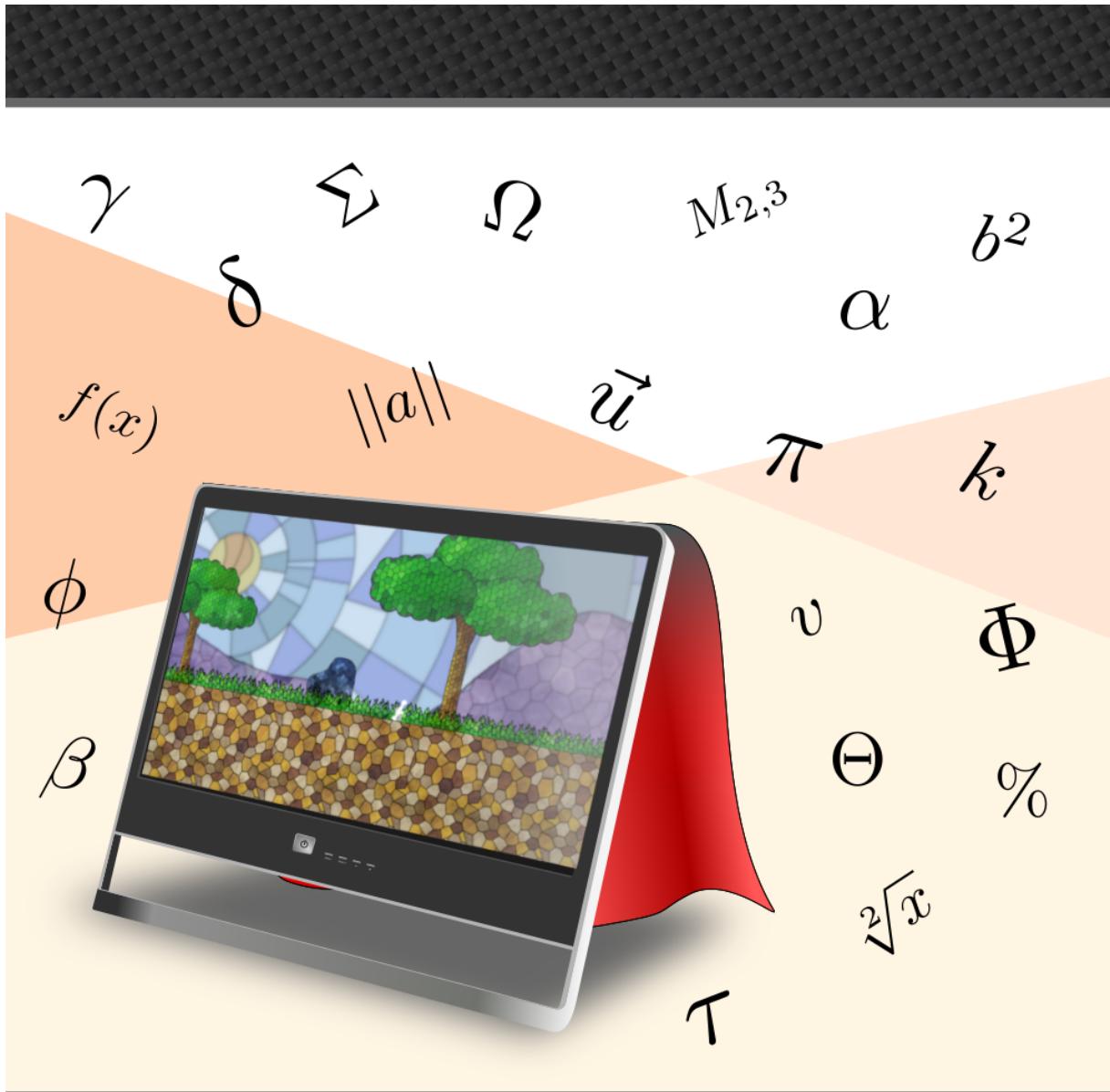


# 2D Game Development: From Zero to Hero

A compendium of the community  
knowledge on game design and development



# 2D Game Development: From Zero to Hero

A compendium of the community  
knowledge on game design and development

# **2D Game Development: From Zero To Hero**

Daniele Penazzo

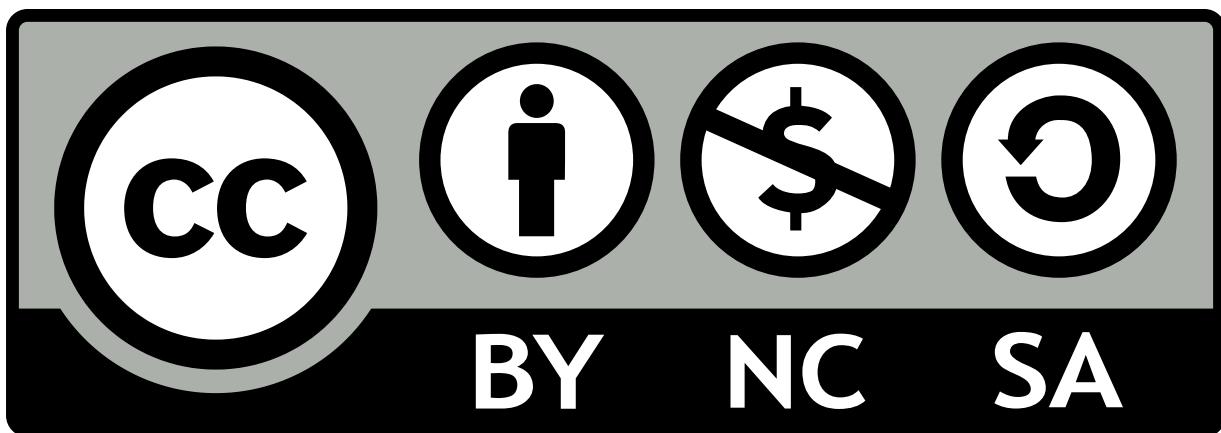
Creative Commons Attribution Non-Commercial 4.0

# 2D Game Development: From Zero To Hero

Copyright © 2019-2022 Daniele Penazzo

2D Game Development: From Zero To Hero ( pseudocode edition, version 0.7.2-r2-1-g55313b7 ) is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 license.

If you want to view a copy of the license, visit <http://creativecommons.org/licenses/by-nc-sa/4.0/> or check the LICENSE file in the book repository.



The PDF and EPub releases of this book can be found at the following address:

- <https://therealpenaz91.itch.io/2dgd-f0th> (Official Itch.io Page)

This book's source code can be found in the following official repositories:

- [https://gitlab.com/Penaz/2dgd\\_f0th](https://gitlab.com/Penaz/2dgd_f0th) (Official GitLab Repository)
- [https://github.com/Penaz91/2DGD\\_F0TH/](https://github.com/Penaz91/2DGD_F0TH/) (Official GitHub Repository)

This work shall be attributed to Daniele Penazzo and the “2D Game Development: From Zero To Hero” community, to see a full list of the contributors, please check the CONTRIBUTORS file in the repository, or head to the contributors section in this book.

Perseverance is the backbone of success.

Anonymous

- To my family
- To my friends, both international and not
- To the ones who never give up

*Daniele Penazzo*

# 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 favorite [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.

# 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.

## 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!

## Conventions used in this book

### 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  $\bar{A}$

## 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:

```
function Example(string phrase){  
    print(phrase);  
}  
  
class ExampleClass{  
    // This is a simple example class  
    ExampleClass(){  
        // This is an example constructor  
    }  
}
```

Code: Example code listing

## 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!

## Boxes

In your journey through this book, you may find some boxes, let's see which ones you may come across.

### **Tip!**

This is a tip box, here you will find tips that are loosely related to the chapter at hand. These small tips will help you make a better game, or wiggle your way through something difficult.

### **Pitfall Warning!**

This is a pitfall box, it will warn you of traps behind the corner, as well as possible shortcomings of a certain solution.

### **Random Trivia!**

This is a trivia box, it will give out some small facts that can help you understand things better, or just give you a small break from all the learning.

### **Note!**

This is just a note box, it's not a pitfall, a tip or a trivia. This is used for reminders and just as a general purpose note

## **Engine Used**

This book does not use any engine. All algorithms will be presented pretending there is some “generic engine” behind the scenes that handles sprites, vectors and the like. The objective of this book is teaching algorithms, tips and tricks and game design in the most engine-agnostic

(and language-agnostic, if you’re looking at the “pseudocode edition”) way possible.

## 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 video game.
- **Collision Detection and Reaction:** In this section we will talk about one of the most complex and computationally expensive operations in a video game: 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.
- **Design Patterns:** A head-first dive into the software engineering side of game development, in this section we will check many software design patterns used in many games;
- **Useful Containers and Class:** A series of useful classes and containers used to make your game more maintainable and better performing.
- **Artificial Intelligence in Video games:** 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 video game;
- **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.
- **Accessibility in video games:** Here we will learn the concept of “accessibility” and see what options we can give to our players to make our game more accessible (as well as more enjoyable to use).
- **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 controversial topics like loot boxes, micro transactions and season passes.
- **Engaging your community:** a lot of a game's power comes from its community, in this section we will take a look at some suggestion you can implement in your game (and out-of-game too) to further engage your loyal fans.
- **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.
- **Project Ideas:** In this section we take a look at some projects you can try and make by yourself, each project is divided into 3 levels and each level will list the skills you need to master in order to be able to take on such level.
- **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 <sup>g</sup> 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 tool kits 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!

# **Part 1: The basics**

# 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
- Cartesian Coordinate system

Also we will represent derivatives with the  $f'(x)$  symbol, instead of the more verbose  $\frac{\partial f}{\partial x}$ .

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.

## Some useful symbols

While reading this book, we may need to delve into some mathematical lingo that not everyone may understand immediately, so here's a small glossary of some of mathematical the symbols we may use.

- $x \in S$  Denotes a “set membership”, so the object to the left of the symbol is an element of the set at the right: x is an element inside the set S;
- $A \subset B$  Denotes a “subset relationship”: A is a subset of B;

- $A \subseteq B$  Denotes a “subset relationship” where equality is possible: A is a subset of B, but also it may happen that A equals B;
- $A \cup B$  Denotes “set union”, the result is composed by all elements of A and B, combined;
- $A \cap B$  Denotes “set intersection”, the result is composed by all elements of A that are also found in B;
- $\forall$  Means “for all”;
- $\exists$  Means “exists”;
- $\exists!$  Means “exists only one”;
- $P \rightarrow Q$  Means “implies”, so you can read this as “P implies Q” or “if P is true then Q is true”;
- $P \leftrightarrow Q$  Logical equivalence: means “if and only if” or “is equivalent”, so you can read this as “P is equivalent to Q” or “P if and only if Q”;

## 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  $\text{mod}(3, 2) = 1$  or  $\text{mod}(4, 5) = 4$  and  $\text{mod}(8, 4) = 0$ .

In most programming languages the modulo function is hidden behind the operator “%”, which means that the function  $\text{mod}(3, 2)$  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](#)).

### Pitfall Warning!

Be careful when using the modulo operator with negative arguments: it may lead to unexpected results, which may depend on the programming language you are using.

# 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.

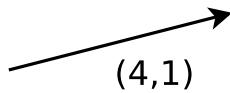


Image of a vector

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.

## Adding and Subtracting Vectors

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

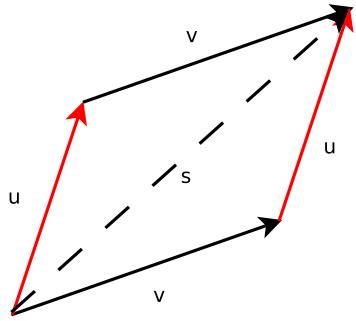
$$v = (2, 4)$$

$$u = (1, 5)$$

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:



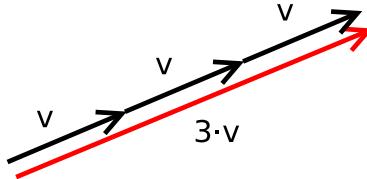
Graphical representation of a sum of vectors

## 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:

$$v = (2, 4)$$

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

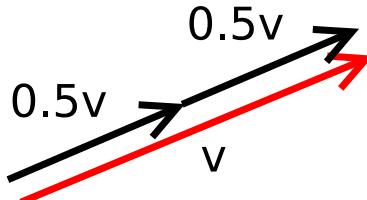


Example of a vector multiplied by a value of 3

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

$$v = (2, 4)$$

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

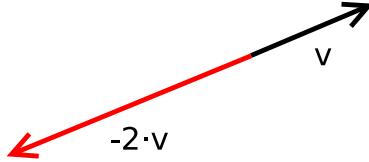


Example of a vector multiplied by a value of 0.5

When you multiply the vector by a value less than 0, the vector will rotate by  $180^\circ$ .

$$v = (2, 4)$$

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



Example of a vector multiplied by a value of -2

## 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:

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

## 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](#))

## “Clamping” a Vector

This is not an operation “per se”, but there are occasions where we need to limit the length of a vector: this usually happens when we are working with velocity, as not limiting it would allow an object to change position faster and faster, making the game less playable and even breaking time-stepping collision detection algorithms.

To clamp a vector, we need to find its magnitude and direction first, which is the “normalized vector”. Let’s think about the vector  $v$ , its magnitude and direction are:

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

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

After that, we can build a new vector using the “clamped magnitude” (which we’ll call  $\|v\|_{clamp}$ ), calculated as such:

$$\|v\|_{clamp} = \begin{cases} \|v\| & \text{when } \|v\| < \|v\|_{max} \\ \|v\|_{max} & \text{otherwise} \end{cases}$$

To build the new vector, we just need to multiply  $\|v\|_{clamp}$  by  $\hat{v}$ :

$$v = \|v\|_{clamp} \cdot \hat{v}$$

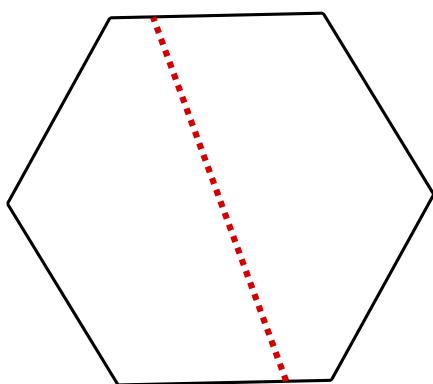
The new vector will have the same direction as the old one, but its magnitude will be clamped, just like we wanted.

## Geometry

Among all the maths we found so far (and the maths we will explain later), we cannot avoid talking a bit about geometry: in this book we will talk about the minimal amount of geometry necessary to understand the underlying concepts of what’s coming up.

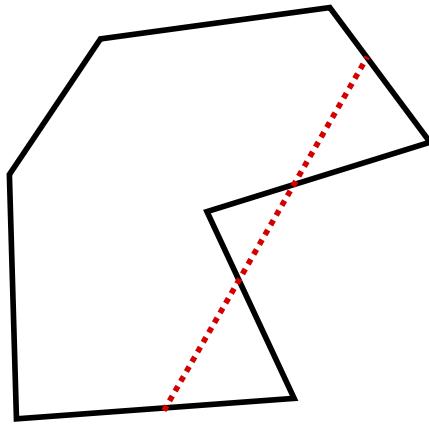
### Convex vs Concave polygons

A polygon is considered convex essentially when **any line** (not tangent to an edge or corner) drawn through the shape crosses the shape itself only twice (at its ends).



### Example of a convex shape

Any shape where you can find at least one line that crosses the shape more than twice is considered “non-convex” (commonly referred as “concave”).



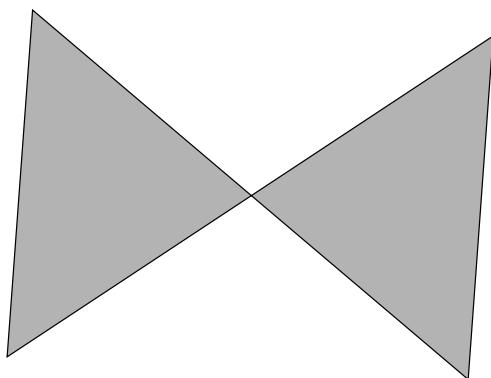
### Example of a concave shape

#### Note!

Not all non-convex shapes are technically called “concave” (they should be called “non-convex”), but for the sake of simplicity we’ll use the term “non-convex” and “concave” interchangeably in this book.

### Self-intersecting polygons

Contrary to what many think, polygons can self-intersect too, which can make calculations a lot harder.



## Example of a self-intersecting polygon

For the sake of game development, we will usually talk about simple polygons which are polygons that don't self-intersect and have no holes in them. More strictly we will (for 99.9% of the time) talk about **convex simple polygons**.

## Straight Lines and their equations

One of the main topics we will encounter over and over in our game development adventure will be "straight lines". We will need to draw them, see if two straight lines collide, project stuff onto them, and much more. So it's important that we know them well.

Here's a straight line:

$$ax + by + c = 0$$

That's not what you expected, right? What you've seen is the "general form" of a straight line's equation, because you can represent lines using equations (also circles, and other stuff). This is not a much-used form, though, probably the most used form is called the "slope-intercept form":

$$y = mx + q$$

### Random Trivia!

To transform a "general form" equation into the relative "slope-intercept from" just remember the following formulas:

$$m = -\frac{a}{b} \quad q = -\frac{c}{b}$$

This doesn't work well when  $b = 0$ , which will be subject of the next "pitfall".

Where in this case  $m$  is the *slope* of our straight line, and  $q$  represents the so-called *y-intercept* (the value of  $y$  when  $x = 0$ ). If  $q = 0$  the line goes through the origin of the Cartesian coordinate system, if  $m = 0$  the line is horizontal.

### Pitfall Warning!

“Vertical straight lines” is where the slope-intercept form fails, in fact vertical straight lines have an equation in the form of  $x = k$ , which would mean that  $b = 0$  which is problematic (see previous trivia).

## Getting the equation of a straight line, given two points

We all know that given two points we can strike one and only one line. How many times did you measure two points (maybe while doing some D.I.Y.) and stroke a line between them?

It will be useful in our adventure to be able to get the equation of a straight line starting from two points, so let's call our two points  $P(x_1, y_1)$  and  $Q(x_2, y_2)$ , then the straight line that crosses both those points will have equation:

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1}$$

This may seem really complicated, but with some small calculations we can reach a formula for our straight line in any form (generic or “slope-intercept”).

### Pitfall Warning!

Again, this formula fails when we are dealing with “vertical lines”, because the denominator at the right side of the equation will be zero. But in that case we'll already know the formula: it will be  $x = x_1$  (which in turn will be equal to  $x_2$ )

## Getting the equation, given the slope and a point

If we have a point  $P(x_p, y_p)$  and the slope  $m$  (for instance if we need to find a line perpendicular to another line), in that case we can use the following formula:

$$y - y_p = m(x - x_p)$$

### Pitfall Warning!

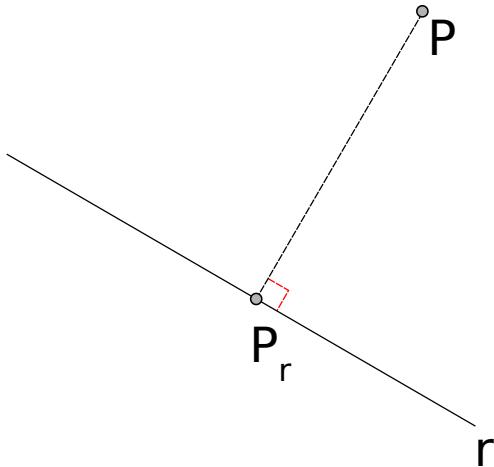
Guess what? This (again) doesn't allow us to create "vertical lines", because we need a slope value, which we don't have when it comes to vertical lines. You can see (non rigorously) a vertical line as a line with "infinite slope".

## Projections

In some situations (as you will see in the [SAT](#)), we may need to get to project polygons onto a line, this usually involves projecting **points** to a line.

Given the formulas we've seen earlier, and doing some thinking, we can easily project a point onto any straight line. Let's see how to do it.

We will assume that we have a point  $P(x_p, y_p)$  that we want to project onto a line  $r$  with equation  $y = mx + q$ , with  $m \neq 0$  (thus excluding horizontal lines). We will call the projected point "P onto r" with the name  $P_r(x_r, y_r)$ .



Projecting the point  $P$  onto the line  $r$

First, we need to find the line that goes through  $P$  and is perpendicular to  $r$ , this is really easy. To find a slope  $m_1$  of a line perpendicular to another line with slope  $m$  we use the formula

$$m_1 = \frac{1}{m}$$

### Pitfall Warning!

This is why we excluded the case  $m = 0$  (horizontal lines), if we didn't we would have the chance of having  $m_1 = \frac{1}{0}$  which doesn't make sense

Now we have a point and a slope, so we can use one of the formulas we've already seen to find the line with that slope that crosses  $P$ :

$$y - y_p = m_1(x - x_p) \Leftrightarrow y - y_p = \frac{1}{m}(x - x_p)$$

To find  $P_r$  we just need to find the point where the two lines collide, which is the solution to the equation system:

$$\begin{cases} y = mx + q \\ y - y_p = \frac{1}{m}(x - x_p) \end{cases}$$

Which finds solution in:

$$\begin{cases} x = \frac{x_p + mq}{1 - m^2} \\ y = \frac{mx_p + q}{1 - m^2} \end{cases}$$

The coordinates  $x$  and  $y$  we just found are actually the coordinates  $x_r$  and  $y_r$  of our projected point  $P_r$ .

### Projecting arbitrary lines on the axes

Similarly to what we've done with points, we can project arbitrary lines (or better, the ends of such lines) onto the axes. This will help us in doing some calculations later.

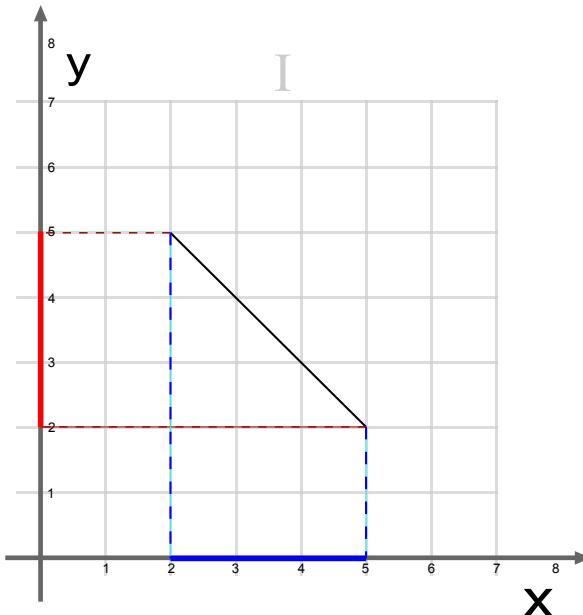
To project any line to the x-axis we can just “pass all the line's points through” the following function:

$$proj_x(x, y) = x$$

If we want to project such line on the y-axis, we can just use this other function:

$$proj_y(x, y) = y$$

We can see the graphical representation of projecting a line onto the axes below:



Projecting a line onto the axes

# Matrices

## 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  $2 \times 3$  matrix.

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

## 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:

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

## 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:

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

## 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:

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

## 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 combined 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:

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

### Note!

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.

## 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.

## 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!

### Radians vs Degrees

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

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

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

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

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

Conversion between degrees  
and Radians

Degrees	Radians
---------	---------

0°	0
----	---

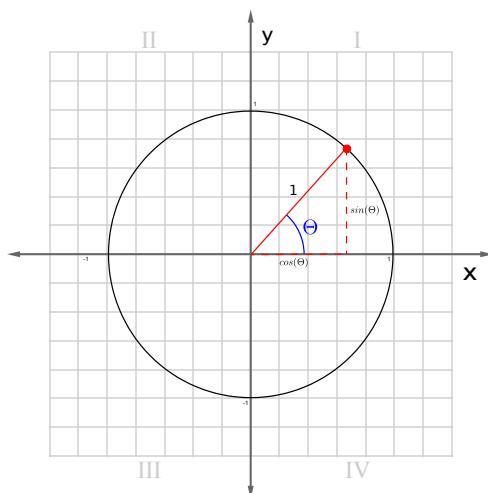
## Degrees    Radians

Degrees	Radians
$30^\circ$	$\frac{\pi}{6}$
$45^\circ$	$\frac{\pi}{4}$
$60^\circ$	$\frac{\pi}{3}$
$90^\circ$	$\frac{\pi}{2}$
$180^\circ$	$\pi$
$360^\circ$	$2\pi$

## 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  $\theta$  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  $\sin(\theta)$ .



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:

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

## 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  $\theta$ :

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

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  $\cos^2(\theta) = (\cos(\theta))^2$ .

## 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:

Some reflection formulas for trigonometry

### Reflection Formulas

---

$$\sin(-\theta) = -\sin(\theta)$$

$$\cos(-\theta) = \cos(\theta)$$

## Reflection Formulas

---

$$\sin\left(\frac{\pi}{2} - \theta\right) = \cos(\theta)$$

$$\cos\left(\frac{\pi}{2} - \theta\right) = \sin(\theta)$$

$$\sin(\pi - \theta) = \sin(\theta)$$

$$\cos(\pi - \theta) = -\cos(\theta)$$

$$\sin(2\pi - \theta) = -\sin(\theta) = \sin(-\theta)$$

$$\cos(2\pi - \theta) = \cos(\theta) = \cos(-\theta)$$

## 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:

Some Shift Formulas for  
Trigonometry

## Shift Formulas

---

$$\sin(\theta \pm \frac{\pi}{2}) = \pm \cos(\theta)$$

$$\cos(\theta \pm \frac{\pi}{2}) = \mp \sin(\theta)$$

$$\sin(\theta + \pi) = -\sin(\theta)$$

$$\cos(\theta + \pi) = -\cos(\theta)$$

$$\sin(\theta + k \cdot 2\pi) = \sin(\theta)$$

$$\cos(\theta + k \cdot 2\pi) = \cos(\theta)$$

## 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:

Some addition and difference identities in trigonometry

## Addition/Difference Identities

---

$$\sin(\alpha \pm \beta) = \sin(\alpha)\cos(\beta) \pm \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha \pm \beta) = \cos(\alpha)\cos(\beta) \mp \sin(\alpha)\sin(\beta)$$

## 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.

Some double-angle formulae used in trigonometry

## Double-Angle Formulae

---

$$\sin(2\theta) = 2\sin(\theta)\cos(\theta)$$

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

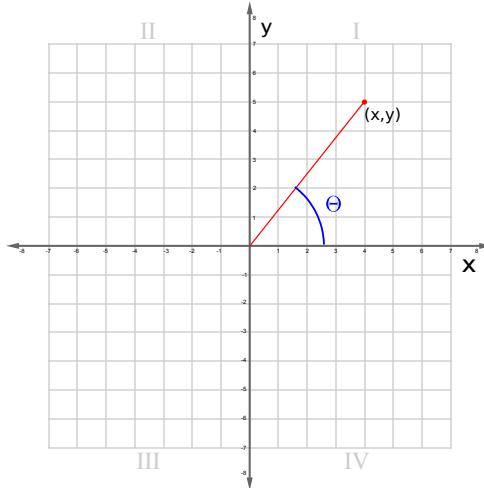
## Inverse Formulas

As with practically all maths 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  $\theta$  is defined as:

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



Graphical plotting of the angle of a vector

## Numerical Analysis

Here we will give some pointers over some algorithms and methods that may be useful to better explain some topics treated in this book. Feel free to skip or quickly read this section if you don't want to dive into too much detail over this kind of maths.

### Newton-Raphson method

Also known as Newton's method, this is an iterative algorithm that is used to get progressively better approximations to the roots of a function.

The algorithm starts with a “guess”, called  $x_0$ , and produces the first approximation using the formula:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Each subsequent guess (and thus iteration) can be obtained similarly by using the formula:

$$x_{n+1} = x_n - \frac{f(n)}{f'(n)}$$

And such guess will be more precise than the previous one (if we don't consider some situations where approaching the root can be problematic or not possible). The algorithm will stop when you reach an approximation that is "good enough".

Obviously all limitations of standard functions apply, such as domain and trouble with divisions by zero.

## Coordinate Systems on computers

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

In our maths courses we learned about the Coordinate Plane, with an origin and 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".

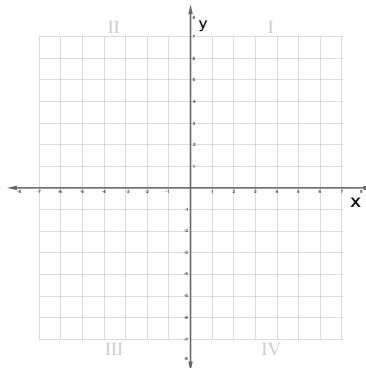


Image of a coordinate plane

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

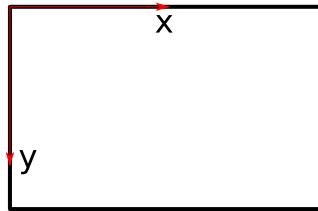


Image of a screen coordinate plane

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.

## 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 tool kits 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.

### 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:

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

## Rotation

If you want to rotate an object by a certain angle  $\theta$ , 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.

## 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  $\theta$  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 **councclockwise**, we will instead use the following matrix:

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

### Pitfall Warning!

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.

## 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.

## Shearing

During stretching, we used the elements that are in the “main diagonal” to stretch our objects. If we modify the elements in the “anti-diagonal”, 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:

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

# 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

## Recursion

Starting from the (arguably hard) theme of recursion may seem weird, but it is important to understand recursion as soon as possible so we can make the best use of it.

There is a joke I like telling around about recursion:

| To understand recursion, you must first understand recursion.

What is recursion? Recursion is the usage of a function that calls itself.

Your first question will probably be: wouldn't that make the program lock up forever in some kind of loop? It may. But if you're careful, recursion is an

amazing tool that allows you to earn a lot of clarity and brevity.

Let's imagine a simple algorithm: we want to make our program count backwards from a number  $n$  to 0. In a simple "loop" fashion, we may write the following:

```
function count_backwards(int n) {
    // Condition for the loop
    while (n != 0) {
        // The function body
        print(n);
        // We update the condition to count down
        n = n - 1;
    }
}
```

Code: Counting from  $n$  to 0 using a loop

Pretty simple, right? A real-world example would be counting back from 10 to 0: we print 10, we subtract 1 to get 9, we print 9, subtract 1 to get 8, ...

Let's turn our thinking around for a second. We can see counting back from 10 to 0 like this: we print 10 and then we count backwards from 9. Counting backwards from 9 would just mean printing 9 and then counting back from 8, etc...

We just turned our simple loop into a recursive function:

```
function count_backwards(int n) {
    // Stop condition
    if (n == 0) {
        return;
    }
    // Procedure
    print(n);
    // Recursive call
    count_backwards(n-1);
}
```

Code: Counting from  $n$  to 0 using recursion

Recursive functions have three main components:

- A **base case** (sometimes called a “stop condition”): this allows the function to stop calling itself when a certain condition is reached;
- A **procedure** that elaborates on data or simply does something (in our example, it just prints the number);
- A **recursive call** to the same function we are writing, the call is done in a way that every call gets closer to the “stop condition”. It can be done by calling the function on a subset of its argument (if it is a list), until the list has only 1 item or on a smaller number (if the function argument is a number instead).

Recursion can be classified in many ways:

- **By the number of recursive calls:** single vs multiple recursion;
- **By how the recursive call is made:** direct (a function calls itself directly) vs indirect (a function A is called by another function B, which in turn is called by function A)
- **By the position of the recursive call:** head vs tail recursion.

I want to underline the last distinction: what we’ve seen in the previous listing is called “tail recursion”: the recursive call is done **after** everything else (the procedure).

Head recursion is instead done when the recursive call is done **before** the procedure starts, so we can transform our “count down” function to a “count up” just by switching from “tail” to “head” recursion and adding a print statement.

```
function count_backwards(int n){  
    // Stop condition  
    if (n == 0){  
        // If we don't do this, we won't print 0  
        print(n);  
        return;  
    }  
    // Recursive call  
    count_backwards(n-1);  
    // Procedure  
    print(n);
```

```
}
```

Code: Counting from 0 to n using head recursion

## Computers are (not) precise

There are many differences between humans and computers, among those there is one that will keep haunting us in our journey: humans make calculations in “base 10” (decimal), computers make calculation in “base 2” (binary).

This requires computers to represent numbers differently, usually with the exponent+fraction representation (IEEE 754). Also computers have limited resources, thus have no concept of “infinity” (and conversely of “infinitesimal”).

Let’s assume a computer with a reduced precision and we execute the following C++ program:

```
#include <iostream>
#include <iomanip>

int main ()
{
    // This will reduce the computer's precision for
    std::cout << std::setprecision(20);

    float d1(1.0);
    std::cout << "This should be 1.0: " << d1 << std::endl;

    float d2(0.1);
    std::cout << "This should be 0.1: " << d2 << std::endl;

    float d3(0.1*0.1);
    std::cout << "This should be 0.01:" << d3 << std::endl;

    bool x (0.1 + 0.1 + 0.1 == 0.3);
    std::cout << "This should be true (1): " << x << std::endl;
```

```
    return 0;  
}
```

We save it as “precision\_test.cpp” and compile it with the following command line (on Linux):

```
g++ -Wall -Wextra -Werror -O0 precision_test.cpp -o precision_te
```

This program will temporarily set a reduced precision in our number representation, and try to output the values of the numbers 1, 0.1 and  $0.1^2 = 0.01$ , let's see the results:

```
penaz@PenazMW2 ~ % ./precision_test.bin  
This should be 1.0: 1  
This should be 0.1: 0.1000000149011611938  
This should be 0.01: 0.0099999997764825820923  
This should be true (1): 0
```

Results of the simple float precision test

With the number 1 it's all good, but... what is going on with 0.1? What is all that garbage? The number 0.01 is even worse! That's not even close! Why  $0.1 + 0.1 + 0.1$  comes out as not 0.3! **What is maths anymore?**

We have just met one of the (many) limitations of computers: computers cannot represent certain numbers without “approximating”. Compilers and libraries exist to work around these issues, but we need to be ready to avoid surprises.

Just to reiterate: this is not a problem of the single programming language, we can see that C++ is affected, but also Python has the same issue:

```
penaz@PenazMW2 ~ % python2  
Python 2.7.18 (default, Mar 15 2021, 17:39:13)  
[GCC 10.2.0] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 0.1 + 0.1 + 0.1 == 0.3  
False  
>>> 
```

Python 2 has the same issues with precision as C++

```
penaz@PenazMW2 ~ ipython
Python 3.9.5 (default, May 12 2021, 19:13:47)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.23.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 0.1 + 0.1 + 0.1 == 0.3
Out[1]: False
```

Python 3 doesn't fare much better when it comes to precision

This is a computer issue in general, this may not be a huge problem for general use but if we try to be too precise with our calculations this may come back to bite us.

## Catastrophic cancellation

With a name as dangerous-sounding as “catastrophic cancellation”, this sure looks like a dangerous phenomenon, but it's only dangerous if we don't know what it is.

Catastrophic Cancellation (sometimes called “cancellation error”) is an event that may happen when subtracting two (usually large) numbers that are close to each other in value.

**Warning:** from here on, in this section, there will be some technical language. I will try to make it as simple and understandable as possible.

Let's imagine a computer, such computer's memory can handle at most 8 decimals while its A.L.U. (the unit that takes care of “doing maths”) can handle at most 16 decimal places.

Now let's take two numbers:

$$x = 0.5654328749846 \quad y = 0.5654328510104$$

When we transfer such numbers in our memory, the computer will approximate such numbers to fit in its memory constraints. We'll represent that by applying to each number a function `fl()` that we can read as “float representation of this number”. So we'll end up having:

$$fl(x) = 0.56543287 \quad fl(y) = 0.56543285$$

This is generally called an “assignment error”, where during the assignment to a variable, a number loses part of its information.

Let's try to calculate how off those approximations are (by calculating the percent “relative error”), just to get an idea of what we lost by just loading the numbers on our “fake computer”:

$$\delta_x = \frac{|x - fl(x)|}{x} = 0.00000088\%$$

$$\delta_y = \frac{|y - fl(y)|}{y} = 0.00000017\%$$

We can see that our approximations are **very close** to the numbers we want to calculate, now let's calculate  $x - y$ . Making things by hand we would have:

$$x - y = 0.239772 \times 10^{-7}$$

That's a tiny number right there. Now let's calculate  $fl(x) - fl(y)$ , remembering that the A.L.U. will fill up to 16 decimals:

$$fl(x) - fl(y) = 0.5654328700000000 - 0.5654328500000000 = 0.000000200000000 = 0.2 \times 10^{-7}$$

That doesn't look so bad, unless we look at the “relative error”:

$$\delta = \frac{|0.239772 \times 10^{-7} - 0.2 \times 10^{-7}|}{0.239772 \times 10^{-7}} = 16.6\%$$

Oh no... We're off by 16% of the total result! That's a huge loss! A real catastrophe.

What happened? If you look closely, the numbers are really close and even have 7 decimal digits in common, since our computer can memorize only 8 digits, the 9th to 13th decimal digits that looked so unimportant suddenly become a huge part of the result (due to the subtraction) but are already lost.

## Random Numbers are not really random

Computers are deterministic machines, given the same set of instructions and inputs, they will **always** return the same output. Someone may think about “random number generators” and sure, those programs look like they spit random numbers on your screen, but they actually don’t.

The most important number when generating random numbers is called *seed* and it’s the number used by the *generator* to produce random numbers.

Let’s see an example of a random number generator in C++:

```
#include<iostream>
int main() {
    // First of all we get the seed
    unsigned int seed;
    std::cout << "Type the seed: ";
    std::cin >> seed;
    // Now we seed the randomizer
    srand(seed);
    // Small presentation
    std::cout << "This generator will now generate 10
    // Output 10 random numbers
    for (int i = 0; i < 10; ++i) {
        std::cout << rand() << std::endl;
    }
    // Finish the program
    return 0;
}
```

We can save this program as `random_seed.cpp` compile this program with the following command:

```
g++ -Wall -Wextra -Werror -O0 random_seed.cpp -o random_seed.bin
```



When we run the program, it will ask us to input a seed (which in our case is a number), after that it will just print 10 random numbers based on that seed. What would happen if we ran the program twice and use the same seed?

```
penaz@PenazMW2 ➤ ./random_seed.bin
Type the seed: 14
This generator will now generate 10 random numbers
2146406683
565464452
463529751
988319322
113724673
1215838388
61920407
1293858600
1603362519
1406570506

penaz@PenazMW2 ➤ ./random_seed.bin
Type the seed: 14
This generator will now generate 10 random numbers
2146406683
565464452
463529751
988319322
113724673
1215838388
61920407
1293858600
1603362519
1406570506
```

Running a random number generator with the same seed will always output the same numbers

Random numbers generated by computers are never truly random, that's why they are more properly called "pseudo-random numbers".

## 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:

$$\text{not } (A \text{ and } B) = \text{not } A \text{ or not } B$$

$$\text{not } (A \text{ or } B) = \text{not } A \text{ and not } B$$

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.

## Estimating the complexity 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:  $\Omega$ ,  $\Theta$  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- $\Omega$  notation have the same value, which can help define the behavior of the algorithm better.

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

**Pitfall Warning!**

Be mindful of one specific thing: these notations simply tie how the algorithm performs in relation to how a certain variable grows (usually a dataset). If you know for certain that a dataset stays relatively small, a less efficient algorithm may not make a huge difference.

## 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`.

```
function isFirstElementNull(elements) -> bool{
    if (elements[0] is null) {
        return True;
    }else{
        return False;
    }
}
```

Code: Example of an O(1) algorithm

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

## 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.

```

function binarySearch(item[] elements, item element_to_find) ->
int{
    get middle_element;
    if (element_to_find == middle_element){
        return the middle element position;
    }else{
        if (element_to_find > middle_element){
            perform binarySearch on the half of the
list bigger than middle_element;
        }else{
            perform binarySearch on the half of the
list smaller than middle_element;
        }
    }
    return null;
}

```

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

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  $\Theta(\log_2 n)$

## **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.

```

function printList(items[] list){
    for (each element in list){
        print element;
    }
}

```

```
}
```

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

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.

## **$O(n \cdot \log(n))$**

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 (F.F.T.)

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.

## **$O(n^2)$**

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.

```
function bubbleSort(items [] A) {
    int n = length(A);
    bool swapped = false;
    do{
        swapped = false;
        for (i from 1 to n-1 inclusive){
            if (A[i-1] > A[i]){
                swap( A[i-1], A[i] );
                swapped = true;
            }
        }
    } until (not swapped);
}
```

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

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

## **$O(2^n)$**

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!

# A primer on calculating the order of your algorithms

## 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:

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

Code: A simple O(1) algorithm

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

Now let's see another algorithm:

```
// A simple O(n) algirithm: iterating through a list
    for (item in my_list) {
        print(item);
    }
```

Code: A simple  $O(n)$  algorithm

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:

```

function bubbleSort(items [] A) {
    int n = length(A);
    bool swapped = false;
    do{
        swapped = false;
        for (i from 1 to n-1 inclusive) {
            if (A[i-1] > A[i]){
                swap( A[i-1], A[i] );
                swapped = true;
            }
        }
    } until (not swapped);
}

```

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

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  $O(n^2)$ .

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

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

```

// -----
int n = length(A);
bool swapped = false;

```

```

do{
    swapped = false;
    for (i from 1 to n-1 inclusive){
        if (A[i-1] > A[i]){
            swap( A[i-1], A[i] );
            swapped = true;
        }
    }
} until (not swapped);

// ----

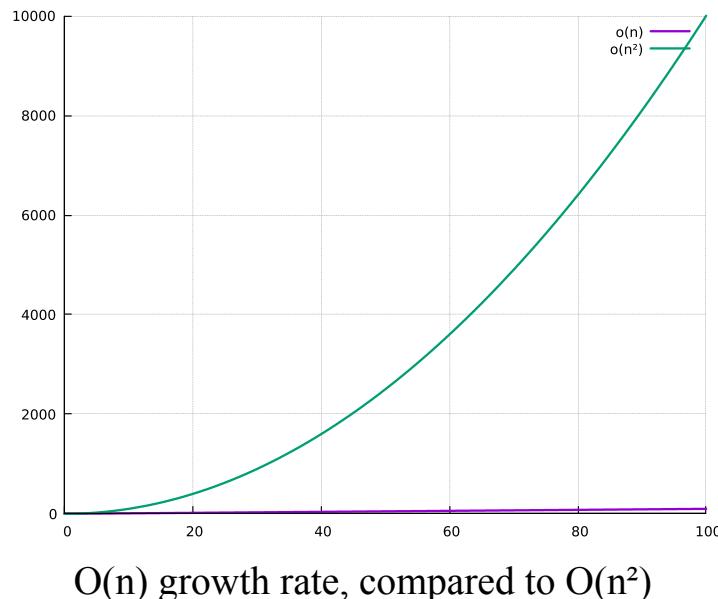
for (item in A){
    print(item);
}

```

Code: A more complex algorithm to estimate

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:



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).

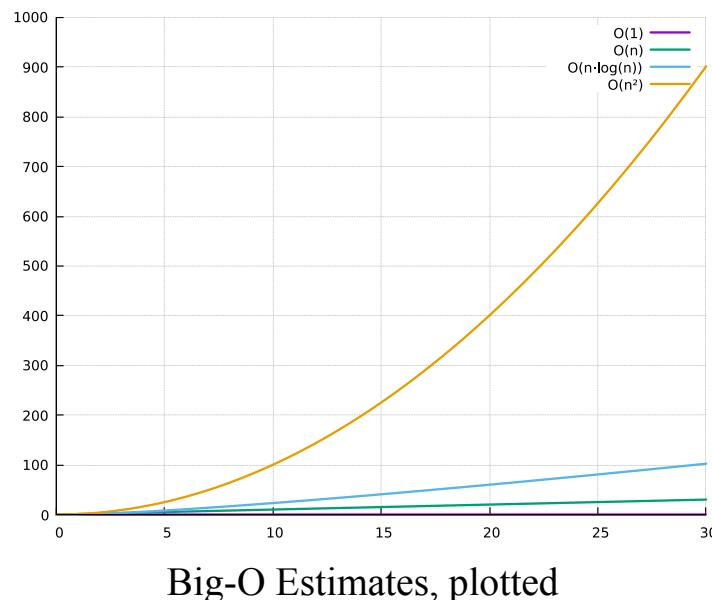
## What do we do with recursive algorithms?

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”.

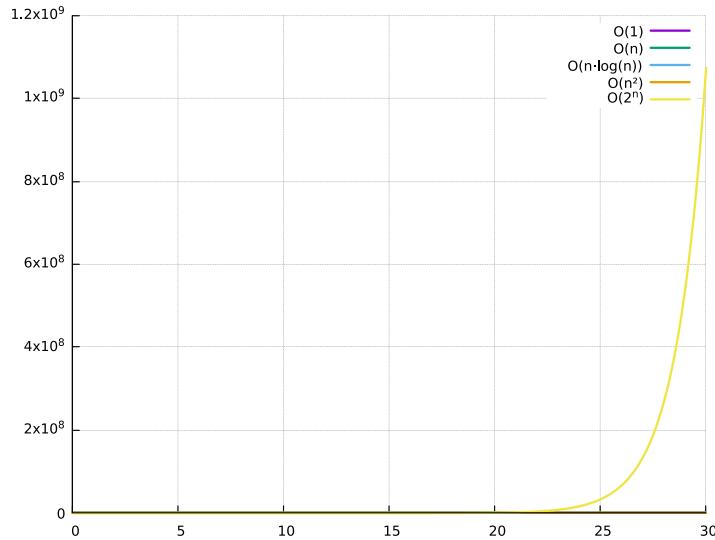
Such methods are outside the scope of this book as of now.

## 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.



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:



How  $O(2^n)$  overpowers lower complexities

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

## 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:

The first truth  
table we'll  
simplify with  
Karnaugh Maps

A	B	f
0	0	0

A	B	f
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):

		A	
		0	1
B	0	0	1
	1	1	0

Karnaugh Map for A XOR B

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:

		A	
		0	1
B	0	0	1
	1	1	0

Karnaugh Map where the elements of the two "rectangles" have been marked green and red

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.

## “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.

Truth table with  
a “don’t care”  
value

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

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

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

Karnaugh Map with a “don’t care” value

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”.

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

Karnaugh Map where we pretend the “don’t care” value is equal to 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

First Rectangle in the Karnaugh map

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

Second Rectangle in the Karnaugh map

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  $B = 1$ .

## A more complex map

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

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

A	B	C	D	f
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	x

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

## A more complex Karnaugh map

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

First rectangle of the more complex Karnaugh map

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

Second rectangle of the more complex Karnaugh map

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.

## Guided Exercise

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

A	B	C	D	f

<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<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	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

Guided Exercise: Karnaugh Map (1/4)

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

Guided Exercise: Karnaugh Map (2/4)

		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

Guided Exercise: Karnaugh Map (3/4)

		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

Guided Exercise: Karnaugh Map (4/4)

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

## Object Oriented Programming

### Introduction

One of the biggest programming paradigms in use is surely the “Object Oriented Programming” (from now on: “O.O.P.”) 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*.

## Objects

Objects are the fundamental unit in O.O.P., 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).

## Abstraction and Interfaces

Abstraction is a fundamental point in O.O.P., 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*.

## Inheritance and Polymorphism

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

## Mixins

Mixins are classes that contain certain methods that are made to be used by other classes. We can see mixins as some kind of interface with methods already implemented.

Mixins encourage the reuse of code (since the common functionalities get separated into their own classes), allowing for some interesting mechanisms and enforcing the Dependency Inversion principle.

Many times, Mixins are described as “included” rather than “inherited”, due to their nature.

### Random Trivia!

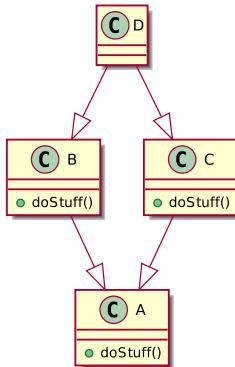
The python web framework Django makes heavy use of mixins in its class-based views: you can create a standard “View” (representing a web page, for instance), and then add login protection (via `LoginRequiredMixin`) or permissions (via `PermissionRequiredMixin`). This is all done using Python’s multiple inheritance.

A code example of mixins is beyond the scope of this book, since each language has its own way of implementing mixins, some easy (like Python), other a bit more complex (like C++, see “Curiously Recurring Template Patterns”, or C.R.T.P.).

## 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



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.

## 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]*

## 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...

The parts involved are defined as “coupled” because, even though they are separated, their maintenance very much behaves like they were a single entity. This also means that the elements that are coupled are harder to reuse, since they are so tightly related that they end up serving each other and nothing else.

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.

Reducing coupling is done by reducing interdependence, coordination and information flow between elements of a program.

Examples of coupling include:

- A module uses code of another module (this breaks the principle of *information hiding*);
- Many modules access the same global data;
- A module controls the flow of another module (like passing a parameter that decides “what to do”);
- Subclassing.

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.

## Note!

We may be tempted to try and “remove coupling completely”, but that’s usually a wasted effort. We want to reduce coupling as much as possible and instead improve “cohesion”. Sometimes coupling is unavoidable (as in case of subclassing). Balance is the key.

## 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.

## 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 O.O.P. - 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.

## “Composition over Inheritance” design

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

## 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 behavior but different animations and sprites.

These classes can inherit from an “entity” abstract class which defines the base behavior 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, using YAML:

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

Another very used alternative is JSON, which would look like this:

```
{  
    "entity": {  
        "name": "skeleton",  
        "health": 10,  
        "damage_on_hit": 2.5,
```

```

    "spritesheet": "./skelly.png",
    "animations": {
        "walking": {
            "start_sprite": 4,
            "frame_no": 4,
            "duration": 0.2
        },
        "attacking": {
            "start_sprite": 9,
            "frame_no": 2,
            "duration": 0.1
        }
    }
}

```

With more complex building algorithms, it is possible to change behaviors and much more with just a configuration file, and this gives itself well to rogue-like 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.

