functionalities listed above, our game entity could have the following components:

- Input Component
- Graphics Component
- Sound Component
- Physics Component

Which can be reused, extended and allow for further flexibility and follows more closely the DRY principle.

Here we can take a look at a sample implementation of the Component Design Pattern:

Listing 47: Example Implementation Of the Component Pattern

```
abstract class Component{
2     // Defines the abstract class/interface for the component pattern
    function update():
4         // Do nothing, this is an abstract class
}
6
class ConcreteComponent1 inherits from Component{
8     // Defines the concrete component number 1
    function update():
10         // Do Stuff
}
12
class ConcreteComponent2 inherits from Component{
14     // Defines the concrete component number 2

16     // The component can contain a list of other components that get updated
    Component[] list = new vector of 9 Component
18     function update():
        for each component comp in list:
20         comp.update()

22         // Do Other Stuff
}
24
class Client{
26     ConcreteComponent1 first_component;
    ConcreteComponent2 second_component;
28
    function update():
30         // This is the Client's update function
        first_component.update();
32         second_component.update();
}
```

14.2 Resource Manager

A useful container is the “resource manager”, which can be used to store and manage textures, fonts, sounds and music easily and efficiently.

A resource manager is usually implemented via generic programming, which helps writing DRY code, and uses search-efficient containers like hash tables, since we can take care of loading and deletion during loading screens.

First of all, we need to know how we want to identify our resource; there are many possibilities:

- **An Enum:** this is usually implemented at a language-level as an “integer with a name”, it’s light but every time we add a new resource to our game, we will need to update the enum too;

- **The file path:** this is an approach used to make things “more transparent”, but every time a resource changes place, we will need to update the code that refers to such resource too;
- **A mnemonic name:** this allows us to use a special string to get a certain resource (for instance “skeleton_spritesheet”), and every time our resource folder changes, we will just need to update our loading routines (similarly to the Enum solution).

Secondarily, we need to make sure that the container is **thread-safe** (see more about multithreading in the [multithreading section](#)), since we will probably need to implement a threaded loading screen (see how to do it [here](#)) to avoid our game locking up during resource loading.

[This section is a work in progress and it will be completed as soon as possible]

14.3 Animator

This can be a really useful component to encapsulate everything that concerns animation into a simple and reusable package.

The animation component will just be updated (like the other components) and it will automatically update the frame of animation according to an internal timer, usually by updating the coordinates of the rectangle that defines which piece of a sprite sheet is drawn.

[This section is a work in progress and it will be completed as soon as possible]

14.4 Finite State Machine

A finite state machine is a model of computation that represents an abstract machine with a finite number of possible states but where one (or a finite number) of states can be “in execution” at a given time.

We can use a finite state machine to represent the status of a player character, like in the following diagram:

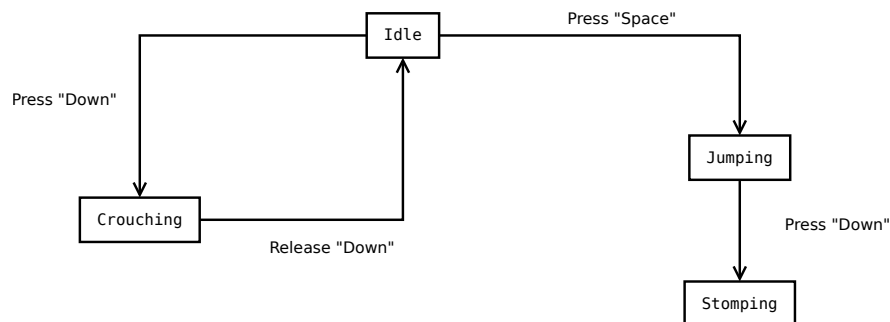


Figure 126: Diagram of a character’s state machine

Each state machine is made out of two main elements:

- **states** which define a certain state of the system (for the diagram above, the states are: Idle, Crouching, Jumping and Stomping);
- **transitions** which define a condition and the change of state of the machine (for the diagram above there are two “Press Down” transitions, one “Release Down” and one “Press Space”)

State machines are really flexible and can be used to represent a menu system, for instance:

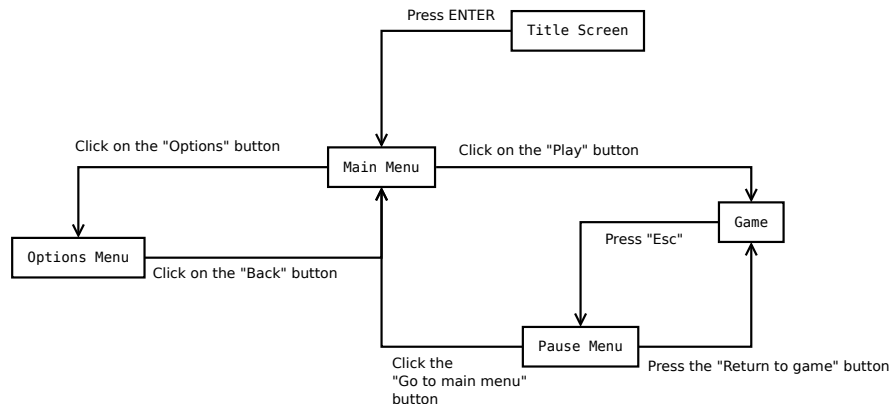


Figure 127: Diagram of a menu system's state machine

In this more convoluted diagram we can see how pressing a certain button or clicking a certain option can trigger a state change.

Each state can be created so it has its own member variables and methods: in a menu system it can prove useful to have each state have its own `update(dt)` and `draw()` functions to be called from the main game loop, to improve on the code readability and better usage of the nutshell programming principle.

[This section is a work in progress and it will be completed as soon as possible]

14.5 Menu Stack

Although menus can be represented via a finite state machine, the structure of an User Interface (UI) is better suited for another data model: the stack (or rather, something similar to a stack).

A stack allows us to code some other functions in an easier way, for instance we can code the “previous menu” function by just popping the current menu out of the stack; when we access a new menu, we just push it into the menu stack and the menu stack will take care of the rest.

Unlike the stacks we are used to, the menu stack can also be accessed like a queue (first in - first out) so you can draw menus and dialogs on top of each other, while the last UI element (on top of the stack) keeps the control of the input-update-draw cycle.

In the menu stack we also have some functionalities that may not be included in a standard stack, like a “clear” function, which allows us to completely clean the stack: this can prove useful when we are accessing the main game, since we may not want to render the menu “below” the main game, wasting precious CPU cycles.

[This section is a work in progress and it will be completed as soon as possible]

14.6 Particle Systems

[This section is a work in progress and it will be completed as soon as possible]

14.6.1 Particles

[This section is a work in progress and it will be completed as soon as possible]

14.6.2 Generators

[This section is a work in progress and it will be completed as soon as possible]

14.6.3 Emitters

[This section is a work in progress and it will be completed as soon as possible]

14.6.4 Updaters

[This section is a work in progress and it will be completed as soon as possible]

14.6.5 Force Application

[This section is a work in progress and it will be completed as soon as possible]

14.7 Timers

[This section is a work in progress and it will be completed as soon as possible]

14.8 Inbetweening

Inbetweening, also known as “tweening”, is a method that allows to “smear” a value over time, this is usually done with animations, where you set the beginning and end position of a certain object, as well as the time the movement should take, and let the program take care of the animation.

This is particularly useful in animating *UI_g* objects, to give a more refined feel to the game.

Here we will present some simple tweenings that can be programmed, and explain them.

Let’s start with a *linear* tweening, usually the following function is used:

Listing 48: Linear Tweening

```
function linearTween(time, begin, change, duration):  
2   return change * (time / duration) + begin
```

Let’s explain the variables used:

- **time**: The current time of the tween. This can be any unit (frames, seconds, steps, ...), as long as it is the same unit as the “duration” variable;
- **begin**: represents the beginning value of the property being inbetweened;
- **change**: represents the change between the beginning and destination value of the property;
- **duration**: represents the duration of the tween.

Note that the measure (time / duration) represents the “percentage of completion” of the tweening.

In some cases a Linear tweening is not enough, that’s where *easing* comes into play.

Before introducing easing let’s analyze the function again, if you try plugging in some data into the function, you will find that there is always going to be:

$$(change\ in\ property) \cdot (factor) + (beginning\ value)$$

So we can use our function substituting **begin** with 0 and **change** with 1 to calculate **factor** and have a code similar to this one:

Listing 49: Example of a simple easing function

```
factor = linearTween(time, 0, 1, duration)  
2 object.property = property_original_value + (destination_value - property_original_value)  
  * factor
```

With linear tweening, the function degenerates to $\frac{time}{duration}$, but now we can replace our linear tween with the following function:

Listing 50: Ease-in

```
function easeIn(time, duration, power):  
2   return (time/duration)^power
```

By changing the `power` parameter, we change the behaviour of the easing, making the movement slower at the beginning and pick up the pace more and more, until the destination is reached. This is called a “ease-in”.

For an “ease-out”, where the animation starts fast and slows down towards the end, we use the following function instead:

Listing 51: Ease-out

```
function easeOut(time, duration, power):  
2   return 1 - (1 - (time / duration)) ^ power
```

With some calculations, and if statements on the time passed, you can combine the two and get an “ease-in-out” function.

Obviously these functions have an issue: they don’t clamp the value between 0 and 1, that will have to be done in the movement function or by adding a check, or using some math, for instance using `min(calculated_value, 1)`.

[This section is a work in progress and it will be completed as soon as possible]

14.9 Chaining

[This section is a work in progress and it will be completed as soon as possible]

15 Artificial Intelligence in Videogames

The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.

Edsger W. Dijkstra

In this part of the book we will take a look at some data structures and algorithms that will help you building your game's Artificial Intelligence.

15.1 Path Finding

Path Finding is that part of AI algorithms that takes care of getting from point A to point B, using the shortest way possible and avoiding obstacles.

15.1.1 Representing our world

Before undertaking the concept of path finding algorithms, we need to decide in which way we should represent our world to the AI we want to code. There are different ways to do it and here we will discuss some of the most used ones.

15.1.1.1 2D Grids

The simplest way to represent a world to an AI is probably using 2D grids: we can represent the world using a 2-dimensional matrix, where every cell is a free space, an obstacle, a start or goal cell.

This representation works well with top-down tile-based 2D games (with tile-based or free movement).

Listing 52: Possible representation of a 2D grid

```
class 2D_Grid():
2   // Represents a 2D grid of "Tile" classes
   Tile[][] grid = null
4   int width = 0
   int height = 0
6
   function constructor(int rows, int cols):
8       // Prepares the memory for the grid
       grid = new Tile[rows][cols]
       height = rows
10      width = cols
12
   function getCell(int row, int col):
14       // Gets a cell from the 2D Grid
       if (row >= 0 AND row < height) AND (col >=0 AND col < width):
16           // We better check if we are inside the grid
           return grid[row][col]
18       else:
           return null
20
   function getAdjacentCells(int row, int col):
22       /* Returns a list of cells adjacent the ones we give
          REMEMBER: We index at 0 so the first row is 0, the last one is at
24       "height - 1", same goes for columns */
       Tile[] toReturn = new Tile[] // We assume arrays can resize when necessary
26       if (row >= 0 AND row < height) AND (col >=0 AND col < width):
           // We better check if we are inside the grid
28           if (row > 0):
               // We are not on the first row, we can add the cell above
               toReturn.append(getCell(row - 1, col))
30           if (row < height - 1):
```

```
32         // We are not on the last row, we can add the cell below
        toReturn.append(getCell(row + 1, col))
34     if (col > 0):
        // We are not on the first column, we can add the cell on the left
36         toReturn.append(getCell(row, col - 1))
        if (col < width - 1):
38             // We are not on the last column, we can add the cell on the right
            toReturn.append(getCell(row, col + 1))
40     /* If the checks went well, toReturn will have
        a list of the adjacent cells, if not it will be empty */
42     return toReturn
```

Even though this is probably the most straightforward way to represent a world in many cases, most of the algorithms used work on a graph structure instead of a 2D grid. It shouldn't be too hard to adapt the algorithms presented here to work with this structure.

15.1.1.2 Path nodes

A more flexible way to represent our world is using “Path nodes”, where each “path node” is represented by a node in a graph.

Graphs can be represented in code in two common ways (there are surely other ways to do so): using *adjacency lists* or using *adjacency matrices*.

This type of graph-based abstraction is the most used when teaching path finding algorithms like **A*** or **Dijkstra**.

15.1.1.2.1 Adjacency Lists

[This section is a work in progress and it will be completed as soon as possible]

15.1.1.2.2 Adjacency Matrices

[This section is a work in progress and it will be completed as soon as possible]

15.1.1.3 Navigation meshes

Navigation meshes are used to solve a problem that can arise when we try to represent our world using path nodes: we can't represent “safe areas” (where the AI-driven entity can cross) without using possibly thousands of path nodes.

Navigation meshes are constituted by a collection of convex polygons (the meshes) that define the “safe areas”, each mesh has no obstructions inside of itself, so the AI-driven entity can safely cross the entire mesh freely.

This abstraction allows to use **A*** and **Dijkstra** algorithms, but instead of trying to navigate a graph, you look for a path between meshes (which are saved in a graph structure).

[This section is a work in progress and it will be completed as soon as possible]

15.1.2 Heuristics

In path finding there can be “heuristics” that are accounted for when you have to take a decision: in path finding an heuristic $h(x)$ is an estimated cost to travel from the current node to the goal node.

An heuristic is admissible if it *never overestimates* such cost: if it did, it wouldn't guarantee that the algorithm would find the best path to the goal node.

In this book we will present the most common heuristics used in game development.

15.1.2.1 Manhattan Distance heuristic

The Manhattan Distance heuristic doesn't allow diagonal movement (allowing it would allow the heuristic to overestimate the cost), and for a 2D grid is formulated as follows:

$$h(x) = |start.x - goal.x| + |start.y - goal.y|$$

Graphically:

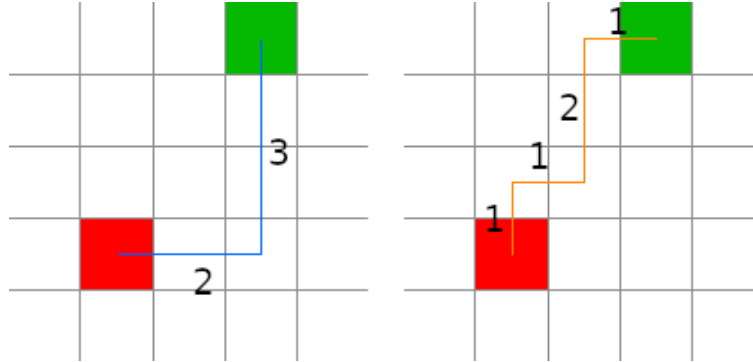


Figure 128: Example of Manhattan distance

On the left we can see how we calculate the Manhattan distance, on the right you can notice how all the “possibly shortest” alternative paths have the same Manhattan distance.

Since all “possibly shortest” paths between two points have the same Manhattan distance, this guarantees us that the algorithm will never overestimate (all the other paths will be longer, so the Manhattan distance will underestimate the cost), which is required for this heuristic to be considered “admissible”.

Listing 53: Example code calculating the Manhattan distance on a 2D grid

```
struct Tile{
2     integer x;
    integer y;
4 }

6 function manhattan_distance(Tile start, Tile goal):
    return abs(start.x - goal.x) + abs(start.y - goal.y)
```

This works well with 2D grid-based worlds.

15.1.2.2 Euclidean Distance heuristic

Euclidean Distance works well when diagonal movement in a 2D grid is allowed, Euclidean distance is calculated with the standard distance formula:

$$h(x) = \sqrt{(start.x - end.x)^2 + (start.y - end.y)^2}$$

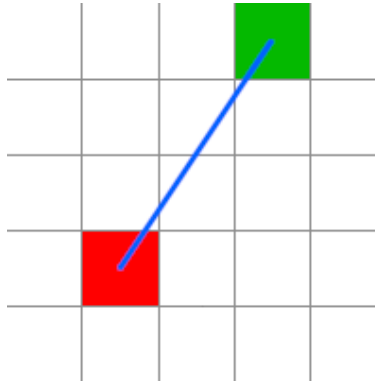


Figure 129: Example of Euclidean Distance

Listing 54: Example code calculating the Euclidean distance on a 2D grid

```
struct Tile{  
2     integer x;  
     integer y;  
4 }  
  
6 function euclidean_distance(Tile start, Tile goal):  
     return square_root((start.x - goal.x) ^2 + (start.y - goal.y)^2)
```

15.1.3 Algorithms

Before getting to the algorithms, we need to consider two supporting data structures that we will use:

- **Open Set:** a sorted data structure that contains the nodes that currently need to be considered. It should be an heap or any kind of structure that can be quickly be sorted as we will have to often refer to the node/cell/mesh with the lowest heuristic.
- **Closed Set:** a data structure that will contain all the nodes/meshes/cells that have already been considered by the algorithm. This structure should be easily searchable (like a binary search tree), since we will often need to see if a certain node/cell/mesh is part of this set.

We will reference this image when we check what path the algorithm will take:



Figure 130: Pathfinding Algorithms Reference Image

The shaded rows represent the indexes we'll refer to when operating the algorithm.

In each algorithm there will be the path taken by its implementation, so we invite you to execute the algorithm's instructions by hand, taking account of the heuristics pre-calculated and shown in the images below.

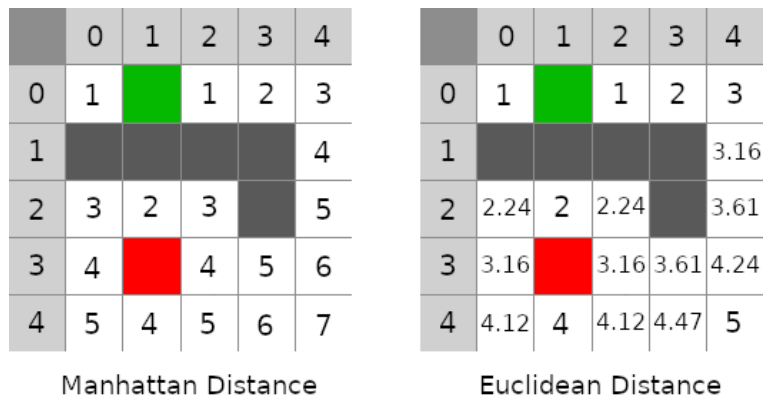


Figure 131: Pathfinding Algorithms Heuristics Reference Image

15.1.3.1 The Greedy “Best First” Algorithm

This is a *greedy algorithm* that searches the “local best” (what is best in a certain moment, without planning future decisions) that makes use of heuristics.

For each of the neighbouring cells/meshes/nodes that have not been explored yet, the algorithm will take the one that has the lowest heuristic cost. Since this algorithm doesn't make any planning, this can lead to results that are not optimal, usually translating in entities hugging walls to reach their goal, as well as taking longer paths.