## 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.

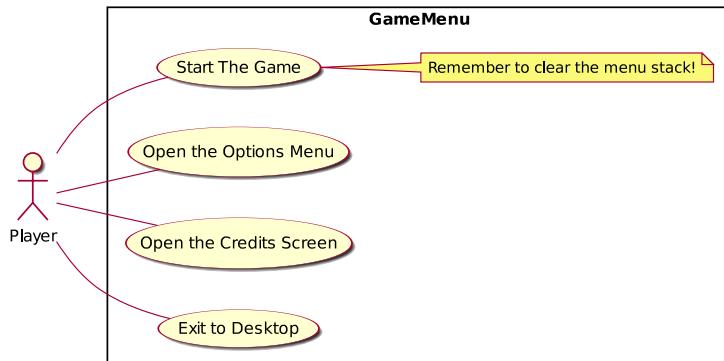
In this chapter we will take a look at 4 diagrams used in UML:

- Use Case Diagrams
- Class Diagrams
- Activity Diagrams
- Sequence Diagrams

## 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:

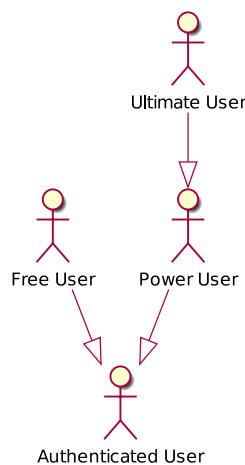


Example of a use case diagram

## 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.



## 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”.

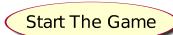
As seen, inheritance between actors is represented with a solid line with a hollow closed arrow. Such arrow points towards the “super-type” or “parent” from which the subject (or “sub-type”, or “child”) inherits.

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

## Use Cases

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

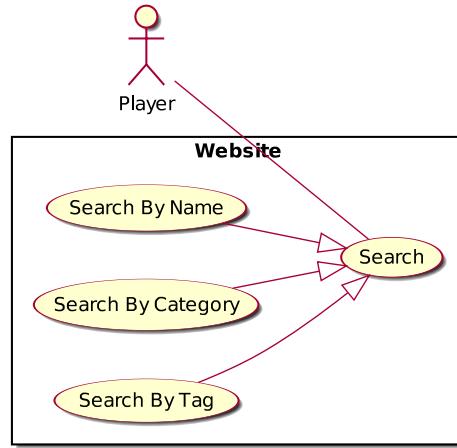
Use cases are represented with an ellipse with the name of the use case inside. Choosing the right name for a use case is extremely important, since they will represent the functionality that will be developed in our game.



Example of a use case

## 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.

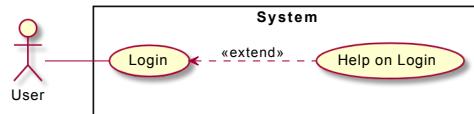


Example of a use case hierarchy

### Extensions

Use case extensions specify how and when optional behavior 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 behavior 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.



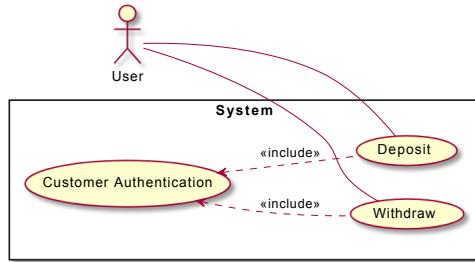
Example of a use case extension

### Inclusions

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



Example of a use case inclusion

## Notes

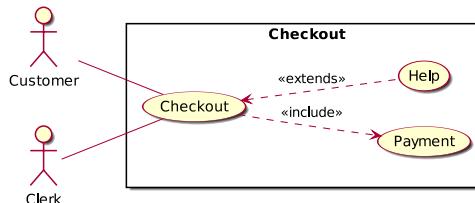
In use case diagrams, as well as in many other UML diagrams, notes are used to jot down conditions, comments and everything useful to better understanding the diagram that cannot be conveyed through a well definite structure inside of UML.

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.

## Sub-Use Cases

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

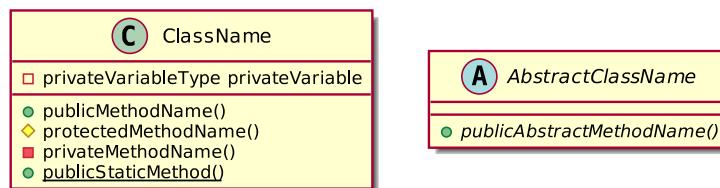


Example of a sub-use case

# Class Diagrams

## 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:



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*.

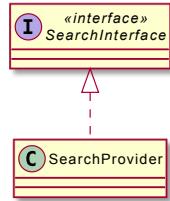
Public members are highlighted by a “+” symbol (or in our case, a green symbol) before their name, protected members use a “#” symbol (or a yellow symbol) and private members use a “-” symbol.

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

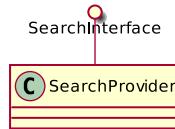
## Interfaces

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” (also called “interface realization”).



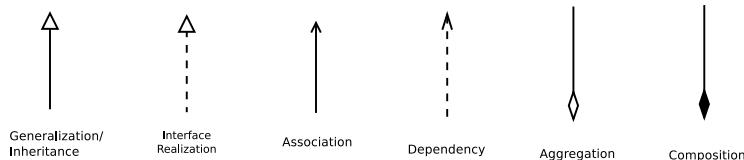
Defining an interface in UML



Interface Realization in UML

## Relationships between entities of the class diagram

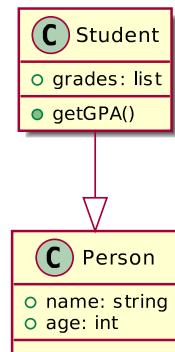
Expressing only single classes on their own doesn't give UML a lot of expressive power when it comes to planning your games. Here we'll take a quick look at the most used relationships between classes.



Relationships between classes in an UML Diagram

### 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 “super-type and sub-type” relationship.



## Example of inheritance in UML class diagrams

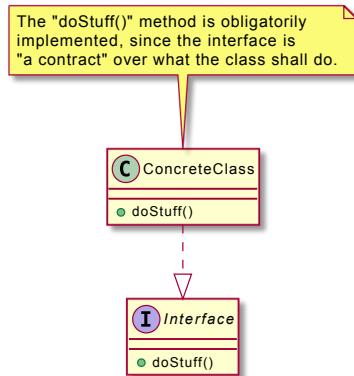
In this example we say that “Student IS-A Person” and inherits all Person’s methods and fields.

### Interface realization

Interface realization can be complex to understand at first, given its formal definition:

The interface realization relationship specifies that the realizing class must conform to the contract that the provided interface specifies.

In short, it means that the class is implementing all the methods specified by the interface (thus “realizing” it, as in making it real).

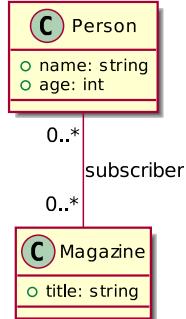


## Example of interface realization in UML class diagram

### 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.



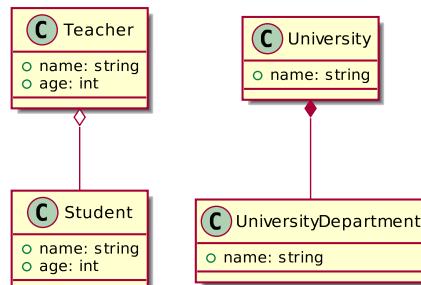
Example of association in UML class diagrams

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.

### 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 dependent 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.



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.

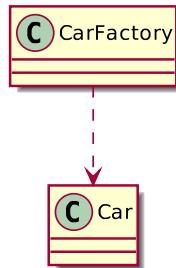
### Dependency

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.



Example of dependency in UML class diagrams

### 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.

## 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.



Example of an activity diagram

## Start and End Nodes

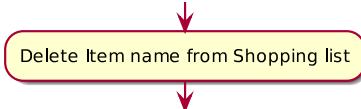
Each diagram begins what a “start node”, represented with a filled black circle, and they end with an “end node” which is represented with a filled black circle inside of a hollow circle.



Example of activity diagrams start and end nodes

## Actions

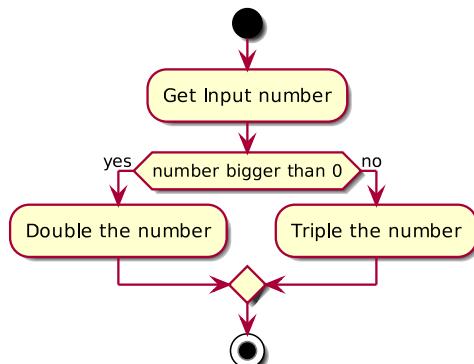
Each action taken by the software is represented in the diagram via a rounded rectangle, a very short description of the action is written inside the rounded rectangle space.



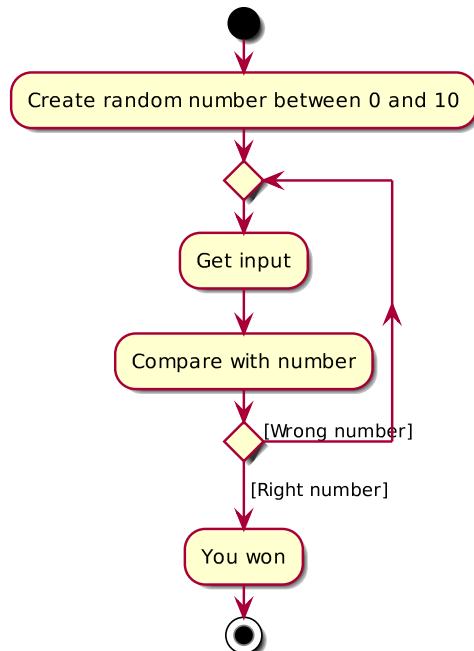
Example of Action in activity diagrams

## 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.

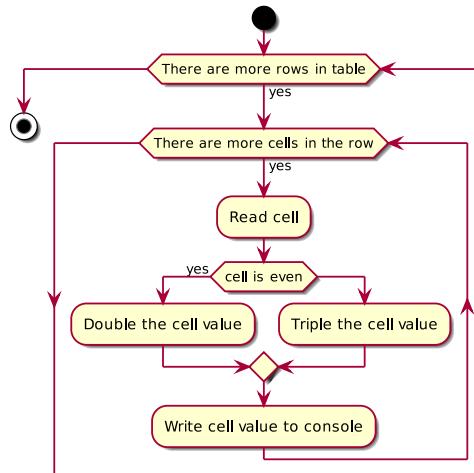


Example of decision, using hexagons to represent the condition



Example of loops, using guards to represent the condition

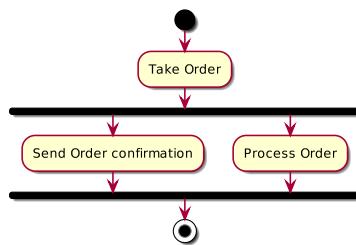
All the branches that depend on a condition or are part of a loop start and end on a diamond, as shown below.



Example of how nested loops and conditions are performed

## 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”

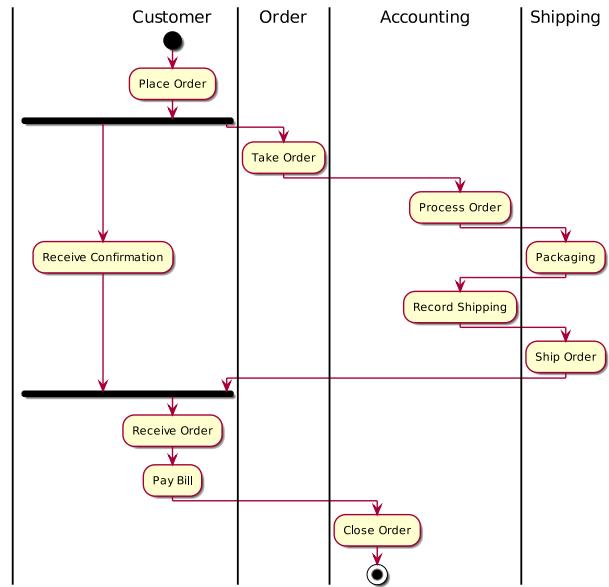


Example of concurrent processes in activity diagrams

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.

## 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.

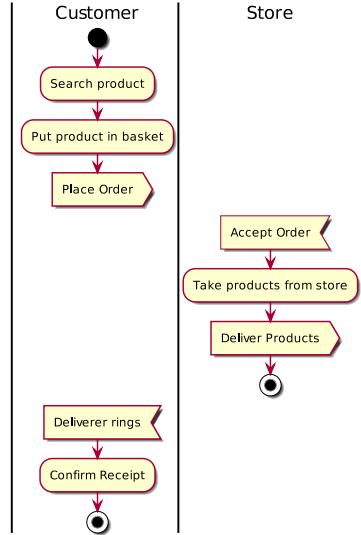


Example of swimlanes in activity diagrams

## 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.

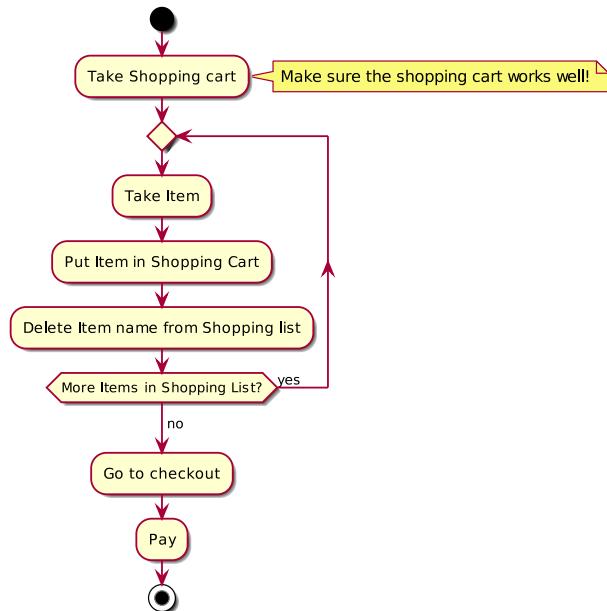
The “Sent Signal” symbol is represented with a convex pentagon (which reminds an arrow going away from our system), while the “Received Signal” is represented by a concave pentagon (which reminds a “slot” where the “sent signal” symbol can connect to).



Example of signals in activity diagrams

## 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.



Example of a note inside of an activity diagram

## A note on activity diagrams

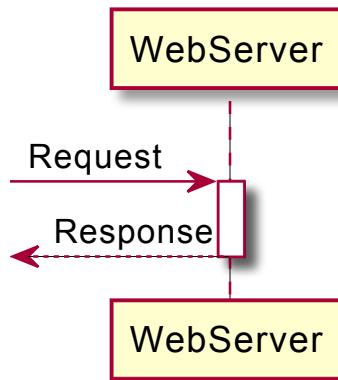
The components of activity diagrams shown here are just a small part of the used components, but they should be enough to get you started designing and reading most of the activity diagrams that exist.

## Sequence Diagrams

Sequence diagrams are used to represent how objects (called “participants”) interact with each other and such interactions are represented in a time sequence.

### Lifelines

The central concept of sequence diagrams are lifelines: they represent the time a participant is “alive” and when it is doing something.



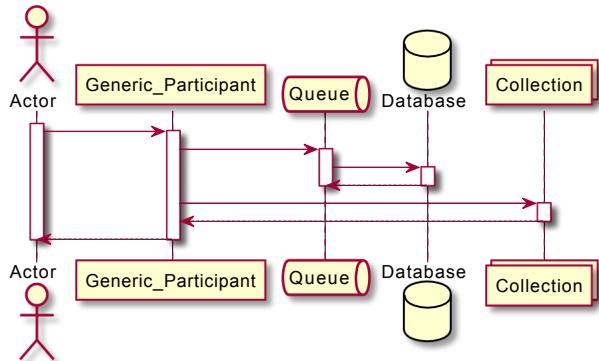
Example of a sequence diagram lifeline

The time flows from top to bottom, a dashed line represents the participant exists (for instance an object is instantiated in memory), while the rectangle that replaces the dotted line represents the participant being “active” (for instance one of the object’s functions is called).

### Participants

The participants don’t have to be actual classes, since sequence diagrams represent interactions at a “higher level” than mere code-bound planning.

Some UML drawing software allows for custom shapes for each participant, like the following:

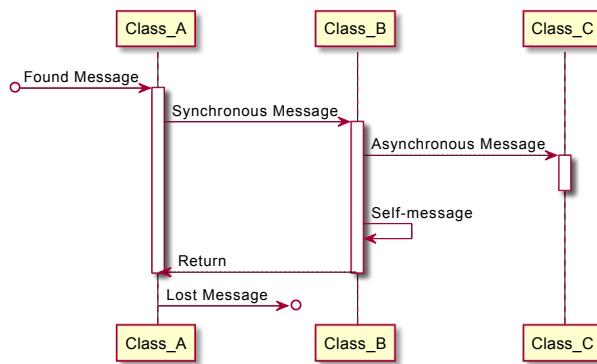


Some alternative shapes for participants

## Messages

Each object (represented by a lifeline) communicates with other objects (and the “outside”) through “messages”.

Messages are represented by arrows and an example can be seen here:



Messages in a sequence diagram

Let's analyze them one by one:

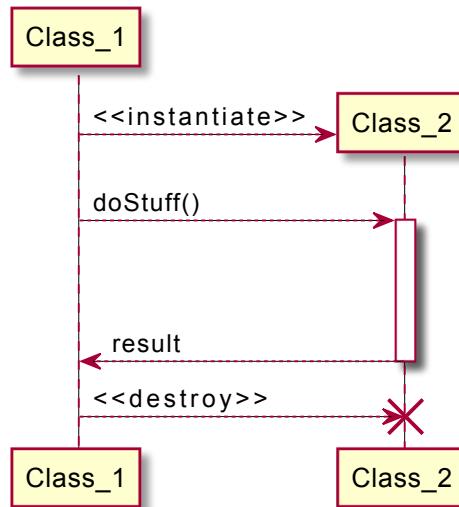
- **Found Messages** are messages that come from “outside”, from the perspective of the part of the system we are analyzing, they may come from another system or even the user.
- **Synchronous Messages and returns** are messages that activate a class and wait for a “return message”. These usually represent a synchronous

function call (but it can represent a more abstract concept).

- **Asynchronous Messages** are messages that activate a class but don't wait for a return value. These usually represent asynchronous function calls.
- **Self-messages** are messages from an object to itself, they usually represent an internal function call.
- **Lost Messages** are messages sent towards the “outside”, from the perspective of the part of the system we are analyzing.

## Object Instantiation and Destruction

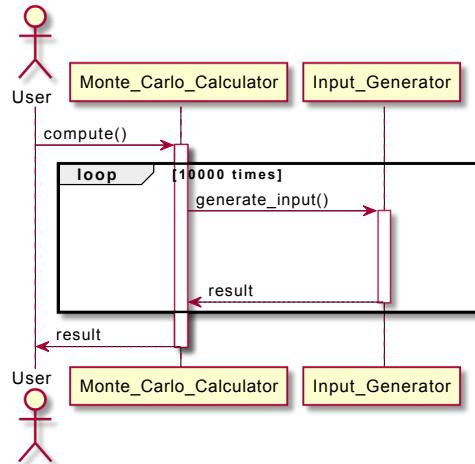
Sometimes it may be useful to represent the instantiation and destruction of objects in a sequence diagram. UML provides such facilities via the <<instantiate>>, <<create>> and <<destroy>> keywords, as well as a symbol for the destruction of an object.



Object instantiation and destruction in a sequence diagram

## Grouping and loops

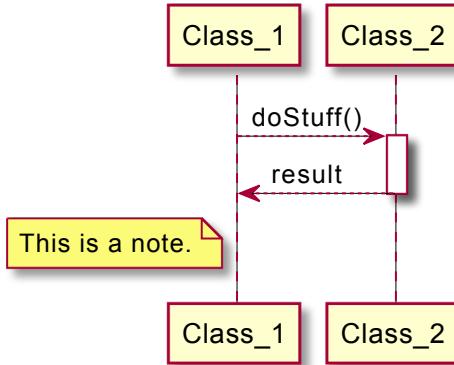
From time to time, we may need to represent a series of messages being sent in parallel, a loop, or just group some messages to represent them in a clearer manner. This is where grouping comes of use: it has a representation based on “boxes”, like the following:



A loop grouping in a sequence diagram

## Notes

Like all UML diagrams, it is possible to use notes to add some comments that may be useful for the interpretation of our diagrams, like follows.



Example of notes in a sequence diagram

## Other diagrams

UML is composed by a ton of diagrams that can be used to communicate with your teammates and organize your work, among them we find:

- Component Diagrams;
- Communication Diagrams;
- Composite Structure Diagrams;
- Deployment Diagrams;

- Package Diagrams;
- Profile Diagrams;
- State Diagrams;
- Timing Diagrams.

In this chapter we just saw the ones that will help you the most when reading the rest of this book, as well as effectively planning any project you have in mind.

## 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*<sub>g</sub>.

## 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.

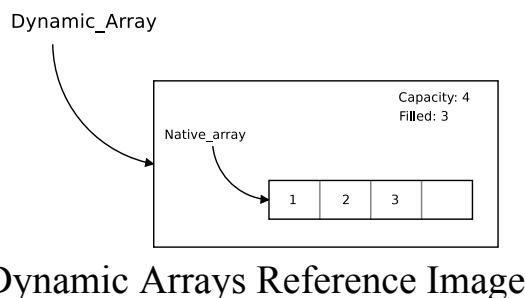
## Note!

This section will be purely theoretical and no code will be shown for any container, this is because implementations vary wildly between programming languages and some of these “advanced containers” are integrated in such languages.

## 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.

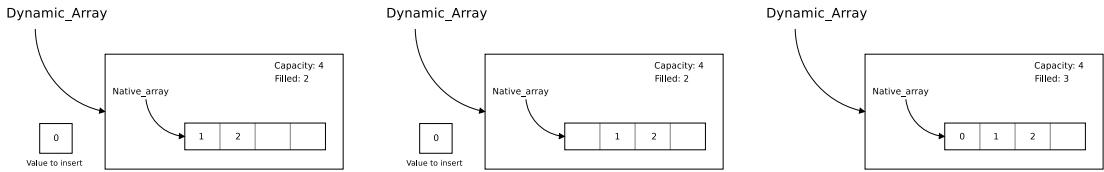


Dynamic Arrays Reference Image

## Performance Analysis

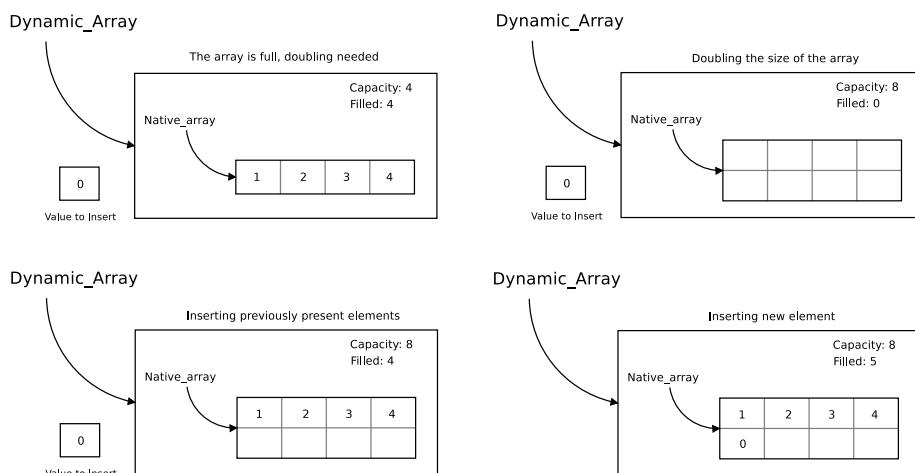
Indexing an item is immediate, since arrays allow to natively index themselves.

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  $O(n)$ .



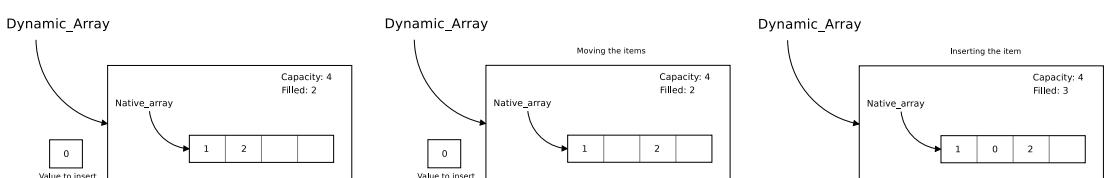
Adding an element at the beginning of a Dynamic Array

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  $O(1)$ .



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.



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.

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	$O(n)$

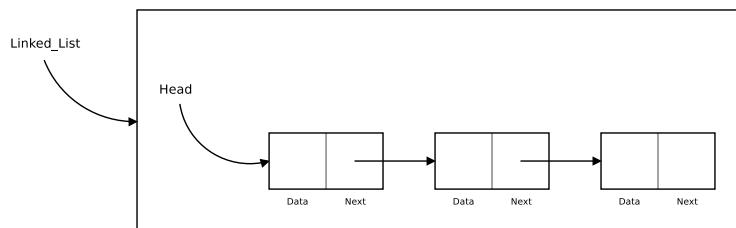
Summary Table for Dynamic Arrays

<b>Container Name</b>	Dynamic Array
<b>When To Use it</b>	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)
<b>Advantages</b>	Direct Indexing, Fast iteration through all the elements, given by the fact that arrays are stored compact in memory, fast appending.
<b>Disadvantages</b>	Slow insertions in arbitrary positions and at the head

of the array.

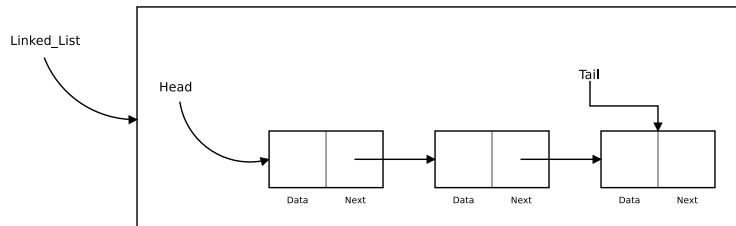
## Linked Lists

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.



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.

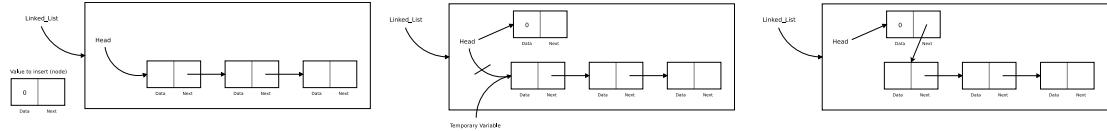


Double-Ended Linked List Reference Image

## Performance Analysis

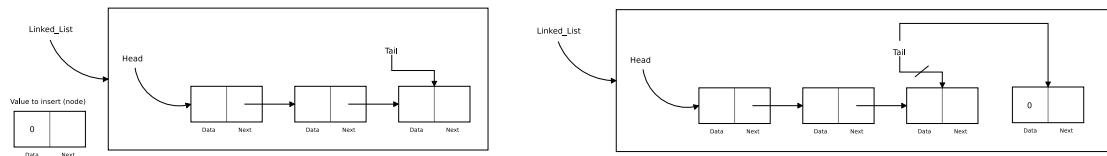
Since we only have a handler on the first node, indexing requires us to scan all the elements until we reach the one that was asked for. This operation has a potential time complexity of  $O(n)$ .

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  $O(1)$ .



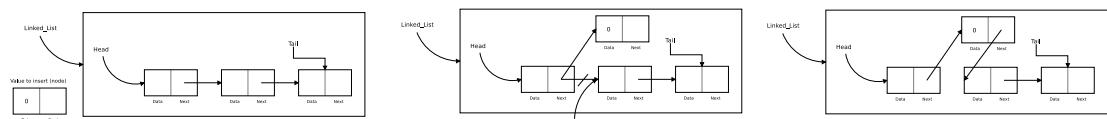
Inserting a new node at the beginning of a linked list

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  $O(n)$ ).



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.



Inserting a new node at an arbitrary position in a (double-ended) linked list  
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 + $O(1)$

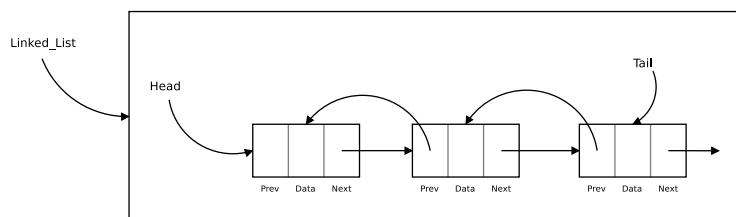
Summary Table for Linked Lists

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

## 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.



Doubly Linked List Reference Image  
Performance table for Doubly-Linked Lists

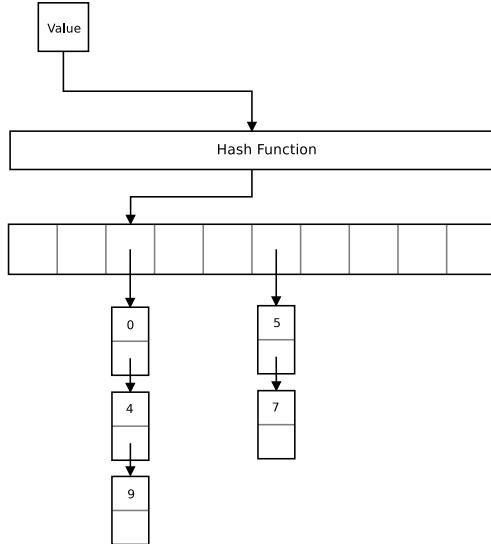
Operation	Average Cost
Indexing	$O(n)$
Insert/Delete At Beginning	$O(1)$

	<b>Operation</b>	<b>Average Cost</b>
	Insert/Delete At End	O(1)
	Insert/Delete at position	time to search + O(1)
		Summary Table for Linked Lists
<b>Container Name</b>	Doubly-Linked List	
<b>When To Use it</b>	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.	
<b>Advantages</b>	Very fast insertions/removals, quite fast iteration through all the elements. Possibility of easily iterating the list in reverse order.	
<b>Disadvantages</b>	Slow indexing at an arbitrary position. Sorting can be complex.	

## 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<sub>g</sub>* that will decide where the item will be positioned in the internal structure.



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  $O(1)$

Performance table for Hash Tables

<b>Operation</b>	<b>Average Cost</b>
------------------	---------------------

Searching	$O(1)$
-----------	--------

Insert	$O(1)$
--------	--------

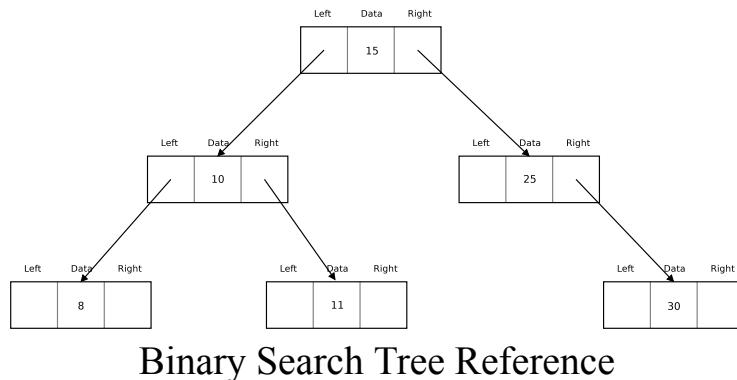
Delete	$O(1)$
--------	--------

Summary Table for Hash Tables

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

## Binary Search Trees (BST)

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



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  $O(\log(n))$ .

Performance table for Binary Search  
Trees

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

Summary Table for Binary Search Trees

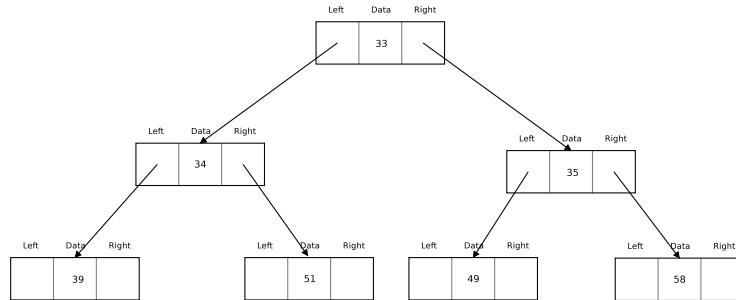
<b>Container Name</b>	Binary Search Tree
<b>When To Use it</b>	Situations that require good overall performance and requires fast search times.
<b>Advantages</b>	Good insertion and removal times, searching on this structure is fast.
<b>Disadvantages</b>	Given the nature of the data structure, there is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

## 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  $N$ ;



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.

Performance table for Heaps

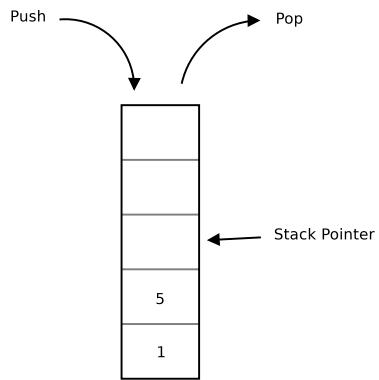
Operation	Average Cost
Find Minimum	$O(1)$ to $O(\log(n))$ , depending on the implementation
Remove Minimum	$O(\log(n))$
Insert	$\Theta(1)$ to $O(\log(n))$ depending on the implementation
When To Use it	All situations where you require to find and/or extract the minimum or maximum value in a container quickly; like priority queues.

Summary Table for Heaps

<b>Advantages</b>	Good general time complexity, maximum performance when used as priority queues.
<b>Disadvantages</b>	No inherent ordering, there are better solutions for general use.

## Stacks

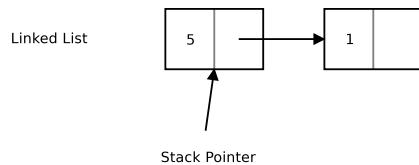
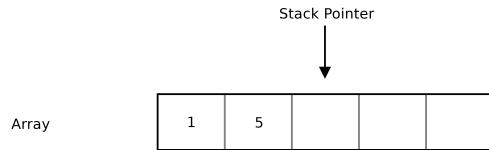
Stacks are a particular data structure, they have a limited way of working: you can only put or remove items on top of the stack, plus being able to “peek” on top of the stack.



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.

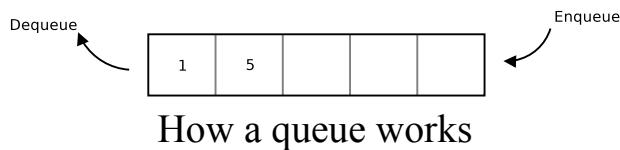
Depending on the single implementation, the operation used to “pop” an item from the stack will also return the element, ready to be used in an upcoming computation.



Array and linked list implementations of a stack

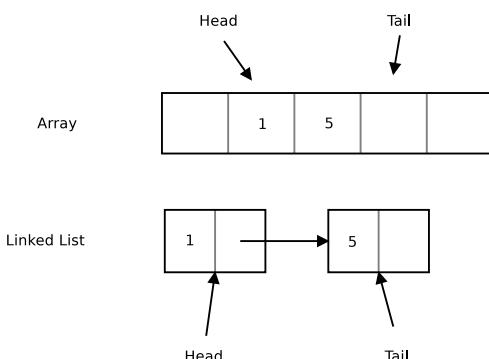
## Queues

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.



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.

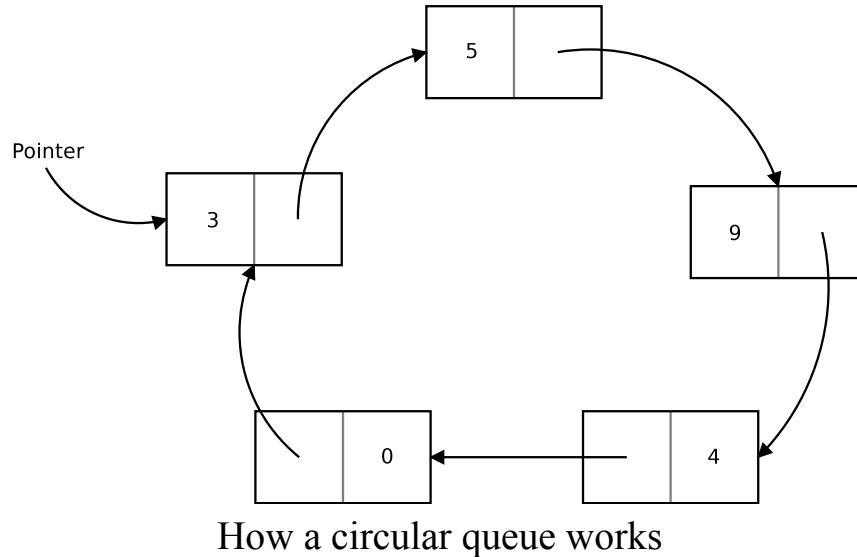
As with stacks, queues leverage limitations in their way of working for greater control over the structure itself. Usually queues are implemented via linked lists, but can also be implemented via arrays, using multiple indexes and index-wrapping when iterating.



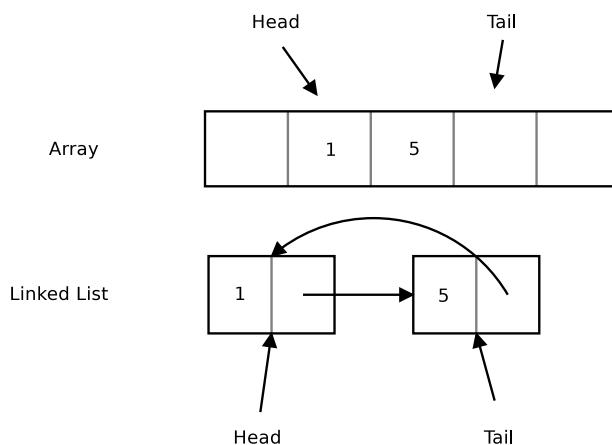
## Array and linked list implementation of a queue

### 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.



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



Array and linked list implementation of a circular queue

# Treating multidimensional structures like one-dimensional ones

This is usually done when dealing with pointers, but we may need to use some math to deal with sprites and animations too.

As we'll see in the [Sprite sheets section](#), it is more efficient to store sprites and animation frames in sprite sheets.

When dealing with frames of animation, we like our frames to be “one-dimensional”, each “cell” represents a certain “time”.

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

The “easy way” of dealing with frames

When dealing with sprite sheets, we may find that our animation has frames saved in a “matrix” of some sort, like so:

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	

A sample sprite sheet with the same frames as before

The images we've just seen will help you understand how the following formulas work.

To convert from 2-dimensional (*row, column*) coordinates to a single index, the formula is:

$$\text{index} = \text{width} \times \text{row} + \text{column}$$

**Note!**

Remember that in many programming languages arrays and similar structures are 0-indexed. This is the system that will be used here.

So if I want to know the index of the 3rd element of the second row, with index (2,1), the formula becomes:

$$index = 3 \times 1 + 2 = 5$$

The inverse formula is the following:

$$\begin{cases} row = \lfloor \frac{index}{width} \rfloor \\ column = index \% width \end{cases}$$

So if we wanted to know the (row,column) position of the frame with index 7 we would have:

$$\begin{cases} row = \lfloor \frac{7}{3} \rfloor = \lfloor 2.33333 \rfloor = 2 \\ column = 7 \% 3 = 1 \end{cases}$$

**Note!**

This can be done with structures with n dimensions, but the formula becomes a lot more complex the more dimensions you add. We'll stop at 2 for now.

## Data Redundancy

When dealing with certain structures, there are operations that are inherently complex to do: let's take for example counting the elements in a list:

```
class List{
    Node nodeList;
    // ...
    function getLength() -> int{
        int counter = 0;
        for (item in nodeList){
            counter = counter + 1;
        }
        return counter;
    }
}
```

Code: Counting the elements in a list

It's easy to see that an algorithm like this has a  $\Theta(n)$  complexity, which may not be ideal for an operation as common as finding the length of a list.

This is where data redundancy comes into play: the length of a list is an intrinsic property of the list itself, so why not save it inside the “head” of our structure?

This will obviously require a bit more work in all the methods that will change the number of elements inside the list, since we need to keep the “length” property in sync with the actual length of the list, but in exchange we can count the elements in a list by doing a simple lookup.

Let's see an example implementation:

```
class List{
    Node nodeList;
    int length;
    // ...
    function getLength() -> int{
        return length;
    }

    function addItem(Node node) {
        // ... Normal operation ...
        // ...
    }
}
```

```

        // We update our length counter
        length = length + 1;
    }

    function removeItem(Node node) {
        // ... Normal removal operation ...
        // ...
        // We update our length counter
        length = length - 1;
    }

    function clear() {
        // ... Normal clear operation ...
        // ...
        // We clear the length too
        length = 0;
    }

    // ...
}

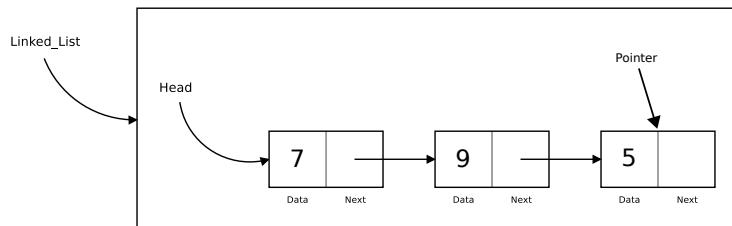
```

Code: Counting the elements in a list with data redundancy

### Pitfall Warning!

It is extremely important that we keep our “redundant properties” synchronized with the actual state of our objects, even when exceptions are raised. Not doing so will create bugs.

Let's consider another example: we have a standard linked list, like the one that follows:



Singly-Linked List has no redundancy

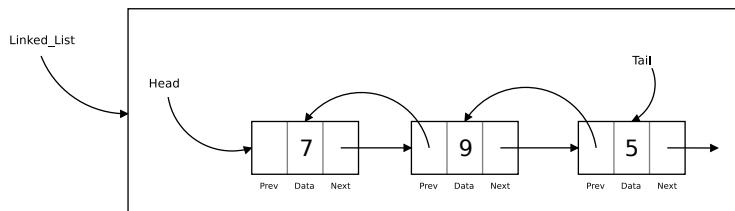
Our “pointer” is pointing the node containing the number “5”, and now we want to know the value of the node that precedes it. To do that we need to start from the head, saving in a temporary variable our nodes, until we find the node pointed by our “pointer”.

```
function get_previous_node(List lst, Node current_node) ->
Node{
    Node pointer = lst.head;
    Node previous = null;
    while (pointer != current_node) {
        previous = pointer;
        pointer = pointer.next;
    }
    return previous;
}
```

Code: Finding the previous element in a singly linked list

This operation has  $O(n)$  complexity, which is not great. If we wanted to print a list in reverse with such technique, the situation would be even worse.

Doubly-linked lists are another example of data redundancy. We are saving the content of the “previous” node, so that we can do a simple lookup with complexity  $O(1)$  and easily (and efficiently) do our “reverse printing”.



A doubly linked list is an example of redundancy

## Introduction to Multi-Tasking

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*<sub>g</sub>, the most used is *pre-emption*<sub>g</sub> where there are forced context switches between processes, to avoid one hogging the CPU.

## Coroutines

If you search for the word “coroutine” online, you will find a lot of extremely convoluted explanations involving the knowledge of the difference between *preemptive*<sub>g</sub> and *non-preemptive* multitasking, subroutines, threads and lots more. Let’s try to make sense of this.

First of all, coroutines are computer programs can run in multitasking (so it can run separated from our main game loop) which are used in non-preemptive multitasking. Differently from the preemptive style defined in the glossary, in non preemptive multitasking the operating system never forces a context switch, but it’s the coroutine’s job to **yield** the control over to another function.

Instead of “fighting for resources”, coroutines politely free the processor and give control of it to something else (could be the caller or another coroutine), this form of multitasking is often called **cooperative multitasking**.

A particularly interesting point of coroutines is the fact that their execution can be “suspended” and “resumed” without losing its internal state. Coroutines are used in more advanced engines (using the *Actor Model*) and in some particular situations. You may never need to use a single coroutine, or you may need to use them every day, so it’s worth knowing at least what they are.

## Introduction to Multi-Threading

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.

## What is Multi-Threading

Multi-Threading 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
- ...

Multi-Threading is also useful when we want to perform concurrent execution of activities.

## Why Multi-Threading?

Many people think of Multi-Threading as “parallel execution” of tasks that leads to faster performance. That is not always the case. Sometimes Multi-Threading 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?

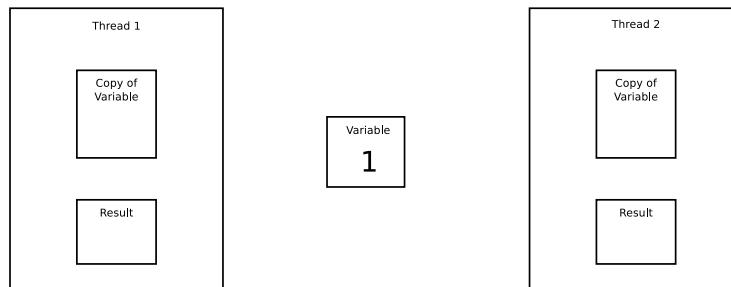
The update function will keep running until all the resources are loaded, the game loop is stuck and no drawing will be executed until the loading has finished. The game is essentially hung, frozen and your operating system may even ask you to terminate it. In this case we need the main game loop to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

## Thread Safety

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

## Race conditions

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

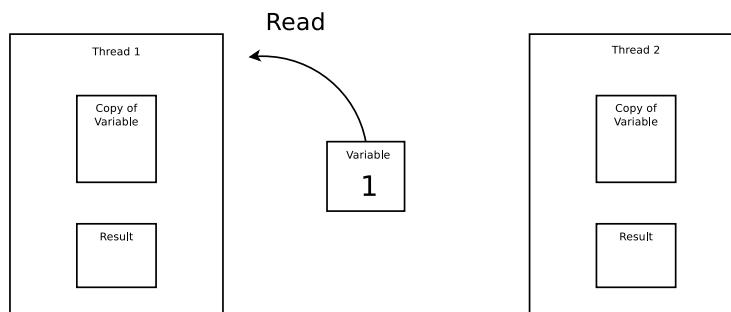


Two threads and a shared variable

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.

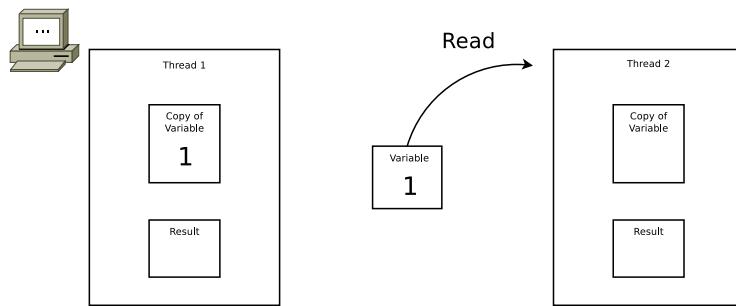
This seems simple enough for us humans, but there is a situation that can be really harmful: let's see, in the following example each thread will be executed only once. So the final result, given the example, should be “3”.

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



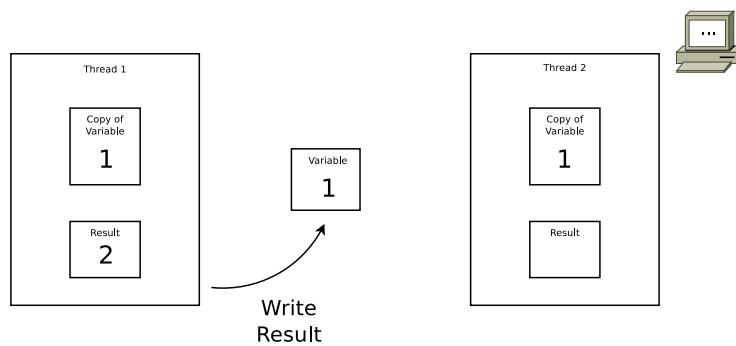
Thread 1 reads the variable

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



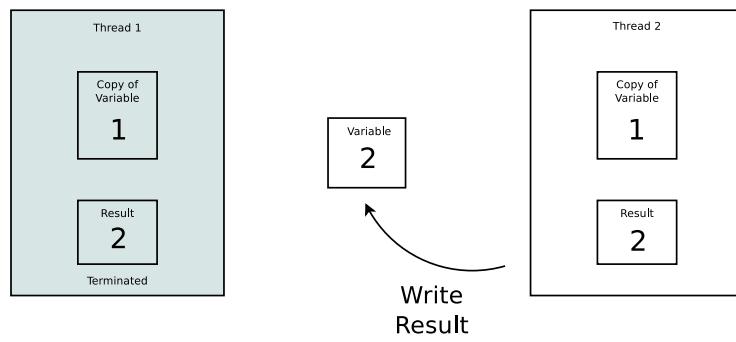
While Thread 1 is working, Thread 2 reads the variable

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



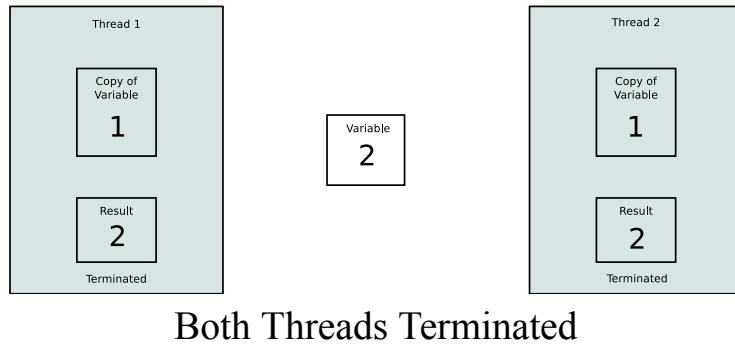
Thread 1 writes the variable

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



Thread 2 writes the variable

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



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).

## 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 behaviors. Such sections must be protected from concurrent access, which means only one process or thread can access them at one given time.