In this algorithm we will use a “Node” structure composed as follows:

Listing 55: The node structure used in the greedy "Best First" algorithm

```

1 struct Node{
2     Node parent; // This will be used to build the path
3     float h; // The h(x) value for the node
4 }

```

Let's look at the algorithm itself:

Listing 56: The greedy "Best First" algorithm

```

1 // We bootstrap the variables
2 currentNode = start
3 add currentNode to closedSet // This will avoid re-analizing this node
4 do{
5     for each Node n adjacent to currentNode{
6         if closedSet contains n{
7             // We already analyzed this node, continue to next n
8             skip the node n
9         }else{
10            n.parent = currentNode
11            if openSet does not contain n{
12                compute n.h // Computes the value of n's h(x)
13                add n to openSet
14            }
15        }
16    }
17 }

```

```

16     }
17     if openSet is empty{
18         // We exhausted all the possibilities
19         break do loop
20     }
21     // Select a new "currentNode"
22     currentNode = Node with lowest h inside the openSet
23     remove currentNode from openSet
24     add currentNode to closedSet
25 }until (currentNode == end)
26
27 if currentNode == end{
28     /* We reached the end and solved the path, we need to do a stack
29        reversal to find the path */
30     Stack finalPath
31     Node n = end
32     while n is not null{
33         add n to finalPath
34         n = n.parent // We use "parent" to run the found path backwards
35     } loop
36 }else{
37     // We cannot find a path between "start" and "end"
38 }

```

An interesting idea to optimize this algorithm and avoid the final “stack reversal” would be to find the path starting from the end node, towards the starting node.

In the image below we can see the path taken by the algorithm, and how it is not the most optimal path.

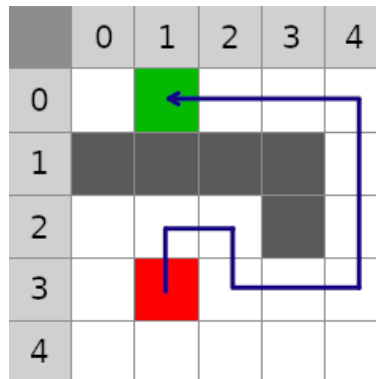


Figure 132: The path taken by the greedy “Best First” algorithm

15.1.3.2 The Dijkstra Algorithm

The idea behind the Dijkstra algorithm is having a “cost” component that expresses the cost that has to be incurred when traveling from the start node to the current node. This will allow our previous algorithm to evolve and take the shortest path possible.

To be able to keep track of such “path-cost” component, we will use a different “Node” structure from the one used in the greedy “best-first” algorithm:

Listing 57: The node structure used in the Dijkstra Algorithm

```

struct Node{
2     Node parent; // This will be used to build the path
    float g; // The path cost value for the node
4 }

```

The idea behind the whole algorithm is that “if we find a quicker way to get from the start node to the current node, we should take it”.

Let’s take a look at the algorithm:

Listing 58: The Dijkstra Algorithm

```

currentNode = start
2 add currentNode to closedSet
do{
4   for each Node n adjacent to currentNode{
       if closedSet contains n{
6           skip the node n
       }else if openSet contains n{ // Check if this path is better
8           compute new_g (path cost value)
           if new_g < n.g{
10              // We found a better path from start to currentNode
                  n.parent = currentNode
12              n.g = new_g
           }
       }else{
14           n.parent = currentNode
16           compute n.g
           add n to openSet
18       }
   } loop
20
22   if openSet is empty{
       // We exhausted all possibilities
       break loop
24   }
   currentNode = Node with the lowest g in the openSet
26   remove currentNode from openSet
   add currentNode to closedSet
28 } until(currentNode == end)

30 // Reconstruct the path like in the greedy "best-first" algorithm

```

As with the greedy “best-first” algorithm we can optimize the “stack reversal” stage by starting from the end node.

Below we can see the path taken by the algorithm:

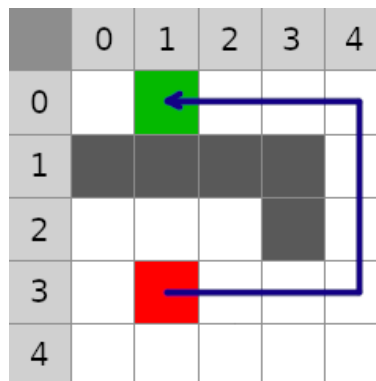


Figure 133: The path taken by the Dijkstra Algorithm

15.1.3.3 The A* Algorithm

The A* Algorithm joins the “path-cost” idea with the heuristic to have a more efficient path-finding algorithm. The algorithm is really similar to Dijkstra, but it orders the open set by a new formula f , that is calculated as follows:

$$f(x) = g(x) + h(x)$$

Where $g(x)$ is our path cost and $h(x)$ is the heuristic we selected.

Let’s see the code:

Listing 59: The A* Algorithm

```
currentNode = start
2 add currentNode to closedSet
do{
4   for each Node n adjacent to currentNode{
       if closedSet contains n{
6           skip the node n
       }else if openSet contains n{ // Check if this path is better
8           compute new_g (path cost value)
           if new_g < n.g{
10              // We found a better path from start to currentNode
                n.parent = currentNode
12              n.g = new_g
                n.f = n.g + n.h
14          }
       }else{
16          n.parent = currentNode
           compute n.g
18          compute n.h
           n.f = n.g + n.h
20          add n to openSet
       }
22 } loop

24 if openSet is empty{
       // We exhausted all possibilities
26   break loop
}
28 currentNode = Node with the lowest f in the openSet
   remove currentNode from openSet
30   add currentNode to closedSet
} until(currentNode == end)
32 // Reconstruct the path like in the greedy "best-first" algorithm
```

The path taken by the A* Algorithm is exactly the same as the one taken by the Dijkstra Algorithm, but the heuristic helps the A* Algorithm in visiting less nodes than its counterpart.

15.1.3.3.1 Dijkstra Algorithm as a special case of the A* Algorithm

The Dijkstra Algorithm can be implemented with the same code as the A* Algorithm, just by keeping the heuristic cost $h(x) = 0$.

The absence of the heuristics (which depends on the goal node) leads the Dijkstra Algorithm to visit more nodes, but it can be useful in case there are many valid goal nodes and we don’t know which one is the closest.

15.2 Finite state machines

We can use finite state machines, introduced previously, to define some quite complex Artificial Intelligence schemes.

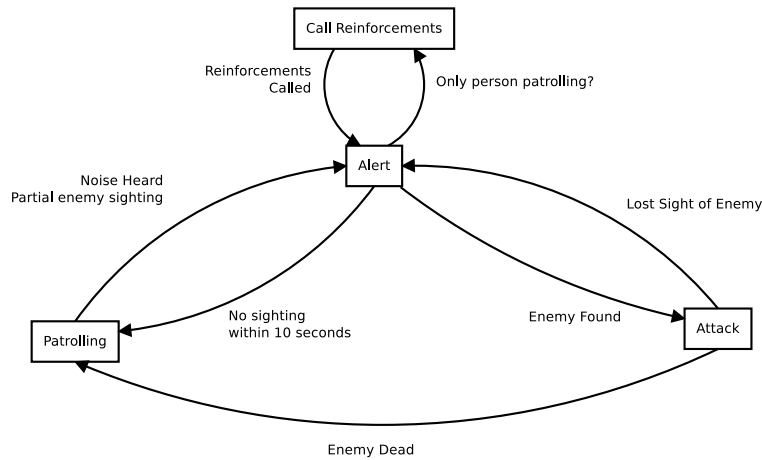


Figure 134: Finite state machine representing an enemy AI

We can see in the previous image how we can use conditions as transition between different “states of mind” of our enemy AI, making it act differently.

The enemy will be patrolling by default, but if the player is heard or seen the enemy will enter its “alert state”, where it will either call for backup or actively search for the player. As soon as the player is found, the enemy will attack and try to kill the player.

15.3 Decision Trees

[This section is a work in progress and it will be completed as soon as possible]

15.4 Behaviour Trees

[This section is a work in progress and it will be completed as soon as possible]

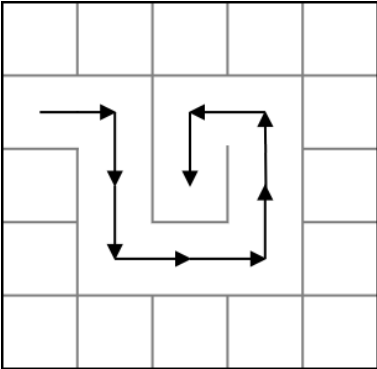


Figure 136: How the recursive backtracker algorithm works (2)

In such case, we “backtrack” until we find a cell with at least one available direction and continue our exploration.

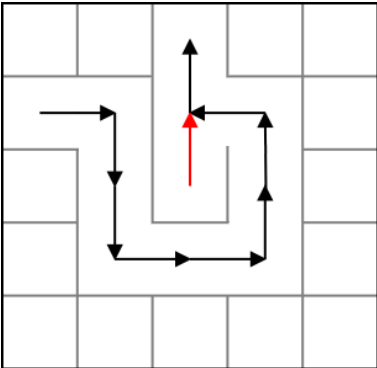


Figure 137: How the recursive backtracker algorithm works (3)

This “digging and backtracking” keeps going until there are no other cells that have not been visited.

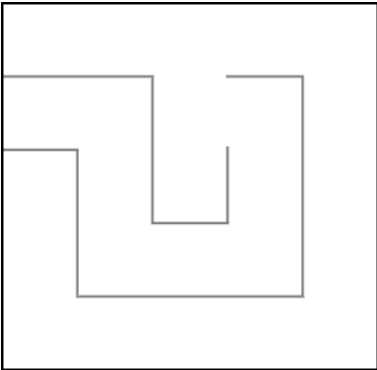


Figure 138: How the recursive backtracker algorithm works (4)

In some versions of the algorithm we need to also keep track of cells that will be used as “walls”, so the actual implementation varies.

[This section is a work in progress and it will be completed as soon as possible]

This algorithm can involve a big deal of recursion, which can lead to a *stack overflow*_g in your program, stopping the algorithm from working and your game in its entirety. It is possible to work around this issue by using an explicit stack, instead of using the call stack.

[This section is a work in progress and it will be completed as soon as possible]

This algorithm, being taken from a Depth-First search algorithm, is biased towards creating very long corridors.

16.1.1.2 Randomized Kruskal's Algorithm

This algorithm is based on a randomized version of the minimum-spanning tree algorithm known as Kruskal's algorithm.

The algorithm needs the following data structures to work:

- One structure that contains all the “walls” of our maze, this can be a list
- One structure that allows for easy joining of disjoint sets, this will contain the cells

Initially all the cells are separated by walls, and each cell is its own set.

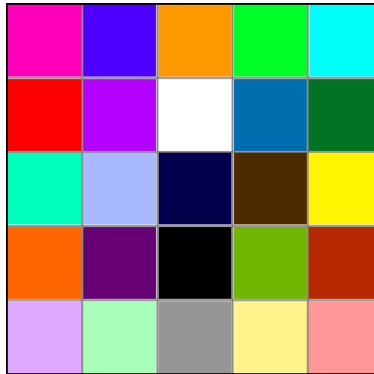


Figure 139: How the Randomized Kruskal's Algorithm Works (1/6)

Now we select a random wall from our list, if the cells separated by such wall are part of different sets, we delete the wall and join the cells into a single set.

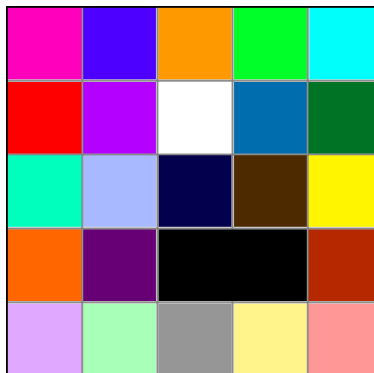


Figure 140: How the Randomized Kruskal's Algorithm Works (2/6)

The “different sets” check allows us to avoid having loops in our maze (and also deleting all the walls, in some cases). Next we select another wall, check if the cells divided by the wall are from different sets and join them.

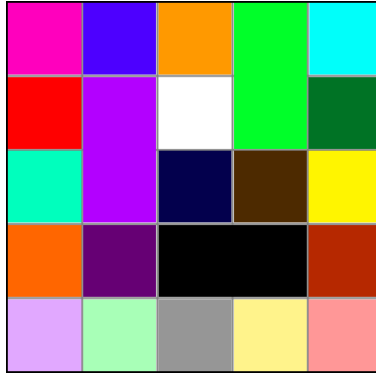


Figure 141: How the Randomized Kruskal’s Algorithm Works (3/6)

This doesn’t look much like a maze yet, but by uniting the cells we can start seeing some short paths forming in our maze.

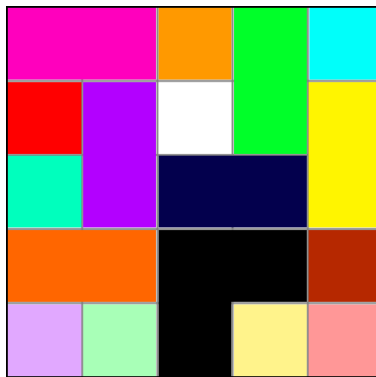


Figure 142: How the Randomized Kruskal’s Algorithm Works (4/6)

The black cells are starting to develop a path, as stated earlier. As the sets get bigger, there will be less walls we can “break down” to join our sets.

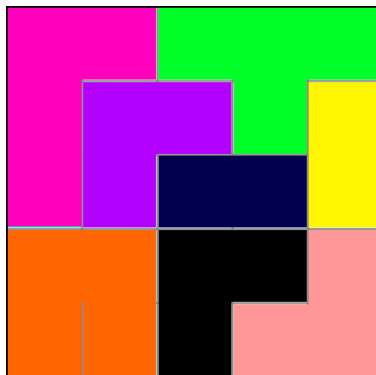


Figure 143: How the Randomized Kruskal’s Algorithm Works (5/6)

When there is only one set left, our maze is complete.

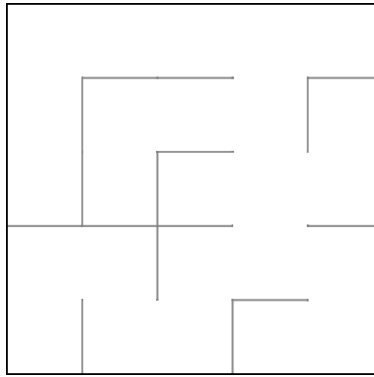


Figure 144: How the Randomized Kruskal's Algorithm Works (6/6)

This algorithm, being based on a minimum-spanning tree algorithm, this algorithm is biased towards creating a large number of short dead ends.

Now let's see an example implementation of the Randomized Kruskal's Algorithm:

[This section is a work in progress and it will be completed as soon as possible]

16.1.2 Recursive Division Algorithm

This algorithm is a bit similar to the recursive backtracker, but instead of focusing on passages, this algorithm focuses on walls: the idea is recursively dividing the space available with a horizontal or vertical wall which has a "hole" placed randomly.

This algorithm can give better results when the choice between "vertical" and "horizontal" walls is biased by the sizes of the sub-areas given by the last division.

Let's see how the algorithm works.

Starting from an empty maze, with no walls, we decide the direction (horizontal or vertical) of our first wall and add it, in a random position, making sure that there's an opening in such wall.

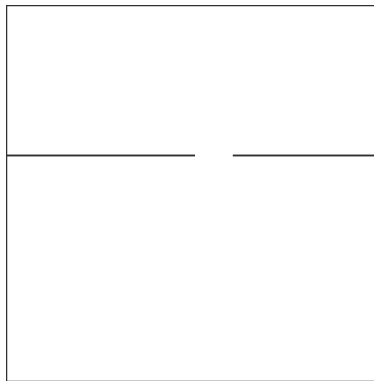


Figure 145: How the Recursive Division Algorithm Works (1/6)

We select one of the two sub-areas we find, recursively and we add another wall in a random position and with a random direction.

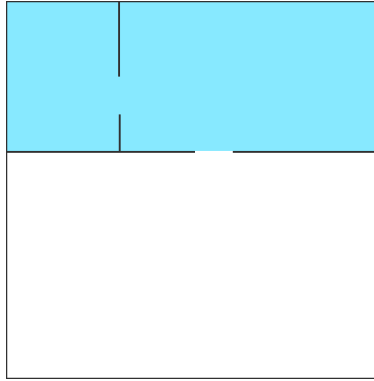


Figure 146: How the Recursive Division Algorithm Works (2/6)

We select one of the two sub-sub-area, and add another wall, with a random position and direction.

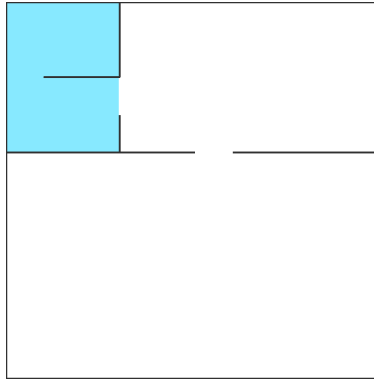


Figure 147: How the Recursive Division Algorithm Works (3/6)

We keep on diving each sub-area recursively, adding walls, until the sub-area had one of its 2 dimensions (horizontal or vertical) equal to 1 cell.

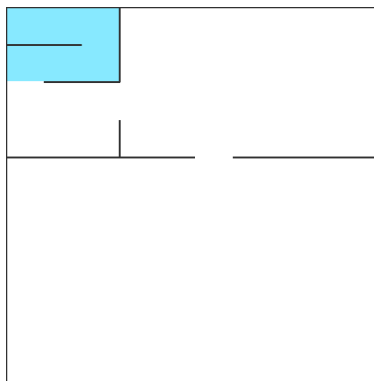


Figure 148: How the Recursive Division Algorithm Works (4/6)

When that happens, we backtrack to one of the previous sub-sections and continue.

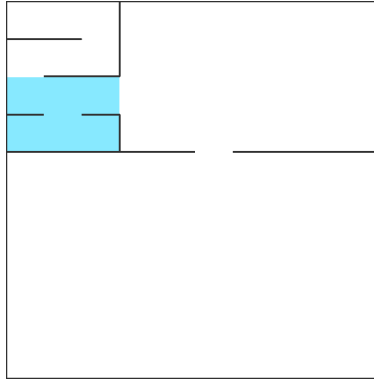


Figure 149: How the Recursive Division Algorithm Works (5/6)

This keeps going until the maze is complete.

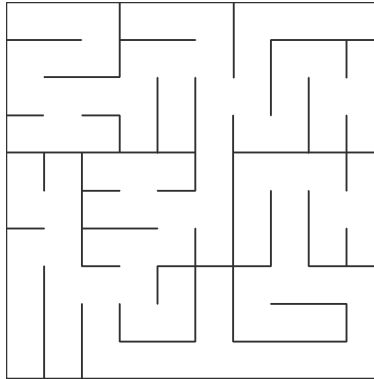


Figure 150: How the Recursive Division Algorithm Works (6/6)

Although it's one of the most efficient algorithms out there (considering that it can easily be converted to a multi-threaded version), given its nature, this algorithm is naturally biased towards building long walls, which give the maze a very “rectangle-like” feeling.

This bias is more noticeable with bigger mazes, like the following one.