## Ensuring determinism

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

## 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.

## Mutex

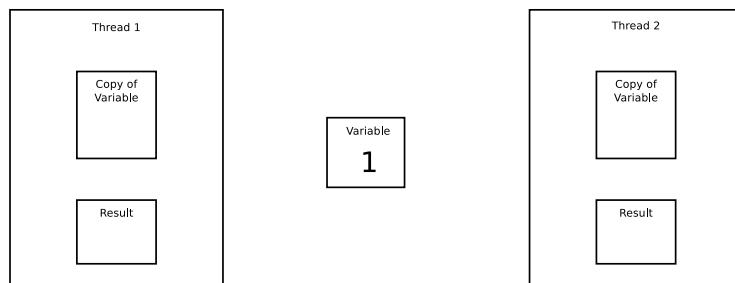
Mutex (Short for **mutual exclusion**) means that the access to the shared data is serialized in a way that only one thread can read or write to such data at any given time. Mutual exclusion can be achieved via algorithms (be careful of *out of order execution*), via hardware or using “software mutex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

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

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

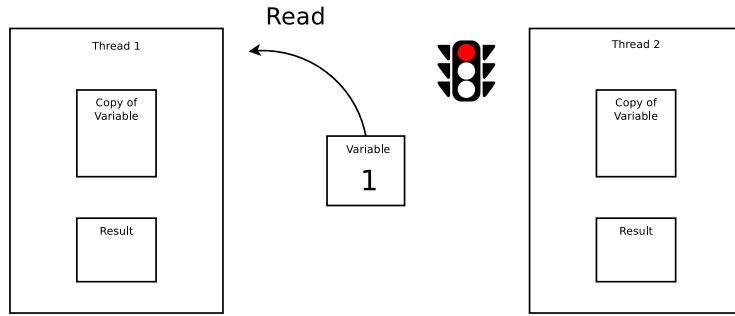
Let's see how Mutex solve our concurrency problem.

As seen before, we have a shared variable and two threads that want to add one to it.



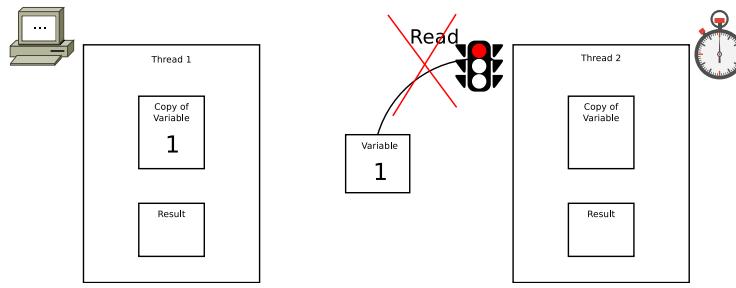
How mutex works (1/8)

Now the first thread reads the variable and “locks” the mutex (thus stopping other threads from accessing the variable).



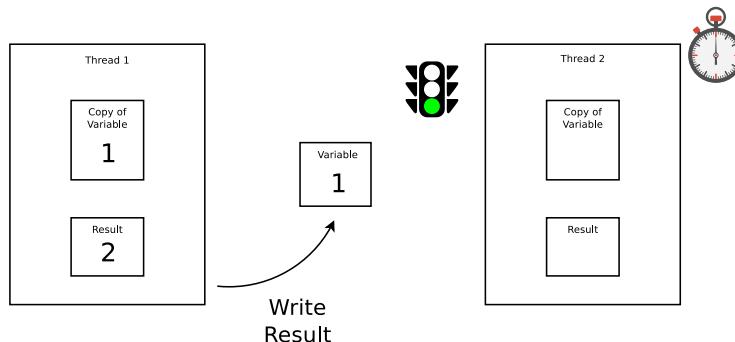
How mutex works (2/8)

When the second thread wants to access the “critical region”, it will check on the Mutex, find it “locked” and be forced to wait: it cannot read the variable, because we would have a “race condition” otherwise.



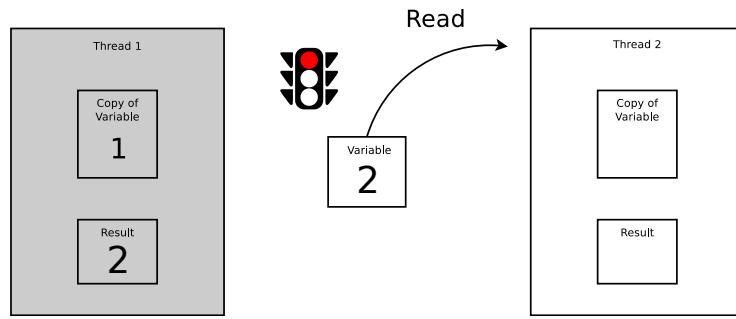
How mutex works (3/8)

As soon as the first thread finishes its job, it will write the result in the variable and “unlock” the mutex, allowing others to access the variable.



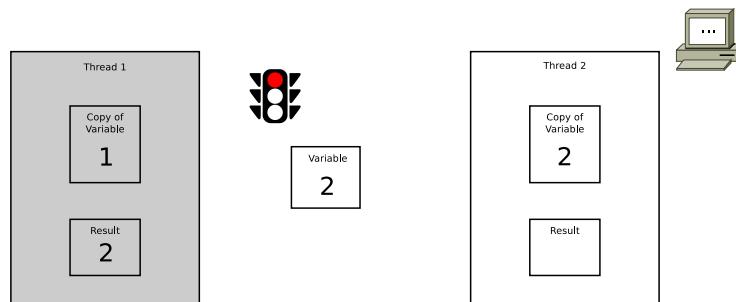
How mutex works (4/8)

Since the second thread was waiting, it will read the variable result (now 2) and “lock” the mutex for safety. The second thread entered the “critical region”.



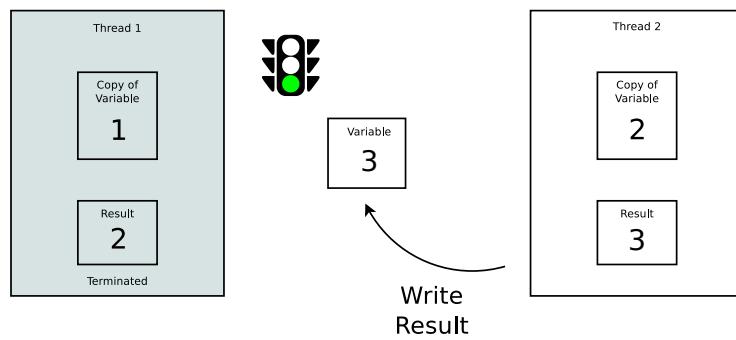
How mutex works (5/8)

The second thread will do its job as normal, if a third thread tried to access the variable, it would be stopped by the locked mutex.



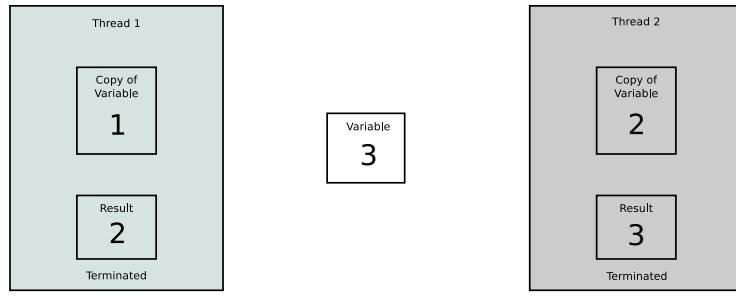
How mutex works (6/8)

When its job is done, the second thread will write to the variable and “unlock” the Mutex, thus allowing other threads or processes to access the variable.



How mutex works (7/8)

Now both threads finished their jobs and the result inside the variable is correct.



How mutex works (8/8)

## Atomic Operations

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

# A Game Design Dictionary

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. The objective of this chapter is giving you some terminology and knowledge about game design, before deep-diving into the topic.

## Platforms

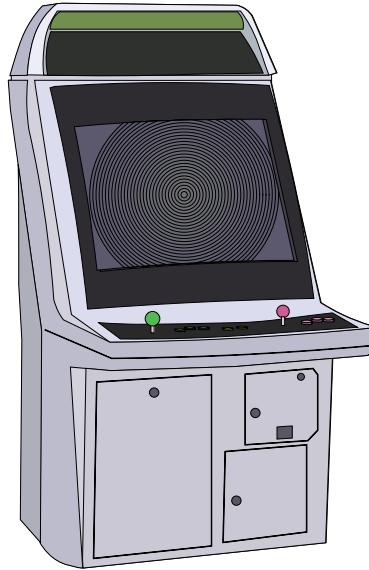
There are several different platforms a game can be developed for, and each one has its own advantages and drawbacks. Here we will discuss the most notable ones.

### 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 (sometimes due to the fact that you’re popping quarters into the machine for the “right to play”), 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.



How an arcade machine usually looks like

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.

## 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:

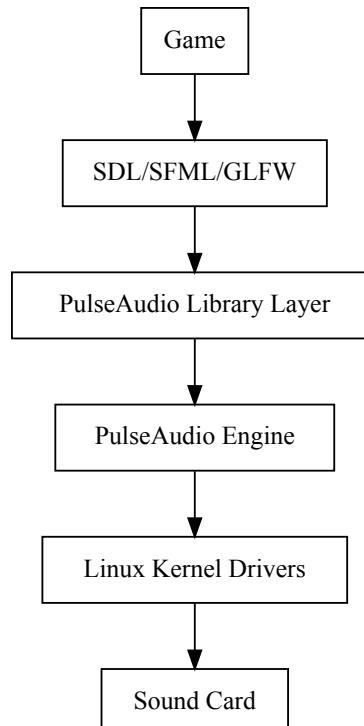
1. New Generation is introduced
2. Initial confusion, with poor performance and graphics

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.

## 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 compounds with the fact that “abstraction layers” used by the developer (like SDL, SFML or GLFW) are running on top of other “abstraction layers”, like sound servers, device drivers, etc... which can be littered with bugs themselves. Just look at how many indirections we have on a modern Linux system (which is usually bundled with PulseAudio):



How many abstraction layers are used just for a game to be able to play sounds

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.

## **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.

## **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).

## **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.

### **Mouse and Keyboard**

One of the most common input devices, most of the currently available frameworks and engines 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”.

## **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.

## **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 frameworks 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.

## **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.

## 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.

## 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).

### 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.

## 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;

## 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.

## 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.

## **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).

## **Simulation**

Simulation games cover a huge variety of games that are created 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.

## **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).

## **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.

The more modern versions feature more interactive components and fully-fledged 3D graphics, but what ties the genre together is usually a “point and click” style of gameplay.

# **Part 2: Project Management**

# **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.

## **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.

### **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.

Under the term “project manager” you can find different roles, among them:

- Product Manager;
- Assistant Producer;
- Executive producer.

A good project manager will need tools to manage tasks (Like a Kanban Board), as well as tools that promote communication in the team (Chats, VoIP) and information repositories (having all information in the same place is important!).

## **Game Designer**

The game designer takes care of the game concept, usually (but not only!) 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.

Under the “Game Designer” term you can find different roles, among them:

- Level Designer;
- World Builder;
- Narrative Designer;
- Quest/Mission Designer.

A good game designer must know mathematics, some scripting and be able to use planning tools (again, our friendly Kanban Board comes into play) as well as diagram drawing tools.

## **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”).

Under the term of “Writer” you can find more roles, like:

- Editor;
- Narrative Designer;
- Creative Writer.

A good writer must have good language skills, as well as creativity. They must be able to use planning programs (like everyone, communication is important) as well as writing programs, like LibreOffice/OpenOffice Writer.

## **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;
- ...

They must be familiar with IDEs and programming environments, as well as Source Control Tools (Like Git), knowledge of game engines like Unity is preferred, but also tied to the kind of game that is made.

## **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;
- Environment Artists;
- UI Artists/Designers;
- Conceptual Artists.

In 3D games:

- 3D Modelers;
- Texture Artists.

Visual Artists must be knowledgeable in the use of drawing programs, like Krita, GIMP or their commercial counterparts.

## **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.

Under the umbrella of a sound artist, you can find:

- Audio Engineers;
- Game Composers;
- Music Mixers;
- Audio Programmers.

The knowledge of DAW (Digital Audio Workstation) software is fundamental, as well as knowing some so-called “middlewares”, like FMOD. Another important bit of knowledge is being able to use Audio editors effectively.

## **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 misbehaviors of your game.

## **Some generic tips**

### **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 <insert something here>: 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.

### **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.

That may seem a lot like an exaggeration, but unforeseen events happen and they can have a huge impact on the release of your game. It's better to err on the side of caution and even delay the release if something goes wrong. Shigeru Miyamoto said the following:

| A delayed game is eventually good, a bad game is bad forever.

so maybe being “abundant” with your time estimates is not that wrong.

## **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.

## **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 needs 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.

## 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.

### 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.

### 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.

## **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, get feedback and encouragement. Create trailers towards the end of development, put them on YouTube or Vimeo and if you want to go all out, get in touch with the press (locally first) and create hype around your product.

## **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.

## Misusing of the Digital Millennium Copyright Act

This is what could be considered the apex of mishandling criticism: the usage of DMCA takedowns to quash criticism towards your game.

### Note!

What follows **is not legal advice**. I am not a lawyer.

If you want to know more (as in quantity and quality of information), contact your favorite lawyer.

Sadly, mostly in the YouTube ecosystem, DMCA takedowns are often used as a means to suppress criticism and make video-reviews disappear from the Internet. Useless to say that this is **potentially illegal** as well as **definitely despicable**.

Takedowns according to the DMCA are a tool at your disposal to deal with copyright infringements by people who steal part (or the entirety of) your work, allowing (in the case of YouTube at the very least) to make the allegedly infringing material. This should be used carefully and just after at

the very least contacting the alleged infringer privately, also because there is an exception to the copyright rule.

### The Fair Use Doctrine

The so-called “Fair Use” is a limited exception to the copyright law that targets purposes of review, criticism, parody, commentary, and news reporting, for instance.

The test for “Fair use” has four factors (according to 17 U.S.C. §107):

1. **The Purpose and character of the use:** if someone can demonstrate that their use advances knowledge or the progress of arts through the addition of something new, it’s probably fair use. This usually is defined by the question “is the work **transformative** enough?”
2. **The nature of the copyrighted work:** For instance, facts and ideas are not protected by copyright, but only their particular expression or fixation is protected. Essentially you can’t really sue someone for making a game very similar to yours (For instance making a 2D sidescrolling, run’n’gun platformer).
3. **The amount and substantiality of the portion used in relation to the work as a whole:** If someone uses a small part (compared to the whole) of the work, and if that part is not really substantial, then it’s probably fair use.
4. **The effect on the potential market for the copyrighted work:** this defines if the widespread presence of the “allegedly infringing use” can hinder on the copyright owner’s ability to exploit (earn from) their original work.

There can also be some additional factors that may be considered, but these four factors above are usually enough to decide over the presence (or absence) of fair use.

### The “Review Case”

Let’s take a simple example: a video-review on our brand new videogame, that takes some small pieces of gameplay (totaling about 5 minutes), on

video and comments on the gameplay, sound and graphics. A very common scenario with (I hope) an unsurprising turnout.

Let's take a look at the first point: the purpose is criticism, the review brings something new to the table (essentially **it is transformative**): someone's impression and comments about the commercial work.

Second point: the game is an interactive medium, while the review is non-interactive by nature, the mean of transmission is different.

Third point: considering the average duration of 8 to 10 hours of a videogame, 5 minutes of footage amounts for around 0.8% to 1% of the total experience, that's a laughable amount compared to the total experience.

Fourth Point: this is the one many people may get wrong. A review can have a huge effect on the market of a copyrighted work (a bad score from a big reviewer can result in huge losses), but that's not really how the test works. The fourth test can usually be answered by the following questions:

What's the probability that someone would buy (or enjoy for free) the work from the alleged infringer, instead than from me (the copyright owner)?

This is called “being a direct market substitute” for the original work. The other question is:

Is there a potential harm (other than market substitution) that can exist?

This usually is related to licensing markets. And here lies the final nail on the coffin: there is no direct market substitution and courts recognize that certain kinds of market don't negate fair use, and reviews are among those kinds of market. In essence **Copyright is not a shield against adverse criticism**.

## 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.

## **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?

## **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.

## 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.

### 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.

## 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.

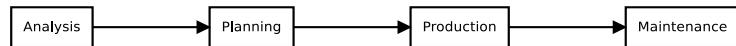


Diagram of the waterfall life cycle model

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.

## 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.

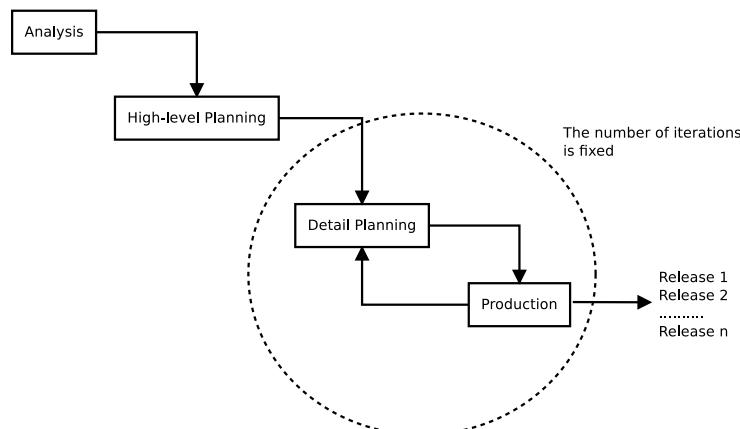


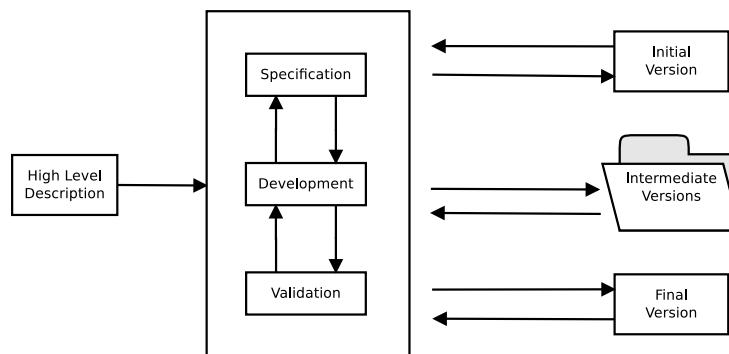
Diagram of the incremental life cycle model

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.

## 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.



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.

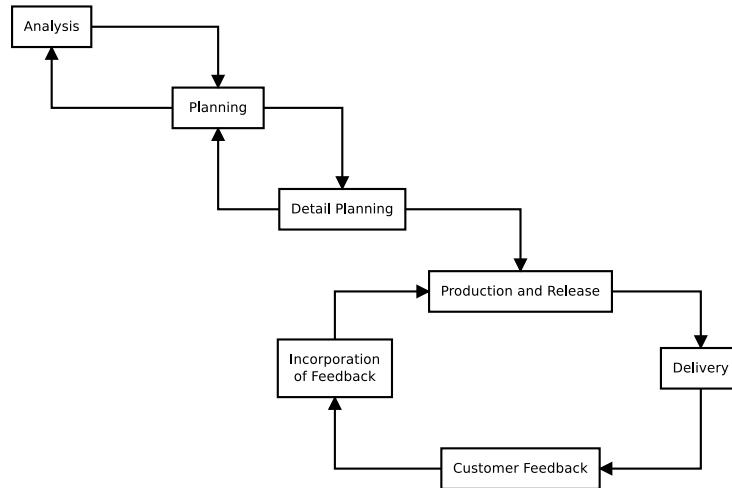


Diagram of the evolutionary life cycle model

## 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.

## 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.

## 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.

#### 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.

## 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.



Example of a Kanban Board

## 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.

## Lean Development

Lean development tries to bring the principles of lean manufacturing into software development. The basis of lean development is divided in 7 principles:

- **Remove Waste:** “waste” can be partial work, useless features, waiting, defects, work changing hands...
- **Amplify Learning:** coding is seen as a learning process and different ideas should be tested on the field, giving great importance to the learning process;
- **Decide late:** the later you take decisions, the more assumptions and predictions are replaced with facts. Also strong commitments should happen as late as possible, as they will make the system less flexible;
- **Deliver early:** technology evolves rapidly, and the one that survives is the fastest. If you can deliver your product free from defects as soon as

possible you will get feedback quickly, and get to the next iteration sooner;

- **Empower the team:** managers are taught to listen to the developers, as well as provide suggestions;
- **Build integrity in:** the components of the system should work well together and give a cohesive experience, giving the customer and impression of integrity;
- **Optimize the whole:** optimization is done by splitting big tasks into smaller ones which helps finding and eliminating the cause of defects.

## 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.

## 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")
```

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.

## 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.

### SLOC

This is probably the simplest metric out there: The “Source Line of Code” (SLOC). It is used to measure the size of a program by counting its lines of code. Once Bill gates said the following:

Measuring programming progress by lines of code is like measuring aircraft building progress by weight.

An aircraft must be lightweight and robust, and being heavier than necessary will stop it from flying. The same reasoning should be applied

here: a longer source code doesn't mean a better product.

It is important to strike a balance between “readability” and “brevity”: your code should be short, but being source code, it is still meant for humans to read, so readability matters more than brevity.

Usually the SLOC metric is used to give a “order of magnitude” impression of the program: considering 2 programs that do exactly the same thing, one is 10.000 lines of code, the other one is 100.000, you may start to suspect that the bigger program is more (probably uselessly) complex and less maintainable.

## Cyclomatic Complexity

More precisely called “McCabe’s Cyclomatic Complexity”, this metric defines the number of linearly independent paths through a program’s source code: the higher the metric, the higher is the number of paths a piece of code can take in its elaboration.

This means that a higher number of paths takes into account a higher number of conditions and decisions and when such number becomes too high, the code becomes hard to maintain.

The maximum complexity suggested is 10, although sometimes it’s good to relax such metric to a maximum of 15. When the cyclomatic complexity becomes higher than the maximum value, it is suggested to split the module into smaller, more maintainable modules.

Your IDE, if advanced enough, should already be able to warn you of a high cyclomatic complexity.

### Pitfall Warning!

Be mindful that cyclomatic complexity may have issues of “over-estimation” or “under-estimation”, depending on a case-by-case basis. McCabe’s cyclomatic complexity is far from a “silver bullet” that will

suit all your needs, but as all other metrics, it can give a pointer over where refactoring may be necessary.

## Code Coverage

When you have a test suite, you may already be thinking about a metric that tells you how much of your code is tested. Well, here it is: the *code coverage* metric tells you what percentage of your code base has been run when executing a test suite.

That is both the useful and damaging part of this metric: *code coverage* doesn't tell you **how well** your code is tested, just **how much code was executed**, so it's easy to incur into what I like to call "incidental coverage": the code coverage presents a higher value, when the code is merely "executed" and not thoroughly "tested".

Code coverage is split in many "sub-sets", like:

- **Statement Coverage:** how many statements of the program are executed;
- **Branch Coverage:** defines which branches (as in portions of the if/else and "switch" statements) are executed;
- **Function Coverage:** how many functions or subroutines are called.

This is also why it's better to prepare unit tests first, and delay the integration tests for a while.

To know more about those terms, head to the [testing section](#).

## Code Smells

Code Smells is a blanket term representing all the common (and thus known) mistakes done in a certain programming language, as well as bad practices that can be fixed more or less easily.

Some of these smells can be automatically detected by static analysis programs (sometimes called Linters), others may require dynamic execution, but all code smells should be solved at their root, since they usually entail a deeper problem.

Among code smells we find:

- Duplicated Code;
- Uncontrolled Side Effects;
- Mutating Variables;
- God Objects;
- Long Methods;
- Excessively long (and thus complex) lines of code.

## Coding Style breaches

When you are collaborating with someone, it is absolutely vital to enforce a coding style, so that everyone in the team is able to look at everyone else's code without having to put too much effort into it.

Coding style can be enforced via static analysis tools, when properly configured.

Counting (automatically) the number of coding style breaches can help you estimate how much effort working on the code is necessary, thus you would be able to foresee slowdowns in the development process.

## Depth of Inheritance

Some people say that inheritance is evil and should be avoided, some other say it's good. As with all things, *in medio stat virtus* (virtue stands in the middle), sometimes inheritance is better left where it is, other times its usage is necessary for things to make sense.

The *depth of inheritance* metric tells us how deep the inheritance hierarchy is, thus this metric will tell us the strength of one of the possible dependency types. The deeper the inheritance, the more dependencies we

have, which means that we have more classes that, if edited, will change the behavior of the “children classes”.

It’s better having a short inheritance depth, (although it’s not necessarily wrong) having a longer chain of dependencies might mean we have a structural problem, where some classes are “too generic” and at the top of the hierarchy we have some kind of “universal object”.

## **Number of methods / fields / variables**

Let’s talk numbers: having too many methods or fields in a class can be an indicator of a so-called “god object”: an object that has too many responsibilities under its wing (does too many things), this is a breach of the *single responsibility principle* and should be avoided.

We can fix this by splitting the class into smaller classes, each with its own single responsibility.

A high number of local variables instead may point to a complexity issue: your algorithm may be more complex than needed, or needs to be split into different functions.

## **Number of parameters**

This metric is specific for functions, when a function has a lot of parameters, it’s harder to call and harder to understand. Functions should have no more than 5 parameters in most cases, more and it will be complex.

Some automated tools in your IDE may be able to warn you in case methods and functions have too many parameters.

To solve this issue, you may need to review the function (maybe it has too many responsibilities?) or pass a so-called “complex structure” to it (thus merging all the parameters into one).

## **Other metrics**

The metrics listed above are not the only ones available to you, some IDEs have aggregated metrics (like the “maintainability index” in Visual Studio), while there may be other metrics you want to measure, some follow:

- **Lead Time:** Time elapsed between the start and end of a process (may be a ticket, or a task);
- **MTBF:** (Mean Time Before Failure) represents the mean time before the software crashes;
- **Crash Rate:** The number of times a software crashes, over the number of times it's used.

# 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.

## 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<sup>g</sup>;
- A Kanboard <sup>g</sup>.

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”.

## 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.

## 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:

## 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.

## 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.

## 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.

## 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**.

...

## 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.

## 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.

## 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

## 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 color 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 color 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.

...

## 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)

...

## 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, the 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.

## **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.

## 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.

## 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.

...

## 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.

## 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.

## 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.

## 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 colors to be colorblind-friendly: such mode will include 3 options: Deutanopia, 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.

## 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

## 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.

## 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

## 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 +

| ...

## 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.

## 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

## 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

## 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.

# **Part 3: Game Development Basics**

# The Game Loop

All loops are infinite ones for faulty RAM modules.

---

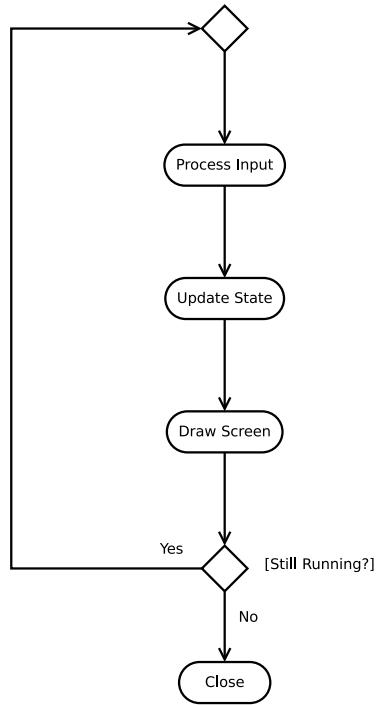
Anonymous

## 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



UML Diagram of the input-update-draw abstraction

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

```

function game() {
    bool game_is_running = True;
    while (game_is_running) {
        process_user_input();
        update_world();
        draw();
    }
}
  
```

Code: Game Loop example

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

# Input

## Events vs Real Time Input

Some frameworks may be able to further abstract how they process input by giving an *API<sub>g</sub>* 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.

This usually depends on the implementation of the event queue: 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.

### Note!

A well-configured event-based input system **is the most efficient way of doing things**, allowing code to be executed only when 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.

Again, 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](#).

## 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.

### 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.

### 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  $dt = \frac{1}{60}$ ):

```
float dt = 1.0/60.0;
bool game_is_running = True;

while (game_is_running){
    process_user_input();
    update_world(dt);
```

```
        draw();
    }
```

### Code: Game loop with fixed timesteps

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.

## 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:

```
bool game_is_running = True;

while (game_is_running) {
    float dt = measure_time_from_last_frame();
    process_user_input();
    update_world(dt);
    draw();
}
```

### Code: Game loop with variable time steps

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  $dt$  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  $dt$ .

## 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  $dt = \frac{1}{60}$ ):

```
float dt = 1.0/60.0;
bool game_is_running = True;

while (game_is_running) {
    float frametime =
measure_time_from_last_frame();

    while (frametime > 0.0) {
        float deltaTime = min(dt, frametime);
        process_user_input();
        update_world(deltaTime);
        frametime = frametime - deltaTime;
    }
    draw();
}
```

Code: Game loop with Semi-Fixed time steps

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.

## 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  $dt = \frac{1}{60}$ ):

```
float targetTime = 1.0/60.0;
bool game_is_running = True;

while (game_is_running) {
    float dt = measure_time_from_last_frame();
    process_user_input();
    update_world(dt);
    draw();
    wait(targetTime - time_spent_this_frame());
}
```

Code: Game loop with Frame Limiting

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.

## 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.

## 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	1	2	3	4	5	6
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.

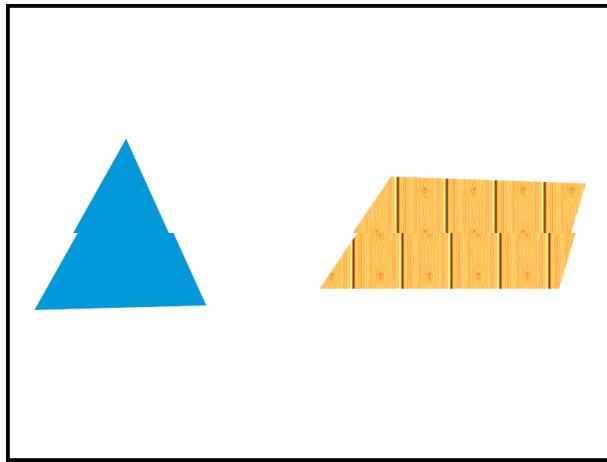
## Issues and possible solutions

In this section we have a little talk about some common issues related to the game loop and its timing, and some possible solutions

### 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.



An example of screen tearing

A very common 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.

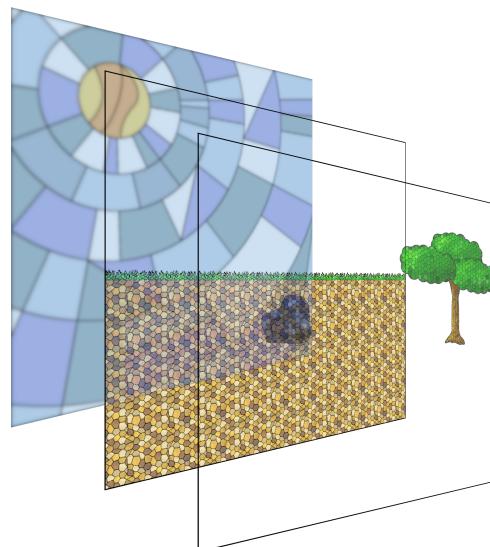
To smooth the game, a technique called “triple buffering” can be used, which adds a third color buffer is used to make the animation smoother at the cost of a higher input lag.

## 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

If we divide each “layer” we can see how the painter’s algorithm works:

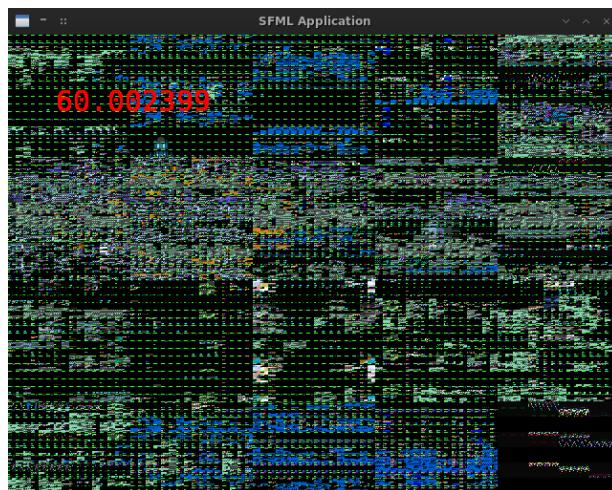


## A small example of the “painter’s algorithm”

Just like a real 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.

## Clearing the screen

Special note about 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.



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.

# 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.

## Why Collision Detection is done in multiple passes

Collision detection algorithms can be quite costly, even more when you are using a [brute force approach](#), but 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 building quad trees.

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.

### Note!

In this chapter we'll see the easier narrow-pass detection first, followed by the more complex broad-pass algorithms, but remember that a good collision detection system does a “broad-pass” first, before delving into the “narrow-pass”.

## Narrow-Phase 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.

### 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:

```
function point_collision(point A, point B) -> bool{
    if (A.x == B.x AND A.y == B.y) {
        return True;
    } else{
        return False;
    }
}
```

Code: Point to point collision detection

A possible lazy/shorter version could be:

```
function point_collision(point A, point B) -> bool{
    return A.x == B.x AND A.y == B.y;
}
```

Code: Shortened version of a point to point collision detection

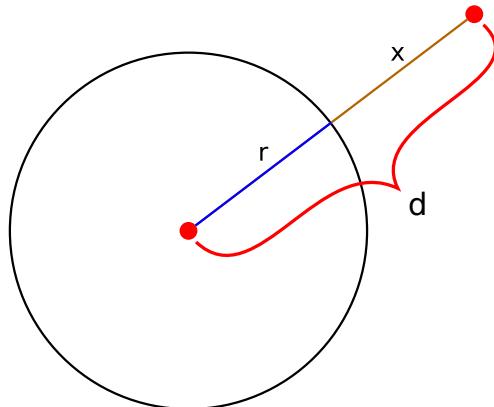
This algorithm consists in a constant number of operations, so it runs in  $O(1)$ .

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**.

In this case, it may prove to be a lot more useful to do a [point vs circle](#) detection, or even a [circle vs circle](#) collision detection, in that case the “radius” would be the “approximation” of a point.

## 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**.



Reference image for Point-Circle Collision detection

We can see that the distance between the center of a circle and our point can be expressed with a formula:

$$d = r + x$$

Where  $r$  is the circle radius and  $x$  is the difference of the distance between the center of the circle and the point (which can be negative):

$$x = d - r$$

The point is inside the circle when  $x \leq 0$ , which means:

$$x \leq 0 \Leftrightarrow d - r \leq 0 \Leftrightarrow d \leq r$$

We can express this in a few words:

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:

```
structure Circle{
    // Let's define a circle class/structure
    Point center;
    int radius;
}

function distance(Point A, Point B) -> float{
    // Calculates the distance between two points
    return square_root((A.x - B.x)^2 + (A.y - B.y)^2);
}

function circle_point_collision(Circle A, Point B)
-> bool{
    if (distance(A.center, B) <= A.radius) {
        return True;
    }else{
```

```

        return False;
    }
}

```

### Code: Point to circle collision detection

Again, the lazier version:

```

structure Circle{
    // Let's define a circle class/structure
    Point center;
    int radius;
}

function distance(Point A, Point B) -> bool{
    // Calculates the distance between two points
    return square_root((A.x - B.x)^2 + (A.y -
B.y)^2);
}

function circle_point_collision(Circle A, Point B)
-> bool{
    return distance(A.center, B) <= A.radius;
}

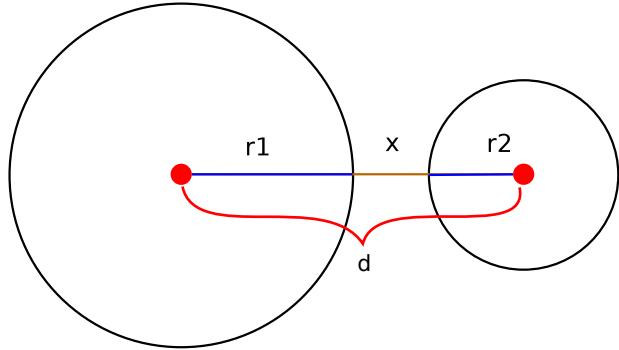
```

### Code: Shorter version of a point to circle collision detection

Although slightly more heavy, computation-wise, this algorithm still runs in O(1).

## Collision Between Two Circles

Let's add another circle into the mix now, and think in more or less the same way as before:



Reference image for Circle-Circle collision detection

We can see the distance between the center of the circles as expressed with the following formula:

$$d = r_1 + x + r_2$$

Where  $r_1$  and  $r_2$  are the radii, and  $x$  is defined as follows:

$$x = d - (r_1 + r_2)$$

As before, our  $x$  can be negative, which means that the circles are colliding if  $x \leq 0$ , which means:

$$x \leq 0 \Leftrightarrow d - (r_1 + r_2) \leq 0 \Leftrightarrow d \leq r_1 + r_2$$

We can express the concept in words again:

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:

```
structure Circle{
    // Let's define a circle class/structure
    Point center;
    int radius;
}
```

```

        function distance(Point A, Point B) -> float{
            // Calculates the distance between two points
            return square_root((A.x - B.x)^2 + (A.y -
B.y)^2);
        }

        function circle_circle_collision(Circle A, Circle
B) -> bool{
            if (distance(A.center, B.center) <= A.radius +
B.radius){
                return True;
            }else{
                return False;
            }
        }
    }
}

```

### Code: Circle to Circle Collision Detection

The shorter version would be:

```

structure Circle{
    // Let's define a circle class/structure
    Point center;
    int radius;
}

function distance(Point A, Point B) -> float{
    // Calculates the distance between two points
    return square_root((A.x - B.x)^2 + (A.y -
B.y)^2);
}

function circle_circle_collision(Circle A, Circle
B) -> bool{
    return distance(A.center, B.center) <= A.radius
+ B.radius;
}

```

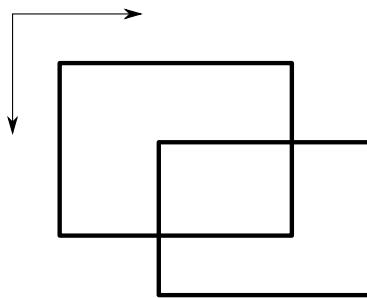
### Code: Shorter Version of a Circle to Circle Collision Detection

Again, this algorithm performs a number of operations that is constant, so it runs in O(1).

## Collision Between Two Axis-Aligned Rectangles (AABB)

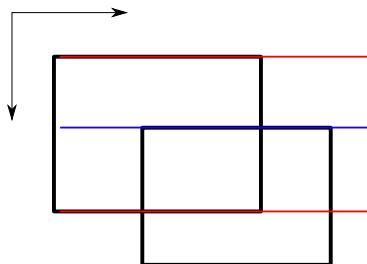
This is one of the most used types of collision detection used in games: it's a bit more involved than other types of collision detection, but it's still computationally easy to perform. This is usually called the “Axis Aligned Bounding Box” collision detection, or AABB.

Let's start with a bit of theory. We have two squares:



Example used in the AABB collision detection

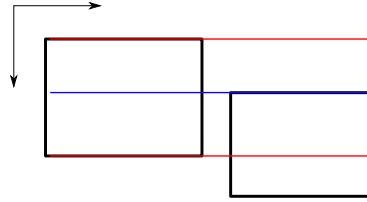
To know if we may have a collision, we need to check if one of the sides is “inside” (that means between the top and bottom sides) of another rectangle:



Top-Bottom Check

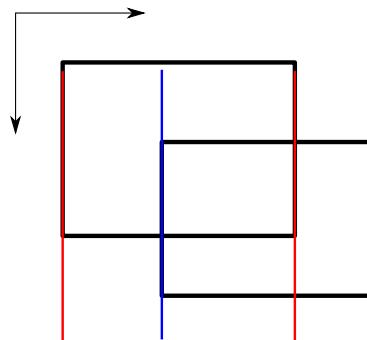
In this case we know that the “top side” of the second rectangle (highlighted in blue) has a  $y$  coordinate between the first rectangle's top and bottom sides'  $y$  coordinates (highlighted in red).

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:



Top-Bottom Check is not enough

So we need to check the other sides also, in a similar fashion:



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:

```
structure Point{
    // Rewritten as a memo
    int x;
    int y;
}

structure Rectangle{
    Point corner;
    int width;
    int height;
}

function rect_rect_collision(Rectangle A, Rectangle
B) -> bool{
    if ((A.corner.x < B.corner.x + B.width) AND
        (A.corner.x + A.width > B.corner.x) AND
        (A.corner.y < B.corner.y + B.height) AND
```

```

        (A.corner.y + A.height > A.corner.y)) {
            return True;
        }else{
            return False;
        }
    }
}

```

### Code: Axis-Aligned Bounding Box Collision Detection

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.

## Line/Point Collision

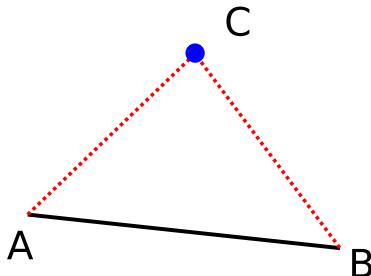
We can represent a segment by using its two extreme points, which proves to be a quite inexpensive way to represent a line (it's just two points). Now how do we know if a point is colliding with a line?

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:



Example of the triangle inequality theorem

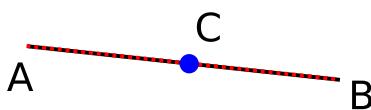
We get the following 3 inequalities:

$$\overline{AB} + \overline{BC} \leq \overline{AC}$$

$$\overline{AC} + \overline{BC} \leq \overline{AB}$$

$$\overline{AB} + \overline{AC} \leq \overline{BC}$$

What is more interesting to us is that when one of the vertices of the triangle is **on** its opposite side, the triangle degenerates:



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:

```
structure Point{
    int x;
    int y;
}

structure Line{
    Point A;
    Point B;
}

function distance(Point A, Point B) -> float{
    // Calculates the distance between two points
    return square_root((A.x - B.x)^2 + (A.y - B.y)^2);
}

function line_point_collision(Point pt, Line ln) ->
bool{
    // First, let's calculate the length of the
line
    float length = distance(ln.A, ln.B);
    // Now let's calculate the distance between the
point pt
    // and the point "A" of the line
    float pt_a = distance(ln.A, pt);
    // Same Goes for the distance between pt and
"B"
    float pt_b = distance(ln.B, pt);
    // Now for the detection
    if (pt_a + pt_b == length){
        return True;
    }else{
        return False;
    }
}
```

### Code: Line to Point Collision detection

It could prove useful to put a “buffer zone” in here too, so that the collision detection doesn’t result too jerky and precise. In that case you may want to take a look at the [line vs circle](#) algorithm, in that case the radius would be the “approximation” of the point.

## 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.

```
structure Point{
    int x;
    int y;
}

structure Line{
    Point A;
    Point B;
}

structure Circle{
    Point center;
    int radius;
}

// ...

function line_circle_collision(Circle circle, Line
line) -> bool{
    bool collides_A = circle_point_collision(circle, line.A);
    bool collides_B = circle_point_collision(circle, line.B);
    if (collides_A OR collides_B){
        return True;
    }
    // ...
}
```

Code: Partial Implementation of a Line to Circle Collision Detection

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:

```
structure Point{
    int x;
    int y;
}

structure Line{
    Point A;
    Point B;
}

structure Circle{
    Point center;
    int radius;
}

function distance(Point A, Point B) -> float{
    // Calculates the distance between two points
}
```

```

        return square_root((A.x - B.x)^2 + (A.y - B.y)^2);
    }

        function line_point_collision(Line line, Point point) -> bool{
            // ...
        }

        function circle_point_collision(Circle circ, Point point) -> bool{
            // ...
        }

        function line_circle_collision(Circle circle, Line line) -> bool{
            // We check the ends first
            bool collides_A = circle_point_collision(circle, line.A);
            bool collides_B = circle_point_collision(circle, line.B);
            if (collides_A OR collides_B){
                return True;
            }
            // We pre-calculate "u", we'll use some variables for readability
            int x1 = line.A.x;
            int x2 = line.B.x;
            int xk = circle.center.x;
            int y1 = line.A.y;
            int y2 = line.B.y;
            int yk = circle.center.y;
            float u = ((xk - x1) * (x2 - x1) + (yk - y1) * (y2 - y1)) / (distance(line.A, line.B))^2;
            // Now let's calculate the x and y coordinates
            float x = x1 + u * (x2 - x1);
            float y = y1 + u * (y2 - y1);
            // "Reuse": we'll use some older functions, let's create a point, with the coordinates we found
            Point P = Point(x,y);
            // Let's check if the "closest point" we found is on the line
            if ((line_point_collision(line, P)) == False){
                // If the point is outside the line, we return false, because the ends have already been checked against collisions
                return False
            }else{

```

```

        // Let's Reuse the Point/Circle Algorithm
        return circle_point_collision(circle, P);
    }
}

```

Code: Line to circle collision detection

## 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.

```

function pointRectCollision(float x1, float y1, float rectx,
float recty, float rectwidth, float rectheight) -> bool{
    // We check if the point is inside the
rectangle
    return x1 >= rectx AND x1 <= rectx + rectwidth
AND y1 >= recty AND y1 <= recty + rectheight;
}

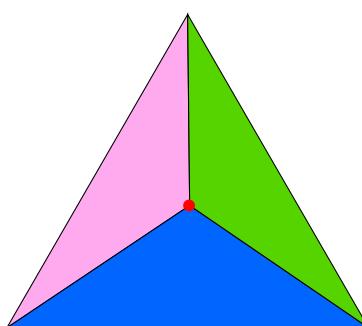
```

Code: Point/Rectangle collision detection

## 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.



## 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:

```
function point_triangle_collision(float px, float py, float x1,  
float y1, float x2, float y2, float x3, float y3) -> bool{  
    float original_area = abs((x2 - x1) * (y3 - y1)  
- (x3 - x1) * (y2 - y1));  
    float area1 = abs((x1-px) * (y2-py) - (x2-px) *  
(y1-py));  
    float area2 = abs((x2-px) * (y3-py) - (x3-px) *  
(y2-py));  
    float area3 = abs((x3-px) * (y1-py) - (x1-px) *  
(y3-py));  
    if (area1 + area2 + area3 == original_area){  
        return True;  
    }else{  
        return False;  
    }  
}
```

### Code: Point/Triangle Collision Detection

This obviously presents the (usual) problem when it comes to precision: computers have no knowledge of infinity (due to their finiteness, see [computers are \(not\) precise](#)). This means that we need (again) to give some leeway and define an “acceptable error” in our calculations, thus we will create a “small enough value” (which in math is represented by the greek letter “epsilon”:  $\epsilon$ ) and change our algorithm accordingly.

Let's see how we can change that.

Our main test is that the sum of the area of the 3 triangles we create ( $A_1, A_2, A_3$ ) is equal to the area of the original triangle ( $A_0$ ), in math terms:

$$A_1 + A_2 + A_3 = A_0$$

We can also rewrite such equation this way:

$$A_1 + A_2 + A_3 - A_0 = 0$$

Due to possible precision issues we know that there are some values where the equation above is not true, so we choose a “low enough error” that we are willing to accept, for example  $\epsilon = 0.0001$ , and use this test instead:

$$|A_1 + A_2 + A_3 - A_0| < \epsilon$$

Which can be expanded (if you want) to

$$-\epsilon < A_1 + A_2 + A_3 - A_0 < \epsilon$$

The code wouldn't change much, but for sake of clarity, here it is:

```
function point_triangle_collision(float px, float py, float x1,
float y1, float x2, float y2, float x3, float y3) -> bool{
    // We accept anything that is closer than
1/1000th of unit
    const float epsilon = 0.0001;
    float original_area = abs((x2 - x1) * (y3 - y1)
- (x3 - x1) * (y2 - y1));
    float area1 = abs((x1-px)*(y2-py) - (x2-px)*
(y1-py));
    float area2 = abs((x2-px)*(y3-py) - (x3-px)*
(y2-py));
    float area3 = abs((x3-px)*(y1-py) - (x1-px)*
(y3-py));
    if (abs(area1 + area2 + area3 - original_area)
< epsilon) {
        return True;
    }else{
        return False;
    }
}
```

Code: Point/Triangle Collision Detection with epsilon

## 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.

```
structure Point{
    // Rewritten as a memo
    int x;
    int y;
}

structure Rectangle{
    // Let's define a rectangle class/structure
    Point corner;
    int width;
    int height;
}

structure Circle{
    // Let's define a circle class/structure
    Point center;
    int radius;
}

function circle_rectangle_collision(Circle circ,
Rectangle rect) -> bool{
    // Detects a collision between a circle and a
rectangle

    // 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
    int tx = circ.center.x;
    int ty = circ.center.y;

    // Let's detect which edge to test against on
the x axis
    if (circ.center.x < rect.corner.x){
        // We're at the left of the rectangle, test
```

```

against the left side
    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
    tx = rect.corner.y + rect.width;
}

// Same thing on the vertical axis
if (circ.center.y < rect.corner.y) {
    // We're above the rectangle, test against
the top side
    ty = rect.corner.y;
} else if (circ.center.y > rect.corner.y +
rect.height) {
    // We're below the rectangle, test against
the bottom side
    ty = rect.corner.y + rect.height;
}

// Let's get the distance between the testing
coordinates and the circle center
int distanceX = circ.center.x - tx;
int distanceY = circ.center.y - ty;
float distance = square_root(distanceX^2 +
distanceY^2);

// Note that if the center of the circle is
inside the rectangle, the testing coordinates will be the
circle's center itself, thus the next conditional will always
return true

if (distance <= circ.radius) {
    return True;
}

// Default to false in case no collision occurs
return False;
}

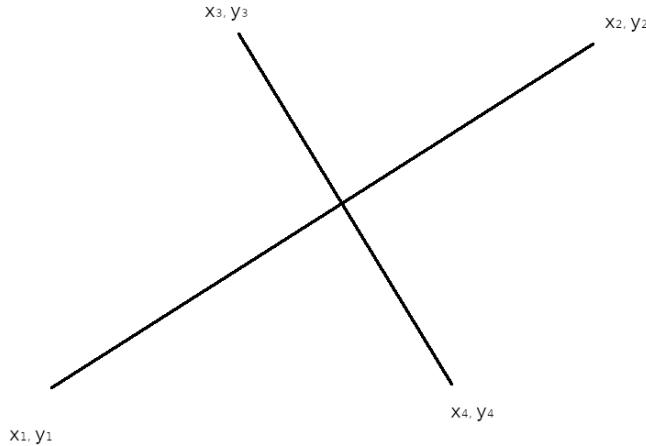
```

Code: Rectangle to Circle Collision Detection

## 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:



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:

```
function lineLineCollision(float x1, float y1, float x2, float y2, float x3, float y3, float x4, float y4) -> bool{
    // Let's calculate the denominator, this will
    // allow us to avoid a
    // "divide by zero" error
    float den = ((y4 - y3) * (x2 - x1) - (x4 - x3)
* (y2 - y1));

    if (den == 0){
        // The lines are parallel
        return false;
    }

    float uA = ((x4 - x3) * (y1 - y3) - (y4 - y3) *
(x1 - x3)) / den;
    float uB = ((x2 - x1) * (y1 - y3) - (y2 - y1) *
(x1 - x3)) / den;

    // 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;
    }

    // If not, they don't collide
    return false;
}
```

Code: Implementation of the line/line collision detection

This collision detection algorithm can be useful for line-based puzzle games, like the untangle puzzle.

## 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.