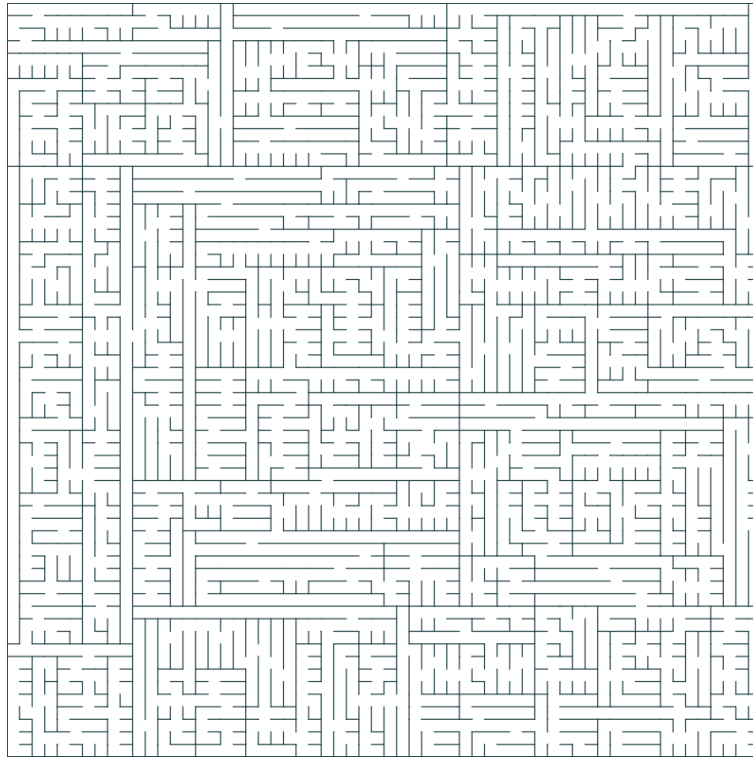


Figure 151: The bias of Recursive Division Algorithm

Let's see an example implementation of this algorithm:

[This section is a work in progress and it will be completed as soon as possible]

16.1.3 Binary Tree Algorithm

This is another very efficient “passage carver” algorithm: for each cell we carve a passage that either leads upwards or leftwards (but never both).

Let's see how the algorithm works: we start from the bottom-right cell of the maze (the last one).

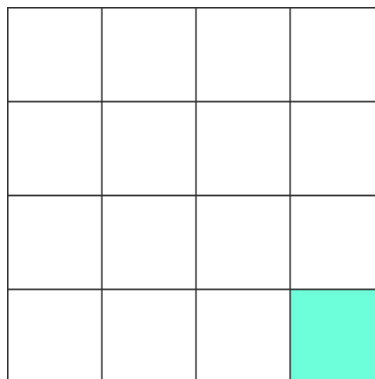


Figure 152: How the Binary Tree Maze generation works (1/6)

Now we decide, randomly, to carve a passage either upwards or leftwards (we will not carve a passage that “creates a hole in a wall”). In this case we go leftwards.

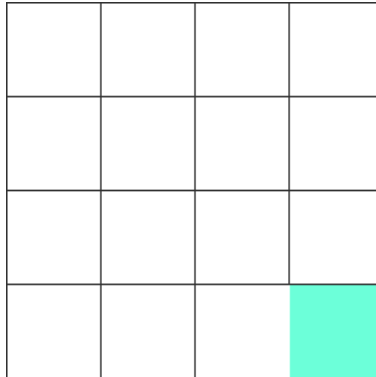


Figure 153: How the Binary Tree Maze generation works (2/6)

Now let's go to the second-to-last cell...

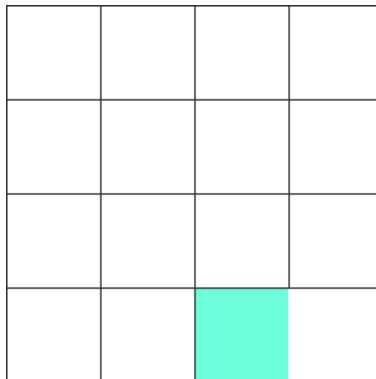


Figure 154: How the Binary Tree Maze generation works (3/6)

And again, decide randomly to carve a passage either upwards or leftwards, this time we chose upwards.

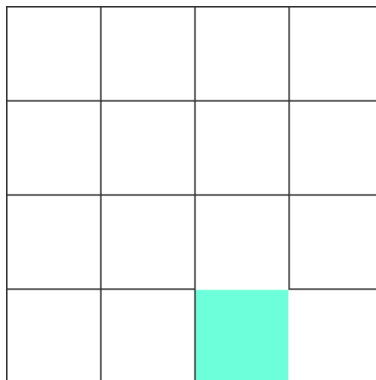


Figure 155: How the Binary Tree Maze generation works (4/6)

Again we go to the “previous” cell and continue with our process, until we hit the left wall (which will force us to carve a passage upwards) or the top wall (which will force us to go left); when we hit both the top and the left walls, we stop.

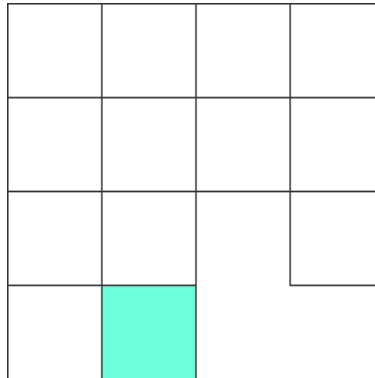


Figure 156: How the Binary Tree Maze generation works (5/6)

Here is the result of the algorithm:

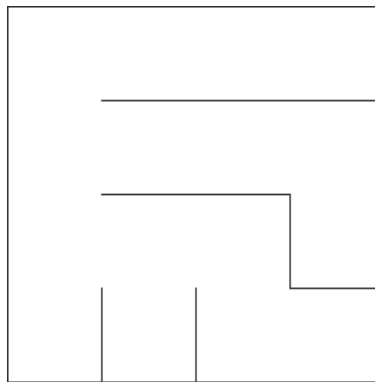


Figure 157: How the Binary Tree Maze generation works (6/6)

Given its deep roots into the computer science “Binary Tree” structure (where the root is the upper-left corner), this algorithm shows only half of the cell types available in mazes: there are no crossroads and all dead ends will either have a passage upwards or leftwards (but again, never both at the same time).

Let’s see an example implementation of the “binary tree algorithm”:

[This section is a work in progress and it will be completed as soon as possible]

16.1.4 Eller’s Algorithm

Eller’s algorithm is the most memory-efficient maze-generation algorithm known so far: you generate the maze row-by-row, without needing to memorize the whole maze in memory while creating it.

To start, we decide the width and height of the maze, and create a single row, large as the width we want. Then we set each cell to be its own “set”.



Figure 158: How Eller's Maze Generation Algorithm Works (1/7)

Now we scroll through the cells and randomly join adjacent cells that are part of two different “sets”.



Figure 159: How Eller's Maze Generation Algorithm Works (2/7)

After joining we create some “holes” in the bottom wall, making sure that each “set” has at least one hole to get to the next row.



Figure 160: How Eller's Maze Generation Algorithm Works (3/7)

After that we start creating the next row, connecting the cells that have a “hole” with the previous row and assigning them the same set. In the picture the gray cells didn't get a set assigned yet.

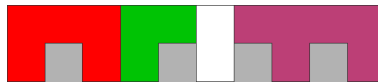


Figure 161: How Eller's Maze Generation Algorithm Works (4/7)

After that we assign a new set to the remaining cells.



Figure 162: How Eller's Maze Generation Algorithm Works (5/7)

At this point we just need to iterate, ignoring the previous row: we join adjacent cells that are not part of the same “set” (we grayed out the previous row).



Figure 163: How Eller's Maze Generation Algorithm Works (6/7)

Then we create “holes” for each set and prepare the next row. In case we want the maze to be wholly interconnected then if the row is the last row, we can just join all the cells.

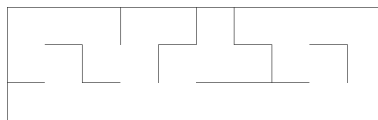


Figure 164: How Eller's Maze Generation Algorithm Works (7/7)

Obviously we can repeat the iteration as many times as we want, and we get a maze as big as we want. This algorithm has no obvious biases and is good for very efficient dungeon generation, if you add rooms, for instance.

Let's see a possible implementation of this strange, but interesting algorithm:

[This section is a work in progress and it will be completed as soon as possible]

16.2 Noise Generation

“Noise” can be a very important part of game development: we can create textures with it, or even use it to generate worlds. In this section we will take a look at how to create “noise” efficiently and with the desired result: from completely randomized to more “natural looking” noise we can use to create maps.

16.2.1 Randomized Noise (Static)

The simplest kind of noise we can generate is also known as “static”, for each unit of our elaboration (it can be a pixel, for instance), we generate a random number between two bounds.

Here is an example of random noise:

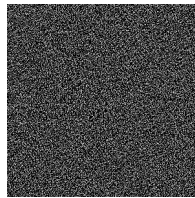


Figure 165: Example of Random Noise

We can create some “TV-like” static with a few lines of code, like the following:

Listing 60: Example implementation of randomized noise

```
1 int WIDTH = 800
2 int HEIGHT = 600
3 // We create an empty texture
4 Texture texture = Texture(WIDTH, HEIGHT)

6 // Now we iterate through each pixel of the texture
7 for row in texture:
8     for pixel in row:
9         // We create a random gray color (0 is black, 255 is white)
10        int rand_gray_tone = random_int(0,255)
11        // Most colors are made of Red Green and Blue, by placing them at the
12        // same value, we get a tone of gray
13        Color rand_color = Color(rand_gray_tone, rand_gray_tone, rand_gray_tone)
14        pixel.setColor(rand_color)
```

16.2.2 Perlin Noise

[This section is a work in progress and it will be completed as soon as possible]

17 Developing Game Mechanics

Fools say that they learn by experience. I prefer to profit by others' experience.

Otto von Bismarck

In this section we will take a look at how to develop some known game mechanics, with pieces of code to aid you.

17.1 General Purpose

17.1.1 I-Frames

I-Frames (also known as “invincibility frames”) is a term used to identify that period of time after being hit, where the character either flashes or becomes transparent and is immune to damage.

This mechanic can be seen as “giving the player an advantage” but instead it has deeper roots into “fairness” than “difficulty management”, let’s see why.

Let’s assume that our main character has 100 health points, and touching an enemy deals 5 points of damage. In absence of I-Frames, this would translate into 5 points of damage every frame, which would in turn come out between $5 \cdot 30 = 150$ and $5 \cdot 60 = 300$ points of damage per second (at respectively 30 and 60fps).

The average human reaction time is around 1 second, this would mean that touching an enemy would kill us before we even realize we are touching such enemy.

Checking if we’re still colliding with an enemy after receiving damage is not a good strategy, since that would allow the player get only one point of damage from a boss, and then carefully stay inside the boss’s hitbox while dealing damage to the enemy. Thus allowing the player to exploit the safeguard.

Giving a brief period (usually between 0.5 and 2 seconds) of invincibility after being hit, allows the player to understand the situation, reorganize their strategy and take on the challenge at hand. After the invincibility period, the player will take damage again, patching the exploit we identified earlier.

I-Frames can be easily implemented via timers, in a way similar to the following:

Listing 61: Example of I-Frames Implementation

```
const float INVINCIBILITY_TIME = 0.75 // Seconds of invincibility
2 ...
function update(dt):
4     float inv_time = 0
    ...
6     if inv_time == 0:
        // Check for collision
        ...
8         // Collision has been detected here, we have a hit
        inv_time = INVINCIBILITY_TIME // Start of the invincibility period
10    else:
        // We are currently invincible
12        inv_time = inv_time - dt // We decrease the invincibility time
```

17.1.2 Scrolling Backgrounds and Parallax Scrolling

17.1.2.1 Infinitely Scrolling Backgrounds

When doing any kind of game that features a scrolling background, you should construct your art accordingly, allowing for enough variety to make the game interesting while avoiding creating huge artwork that weighs on the game’s performance.

In a game that uses a scrolling background, the background used should be at least two times the screen size, in the scrolling direction (**Virtual Resolution** can prove really useful in this case) and the image should have what are called “loop points”.

Loop points are points where the image repeats itself, thus allowing us to create an image that is virtually infinite, scrolling through the screen. To have a so-called “loop point” the image should be at least twice the size of the screen, in the scrolling direction.

The image below shows a background and its loop points.

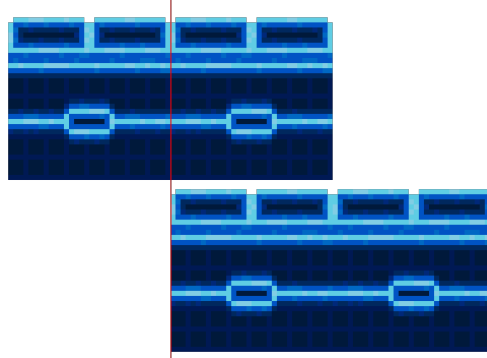


Figure 166: Demonstration of an image with loop points

To make an image appear like it’s scrolling infinitely we need to move it back and forth between loop points when the screen passes over them.

For ease of explanation let’s consider a screen scrolls towards the right, when we have reached a loop point, we reset the image position back to the position it was at the beginning and, since the image has been crafted to loop, the player won’t notice that the background has been reset.

Listing 62: Example of an infinitely scrolling background implementation

```
float background_x_offset = 0.0 // The x offset of the background
2 const float BACKGROUND_X_SIZE = 512 // The horizontal size of the background
const float LOOP_POINT = 256 // The horizontal loop point of the image
4 const float DISTANCE_FACTOR = 0.5 // The background moves at half the player speed

6 function update(dt):
    ...
8 // In case we're moving right, the background scrolls left slightly
if player.speed_x > 0:
10 // Update player's position and state
    ...
12 // Update the background position
background_x_offset = background_x_offset - player.speed_x * DISTANCE_FACTOR * dt
14 // If we passed the image's loop point
if background_x_offset <= - LOOP_POINT:
16 // We reset the coordinates, keeping note of the remainder
background_x_offset = background_x_offset % LOOP_POINT
18 // In case we're moving left, the background scrolls right slightly
if player.speed_x < 0:
20 // Update player's position and state
    ...
22 // Update the background position
background_x_offset = background_x_offset - player.speed_x * DISTANCE_FACTOR * dt
24 if background_x_offset >= 0:
// We reset the coordinates, keeping note of the remainder, just backwards
26 background_x_offset = background_x_offset - BACKGROUND_X_SIZE
```

```
28 function draw():  
    ...  
30    // Draw the background  
    screen.draw(background, (background_x_offset, 0))  
32    ...
```

17.1.2.2 Parallax Scrolling

Parallax in games is an effect that can give more depth to our environment: it looks like objects farther away are moving much slower than the objects closer to us.

This can be used to our advantage, along with some other tricks to enhance the perception of depth explained in [the chapter dedicated to graphics](#).

Creating a parallax effect is quite easy: first we need at least two background layers (although three seems to be the best compromise between performance and depth of the effect):

- The **sprite layer** that will represent the closest layer to us, that will move at a certain speed that will be decided while developing the game;
- A **moving background** that will move slower compared to the sprite layer, thus giving the parallax effect;
- A **fixed background** that will represent our horizon and the farthest objects.

As stated earlier, a third optional background can be added to deepen the parallax scrolling effect, such background can take any of these positions:

- **Above** the sprite layer: in this case this “foreground layer” will need to move **faster** than the sprite layer and it should include very unobtrusive graphics, to avoid hiding important gameplay elements (like enemies);
- **Between the sprite layer and the first moving background**: in this case, the optional background should move **slower** than the sprite layer, but **faster** than the first moving background;
- **Between the first moving background and the fixed background**: in this case, the optional background will have to move **slower** than the first moving background.

The backgrounds should move all in the same direction, depending on the direction our character is moving: if our character is moving right, our moving backgrounds should move left.

17.1.3 Avoid interactions between different input systems

This is a small improvements that can be done on menu systems: if the player is using a keyboard to navigate the menu, the mouse should not interact with the menu.

In many frameworks when a fullscreen game window “captures” the mouse cursor, this is put on the center of the screen, which could be where a menu item is positioned.

Now imagine you are “flowing through” the menu, trying to load a saved file and the cursor is detected pointing at the “delete savefile” option; you are briskly walking through the menu and what you think is the “do you want to load this file?” dialog is actually asking “do you want to **delete** this savefile?”. You click “yes” and your savefile is gone!

This is an extreme edge case, but it could happen. Even if it is a minor annoyance like starting a new savefile when instead you want to load an existing one, it diminishes the quality of your experience.

17.1.4 Sprite Stenciling/Masking

[This section is a work in progress and it will be completed as soon as possible]

17.1.5 Loading screens

If you load your resources in the same thread that executes the main game loop, your game will lock up while loading, which may trigger windows to ask you if you want to terminate the task. In this case it is better to dip our toes into **multithreading** and create a proper loading screen.

The loading screen will be composed of two main components:

- **The graphical loading screen:** that will show the progress of the resource loading to the user, as well as tips and animations;
- **The actual resource loading thread:** that will take care to load the resources to the right containers, as well as communicating the global loading status to the loading screen in the main game loop.

[This section is a work in progress and it will be completed as soon as possible]

17.2 2D Platformers

17.2.1 Simulating Gravity

Gravity in a 2D platformer is quite easy to simulate: you just need to apply a constant acceleration towards the direction your gravity is pulling (it doesn't have to be towards the bottom of the screen!) and move your objects accordingly to such acceleration.

Your acceleration should not be precise (like the physics constant $9.81m/s^2$), you don't want to make a physics engine: you want to make a somewhat convincing (or even better: entertaining) approximation of reality.

This is usually done before the player movement is used to update the character's status (but after the player input has been captured). Remember to add this acceleration before the collision detection is processed.

A useful precaution to avoid the **bullet through paper** problem when you are working with long falls: put a limit at the fall velocity (kind of like air friction limits an object's fall velocity) of your objects. By applying a hard limit to the velocity, your gravity will be realistic but won't break your simulation.