```

function lineLineCollision(float x1, float y1, float x2, float
y2, float x3, float y3, float x4, float y4) -> bool{
    // our previous implementation of the line/line
    collision detection
    // ...
}

        function pointRectCollision(float x1, float y1,
float rectx, float recty, float rectwidth, float rectheight) ->
bool{
    // our previous implementation of a
    point/rectangle collision detection
    // ...
}

        function lineRectangleCollision(float x1, float y1,
float x2, float y2, float rectx, float recty, float rectwidth,
float rectheight) -> bool{
    // 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
    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
        return True;
    }
    // Now to test the rectangle against the line,
    if it's not completely inside
    bool left = lineLineCollision(x1, y1, x2, y2,
rectx, recty, rectx, recty + rectheight);
    bool right = lineLineCollision(x1, y1, x2, y2,
rectx + rectwidth, recty, rectx + rectwidth, recty +
rectheight);
    bool top = lineLineCollision(x1, y1, x2, y2,
rectx, recty, rectx + rectwidth, recty);
    bool bottom = lineLineCollision(x1, y1, x2, y2,
rectx, recty + rectheight, rectx + rectwidth, recty +
rectheight);

    if (left OR right OR top OR bottom){
        // We hit one of the sides, we are
        colliding
        return True;
    }
}

```

```
        // In any other case, return false
        return False;
    }
```

Code: Implementation of the line/rectangle collision detection

This can prove useful to test for “line of sight” inside an AI algorithm.

## Point/Polygon Collision

Here we are, the most complex matter when it comes to narrow-phase collision detection: detecting collisions between arbitrary convex polygons.

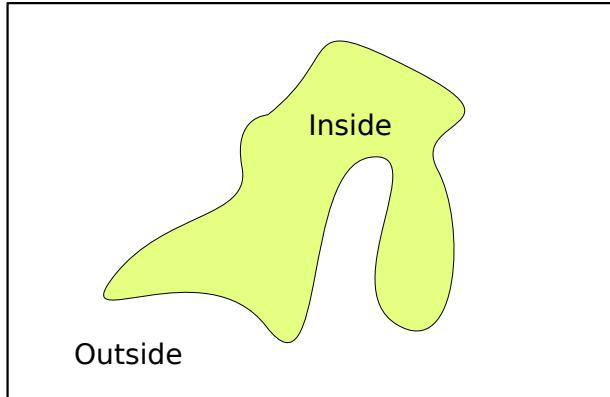
### Note!

In this book we will focus on convex polygons “without holes”, which is the most common situation you’ll find yourself in.

First of all, we will start by talking about some theorems and requirements that will help us on the way to build a “polygon vs polygon” collision detection algorithm.

## Jordan Curve Theorem

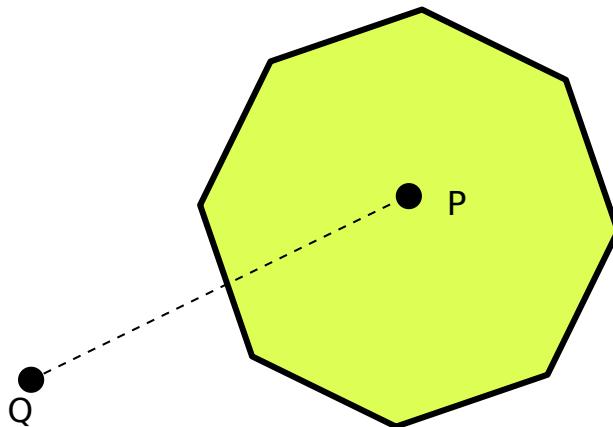
Let’s imagine a plane, like our 2D screen: if we draw a non-self-intersecting, continuous loop in the plane we obtain a *Jordan Curve*. This curve separates the plane in two distinct regions: the “inside” and the “outside”.



Example of a Jordan Curve

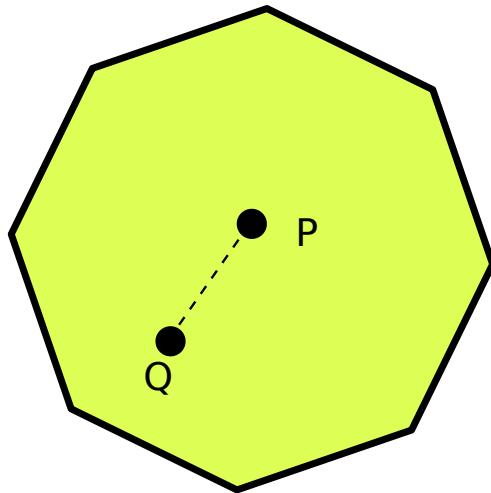
Any non-self-intersecting polygon (be it convex or non-convex) can be seen as a Jordan curve, this means that we can easily identify (programmatically) if a point is inside or outside the polygon. At least in the “convex” case.

Let's take a convex polygon, and a point inside such polygon: we can see that if we choose a point outside the polygon (non-colliding) we can strike a line between the “inside point” and the chosen point, and such line will intersect one of the polygon’s edges. This gives us an idea on how to check for “point vs. polygon”.



A simple case where a point is outside the polygon

This doesn't happen if the point is inside the polygon, obviously:



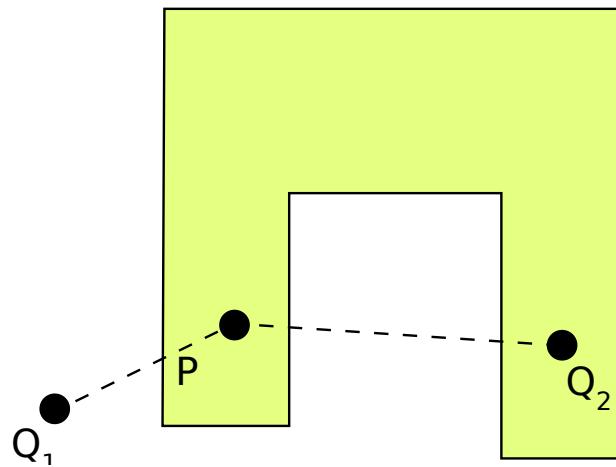
A simple case where a point is inside the polygon

This is all well and good, but we have two problems on hand:

- Finding a point inside the polygon;
- We have a non-convex polygon;

Let's leave the first problem aside, since talking about it may end up being confusing and just empty talk (or writing, being this a book), and let's focus on the second problem.

If we have a non-convex polygon, we may end up with a line that intersects the polygon's perimeter even if the point is colliding:



How a non-convex polygon makes everything harder

Here we call  $P$  the “point inside the polygon” while  $Q_1$  and  $Q_2$  are the points we are testing: as we can see  $Q_2$  triggers our “non-colliding” test even though it is inside the polygon.

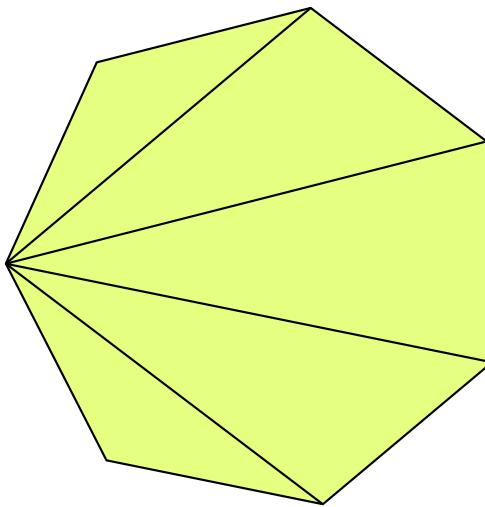
Can you see what can help us solving this issue? I’m sure you have a *number* of ideas in mind, we’ll talk about it in the [non-convex polygon collision detection section](#).

## Thinking outside the box: polygon triangulation

As you can see, as simple as it can be, the Jordan curve theorem poses some problems that may be a bit out of our reach as of now, so let’s try to find a less ideal but easier to understand solution.

Let’s now limit ourselves to convex polygons, which (again) is the most common situation.

We can take inspiration from 3D graphics, where any solid shape (and thus the polygons that make those up) are decomposed to a bunch of triangles. Nothing stops us from doing the same and taking any polygon and decomposing it to a group of triangles, like follows:



Decomposing a polygon into triangles

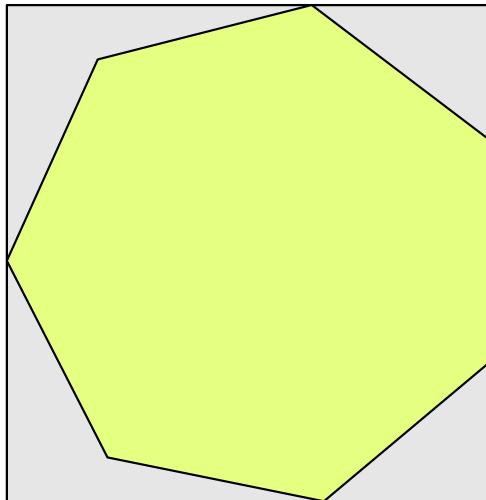
This specific triangulation is called “fan triangulation” and it is chosen for its  $\Theta(n)$  (where  $n$  is the number of vertices) execution time.

## Bounding Boxes

Before making our poor CPU undertake big calculations, we may want to check if there is even a possibility of a collision, maybe with a simpler algorithm.

The great majority of the lifetime of our game objects is spent not colliding with anything, so if we can easily exclude a collision before starting complex algorithms, our game will just benefit from it.

We can take our complex polygon and give it a “bounding box”, any point that is inside such box *has a possibility* of colliding with our polygon, but any point *outside* the bounding box *surely will not collide*.



Example of a polygon with its bounding box

How do we calculate a bounding box? Simple, we just need 4 coordinates:

- The smallest x (which we'll call  $x_{min}$ )
- The smallest y ( $y_{min}$ )
- The biggest x ( $x_{max}$ )
- The biggest y ( $y_{max}$ )

The vertices of our bounding box will always be:

$$A(x_{min}, y_{min}) \ B(x_{max}, y_{min}) \ C(x_{max}, y_{max}) \ D(x_{min}, y_{max})$$

## Tip!

Thanks to how rectangles work, we can just use the points  $A$  and  $C$  to build a rectangle: since they contain all 4 coordinates, we can infer  $B$  and  $D$  from them.

This is simple to achieve: we just need to loop over all the vertices and find our coordinates. The algorithm here below:

```
structure Point{
    // Rewritten as a memo
    int x;
    int y;
}

class Rectangle{
    Point corner;
    Integer width;
    Integer height;
    // ...
    @staticmethod
        function from_points(Point topleft, Point
bottomright) -> Rectangle{
            // ...
        }
    // ...
}

function bounding_box(Point[] vertices) ->
Rectangle{
    // First we create and bootstrap the variables
    int xmin = vertices[0].x;
    int xmax = vertices[0].x;
    int ymin = vertices[0].y;
    int ymax = vertices[0].y;
    // Now we iterate through all the other
vertices
    for (each vertex in vertices) {
        if (vertex.x < xmin){
            xmin = vertex.x;
        }
        if (vertex.x > xmax){
            xmax = vertex.x;
        }
        if (vertex.y < ymin){
            ymin = vertex.y;
        }
        if (vertex.y > ymax){
            ymax = vertex.y;
        }
    }
    return new Rectangle(xmin, ymin, xmax - xmin, ymax - ymin);
}
```

```

        }
        if (vertex.y < ymin) {
            ymin = vertex.y;
        }
        if (vertex.y > ymax) {
            ymax = vertex.y;
        }
    }
    // Now we can build the needed points for the
bounding box
    A = new Point();
    C = new Point();
    A.x = xmin;
    A.y = ymin;
    C.x = xmax;
    C.y = ymax;
    // We build our bounding box
    Rectangle boundingBox =
Rectangle.from_points(A, C);
    // and return it
    return boundingBox;
}

```

Code: How to find the bounding box of a polygon

To check if the collision “may happen”, we can just use a simple [Point vs Rectangle collision check](#).

### Point/Polygon collision detection with triangles

Finally, after all the math and preparations, we can start working towards our collision detection algorithm.

#### Pitfall Warning!

This algorithm works only with convex polygons that have no holes, also it probably is not the most efficient way to check for collisions between a point and a polygon.

This is more akin to an exercise in creativity and less about “notions”: we found a simple solution to a complex problem. Even if it is not the most efficient, it may be “efficient enough”.

### The “Polygon” class

Differently from previous classes and structures, the “polygon” class will need a little more work. This is because we are going to do more than just merely memorize vertices.

First of all we need an **ordered** list (or array) of vertices, which will be represented by points. Secondly, we need facilities to calculate list of triangles, as well as their areas.

#### Pitfall Warning!

You may be tempted to memorize the “triangles” that are an output of the “fan triangulation”, as well as their areas. This may be a good idea if well managed, but we will need to take care of “moving” those triangles and manage when the polygon gets deformed: in that case all the triangle areas will have to be recalculated.

Same goes for the bounding box, which will change in size when the polygon rotates or deforms. In this book we will try to keep the class as generic as possible (as well as simple), thus we will just recalculate everything every frame as needed.

Thirdly, we need the constructor to do some math before we can use the polygon. Finally we need to integrate a “fanning” function.

Whew... That's a lot of work, but here's the code for the polygon class:

```
class Polygon{  
    Point[] vertices;
```

```

        function calculate_bounding_box() -> Rectangle{
            // This function calculates the bounding
box
            // -----
            // First we create and bootstrap the
variables
            int xmin = vertices[0].x;
            int xmax = vertices[0].x;
            /*
             * ...
             * see the bounding box algorithm for the
full version
             * ...
             */
            // We build our bounding box
            Rectangle boundingBox =
Rectangle.from_points(A, C)
            // and return it
            return boundingBox;
        }

        function do_fanning() -> Triangle[]{
            /*
             * This function iterates over the vertices
and returns
             * an array of triangles corresponding to
the "fan triangulation"
             */
            // We fix the "base" of the fan on the
first vertex
            Point root_vertex = vertices[0];
            Triangle[] temp_triangles = new Triangle[];
            // Now we iterate through all the other
vertices
            for (each j from 2 to vertices.length() -
1) {
                // j goes from the third vertex, to the
last
                // j - 1 goes from the second to the
second to last

temp_triangles.append(Triangle.from_points(root_vertex, j - 1,
j));
            }
            // In the end, we will have the triangles
array, we can just return it
            return temp_triangles;
        }
    
```

```
}
```

## Code: A (not so) simple polygon class

### The algorithm

After all this preparation, we are finally ready for the algorithm, which will happen in two passes:

1. A “broad”-ish pass, where we compare the point to the polygon’s bounding box
2. A “proper-narrow” pass, where we do a series of triangle vs point collision detections

Here’s the code:

```
// ...
    function polygon_point(Polygon poly, Point point) -
> bool{
    // First of all, we get the polygon's bounding
    box
    Rectangle bounding_box =
poly.calculate_bounding_box();
    // Then we do a simple point vs. rectangle
    check
    if (not point_rectangle(bounding_box, point)){
        // We are not even in the bounding box, we
        can't collide
        return False;
    }
    // If instead we are in the bounding box, we
    need to get the "fan triangulation"
    Triangle[] triangles = poly.do_fanning();
    // Now we check, for each triangle, if the
    point collides
    for (triangle in triangles){
        if (point_triangle(triangle, point)){
            // We found the "slice" of the polygon
            that the point collides with
            return True;
        }
    }
    // If we pass all triangles without a hit, we
```

```

        are in the bounding box
            // but outside the polygon, that's the worst
        case, and we are not colliding
            return False;
    }
}

```

## Code: Polygon vs Point collision detection

### Performance analysis

The algorithm seems fairly simple, but we may want to check its performance to see how efficient it is. In this analysis  $n$  will be the number of vertices, while  $m$  is the number of triangles.

The best case is that the point we're testing is outside the polygon's bounding box: this means that we calculate the bounding box (which is  $\Theta(n)$ ) and we check the point against it (which is  $\Theta(1)$ ), thus our best case (lower bound) is  $\Omega(n)$ .

The worst case is when the whole algorithm is performed to the end, which means the point is inside the bounding box, but outside the polygon: this means we calculate the bounding box ( $\Theta(n)$ ), check against it ( $\Theta(1)$ ), do the “fan triangulation” ( $\Theta(n)$ ), check each triangle without finding any collision ( $O(m)$ ) and get to the end. Our worst case (upper bound) is  $O(n + m)$ .

Considering the fact that the number of triangles  $m$  is tied to the number of vertices  $n$  by the formula (valid for simple convex polygons)

$$m = n - 2$$

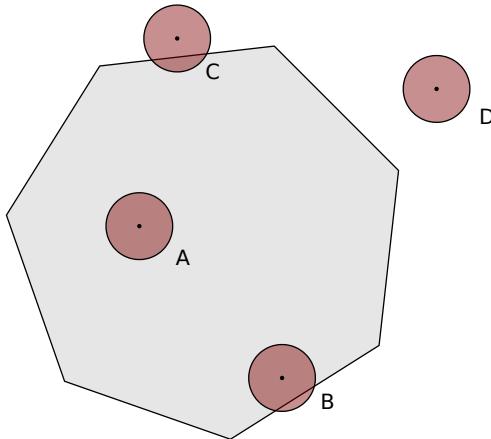
We have an upper bound of  $O(n + m) = O(n + n + 2) = \sim O(n)$ , this is because the constant gets “squashed by the linear behaviour” of  $n$ , and  $2 \cdot n$  behaves asymptotically in the same way as  $n$  when the dataset grows.

Even though we have a tight bound of  $\Theta(n)$  in our entire algorithm (which means the amount of calculations goes up slowly with the addition of new vertices), we need to be mindful of the amount of calculation that is done, including some heavy operations like square roots.

## Circle/Polygon Collision

Now that we got one of the hardest topics out of the way, we can focus on other types of collision detection between arbitrary convex polygons: one of those is the “circle vs polygon” collision detection.

Let’s see an example image first:



Example image used for circle/polygon collision detection

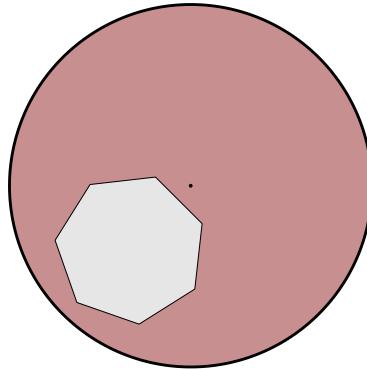
Here we can see four different cases of collision (or lack thereof) between a circle and a polygon (if you’re particularly acute, you may have noticed we’re missing a 5th case, but we’ll talk about it shortly):

- **Case A:** The circle is completely inside the polygon;
- **Case B:** The circle is partially inside the polygon, with the center being **inside** the polygon;
- **Case C:** The circle is partially inside the polygon, with the center being **outside** the polygon;
- **Case D:** The circle is completely outside the polygon.

Case A and B can be solved together with a point/polygon check, where the point is the center of the circle, while case C can be solved by a line/circle

check between the circle and all the edges of the polygon.

What about the “missing 5th case”? Here it is:



An edge case of the circle/polygon check

In this case the circle contains the polygon completely, with its center outside of the polygon area, so the check used in cases A and B wouldn't work and neither would the one used in case C.

This is a really rare edge-case, since usually the game does its checks so fast that you'd end up in case C long before this edge-case sees the light of day. In the event this happens, we just need to check if any of the vertices of the polygon is inside the circle.

Here's the full algorithm:

```
// ...
    function circle_polygon(Polygon poly, Circle circ)
-> bool{
    // Case C (and partly B) are less resource-
intensive than
    // a point/polygon check, so let's do them
first
    for (i from 0 to poly.vertices.length() - 1){
        // We iterate through all the vertices
        int j = i + 1;
        // If we get to the end, we wrap around j
        if (j==poly.vertices.length()){
            j = 0;
        }
        Line temp_line =
Line.fromPoints(poly.vertices[i], poly.vertices[j]);
```

```

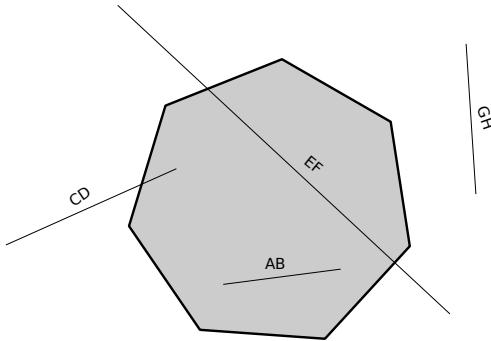
        // In case we find a hit, we already know
there is a collision
        if (line_circle_collision(circ, temp_line))
{
            return True;
}
}
// Now Let's check for cases "A" and "B"
if (polygon_point(poly, circ.center)){
    // If the center is inside the polygon, we
have a collision
    return True;
}
// Now let's check for the rare edge-case: if
this case happens, all the vertices
    // are inside the circle, so we can only check
one of them
    if (circle_point_collision(circ,
poly.vertices[0])){
        // If any vertex is inside the circle, we
have a collision, so we check the first
        return True;
}
// If none of the checks above returned, we
don't have a collision (case D)
return False;
}

```

Code: Polygon vs Circle collision detection

## Line/Polygon Collision

The line vs polygon collision detection algorithm is not really different from the ones we have seen previously. Let's take a look at an image with all the cases we can think about:



Example image used for line/polygon collision detection

Here we can see 4 cases (this time for real):

- **Line  $\overline{AB}$ :** The segment is completely inside the polygon (including its ends);
- **Line  $\overline{CD}$ :** The segment is partially inside the polygon (one of its ends is inside the polygon);
- **Line  $\overline{EF}$ :** The segment crosses the polygon, but both its ends are outside the polygon;
- **Line  $\overline{GH}$ :** The segment is completely outside the polygon;

We can solve the cases involving the lines  $\overline{AB}$  and  $\overline{CD}$  by checking if either of the ends is inside the polygon, using a point/polygon collision check.

The case involving the line  $\overline{EF}$  can be solved by a line/line collision check between the  $\overline{EF}$  and all the edges of the polygon.

Let's take a look at the full algorithm:

```
// ...
function line_polygon(Line line, Polygon poly) ->
bool{
    // First of all, let's check if either of the
    line ends are inside the polygon
    // This covers cases AB and CD
    if (polygon_point(poly, line.A)){
        // One of the ends is inside the polygon,
        we have a hit
        return True;
    }
    if (polygon_point(poly, line.B)){
        // One of the ends is inside the polygon,
        we have a hit
    }
}
```

```

        return True;
    }
    // Now we check for case EF
    for (i from 0 to poly.vertices.length() - 1){
        // We iterate through all the vertices
        j = i + 1;
        // If we get to the end, we wrap around j
        if (j == poly.vertices.length()){
            j = 0;
        }
        Line temp_line =
Line.fromPoints(poly.vertices[i], poly.vertices[j]);
        if (line_line_collision(temp_line, line)){
            return True;
        }
    }
    if (line_line_collision(temp_line, line)){
        return True;
    }
    // If none of the previous checks was
triggered, we don't have a collision
    return False;
}

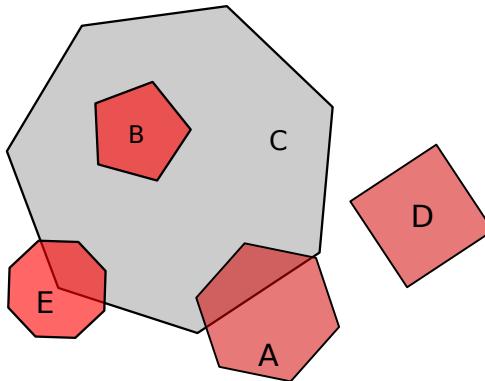
```

Code: Polygon vs Line collision detection

## Polygon/Polygon Collision

Here we are, the final frontier, polygon vs polygon collision detection. We went through a lot of pages of notions and reasoning to get here, now we have the tools to undertake one of the more complex collision detection methods.

Remember: we are checking if two **convex** polygons are colliding, let's see an example image first.



Example image used for polygon/polygon collision detection

We can see 4 cases here, from the simplest to the hardest:

- The Square **D** is outside the polygon;
- The Pentagon **B** is completely inside the polygon
- The Octagon **E** is colliding with the heptagon **C** and a vertex of **C** is inside of **E**;
- The heptagon **C** is colliding with the hexagon **A**, but none of its vertices of **C** are inside of **A**;

We can easily solve the cases involving **A** and **E** with a “polygon vs line” collision detection, while the case involving **B** can be checked by doing a “polygon vs point” check.

Let's take a look at the algorithm:

```
// ...
function polygon_polygon(Polygon p1, Polygon p2) ->
bool{
    // First we do a polygon vs line check for all
the edges
    for (i from 0 to p2.vertices.length() - 1){
        int j = i + 1;
        if (j == p2.vertices.length()){
            // Wrap around in case we get to the
end
            j = 0;
        }
        Line temp_line =
Line.fromPoints(p2.vertices[i], p2.vertices[j])
        if (polygon_line(p1, temp_line)){
            // We have a hit
        }
    }
}
```

```

        return True;
    }
}

// Now we check in case one polygon contains
the other, we can just check a single vertex
if (polygon_point(p1, p2.vertices[0]) or
polygon_point(p2, p1.vertices[0])){
    return True;
}
// None of the checks was triggered, there is
no collision
return False;
}

```

### Code: Polygon vs Polygon collision detection

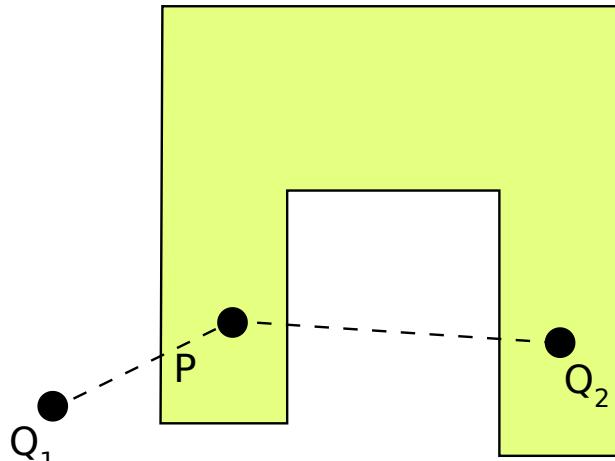
As you can see, the algorithm is quite short, but it builds on a lot of previous algorithms that we already studied, so there is a lot of “hidden complexity” behind these few rows of code.

#### Tip!

We can make the algorithm perform a bit better by adding a check between the (axis aligned) bounding boxes first: this will drastically reduce the amount of “line vs polygon” and “point vs polygon” checks, at the expense of a slightly heavier algorithm when a collision happens.

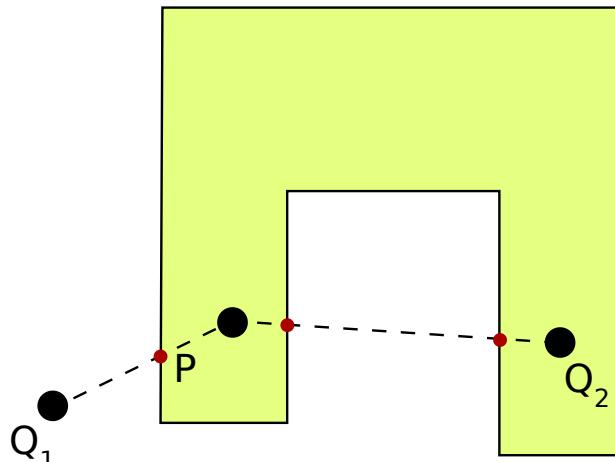
## Non-convex polygons collision

Let’s go back to our previous example, using a non-convex polygon: we have an “inside point” and two points to test, one inside and one outside.



How a non-convex polygon still makes everything harder

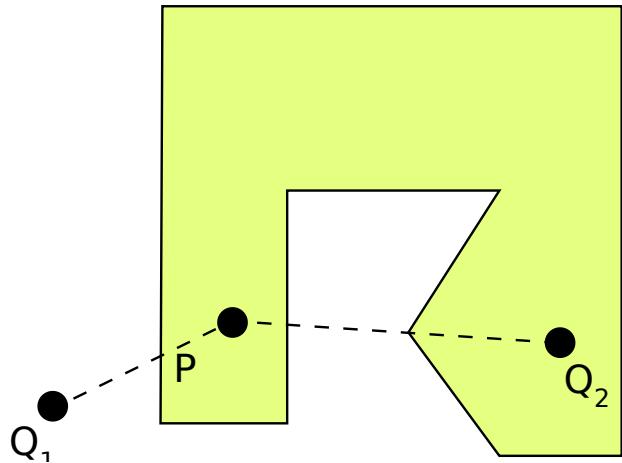
The trick is counting the number of times our “segment between the points” hits the perimeter of the polygon:



Counting how many times we hit the perimeter gives us the result

If the number of “hits” is odd, we know the point tested is outside, if the number of “hits” is even, the point is inside the polygon.

The previous statement fails when we hit a vertex in our way: we can't really count it as a “double hit”, because there's the possibility that we are hitting it while “entering” the polygon.



Issues with vertices make everything even harder

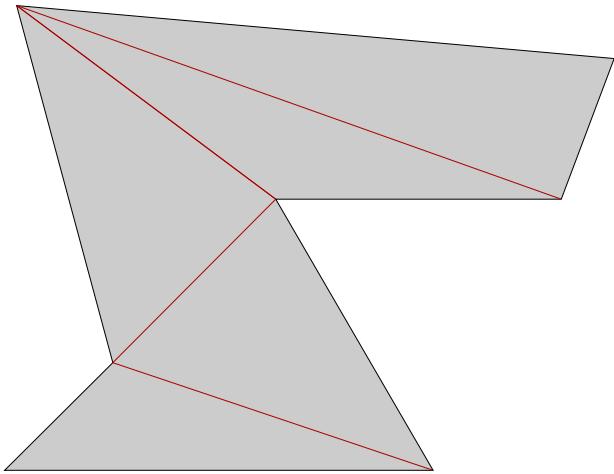
If we counted the vertex hit as a “double hit”, we would end up having a point “inside the polygon” figuring as a “point outside the polygon”.

The complications and edge cases are many and beyond the scope of this book, so we’ll stop here and instead continue with the ways we discussed earlier.

### Polygon triangulation: the return

We can extend the reasoning we made with simple convex polygons earlier to all simple polygons (so we can include non-convex ones too): any non-self-intersecting polygon without holes can be decomposed into triangles.

The only limitation we have is the method: the “fan triangulation” method works only with convex polygons and a very limited set of non-convex ones; so we need to find a different way of triangulating those polygons.



Triangulating a non-convex polygon

Triangulation methods include “ear clipping” and “monotone polygon triangulation”, but their implementation is beyond the scope of this book.

**Tip!**

You can always take any type of polygon (even with holes) and decompose it into a bunch of convex polygons that can be fan-triangulated. Many graphical libraries represent polygons based on the fan-triangulation method.

## Pixel-Perfect collision

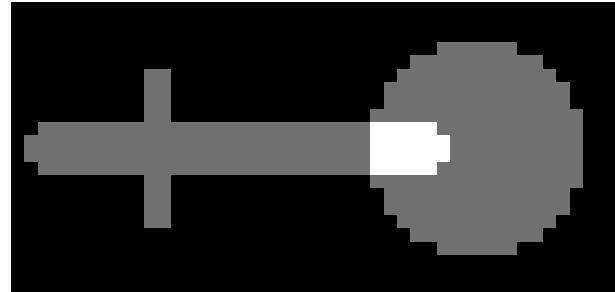
Pixel perfect collision is the most precise type of collision detection, but it's also by far the slowest.

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”).



## Two Bitmasks that will be used to explain pixel-perfect collision

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 operation returns a “True” a collision occurred.



Two Bitmasks colliding, the ‘AND’ operations returning true are highlighted in white

```
class Color{
    int colorData;
    bool function isWhite();
}

structure Bitmask{
    Color[] data;
    Color getColor(x, y);
}

structure Sprite{
    Bitmask bitmask;
    int x;
    int y;
    int width;
    int height;
}

function pixel_perfect_collision(Sprite A, Sprite
B) -> bool{
    // Calculate the intersecting rectangle to
limit checks
    int x1 = max(A.x, B.x);
    int x2 = min((A.x + A.width), (B.x + B.width));

    int y1 = max(A.y, B.y);
    int y2 = min((A.y + A.height), (B.y +
B.height));
}
```

```

        // For each pixels in the intersecting
rectangle, let's check
        for (each y from y1 to y2){
            for (each x from x1 to x2){
                a = A.bitmask.getColor(x - A.x, y -
A.y);
                b = B.bitmask.getColor(x - B.x, y -
B.y);

                if (a.isWhite() AND b.isWhite()){
                    return True;
                }
            }
        }

        // If no collision is occurred by the end of
the checking, we're safe
        return False;
    }
}

```

Code: Example of a possible implementation of pixel perfect collision detection

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.

## Broad-phase collision detection: is a collision even possible?

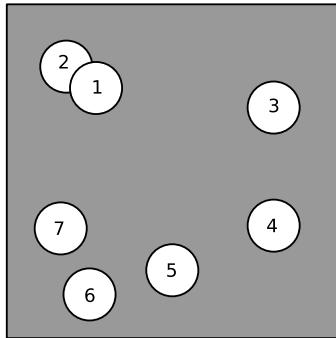
Now we need to find which game objects collided, and this can be easily one of the most expensive parts of our game, if not handled correctly.

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 it having “someone else” take care of checking for collisions.

As an example, we’ll take the following situation:



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.

## 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:

```
function is_collision(Item A, Item B) -> bool{
    // Defines how two items collide (being
    // circles, this could be a difference of radii)
    // ...
}

Item[] items = [item1, item2, ...];

-> Item[] {
    function get_colliding_items(Item[] items_to_check)
        Item[] colliding_items = [];
```

```

        for (A in items_to_check) {
            for (B in items_to_check) {
                if (A is not B) {
                    // We avoid checking if an item
collides with itself, for obvious reasons
                    if (is_collision(A, B)) {
                        // We keep together the pair of
items that collided
                        colliding_items.add(new pair(A,
B));
                    }
                }
            }
        }
        return colliding_items;
    }
}

```

### Code: Brute Force Method of collision search

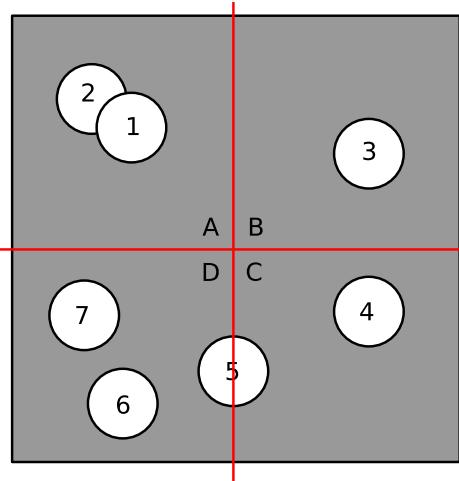
This algorithms runs in  $O(n^2)$ , because it checks every item with every other, even with itself.

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).

## Building Quad Trees

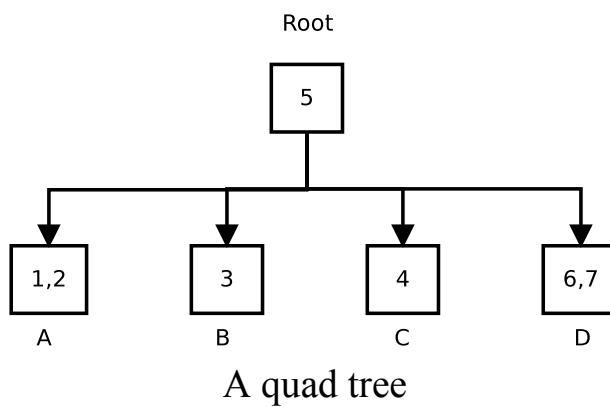
A nice idea would be being able to limit the number of tests we perform, since the brute force method can get really expensive really quickly.

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.



Graphical example of a quad tree, overlaid on the reference image

And here below we can see how a quad tree would look, in its structure:



The rules to follow in a quad tree are simple, both in filling and retrieval.  
When we are filling a quad tree:

- Each node starts by being inserted in the root;
- If the root is “full” (exceeds a set quantity of nodes), it “splits” into 4 sub-trees;
- If a node would fit in two quadrants (like #5), it gets put inside the parent of both quadrants.

When we are retrieving the nodes we will know that an object inside a certain node can collide only with the objects in the same nodes or in the subtree rooted at such node.

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:

- 6 Tests against node 5 (5-1, 5-2, 5-3, 5-4, 5-6, 5-7);
- 1 Test against node 1 (1-2);
- 1 Test against node 2 (2-1);
- No tests against node 3, because it's on its own and there are no subtrees;
- No tests against node 4, for the same reason;
- 1 Test against node 6 (6-7);
- 1 Test against node 7 (7-6).

For a total of 10 tests, which can be further optimized by avoiding testing pairs of objects that have already been tested.

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

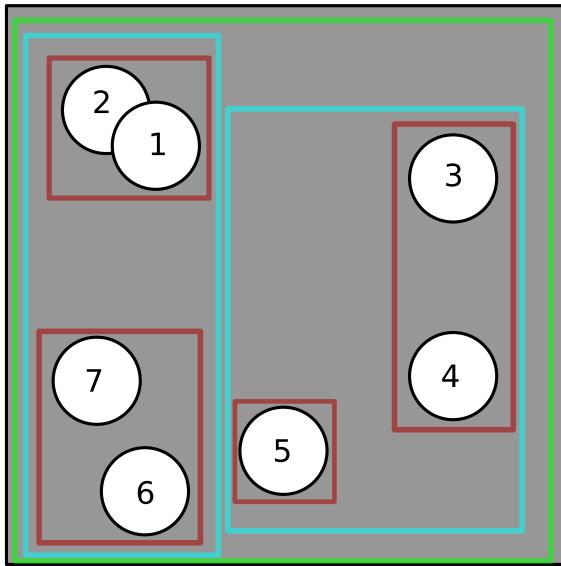
## **Building AABB-Trees**

Another way to efficiently execute a broad-phase collision detection is by building trees containing Axis-Aligned Bounding Boxes.

The main idea is similar to what we've seen with binary search trees, mixed with the quad-trees we've just talked about: we are trying to keep track of objects that are close together (like Quad-Trees do) and when searching, we try to eliminate a good portion of data each time we descend the tree (similarly to binary search trees).

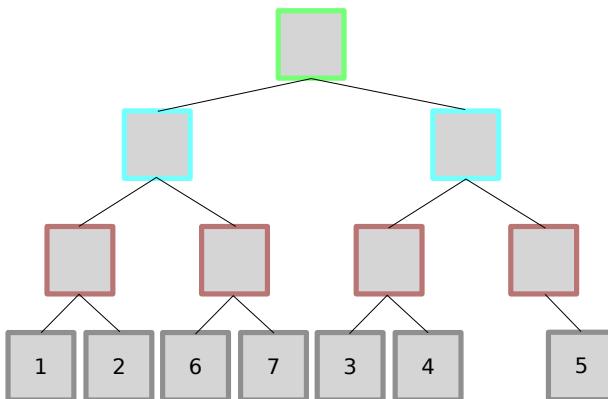
This is done by calculating a “cost function” every time we insert an object into the tree: our objective is making the cost as little as possible. An idea for the cost function could be the size of the rectangle (expressed by its perimeter, or just *width + height*).

Our example image, would be represented this way:



How an AABB-tree would process our example image

This can look a bit confusing, let's see how the tree would look like:



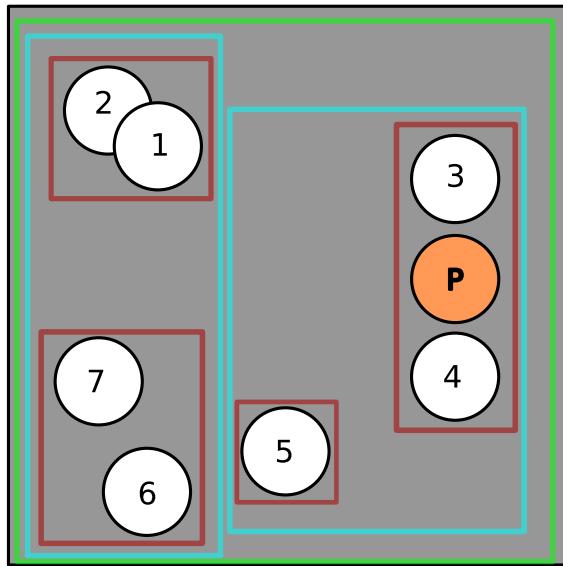
How a possible AABB-tree structure would look like

The performance of this tree is tightly related to its “balancing”: differently from other types of “balanced trees”, AABB-trees rely on how evenly each parent node is split by its children (instead of the usual “depth” metric). If an AABB-tree doesn’t split evenly, the algorithm won’t be able to “exclude” as many nodes on each iteration, thus degrading to a brute-force method (trying the given AABB against all other bounding boxes).

## Querying

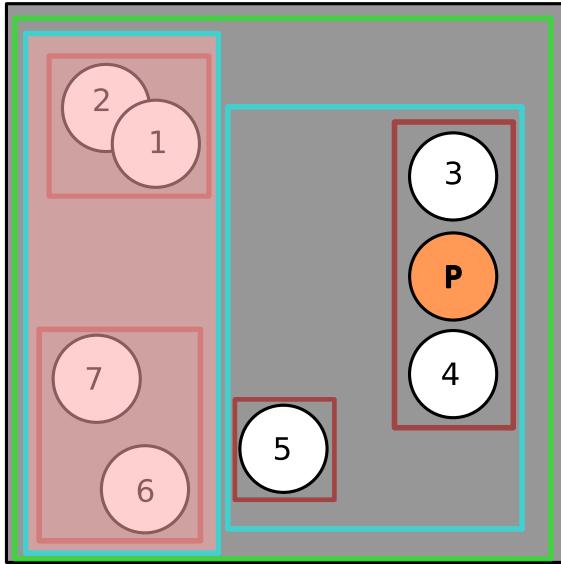
The idea behind this type of tree is making queries as fast as we can, and that can be done by checking on smaller rectangles on every iteration of our search algorithm. For instance we can find a list of possible colliding entities with a given bounding box in only a few tests (in our example).

Let's take for instance a circle "P" that is exactly between the points 3 and 4:



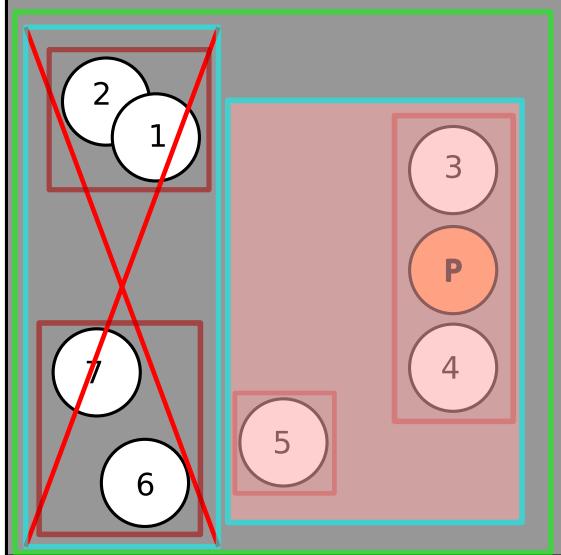
Example of a search in an AABB-Tree

First we do the root test, to see if it may collide with any of the 7 circles we have (if it was outside of the green rectangle, we would have finished already). Then we do the "left (cyan) child" test, in this case we're not colliding with the relative bounding box, so we keep going.:



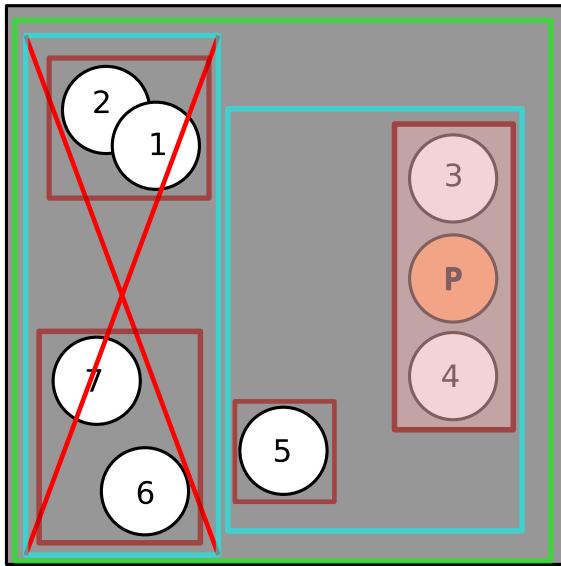
Querying an AABB-tree (1/3)

This way we excluded 1,2,6, and 7. We now do the “right (cyan) child” test, we’re colliding with the relative bounding box, we continue on this branch.



Querying an AABB-tree (2/3)

We do the “left (red) child” test, we’re colliding with the relative bounding box, now we can do a narrow-phase collision detection with the leaves of this node (and in the meantime we also excluded 5).



Querying an AABB-tree (3/3)

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

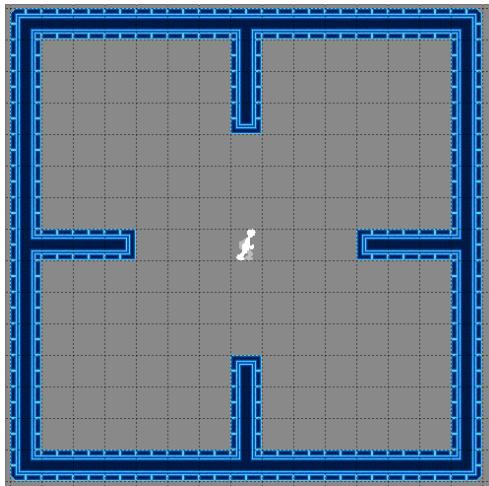
## Other Collision Detection Methods

### Calculating the position of tiles

When you are using tiles to build a level, being able to use quad trees or brute force methods to limit the number of collision checks inside your game may be harder than other methods.

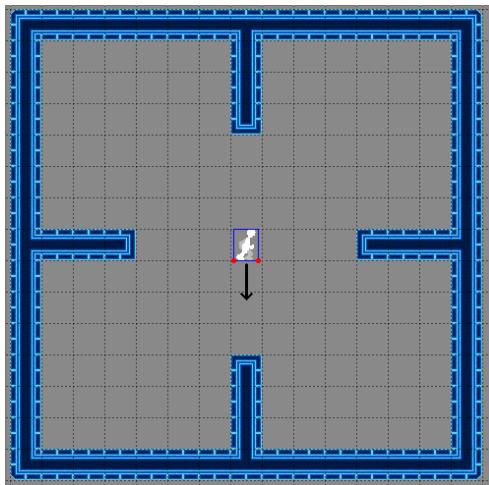
Using a bit of math is probably the easiest and most efficient method to find out which collisions happened.

Let's take an example level:



Example tile-based level

If a game entity is falling, like in the following example:



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:

```

constant TILE_WIDTH = 32;
    constant TILE_HEIGHT = 32;

        function convert_pixels_to_tile(int x, int y) ->
int[] {
    // Converts a point into tile coordinates
    int tile_x = floor(x / TILE_WIDTH);
    int tile_y = floor(y / TILE_HEIGHT);
    return [tile_x, tile_y];
}

```

### Code: Converting player coordinates into tile coordinates

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.

```

constant TILE_WIDTH = 32;
    constant TILE_HEIGHT = 32;

    struct Rectangle{
        // A rectangle will represent the player
        Point corner;
        int width;
        int height;
    }

        function convert_pixels_to_tile(int x, int y) ->
int[] {
    // Converts a point into tile coordinates
    int tile_x = floor(x / TILE_WIDTH);
    int tile_y = floor(y / TILE_HEIGHT);
    return [tile_x, tile_y];
}

    // We assume the player is falling, so no check
will be shown here
    Point[] points_to_check = [
        Point(player.corner.x, player.corner.y +
player.height),
        Point(player.corner.x + player.width,
player.corner.y + player.height),
    ]
    for (each point in points_to_check){
        int[] detected_tile_coordinates =

```

```

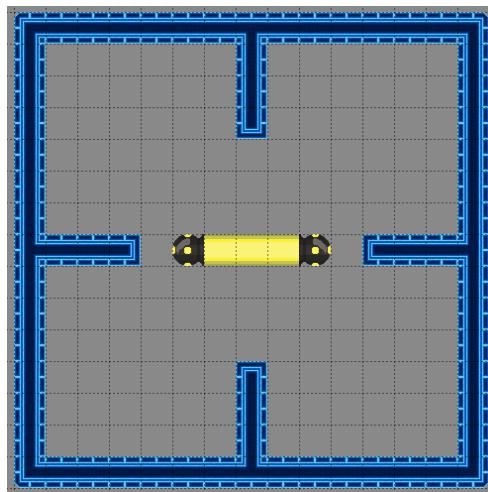
convert_pixels_to_tile(point.x, point.y)           Tile    detected_tile = 
get_tile(detected_tile_coordinates[0],             // 
detected_tile_coordinates[1])                     // 
if (AABB(player, detected_tile.rectangle)){        // 
    // React to the collision
    // ...
}
}

```

### Code: Tile-based collision detection

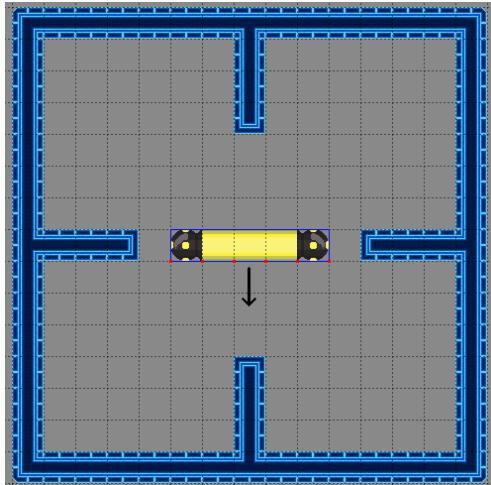
Considering that this algorithm calculates its own colliding tiles, we can state that its complexity is  $O(n)$  with  $n$  equal to the number of possibly colliding tiles calculated.

If an object is bigger than a single tile, like the following example:



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



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.

## 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.

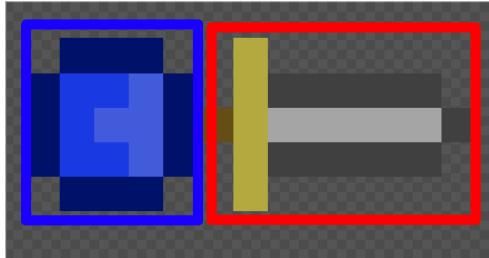
### HitBoxes vs HurtBoxes

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.



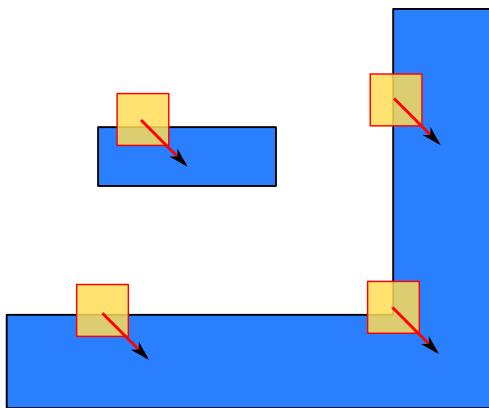
Example of a hitbox (red) and a hurtbox (blue)

## 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:



Images used as a reference for collision reaction

We will study each case separately, at the time the collision is detected (so the two objects are already interpenetrating), and each case will be a piece of this reference image.

## A naive approach

This is the simplest method we can think of: as soon as the object gets inside of a wall, you push it back to one of the edges of the block, while keeping an eye on the direction it's moving.

### 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.

```
// Naive collision reaction with rectangles
    function update(float dt) {
        // ...
        player.position = player.position +
player.speed * dt;
        // Refer to your favourite collision detection
and broad/fine passes
        if (collision(player, object)){
            if (player.x_speed > 0){ // going right
                player.position.x =
object.rectangle.left; // reset position
                player.x_speed = 0; // stop the player
            }
            if (player.x_speed < 0){ // going left
                player.position.x =
object.rectangle.right; // reset position
                player.x_speed = 0; // stop the player
            }
        }
        // Again, refer to your favourite collision
detection and broad/fine passes
        if (collision(player, object)){
            if (player.y_speed > 0){ // going down
                player.position.y =
object.rectangle.top; // reset position
                player.y_speed = 0; // stop the player
            }
            if (player.y_speed > 0){ // going up
                player.position.y =
object.rectangle.bottom; // reset position
                player.y_speed = 0; // stop the player
            }
        }
    }
```

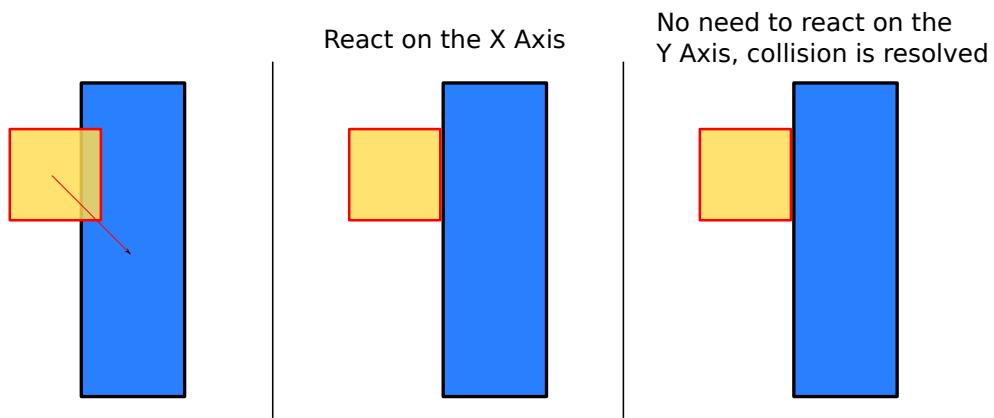
```
    }  
    // ...  
}
```

Code: Code for the naive collision reaction

### Analysis

Let's see how this method reacts in each situation.

When we are trying to slam against the wall, this method works as follows:



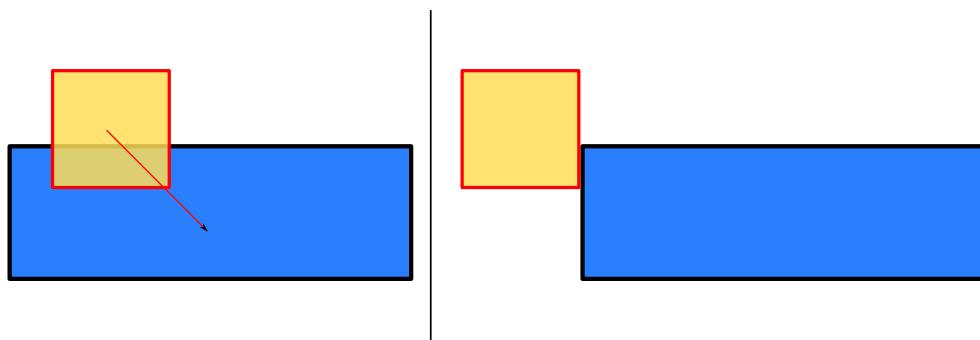
How the naive method reacts to collisions against a wall

1. We separate our position vector in its  $x$  and  $y$  components.
2. We check for collisions, and if so, we react on the  $x$  axis in a direction opposite to the  $x$  component of the velocity.
3. We check for collisions again, if there are any, we react on the  $y$  axis, in a direction opposite to the  $y$  component of the velocity.

### Problems

Problems arise when we try to use the same method to react to a collision on a horizontal plane. In that case reacting on the x axis first will bring some unexpected surprises.

Reacting on the X axis  
doesn't look right



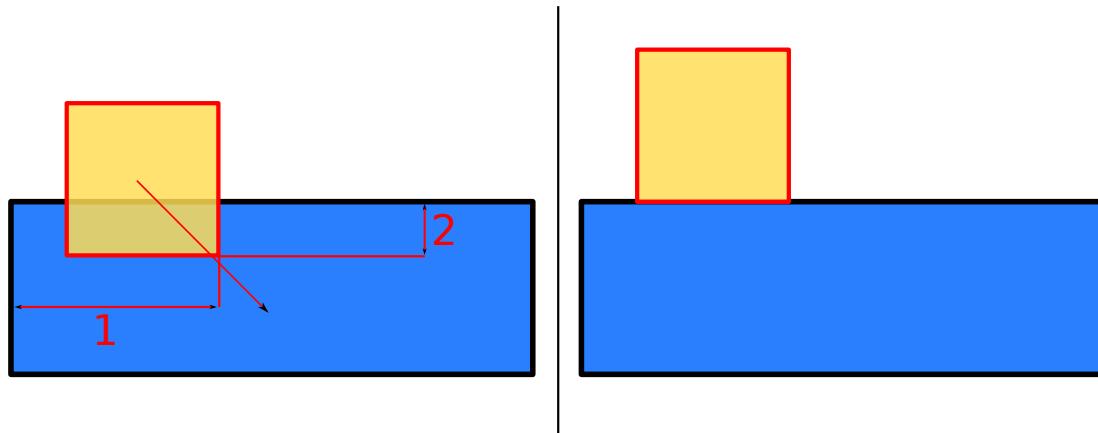
How the naive method reacts to collisions against the ground

We need to find a way to decide which axis we should correct first.

### Shallow-axis based reaction method

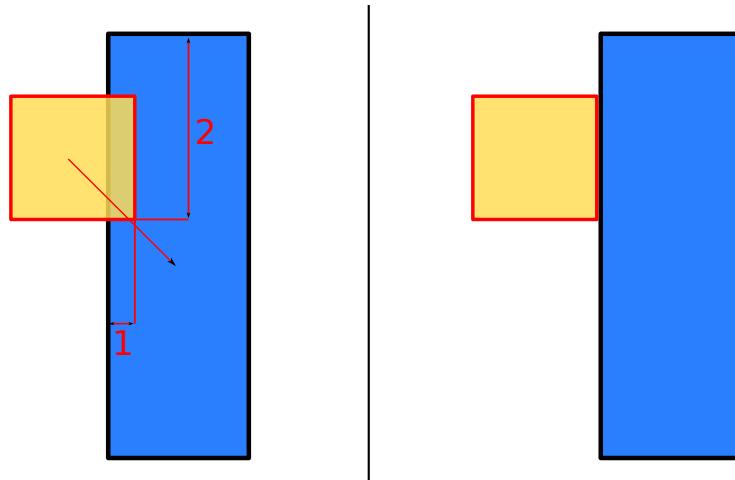
This method works in a similar fashion to the naive method, but prioritizes reactions on the axis that shows the shallowest overlap.

This requires measuring how much the objects overlap on each axis, which can be a little more involved, but not really expensive.



Example of shallow-axis based reaction on a horizontal plane

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.



Example of shallow-axis based reaction on a vertical plane

In this new situation, the algorithm chooses to solve the collision on the  $x$  axis first; but since solving the  $x$  axis solves the collision, no reaction is performed on the  $y$  axis.

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

## Interleaving single-axis movement and collision detection

This is a method quite simple to understand: you split the movement in its  $x$  and  $y$  components, move on the first component, check and react, move on the other component, check and react again.

### How it works

This works by treating the  $x$  and  $y$  axes separately, updating one, checking the collisions that come up from it, update the other axis and check for new collisions.

```
// Interleaving movement and collision reaction with rectangles
function update(float dt) {
    // ...
    player.position.x = player.position.x +
player.x_speed * dt;
    // Refer to your favourite collision detection
and broad/fine passes
```

```

        if (collision(player, object)){
            if (player.x_speed > 0){ // going right
                player.position.x =
object.rectangle.left; // reset position
                player.x_speed = 0; // stop the player
            }
            if (player.x_speed < 0){ // going left
                player.position.x =
object.rectangle.right; // reset position
                player.x_speed = 0; // stop the player
            }
        }
        player.position.y = player.position.y +
player.y_speed * dt;
        // Again, refer to your favourite collision
        detection and broad/fine passes
        if (collision(player, object)){
            if (player.y_speed > 0){ // going down
                player.position.y =
object.rectangle.top; // reset position
                player.y_speed = 0; // stop the player
            }
            if (player.y_speed < 0){ // going up
                player.position.y =
object.rectangle.bottom; // reset position
                player.y_speed = 0; // stop the player
            }
        }
        // ...
    }
}

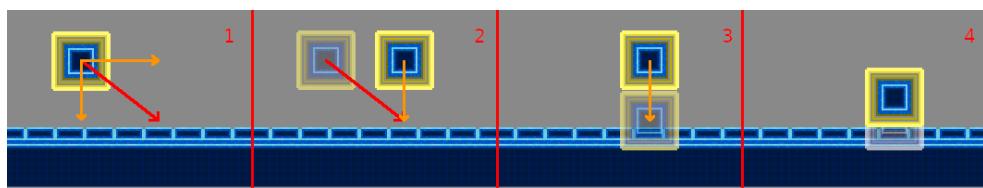
```

Code: Code for interleaving movement and collision reaction

### 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:



How the interleaving 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.

## 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.

```
// Snapshot collision reaction
    // All the sprite origins are on the top-left
corner of the entity
    Player snapshot = player_instance.copy(); // The
"snapshot"

    // Update the player_instance here
    player_instance.position = player_instance.position
+ (velocity * dt);

    // Now check for collisions
    ...
    for (each block colliding with player_instance){
        if ((snapshot.y >= block.y + block.height) AND
(player_instance.y < block.y + block.height)){
            // We are coming on the block from below,
react accordingly
            // Ignoring this reaction will allow
players to phase through blocks when coming from below
            player_instance.position.y = block.y +
block.height;
        }

        if ((snapshot.y + snapshot.height <= block.y)
AND (player_instance.y + snapshot.height > block.y)){
```

```

        // We are coming on the block from above
        player_instance.position.y = block.y;
        player_instance.on_ground = true;
    }

    if ((snapshot.y + snapshot.width <= block.x)
AND (player_instance.x > block.x)){
        // We are coming on the block from left
        player_instance.position.x = block.x -
player_instance.width;
    }