Listing 63: Code for applying gravity to an object

```
const int GRAVITY_ACCELERATION = 10
2 const float MAX_FALL_VELOCITY = 500
  // ...
4 // Apply Gravity
  speed_y = speed_y + GRAVITY_ACCELERATION
6 // Cap the fall speed
  if speed_y > MAX_FALL_VELOCITY:
8     speed_y = MAX_FALL_VELOCITY
  // ...
```

17.2.2 Avoiding “Floaty Jumps”

The previous trick shows a physics-accurate jumping: if we plot the height against time, we would get something that represents the curve of jump like the following:

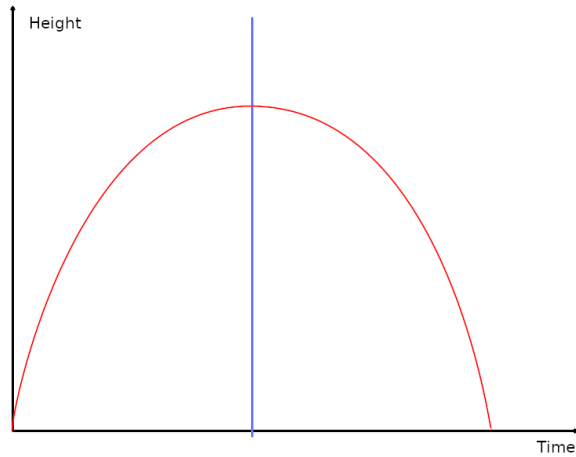


Figure 167: Plotting a physics-accurate jump

Although this can give the sensation that the character we’re controlling is “floaty”, which is not fun. In this case it’s a better idea to enhance gravity when falling, to give the character some more “weight”, which would be represented, more or less, by the following curve:

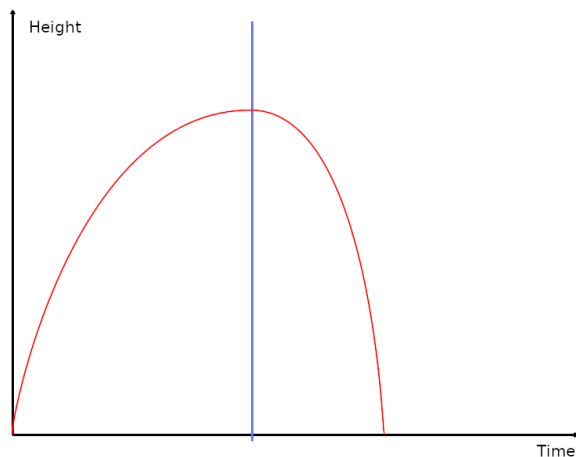


Figure 168: Plotting a jump with enhanced gravity

This can be obtained with few simple lines of code, not very different from the gravity example of earlier:

Listing 64: Code for jump with enhanced gravity while falling

```
const int GRAVITY_ACCELERATION = 10
2 const float MAX_FALL_VELOCITY = 500
const float GRAVITY_FALL_MULTIPLIER = 1.5
4 // ...
// Are we jumping?
6 if speed_y < 0:
    // Apply Gravity Normally
8     speed_y = speed_y + GRAVITY_ACCELERATION
    else:
10     // We're falling, enhance gravity
        speed_y = speed_y + GRAVITY_ACCELERATION * GRAVITY_FALL_MULTIPLIER
```

```
12 // Cap the fall speed
13 if speed_y > MAX_FALL_VELOCITY:
14     speed_y = MAX_FALL_VELOCITY
15 // ...
```

In this example we are assuming that the framework used uses the screen coordinate system, and jumping brings the player from bottom towards the top of the screen. If you want different behaviour (like gravity inversion in puzzle games), something a tiny bit more involved may be in order.

17.2.3 Ladders

[This section is a work in progress and it will be completed as soon as possible]

17.2.4 Walking on slanted ground

[This section is a work in progress and it will be completed as soon as possible]

17.2.5 Stairs

[This section is a work in progress and it will be completed as soon as possible]

17.2.6 Jump Buffering

A nice trick used mostly in 2D platformers to allow for smoother gameplay is “jump buffering”, also known as “input buffering”.

Normally when a character is mid-air, the jump button does nothing, in code:

Listing 65: Code for jumping without buffering

```
function update(float dt):
2     ...
    if controls.jump.isPressed():
4         if player.on_ground:
            // Jump
6     ...
```

Jump Buffering consists in allowing the player to “buffer” a jump slightly before the character lands, making the controls a bit less stiff and the gameplay more fluid.

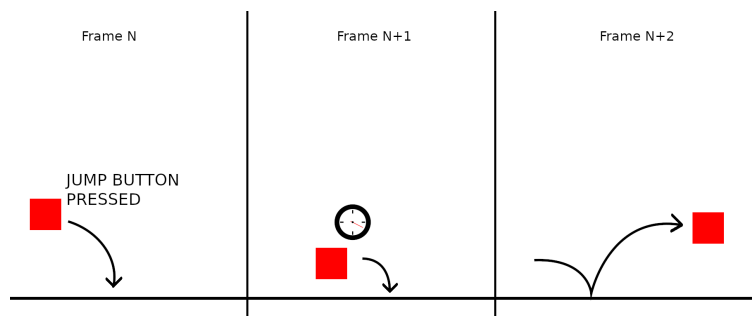


Figure 169: Example of how jump buffering would work

Jump buffering usually is put into practice using a timer, in a fashion similar to the following:

Listing 66: Jump buffering example

```
...
2  const float jumpBufferTime = 5.0
...
4  function update(float dt):
    ...
6      if controls.jump.isPressed():
            player.hasBufferedJump = True
8          player.jumpBufferCountdown = jumpBufferTime
            // Take note on how this piece is outside the "jump is pressed" section
10         if player.hasBufferedJump:
            player.jumpBufferCountdown = player.jumpBufferCountdown - dt
12         if player.on_ground:
            if player.jumpBufferCountdown > 0.0:
14             // Jump
                player.jumpBufferCountdown = 0.0
                player.hasBufferedJump = False
16         ...
    ...
```

17.2.7 Coyote Time

Coyote time (also known as “edge tolerance”) is a technique used to allow a player to jump a few frames after they fall off a platform, allowing for a more fluid gameplay.

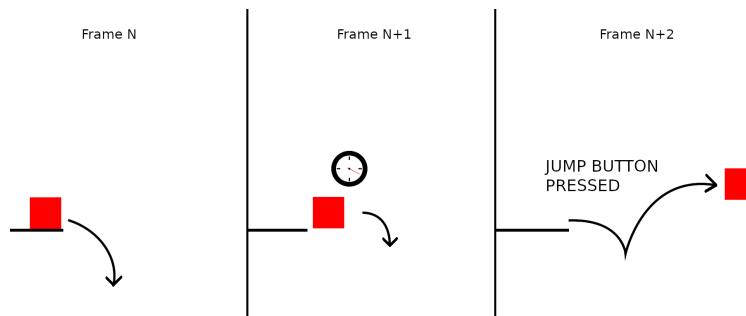


Figure 170: Example of how coyote time would work

The trick is starting a countdown as soon as the player leaves a platform without jumping, then if the player presses the jump button while that time is still going, they will perform the jump action, like they still were on a platform.

Listing 67: Coyote time code example

```
class Player:
2     coyote_time_started = False
        coyote_time = 0
4     onground = False
        has_jumped = False
6     // ...
    function update(dt):
8         // ...
            if onground:
                // Do stuff when player is on ground
                // ...
12        else:
            if not has_jumped:
14                // Player is not on ground and has not jumped, the player's falling
                    if not coyote_time_started:
```

```
16         coyote_time_started = True
17         coyote_time = 5
18     else:
19         coyote_time = coyote_time - dt
20
21     function jump():
22         // This function takes care of jumping
23         // ...
24         if coyote_time > 0:
25             // Do Jump
26         // ...
```

17.2.8 Timed Jumps

A way to extend the mobility and challenge of a 2D platformer game is allowing players to jump higher the more the jump button is pressed: this allows the character to perform low and high jumps without much effort, making timing the jump button press a variable that adds to the challenge of a game.

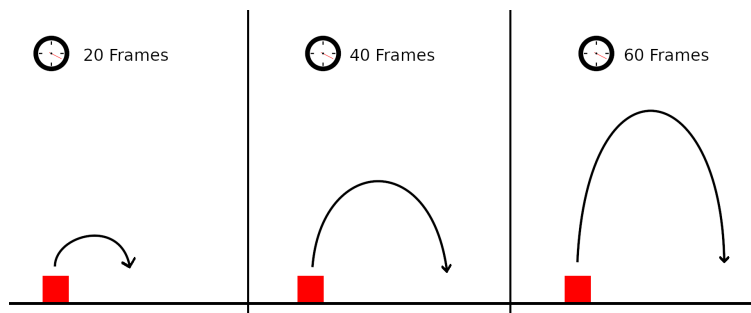


Figure 171: Example of how timed jumps would work

To work well, timed jumps need to be implemented by tracking the jump button's `onPress` and `onRelease` events. When the jump button has just been pressed, the character's `y` velocity will be set, as soon as the button is released, such velocity will be capped, shortening the jump height.

Listing 68: Example code of how timed jumps work

```
class Player:
2     const float JUMP_VELOCITY = -12.0
3     float y_speed
4     // ...
5     function onJumpKeyPressed():
6         /* The jump key has just been pressed (doesn't account the jump key being
7         pressed from previous frames) */
8         y_speed = JUMP_VELOCITY
9
10    function onJumpKeyReleased():
11        // The jump key was just released, cut the y_speed so the jump is lower
12        if y_speed < JUMP_VELOCITY / 2:
13            // The speed is higher than the cutoff speed (in absolute value)
14            y_speed = JUMP_VELOCITY / 2
```

17.3 Top-view RPG-Like Games

17.3.1 Managing height

[This section is a work in progress and it will be completed as soon as possible]

17.4 Rhythm Games

17.4.1 The world of lag

[This section is a work in progress and it will be completed as soon as possible]

17.4.1.1 Video Lag

[This section is a work in progress and it will be completed as soon as possible]

17.4.1.2 Audio Lag

[This section is a work in progress and it will be completed as soon as possible]

17.4.1.3 Input Lag

[This section is a work in progress and it will be completed as soon as possible]

17.4.1.4 Lag Tests

[This section is a work in progress and it will be completed as soon as possible]

17.4.2 Synchronizing with the Music

[This section is a work in progress and it will be completed as soon as possible]

17.4.2.1 Time domain vs. Frequency Domain

[This section is a work in progress and it will be completed as soon as possible]

17.4.2.2 The Fast Fourier Transform

[This section is a work in progress and it will be completed as soon as possible]

17.4.2.3 Beat Detection

[This section is a work in progress and it will be completed as soon as possible]

17.5 Match-x Games

17.5.1 Managing and drawing the grid

When it comes to a match-x game, a good data structure for the play field is a matrix.

In most programming languages, a matrix is saved as an “array of arrays”, where you have each element of an array representing a row, and each element of a row is a tile.

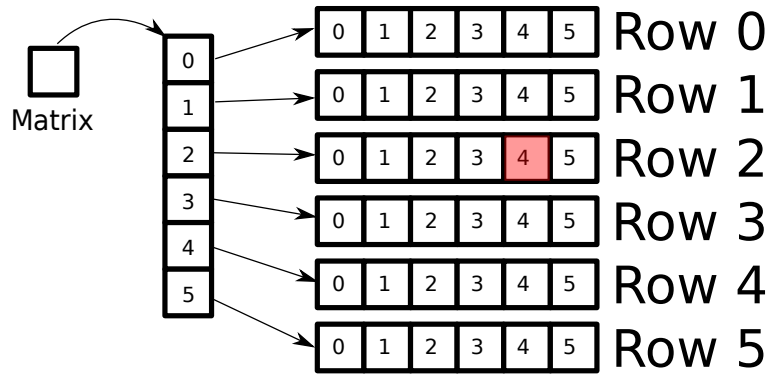


Figure 172: Example of a matrix, saved as “array of arrays”

This is a really nice way to interpret a grid, but at the same time it can open the door to some pitfalls if you’re not careful.

Many programming languages allow for direct access to an element inside an “array of arrays” by using multiple access operators (usually the `[]` operator) in a row.

Usually each element you access with the first `[]` operator represents the rows, while the second time you use `[]` you will access the columns, this will make it so you need to access an element directly as follows: `matrix[y][x]` where “y” is the row number and “x” is the column number, which can prove counter-intuitive.

In the previous example, if we want to access the highlighted item, at the third row (indexed at 2), and in the fifth column (indexed at 4). We have to use `matrix[2][4]`, which is the opposite of what many people are used to when they think in (x,y) coordinates.

Try to keep visual representation and data structures separated in your mind, to avoid confusion.

17.5.2 Finding and removing Matches

If you’re doing a simple match-x game where you can only match tiles horizontally or vertically, the algorithm to find matches is quite simple, both conceptually and computationally.

The main idea is dividing the “horizontal matches” from the “vertical” ones. This will allow to simplify the algorithm and avoid some pitfalls (unless you want to give bonuses for “T-shaped” and “L shaped” matches).

For horizontal matches the idea is running through each row, keeping some variables representing the length of the match, as well as the color of the current “ongoing” match. As soon as we find a different color, if the length of the “ongoing” match is higher than “x” (usually 3), we save the references to the tiles involved for later removal.

Similarly we can do the same algorithm for vertical matches, by running through each column and saving the matches.

Here is a pseudo-code example:

Listing 69: Finding horizontal matches in a match-3 game

```
function findHorizontalMatches():
2   matchLength = 0
   minMatchLength = 3
4   for each row in matrix:
       lastMatchingTile = null
6       for each column in row:
           currentTile = matrix[row][column].tile
8           if currentTile == lastMatchingTile:
```

```
        matchLength = matchLength + 1
10    else:
        if matchLength >= minMatchLength:
12            // We need to memorize all the tiles involved in the match
            for each tile from matrix[row][column-matchLength] to matrix[row][
column]:
14                memorize(tile)
        else:
16            // No matches, reset the counter and set the current tile as last
            matching
                matchLength = 1
                lastMatchingTile = currentTile
18            // We need to account for the right-hand border corner case
            if column == size(matrix[row]):
20                if matchLength >= minMatchLength:
22                    // We need to memorize all the tiles involved in the match
                    for each tile from matrix[row][column-matchLength] to matrix[row][
column]:
24                        memorize(tile)
```

Let's talk a second about the last rows in the algorithm: they are specifically tailored to address a corner case that happens when there is a match that ends on the right border of the screen.

If such code was not there, the match number would grow by one, then the for loop would reset everything and we'd lose such match.

Similarly, we can make an algorithm that allows for vertical matches to be memorized for later removal:

Listing 70: Finding vertical matches in a match-3 game

```
function findVerticalMatches():
2    matchLength = 0
    minMatchLength = 3
4    for each column in matrix:
        lastMatchingTile = null
6        for each row in column:
            currentTile = matrix[row][column].tile
8            if currentTile == lastMatchingTile:
                matchLength = matchLength + 1
10        else:
            if matchLength >= minMatchLength:
12                // We need to memorize all the tiles involved in the match
                for each tile from matrix[row-matchLength][column] to matrix[row][
column]:
14                    memorize(tile)
            else:
16                // No matches, reset the counter and set the current tile as last
                matching
                    matchLength = 1
                    lastMatchingTile = currentTile
18                // We need to account for the bottom border corner case
                if row == size(matrix[column]):
20                    if matchLength >= minMatchLength:
22                        // We need to memorize all the tiles involved in the match
                        for each tile from matrix[row-matchLength][column] to matrix[row][
column]:
24                            memorize(tile)
```

Both algorithms run in $O(n)$, where “n” is the number of tiles on the screen.

Now we can proceed to remove every tile that has been memorized as “part of a match”, the quickest way may be to set such tile to “null” (or an equivalent value for your programming language).

17.5.2.1 Why don't we delete the matches immediately?

We could, but that would open the door to a pitfall that could be tough to manage: in case of a “T” match, we would find that the “horizontal matches” algorithm deletes part of said match, and the “vertical matches” algorithm wouldn't be able to complete the “T match”, because the necessary tiles are deleted.

Let's see the image to understand better:

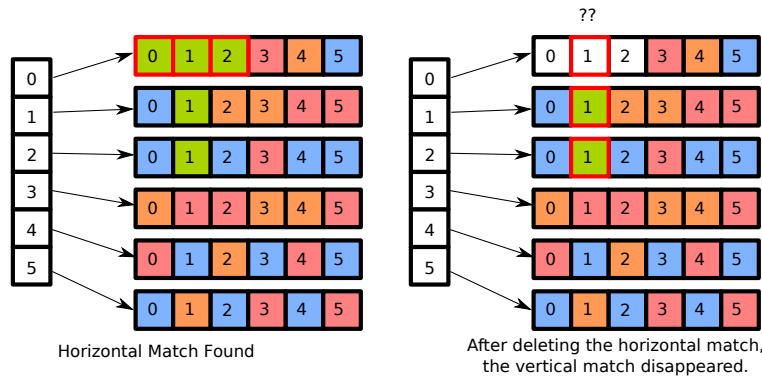


Figure 173: What happens when deleting a match immediately

As visible from the first image, there is a T-shaped match involving cells 0,1,2 of row 0, cell 1 of row 1 and cell 1 of row 2.

If we deleted the horizontal match immediately, we would lose the possibility of completing the vertical match (highlighted in the second image).

Instead we memorize everything first, and then delete all the matches at once, without the risk of losing anything.

17.5.3 Replacing the removed tiles and applying gravity

At this point, it is easy to make the “floating tiles” get into the right position: the hardest part is taking care of the graphics inbetweening that will give us that “falling effect” that we see in many match-x games.

After the graphics tweening, we need to create the new tiles that will go and fill up the holes that have been created by our matches and moved tiles.

After creating the tiles and tweening them in place, it will be necessary to check for more matches that have been created from the falling tiles (and eventually notify some kind of “combo system” to apply a “score multiplier system” or even an achievement system using the [Observer Pattern](#)).

[This section is a work in progress and it will be completed as soon as possible]

17.6 Cutscenes

When you want to advance your storyline, a great tool is surely the undervalued cutscene. The game temporarily limits its interactivity and becomes more “movie-like”, making the storyline go forward. Cutscenes can be scripted or just true video files; in this chapter we will analyze the difference between the two, advantages and disadvantages of both and how to implement each one, from a high-level perspective.

17.6.1 Videos

The easiest way to implement cutscenes in most engines and frameworks, is to use videos. Many frameworks have native support for reproducing multimedia files with just a few lines of code, which makes this the

preferred choice when it comes to the code.

The bad thing is that videos are a “static” format. They have their own resolution, their own compression and characteristics, this means that when a video is presented at a higher resolution than its own native one, we’re bound to have artifacts due to upscaling.

[This section is a work in progress and it will be completed as soon as possible]

17.6.2 Scripted Cutscenes

[This section is a work in progress and it will be completed as soon as possible]

18 Testing your game

When you want to assure the quality of your product, you need to test it. Testing will be probably be the bane of your existence as a developer, but it will also help you finding bugs and ensure that they don't come back.

Remember that you should be the one catching bugs, or your players will do that for you (and it won't be pretty).

Let's take a deep dive into the world of testing!

18.1 When to test

18.1.1 Testing “as an afterthought”

Most of the time, testing is left “as an afterthought”, something that comes last, or at least something that is done way too late in the development process.

Testing is made difficult because the code is all interconnected, making it really hard (if not impossible) to separate and test each component.

Leaving testing as an afterthought is not a good idea, let's see the alternatives.

18.1.2 Test-Driven Development

Some swear by the “Test-Driven Development” (from now on “TDD”) approach. TDD consists in developing the test before writing a single line of code, then after the test is ready, create the code that solves that test.

That way the test will “drive us” to the solution of the problem. At least that's what it is supposed to do.

As a critique to the TDD approach, I personally think that people will end up trying to “solve the test”, instead of “solving the problem”. This means that sub-optimal solutions may be adopted, and “edge cases” will be missed.

18.1.3 The “Design to test” approach

To get the best of both worlds, we need to work a bit more on our software design, by designing by having its testing in mind: functions should be self-contained, the weakest amount of coupling should be present (if any) and it should be possible to **mock** elements easily.

Paying attention and going the extra mile to create an architecture that is easy to test will reward you in the long run.

18.1.4 You won't be able to test EVERYTHING

Before trying to test anything, remember that testing is **hard**. You won't be able to test everything, and the situation will be worse when you will be in a time crunch to deliver your game.

Remember that hacking is not always a bad thing, sometimes cutting corners will get your game shipped and having a solid (and tested) basis will help you with that too.

18.2 Mocking

Before talking about the nitty-gritty of testing, we need to talk about “Mocking”.

Mocking is a procedure that is usually performed during tests, that substitutes (in-place) an object or function with a so-called “mock”. A mock is something designed to have no real logic, and just return a pre-defined result, or just have a pre-defined, very simple, behaviour.

Mocking will help you “detaching” objects that depend on each other, substituting such dependencies with “puppets” (the mock objects) that behave consistently and are not affected by bugs that may be present in the object that you are mocking.