    if ((snapshot.y >= block.x + block.width) AND
(player_instance.x < block.x + block.width)){
        // We are coming on the block from right
        player_instance.position.x = block.x +
block.width;
    }
}

```

Code: Example of the "snapshot" collision reaction method

This method solves the problem given by platforms that can be crossed one-way.

## The “Tile + Offset” Method

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

## When two moving items collide

So far we've seen methods that involve a moving object colliding with a stationary one, but what if we wanted to react to a collision between two moving objects?

Some more math will be needed but it's not extremely difficult to pull off.

First of all, we need to find the “collision vector” (we'll call that  $\mathbf{u}_{coll}$ ), which is simply a vector that is calculated using the difference of the

objects' positions. We'll need just the direction, so we will normalize it too (so it will become  $\hat{u}_{coll}$ ).

Let's imagine two objects, with the following positions:  $A(x_1, y_1)$  and  $B(x_2, y_2)$

$$u_{coll} = (x_2 - x_1, y_2 - y_1)$$

$$\hat{u}_{coll} = \frac{u}{\|u\|}$$

Now we need to know how the objects are moving in relation to each other, this will allow us to see if and how we need to react. Let's calculate the "relative velocity" of the objects.

$$v_{rel} = (v_{x2} - v_{x1}, v_{y2} - v_{y1})$$

Now we need to see how the relative velocity affects the collision, which means we need to project such velocity onto the collision vector. Sounds like a job for the dot product.

$$s = \hat{u}_{coll} \cdot v_{rel}$$

$s$  can be called "the speed of collision" (it's a scalar number, not a vector) and tells us what we need to know: if  $s < 0$  then the objects are moving away from each other already and we don't need to do anything. If  $s > 0$  then the objects are moving towards each other

To react to objects that are moving towards each other, we just need to change their velocity by a factor of  $s \cdot \hat{u}_{coll}$ .

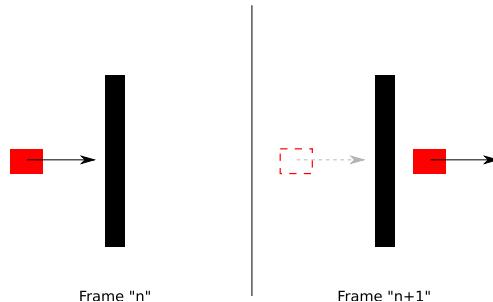
*[This section is a work in progress and it will be completed as soon as possible]*

# Common Issues with time-stepping Collision Detection

The methods we saw so far when checking for collisions are called “time-stepping techniques” due to the fact that each loop we “take a snapshot” of the situation and analyze it, this opens the door to a series of issues that may be annoying and we may find in our game development endeavors.

## 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.



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.

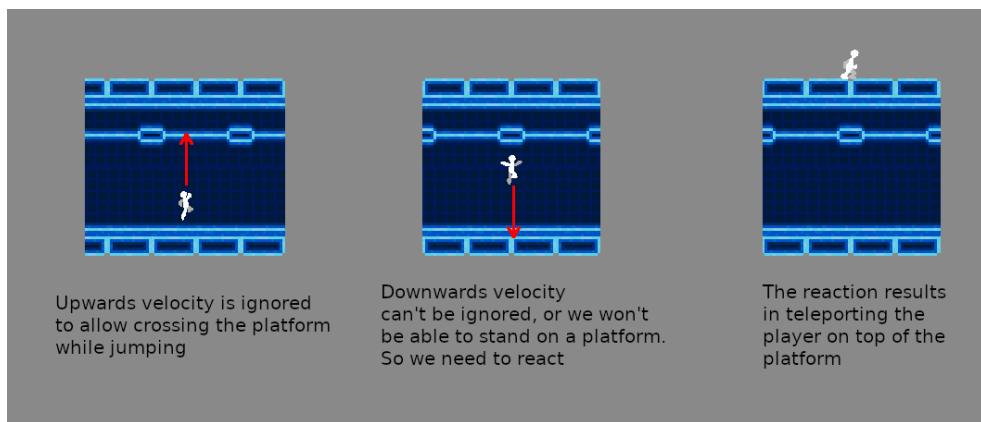
## Precision Issues

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.

## One-way obstacles

Some of the methods exposed can be used only with completely solid obstacles. If you want to make use of platforms that you can cross one-way you should pay attention, since you may get teleported around when your velocity changes direction.



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.

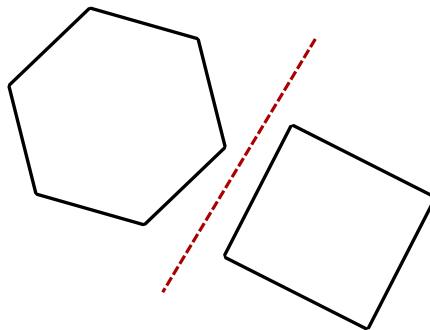
## Separating Axis Theorem

We have taken an in-depth look at a series of specialized algorithms, but there is a more generic theorem that allows us to determine if two convex polygons are colliding: The *Separating Axis Theorem* or SAT. This theorem states:

If two convex objects are not penetrating, there exists an axis for which the projection of the objects will not overlap.

This is connected to a simpler “human” explanation, which is:

If two convex polygons are not colliding, then you can draw a straight line between them.



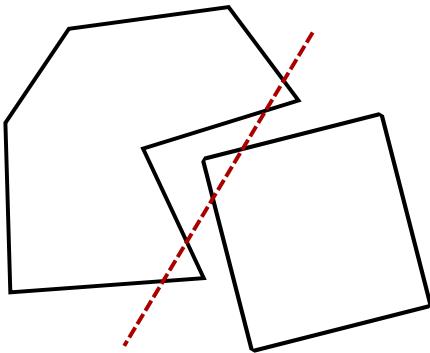
Example of how you can draw a line between two convex non-colliding polygons

Before delving further into the matter, let's see what we need to know:

- [The difference between a Convex and a Concave Polygon](#)
- [What a Projection is](#)
- [Some Vector Maths](#)

## Why only convex polygons?

To explain this, we'll use the “human explanation”: if one of the shapes is concave, there is a possibility that the polygons are not colliding, but we cannot draw a straight line between them.



Why the SAT doesn't work with concave polygons

Thus our algorithm would return a collision where there is none.

### Tip!

This problem can be solved by “decomposing” the concave polygons in two or more convex polygons, but for the sake of simplicity we’ll assume all polygons we are checking for collisions are convex.

Now let’s check the more “technical explanation”.

## How it works

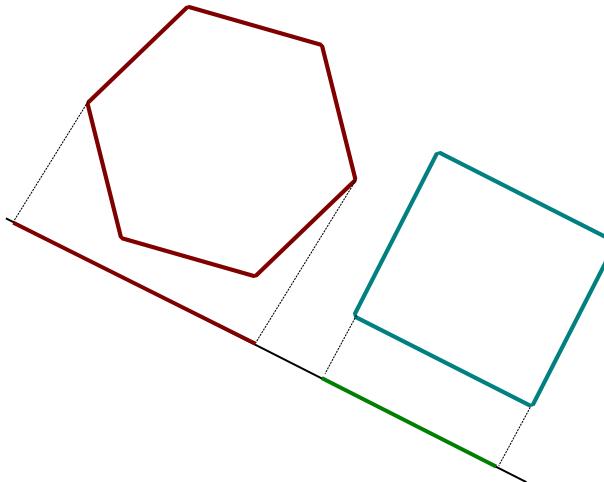
Let’s read the definition of the separating axis theorem again and break it down:

If two convex objects are not penetrating, there exists an axis for which the projection of the objects will not overlap.

The first part defines the condition: in case two objects are not colliding, then what follows is true.

For what we were concerned so far, axes were “aligned to the screen boundaries”, but axes can actually have different orientations and we can project shapes onto them.

The condition in our definition is represented as follows:



How the SAT algorithm works

As we can see, we have found an axis (which in this case is slanted) where the projection of the two shapes don't overlap. The presence of this axis where the projections don't overlap is guaranteed by the fact that the two polygons don't collide.

### Random Trivia!

We can now easily see why the “human explanation” is (for our own purposes) equivalent to the “technical” one: we just need to take a single point inside the “gap” between the two projections and strike a line perpendicular to our axis.

That's our “separating axis”.

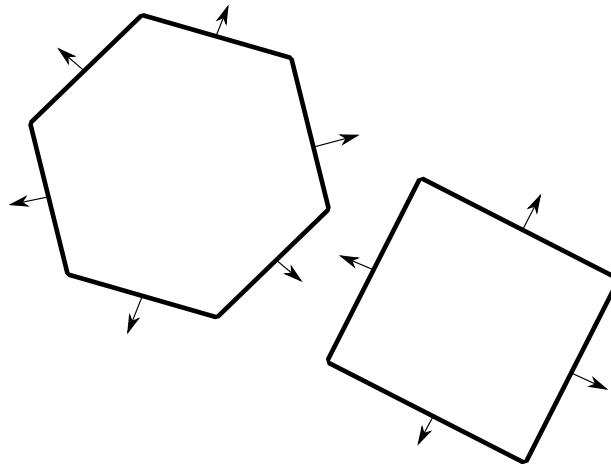
### Finding the axes to analyze

Now we only have a problem: we definitely can't spend an infinite amount of time trying all possible combinations in the hope of finding an axis where the projections don't overlap.

The fact is: we don't need to try them all. Actually we need to try just a few, as many as the sides of the polygons involved.

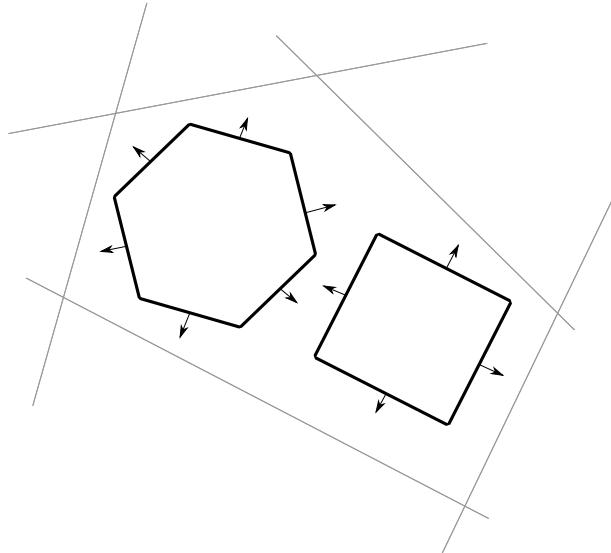
The axes we need to check are actually the axes parallel to the “normal of the polygon’s edges”. In layman’s terms: the axes we need to check are parallel to lines which are perpendicular to the edges of our polygons.

Let’s take it step by step, first we find the “normals”, which are just unit vectors perpendicular to the edges of our polygons.



Finding the axes for the SAT (1/2)

Now we just have to strike axes parallel to those normals, and those are the axes we will need to check against.



Finding the axes for the SAT (2/2)

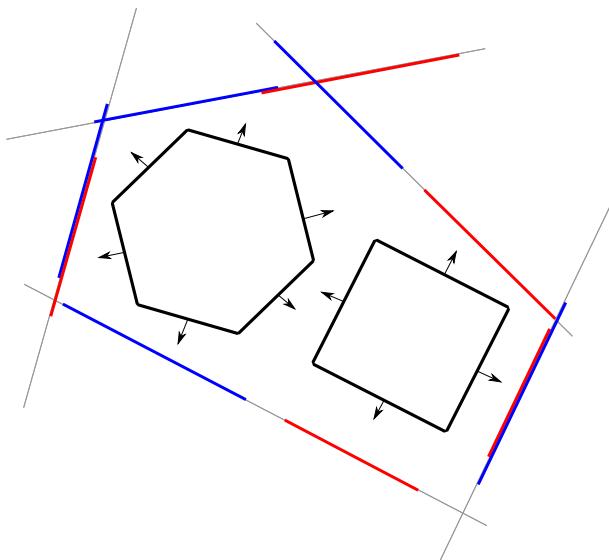
In the previous pictures, I chose axes around the two polygons, for the sake of clarity.

### Pitfall Warning!

Do not think that the axes we found are 5: there actually are 10. This is due to the fact that the figures I chose (for the sake of cleanliness) are a rectangle and an hexagon, which have edges that are parallel in groups of two.

### Projecting the shapes into the axes and exiting the algorithm

Now, for each axis we found, we need to perform a projection of the two polygons onto such axis.



Projecting the polygons onto the axes

Now we consider each axis on its own and see if the projections overlap.

As soon as we find an axis where the two projections don't touch (overlap), we know that the two polygons are not colliding. Thus we exit the algorithm.

If all the axes we scan have overlapping projections, we can say that the polygons we're analyzing are colliding.

In the example, we can find two axes that have non-overlapping projections, thus the worst case is that the algorithm misses both of them 3 times in a row and exits at the fourth iteration.

### Random Trivia!

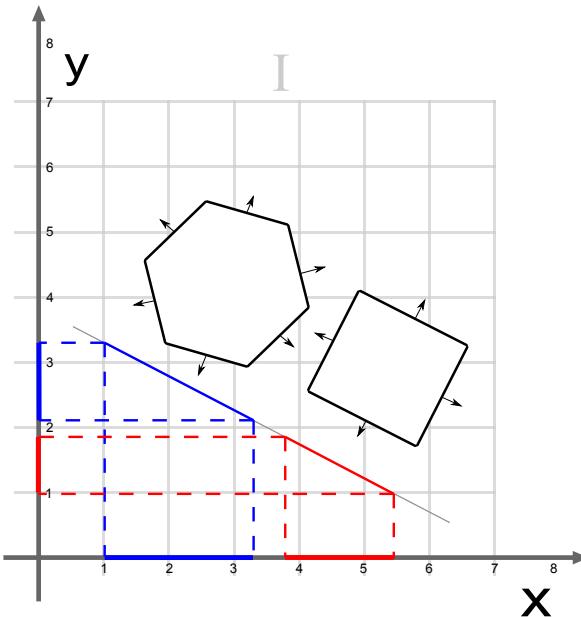
If you use Axis-Aligned rectangles as your “polygons”, you will notice how the Separating Axis Theorem will degenerate into something very similar to a simple AABB collision detection.

The only difference is that we’re checking a condition where the rectangles **don’t** collide.

Due to its nature, this algorithm has higher efficiency when there are few collisions, since it exits as soon as we find a separating axis (a gap in the projections).

### From arbitrary axes to “x and y”

The only thing that remains is how to switch from an “arbitrary axis” to our usual “x and y” axes. Here projections will help us again: we can simply project our projections.



Projecting our projections onto the x and y axes

If we look closely, we're just projecting polygons onto a bunch of axes so that they get "flattened to lines", then we're projecting such lines onto the x and y axes to see if there those lines are touching or not.

## Ray Casting

Sometimes it can necessary to use unusual techniques to detect collisions: ray casting is one of them. If well used (and with some "illusion magic"), ray casting can be a nice way to solve the "bullet through paper" problem.

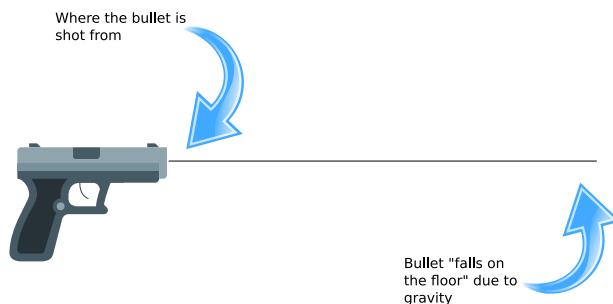
### What is Ray Casting?

Mostly used in 3D, ray casting is a technique where you cast an imaginary ray (usually of light) until it hits something, but its uses can go beyond that.

Let's take for example shooting a simple bullet: this can give some issues when the "bullet" is small and fast, as explained earlier in [The bullet through paper problem](#).

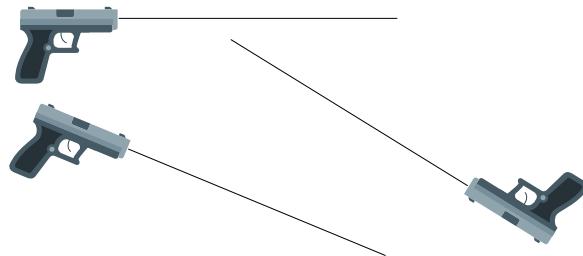
First of all, let's put up some (arbitrary) constraints that will help us making the computation easier and better performing without giving away our tricks too easily.

Our bullet will shoot from the barrel of our gun (duh!), but we also define a point where the bullet will despawn: this will limit our ray length and make our algorithm perform better. We can still give an excuse such as “bullets are affected by gravity” (which actually is true), and maybe use it as a difficulty management technique (stopping people from sniping the enemy can make the game harder and force the player to play the game the way we, the developers, want).



### How Ray Casting Works: Gun (1/2)

Attached to our gun, is an invisible line (our ray), that will follow every movement of the gun itself



### How Ray Casting Works: Gun (2/2)

When we want to shoot the gun, instead of using the previously stated “time-stepping techniques”, we perform a line-to-rectangle (or line-to-circle, or whatever we find best) collision detection, at the same time we play a really fast animation of the bullet shooting along the casted ray. If the cast ray hits an enemy, they’ll die (or get destroyed).

**Tip!**

If you find that the bullet animation won't align well with the enemy dying, the animation may not be fast enough. Some games even give up showing the bullet at all, and instead show a white line for a split second, that fades away. The effect works really well!

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

# Cameras

Nothing's beautiful from every point of view

---

Horace

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.

## 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").



Reference Image for Screen Space and Game Space<sup>1</sup>

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.

## Most used camera types

### 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]*

### 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]*

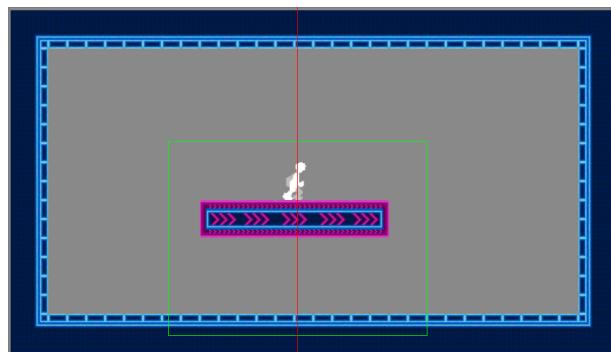
## Position-Tracking Camera

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 have 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.

## 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.



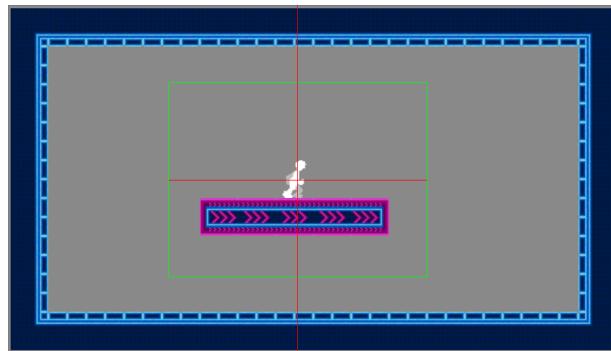
Example of an horizontally-tracking camera

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]*

## 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 manouevring, since precise manouevring could result in way too much movement from the camera.



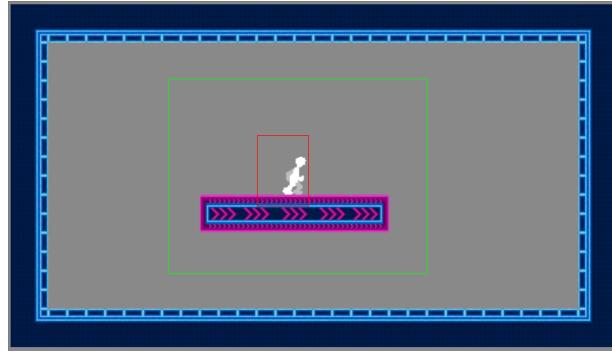
Example of a full-tracking camera

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

## 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.



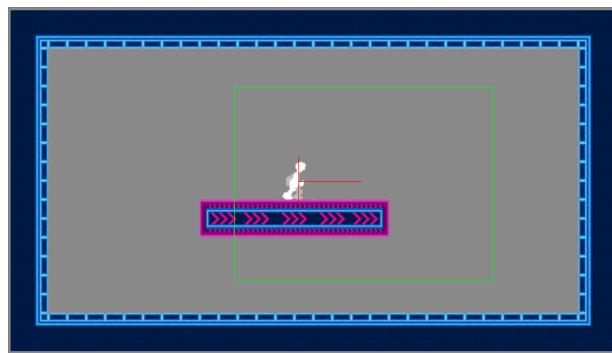
Example of camera trap-based system

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]*

## 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.



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]*

## Hybrid Approaches

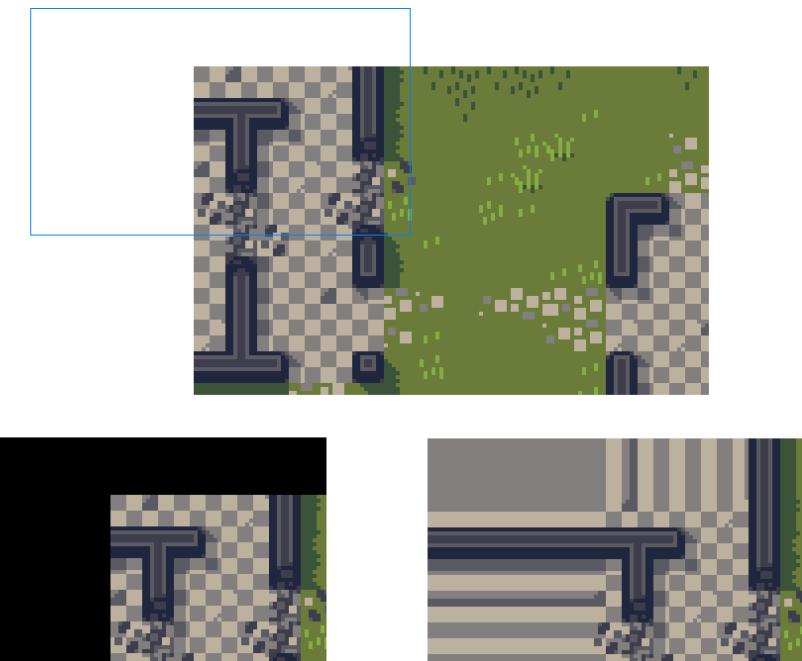
There are hybrid approaches to cameras too, mixing and matching different types of camera can give your game an additional touch of uniqueness. For instance in “Legend of Zelda: A link to the past”, the camera is a mix between a “camera trap” and a “grid camera”, where each zone is part of a grid, and 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!

## 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.



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.

- 
1. Jawbreaker tileset, listed as public domain at <https://adamatomic.itch.io/jawbreaker>

# Game Design Tips

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, talking about some genres and features in games, including some tips and tricks that can make the difference between a “good” and a “bad” experience.

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.

## Tutorials

### **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.

## 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.

## 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

## Consolidating and refreshing the game mechanics

### 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.

### 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.

## Rewarding the player

### 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).



Example of how to induce lateral thinking with environmental damage [1](#) [2](#) [3](#)

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.



Example of how to induce lateral thinking by “breaking the fourth wall”<sup>4</sup>

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).

## 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 undertake a hard task, such feats should be rewarded: for instance with a powerful weapon that can be used after some level-ups.

## 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.

## Reward the player for not immediately following the given direction

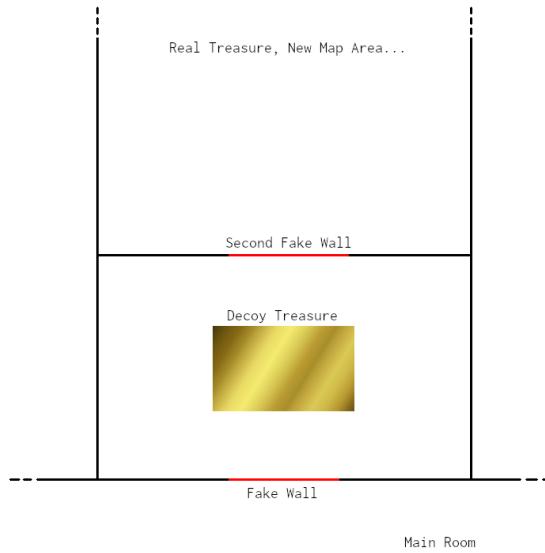
This is an extension of the previous point, the player should be rewarded for their exploratory efforts, even more when those efforts mean not immediately following the direction given by the designer.

“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.

## 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.



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.

## Reward Backtracking (but don't make it mandatory!)

To make the game's experience broader and richer, you may want to reward the player's exploration efforts by hiding treasure behind some backtracking.

For instance you can show the player a locked door somewhere in the level, such door will unlock and open after beating a boss monster or a wave-based challenge in the next room and hide some weapons that would otherwise be unlocked further into the game.

A nice idea would be “suggesting” to the player that something interesting happened, by playing the sound of the door opening as soon as the event is triggered. Another idea would be showing the player that the door opened (for instance if you're in an open area, the player would be able to see clearly that an open gate that was closed before).

The most important thing to remember is that all of this needs to be optional, a reward for the player's willingness to explore your levels further: avoid making backtracking mandatory, this will only feel like you're “padding the game” with nothing worth of note.

The player is paying you with their time and effort, it's only right that you pay them back with a pleasurable experience.

### **Random Trivia!**

A nice example of backtracking bonuses (although mandatory to 100% the game) is used in the level “Sphynxinator” in Crash Bandicoot 3: Warped.

When you start the level, you can run backwards and you'll find 4 crates (which are necessary to get the “Gem” and 100% the game, but not mandatory for the normal ending), one of which is an extra life.

The backtracking is really short and gives you a nice bonus.

## The “lives” system

Extra ships, 1-ups, extends, continues: these are all instances of what we can call the “lives system”. This system gives a more “arcade feel” to your game and adds an important challenge factor to it.

Without something that threatens a game over, beating the game is no longer a challenge, but it’s a matter of time. When overcoming a challenge is inevitable, it is not a challenge anymore, and the player will end up losing interest.

This is what the “lives system” is for: it’s a “sword of Damocles”, hanging over the player’s head, continuously threatening a “game over” and pushing the player to do their best in order to get as far into the game as possible.

“Continues” are just “a lives system for your lives”, they’re in a very limited number (or have a price, like putting another quarter into the arcade cabinet) and allow you to “continue the game” with a new set of “lives” without losing your progress.

As with all things in videogames, it doesn’t need to bring real challenge, but just the “illusion” of it.

Furthermore, lives and continues are a great tool to reward your player for their efforts: giving them an extra life every 20.000 points, granting a continue for a no-hit boss battle, putting a bunch of 1-ups in a hard-to-reach place are all great ways to challenge and reward your player and give your game more depth.

## 1-UPs

When a life system is in place, getting an extra life (a so-called 1-UP) is cause for celebration, since it allows the player to get further into the game or play with new and bolder strategies or just feel more at ease.

There are many ways you can reward the player with an extra life, such as:

- Finding a secret;

- Reaching a certain score threshold (for example every 100.000 points);
- Finding a certain item (a “physical 1-UP”);
- Complete a certain combo-chain (for example kill over 8 enemies without touching the ground);
- ...

No matter how the 1-UP is achieved, this should be celebrated with a jingle that is very recognizable: this will allow the player to “know” that they got a 1-UP without thinking too hard about it. Not “celebrating” this event would make it “ordinary” and uninteresting, while it’s extremely important in the grand scheme of things.

Some games even go as far as temporarily pause the game while the (short) jingle plays, that how important an extra-life is: “Stop everything! We got a 1-UP here!”.

## Loading Screens

Loading screens can be subject to design too and deciding what to put in them can really enhance the player experience with your product.

### What to put in a loading screen

“What can we put in a loading screen?” The answer may sound obvious to some, but a simple loading screen has a lot going on. Let’s think about it, the most barren loading screen imaginable has at least two elements:

- An animation, to make the loading screen less boring and to ensure the player that our game didn’t lock up;
- Some kind of progress indicator, to let the player know how far the loading routine has gone.

But we can put lots more into it, let’s take a

- **Story:** In story-heavy games, it may be a good idea to put a reminder, a briefing or just a couple sentences telling what’s happening story-wise. This was done in DOOM (2016).

- **World Building:** If the story is not-so-linear, a good idea could be just telling some facts about the world of the game, what some primary NPCs like, their habits, etc...
- **Tips:** Putting some tips to help the player is one of the most common things done with loading screens, these tips should be short and useful (no “press F to kill your enemy silently”, that’s basic controls).
- **Minigames:** If your game may take a potentially long time to load, making the player play a simple mini-game (maybe with rewards) while they wait. This was done on the Playstation 2 (and can be enabled on the PC version) of Okami, where you can earn demon fangs with two minigames.
- **Status-related sentences:** This is something that can serve both as a “loading screen filler” as well as “hidden debug feature”
  - **Actual loading information:** Some players may like knowing what their PC is doing, so showing “loading backgrounds” or “loading sound data” is a nice screen filler and can give your players a pointer in case the game locks up while loading.
  - **Funny phrases:** Instead of boring, actual loading information, you can put funny phrases like “inserting buckazoids” (Space Quest, anyone?), if you have a list of funny phrases connected to actual loading checkpoints, it will function as a small debug helper.

## Letting the player “exit” the loading screen

In some cases it can be a good thing to have a loading screen fade into the next stage (or area) directly, while in other situations it may be wiser to prompt the player to “exit” the loading screen themselves (maybe by a “Press any key to continue” prompt).

The most important factor is whether the areas we will load into are safe: if the player is not ready, they may get killed by enemies, and that feels unfair.

You should use a “press to continue prompt” when at least one of the following conditions applies:

- **Any area we load to may be unsafe:** We don't want our players to take a small break, walk around the room and come back to a dead character;
- **There is text on the loading screen:** Be it a tip, world building or story, the player may be reading it, and taking away the text will annoy them.

## Avoiding a loading screen altogether

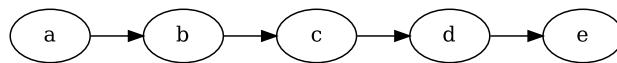
*[This section is a work in progress and it will be completed as soon as possible]*

## Designing the story and gameplay flow

When we are preparing the terrain for our game, it is vital to have an idea of how the story and the gameplay will unfold. There are lots of different types of gameplay, here we present some of them.

### Linear Gameplay

This is the simplest type of gameplay design: all story events come one after the other, without any possibility of deviating from the flow.



Example Scheme of linear gameplay

Very much like a presentation, there is no branching, but such linearity can present some advantages, like ease of testing and possibility of applying traditional storytelling tools which have been developed for thousands of years.

Summary of linear gameplay

<b>Gameplay Type</b>	<b>Flow</b>	Linear Gameplay
----------------------	-------------	-----------------

## **Advantages**

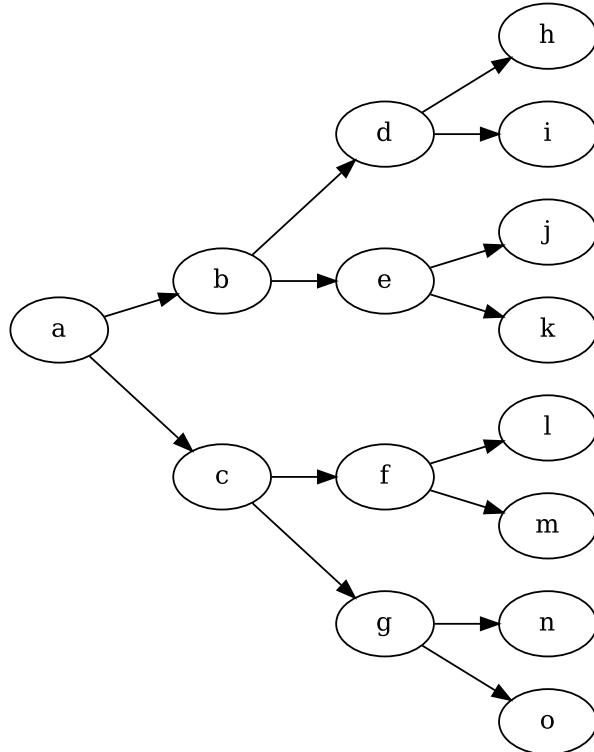
Simple and cheap to test, traditional storytelling tools can be used easily.

## **Disadvantages**

No replayability, the gameplay may not feel very “interactive”

## **Branching gameplay**

Going towards more complex flow types, we can use branching to allow for more interactivity.



Example Scheme of branching gameplay

This type of gameplay flow allows for a lot of interactivity by crafting the game in a way that player decisions have a direct influence on the story flow.

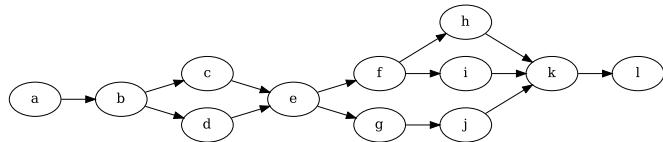
This gameplay flow is harder (and thus more costly) to test, but allows for multiple endings.

Gameplay Type	Flow
<b>Advantages</b>	Branching Gameplay Simple to implement, allows for a strong feel of interactivity, allows for a lot of replayability by giving gameplay paths.
<b>Disadvantages</b>	Hard and costly to test, can get out of hand if not managed correctly.

## Parallel gameplay

The Branching gameplay flow has a huge disadvantage: it can be really hard to manage and doesn't really suit well "more linear" games.

Here's where parallel gameplay comes into play.



Example Scheme of parallel gameplay

In this flow style, there are branches running "parallel" to one another, but merge into "mandatory events" (which are usually story related). This way we have varied gameplay while keeping the story essentially linear.

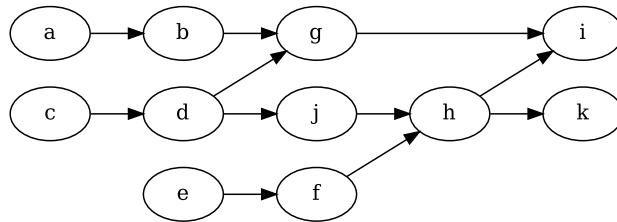
Gameplay Type	Flow
<b>Advantages</b>	Parallel Gameplay Moderately expensive to test, some traditional storytelling tools can be used, story is easier to manage.

## **Disadvantages**

Replayability suffers from a story standpoint. If not well-made the player will feel like the story is “on rails” from the get go.

## **Threaded gameplay**

A different kind of gameplay is the “threaded” version, where there are many “beginnings”, “middles” and “endings”, usually done by playing different characters.



Example Scheme of threaded gameplay

This gives more replayability by giving many different and intertwining stories that allow to better understand a “bigger picture” of some sort. This gameplay flow can be costly, since it requires testing all the possible paths and crossings.

## **Gameplay Type**

**Flow**    Threaded Gameplay

## **Advantages**

Good replayability, great for giving many “sides” to a story.

## **Disadvantages**

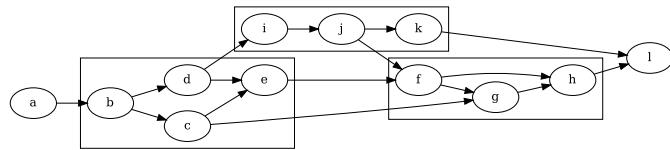
Testing all the paths can be costly, more difficult to manage.

**Random Trivia!**

This was done in Resident Evil 6, where different characters (and teams) have different stories that overlap.

## Episodic gameplay

A more “object-oriented” approach to storytelling can be done by making small “episodes” (like mini-stories) with many entry and exit points.



Example Scheme of episodic gameplay

We need to be mindful of loops (we don’t want to replay an episode that was already completed) when laying out our episodes. This gameplay flow allows for great interactivity, but kind of “forces” replaying the game to see all the possible episodes.

**Gameplay Type** Flow Episodic Gameplay

**Advantages** Great interactivity.

**Disadvantages** Tends to “force” replaying the game to see all episodes and paths, hard (and thus costly) to test and manage.

## Adding parallel paths

Nothing forbids us to mix and match methods to create something that suits our game better.

A very much appreciated and used gameplay flow is having a linear story with lots of “side quests” to give some diversion from normal gameplay, as well as replay value, since people are bound to miss some side quests.

## **Some game genres and their characteristics**

### **Roguelikes and Rogue-lites**

Roguelike games are usually games that involve dungeon-crawling and procedurally generated levels, usually with a fantasy background. In this small section we will take a look at the features that characterize roguelike games.

The most accepted interpretation of a roguelike game is the “Berlin Interpretation”, which is based on the features that follow. When games diverge from these features, but are still loosely based on the classic roguelike design, they are usually called “rogue-lites” or “roguelike-likes”.

### **Use of pseudo-randomness and procedural generation**

This is done to increase replayability: the dungeons (or levels alike) are generated procedurally, with a tinge of randomness added to them. Joining procedural generation and pseudo-randomness is better than simple pseudo-randomness, since the rules applied will make the level beatable without special equipment, as well as lead to more aesthetically pleasing levels overall.

### **Permadeath**

In the great majority of roguelike games, the death of a character is permanent. When a character dies, the player will have to begin a new “run”: the levels will be generated anew and the available loot will change too.

Usually permadeath is joined with an erasure of the savefile connected to the “failed run”, this avoids so-called “save-scumming”: a practice where

players would load back their savefile repeatedly to achieve better results (which is usually considered akin to cheating, in the roguelike field).

Another way to stop “save-scumming” is deleting the savefile when loading it, so when you save the only thing you can do to keep your savefile is exiting the game.

Permadeath makes the “save game” functionality more of a “suspension of the gameplay” instead of giving the player a recoverable state they can limitlessly return to.

## **Turn-based Gameplay**

Like tabletop games, the gameplay of roguelikes is usually turn-based: this allows the player to take as much time as needed to take a decision.

## **Lack of mode-based gameplay**

Roguelikes don’t have a real concept of “progression”: they allow you to do anything from the get-go, without blocking any action just because you’re at a certain point in the game.

## **Multiple ways to accomplish (or fail!) a task**

Roguelikes usually allow you to complete a task in many different ways, so many in fact that it seems the developers thought of everything. Let’s take for example a locked door, a roguelike game would give you many options:

- Find the key or trigger to open such door;
- Lockpick it;
- Burn it down;
- Find a way around it;
- Kick it down;
- ...

This also means that you have to be careful with your actions: if a weapon freezes entities when it touches their flesh, you better have a pair of gloves

handy.

## **Resource Management is key**

Resource Management in roguelike games is vital: usually they feature a hunger mechanic, as well as healing items, weapons and various loot that the player must sort through to be able to survive. The player will be forced to leave some loot on the floor of the dungeon, or choose between a known weapon and something unknown that may be weaker or “cursed”.

## **Peace was never an option**

Most roguelike games are based on hack and slash mechanics, where your main goal is killing monsters. In this kind of games, “peaceful options” don’t exist (although they may exist, in a somewhat temporary fashion, to put leverage on some stealth mechanics - like getting a better weapon to kill a powerful enemy by first sneaking around them).

## **Dealing with the unknown**

Roguelike games are heavily based on the concept of “unknown”: you need to explore an unknown place, finding loot which powers are unknown and should be identified. Magical items change with every run, and give just vague descriptions (like “a red potion”) which may heal in one run and kill you in another.

Furthermore items can be subject to change, acquiring or losing traits due to environmental alterations or player modification.

## **Tips and Tricks**

### **General Purpose**

#### **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.

### **Random Trivia!**

This was implemented in both DOOM and Assassin’s creed, where the last portion of health had more “hit points”.

In Bioshock when you take your last point of damage, you get about 1 or 2 seconds of invulnerability.

### **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.

## **Random Trivia!**

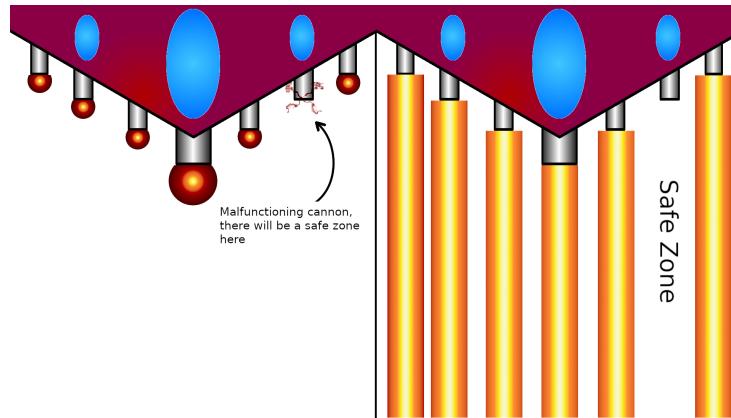
This trick is used in the game “Firewatch”, where not responding to a dialogue prompt is a noted decision.

## **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”).



Example of a telegraphed screen-filling attack in a shooter

## **Random Trivia!**

A form 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 somehow communicates both the presence and position of enemies.

## Minigames

Many times underrated, minigames are a really vital part of a great game experience.

Minigames can be a fun diversion from the main game, extending the engagement time, as well as a priceless resource for bigger open-ended games: you can use common “low-level” materials to feed into the minigame to get better materials, weapons or prizes.

This is a win/win situation, you throw away unused materials to get useful tools, materials or cosmetics, also playing into the mechanism that maybe some people will get things wrong and need the “low-level” materials again, further lengthening the engagement of your game.

## When unlockables are involved, be balanced

When you’re creating a game that involves “unlockables” (for instance a roguelike where you unlock more items for the upcoming runs), you should absolutely balance your unlockables in a way that compels the player to unlock them.

If you hide a “negative item” behind an unlockable, the player will actively avoid doing the actions that lead to unlocking such item. This is especially true now, in the age of widespread Wikis.

If you have to implement unlockables you should either:

- **Make the unlocked item a “good item”:** this will naturally compel the player to unlock such item to make the subsequent runs easier and more fun and varied;
- **Make the unlocked item a “neutral item” with situational good outcomes:** the player will be less attracted by these items, but the

situational good outcomes (we can call them “interactions” or “synergies”) can make the player willing to put in the effort to unlock such item;

- **Unlock 2 items at once: a “good” one and a “bad” one:** the player may be less attracted by this “good+bad” combination, but may still be willing to go through with the unlock effort for the sake of the “good item” and also for added variety in their next runs;

If you really want to gate a “bad item” behind “a gate”, a good idea would make the “bad item” a pre-requisite for unlocking a “very good item”. Alternatively, you can unlock the bad item “on the way” to unlocking a “good item”: for instance you can make “beat the first boss 5 times” a requirement for the bad item to be unlocked, while “beat the second boss 10 times” could be a requirement for the “good item” to be spawned.

### Tip!

Remember: you should always account for wikis, some people think that wikis “ruin the surprise” of the game, while others use wikis just out of curiosity, some again instead use wikis as a “guide” to make the game easier or organize their strategy better.

## Shooters

### 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.

# RPGs

## Grinding and farming

When it comes to games part of the RPG genre, two words must be in your dictionary: **grinding** and **farming**, both as a player and as a game developer.

Sometimes used as synonyms and similar in execution, these terms are actually different and have different objectives. Let's see how.

### Grinding

Much like “grinding an axe”, grinding in RPGs entails cleaning areas from enemies repeatedly (either by re-playing missions or just doing random encounters) with the objective of earning “experience points”, thus making yourself stronger.

Grinding is somewhat a “self-leveling game design hinge”, allowing you to have some leeway when designing the difficulty of your levels: if a player likes having an easier time, they will “grind themselves” to a higher level; if instead they prefer a challenge, they will power through the “easier parts” until they find the challenge they seek (due to being probably “underleveled”).

We can also use “designed grinding”, (as well as “level gates”, where you need to have a certain amount of experience to continue) to pace our game and eventually even lengthen the experience a bit.

### Tip!

When designing your levels and “designing your grind”, you need to be mindful of your target audience.

Some cultures are used to (and enjoy) a higher amount of grinding than others, so too low of an amount may feel unsatisfactory to them, while

an amount too high may be frustrating.

You should also be very careful on “forcing grinding” on your players: players like having choice and really dislike having anything forced on them, and this can change with your target audience.

### **Random Trivia!**

Super Hydlide for the Sega Genesis/MegaDrive is one of the games that had its experience requirements tailored to the tastes of the market it was targeted to.

Considering the Japan release as the baseline, the US release sees its experience requirements halved.

### **Farming**

Farming entails the same actions as grinding, but here we are using enemies as “farming animals”, the objective is obtaining a certain amount of materials to obtain a weapon or item.

The most important aspect of “designing farming” is definitely reward the player for their farming: if a “special item” requires a lot of materials (and thus a lot of farming), such item should be worth the effort, or the player will feel cheated out of their time, effort and materials.

### **Note!**

You should be mindful that some players will exploit some of your more complex mechanics to be able to farm for items and currency faster. This happened in the game “The Witcher III”, where players used to kill cows and then meditate to make such cows respawn.

## Random Trivia!

In the game “The Witcher III”, precisely in patch 1.05, a mechanic to prevent such exploitation was introduced. The “Bovine Defence Force Initiative” consisted of a respawning moderately-leveled monster (called “Chort”) that attacks the player, thus making the farming very dangerous for low level players.

On the flip side, higher level players could exploit this endlessly respawning monster to gain Chort Hides, which are worth more currency than Cow Hides. This was patched by making only one Chort spawn.

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

## Leveling Curves

When it comes to RPGs there are different ways to “design the leveling curve” in a game, depending on how you prefer designing your game.

### “Exponential” curve

One of the most known ways to shape your leveling curve is making each further “skill level” require more experience points than the previous one, for example:

An example of exponential level curve

### Level   Experience Points

Level	Experience Points
1	5.000
2	10.000
3	20.000

<b>Level</b>	<b>Experience Points</b>
4	40.000
...	...

This type of leveling curve entails that each newer (harder) enemy gives out a higher amount of experience (not necessarily the quantity to make the experience feel “linear”).

The exponential nature of this level progression allows the “push” players towards harder enemies, since grinding lower-tier enemies becomes less and less efficient the more your level increases.

#### **“Level-based” experience rewards**

As an addition or an alternative to the exponential curve, some games try to “push” the players more towards harder enemies by scaling (down) their experience points with your level.

This method allows you to further shape the curve not only to try and prevent mindless grind of low-tier enemies, but also by rewarding challenging harder enemies.

This method can be implemented in many ways, one of them could be assigning a level to the enemy itself and then scale the experience rewarded by the difference between the player level and the enemy level.

An example of “level-based” experience rewards

<b>Player Level - Enemy Level</b>	<b>Experience Reward</b>
< -3 (very underleveled)	35.000
-2 (moderately underlevel)	20.000
-1 (slightly underlevel)	10.000

Player Level - Enemy Level	Experience Reward
0 (ideal)	5.000
+1 (slightly overlevel)	2.500
+2 (moderately overlevel)	800
> +3	1

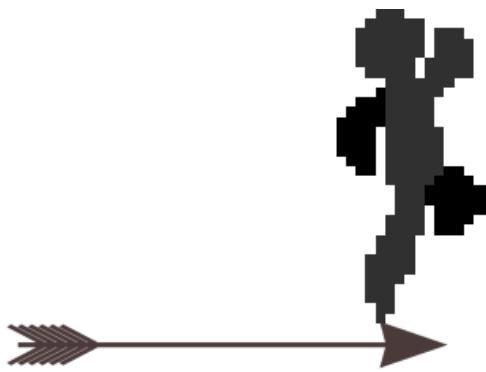
## Perceived Fairness

### You don't need precise collision detection

Here we go, spouting bold claims again. Here's a bolder one: you don't **want** precise collision detection. It's slower, harder to implement and most of all, the player may get annoyed at it.

In the heat of a gaming session, with all the action going on, the player may become a bit "blind" to small things: this means that they will perceive as arbitrary anything that is not "evident".

Let's take this arrow vs "generic jumping man" example:

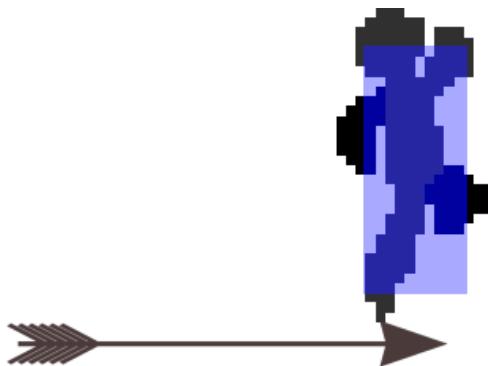


A pixel perfect detection would trigger in this case

With still images, things are obvious: that's a collision, it's on the tip of the character's toe, but it's a hit. It may be 1 or 2 pixels (it's actually 3), but it is

still a hit.

Now let's imagine the image moving: the arrow darting from left to right while the character is ascending, it looks like a near-hit, the player will appreciate the adrenaline rush of a near-death. If we go and declare that as a "hit", the player won't understand why, they will say "that's unfair, it just missed me". Our collision detection was **too precise**.



A smaller hitbox may save the player some frustration

Sometimes we just need a hitbox that is small enough to avoid these "looks-like-a-miss" incidents: the instances where the collision detection triggers are evident and the player will appreciate the sensation given by the near-deaths.

Obviously you need to strike a balance, if the hitbox is so small that evident hits are counted as misses (with few exceptions, like [bullet hells](#)) it will break the immersion.

## Immediate dangers should be well visible

One of the biggest frustrations a player can encounter is definitely being damaged (or killed) by a hidden object.

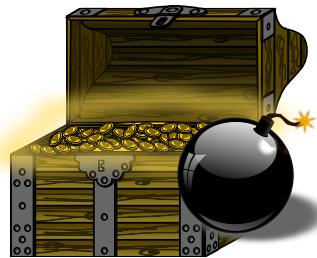
To avoid these "unfair shots", we should draw immediate dangers "late" in the drawing phase of the game loop, but there is an issue: we don't want to break the immersion.

Let's take the following example: a bomb gets spawned just behind a chest by an enemy, and our character is dangerously close to it.



A bomb spawned behind a chest, drawn in a realistic order

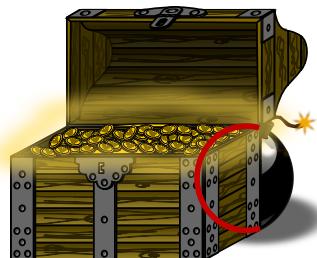
If the bomb is “behind” the chest, we can’t suddenly make it pop “in front of” the chest, that would definitely ruin the immersion, also it may end up messing with the forced perspective that some 2D games use.



Moving the bomb in front of the chest may ruin immersion

As you can see, even though the bomb has a shadow, it looks like the bomb and the shadow are “floating mid-air”, thus ruining immersion.

Different games implement different solutions to the problem, some prefer highlighting the danger with an outline drawn over everything, something like the following:



Highlighting the hidden part of a danger can be useful

Other instead prefer making the “foreground objects” semi-transparent, so that the player can see what lies behind.



Making objects transparent is a solution too

This solution is usually applied when the player themselves are behind the obstacle, giving a more “interactive” and “less confusing” feel to the entire game.

## Miscellaneous

This section denotes some various things that don’t really fit in “tips and tricks” but are still related to game design.

### You cannot use the “Red Cross” in games

#### Note!

What follows **is not legal advice**. I am not a lawyer.

If you want to know more (as in quantity and quality of information), contact your favourite lawyer.

When you are developing a game, you will be tempted to use the famous “red cross” symbol on your health packs or health-related items.

#### Don’t do that

The red cross symbol (a red cross over white background) is not in the public domain, but it’s actually a symbol governed by the ICRC (International Committee of the Red Cross) and you may get in legal trouble for misusing it.

Halo and Doom changed their health packs symbol, from the red cross to a “red H” and a “pill” respectively.

The ICRC enforcement of this rule is inconsistent, but it would technically be a violation of the First Geneva Convention, chapter VII, articles 44 and 53.

Also states themselves (like Canada) tend to have rules of law regulating the use of the symbol in more detail.

## Auto-saving

Some people may consider auto-saving a simple “quality of life improvement”, but it can also save the players a lot of frustration in case your game crashes: trust me, no matter how good your programming is, your game will crash (it may be a buggy graphics driver, an edge case that hits 0.0001% of the time or just bad luck).

If possible, you should provide the player with both an auto-save feature and a “manual save” one, this way the player can save where they want but also have a back-up just in case.

To implement an auto-saving feature, we need a slot to auto-save into, so we can choose one of two ways:

- **Choosing the save slot when starting a new game:** this means that the auto-save feature will auto-save and overwrite the selected save slot at every major event, which may be not desired. This is where the manual saving feature comes handy: allowing the player to save manually will also allow them to create a backup savefile.
- **Dedicated “auto-save” slot:** this leaves the manual saving feature intact, but also adds a “special saving slot” the player can’t save onto. This slot is dedicated to the most recent auto-save (regardless of the save slot we load from).

## Feedback is important

It is extremely important to add feedback to actions, such as hits: a good visual feedback and the right sound can make all the difference in the world for your game.

The most common visual reaction to a hit is lighting up (by adding a white overlay) the sprite that got hit: this way it is really evident that a hit happened.

The visual feedback should also mirror the effectiveness of the hit too. An explosive weapon should do tons more damage than a single bullet: if this doesn't happen the weapons will feel unbalanced and just badly designed.

# **Part 4: Creating your assets**

- 
1. 32x32 Chests attribution: Bonsaiheldin (<http://nora.la>), hosted at [opengameart](#) ↵
  2. Simple SVG ice platformer tiles, listed as “Public Domain (CC0)” at [OpenGameArt.org](#) ↵
  3. Fossil (Undead) RPG Enemy Sprites attribution: Stephen Challener (Redshrike), hosted by [OpenGameArt.org](#) ↵
  4. Jawbreaker tileset, listed as public domain at <https://adamatomic.itch.io/jawbreaker> ↵

# Creating your assets

Art is not what you see, but what you make others see.

---

Edgar Degas

## Assumptions

This book (for now) assumes you already have some minimal knowledge of some matters, including but not limited to:

- Bars (in music)
- Beats (in music)
- Basic music notation (either *letter notation*<sub>g</sub> or the *modern music notation*<sub>g</sub>)

## Graphics

### Some computer graphics basics

Before we start doing anything, we need to know some basics about graphics. In this section we will introduce some terms like “bit depth” (or “color depth”), “filtering”, “sprite sheets”, “virtual resolution” and others.

We will see a lot of stuff, from a bird-eye view of how graphics are stored in video memory to normal maps and more.

### Color Depth

Raster graphics use bits to represent the color of each single pixel, the amount of bits used for each pixel is known as *color depth* (or “bit depth”).

The color depth used for our images can influence the performance and look of our game: more bits means more color choices, but also more memory occupied by those colors. Coupled with high resolutions, an image can easily (if uncompressed) weight MB worth of memory.

Let's see the most common color depths used in history:

- **1-Bit color:** Each pixel gets only 1 single bit, which means it's either black or white. This was used mostly in text-based systems, but some indie games manage to get great results out of just two colors.
- **2-Bit color:** Each pixel can select from a palette of 4 colors (usually fixed). This was used by CGA cards on the IBM, and usually the colors could be chosen from one of four available palettes: "white, black, light cyan, light magenta" or "yellow, black, light red, light green". The other two palettes are just a "low intensity variant" of the ones we've just seen.
- **4-Bit color:** Here we start seeing 16 colors, usually chosen from a selection of fixed palettes. This was used by EGA cards on the IBM, as well as the Commodore 64 (along with "color cells").
- **8-Bit color:** 256 colors chosen from a fully programmable palette (that's some luxury right there!). Used on VGA cards at low resolution.
- **16-Bit color:** Sometimes called "High Color", allows for up to 32768 colors with transparency, or up to 65536 without transparency.
- **24-Bit color:** Sometimes called "True Color", this is the most used color depth, allowing for over 16 million colors. Each red-green-blue channel has 256 possible values.
- **30-Bit color:** Sometimes called "Deep Color", this format allows for 10 bits per channel and over 1 billion colors on screen. Supported by many graphic cards and some high-end mobile phones.
- **48-Bit color:** This format allows for hundreds of thousands of billions of colors (if you want to read the number, it's around 281474976710656). This is used by image editing software to avoid loss of data while working with colors.

## Direct Color vs. Indexed Color

There are two main ways to represent color in digital images.

The first is “Direct Color”, which usually 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.

## **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).

## **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).

## **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.

### **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.

### **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 mean that we cannot make semi-transparent surfaces, since only that specific color will be fully transparent, and that's it.

## Texture Filtering

Sometimes your images will need to be scaled or filtered to avoid annoying artifacts, in this small chapter we will see some filters and how they look.

### Nearest Neighbor Filtering

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

### Bilinear Filtering

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

### Trilinear Filtering

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

## 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.

### 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 been 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.

### **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!

### **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.

### **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.

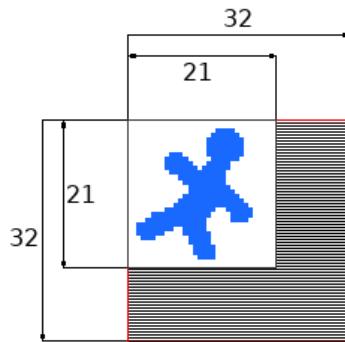
### **Learn to deconstruct objects into shapes**

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.

## 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.



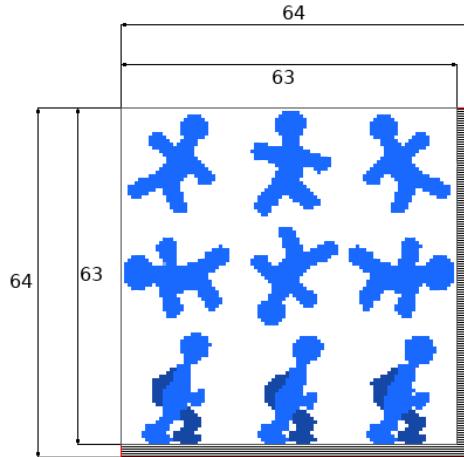
Example sprite that gets padded to match hardware constraints

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.



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]*

## Virtual Resolution

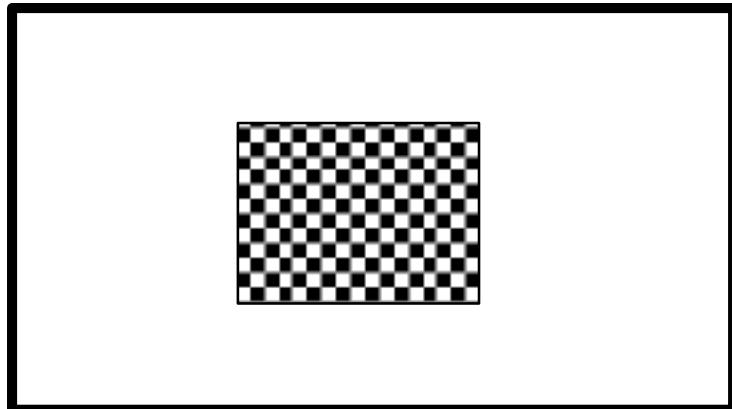
There are times where having the crispest graphics around is not a good idea, for instance in Pixel-Art games.

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”.

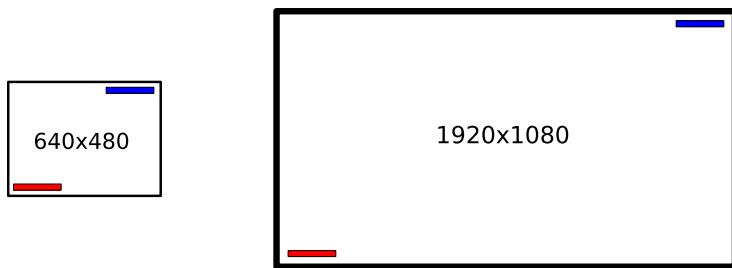
This clashes with the continuous push towards higher resolutions, which can mess up your game in a variety of ways, like the following ones.

If the game is forced in windowed mode, you’ll have a problem like the following:



Windowed Game Example - A 640x480 game in a 1920x1080 Window

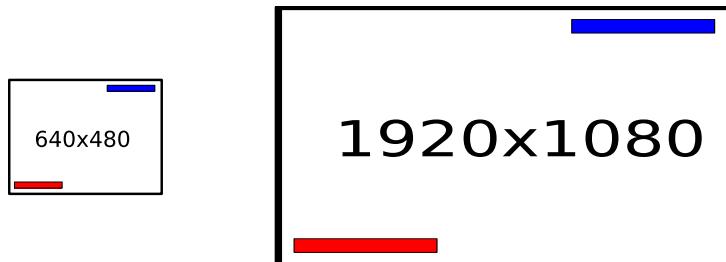
The game window is way too small and hard to see, and can get even smaller if the **HUD<sub>g</sub>** takes even more space out of the window. This can be mitigated by calculating the position of each element in comparison to the window size, this although can result in items too small (or too big if downscaling), like the following HUD example.



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).



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).

For more details, check the [filtering](#) section.

## Using limited color palettes

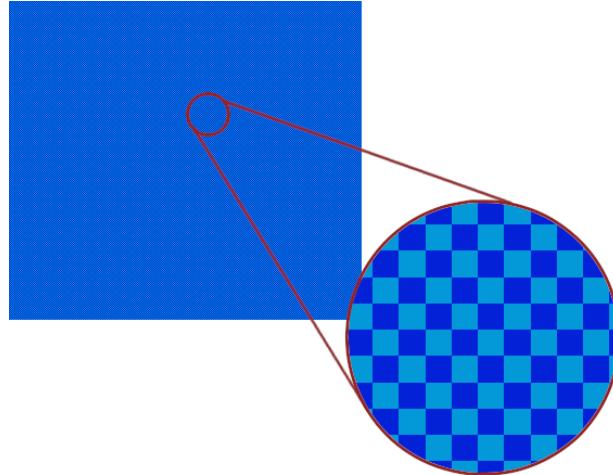
*[This section is a work in progress and it will be completed as soon as possible]*

## 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.

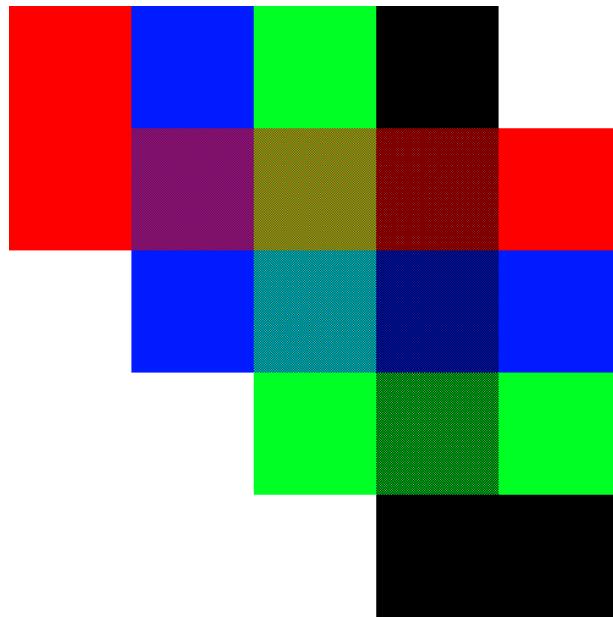
Using two different levels of blue, we can use a dithering pattern to obtain a new tone of blue, like the following:



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:

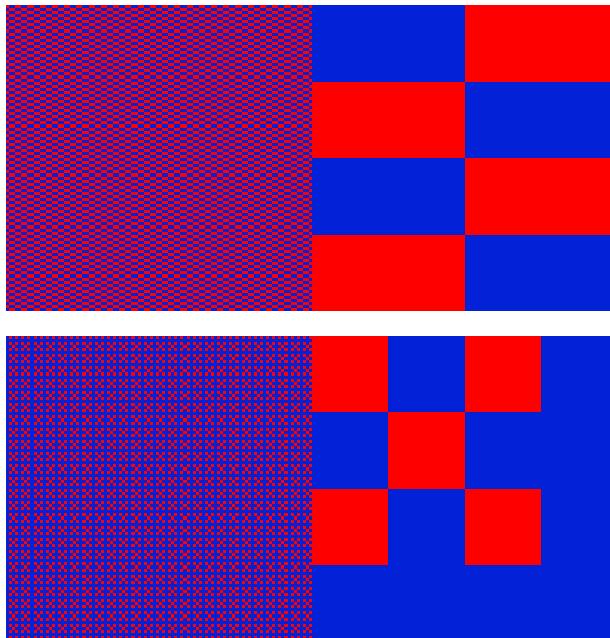


Dithering Table Example

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:



Some more dithering examples

## 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.

### 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.

### **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 muted 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

### **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

## 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 (color addition), not the traditional color wheel*

While, talking about colors made by paint (color subtraction) we have the following color pairs:

- **Magenta and Green:** Choose magenta, then yellow+cyan gives green;
- **Yellow and Purple:** Choose yellow, then magenta+cyan gives purple;
- **Cyan and Orange:** Choose cyan, then magenta+yellow gives you orange.

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

Also the opposite rule may apply:

| Keep the main gameplay elements contrasting, so to attract the attention towards them

An orange-robed character will be easier to follow on a blue-ish background, for instance.

## 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.

## 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 sprite, 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 pokémon).

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.

## Pixel Art

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

### 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]*

## Tools

When it comes to pixel art, your most used tools will be:

- **Pencil:** Its hard borders are ideal for base colors and outlines, allowing per-pixel editing without worrying about blur radius;
- **Brush:** The brush tool's soft borders are great for shading, in case you want to go for a more “hi-res” look;
- **Paint Bucket:** This is great for filling outlined areas with a base color. This allows also to create “zones” that help you distinguish between different parts of a sprite;
- **Eraser:** Great tool for removing stray pixels! Just make sure the eraser is set to delete the whole pixel and not just “make it more transparent looking”.

Opacity options for tools are great for shading: you can take the same color and blend it with other existing colors, or make it stronger by going through the same area more than once. This makes the process more akin to real painting.

## Layers

As with all other drawing styles, layers are your best friends in pixel art too. Just make sure to start with a transparent layer, since some drawing programs (for memory and CPU efficiency) won't let you easily add a “transparent background layer” afterwards.

Layers allow you to work on something new without affecting what's already "ready", as well as separate parts of a character (arms, legs, torso and head, for instance).

You can always "flatten the image" (merge all the layers into one) later.

## Sub-pixel animation

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

## Normal Mapping

If you want your 2D game to look amazing, you can't escape shaders. One of the ways to use shaders is calculating light, but since we're in a 2D environment, we don't have "normal vectors" to use to calculate how the light interacts with our 2D objects.

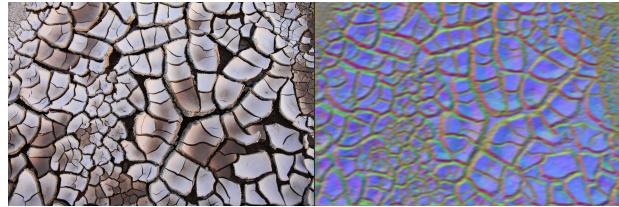
Enter normal maps: these are sprites that "map" their color channels to the direction of the "normal vector": this means that communicate direction with color. Awesome, isn't it?

### Random Trivia!

If you're curious, normal maps usually have this color to direction mapping:

- x (which can go from -1 to +1) is mapped to the red channel (which can go from 0 to 255)
- y (which can go from -1 to +1) is mapped to the green channel (which can go from 0 to 255)
- z (which can go from 0 to -1) is mapped to the blue channel (which can go from 128 to 255)

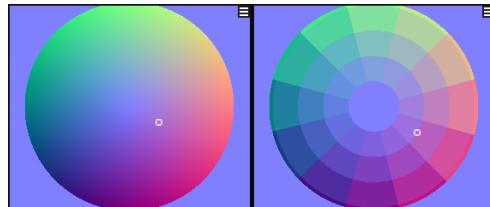
This is an example of a texture, along with a possible normal map:



Example of a normal map

There are many ways to get a normal map: you can try to get a program to generate one for you, make the object in a 3D modeler and extract the normal map from there, or just draw it by hand.

If you choose the last option some tools, like Aseprite, have a “normal mapping” mode that shows you this special color picker:



Aseprite’s normal mapping color picker (both in its normal and discrete versions)

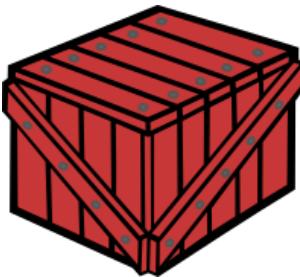
Just imagine this color picker like a “3D sphere” and pick the color of the face of the “3d surface” you’re trying to draw.

### Pitfall Warning!

Be careful with your shaders, some may expect one or more of your channels to be “flipped”.

### A simple example

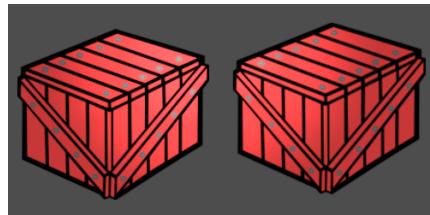
Let’s take a simple box, with no shading, like the one below:



A box that will be used to show how normal maps influence light

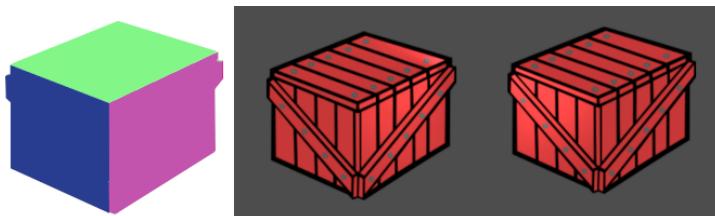
Now we'll shine a light on the box, without any normal map: this will happen twice:

In the first example (left) the light will be a round gradient that will come from the top right corner of the image, while in the second example the light will be a bit stronger and coming from the top left corner. This is the result.



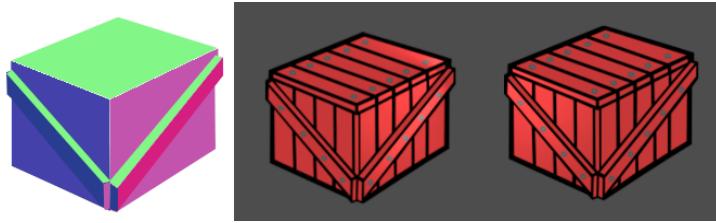
How the lack of normal mapping makes lighting look artificial

Now we'll draw the simplest normal map possible: just filling the 3 faces of the box that we can see with the (kind of) corresponding colors from our “3d sphere” (the normal mapping color picker), we can see the result is very different



How normal mapping changes lighting

Now let's make something a bit more detailed, by highlighting the faces of the cross-braces on the sides of the box, the way they're lit it's again different:



A more detailed normal map results in better lighting

You can get as detailed as you want, but remember that it may have some performance impact if you go overboard with many sprites.

## 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.

### Tiles

#### Getting started

When you are starting to make a new tileset, it's a good idea to begin with a base sized 5x5 tiles or more (so if your single tile is 32x32 pixels, the image will be at least 160x160 pixels): this can give you variations on the tileset that won't make the "tiling" (repetitions) so easy to catch when the map is made.

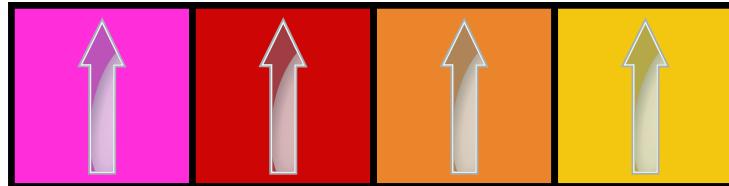


Example of tile “alternatives”

If you think something does not fit in a tileset, try to think what the surrounding area would look like: darker grass could be overgrown, a darker spot in the water could signify deeper water, shadows and light help too.

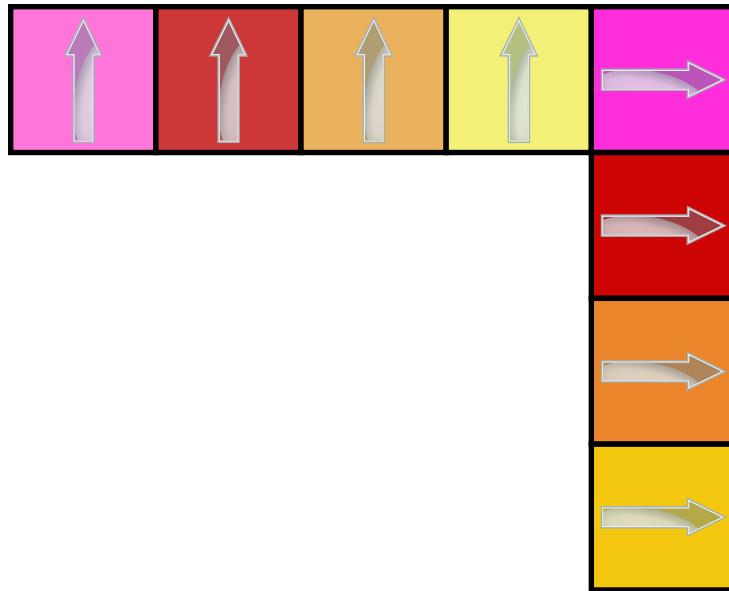
When it comes to “corner tiles”, using the same tile, rotated in 90 degree steps is a great basis to build upon, after that you can edit the tiles accordingly.

This “rotation trick” can be used for most of the tiles you create, let’s take for instance the following diagram, representing some tiles:



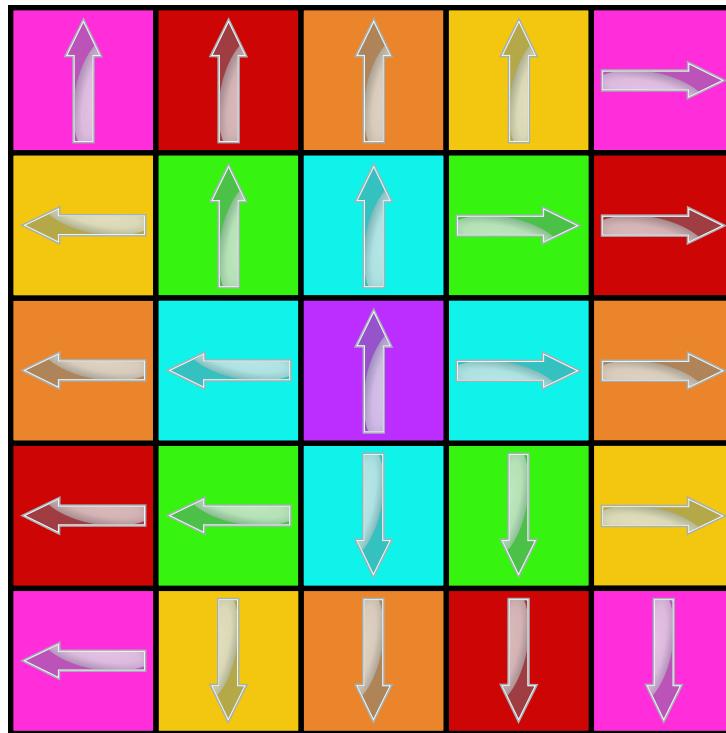
Tile “Rotation Trick” (1/3)

We can take the tiles and rotate them, to get something like the following:



Tile “Rotation Trick” (2/3)

If we continue with copying, rotating and pasting, we can obtain a great basis for our tileset:

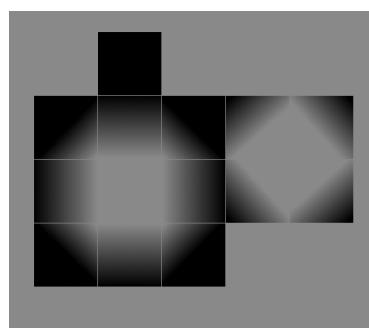


Tile “Rotation Trick” (3/3)

After that we can edit and make it so tiles are seamless, while putting the minimum amount of necessary effort to create something convincing.

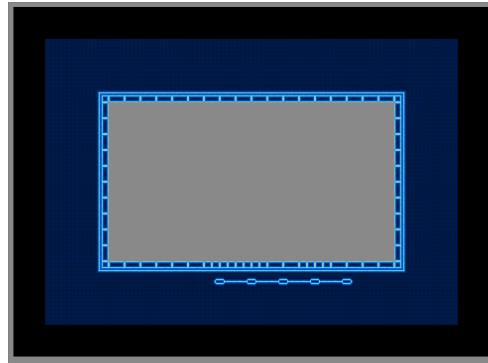
#### **Creating “Inside rooms” tilesets**

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:



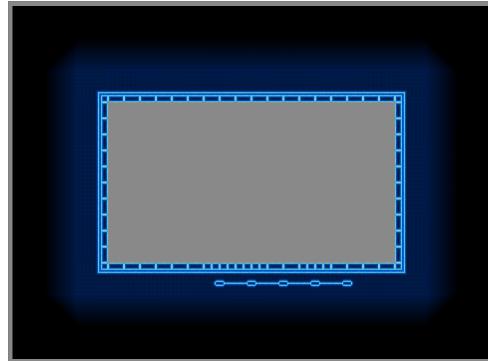
Example of black and transparent tileset used in “inside rooms”

Such tiles can then be overlaid onto something like the following:



Example of incomplete “inside room”

And we obtain the following result:



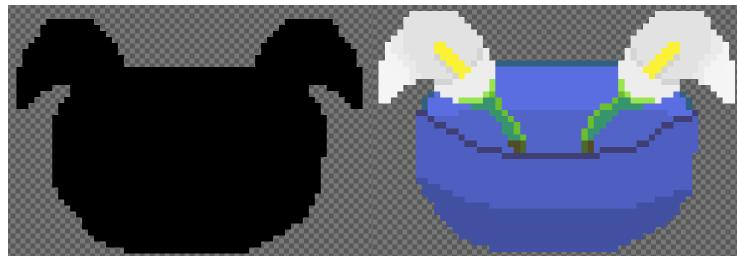
Example of “inside room” with the black/transparent overlay

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

## Sprites and icons

### Shape first

When you’re making some kind of sprite or icon, you should always get the basic shape of the object down first, then you can give the object more depth and detail with colors. This will help you understanding the space occupied by your object.



How color can completely change an object

It's a puppy! It's a jester! It's a... hastily drawn flower vase?

This is the power of color, you can change the entire nature of an object by changing how it's colored, but having the basic shape of the object down first will help you a long way.

## Sounds And Music

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

### Some audio basics

Before creating sounds and music, we need to clarify some terminology, as well as learn some basics before diving into FM synthesis like wave forms. After that we can learn about trackers and Software DAWs.

In this section we will learn about sample rate, bit depth, lossy/lossless formats and clipping, among other things.

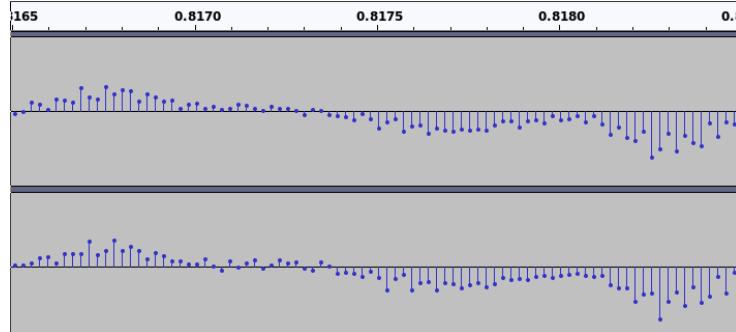
#### Sample Rate

Differently from Analog Audio, which is continuous (as it 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*.

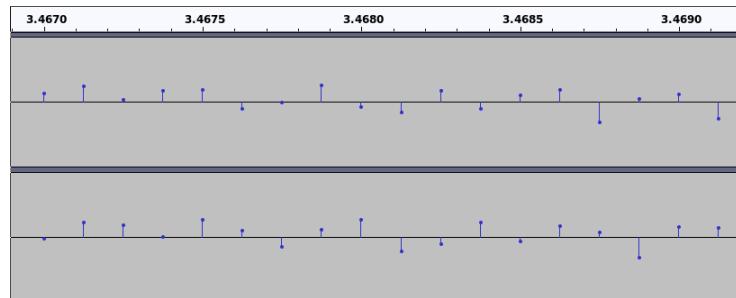
The more the samples per second, the more detail we can squeeze into our audio files, but at the same time the bigger the file will become too.



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.



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.

## 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 bit depth, each sample will have only 2 values:

- **0:** Mute
- **1:** Blast at full volume

Which strongly reduces the quality of the audio.

### Random Trivia!

In Pokèmon Yellow for GameBoy, Pikachu’s voice was encoded with in 1-bit depth.

If we had a 2-bit depth, we could make each sample have more volume values:

- 00: Mute
- 01: 33% volume
- 10: 66% volume
- 11: 100% volume

Usually audio has a 16 Bit depth, but more modern systems make use of 24 Bits or even 32 Bits.

## 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 (uncompressed);

- AIFF (usually uncompressed, but AIFF-C supports both lossless and lossy compression);
- FLAC (lossless compression).

## **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 considered 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).

## **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.

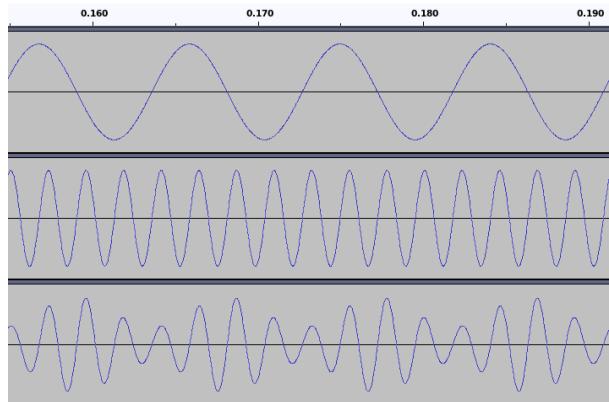
## **Sound Synthesis**

Now we’re entering technical territory. We’re going to talk about sound synthesis: the art of creating sounds, also called “sound synthesis”.

## AM Synthesis

The first, and technically simplest way to generate sound is via AM (amplitude modulation) synthesis.

With this technique you take the wave form created by an *oscillator<sub>g</sub>* and modulate its amplitude (volume) according to a second wave form.

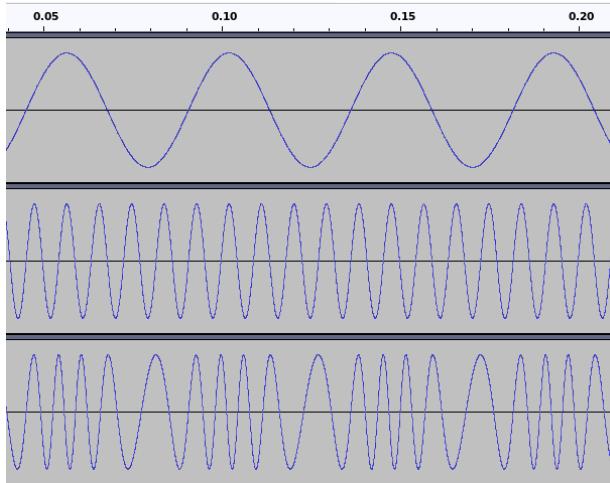


Example of AM Synthesis

In this example we see a 440Hz sine wave (in the middle) having its amplitude (quite heavily) modulated by a 110Hz sine wave (on top): the resulting wave form on the bottom has a “tremolo effect” to it.

## FM Synthesis

With this technique you take the wave form created by an *oscillator<sub>g</sub>* (called “carrier frequency”) but instead of modulating its amplitude, you modulate its frequency (pitch) according to another wave (called “modulator frequency”).



Example of FM Synthesis

In this example (for sake of visibility) we have a 110Hz sine wave (in the middle) having its frequency (again, heavily) modulated by a 22Hz sine wave (on top): we can see the result in the bottom of the figure.

This frequency modulation happens so fast that we end up with something that sounds completely different from the original wave form.

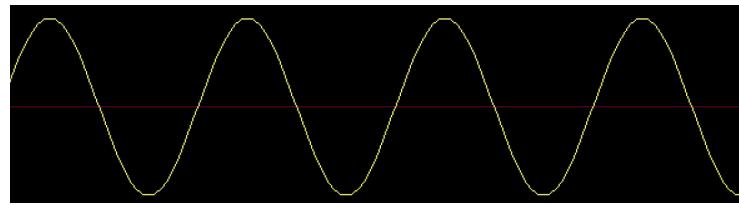
## Basic Wave Forms

It is important to know how the main wave forms look and sound in order to understand how to create your own instruments, as well as having a further insight on how older 8-bit games sounded.

It is suggested to look for each sine wave on the internet and hear how it sounds, here we will briefly talk about the main waveforms, their shape and uses.

### Sine Wave

A sine wave has an amplitude that follows a trigonometric sine wave, it sounds really “pure” and is usually used at 440Hz to tune instruments in A.

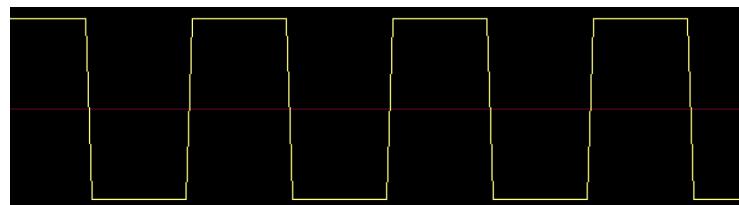


How a sine wave looks

In games it is usually used to give out the impression of a flute-like instrument.

## Square Wave

A square wave looks... square-like. It is one of the most used waves in 8-bit music, sounds a bit rougher than a sine wave and is used for beeps and blips.



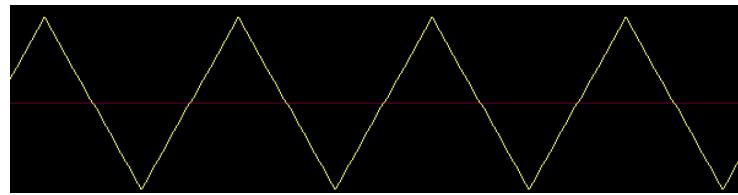
How a square wave looks

In game music it is normally used as lead instrument, and with various modulations, it can sound like a xylophone or a piano, or at least a very artificial rendition of those.

The NES had 2 voices (or channels) dedicated to square waves.

## Triangle Wave

A triangle wave is another very used wave in 8-bit music, given it's very “muted” characteristics it can be used to give songs a “bass track” of some sort.

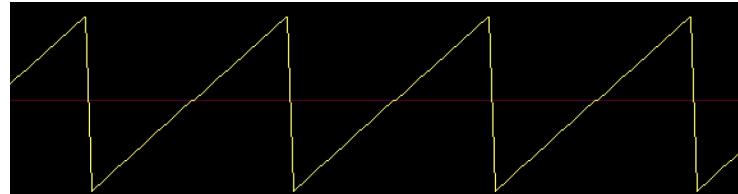


How a triangle wave looks

The NES had one channel completely dedicated to triangle waveforms.

## Sawtooth Wave

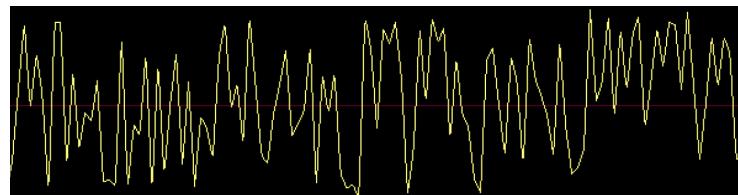
Sawtooth waves were a staple of the Commodore64 era, with its “gritty”, “bzzt” sound which can sound a lot like a trombone on long notes.



How a sawtooth wave looks

## Noise

Noise is not a real, static waveform, but more like what comes out when the amplitude has a random value on each sample.



How a noise wave looks

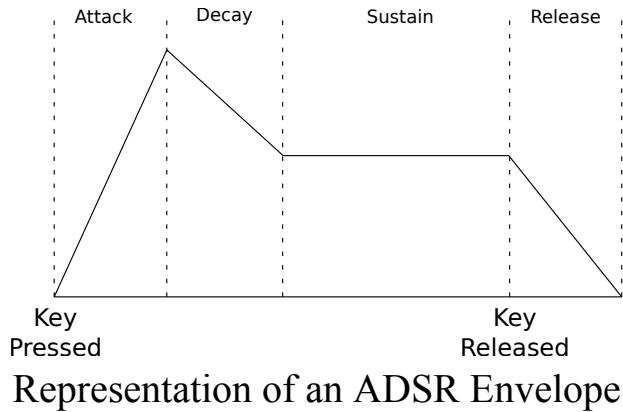
Noise can be used (with the right modulation and processing) to simulate percussions.

The NES had one channel completely dedicated to noise waveforms.

## ADSR Envelope

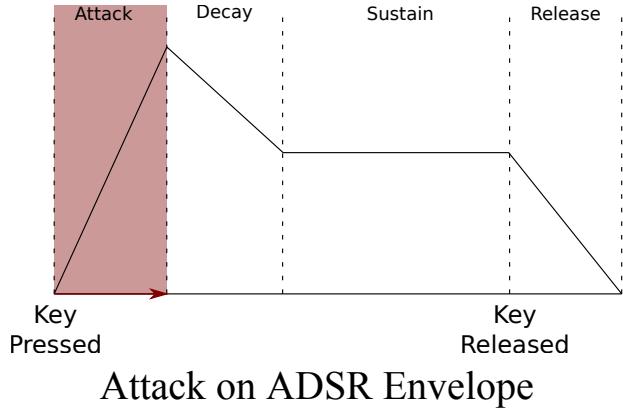
Now that we've seen how waveforms look like, we need to understand how such sounds change over time. The sound envelope is used to describe exactly that.

The most common way to control the signal envelope is through four parameters: Attack, Decay, Sustain and Release. Thus the name "ADSR envelope".



## Attack

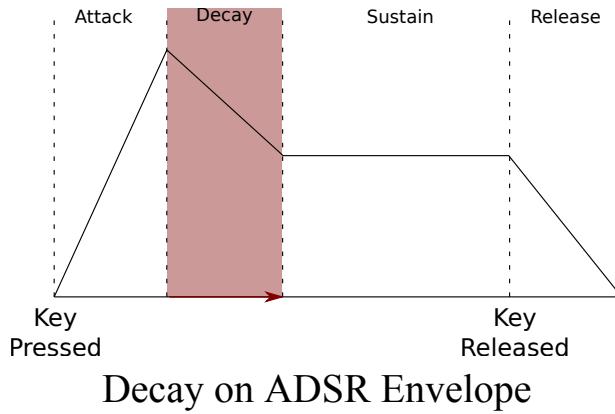
The attack is the measure of time the sound takes from zero to its initial peak, when the key is pressed.



The longer the attack, the slower the sound will "rise" when a key is pressed.

## Decay

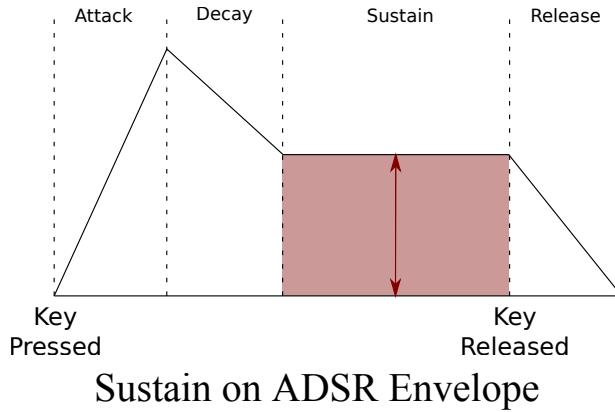
After the attack, comes the decay, which is the measure of time it takes the sound to drop to “sustain level” after the initial peak.



The longer the decay, the slower the sound will drop to sustain level.

## Sustain

After the decay is completed, we are now sustaining the signal. Sustain is *not* a measure of time, but **it is a measure of volume**.

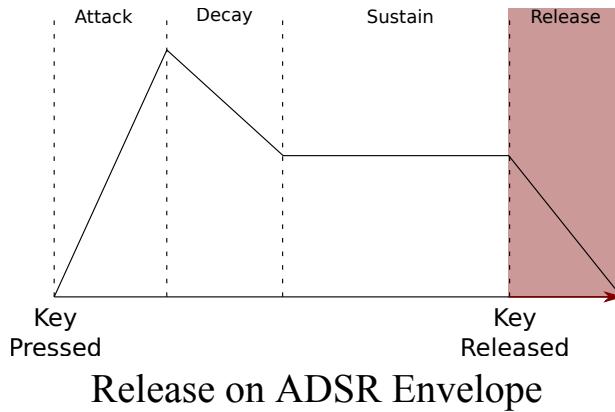


The higher the sustain level, the louder the signal will be when at “sustain level”. This signal will last until we release the key.

## Release

After we release the key, the sound will have to fade out somehow. Release is the measure of time it takes the sound to go from sustain level back to

zero.



The higher the release time, the longer the sound will take to “fade out”.

## 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 may run out of memory on low-end systems (or we can “run out” of patience in creating tens of variants of sound effects).

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.

## 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.

## 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.

## 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).

## 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).

## FM Synthesis vs Sample-based music

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

## Simulating older consoles’ audio

Sometimes we want to give our players a sensation of nostalgia, or we just want to limit ourselves to get the most out of our creativity (creativity comes from limitations), so we may decide to emulate the audio of a famous console (or at least remember of it in some way).

It's not sufficient to "make it sound" the same way (same pitch, same general sound) but we also need to adhere at the limitations of the consoles. Usually such limitations are in the number of channels (which means the number of notes that can be played at once by all instruments), but sometimes it's more structural.

### Note!

These are just simplified versions of the limitations of each console. Also if you want you can freely break any of the "rules" and make something original that has the "taste" of something "classic".

Let's take a look at some of the most famous.

## Commodore Vic20

The Commodore Vic20 is one of the first famous home computers; its audio comes from the VIC chip, the same chip that takes care of the video output.

The VIC chip has 3 channels dedicated only to square/pulse waves, with a range of 3 octaves where each octave is a single octave apart from the others (So its octave structure would be something like 1st - 3rd - 5th).

The VIC chip also features a noise channel, for a total of 4 voices. Remember that these voices are shared between music and sound effects.

## Commodore 64

Probably the most famous home computer of the 8-bit era and had an amazing sound chip for the time: the timeless SID chip, which was used for audio output and controlling paddles/joysticks.

The SID chip (in its two main iterations: the MOS 6581 and the 8580) has 3 channels, each one can be reprogrammed in real time to use one of four wave forms:

- Square/pulse wave
- Triangle
- Sawtooth
- Noise

This real-time programming capability makes it easier to give the “illusion of more instruments”, also the SID chip features ADSR controls for each channel, giving more possibilities.

### Random Trivia!

The MOS 6581 (the first SID Chip model) had a flaw in its volume register that was very special: if you could change the value of the \$D418 register fast enough, you could play audio samples with a 4-bit resolution, effectively giving the C64 a 4th channel for playing PCM samples!

This issue was fixed in the later MOS 8580 revision, but it can be “added back” by adding a resistor on the board.

## Commodore Amiga

This is another famous home computer, although some could argue that it was being produced during the fall of Commodore, it is still the cradle of sample-based music and music trackers.

The Commodore Amiga’s sound chip, name Paula, had a 4-channel PCM sample-based sound system, where each sample has an 8-bit resolution. Nothing stops people from just mixing more samples together and give the illusion of more channels.

Another limitation of the Paula chip is that 2 channels are strictly dedicated to the “left” stereo channel, while the other 2 are for the “right” stereo channel.

## **Sega Master System / GameGear**

The Sega Master System is a quite famous 8-bit console, which had moderate success, and has a lot in common (hardware-wise) with the portable Sega GameGear and those similarities extend to the sound chip too.

The sound chip used is an equivalent of the Texas Instruments SN76489 which features 3 channels dedicated to square/pulse waves + a noise channel.

### **Random Trivia!**

The ancestor of the Master System, the SEGA SG-1000 used a real TI-SN76489, while the Master System uses a “clone” integrated into its VDP (Video Display Processor).

## **Sega Genesis/MegaDrive**

Probably Sega’s most famous console: the Genesis/MegaDrive is a bit of a weird beast when it comes to sound. You’ll see why.

Mainly the console uses a Yamaha YM2612 chip for sound, which offers 6 programmable FM channels + 1 DAC (digital to analog converter) that can play small samples. The sound chip is technically stereo, but the feature is underused due to the fact that in the original console stereo sound could be heard only through the headphone jack.

In addition, mostly for Master System compatibility, the console features a TI-SN76489 chip equivalent (integrated into its VDP), adding 3 square wave channels and a noise channel.

### **Random Trivia!**

To underline how important the sound was in this console, just think that sound had its own dedicated fully-fledged CPU! It was a Zylog Z80, the same used as main CPU in the Sinclair ZX Spectrum.

## NES

Probably the most famous console in the world, the NES had a limited but interesting toolkit for its sound.

The base console had 5 channels, distributed as follows:

- 2 channels dedicated to square waves;
- 1 channel only for triangle waves;
- 1 noise channel;
- 1 channel dedicated to playing small digital sound samples (used normally for drums).

But what's most interesting is that such capabilities could be extended by cartridge hardware, the most famous is probably the Konami VRC6, which added 2 more square wave channels, as well as a sawtooth one, used in Castlevania III.

### Random Trivia!

Just to underline how extensible this system was, the Konami VRC7 contained a sound chip that provided a 6-channel FM synthesizer. Sadly its extended audio capabilities were used in a single, Japan-exclusive game.

## SNES

The SNES is a huge step forward in time for audio on Nintendo systems, featuring 8 channels that make use of 8-bit samples.

The S-SMP Chip also features a variety of filters and effects, so you have pretty much full freedom except for the number of channels.

## AdLib / SoundBlaster

The AdLib and SoundBlaster cards are based on the Yamaha YM3812 chip, which features 9 channels that use a digital oscillator. Given the high number of channels and the freedom given by them, it's pretty easy to get a result that sounds like old DOS game as soon as you get the tone down.

## “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:

```
footsteps:
    grass:
        - grasswalk1.wav
        - grasswalk2.wav
    stone:
        - stonewalk1.wav
        - stonewalk2.wav
    metal:
        - metalstep1.wav
        - 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](#)).

## Some audio processing tips

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

### Prefer cutting over boosting

Sometimes we may find our audio samples lacking that “punch” they would need, the first idea we may have would be to use a “bass boost” filter to make the low frequencies more prominent. Most of the time, this is not a good idea, since boost filters can create artifacts.

It’s better to cut the higher frequencies instead, and eventually boost the entire volume of the sample during mixing. This way the nature of the sample doesn’t get tainted by boosting, and we obtain the result we wanted.

## DAW Basics

### What is a DAW Software?

Digital Audio Workstation Software (DAW Software) are pieces of software that have extensive recording, playback and editing features, allowing you to create your own songs, given some instrument samples (or pre-recorded tracks).

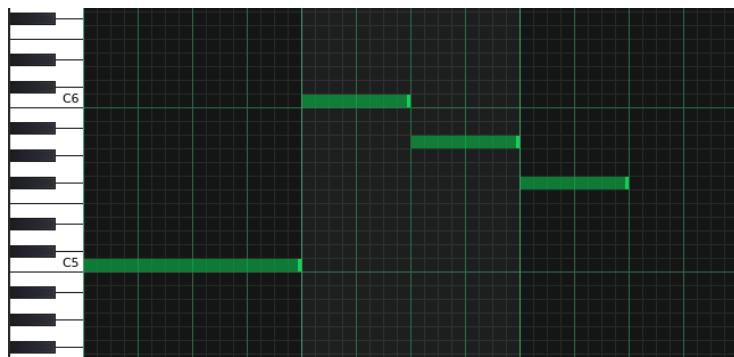
They also feature mixing facilities, waveform display and track controls, some even feature (the software equivalent of) effect racks, such as equalizers, to further the possibilities of creating your work in the best way possible.

### The Piano Roll

Have you ever seen one of those pianos that seem to have an integrated music box? That is a so-called “piano roll” and is used to store music and

play it back on the piano.

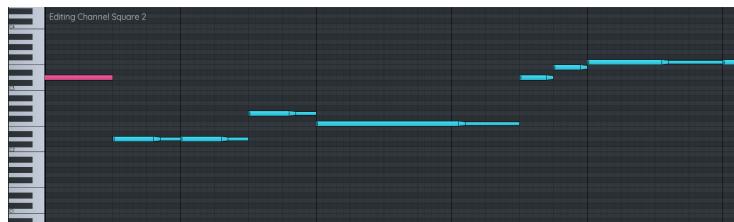
Most Software DAW have a similar abstraction, called “piano roll” too.



Example of a piano roll in LMMS

In the previous image, we can see four notes, each defined by its vertical position (respectively a C5, C6, A5, F#5) and length (respectively a single whole/semibreve and three half/minim notes). In the previous example each vertical green bar represents  $\frac{1}{4}$  and each light green bar represents a beat. This should help you imagining how to compose something.

Different systems have different piano roll variations. For instance:



Example of a piano roll in FamiStudio

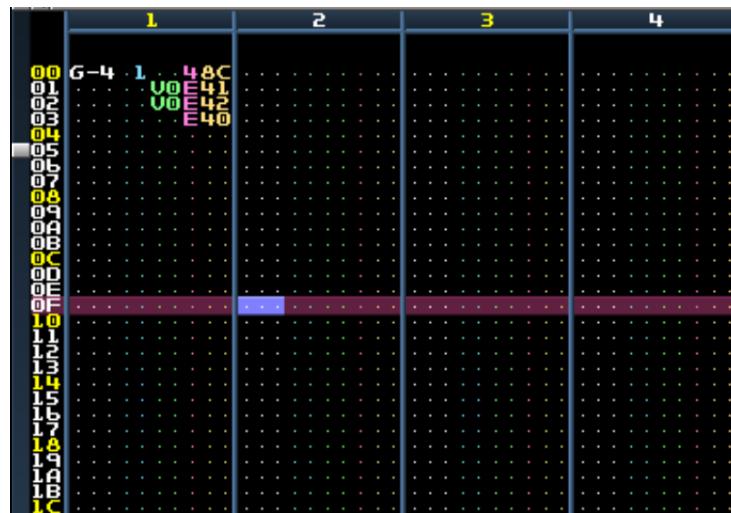
This Software DAW has a different shape for the notes, meaning a certain kind of effect is applied to them. All the basic principles still apply, though.

The piano roll abstraction allows you to edit your music easily, by grabbing notes and moving them, or making them longer by dragging their edges.

## Music Tracker Basics

### What is a Music Tracker Software?

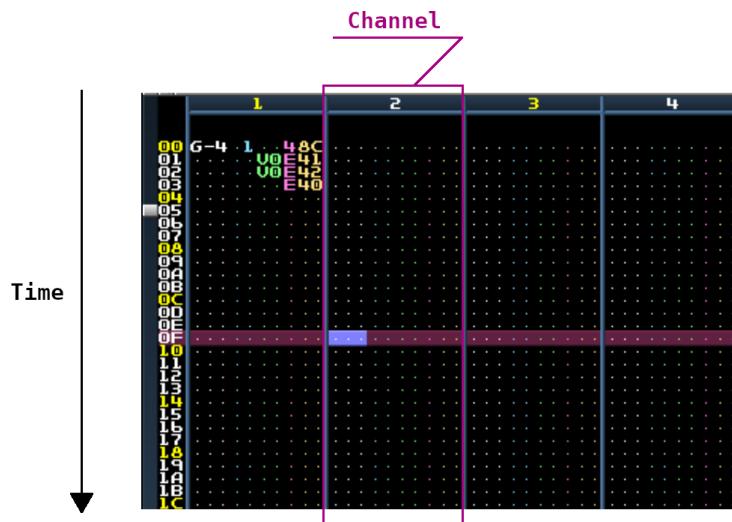
Born with the 4-voice sampling system in the Commodore Amiga, Music Trackers are essentially a type of music sequencer. The great majority of music trackers have the majority of their screen occupied by the tracker version of a music sheet.



A screen from MilkyTracker

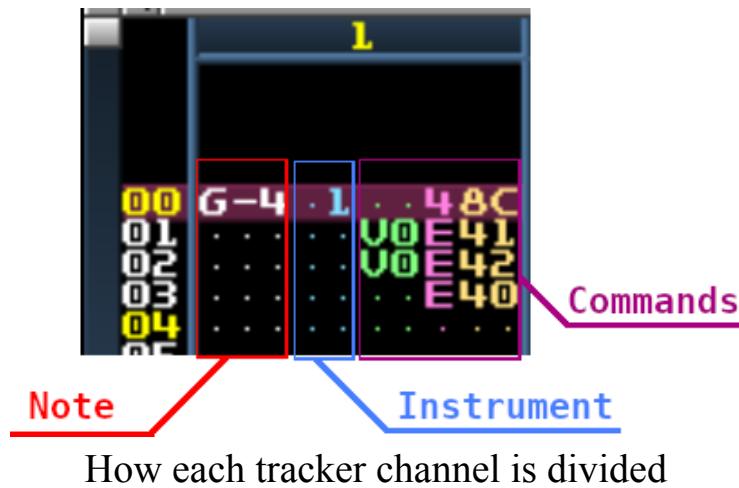
Differently from how the great majority Software DAW work, notes are positioned in channels at certain points of a timeline that spans vertically.

Usually each channel contains 64 rows (16 beats), but it can be changed to the composer's preference.



Simple overview of a tracker

Each row of a channel contains instructions for the tracker to execute, in the form of notes, instruments and commands (or effects).



Notes are written in the usual “Letter Notation” that you see in many music environments, while instruments are enumerated, then there are commands: commands can instruct the tracker to apply a temporary effect on the note, like portamento or vibrato.

In the previous example, there is a “vibrato” command going on, starting with the `48C`: 4 is the “vibrato command”, 8 is the vibrato speed and C (hex for “12”) is the vibrato depth.

The vibrato continues with a `V0 E41` command pair, where `V0` changes the vibrato depth to 0 (the speed is the same defined in the previous command), while `E41` is a “vibrato control command” (`E4`) which changes the waveform of the vibrato to 1 which is “ramp down”.

The next command does more or less the same thing, besides changing the vibrato waveform to “square”.

The `E40` command resets the vibrato waveform to the default “sine wave”.

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

## Samples

Samples are the basis of a music tracker: they are essentially wave forms which can be sped up or slowed down to create different notes. Without any sample, you wouldn't have any instrument, which in turn would mean you'd have no sound at all.

Usually samples come in the form of small digital sound files, most trackers allow the sample to be looped (wholly or in part) to simulate a “sustain” effect.

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

## Instruments

An instrument is a set of a sound sample, with some effects applied by default (if you want). Essentially an instrument is a “container” for a sample and some parameters to allow the change of pitch and effects.

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

## Channels

A “channel” (also called a “voice”), is a space where one sample is played back at a time. One channel is not “fixed” to a certain instrument and modern music trackers can mix an unlimited number of channels. Many times music makers limit themselves to a certain number of channels to achieve a “retro feeling” or to challenge themselves.

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

## Patterns

A pattern is essentially a piece of a song: a group of tracks with their own instruments, settings and notes written in them. The “pattern” abstraction allows you to easily repeat pieces of a song by just referring to the pattern.

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

## Basic Rhythms

When composing music, we may not know where to start: this is the objective of this section: give you some easy rhythms to start with. Here I will use LMMS's beat+bassline editor to represent the notes, as it's the easiest to understand.

Remember to check your tempo too, since it may make the difference between something akin to the “house” genre and something more “techno”.

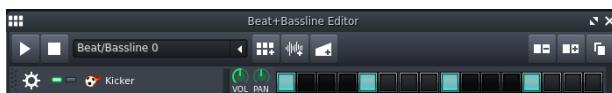
### Four on the floor

This is the most basic rhythm there is: let's consider a situation where we have 16 beats per bar (so each note is 1/16th):



A single bar in our basic rhythm

The four on the floor rhythm uses a kick drum in the 1st, 5th, 9th and 13th beat, giving a constant rhythm.



A basic four on the floor rhythm

This is practically the basic of all dance-based music.

### Four on the floor with off-beat hi-hats

A basic rhythm like the “four on the floor”, by itself, can prove to be quite boring. To spice it up, we can add some closed hi-hats in the off-beats, like the following:



Four on the floor with off-beat hi-hats

If you listen closely, a lot of music has this basic pattern mixed into it, in one way or another.

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

## Adaptive Music

Adaptive Music (sometimes called “dynamic music” or “interactive music”) is a background music track that reacts to the events of the game. Such reactions can involve volume, rhythm, tune, adding new instruments (adding drums, for instance).

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

## Fonts

### 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.

#### 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

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

Example of a sans-serif font (DejaVu Sans)

A famous sans-serif font is Arial.

## 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.

## Proportional

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.



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.

## Using textures to make text

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

## Using Fonts to make text

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

## Shaders

### 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.

## 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**.

```
#ifdef GL_ES
    precision lowp float;
#endif

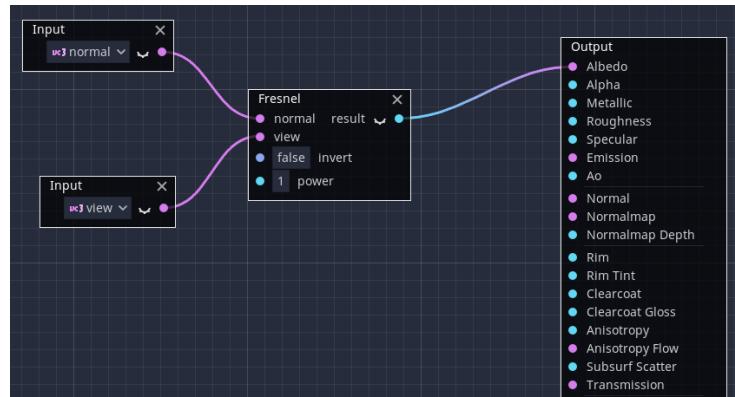
uniform float u_time;

void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

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.



Godot’s “Visual Shader” Editor

## The GLSL Programming Language

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

### The data types

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

### Some GLSL Shaders examples

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

# Part 5: Advanced Topics

# Design Patterns

Patterns mean “I have run out of language.”

---

Rich Hickey

Design Patterns are essentially “pre-made solutions for known problems” and can help decoupling elements of your game, increasing maintainability.

## Note!

This book will introduce design patterns as they were explained in the famous “Gang of Four” book: “Design Patterns: Elements of Reusable Object-Oriented Software”. In some languages (for instance ones that have functions as first-class objects) such patterns can take different forms. This book will tell you which pattern does what, along with a rough implementation, but it’s up to you to check the most efficient one in your favourite language.

## Pitfall Warning!

When people find out about design patterns, they usually come up with the “everything is a nail syndrome”. Don’t overuse design patterns only because it looks “cool” or “because it may solve a future problem”. What matters is your game, right now.

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).

Leave future problems to your future self.

# 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.



The UML diagram for a singleton pattern

```
class Singleton {  
  
    static Singleton INSTANCE = new Singleton();  
  
    // This makes the constructor impossible to  
    use outside of the class  
    constructor() {}  
  
    @staticmethod  
    function getInstance() -> Singleton{  
        return INSTANCE;  
    }  
}
```

Code: Example of a singleton pattern

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.

```
class LazySingleton {  
  
    static LazySingleton instance = null;  
  
    // Block the constructor from working, we  
    don't want to build more than one instance  
    constructor() {}  
  
    @staticmethod  
    function getInstance() -> LazySingleton{  
        // Multi-threading: manage race conditions  
        // ----- Critical region start -----  
        if (instance == null) {  
            instance = new LazySingleton();  
        }  
        // ----- Critical region end -----  
  
        return instance;  
    }  
}
```

Code: Example of a singleton pattern with lazy loading

If multiple threads are involved in using a lazy-loading singleton, you may need to take care of preventing *race conditions*<sup>g</sup> 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?

Summary table for the Singleton Pattern

<b>Pattern Name</b>	Singleton
<b>When to Use it</b>	In all situations that strictly require one instance of an object, accessible globally.
<b>Advantages</b>	Allows the class to control its own instantiation, allows for easier access to the sole instance of a class.
<b>Disadvantages</b>	Introduces some restrictions that may be unnecessary, introduces a global state into the application.

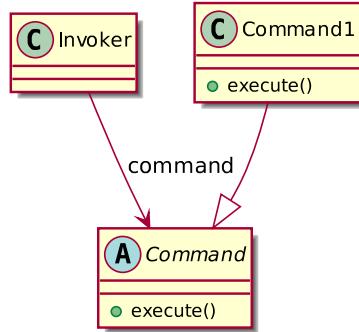
## Command Pattern

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.



UML diagram for the Command Pattern

```

abstract class Command{
    // This is the abstract class that will be
used as interface
    function execute();
}

class JumpCommand inherits from Command{
    // This will implement the execute method
    function execute(){
        jump();
    }

    function jump(){
        // DO STUFF
    }
}

```

Code: Example code for the Command Pattern

Summary table for the Command Pattern

<b>Pattern Name</b>	Command
<b>When to Use it</b>	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.
<b>Advantages</b>	Allows for encapsulation, less coupling, more flexibility and customization at runtime.
<b>Disadvantages</b>	Late binding and objects may introduce some

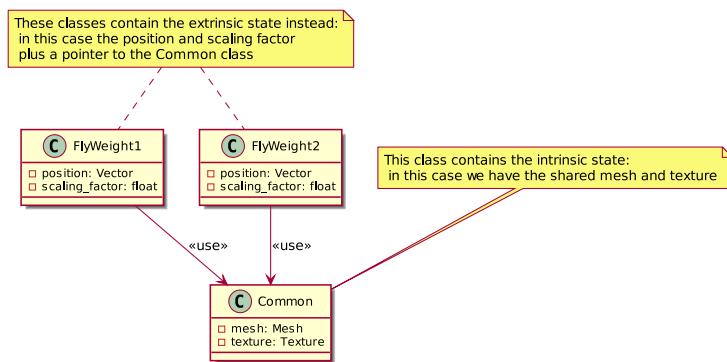
overhead.

## 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”.



UML Diagram of the Flyweight pattern

Below is an example code for the flyweight pattern.

```
class Common{
    // Contains the common data for a 3D Object to
    be replicated
    Mesh mesh;
    Texture texture;
}

class FlyWeight{
    // Contains only the necessary data to create
    an instance of the item
```

```
    Common common_pointer;
    Vector position;
    float scale_factor;
}
```

Code: Code for a flyweight pattern

### Random Trivia!

This is just speculation, but SFML's graphics system may make use of the Flyweight pattern: you need to load the image into a “Texture” first (which does all the low-level lifting) and then you can instance an “Image” class which is more high-level. Many images can refer to the same texture (which may be a Sprite Sheet).

Summary table for the Flyweight Pattern

<b>Pattern Name</b>	Flyweight
<b>When to Use it</b>	When you need to support a large number of similar objects efficiently, when you need to avoid creating a large number of objects.
<b>Advantages</b>	Saves memory when a large number of similar objects is involved, avoids some of the overhead given by the creation of many objects.
<b>Disadvantages</b>	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.

## Observer Pattern

The observer pattern is used to implement custom event handling systems, where an object automatically reacts to events generated by another object.

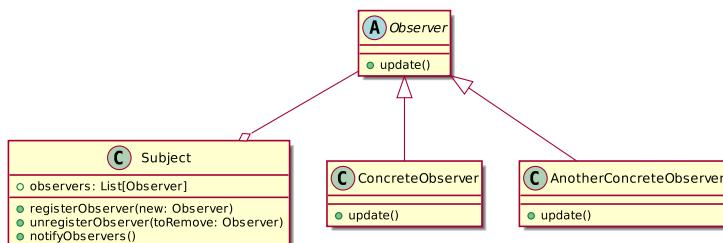
There are 2 main objects used in an observer pattern:

- **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:



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:

```
class Subject{
    /* This is the observed class that contains
    the list of observers and
     * the notifyObservers method */

    Observer[] observers = new list of Observer;
```

```

        function register_observer(observer) {
            observers.append(observer);
        }

        function notifyObservers() {
            for (each observer in observers) {
                observer.update();
            }
        }
    }

    class Observer{
        /* This is the class that contains the update
method, used to force
         * an update in the observer */

        function update() {
            print("I have been updated!");
        }
    }

    subject = Subject();
    observer = Observer();
    subject.register_observer(observer);
    subject.notifyObservers();

```

Code: Code for an observer pattern

If needed, you can pass information between the subject and the observers just by calling each `update()` method with the necessary arguments.

Summary table for the Observer Pattern

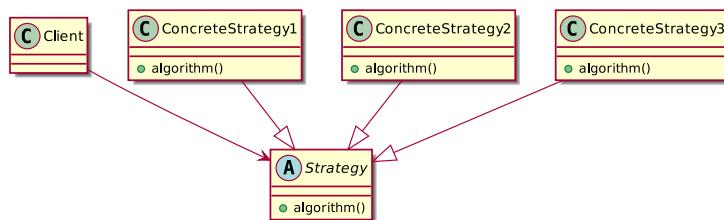
<b>Pattern Name</b>	Observer
<b>When to Use it</b>	Event Handling systems, making objects react to other objects' actions
<b>Advantages</b>	Decoupling, added flexibility, more performant than if statements for conditions that happen rarely.
<b>Disadvantages</b>	Can be a bit hard to set up, makes the architecture more

complex, if un-registration is not done well there could be serious memory leaks (even in garbage-collected languages).

## 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.



The UML diagram of the strategy pattern

```
abstract class Strategy{
    // This class defines the strategy interface
    // the client will refer to

    function algorithm(){
        // This algorithm will be implemented by
        // the subclasses
    }
}

class ConcreteStrategy1 inherits from Strategy{
    function algorithm() override{
        // Real implementation of the algorithm
        // DO STUFF
    }
}

class ConcreteStrategy2 inherits from Strategy{
    function algorithm() override{
        // Real implementation of the algorithm
    }
}
```

```

        // DO STUFF SLIGHTLY DIFFERENTLY
    }
}

// Example Usage
function main() {
    Strategy to_execute = null;
    if (condition) {
        to_execute = new ConcreteStrategy1();
    }else{
        to_execute = new ConcreteStrategy2();
    }
    to_execute.algorithm(); // This will execute
1 or 2 depending on "condition"
}

```

**Code:** Code for a strategy pattern

Summary table for the Strategy Pattern

**Pattern Name** Strategy

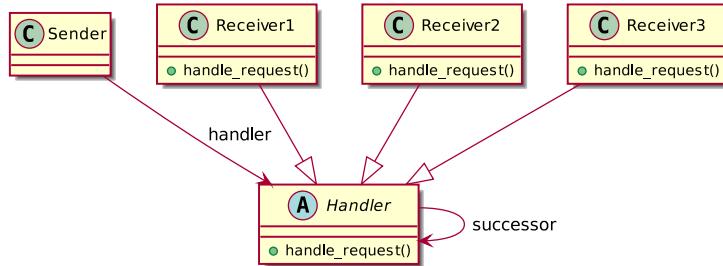
**When to Use it** Every time you need to decide which algorithm to execute at runtime.

**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.

## 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.



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.

```

abstract class Handler{
    // This is the handler abstract/class
    // interface that the sender connects to
    Handler next; // The next handler in the
    chain

    function handle_request(){
        if (condition){
            // In case I can handle this request
            real_handler();
        }

        if (next is not null){
            next.handle_request();
        }
    }

    function real_handler(){
        // This function gets implemented in the
        concrete classes
    }

    function add_handler(Handler new_handler) ->
    Handler{
        next = new_handler;
        return next; // Allows for chaining
    }
}

```

```
.add_handler().add_handler()...
    }
}
```

Code: Code for a chain of responsibility pattern

Summary table for the Chain of Responsibility Pattern

<b>Pattern Name</b>	Chain of Responsibility
<b>When to Use it</b>	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.
<b>Advantages</b>	Decoupling, added flexibility.
<b>Disadvantages</b>	Some overhead is added by the objects and late binding, could lead to proliferation of similar-looking handlers/receivers.

## 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.

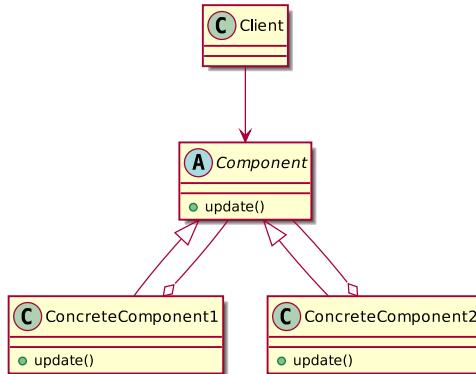


Diagram of the Component Design Pattern

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:

```
abstract class Component{
    // Defines the abstract class/interface for
    the component pattern
    function update(){
        // Do nothing, this is an abstract class
    }
}
```

```

        class ConcreteComponent1 inherits from Component{
            // Defines the concrete component number 1
            function update() {
                // Do Stuff
            }
        }

        class ConcreteComponent2 inherits from Component{
            // Defines the concrete component number 2

            // The component can contain a list of other
            components that get updated
            Component[] list = new vector of 9 Component;

            function update() {
                for (each component in list) {
                    component.update();
                }
            }

            // Do Other Stuff
        }
    }

    class Client{
        ConcreteComponent1 first_component;
        ConcreteComponent2 second_component;

        function update() {
            // This is the Client's update function
            first_component.update();
            second_component.update();
        }
    }
}

```

## Code: Example Implementation Of the Component Pattern

Summary table for the Component/Composite design pattern

**Pattern Name** Component/Composite

**When to Use it** When you need to deal with a part-whole hierarchy where each component needs to be treated equally.

**Advantages** Decoupling, added flexibility.

<b>Disadvantages</b>	On bigger systems, the management may become really complex.
----------------------	--

## Dependency Injection

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

## Decorator

There are some cases where we need to add or remove behaviours from a class at runtime, dynamically. The decorator pattern gives a flexible alternative to subclassing and addresses this need.

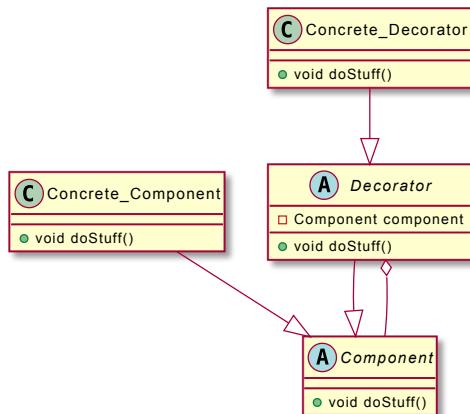


Diagram of the Decorator Pattern

As you can see the decorator makes heavy use of abstract classes and interfaces, which most programming languages implement without any issue.

Summary table for the Decorator design pattern

<b>Pattern Name</b>	Decorator
<b>When to Use it</b>	When you need to add or remove functionalities from a class dynamically. Very useful for applying <a href="#">memoization techniques</a> .

<b>Advantages</b>	Decoupling, added flexibility.
<b>Disadvantages</b>	Usage of interfaces or abstract classes can seem a bit daunting at the beginning, it may cause an explosion in number of classes.

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

## Visitor

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

## Adapter

Let's face it, not everything is straight out compatible with everything else. It happens with power plugs, why wouldn't it happen in a world as varied as software development?

Sometimes we need an adapter, and that's exactly what this design pattern is: provide a layer of compatibility between two incompatible interfaces.

The adapter design pattern can be implemented in two ways, but first let's check the summary table.

Summary table for the Adapter design pattern

<b>Pattern Name</b>	Adapter
<b>When to Use it</b>	When you need to provide a layer of compatibility between two incompatible interfaces or you need to provide an “alternative interface” to a class.
<b>Advantages</b>	Decoupling, added flexibility, better compatibility, code reuse.

**Disadvantages** Needing many adapters may mean there is a deeper structural problem with your program.

## Object Adapter

The “object adapter” is the version where the adaptor delegates the task to the adaptee at runtime. To do so, it has an instance of the adaptee class as its class field.

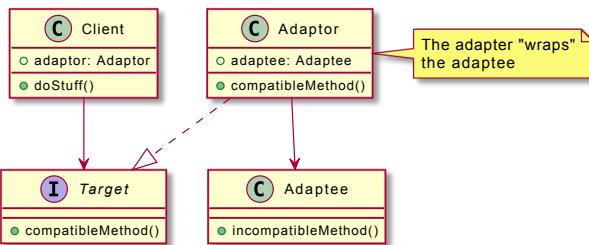


Diagram of the Object Adapter Pattern

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

## Class Adapter

The “class adapter” version instead inherits the adaptee class at compile-time. Since the adaptor inherits from the adaptee class, the adaptee’s methods can be called directly, without needing to refer to a class field.

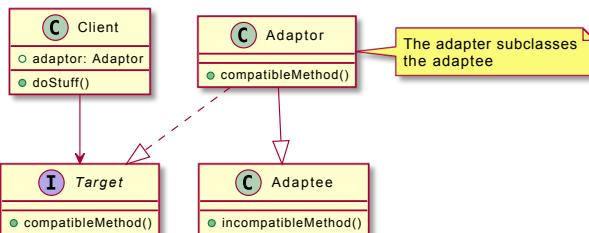


Diagram of the Class Adapter Pattern

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

# Prototype

Sometimes, in our game, we need to decide which objects to create at runtime, as well as instantiate dynamically loaded classes. In these cases the prototype pattern comes to the rescue: we define a “prototype” that allows to create classes by cloning itself.

There is the UML diagram for the pattern:

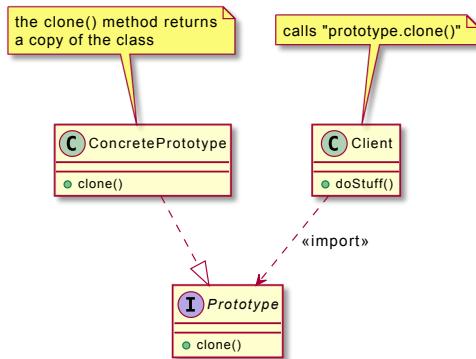


Diagram of the Prototype Pattern  
Summary table for the Prototype design pattern

**Pattern Name**

Prototype

**When to Use it**

When you need to either decide the objects to create at runtime or instantiate dynamically loaded classes.

**Advantages**

Decoupling, added flexibility.

**Disadvantages**

May become overused, depending on the situation can be difficult to implement.

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

# Facade

There are times where you have a very complex library, with a very complex interface, that is extremely complex to interact with. The Facade

pattern hides such complexity behind a simple-to-use interface that works by delegation.

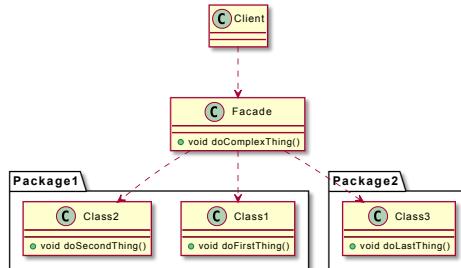


Diagram of the Facade Pattern

This pattern should be used with extreme care and only when necessary, since adding “levels of indirection” will make the code more complex and harder to maintain.

Summary table for the Facade design pattern

<b>Pattern Name</b>	Facade
<b>When to Use it</b>	When you need to present a simple interface for a complex system or you want to reduce the dependencies on a subsystem.
<b>Advantages</b>	Decoupling, added readability.
<b>Disadvantages</b>	May become overused, delegating adds a bit of overhead, sometimes it may be wrongly used where either an adapter or a decorator is needed.

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

## Service Locator

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

# Useful 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 data containers and classes that could help you solve some issues or increase your game's maintainability and flexibility.

## 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]*

## Animator

This can be a really useful component to encapsulate everything that concerns animation into a simple and reusable package.

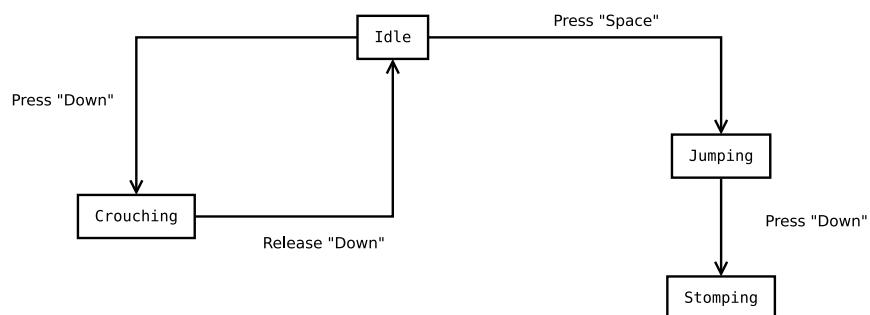
The animation component will just be updated (like the other components) and it will automatically update the frame of animation according to an internal timer, usually by updating the coordinates of the rectangle that defines which piece of a sprite sheet is drawn.

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

## 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:



## 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:

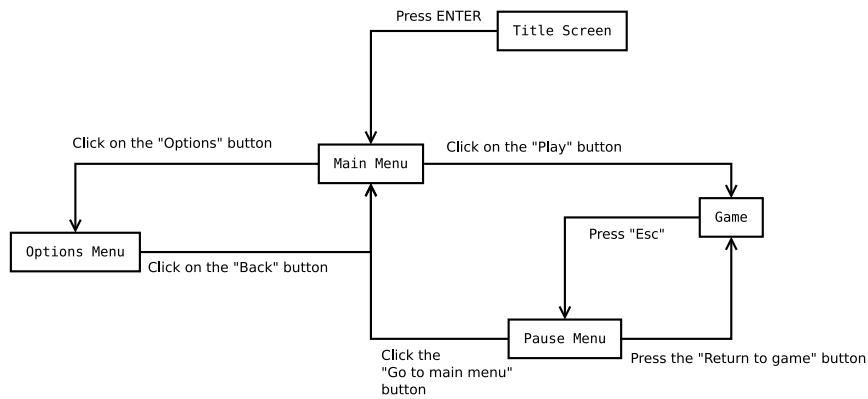


Diagram of a menu system’s state machine

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]*

## 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]*

## Particle Systems

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

### Particles

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

### Generators

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

## Emitters

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

## Updaters

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

## Force Application

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

## Timers

Timers are an essential component in many game mechanics: [Coyote Time](#) and [Jump Buffering](#) are two prime examples of timer-based mechanics.

If we want to execute a function every few seconds we need timers.

If we need to cap how many bullets we can shoot from a weapon, guess what? Timers.

Making a timer is not as complicated as it may seem, we need:

- A variable that keeps track of the time passed;
- A variable that keeps track of how much time needs to pass before the function gets executed;
- A pointer to said function;
- A boolean to track whether the timer is active or not;
- A boolean to decide whether the timer should be “one shot” or “continuous”.

```
class Timer{  
    /*  
     * This is a simple timer class that executes a
```

```

function after
    * a certain amount of time
    */
float time;
float set_time;
function_ptr function_to_execute;
bool one_shot;
bool active;

constructor(float time, function_ptr function,
bool one_shot = False, bool active=False){
    // We prepare the timer and memorize the
setting
    this.time = time;
    this.set_time = time;
    // The function pointer should already be
prepared with the arguments
    this.function_to_execute = function;
    // Is this timer one-shot then disable?
    this.one_shot = one_shot;
    // Does this timer need to be active when
constructed?
    this.active = active;
}

function update(float dt){
    if (not this.active){
        // We return directly if the timer is
disabled
        return;
    }
    // Like any other entity, we update it
    this.time = this.time - dt;
    // When the timer "ticks", we execute the
function
    if (this.time <= 0){
        this.function_to_execute();
        if (this.one_shot){
            // If this timer is a one-shot, we
disable it
            this.active = False;
        }
        // We reset the timer (we may need to
re-activate it manually later)
        this.time = this.set_time;
    }
}

```

```
}
```

Code: A simple timer class

## Accounting for “leftover time”

This timer is a nice and simple solution, but it has a small flaw: when the timer is set to execute continuously and the function is executed, it doesn't account for “leftover time”. This may be easier to understand with an example.

Let's imagine that we have a timer that shoots a bullet every quarter of a second (250ms), our game is running at a steady 30fps (which means each frame takes 33.33ms), our timer's internal counter will behave like this:

- **Frame 1:**  $250 - 33.33 = 216.67$
- **Frame 2:**  $216.67 - 33.33 = 183.34$
- ...
- **Frame 7:**  $50.20 - 33.33 = 16.87$
- **Frame 8:**  $16.87 - 33.33 = -16.46$  (Trigger the function and reset the timer)
- **Frame 9:**  $250 - 33.33 = 216.67$

Our timer doesn't account for the over 16ms leftover that we had between frame 8 and frame 9, thus our timer will be imprecise. This may seem an easy fix at first glance, but it is not.

## A naive solution

The first solution that would come to mind would be substituting the timer variable assignment with a sum, thus frames 7,8 and 9 would look like this:

- **Frame 7:**  $50.20 - 33.33 = 16.87$
- **Frame 8:**  $16.87 - 33.33 = -16.46$  (Trigger the function and reset the timer, by adding 250)
- **Frame 9:**  $233.54 - 33.33 = 200.21$

Here is the code:

```
class Timer{
    /*
     * ...
     * This is the same as the older version
     * ...
     */

    function update(float dt) {
        if (not this.active){
            // We return directly if the timer is
disabled
            return;
        }
        // Like any other entity, we update it
this.time = this.time - dt;
        // When the timer "ticks", we execute the
function
        if (this.time <= 0){
            this.function_to_execute();
            if (this.one_shot){
                // If this timer is a one-shot, we
disable it
                this.active = False;
            }
            // We reset the timer differently, by
adding the "set time"
            this.time = this.time + this.set_time;
        }
    }
}
```

Code: A naive approach to account for leftover time

But what happens if we have a sudden lag spike, longer than the timer itself?

On **Frame 7** we have  $50.20 - 500 = -449.8$ , due to a Lag spike: last frame took a lot longer to process, we have to execute our function and reset the timer, adding 250.

On **Frame 8** we have  $-199.8 - 33.33 = -233.13$ : The timer is trying to catch up to the lag spike, since the trigger condition is still happening, we execute

the function again and add 250 to the timer.

On **Frame 9** we have  $16.87 - 33.33 = -16.46$  We've almost caught up with the lag spike, but we need to execute the function a third time and add 250 to our timer.

**Frame 10** behaves normally with  $233.54 - 33.33 = 200.21$ .

Due to the lag spike the function gets executed three times, which is the technically correct way to “catch up” with the number of times the timer *should have* triggered. But this may be an undesirable side effect.

If our game is already slowing down, executing even more functions won't help, so a better approach would definitely be avoiding calling functions more than we strictly need.

## A different approach

To avoid this “catching up”, there are many ways, I'm going to write two of them in this book. The first is quite simple: we add the set timer in a loop until we reach a value higher than zero.

```
class Timer{
    /*
     * ...
     * This is the same as the older version
     * ...
     */

    function update(float dt){
        if (not this.active){
            // We return directly if the timer is
disabled
            return;
        }
        // Like any other entity, we update it
        this.time = this.time - dt;
        // When the timer "ticks", we execute the
function
        if (this.time <= 0){
            this.function_to_execute();
            if (this.one_shot){
```

```

        // If this timer is a one-shot, we
        disable it
                this.active = False;
            }
            // We reset the timer differently, by
            adding the "set time" until we have a positive value
            while (this.time <= 0){
                this.time = this.time +
this.set_time;
            }
        }
    }
}

```

### Code: A possible solution to account for leftover time

This approach has a very minor issue: we are using a loop, so the further we stray away from zero, the more times we will have to add. A second approach would be calculating a “multiplier” and directly apply that to the added value, thus avoiding a loop.

```

class Timer{
    /*
     * ...
     * This is the same as the older version
     * ...
     */

    function update(float dt){
        if (not this.active){
            // We return directly if the timer is
disabled
            return;
        }
        // Like any other entity, we update it
        this.time = this.time - dt;
        // When the timer "ticks", we execute the
function
        if (this.time <= 0){
            this.function_to_execute();
            if (this.one_shot){
                // If this timer is a one-shot, we
disable it
                this.active = False;
            }
        }
    }
}

```

```

        // We reset the timer differently, by
        // adding the "set time" with a multiplier
        // this.time is guaranteed to be
        // negative or zero, by dividing by a negative number
        // we have a positive multiplier
        int multiplier = ceil(this.time / -
this.set_time);
        this.time = this.time + (multiplier *
this.set_time);
    }
}

```

Code: Another possible solution to account for leftover time

This second approach has an issue too: we will need to calculate the ceiling of a value, which may require a bit more CPU time (although most modern CPUs don't require more than a single cycle to do so).

Both approaches are valid and for longer timers even the “naive” approach is valid and fast. The choice is up to your personal taste and sensibility.

## 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<sub>g</sub>* 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:

```

function linearTween(float time, float begin, float change,
float duration) -> float{
    return change * (time / duration) + begin;
}

```

```
}
```

## Code: Linear Tweening

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:

$$(change \text{ in property}) \cdot (factor) + (beginning \text{ value})$$

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:

```
// We calculate the tweening factor
    float factor = linearTween(time, 0, 1, duration);
    // Now we change the property we're tweening
    object.property = property_original_value +
(destination_value - property_original_value) * factor;
```

## Code: Example of a simple easing function

With linear tweening, the function degenerates to  $\frac{\text{time}}{\text{duration}}$ , but now we can replace our linear tween with the following function:

```
function easeIn(float time, float duration, float power) ->
float{
    return (time/duration) ^ power;
}
```

### Code: Ease-in

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:

```
function easeOut(float time, float duration, float power) ->
float{
    return 1 - (1 - (time / duration)) ^ power;
}
```

### Code: Ease-out

With some calculations, and if statements on the time passed, you can combine the two and get an “ease-in-out” function.

```
function easeInOut(float time, float duration, float power) ->
float{
    float threshold = duration / 2;
    if (time <= threshold){
        return easeIn(time, duration, power);
    }else{
        return easeOut(time, duration, power);
    }
}
```

### Code: Ease-in-out

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)`.

```
function clamp(float value, float min, float max) -> float{
    // Clamps "value" so it is always between "min"
    and "max"
    if (value < min) {
        return min;
    }
    if (value > max) {
        return max;
    }
    return value;
}
```

### Code: Clamping values

In that case, calling the clamping function with values 0 and 1 would solve the issue.

A think that many people tend to forget, but that is really important is that you can tween any property of any entity or object in your game, for instance:

- The position of a UI item;
- The width of a health bar;
- The rotation of an on-screen map or compass;
- The colour of the sky while doing a day-to-night transition.

In short: any numeric value that can transition “smoothly” between two values in a certain amount of time can be tweened.

## Bouncing

Easing is a great way to make more “natural looking” transitions, but in nature nothing is absolutely precise: we may want to overshoot a bit and correct it so it looks a bit like a “bounce”.

The idea behind this is the following: when around 80% of the time has passed, our property should have overshot its maximum value by something between 10% and 20% (more would look unnatural). The remaining 20% of the time will be used to “ease back” to the final value. This will give us the bouncy feeling we want.

Here is a quick example of a function that will help us create a bounce effect: the `t` variable represents the time of the animation that has passed, with `0` being the beginning and `1` being the end of the animation itself.