18.3 Types of testing

Let’s take a look at how we can test our game, because (as with many things in life), testing can be more than meets the eye. In this section we will talk about both manual and automated testing, the difference between them and what method suits what situation.

18.3.1 Automated Testing

Automated testing is usually performed when new code is pushed to the repository, or on-demand. Automated testing makes use of a “test suite”: a bunch of tests that are run on the game’s elements to test their correctness.

Here we can see a small example of a simple function, along with a test, in pseudo-code:

```
function sum_two_numbers(a, b):
2   // This function takes two numbers and sums them
   return a + b
4
function test_sum():
6   // This function tests the sum function
   int result = sum_two_numbers(2, 2)
8   assert result == 4
```

As we can see, the test makes use of the “assert” statement, which in many languages raises an error when the “assert” is false. This means that if, for some reason, the `sum_two_numbers` function was edited to return “2+2=5”, an exception would be raised when the test is run.

Care should be taken when making automated tests: they should be as simple as possible, to avoid the presence of bugs in the tests themselves, also, like all code in the world, it’s subject to its own maintenance: you should thoroughly comment your tests, and if the tested component changes, the connected test should change too, or it may fail.

18.3.2 Manual Testing

Sometimes also called “play testing”, this is usually divided in time periods, where more and more people participate:

- **In-house testing** with a small team of dedicated testers;
- **Closed Beta** where a small number of selected players is able to test the game, report bugs and issues. Usually at this stage, the game is already mostly finished and playable;
- **Open Beta** similar to the closed beta, but players can freely subscribe to play the game.

We will talk specifically about each one of these test types in detail in the following sections.

18.4 Unit Testing

Unit Testing takes care of testing the smallest component possible inside your system: that usually means a single function. Such “units” must be separated from all their dependencies (via **mocking**) to avoid bugs in their dependencies to interfere with our testing efforts.

Many programming languages have their own unit testing frameworks, among the most used:

- **Python:** unittest (in the standard library);
- **Javascript:** unit.js;

- **C++:** Boost Testing Library;
- **Java:** JUnit;
- **Lua:** luaunit.

Remember: during unit testing, you need to make sure that the unit that you're testing has its dependencies mocked, or the results of your test will depend on the performance and correctness of said dependencies.

18.5 Integration Testing

Integration testing is a step further from “unit testing”, here you take different “units”, put them together and check how they behave as a group.

There are many approaches to integration testing, the most used are:

- **Big bang integration:** The majority of the units are put together to form a somewhat complete system (or at least a major part of it), after that the integration testing starts. This can lead up to an “explosion of complexity” if the results are not accurately recorded.
- **Bottom-up:** Test the “lowest level” components first, this should help testing the “higher level” ones. After that you test the components that are “one level above” the previous ones. Keep going until the component at the top of the hierarchy is tested.
- **Top-down:** Opposite of the previous approach, you test the integrated modules first, then you branch into the lower-level modules, until reaching the bottom of the hierarchy.
- **Sandwich Testing:** a combination of Bottom-Up and Top-Down.

18.6 Regression Testing

Regression testing is a bit of an outlier in our “specific to general” testing structure, but that's because their objective is different.

Regression testing (sometimes also called *non regression testing*, you'll see why) is used to avoid our software from regressing into previous bugs.

This means that every time you find a serious bug in your software, you should fix it and make a test that will check for you if said bug is resurfacing.

With time, bugs and regression tests will accumulate, which usually means that automation is involved (like continuous intergration and delivery).

18.7 Playtesting

[This section is a work in progress and it will be completed as soon as possible]

18.7.1 In-House Testing

[This section is a work in progress and it will be completed as soon as possible]

18.7.2 Closed Beta Testing

[This section is a work in progress and it will be completed as soon as possible]

18.7.3 Open Beta Testing

[This section is a work in progress and it will be completed as soon as possible]

19 Profiling and Optimization

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

Donald Knuth - Computer Programming as an Art

19.1 Profiling your game

19.1.1 Does your application really need profiling?

In this section we will have a small check list that will let us know if our videogame really needs to be profiled. Sometimes the FPS counter is trying to tell us a different story than the one we have in our heads.

19.1.1.1 Does your FPS counter roam around a certain “special” value?

There are cases where the FPS counter shows a low counter, but it stays around a certain value. This means that the FPS value is artificially limited somewhere, either by VSync or something else.

Some special values you may see are:

- 25 FPS: PAL refresh rate
- 29.970 FPS: NTSC Refresh Rate
- 30 FPS: Used in some games
- 50 FPS: Used in some games
- 60 FPS: Used in most games
- 75 FPS or 80 FPS: Used in some LCD Monitors
- 144 FPS: Used in more modern, high-refresh rate monitors

19.1.1.2 Is the animation of your game stuttering but the FPS counter is fine?

If your animation stutters or its speed varies according to the load of your platform but your FPS counter is still stuck at the maximum allowed framerate, you may have forgotten to tie the animation to the delta-time in your game loop. Check the [timing your game loop](#) section for more information.

[This section is a work in progress and it will be completed as soon as possible]

19.1.2 First investigations

First of all, we need to understand what is the bottleneck of your game: check your task manager and see how your game is performing.

19.1.2.1 Is your game using 100% of the CPU?

Is your game using 100% of the CPU (if you're on Linux, you may see percentages over 100%, that just means your game is using more than one CPU core)?

First of all, you should check if you're using the frame limiting approaches offered by your framework or game engine: if they're not active, your game will run “as fast as possible”, which means it will occupy all the CPU time it can. This can result in high FPS count (in the thousands) but high energy consumption and slowdowns in other tasks.

If you have taken all the frame limiting approaches as stated above, that may mean that the game is doing a lot of CPU work and you may need to make the game perform less work for each frame. In this case profiling

tools are precious to find the spots where the CPU spends most of its time: Valgrind or GProf are great profiling tools.

If instead your game is not using all of the CPU computing power, you may have a problem on the GPU: your game may be calling the drawing routines too often. The less a game has to communicate with the hardware, the higher the performance. In that case using Sprite Atlases and other “batching techniques” that allow to draw many objects with only one call will help your game perform better.

[This section is a work in progress and it will be completed as soon as possible]

19.2 Optimizing your game

After accurate profiling, you need to intervene and try to get more out of your code. In this section we’ll talk about some guidelines and tips on how to optimize your game.

19.2.1 Working on references vs. returning values

Depending on the programming language you’re using, and the amount of internal optimization its compiler/interpreter has, you may have the possibility to choose between two main ways of working, when it comes to functions:

- Returning a value from a function;
- Passing a reference to variables into the function and use that reference in your function (for instance in C++).

“Value Copying” can be a real resource hog when your functions work with heavy data. Every time you return a value, instead of working on a reference, you are creating a new copy of the data you’re working on, that will be later assigned.

This can happen also when passing parameters to a function (in this case you say the “parameter is passed by value”): a new copy of the parameter is created locally to the function, using up memory. “Value Copying” can help when you don’t want to modify the data outside your function, but is a waste when instead you **want** to modify such values.

Using things like “references”, “constant references” and “pointers” can be really precious in making your game leaner memory-wise, as well as saving you all the CPU cycles wasted in memory copying.

19.2.2 Optimizing Drawing

This heavily depends on the type of framework and engine you are using, but a good rule of thumb is using the lowest amount of calls to the draw routines as possible: drawing something entails a great amount of context switching and algorithms, so you should do it only when necessary.

If your engine/framework supports it, you should use sprite atlases/batches, as well as other interesting structures like Vertex Arrays (used in SFML), which can draw many elements on the screen with only one draw call.

Another way to optimize your drawing routine is avoiding to change textures often: changing textures can result in a lot of context changes (like copying the new texture from the RAM to the GPU memory), so you should use only one oversized texture (in the form of a **Sprite Sheet**) and draw only a part of it, changing the coordinates of the rectangle that gets drawn. This way you’ll save the PC a lot of work.

[This section is a work in progress and it will be completed as soon as possible]

20 Balancing Your Game

The trick to balance is to not make sacrificing important things become the norm

Simon Sinek

An imbalanced game is a frustrating game, and most of the time balancing a game is one of the toughest challenges a game developer/designer can find themselves to have to face.

Let's talk about some principles and guidelines that can help you balancing your game and keep your players challenged but not frustrated.

20.1 The “No BS” principle

The “master principle” everyone should follow (in my humble opinion) is what I call the “no BS” principle.

You should not trade the “fun” of your game for any other mechanic (like showing an advertisement to allow them to continue playing), that is equivalent to betraying your player, makes the game feel unfair and un-fun.

Here are some examples of mechanics that break the “no BS” principle:

- **Sudden spikes in difficulty:** when you have a sudden spike in difficulty, the player feels stumped and the game tends to lose its charm, you are “interrupting the flow” of the game by placing an arbitrary hurdle on your players’ road;
- **Off-screen instant-death traps:** having something deadly that pops out from off-screen and kills the player is unfair and will make your players scream “that’s BS!” all the time, if you want to place some obstacles that pop from off-screen you should “telegraph” them. “Telegraphing” is a technique where you send a warning signal to the player that danger is coming. For instance a huge laser that instantly kills you should be preceded by a couple seconds by a yellow “!” signal on the right side of the screen, where the laser is due to strike. Another way to telegraph said laser would be to illuminate the part of the screen that is about to be stroke, like the light of the laser is coming up;
- **Arbitrary invisible time limits:** If you suddenly interrupt the player’s game with a “time up” and you have no countdown on the screen, the player will get frustrated, that’s for sure;
- **Taking control away from the player:** Not allowing the player to move (getting blocked by an enemy and killed) or just not allowing the player to adjust their jump mid-air is a surefire way to make them not play your game anymore.

20.2 Always favour the player

In the process of balancing a game, as a game developer/designer you will surely find yourself in front of the following decision time and time again:

■ Shall I favour the game’s precision or should I give some leeway to the player?

The answer is always the latter.

Giving some leeway to the player, for instance by having a more generous hit-box that allows you to stay alive even if a bullet grazes your character makes the game seem more “fair”.

There are infinite ways to make a game challenging without having to force the player into accepting very precise hit-boxes or extremely tight gameplay.

20.3 A primer on Cheating

Cheating is the act of fraudulently altering the game’s mechanics in advantage of the player, performed by the players themselves.

It is something that many game developers and designers have to battle against, so here are some suggestions and tips to limit cheating in your game.

20.3.1 Information-based cheating

Information-based cheats are all those cheats that rely additional information to the cheater, such information can give a sizeable advantage. A possible example is a cheat that removes the so-called “Fog of War” in a real-time strategy (RTS) game: having possibility of seeing all the enemy units allows the cheater to put up some countermeasures against the units that are being created.

These cheats include also x-ray hacks, all cheats that invalidate invisibility (as the server or peer would still need to transmit the coordinates of the hidden unit) and anything that can show information that is not meant to be shown to the user.

A possible solution is for the game to just “not transmit” the data, making the cheat useless, but sometimes that is just not possible.

20.3.2 Mechanics-based cheating

Another category of cheats is comprised of all those hacks that alter the game mechanics themselves, like killing all the players on the map. These kind of cheats are usually made possible by exploits or just because the cheater owns the server people are playing on.

These kinds of cheats can easily hinder the playability of a game, or even make it outright unplayable.

A possible solution to these cheats would be using a cheat-detection program (which would start a “cat and mouse” game, where hacks are updated to avoid detection, and detection programs are updated to detect new hacks) and also inserting some client-side verification of server commands (in case the server contains the “authoritative game state”); for instance if all players are killed at the same time, the clients could flag the server as possibly cheating.

20.3.3 Man-in-the-middle

This attack is well known in the networking environment: an external machine is used to route and intercept all the traffic directed to the game. This can be a real issue since the attacking program is “outside of the game environment”, making nearly all cheat-detection programs useless.

A man-in-the-middle attack can also be used to further exploit the game and find new vulnerabilities.

A possible solution could be completely encrypting all the game’s traffic, but that will be an issue since encryption takes away precious CPU cycles, and this could lead to an hindered gaming experience.

20.4 How cheating influences gameplay and enjoyability

20.4.1 Single Player

Cheating in single player is an act that doesn’t usually do a massive amount of damage, and such damage is usually confined inside the single “single-player” game.

Playing outside of the rules can be really fun (that’s one of the principles the “glitch hunters” love: doing something outside of what another person imposed them), for instance some people cheat in games to bring some mayhem into their gameplay, or they use cheats implemented inside the game itself for a comedic factor (like the omnipresent “giant head” cheat).

Sometimes cheating happens because the game is unbalanced and breaks the “no BS” principle, an instance of this happening could be when a game has a great story and gameplay but there is a boss that is so hard the game just stops there. You want to see how the story continues, but the game has gone so much out of balance you are willing to break its own mechanics to be able to continue it.

In this case the approach you should have is rebalancing the game, instead of limiting your players.

When it comes to cheat prevention, usually the first order of action is giving the game the ability to “check the validity” of an instruction.

For instance if a player character has its coordinates at (5,5) on frame n and coordinates at (1500, 5) at frame $n + 1$, there is something fishy going on, since maybe the player can only move 500 pixels per second (while it moved 995 in one frame: $\frac{1}{60}$ of a second).

Such checks will slow down the processing, but will allow you to put a limit to cheating, possibly intervening in an active way, by resetting the space walked to the maximum amount possible in one frame, although this could give some issues with slower computers and **variable time steps**.

20.4.2 Multiplayer

When it comes to multiplayer and “leaderboards”, cheating can create some major damage to the game’s enjoyability. It is honestly disheartening seeing a level that has been completed in 0 seconds on top of the leaderboard, totally unreachable with normal gameplay.

When competitive gameplay comes into the picture, playing against a cheater is frustrating and maddening, you feel powerless, the game is not fun and sometimes it even feels “broken”, even though it is stable and playable.

Here we will distinguish between the two main forms of multiplayer: Peer-to-peer gameplay and dedicated servers.

20.4.2.1 P2P

Peer-to-peer multiplayer is the economically cheapest and easiest way to implement multiplayer, two or more computers (or consoles) are on “the same level” and communicate directly with each other, without a tertiary server in the middle.

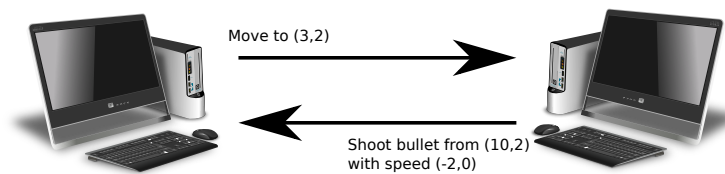


Figure 174: Example of a P2P connection

The main difficulty in preventing cheating is that there is no “authoritative game state”, the program cannot know if either player is cheating besides having an array of “possible actions”, like in single player, but with the added difficulty of network lag.

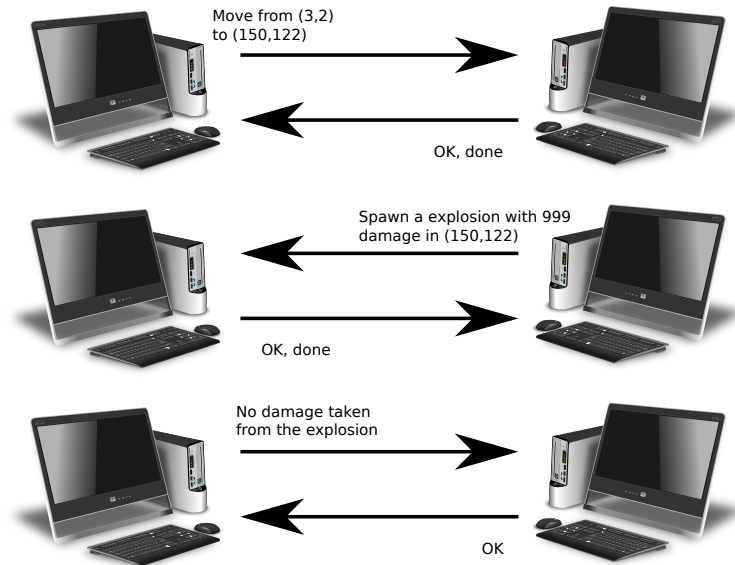


Figure 175: Two cheaters meet in P2P

Giving such “authoritative game state” to either of the players is not a good idea, because that way they would be able to cheat themselves and since they’re the “game master”, everything they do would be accepted.

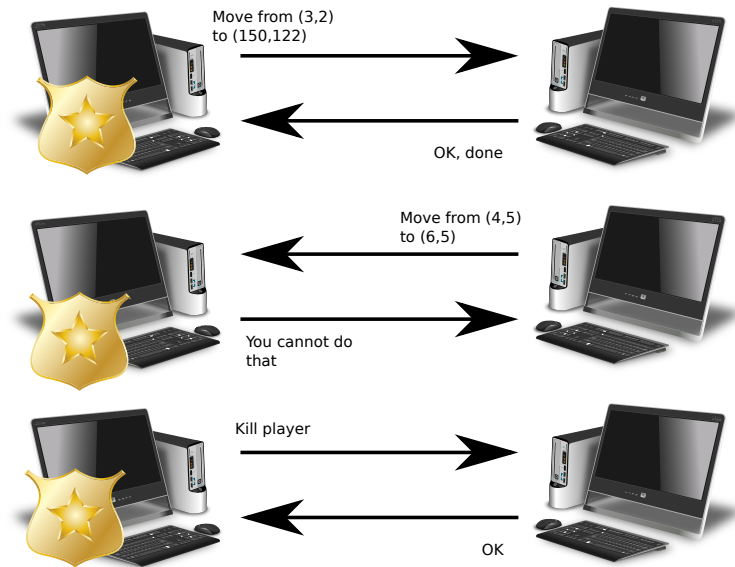


Figure 176: What would happen if one of the Peers had the autoritative game state

This is also the reason why many games that make use of P2P connections have implementations of anti-cheat systems that are shoddy at best.

20.4.2.2 Dedicated Servers

Dedicated servers is usually the best way to prevent cheating, a tertiary server is added to the mix, and said server is either controlled by the game creators or uses a software specifically tailored to work as a “multiplayer

server”.

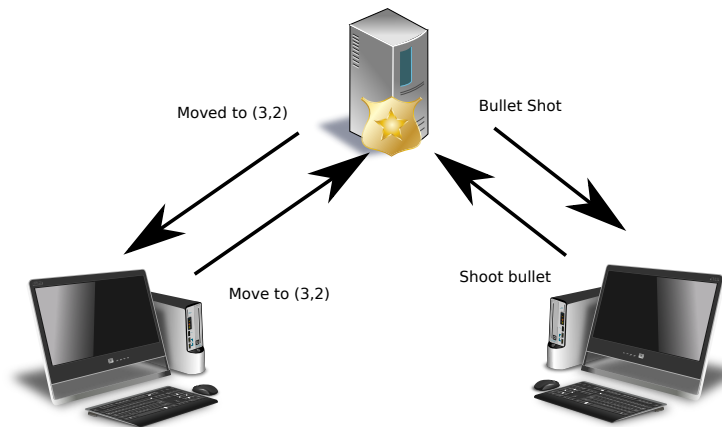


Figure 177: Example of a dedicated server

Such server contains the authoritative game state and decides what is right and what is wrong, or either what is possible and not possible.

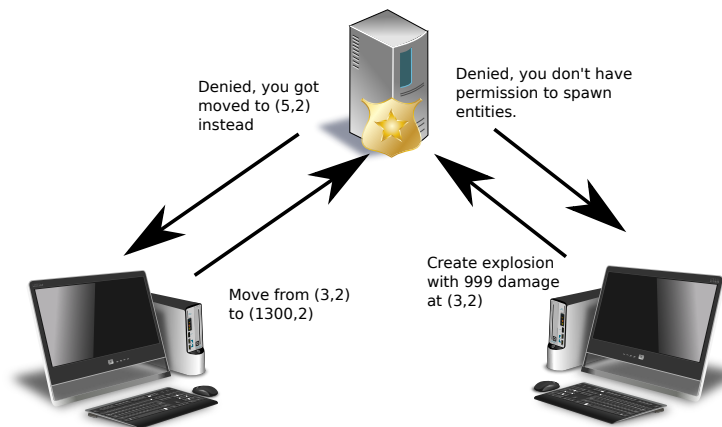


Figure 178: Two cheaters vs. Dedicated server

Usually a dedicated server software has been specifically made to limit cheating, as well as offering better performance than the P2P counterpart (it doesn't have to run graphics, for instance).

If a consistent leaderboard and lack of cheating is important (for instance in a heavily competitive multiplayer game), you should probably choose this option.

This section assumes that the game is using a third-party game server, and none of the players has direct access to said server, as this would enable cheating on the owner's part.

21 Marketing your game

Here's my whole marketing idea: treat people the way you want to be treated.

Garth Brooks

21.1 The importance of being consumer-friendly

We live in a “Money-Driven” world. This is a fact, not an opinion. So, leaving morality out of the discussion, you can argue that the general game publisher mentality of “getting all the money” is not that wrong. But you don't have to be an idealist to realize that we should see the world for what it could be, and not for what it is at the moment. We should apply this mentality to every aspect of our world, game industry included.

We are NOT here to enlighten you about how game industry has to change, but every game developer should realize that the true success of a game is **not** based on sales, it is based on customer satisfaction. So, even if this cursed “publisher mentality” could be applied to small indie developers (spoiler: it is not), we have to fight it back, and restore the “customer satisfaction above all” philosophy. So, fun fact: the only thing we (small indie developers) can do is *doing the right thing*.

Focus your effort on customer satisfaction, you have to be consumer-friendly!

My first advice is: instead of implementing a thousand features, make one single thing a thousand times better (Bruce Lee style). If you promise the client a ton of features to do you can generate hype (we will discuss it later), but if you are an indie developer (and most of times even if you're not) making one thing extremely enjoyable is way better.

Why? Because your goal is to create something (even if it's only one thing) that will make the customer remember your game: Quality over Quantity.

Satisfying customers is no easy feat, we all know this. So one question you may ask is: How in hell can I be original? Answer is: You don't have to.

So, my second advice: There are million of games out there, so creating something never seen before is very, very, very difficult. So, try to innovate what already exists, make usual features in your own way. This is how sub-genres come to life.

You think this is not the right way? Then go and tell the people from *From Software* that the “Souls-like” sub-genre was not innovating.

Last advice: Gameplay over Graphics.

We're not Activision, we can't afford to spend 10 million bucks for a cutscene. Aim for the “old fashion gamers”, the customers that play for fun, not because some game is gorgeous to see. Whatever your working on, whatever your game genere is, focus on making the gameplay fun to play. Things are really this simple. Work 2 hours less on a model or sprite and 2 hours more on thinking about how to not get your customer bored.

21.2 Pricing

If you want to sell your game, *Pricing* is one of the 4 “P”s of marketing you should pay attention to (along with “Place”, “Product” and “Promotion”), here you will find some tips on how to price your game.

21.2.1 Penetrating the market with a low price

The video games market is quite large but also quite saturated, and it is really hard to stand out without a competitive visuals, mechanics and most of all: price.

Penetrating the market is easier when done with a low price, even more when your game does not boast “innovation” as one of its own strong points. Usually “retro” titles benefit from this pricing model.

After the initial “cheap” phase, you should be able to recover your investment by selling “premium” products later (like higher-budget games). The main objective of this strategy is to create the so-called “brand awareness”, in short: letting yourself get known.

21.2.2 Giving off a “premium” vibe with a higher price

When instead your game has something more (like cutting-edge, never seen before graphics capabilities), you may be able to keep a high price, giving off the vibe that your product is “superior” to its competition.

This can also work with products that offer really innovating concepts, and rarely works with “run of the mill” games that you see almost anywhere.

21.2.3 The magic of “9”

There is something psychologically magic about the number 9. When you price something at \$0.99 instead of \$1.00, it looks like you’re giving away the product for almost-free. Same goes for pricing your game \$39.99 instead of \$40.00, the price will look a lot closer to \$30.00 than \$40.00, even though the difference to \$40.00 is just 1 cent.

21.2.4 Launch Prices

Another way to create product awareness is setting a “launch price”: a discounted price for the period the game is launching (for example a week), after that the game’s price goes back to its “listing price”.

This will ensure a quick influx of players (even more if you built hype well) that want to get your game for a bargain.

A variation could be having a low-priced base game, which then can be monetized with Downloadable Content (DLC) that add more to the base game.

21.2.5 Bundling

Another way to entice players to buy your product is “bundling”, that is “offering more for the same price”. If your game has a really notable soundtrack, a good idea could be selling such soundtrack along with the game (which could help you pay your composers), or maybe a digital version of an artwork book, or sketches that can give an insight on the development.

Other types of bundling can include other games that you produced, or being part of a bundle of games of the same genre. An instance of “bundling” could be the famous “Humble Bundle”, which bundles games around a certain theme or for a defined occasion (like Christmas Holidays).

It could be interesting to offer flexible prices on your bundles, like the following:

- Base Game: 9.99\$;
- Base Game + Soundtrack: 14.99\$;
- Base Game + Artwork Book: 15.99\$;
- Base Game + Soundtrack + Artwork Book: 18.99\$.

This will allow your players to choose which “extras” they want to buy, making them more comfortable in the process.

21.2.6 Nothing beats the price of “Free” (Kinda)

Free to play games are all the rage these days, you can play a great game for free, if you’re willing to resist all kinds of microtransactions and premium DLCs.

That is actually one of the problems with Free to Play games: you must not get “too naggy” about microtransactions, or your player will get annoyed and stop playing. Also your game must still have enough qualities to be played: if your game is evidently low-quality, no one will play it, not even if it’s free.

21.3 Managing Hype

It’s time to discuss about promoting our works.

Let’s start with an obvious (sadly not that much obvious) statement: We are **not** a big publisher, we cannot afford to make big mistakes.

Truth is that companies like EA, Ubisoft, Square Enix, ActiVision (and many more) can do pretty much everything they want. Why? Because they know that a massive part of their audience will buy the product “sight unseen”, no matter what.

We are not in this situation, so we have to be very cautious. We must be aware that bad moves (not mentioning illegal ones) lead to major consequences, and most of the time indie developers cannot pay this price.

Most important thing to keep in mind is: never promise without keeping your word. This is very important, especially if you’re making people pay and the game is not complete yet (so-called “Early Access”).

You’re not a big publisher, people don’t trust you and will not pay you for a promise. You have to give them something worth the price, immediately enjoyable. Overpricing your game and justifying that because of “future release/contents” is NOT going to work either.

Patreon (or crowdfunding websites in general) could be an interesting compromise for such situation, but you have to use it in the right way, and never abuse your supporters trust. Keep in mind all the basic tips for managing hype we just discussed, and make sure to respect promised releases/updates when related to a certain amount of income from your crowdfunding (so-called crowdfunding “goals”).

21.4 Downloadable Content

DLC changed the game industry, this is a fact. How? You may ask.

The answer is very simple: it’s an excellent way to extend a game’s lifetime, especially for a single-player games. They are both a way to keep people playing a game (or make people come back) and, at the same time, earning money.

As always, there is a good way to do things, and a very bad way.

21.4.1 DLC: what to avoid

In my opinion there are two major behaviors to avoid, besides the obvious “Do NOT overprice”. I’ll illustrate them, showing some examples.

1) Putting the “true” ending in a DLC This behavior is almost insulting for the customer. Creating a game and put its ending in a DLC is equal to sell an unfinished product, this is the truth. For a “casual fan” it may not matter much, but it’s very annoying for a true fan of the game/saga. Why? Because you are implicitly forcing him to pay twice: The game and its DLC. Some major game companies exploit this mechanic to simply cut off some content of the original game and put it in a DLC: same effort and more income. I found the pair EA/BioWare to be the major example for this behavior with Dragon Age Inquisition and its DLC “The Trespasser” comes immediately to mind.

2) Selling a Season Pass “on trust” With this behavior you’re demanding a lot from your customers, things can go really wrong. To put it simple you’re selling a Season Pass without presenting the DLC. Ideally if you’re selling a season pass from day 1 than you should, at least, let the audience know about the number of future DLC and their basic content/quality/quantity. So what happens if an over-hyped game is released alongside its season pass and the whole thing is quite a flop? That’s the case of FFXV. Square Enix made

quite the gamble, and it really didn't pay off. But Square Enix is Square Enix, so it won't go bankrupt any time soon.

21.4.2 DLC: what to do

The Golden Rules are:

- Limit DLC to **additional content**. Related to the main story, but not concerning major twists or the finale itself;
- Create the DLC with an adequate playtime. If your game is an RPG with over 60 hours of content you simply cannot make a 5-hour long DLC;
- Sell a Season pass only if you're going to publish at least 3 distinct DLCs, or two very extensive ones;
- Obviously make the customer pay a fair price.

I find CD Projekt RED the absolute best company when it comes to DLCs. In particular "The Witcher 3 Wild Hunt"'s two DLCs have all the good qualities that we want.

The adequate content, a fair price and we're talking about a combined 50 hours of extra game time.

21.5 Digital Rights Management (DRM)

Another thing that can make or break a game's enjoyability is the presence of a Digital Rights Management System (shortly referred to as "DRM").

A DRM is a piece of software that silently sits in the background of a game to verify its authenticity, thus limiting piracy for the first months of the game being released.

A question comes up to mind: Why would an invisible piece of software influence a game?

There are some major reasons why a DRM System could drive away customers:

- The DRM interacts with the game in an annoying way;
- The DRM behaves outside of its scope;
- (An extension of the previous point) The DRM is known to spy on its users.

We have cases of "DRMs interacting with the games badly" left and right, for instance the "always online" DRMs that stop you from playing the game if your connection drops for a second, or some famous instances where more DRM Systems were stacked on top of each other, slowing down the game's performance.

This brings to something that many find annoying and unfair: people pay to play an original game and are hindered by DRM, while pirates are free from such hindrance for the fact that they pirated the game. You paid for the game, and you pay the price for other people pirating it, in the form of frame drops, slowdowns and sometimes flat out unplayability.

Another thing players hate is having a piece of software that acts out of its scope, or even worse as a *malware_g*.

There have been cases of DRM systems installing themselves as *rootkits_g*, making it impossible for the player to uninstall them, even after removing the game said DRM was tied to.

For instance that's what happened with the free game "Sonic Gather Battle", a tournament fighter fan game starring the beloved Blue Hedgehog. This game contained a piece of software that would track your browser tabs: if you searched for cheats/mods/hacks, it would close your browser and enable a "protection stage", making the game impossible to play, by adding immortal enemies that would one-shot you.

At the same time such DRM would have raw disk access and would also search for Cheat Engine installations. To top it off, the game would send the computer IP to a server to be saved in a list, so that the "protection stage" would still engage, even after a fresh install.

Some research found out that the game would sometimes require administrative access to the system, which shouldn't usually happen.

The bottom line is: a DRM doesn't always guarantee more people will play your game, so if you have to implement a DRM system, **do it well**, do some tests, see if there are framerate drops, stutters, see how the game behaves when the internet goes down while playing, etcetera. . .

Some games even decided to go DRM-less, such as any game sold on the GOG (Good Old Games) marketplace. This is another important decision you have to think through, to ensure your game has the highest chance of succeeding as possible.

21.5.1 How DRM can break a game down the line

In November 2019, Disney's game "TRON: Evolution" has been reported as "unplayable" due to the Disney's SecuROM license expiring. This had serious ramifications.

The game cannot be played, since SecuROM checks periodically the activation status and reports a "serial deactivated" error and the people who bought a legit copy of the game found themselves unable to enjoy their rightfully acquired product.

This is another situation where piracy wins, since people who pirate games have a way better experience compared to the people who legally bought the game. Players are in the hands of the game publisher and their intention of maintaining the game; as well as in the hands of the DRM manufacturer, since no software is eternal, the DRM servers may close down, blocking the authentication.

21.6 Free-to-Play Economies and LootBoxes

Let's face the most controversial topic, here we go.

21.6.1 Microtransactions

This is quite the hot topic at the moment: the infamous, devilish microtransactions. Let's put this clear from the start, microtransactions are NOT intrinsically bad. As always there's a good way and a bad way to make things. In that case the bad way can bring really backfire nastily.

Microtransactions should be limited to fashion accessories and avatar customization. Nothing related to game mechanics or gameplay-related elements. Everyone **hates** pay-to-win.

We must acknowledge that there are people willing to pay in order to have some bling for their avatar. Microtransactions, and lootboxes as well, should be used for this kind of audience.

I found Blizzard to be the best example for a good implementation for Microtransactions and lootboxes. Overwatch, for example, presents the best example of lootboxes, giving the audience access only to fashion/-customization features for the heroes, like: different skins, animations, vocal sounds.

On the other hand EA, with the Fifa series, is the perfect example of how to NOT DO microtransaction/lootboxes. The reason is very simple: Fifa Ultimate Team is the biggest, most greedy, pay-to-win mechanic ever.

You could question me, stating that lootboxes are gambling microtransactions. You're right. It's a tricky topic, so it has to be done right, but it's not bad for the sake of existing. Otherwise we would end up making an inquisition about how people spend their money.

Now many countries are making an effort to regulate lootboxes (some even making them straight up illegal), so you should be always informed about local laws, and if you decide to implement them, you should know the duties that come with it.

21.6.1.1 The human and social consequences of Lootboxes

Among all monetization schemes, lootboxes are probably among the most controversial. At least the ones that are paid with real-life money.

The objective of lootboxes and microtransactions is to generate a secondary revenue that some experts call “recurrent user spending”.

The idea behind lootboxes is having a prize pool from where a random prize is selected, the user buys a lootbox and gets a randomized item (or set of items), usually accompanied by an epic set of animations, colors and music.

This mechanic can remind someone of slot machines, which share the same psychological principle behind their idea.

Flashy lights and music are tailored to trigger certain chemical reactions in our brains, which will make them seem fun. This adds to all the “off by one” psychology, that will hook you to have “one more go” (how many times you heard of people losing the lottery because of numbers “off by one digit”?).

The situation gets even more serious when you want a very specific item in the prize pool, which lowers the chances of a win in the player’s eyes and consolidates the “one more go” way of thinking.

Things become horrible when kids are involved, if they see their father buying them a lootbox and learn how to do it, sooner or later the credit card will be rejected (and it happened - [internet archive link to a BBC article](#)).

The situation becomes catastrophic when you start putting people with compulsive disorders, or people with gaming addiction, gambling addiction and the so-called “whales” (people with high amounts of money and high willingness to spend such money in videogames). Some people use videogames as a mean to escape their gambling addiction and lootboxes can trigger the same mechanisms in the peoples’ brains.

This obviously has serious consequences when it comes to consumer trust and the image of your game development studio (or even yourself) that can culminate into a full-blown boycott of your products.

Remember: you are not a big publisher, you cannot take the hit of a really bad decision.

21.6.2 Free-to-Play gaming and Mobile Games

Let’s talk a bit about free to play games.

We cannot approach that topic without speaking a little about mobile games, the very kingdom of free to play gaming.

First of all: **What NOT to do:**

- Make people wait/pay in order to play (The so-called “energy system”, very popular at the moment). Pay-to-Win mechanics are bad, but almost understandable, but Pay-to-Play ones are simply disgusting;
- Make intrusive banners/advertisements, that stop gameplay or reduce the available screen size;
- Avoid Pay-to-Win, especially in a multiplayer game.

I also want to give you a major advice about how to use banners/advertising in a smart way:

Let people decide if they want to view an advertisement or not. Especially in a single player game, if you give them something big in return (for example double points/coins/gold for next game, or the chance to continue after a death) they will accept the offer.

This mechanism has two major positives:

- Gamers are NOT annoyed by the use of advertising, they usually see this mechanism as a trade deal: “Do ut des” (I give you, you give me in exchange);
- If they do it once, they’re probably going to do it again. At the end they willingly watch advertisement for you.

It is obvious that you shouldn't kill your player in-game over and over so you can offer them the chance of watching an advertisement to continue playing, that will be seen as a form of "pay to play", from the player's perspective.

21.7 Assets and asset Flips

Pre-made assets are both a resource and a liability when it comes to videogame development, depending on how they are used.

Assets can be purchased by developers in a so-called "asset store" **as a basis**, a foundation upon which they can build their own game, that's how they are intended to be used and how they should indeed be used.

Sadly for every good thing, there must be a bad thing too: **asset flippers**.

Like "flipping a house" (minus the work), a group of assets can be put together (a bit like building blocks) into a somewhat working final product without much (or any) work, making a so-called **asset flip**.

These products are usually sold to adventurous or unknowing consumers for a quick buck or people who hope to make some money out of "out-of-game economies" (like Steam Cards or Tradeable Items), while the quality of the final product is lacking at best or terrible at worst.

This circles back to "being consumer-friendly": you're not selling to a single person, but to a lot of people with different backgrounds and expertise, and one of those **will** find out if your game is an "asset flip".

Flipping assets is the equivalent of betraying your customers, as well as living entirely on the work of who made said assets.

In short: assets are a good basis to work on your game (or make placeholders), but they need to be refined to elevate the quality of your game.

21.8 Crowdfunding

Crowdfunding has been a small revolution in the world of the "idea to reality" business. You publish an idea, with a working sample and some explanation and people who are interested in your idea invest money.

This makes transforming an idea into reality a lot easier and removes a lot of economic worries from the developers.

Although for every good thing, there is always a pitfall: be it scams, things that utterly impossible to create or mismanagement, not everything that gets "funded" is going to go well.

Here are some tips that will help you in this confusing world.

21.8.1 Communication Is Key

There is an Italian saying that can be translated to something like the following:

■ A good worker must work, know, know how to work and let know about their work

This underlines that you may have knowledge and practice, but communication is a key factor in the success of any kind of work.

Remember to keep your backers updated at least weekly, avoid any language that could create misunderstandings (someone could misunderstand a "cutesy language" for a "mockery of the backers") and always justify your decisions. After all you're talking to your own investors.

21.8.2 Do not betray your backers

Your backers are your first and most precious customers: they're the ones that bought your product "sight unseen", and invested money without knowing what the result will be.

Betraying your backers by taking unpopular decisions is a surefire way to see chargebacks and retaliation. If you suddenly decide to change the store where your game will be published to something different than promised, you better bring some **very strong** arguments for it.

Don't lie, if you receive a cash infusion to publish your product on a certain platform (even as a "temporary exclusive"), tell your backers about your decision and the reason being said cash infusion: some will understand, some won't like it, some will get angry. It's still better than everyone getting angry when they find out you lied to them.

21.8.3 Don't be greedy

Greed is never well seen in the world of crowdfunding, having many projects asking for funds at the same time does not look good and opening a new crowdfunding campaign while a funded project has not yet been delivered looks even worse.

Having many ways to fund the same project may seem a good idea, but it usually isn't and gives out the impression of greed, not even to give the chance of "people who are late" to fund your project.

Another mistake are the so-called "stretch goals": encouragements to fund the project over its goal, in exchange for more features. This is a serious mistake that can break an entire project if you are not able to manage it correctly, because you're trading money for more work, and more work usually means a delayed release.

Stretch goals should be very few, already planned in the time table and well thought out. If they are not, you will pay for that mistake.

21.8.4 Stay on the "safe side" of planning

A common mistake made by people who are getting started in development, is getting the times wrong. It happens. It's normal.

Planning a project is hard, and it's way too easy to get it wrong and find yourself with a deadline knocking at your door and having nothing functional in your hands.

When you plan, you should always double your project duration - in case of unforeseen events. Then you should double that time again - in case of events you really cannot foresee.

21.8.5 Keep your promises, or exceed them

When you present your idea, with a working sample (usually requested by many crowdfunding websites), you should keep in mind that people will use that sample as an "anchor": a fixed reference for the minimum quality expected for the product.

There is no "not game footage" that will save you. You promised, you have to deliver.

The only way to deliver a product that is different from the promised one, is to deliver something that is objectively better than what you promised (better graphics, gameplay, sound, story, features, ...)

21.8.6 A striking case: Mighty No. 9

I want to spend a few words on a crowdfunding campaign that failed to meet expectations and broke all the rules we just saw.

The game was asking for a pledge of 900000\$, clearing that objective in just 2 days, and managed to raise over 4 Million \$ between kickstarter and PayPal. The game trailers showed great graphics, amazing soundtrack and fast-paced gameplay.

Sadly the crowdfunding was really mismanaged, the campaign updates happened often but were kind of disconnected with the reality of development and the campaign was littered with **SIXTEEN** stretch goals (with more foreseen, given the graphics in the kickstarter page).

The product delivered was under the expectations, the graphics were nothing like the trailers, they looked poor and cheap, the game was set to release in April 2015 but was delayed until June 2016.

The creator of the campaign was dreaming for a whole “Mighty No.9” franchise, with physical rewards and an animated series, but didn’t make the cut, even after receiving almost 5 times the asked money and a partnership with the publisher Deep Silver.

What did this game get wrong?

- Too many stretch goals;
- Not delivered in the time promised (the game was delayed a lot of times);
- The final product was way under the expectations;
- Too many platforms to release on (10 of them!);
- The official trailer’s “Make them cry like an Anime fan on prom night” - given that the game is very anime-like, this sounds a lot like a mockery to the fans and backers;
- Before delivering the game, the kickstarter account used for Mighty No. 9 was used to create a new kickstarter (Red Ash -Magicicada-), which does not look good when you’re late in delivering a product;
- Too much to do for a project in its beginnings: there were physical releases and rewards planned, as well as an animated series and a whole franchise. This is too much of a gamble for a product that has still to get its first sale.

If you want to know more about what happened, there are a lot of YouTube videos and articles talking about the matter, it really makes you think about how you should behave when opening a crowdfunding campaign.

22 Game Jams

You miss 100% of the shots you don't take.

Wayne Gretzky

Game jams are a great opportunity to test your game making skills in a tight situation, where time is at a premium and you need to “hack your way” through a fully-fledged prototype of a videogame.

In this section there will be some suggestions on how to survive a game jam.

22.1 Have Fun

The biggest prize you can get from a Game Jam is experience and comparing yourself to other participants constructively. You shouldn't take part to a game jam just for the prize (although aiming for it could make you strive to do better).

If you don't have fun, then it's probably not worth it.

22.2 Stay Healthy

Don't forget to eat, take regular breaks, go to bed early and just keep some healthy working habits when you're participating a Game Jam.

If you don't keep a healthy work style, your productivity is going to take a dive: you'll find yourself having trouble solving the simplest problems, your creativity will be nonexistent and you'll get extremely frustrated.

22.3 Stick to what you know

A Game Jam event is not a good place where you learn a new language or game engine, you barely have time to make a game, let alone learn a completely new language and even a new engine!

If you find yourself having to choose between the newest game engine and something you already used two or three times: **go with what you know**.

22.4 Hacking is better than planning (But still plan ahead!)

During Game Jams time is at a premium: you shouldn't use complex data structures or be concerned too much about “best practices”.

Sometimes “a hack” is better than “a solution”: you're building what is essentially a prototype, if the game works and is playable, you have already reached your objective.

You should still plan ahead for your Game Jam experience:

- Make sure your PC is well set up;
- Make sure your development environment works correctly;
- Ensure you can compile some test programs correctly;
- Have a good IDE_g ready;
- Plan your meals well, the less time you cook, the more time you can rest and think;
- Have a generic roadmap that tells you how much time you want to dedicate to each phase;
- Have some ready-to-use resources to use as placeholders;
- If the game jam allows it, have a basic game structure ready (Like a title screen with a “Play” button).

22.5 Graphics? Sounds? Music? FOCUS!

When it comes to Game Jams you can't afford to waste much time on graphics or sounds: having a library of ready-to-use resources can prove vital, as well as a good way to test game mechanics while someone else is drawing (if you're participating as a group).

Sometimes it could be better having no graphics at all, to an extent (just see the game: "Thomas was alone", where graphics are rectangles), as long as it doesn't stop the game from being enjoyable. This is even more important for more "extreme" Game Jams, like the "0 Hour Game Jam" (where you make a game in the hour that is affected by the DST time change, usually from 2am to 2am DST).

Be essential, if you want to make a fully-fledged RPG with procedurally generated weapons, a Game Jam is not the right place to create it.

Again, time is at a premium, cut everything that can be cut:

- Menus can be replaced with a simple title screen, no options;
- Don't have a dedicated credits screen if the creator names fit in the title screen;
- Keep the HUD minimal or just remove it entirely;

Focus on one or a small bunch of game mechanics and do them well, if you try to do a lot of stuff nothing will come out good enough to make an enjoyable game.

22.6 Find creativity in limitations

With experience, you'll find out that it's much harder to find inspiration when you have full freedom on a project.

Some Game Jams define a "subject" the games have to stick to, this is a nice way to boost creativity: through limitations. Usually these subjects are conveyed through a single word or by expressing a game mechanic.

An example of "game mechanic" could be the following: "2 button controls" (Which, by the way, was one of the themes of Ludum Dare 34).

A "theme" or "subject" could be "gravity", which could mean any of the following:

- Newton's Gravity
- Solemnness, seriousness
- Weight/Pressure (Usually emotional)

Another one could be "growth" (Ludum Dare 34's other Theme) as in:

- Planting and growing plants
- Personal Growth (getting experience in life)
- Eating, Surviving and Growing to adult

Many times you'll find a hidden meaning behind a theme that could give you something really unique and eventually an edge over the other participants in the Game Jam.

22.7 Involve Your Friends!

A Game Jam doesn't have to be a quest for "lone wolves", you can involve your friends in the experience, even if they're not coders or game developers: talk about your ideas, get new points of view, suggestions and help.

Game Jams can be a really strong bonding experience for friends and colleagues.

22.8 Write a Post-Mortem (and read some too!)

One of the most useful things you can do after a Game Jam, both for yourself and others, is writing a so-called “Post-Mortem” of your game, where you state your experience with the Game Jam itself, what went right, what went wrong and what you would do differently in a following event.

A Post-Mortem is a reflection on what happened and how to make yourself and your work better while leaving a document that allows other people to learn from your mistakes and learn from a “more experienced you”.

Obviously a smart move would be reading a bunch of Post-Mortems yourself, so to learn more about Game Jams and avoid some common pitfalls of such an experience.

An interesting take on Post-Mortems could be making a time-lapse video of your work, there are applications ready to use that will take screenshots at regular intervals of your desktop and your webcam (with a nice picture-in-picture) if enabled. At the end of everything, the program will take care of composing a time-lapse video for you. It's interesting to see a weekend go fast forward and say “oh, I remember that”.

22.9 Most common pitfalls in Game Jams

To conclude this section, we'll see some of the most common problems in game jams:

- **Bite more than you can chew:** Aiming too high and being victims of “feature creep” is a real problem, the only solution is staying focused and keep everything as simple as possible and be really choosy on the features to add to the game: take time to refine what you have, instead of adding features;
- **Limitations related to tools:** If your tools have issues importing a certain asset or you are not able to create your art for the game then you're in real trouble. You can't afford to waste time troubleshooting something that is not even related to your game. You need to be prepared when the jam starts, test your tools and gather a toolbox you can rely on, something that will guarantee you stability and productivity.
- **Packaging:** This is a hard one - sometimes the people who want to try your game won't be able to play it, either because of installer issues or missing DDLs. Make sure to package your game with everything needed and eventually to link to the possible missing components (like the Visual C++ Redist from Microsoft). The easier it is for the evaluators to try your game, the better.
- **Running out of time or motivation:** plan well and be optimistic, you can do it!

23 Dissecting games: two study cases

In this section we will talk about two games, one bad and one good, and study their decisions closely, how a bad game inconveniences the player or how a good one reduces the space between the player's thought and the action on the screen.

This is not intended as a section to diss on a bad game or sing the praises of a good one, it is meant to be a study on how bad decisions pile up into what is universally recognized as a “bad game”, while there are so many good decisions that need to be taken to make a “good game”.

23.1 A bad game: Hoshi wo miru hito

23.1.1 Introduction

“Hoshi wo miru hito” (roughly translated to “stargazers”) is a Japanese turn-based RPG for the NES/Famicom, set in a future where everyone has “Extra Sensory Perception” (ESP) and where a supercomputer has enslaved humanity via brainwashing.

The story may sound thrilling but the game is not. Let's see why this game is also known as “Densetsu no Kusoge” (I will leave finding the meaning to the reader).

23.1.2 Balance Issues

23.1.2.1 You can't beat starter enemies

At the beginning of the game, you deal only 1 point of damage per attack, way less than the health of the common starter enemy. This, together with your starting health points, make the starter enemies unbeatable most of the times. Speedrunners usually end up resetting in case they get an enemy encounter at the beginning of their run.

23.1.2.2 The Damage Sponge

One of the characters, called “Misa”, is the only character that is able to walk on damage tiles without getting hurt. There is no explanation on the reason behind it.

This means that you have to die multiple times before finding out that only one of the four characters in your party is able to cross certain floor tiles, that may be no different than the other tiles.

23.1.2.3 Can't run away from battles, but enemies can

In this RPG you lack the option to run away from battles.

Enemies instead have a chance to run away from battle when their health points drop below 25% of their original health. Talk about fairness.

The “escape” option is instead hidden behind the “teleport” spell that you acquire after leveling up, in addition such spell is really weird in its way of working.

After selecting the “teleport” spell, you select a team mate to target such spell to, the spell can either succeed or fail:

- If the spell succeeds, the selected team member escapes the battle, while the others continue fighting for the turns that follow;
- If the spell fails, the whole team gets ejected (read “escape”) from the battle.

This means that the teleport spell is more beneficial (4 times faster) when it fails than when it succeeds.

23.1.2.4 Statistics

There are some statistics that make sense in other RPGs, but do not in this game.

For instance the “defense” statistic scales so poorly that you barely notice its effect in this game.

In other games the “speed” statistic is tied to the order of attack (from the quickest to slowest character), but in this game the order is always “player’s team” first, and “enemy team” after.

In conclusion, in “Hoshi wo miru hito”, defense is effectively useless while speed is not even implemented.

23.1.3 Bad design choices

23.1.3.1 You get dropped in the middle of nowhere

In the NES era, it was common thing to have the story written in the manual. To save space on the cartridge, the beginning story elements were reduced to a minimum or completely removed, but in most games you still had a sense of where to go.

In this game, you just get dropped in the middle of nowhere, with no direction whatsoever. And you don’t have the “Legend Of Zelda” style of exploration, since any enemy can make minced meat of you.

As a comparison, Dragon Quest, a game from the same period, had at least a hearing with the king to still introduce you into the story.

23.1.3.2 The starting town is invisible

The previous point is not really true, you actually start near a town, but such town is invisible.

The game makes a really lousy attempt to justify the fact that the town is invisible, but such explanation falls absolutely flat.

This just adds to the confusion of the story, as well as the lack of direction given to the player which can result in frustration and abandoning the game.

23.1.3.3 The Jump Ability

At level 1, you acquire a “jump ability”, that allows you to jump over certain tiles, like rivers. The issue is that such tiles are not distinguishable in any way from all the other tiles.

So you will find yourself mashing your main character’s body against various tiles, trying to find which ones you can skip with your jump ability, and probably die in the process by finding an unrecognizable damage tile.

23.1.3.4 Items are invisible

All items in the game are invisible, including all plot-crucial and revive items. The only thing telling you that you found an item, is a “beep” sound when you collect them.

This further piles up with the lack of direction the player faces in this game since the beginning. While it’s understandable that the limited size (and therefore duration) of NES/Famicom games kind of forced the developers’ hands into making harder games (to make them last longer), but introducing confusing of flat-out unfair mechanics is just bad design.

23.1.3.5 Item management

Usually when you buy a new weapon inside an RPG, you get to un-equip the old weapon and substitute it with the new one, then eventually sell the old one to recover some currency. This gives the game’s challenge new dimensions: item management and a simple economy system.

Well, this game instead lacks any kind of item management: every time you buy a new weapon, the old one will be automatically discarded. You cannot sell old weapons, and the auto-discard removes the possibility of trying a new weapon and in case go back to the old one.

And you cannot un-equip items and weapons.

23.1.3.6 Buying Weapons makes you weaker

When unarmed, from level 1 onwards, the fight option lets you deal a damage equal to a random number between 0 and 4 (bounds included), which is a real low amount of attack power.

When armed, the enemies defense values are taken into account instead, which means that most of the time, the boosted attack power given by the weapon doesn't overcome the enemies defense enough to make using weapons an advantage.

In few words: buying weapons makes you weaker.

And, as stated before, you cannot un-equip weapons, so your game session is probably ruined.

23.1.3.7 Enemy Abilities

Many enemies have an ability which is essentially a permanent, non curable in battle, paralysis + poison combo that will make your battle really hard and frustrating. That means that you will lose all the turns of the character that has been hit with such status effect.

And in case all your party members are hit with such status effect, you don't game over immediately, instead you will keep losing turns while the enemies slowly chip away at your party's health until you eventually game over.

Such effect lasts outside of battle too, so every step you take the affected party members will lose health until you see a healer.

23.1.3.8 You can soft lock yourself

In the vast majority of games, keycards are usually a permanent item that can be reused after finding it. In other games instead doors opened with keycards stay open for the rest of the game.

In this game, keycards have to be bought for quite the price, and disappear on use, and there is a serious chance that you softlock yourself somewhere if you don't buy enough.

23.1.4 Confusing Choices

23.1.4.1 Starting level for characters

This may be minor, but your characters start at level 0. Simply confusing and weird from a player standpoint.

As a programmer I find it quite amusing, though.

23.1.4.2 Slow overworld movement

The movement in the overworld is really slow, at around 1 tile every 2 seconds. This is really confusing, since the NES/Famicom is definitely capable of higher performance.

This is not due to lag or performance issues, it is a conscious decision to make the characters walk so slowly.

23.1.4.3 Exiting a dungeon or a town

Every time you exit a town or a dungeon, you won't find yourself at the entrance or exit of such place (like you'd expect), but instead you will find yourself at the default spawn of the world you're in.

So you may find yourself at the beginning of the game or in some not easily recognizable point of the map.

23.1.4.4 The Health Points UI

In the battle UI, the health of your team members is shown on top of their pictures, as an overlay.

Given the size of the font and the size of the pictures, only 4 digits fit. Given the game's health scaling, there is a serious chance that you get your health points to 5 digits.

The solution adopted was to drop the last digit of the health counter in all cases (even if your maximum health has less than 5 digits): so if you see "15" your health is actually between "150" and "159".

Also for some reason, if your health is lower than 10 points, your health shows as 0.

23.1.5 Inconveniencing the player

23.1.5.1 The battle menu order

In the great majority of turn-based RPGs, the options are shown in the following order:

1. Fight
2. Magic (ESP)
3. Items
4. Escape

This is done in order, from most used (fight) to least used (escape).

In "Hoshi wo miru hito", the menu order presents the ESP option as the first option, selected by default, so most of the time you will have to move your cursor to the "fight" option and select it. This compounds with another problem exposed below.

23.1.5.2 Every menu is a committal

There is no "back" option in any menu, this means that every menu is a committal and you can't back off from any decision, even accidental ones.

That means that if you accidentally select the ESP option in battle and you don't have enough energy/mana to execute any attack, you will end up losing a turn.

If you select the wrong ingredient to make a potion, you most probably will have to waste that ingredient.

23.1.5.3 Password saves

In the NES/Famicom era, games that made use of battery-backed RAM modules to save game progress were rare. This means that the most used save method was using "passwords": a jumble of letters (and eventually numbers and symbols) that needed to be written down precisely, or you wouldn't be able to restore your progress.

This game's passwords are **massive** and use a mix of katakana japanese characters and English alphabet, (while the rest of the game uses hiragana characters), which can be confusing.

Also passwords don't really save your progress "as is": your levels are saved in multiples of 4 (so if you're level 6, you will restore your progress but be level 4) and money is saved in multiples of 255 (if you have 3000 gold, you will restore your progress but have $255 \cdot 11 = 2805$ gold)

23.1.5.4 Each character has their own money stash

In most RPGs that feature a party, there is a shared pool of money that is used for all expenses, this may not be “realistic” but it’s a good enough approximation that has a major upside: it is practical.

This game instead inconveniences the player further by giving each party member their own separated money stash. This is realistic and sometimes used in more modern RPGs, but it is not practical: every time you need to purchase potions used by the whole party (remember: there is no item management) you will have to switch characters or you’ll find yourself running out of money.

23.1.6 Bugs and glitches

23.1.6.1 Moonwalking and save warping

This game doesn’t interpret inputs as well as it should, so if you press the up and down buttons at the same time, you will find yourself “moonwalking”.

Besides the perceived coolness of such move, moonwalking will allow you to go through obstacles, and eventually corrupt the graphics of the tilemap (like loading the right side of the map on the left side of the screen).

This is due to the game checking one direction for wall collisions, but moving the character in the opposite direction.

Pressing up and down at the same time on a controller is not possible, due to the fact that the NES/Famicom D-Pad does not have separated buttons, but if you connect any accessory that allows you to connect up to 4 controllers, the game won’t be able to distinguish between the inputs from Controller 1 and the ones Controller 3.

A side effect of moonwalking, used in speedrunning is “Save Warping”, you are able to manipulate the tilemap and your position via moonwalking, then save and *voilà* you will be warped to another point of the map.

23.1.6.2 The final maze

The final maze is divided in multiple floors, and is the greatest proof of how rushed this game was.

In the first floor of this maze, which is supposed to be really hard, no encounter tiles have been programmed: this means you won’t have to fight anything on this floor. Also no “wall collision” was programmed either, so you can go through the maze walls with the same ease you walk on the floor.

In the other maze floors, encounter tiles were programmed, but still no wall collision was implemented, and since you can’t encounter anything on walls, you can just minimize your encounter chance by taking a stroll inside the maze’s walls.

23.1.6.3 The endings

This game, very ambitiously I shall say, features multiple endings. Towards the end you have to take a very hard decision:

- Join your enemy and leave humanity and live peacefully
- Leave the disputed territory and let the enemy live in peace
- Fight to gain control over the disputed territory

This can result in four different endings, which is really ambitious for a NES/Famicom game. If only the final boss fight was implemented. . .

If you choose to fight, you will automatically lose the battle and the game will end with a “bad ending”.

23.1.7 Conclusions

“Hoshi wo miru hito” is a product of its situation, rumors state that this game was programmed by only one person, and rushed beyond belief so it could compete with *Dragon Quest* in some way. For the sake of fairness, I will assume that this game was made by a team.

The game has interesting ideas for its time: a cyberpunk theme, Extra-sensory powers, the character sprites “grow up” as they gain levels, the enemy sprites are artistically well-done, ... but the execution is littered with problems, obstacles to the player, bad design decisions and bugs.

It seems that the developers were not familiar with the NES/Famicom architecture, game designers weren’t really familiar with game design concepts and play testers were completely nonexistent.

Even though this game has earned the status of “legendary bad game” (not a literal translation of “Densetsu no Kusoge”), “Hoshi wo miru hito” has gained a cult following that is really devoted, to the point that a hacker took upon themselves the daunting task of patching the game and redraw the graphics, as well as rebalancing the weapons and fix the walking speed.

There is even a “remake” called “STARGAZER” for windows.

23.2 A good game: Dark Souls

[This section is a work in progress and it will be completed as soon as possible]

24 Where To Go From Here

Be grateful for all the obstacles in your life. They have strengthened you as you continue with your journey.

Anonymous

It has been a very long trip, hopefully this book has satisfied great part of your curiosity about 2D Game Development and Design, but the adventure is far from over. There are infinite resources on- and off-line to further your experience in game development and design.

This book has been mainly about getting you to touch the nitty-gritty of game development, giving you the tools to be able to create your game without overly relying on your toolkit (be it Unity, UDK or any other fully-fledged game engine), or even be able to modify, at a low level, the behaviour of the tools given to you to reach your objective.

There are still countless questions remaining, which we can condense in one big question:

“Where do I go from here?”

Here below, we can see, divided by category, some resources you can check out to become a better game developer and designer.

But first, here’s a small legend to distinguish paid products from free products.

- **Free Product:** [F]
- **Accepts donations or Partially Free:** [D]
- **Paid Product:** [P]

24.1 Collections of different topics

24.1.1 Books

- **Apress-Open Ebooks:** <https://www.apress.com/it/apress-open/apressopen-titles> [F]
- **“OpenBook” offers from O’Reilly:** <https://www.oreilly.com/openbook/> [F]

24.1.2 Videos

- **freeCodeCamp.org’s YT:** <https://www.youtube.com/channel/UC8butISFwT-Wl7EV0hUK0BQ> [F]
- **3DBuzz:** <https://www.3dbuzz.com> [F]

24.1.3 Multiple formats

- **Daily “Free Learning” offer from PacktPub:** <https://www.packtpub.com/free-learning> [F]
- **“Awesome lists”:** <https://github.com/sindresorhus/awesome> [F]

24.2 Pixel Art

24.2.1 Multiple Formats

- **Pedro Medeiros’s Patreon Page:** <https://www.patreon.com/saint11/> [D]

24.3 Sound Design

24.3.1 Multiple Formats

- **EpicSound’s Sound Effects Tutorials:** <https://www.epicsound.com/sfx/> [F]

24.4 Game Design

24.4.1 Books

- A theory of fun for game design (*Raph Koster*) [P]
- The Art of Game Design: A Book of Lenses (*Jesse Schell*) [P]

24.5 Game Development

24.5.0.1 Videos

- OneLoneCoder's YT: <https://www.youtube.com/channel/UC-yuWVUplUJZvieEligKBkA> [F]

Glossary

A

API Short for “Application Programming Interface”, it’s a set of definitions, tools and clearly defined methods of communication between various software components.

D

Dynamic Execution See *out of order execution*

E

EULA Short of “End User License Agreement”, is essentially a contract that establishes the purchaser’s right to use the software, usually with some limitations on how the copy can be used.

G

Greedy Algorithms Class of algorithms that try to solve a problem by making the locally optimal choice **at each stage**, approximating the global solution, without any global planning.

H

Hash Function A hash function is a special function that is used to map data of arbitrary size to fixed-size values. This function has some features like being able to spread values in an uniform way (minimizing the different values that have the same hash, called “hash collisions”), is fast and deterministic (given the same input will generate the same hash).

HUD Short of “Heads Up Display”, in games it usually shows your health, ammunition, minimap and other information.

I

IDE Short for “Integrated Development Environment”, it is a program that integrates a text editor with syntax highlighting, a compiler, a code checker, a project explorer and other features (like a tag explorer, for instance).

K

Kanboard Short for “Kanban Board”, are boards used to manage work. The board is usually divided into swimlanes and “cards” that represent the work to do are moved from left to right, to represent the progress of the work itself.

M

Malware Short for “malicious software”, it’s a “catchall term” for viruses, trojan horses and any kind of software that is programmed to behave maliciously. Such software can steal information (passwords, key presses, habits, etc. . .) or flat out try to make your computer unusable (deleting system files, encrypting your documents and asking for a ransom, etc. . .)

O

Out of order execution Paradigm used in high performance CPUs to reduce the wasted instruction cycles. The instructions are performed in an order given by the availability of the input data and execution units instead than the original order defined by the programmer.

P

Process Starvation See *starvation*

R

Race Condition A condition where two or more threads are competing for a resource (variable, screen buffer, ...) and the order that the threads happen to take hold of such resource changes the result.

Rootkit Usually associated with malware, a rootkit is a software that manages to get access to reserved areas of an operating system, usually to hide its own, or other softwares' presence.

S

Stack Overflow A stack overflow is a situation where too much data is pushed into a data structure called a "stack". One of the most common cases of "stack overflow" happens during recursion: when a function is called all the current work variables are saved and pushed into a stack data structure in memory, along with a "return address" that will allow us to come back to this point of the program. When a recursion is too deep (the recursive function calls itself too many times), the call stack gets filled up and it's not able to continue the execution, leading to an aborted operation.

Starvation Also known as "process starvation", it's a phenomenon where a certain process (or group of processes) has a lower priority than others, and is not able to access resources (like the CPU) because it's always "overtaken" by higher priority tasks. This leads to the process itself never being executed. When this happens, a process is labeled as "in starvation".

Static Typing Languages characterized by *static typing* are the ones where the type of a certain variable (integer, string, class, ...) is known at compile time.

U

UI Short of *User Interface*, defines the elements shown to and interacted by the user in a software or, in this case, a videogame menu.

W

Wiki A wiki usually refers to a knowledge base website where users collaboratively modify content and structure by using their own web browsers.

Engines, Libraries And Frameworks

Here follows a list of game engines, libraries and frameworks, so you can make an informed choice about your platform, or just try one!

For each proposed engine or framework, along with a short description, you will find the following sections:

- **Website:** This contains a link to the official website of the engine/framework/library;
- **Price:** Here you will see if the product is free to use or if you need to pay a price for it, and anything in between;
- **Relevant Bindings:** In this table you will find if the product supports one of the many famous programming languages available;
- **Other Bindings:** Differently from the previous table, this table has only two columns:
 - **Proprietary Language:** This engine/framework makes use of its own scripting language, usually easier to learn than general-purpose languages. You may need to learn it;
 - **Visual Programming:** This product makes use of a “Visual Scripting” (codeless) paradigm, where you can program your own game without writing code, usually by using directed graphs and nodes.
- **Platform compatibility:** This will tell you if the product is good for your objective, by telling you which platforms you can deploy on, like Linux, Windows or even on the web via HTML5;
- **License:** This will tell you how to behave when it comes to the legal stuff, since some engines do not allow commercial use in their free versions.

SFML

Website: <https://www.sfml-dev.org/>

Price: Free

SFML (Simple Fast Multimedia Library) is a library dedicated to creation of multimedia applications (not limited to videogames), providing a simple interface to the system's components.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
✓	✓	✓	✓	✓	✓	✓		✓	

Other Bindings:

Proprietary Language	Visual Programming

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓	*	*	

*: Currently in development

SFML is distributed under the ZLib/png license, which allows for both commercial and personal use, both in proprietary and open-source situations.

SDL

Website: <https://www.libsdl.org/>

Price: Free

SDL (Simple DirectMedia Layer) is one of the most famous libraries around to make multimedia applications as well as videogames.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
✓	✓	✓	✓		✓		✓	✓	

Other Bindings:

Proprietary Language	Visual Programming

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓	✓	✓	

SDL is distributed under the ZLib license, which allows for both commercial and personal use, both in proprietary and open-source situations. Many of the languages listed as “usable” are compatible via extensions.

The versions of SDL up to version 1.2 are instead distributed under the GNU LGPL license, which is more complex and may need to be analyzed by legal experts.

Löve

Website: <http://love2d.org/>

Price: Free

Löve is a lua-based framework for creating games, features an extensive documentation and features some levels of abstraction that help with game development.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
							✓		

Other Bindings:

Proprietary Language	Visual Programming

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓	✓	✓	

Löve is distributed under the ZLib/png license, which allows for both commercial and personal use, both in proprietary and open-source situations.

Unity 3D

Website: <https://unity.com/>

Price: Free for Personal Use + Paid Version for commercial projects

Unity is probably among the most famous 3D engines used to create videogames, as well as other projects that make use of its 3D capabilities (like VR/AR-based projects). It uses the C# programming language.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
		✓							

Other Bindings:

Proprietary Language	Visual Programming

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓	✓	✓	✓

Unity is a proprietary engine, distributed under a proprietary license, with a Free edition available.

IRRlicht

Website: <http://irrlicht.sourceforge.net/>

Price: Free

IRRlicht is a 3D engine (as in does only 3D rendering) made in C++ that aims to be high-performance.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
✓		✓		✓	✓	✓			

Other Bindings:

Proprietary Language	Visual Programming
----------------------	--------------------

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓			

IRRLicht distributed under the ZLib/png license, which allows for both commercial and personal use, both in proprietary and open-source situations.

Ogre3D

Website: <https://www.ogre3d.org/>

Price: Free

Ogre3D is an open source 3D graphics engine (it's used to render 3D graphics only).

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
✓		✓		✓	✓	✓			

Other Bindings:

Proprietary Language	Visual Programming
----------------------	--------------------

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓			

Ogre3D comes in 2 versions: version 1.x is distributed under the GNU LGPL license, while the more recent 2.x version is distributed under the more premissive MIT license.

Panda3D

Website: <https://www.panda3d.org/>

Price: Free

Panda3D is a complete and open source 3D game engine.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
✓					✓				

Other Bindings:

Proprietary Language	Visual Programming

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓			

Panda3D itself is distributed under the modified BSD license, which is very permissive, but it brings together many third-party libraries that are released under different licenses. It is suggested to check the license section of the manual for more information.

Godot

Website: <https://godotengine.org/>

Price: Free

Godot is a fully-fledged engine that makes use of a node-based approach and supports many ways of programming your own game, both in 2D and 3D, including its own language (GDScript) and visual scripting.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
✓		✓							

Other Bindings:

Proprietary Language	Visual Programming
✓	✓

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓	✓	✓	✓

Godot is currently distributed under the MIT license, you should check the Legal section of the Godot Documentation for all the additional licenses that you may need to know about.

Game Maker Studio

Website: <https://www.yoyogames.com/gamemaker>

Price: Commercial

Game Maker Studio is one of the simplest game-making toolkits available on the market, but that doesn't mean it's not powerful. In fact, one of the most famous games of recent history was made with it: Undertale.

It makes use of its own scripting language, and some visual toolkits as well.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript

Other Bindings:

Proprietary Language	Visual Programming
✓	✓

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓	✓	✓	✓

Game Maker Studio is commercial software, regulated by its own EULA, but it was added here for sake of completeness.

MonoGame

Website: <http://www.monogame.net/>

Price: Free

MonoGame is an open-source porting of XNA 4, it allows for people used to Microsoft's XNA framework to port their games to other platforms, as well as creating new games from scratch. Many games make use of this framework, one above all: Stardew Valley.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
		✓							

Other Bindings:

Proprietary Language	Visual Programming

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓	✓	✓	

MonoGame is distributed under a mixed license: Microsoft Public License + MIT License. You may want to check the license yourself on the project's [GitHub](#) page.

GDevelop

Website: <https://gdevelop-app.com/>

Price: Free

GDevelop is an open-source toolkit to make games, mostly based on visual programming, GDevelop supports the use of JavaScript to code parts of the game, as well as JSON and support for HTTP Requests. GDevelop also supports exporting to Android and Facebook Instant Games, as well as exporting to iOS and web-based platforms.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
									✓

Other Bindings:

Proprietary Language	Visual Programming
	✓

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓	✓	✓	✓

GDevelop is distributed under the MIT license (although the name and logo are copyright), although the main license file refers to other license files inside each folder. So you may want to check the [GitHub](#) repository for more information.

GLFW

Website: <https://www.glfw.org/>

Price: Free

GLFW is an Open-Source library for OpenGL, OpenGL ES and Vulkan, that allows to create windows, context and surfaces, as well as receiving input and events. It is written in C.

Relevant Bindings:

C++	C	C#	Go	Java	Python	Ruby	Lua	Rust	JavaScript
✓	✓	✓	✓	✓	✓	✓	✓	✓	

Other Bindings:

Proprietary Language	Visual Programming

Platform Compatibility:

Windows	Linux	Mac OS	iOS	Android	Web-Based
✓	✓	✓			

GLFW is distributed under the ZLib/png license, which allows for both commercial and personal use, both in proprietary and open-source situations.

Some other useful tools

Graphics

Gimp

Gimp is an extensible drawing and photo-manipulation tool, it can be used to draw, edit, filter, balance or compress your graphic resources.

<https://www.gimp.org/>

Krita

Krita is a drawing program principally aimed towards artists, with all kinds of brushes and tools it's a real treat to any graphical artist and whoever wants to give a “painted feeling” to their game.

<https://krita.org/en/>

Aseprite

Aseprite is an open-source and commercial tool for creating pixel-art graphics. It can be either bought or compiled from source code, but cannot be redistributed according to its *EULA_g*.

<https://www.aseprite.org/>

Piskel

Piskel is an open-source web-based tool for creating pixel-art graphics and animations. On the website there is a downloadable version too, but usually the web-based one is a bit more performing.

<https://www.piskelapp.com/>

Inkscape

Inkscape is an open-source software to work with vector graphics, if you want to give a really clean look to your graphics, you should probably take a look at this software.

<https://inkscape.org/>

Blender

Blender is an open-source surface modeling program, used in the movie industry and in many other 3D projects, it allows you to create your models and worlds for free.

<https://www.blender.org/>

Shaders

SHADERed

SHADERed is a fully-featured IDE for coding GLSL and HLSL shaders, it supports debugging as well as live preview of the shader you're coding. It's cross-platform and open source.

<https://shadered.org/>

Sounds and Music

LMMS

LMMS (Linux Multimedia Studio) is a software used to create digital music, it works in a similar fashion to the commercially available FruityLoops.

<https://lmms.io/>

SFXR/BFXR/CFXR

SFXR (and its other iterations) is a small software that can help you create 8-bit style sound effects for your games, easily. There are versions for Windows, Mac and even online.

http://www.drpetter.se/project_sfxr.html

Chiptone

An alternative to SFXR, Chiptone is an online tool (made with Flash) that can be used to create chiptune-like sounds for your games.

<https://sfbgames.com/chiptone/>

Audacity

Audacity is an open-source tool for audio editing and recording, extensible with plugins. In the hands of an expert, this seemingly simple program can perform great feats.

<https://www.audacityteam.org/>

Rimshot

Rimshot is a simple but effective drum machine that can be used to lay down the rhythm of your next jam. Useful to help in the creation of background music for your games.

<http://stabyourself.net/rimshot/>

MilkyTracker

MilkyTracker is an editor for tracker files, Amiga-style that takes a lot of inspiration from FastTracker II, it has a lot of functionality, it's well-documented and the community is active.

<https://milkytracker.titandemo.org/>

Maps

Tiled

Tiled is a map editor tool that can be used to draw your maps using a tilemap, supports orthogonal, isometric and even hexagonal maps.

<https://www.mapeditor.org/>

Free assets and resources

In this appendix we'll take a look at a small list of websites that contain resources to help with the development of your game, like assets, royalty-free music and the like.

For each website we will have, along with a short description:

- **Website:** containing a link to the website where you can find the resources listed;
- **Resource Types:** a small table describing the kind of resources you will find on the website.

The licenses vary from website to website and even from single resource to another, so you should pay close attention to what you use and how you use it.

Openclipart.org

Website: <https://openclipart.org/>

Types of resources available:

Graphics	Music	Sound Effects
✓		

Openclipart is a website that contains lots of freely available cliparts , in vector format, that can be a good starting point for your own art: buttons, characters, symbols, ...

The website is currently (as of April 12th, 2020) undergoing a phase of transition, after switching backend and what seems to have been an attack by cyber-criminals.

It now features quite a confusing interface and lacks a search function, but it's a precious source of art if you know what you're looking for.

Opengameart.org

Website: <https://opengameart.org/>

Types of resources available:

Graphics	Music	Sound Effects
✓	✓	✓

Opengameart is a website specifically dedicated to game development, with all kinds of resources to use in your game, either as placeholders or good-quality assets that are meant to stay in your product.

The interface is quite essential and sometimes the website can be slow, but this is a great source for anything you need to make your own videogame.

Freesound

Website: <https://freesound.org/>

Types of resources available:

Graphics	Music	Sound Effects
	✓	✓

The Freesound project is a collaborative database of sounds licensed under the “Creative Commons” license, it also features a very interesting blog that will teach you about sounds and soundscapes.

PublicDomainFiles

Website: <http://www.publicdomainfiles.com/>

Types of resources available:

Graphics	Music	Sound Effects
✓		

Similarly to OpenClipart, PublicDomainFiles contains a ton of graphics that can be a really good starting point for all your projects. It has a nice search function with filters and a good interface, even though it does not look as modern as other dedicated websites.

CCMixer

Website: <http://ccmixter.org/> and <http://dig.ccmixer.org>

Types of resources available:

Graphics	Music	Sound Effects
✓		

CCMixer is a gold mine when it comes to music you can use in your games: its interface is simple and easy to understand, features a tag-based search and the quality is great.

Definitely a website to check when you want to add some beat to your game.

SoundBible.com

Website: <http://soundbible.com>

Types of resources available:

Graphics	Music	Sound Effects
✓		

SoundBible.com is another great website where you can look for sound effects for your games: the interface is quick and all sound effects are listed along with their license, which is definitely a plus.

Incompetech

Website: <https://incompetech.com/>

Types of resources available:

Graphics	Music	Sound Effects
✓		

One of the most known websites in the field, and sometimes a bit overused, Incompetech contains lots of music for your game, but you need to check the license yourself. It surely gives a great set of placeholders for your project, while waiting for a more fitting soundtrack.

Contributors

This e-book is the collective work of many game developers, critics and enthusiasts. Here is a list of who contributed to the making of this work.

- Daniele Penazzo (Penaz)
- Luca Violato (Rei)