```
function bounce_tween(float t) -> float{
    // This constant will allow us to overshoot the
    max value by around 10%
    const float c = 1.70158;

    return 1 + (c + 1) * (t - 1)^3 + c * (t - 1)^2;
}
```

Code: A bouncy tween function

## Chaining

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

# 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.

## 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.

## 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.

### 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).

```

class 2D_Grid(){
    // Represents a 2D grid of "Tile" classes
    Tile[][] grid = null;
    int width = 0;
    int height = 0;

    function constructor(int rows, int cols){
        // Prepares the memory for the grid
        grid = new Tile[rows][cols];
        height = rows;
        width = cols;
    }

    function getCell(int row, int col) -> Tile{
        // Gets a cell from the 2D Grid
        if (row >= 0 AND row < height) AND (col >=0
AND col < width){
            // We better check if we are inside the
grid
            return grid[row] [col];
        }else{
            return null;
        }
    }

    function getAdjacentCells(int row, int col) ->
Tile[]{
        /* 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
        "height - 1", same goes for columns */
        Tile[] toReturn = new Tile[] // We assume
arrays can resize when necessary
        if (row >= 0 AND row < height) AND (col >=0
AND col < width){
            // We better check if we are inside the
grid
            if (row > 0){
                // We are not on the first row, we
can add the cell above
                toReturn.append(getCell(row - 1,
col));
            }
            if (row < height - 1){
                // We are not on the last row, we
can add the cell below
                toReturn.append(getCell(row + 1,

```

```

        col));
    }
    if (col > 0){
        // We are not on the first column,
we can add the cell on the left
        toReturn.append(getCell(row, col -
1));
    }
    if (col < width - 1){
        // We are not on the last column,
we can add the cell on the right
        toReturn.append(getCell(row, col +
1));
    }
}
/* If the checks went well, toReturn will
have
a list of the adjacent cells, if not it
will be empty */
return toReturn;
}
}

```

### Code: Possible representation of a 2D grid

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.

## 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.

### Adjacency Lists

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

## Adjacency Matrices

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

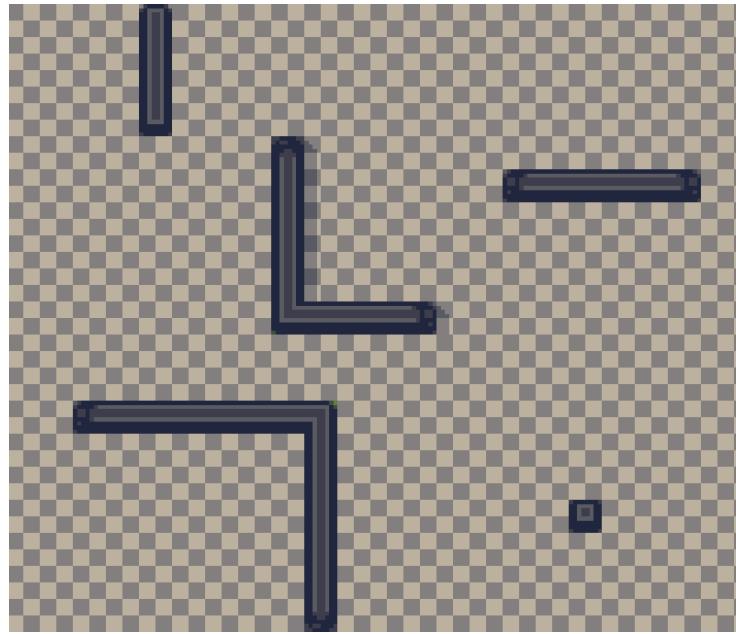
## Navigation meshes

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.

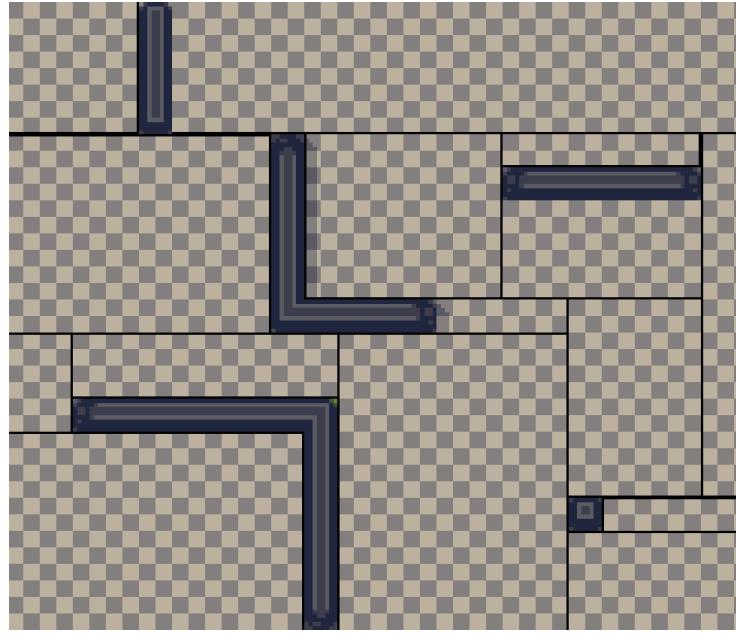
This abstraction allows to use A\* and Dijkstra algorithms, but instead of trying to navigate a graph, you look for a path between meshes (which are saved in a graph structure).

To make the abstraction easier to understand, let's take a look at the following map.



Map we will create a navigation mesh on [1](#)

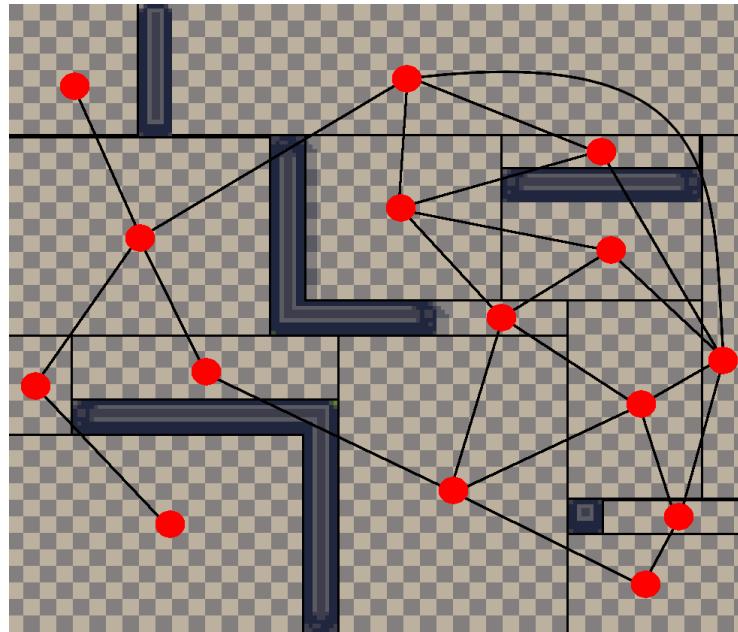
The first step is to divide the map into convex polygons, in our case we will use rectangles.



Dividing the map into many convex polygons [2](#)

Now each rectangle is a node of our graph (and will have to be treated accordingly by the AI, knowing it can navigate freely inside each rectangle),

now we need to connect each node, following the limitations given by the walls.



Creating the graph [3](#)

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

## 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.

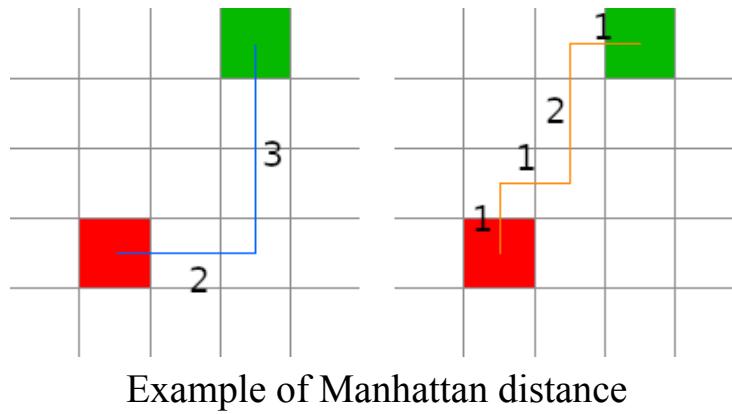
In this book we will present the most common heuristics used in game development.

### 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:



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”.

```
struct Tile{
    integer x;
    integer y;
}

function manhattan_distance(Tile start, Tile goal)
-> int{
    return abs(start.x - goal.x) + abs(start.y - goal.y);
}
```

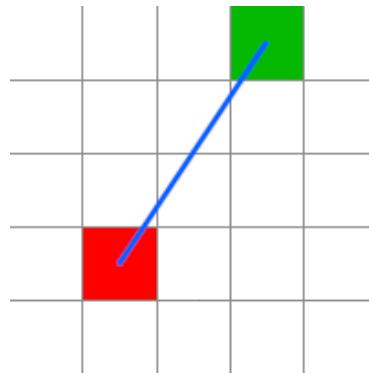
Code: Example code calculating the Manhattan distance on a 2D grid

This works well with 2D grid-based worlds.

## 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:

$$h(x) = \sqrt{(start.x - end.x)^2 + (start.y - end.y)^2}$$



Example of Euclidean Distance

```
struct Tile{
    integer x;
    integer y;
}

function euclidean_distance(Tile start, Tile goal)
-> float{
    return square_root((start.x - goal.x)^2 +
(start.y - goal.y)^2);
}
```

Code: Example code calculating the Euclidean distance on a 2D grid

## 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:

	0	1	2	3	4
0					
1					
2					
3					
4					

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.

	0	1	2	3	4
0	1	1	2	3	
1				4	
2	3	2	3		5
3	4	4	5	6	
4	5	4	5	6	7

Manhattan Distance

	0	1	2	3	4
0	1	1	2	3	
1					3.16
2	2.24	2	2.24		3.61
3	3.16	3.16	3.16	3.61	4.24
4	4.12	4	4.12	4.47	5

Euclidean Distance

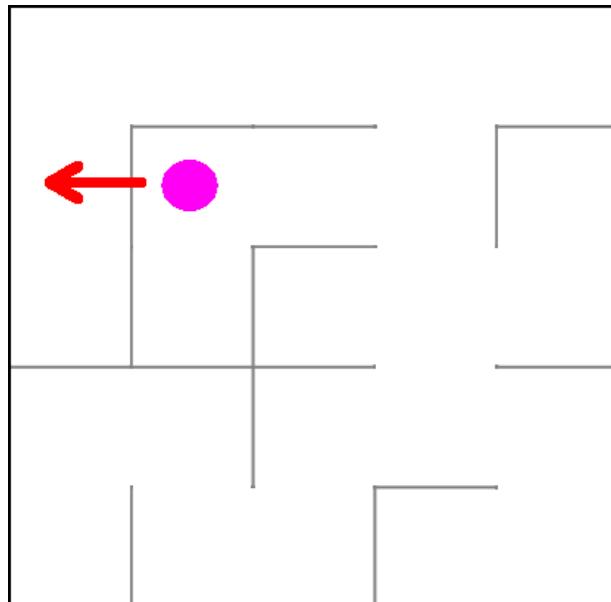
Pathfinding Algorithms Heuristics Reference Image

## A simple “Wandering” Algorithm

This is not a real “pathfinding” algorithm, as much as something that should give the impression of people “wandering around”. This algorithm is really simple and can be summarized in 2 rules:

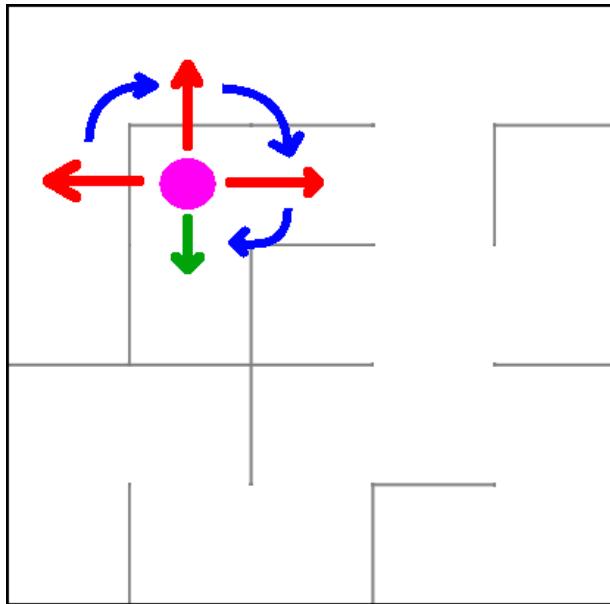
1. Choose one direction at random, if you can't go that way continue your search clockwise;
2. You cannot go back, unless it's the only direction available.

This is good when used on mazes, let's try an example:



Simple wandering algorithm 1/2

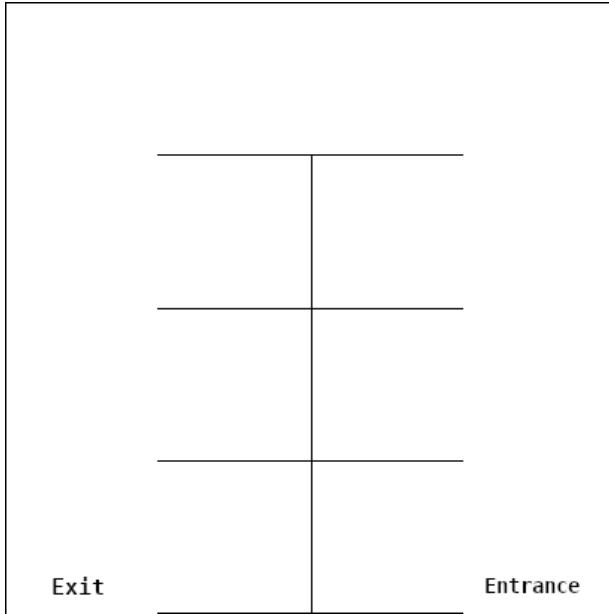
In this case, our random choice is forwards, but that leads us to a wall, so we continue our search clockwise:



Simple wandering algorithm 2/2

After going clockwise once, we hit another wall, so we continue, this direction is good, but it would lead us backwards, so we ignore it. The only direction possible is left.

This algorithm is far from perfect: if we build a very simple maze, we can break the algorithm:



This maze breaks our wandering algorithm

If you calculate the probabilities, you can see that starting from the entrance, the AI has a 75% chance of going into one of those alcoves and only a 25% chance of going forwards.

This means that an entity has only a  $25\%^6 = 0.0244\%$  chance of finishing the maze without ever turning left. Not only that, but every time the entity exits an alcove, it has a 50% chance of turning either direction (turning left means going forwards, going right means going backwards), which means that there is a 37.5% chance (75% chance turning left and 50% turning right) that the entity will go back towards the entrance of our maze.

Let's see a possible implementation of this algorithm.

```
const string[] DIRECTIONS = ["NORTH", "EAST", "SOUTH", "WEST"];
```

```
class AIEntity{
    // 0=North, 1=East, ...
    forward_direction_index = 0;
    current_cell = (1,0);
    // ...
    function get_adjacent_cell(Cell cell, string
direction) -> Cell{
        // Returns the adjacent cell in said
        direction
        cell_copy = cell.copy();
```

```

        if (direction == "NORTH") {
            cell_copy.y -= 1;
        }
        if (direction == "SOUTH") {
            cell_copy.y += 1;
        }
        if (direction == "WEST") {
            cell_copy.x -= 1;
        }
        if (direction == "EAST") {
            cell_copy.x += 1;
        }
        return cell_copy;
    }

    function is_valid(Cell cell) -> bool{
        /* Returns true if the cell is valid, aka
         * does not have a wall and does not go
backwards */
        if (cell.is_wall()){
            // The cell is a wall
            return False;
        }
        if (cell == get_adjacent_cell([(forward_direction_index + 2) % 4])){
            // We're going backwards, we don't want
that
            return False;
        }
        // In all other cases, it's valid
        return True;
    }

    function update(float dt){
        // ...
        // Choose a random direction
        string chosen_direction =
random_choice(DIRECTIONS);
        int i = 0;
        Cell next_cell =
get_adjacent_cell(current_cell, chosen_direction);
        while (not is_valid(next_cell) and i!=4){
            chosen_direction =
DIRECTIONS[ (DIRECTIONS.indexOf(chosen_direction) + 1) % 4];
            next_cell =
get_adjacent_cell(current_cell, chosen_direction);
            i = i + 1;
        }
    }
}

```

```

        if (i == 4) {
            // We exhausted the possibilities, go
backwards
            chosen_direction =
DIRECTIONS[(forward_direction_index + 2) % 4];
            next_cell =
get_adjacent_cell(current_cell, chosen_direction);
        }
        // Move
        move_to(next_cell);
    }
}

```

Code: Implementation of a simple wandering algorithm

## A slightly better “Wandering” algorithm

As we have seen, the previous wandering algorithm has a very heavy bias, let's plan for another algorithm that works a bit better and has a lower bias.

1. Check for all valid directions around you.
2. If no valid direction is found, go back.
3. If at least one valid direction is found, choose a random one between the valid directions found.

This algorithm relies on the fact that randomly selecting an item from a list containing a single item will return that single item in 100% of the cases.

Let's see a code implementation of such algorithm.

```

const string[] DIRECTIONS = {"NORTH", "EAST", "SOUTH", "WEST"};

class AIEntity{
    // 0=North, 1=East, ...
    forward_direction_index = 0;
    current_cell = (1,0);
    // ...
    function get_adjacent_cell(Cell cell, string
direction) -> Cell{
        // Returns the adjacent cell in said
direction
        cell_copy = cell.copy();
}

```

```

        if (direction == "NORTH") {
            cell_copy.y -= 1;
        }
        if (direction == "SOUTH") {
            cell_copy.y += 1;
        }
        if (direction == "WEST") {
            cell_copy.x -= 1;
        }
        if (direction == "EAST") {
            cell_copy.x += 1;
        }
        return cell_copy;
    }

    function is_valid(Cell cell) -> bool{
        /* Returns true if the cell is valid, aka
         * does not have a wall and does not go
backwards */
        if (cell.is_wall()) {
            // The cell is a wall
            return False;
        }
        if (cell == get_adjacent_cell([(forward_direction_index + 2) % 4])) {
            // We're going backwards, we don't want
that
            return False;
        }
        // In all other cases, it's valid
        return True;
    }

    function get_available_directions(Cell cell) ->
string[] {
    /* Returns a list of available directions */
    string[] result = [];
    for (direction in DIRECTIONS) {
        if (is_valid(get_adjacent_cell(cell,
direction))) {
            result.append(direction);
        }
    }
    return result;
}

function update(float dt) {

```

```

        // ...
        // Get a list of the available directions
        string[] available_directions =
get_available_directions(current_cell);
        string chosen_direction = "NORTH"; // Just
a default
        if (available_directions.is_empty()){
            // No directions are available, let's
go back
                chosen_direction =
DIRECTIONS[(forward_direction_index + 2) % 4];
        }else{
            // Choose a random direction among the
available ones
                chosen_direction =
random_choice(available_directions);
        }
        // Move
        next_cell = get_adjacent_cell(current_cell,
chosen_direction);
        move_to(next_cell);
    }
}

```

### Code: Implementation of a better wandering algorithm

This algorithm chooses between all available directions with the same probability, and has a minor bias towards going to the maze entrance.

## The Greedy “Best First” Algorithm

This is a *greedy algorithm*<sub>g</sub> 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:

```
struct Node{  
    Node parent; // This will be used to build the  
path  
    float h; // The h(x) value for the node  
}
```

Code: The node structure used in the greedy "Best First" algorithm

Let's look at the algorithm itself:

```
// We bootstrap the variables  
    currentNode = start;  
    add currentNode to closedSet; // This will avoid  
re-analyzing this node  
    do{  
        for (each Node n adjacent to currentNode) {  
            if (closedSet contains n){  
                // We already analyzed this node,  
continue to next n  
                skip the node n;  
            }else{  
                n.parent = currentNode;  
                if (openSet does not contain n){  
                    compute n.h; // Computes the value  
of n's h(x)  
                    add n to openSet;  
                }  
            }  
        }  
        if (openSet is empty){  
            // We exhausted all the possibilities  
            break do loop;  
        }  
        // Select a new "currentNode"  
        currentNode = Node with lowest h inside the  
openSet;  
        remove currentNode from openSet;  
        add currentNode to closedSet;  
    }until (currentNode == end)  
  
    if (currentNode == end){  
        /* We reached the end and solved the path, we  
need to do a stack
```

```

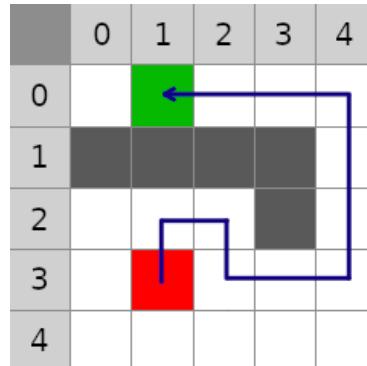
        reversal to find the path */
    Stack finalPath;
    Node n = end;
    while (n is not null){
        add n to finalPath;
        n = n.parent; // We use "parent" to run
the found path backwards
    }
} else{
    // We cannot find a path between "start" and
"end"
    // ... Behave accordingly
}

```

Code: The greedy "Best First" algorithm

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.



The path taken by the greedy “Best First” algorithm

## 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:

```
struct Node{  
    Node parent; // This will be used to build the  
    path  
    float g; // The path cost value for the node  
}
```

Code: The node structure used in the Dijkstra Algorithm

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:

```
currentNode = start;  
add currentNode to closedSet;  
do{  
    for (each Node n adjacent to currentNode) {  
        if (closedSet contains n){  
            skip the node n;  
        }else if (openSet contains n){ // Check if  
this path is better  
            compute new_g (path cost value);  
            if (new_g < n.g){  
                // We found a better path from  
start to currentNode  
                n.parent = currentNode;  
                n.g = new_g;  
            }  
        }else{  
            n.parent = currentNode;  
            compute n.g;  
            add n to openSet;  
        }  
    }  
  
    if (openSet is empty){  
        // We exhausted all possibilities  
        break loop;  
    }  
    currentNode = Node with the lowest g in the
```

```

openSet;
    remove currentNode from openSet;
    add currentNode to closedSet;
}until(currentNode == end)

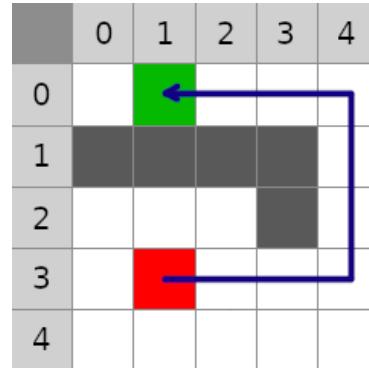
// Reconstruct the path like in the greedy "best-
first" algorithm

```

### Code: The Dijkstra 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:



The path taken by the Dijkstra Algorithm

### 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:

```
currentNode = start;
    add currentNode to closedSet;
    do{
        for (each Node n adjacent to currentNode) {
            if (closedSet contains n) {
                skip the node n;
            }else if (openSet contains n){ // Check if
this path is better
                compute new_g (path cost value);
                if (new_g < n.g){
                    // We found a better path from
start to currentNode
                    n.parent = currentNode;
                    n.g = new_g;
                    n.f = n.g + n.h;
                }
            }else{
                n.parent = currentNode;
                compute n.g;
                compute n.h;
                n.f = n.g + n.h;
                add n to openSet;
            }
        }

        if (openSet is empty){
            // We exhausted all possibilities
            break loop;
        }
        currentNode = Node with the lowest f in the
openSet;
        remove currentNode from openSet;
        add currentNode to closedSet;
    }until(currentNode == end)

    // Reconstruct the path like in the greedy "best-
first" algorithm
```

### Code: The A\* 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.

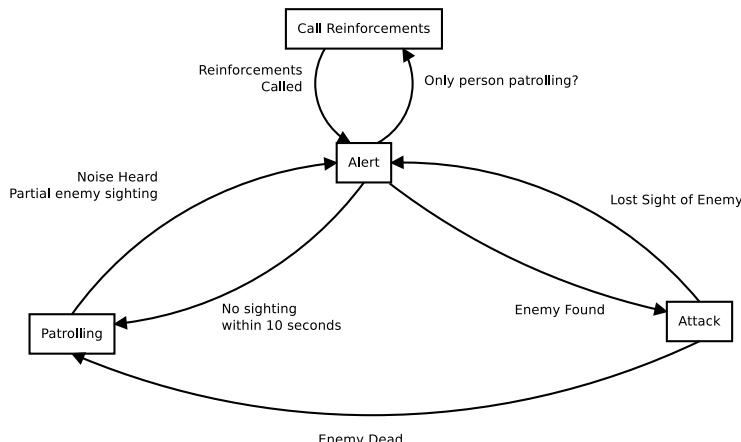
### Dijkstra Algorithm as a special case of the A\* Algorithm

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.

## Finite state machines

We can use finite state machines, introduced previously, to define some quite complex Artificial Intelligence schemes.



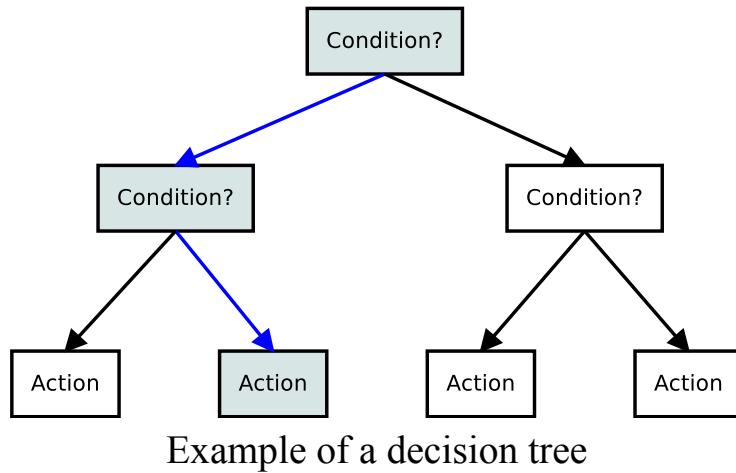
Finite state machine representing an enemy AI

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.

## Decision Trees

Decision trees are a structure used to define the decision process of an AI-controlled entity.

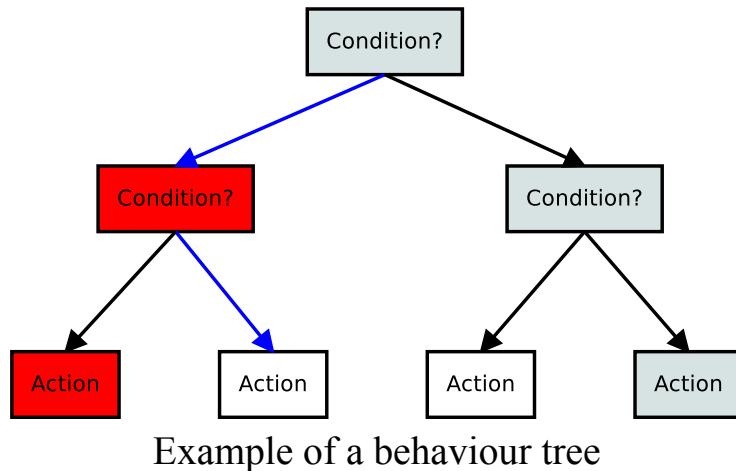


Decision trees are always evaluated from root to leaf, and each node represents a condition that can be more or less complex. In the image above we used a simple “binary tree” to represent conditions that can be answered with “yes” or “no”.

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

## Behaviour Trees

Similar in structure to Decision Trees, *Behaviour Trees* are different in their evaluation.



First of all, the child nodes of a behaviour tree are ordered by priority, if a child node has all of its conditions met its internal state is changed to “running” and the chosen “behaviour” is returned to the caller.

The next evaluation, the tree is again evaluated, in order, if a “running” node is met, such behaviour will continue to persist for the current frame. In case a “higher priority” behaviour is met, that one behaviour will start running (instead of the behaviour chosen earlier).

In case a node doesn’t have all of its conditions met, the algorithm will return to the parent node, which will chose the next node, in priority order.

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

---

1. Jawbreaker tileset, listed as public domain at  
<https://adamatomic.itch.io/jawbreaker> ↵
2. Jawbreaker tileset, listed as public domain at  
<https://adamatomic.itch.io/jawbreaker> ↵
3. Jawbreaker tileset, listed as public domain at  
<https://adamatomic.itch.io/jawbreaker> ↵

# Other Useful Algorithms

Fancy algorithms are slow when n is small, and n is usually small.

---

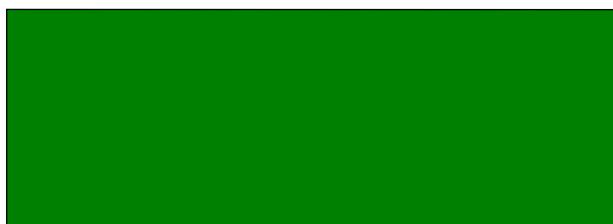
Rob Pike

## World Generation

### Midpoint Displacement Algorithm

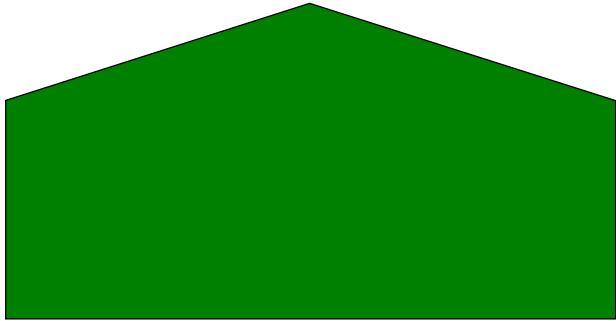
As the name implies, this algorithm entails recursively taking the midpoint between two extremes and “displace” it.

Let’s see how it works first. We have a completely flat terrain:



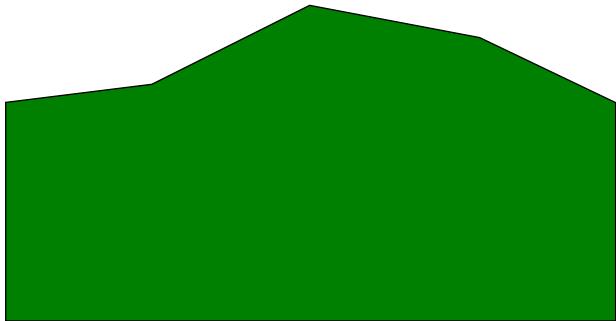
How the Midpoint Displacement Algorithm Works (1/4)

Then we take the midpoint between the two extremes and move it (up or down) by a random amount (between two sensible extremes):



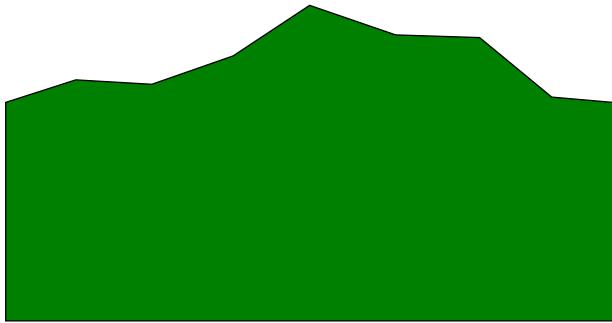
How the Midpoint Displacement Algorithm Works (2/4)

In this case we moved it up by a certain measure, now we take the two midpoints on the left and right sides of the previous midpoint, and displace them too:



How the Midpoint Displacement Algorithm Works (3/4)

Then we take the midpoints between the segments we created, and displace them again:



### How the Midpoint Displacement Algorithm Works (4/4)

Each displacement is usually done by a lower amount of the previous ones, so that the first displacements give a “general shape” of the terrain, while the ones further down the line are going to give “detail” to our terrain.

The algorithm terminates when we reached a pre-defined number of subdivisions, sometimes called “octaves”. In the previous example, we have 4 octaves.

Let's take a look at a possible implementation of the midpoint displacement algorithm:

```
int MIN = 0;
    int MAX = 100;
    int OCTAVES = 5;

    // This will contain the "heights" of our terrain
    float[] terrain = new float[32];

    // We start by deciding the start and end
    "heights" of our terrain
    terrain[0] = random_float(MIN, MAX);
    terrain[31] = random_float(MIN, MAX);
    // We interpolate all the missing values
    interpolate(terrain, 0, 31);

    function midpoint_displacement(int begin, int end,
int octave) {
        // Get the midpoint
        float midpoint = floor((end - begin) / 2);
```

```

        // Get the midpoint value
        float value = (abs(terrain[end] -
terrain[begin])) / 2;
        // Get the possible displacement
        float displacement = MAX / octave;
        // Displace by a random amount
        value += random_float(-displacement,
displacement);
        // Apply the value
        terrain[midpoint] = value;
        // Interpolate the values between begin and
midpoint
        for (each i between "begin + 1" and "midpoint
- 1") {
            terrain[i] = interpolate(terrain, begin,
midpoint);
        }
        // Interpolate the values between midpoint and
the end
        for (each i between "begin + 1" and "midpoint
- 1") {
            terrain[i] = interpolate(terrain,
midpoint, end);
        }
        // Recursion on the subtree
        if (octave < OCTAVES) {
            // Recur left
            midpoint_displacement(begin, midpoint,
octave + 1);
            // Recur right
            midpoint_displacement(midpoint, end,
octave + 1);
        }
    }
}

```

Code: Example implementation of the midpoint displacement algorithm

## Diamond-Square Algorithm

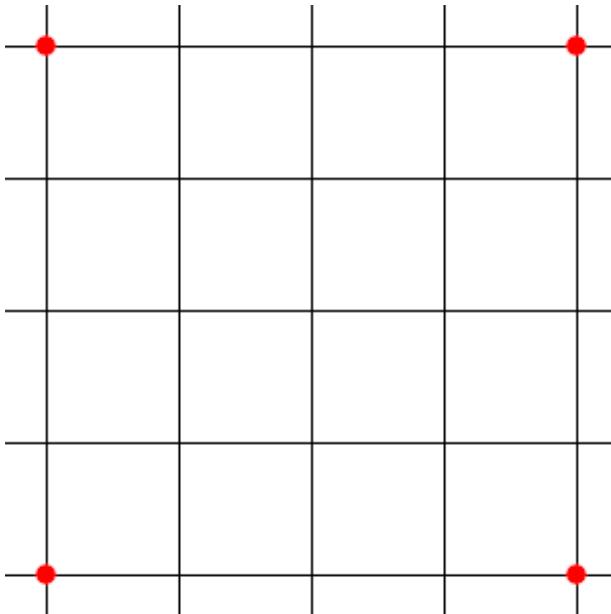
The diamond-square algorithm is an evolution in 2D of the midpoint displacement algorithm (so far, we just changed one value, in one dimension).

The algorithm iteratively repeats two steps:

- a **diamond step** where you find all the squares, take the midpoint and set it as the average of the four corners, plus a random displacement;
- a **square step** where you find all the diamonds, take the midpoint and set it as the average of the four corners, plus a random displacement.

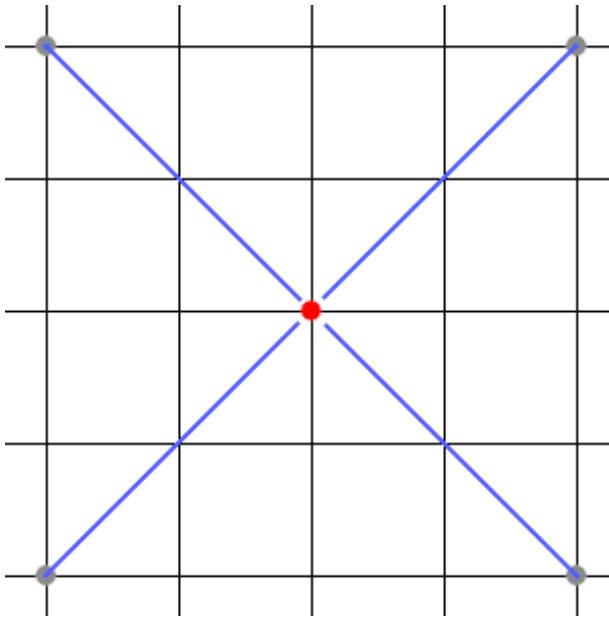
Let's see how it works.

First of all, we bootstrap our square with arbitrary values at the four corners, this will be our starting point.



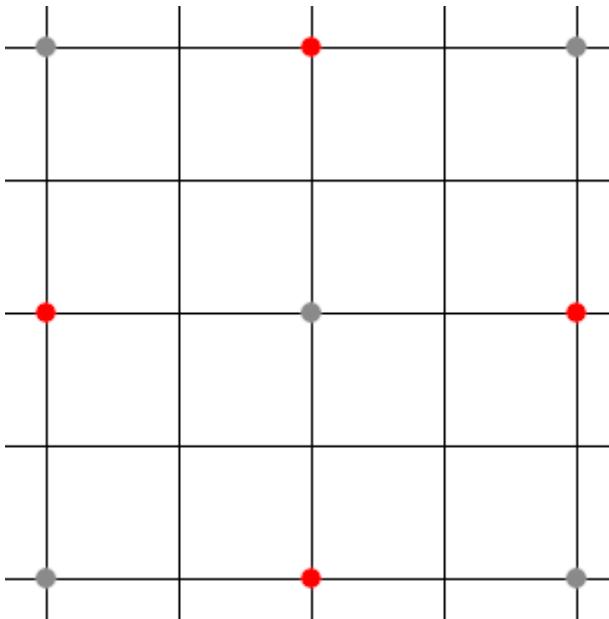
How the diamond-square algorithm works (1/5)

Now we perform the first “diamond step”: we have only one big square, with 4 corners, we identify its center and set its value to the average of the corners and apply a random displacement.



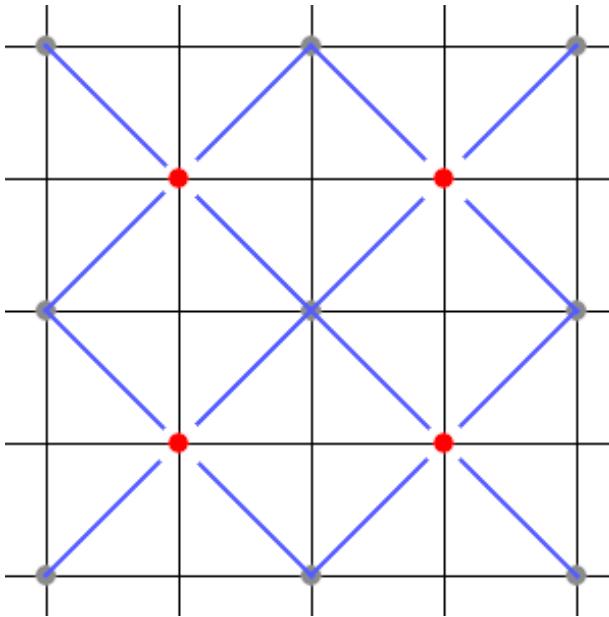
How the diamond-square algorithm works (2/5)

Now we can perform our first “square step”: we have 4 diamonds, we identify their centres (in red) and set them to the average of their neighbours, plus a random displacement.



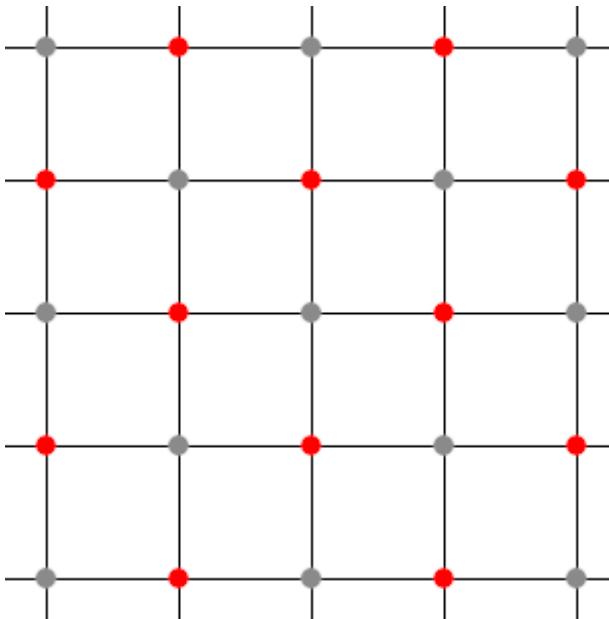
How the diamond-square algorithm works (3/5)

Now we can iterate with another “diamond step” on the four new smaller squares we have, get their centers and again average and displace.



How the diamond-square algorithm works (4/5)

As the last step, for this 5x5 grid, we perform another “square step”, finding 12 smaller diamonds, and again getting the average of the corners and apply a displacement.



How the diamond-square algorithm works (5/5)

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

## Maze Generation

Maze generation is the base of a great majority of dungeon generation systems, you can create a maze, carve out a few rooms, put an entrance and an exit and you have a nice quick dungeon!

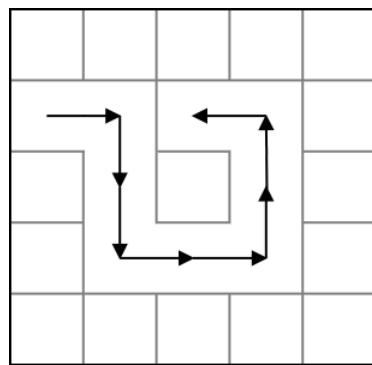
There are many ideas that can be used to generate a maze, some are based on a prepared map that gets refined into a maze, some other are based on walls instead of tiles, here we will see some of the many algorithms that exist.

### Randomized Depth-First Search (Recursive Backtracker)

The Depth-First Search (DFS) algorithm is known in the world of tree and graph structure as a traversal algorithm. We can use a randomized DFS algorithm as a simple maze-generation algorithm.

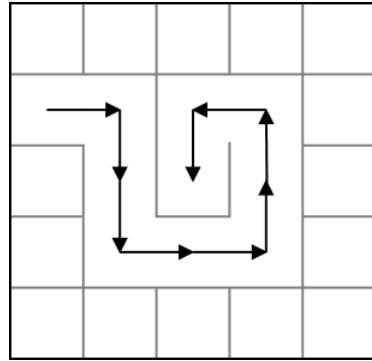
The Randomized DFS Algorithm is usually implemented using the backtracking technique and a recursive routine: such algorithm is called “Recursive Backtracker”.

The idea behind the algorithm is, starting from a defined “cell”, to explore the grid randomly by choosing an available direction, digging a path.



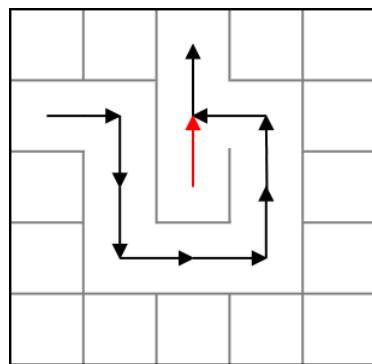
How the recursive backtracker algorithm works (1)

When the algorithm detects that there is no available direction that means that the “head” of our digger is hitting against already explored cells or the map borders.



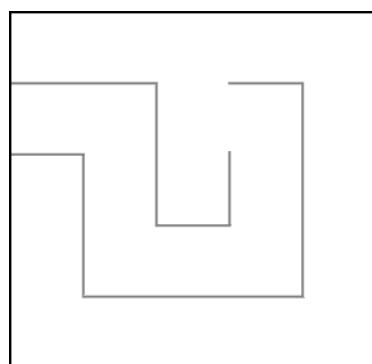
How the recursive backtracker algorithm works (2)

In such case, we “backtrack” until we find a cell with at least one available direction and continue our exploration.



How the recursive backtracker algorithm works (3)

This “digging and backtracking” keeps going until there are no other cells that have not been visited.



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* 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.

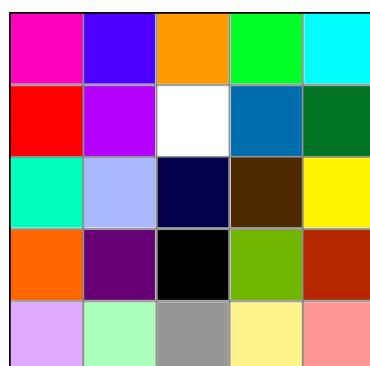
## **Randomized Kruskal's Algorithm**

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:

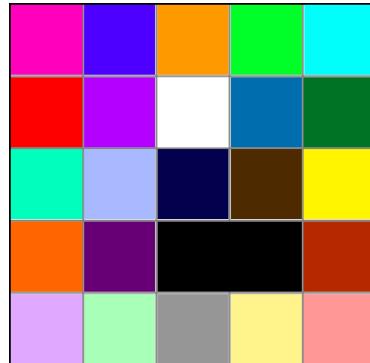
- One structure that contains all the “walls” of our maze, this can be a list
- One structure that allows for easy joining of disjoint sets, this will contain the cells

Initially all the cells are separated by walls, and each cell is its own set.



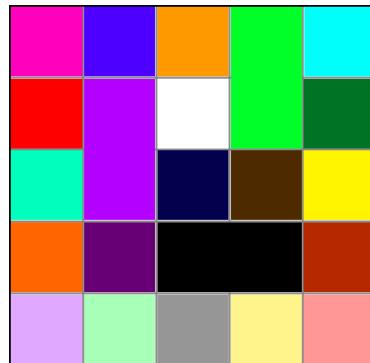
How the Randomized Kruskal's Algorithm Works (1/6)

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.



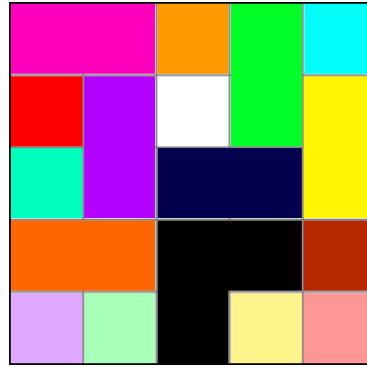
#### How the Randomized Kruskal's Algorithm Works (2/6)

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.



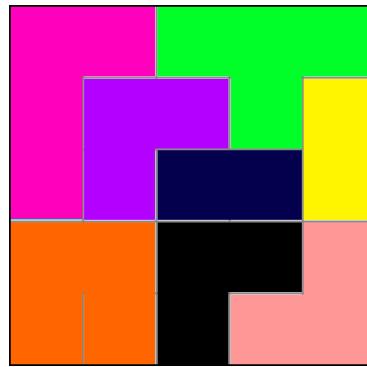
#### How the Randomized Kruskal's Algorithm Works (3/6)

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.



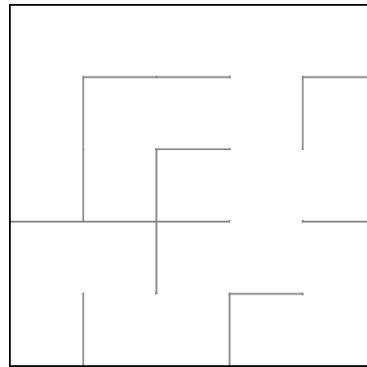
How the Randomized Kruskal's Algorithm Works (4/6)

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.



How the Randomized Kruskal's Algorithm Works (5/6)

When there is only one set left, our maze is complete.



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]*

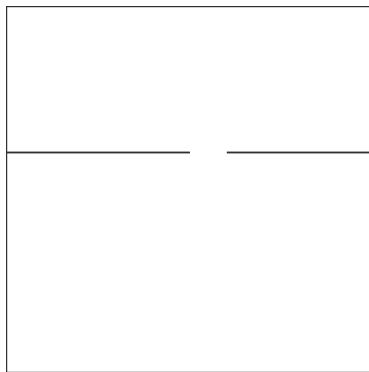
## Recursive Division Algorithm

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.

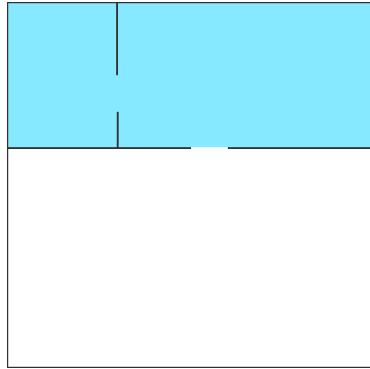
Let's see how the algorithm works.

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.



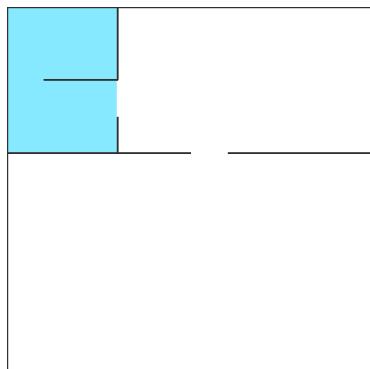
How the Recursive Division Algorithm Works (1/6)

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.



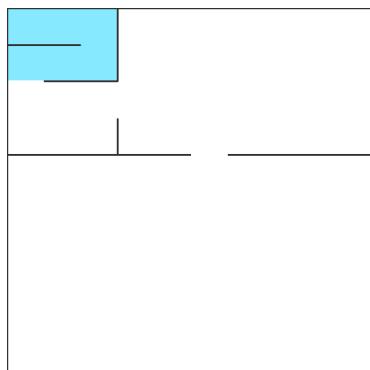
How the Recursive Division Algorithm Works (2/6)

We select one of the two sub-sub-area, and add another wall, with a random position and direction.



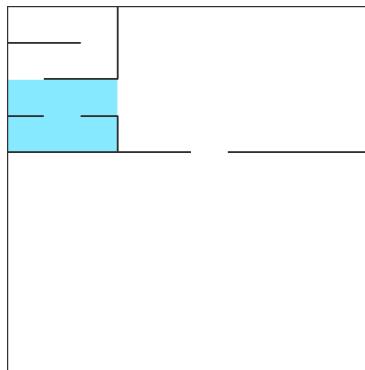
How the Recursive Division Algorithm Works (3/6)

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.



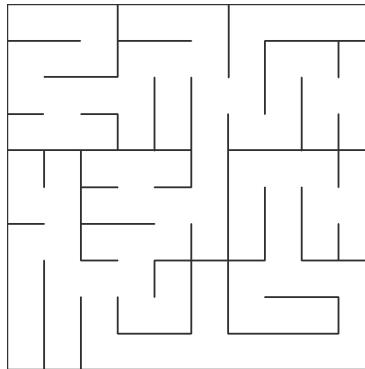
How the Recursive Division Algorithm Works (4/6)

When that happens, we backtrack to one of the previous sub-sections and continue.



How the Recursive Division Algorithm Works (5/6)

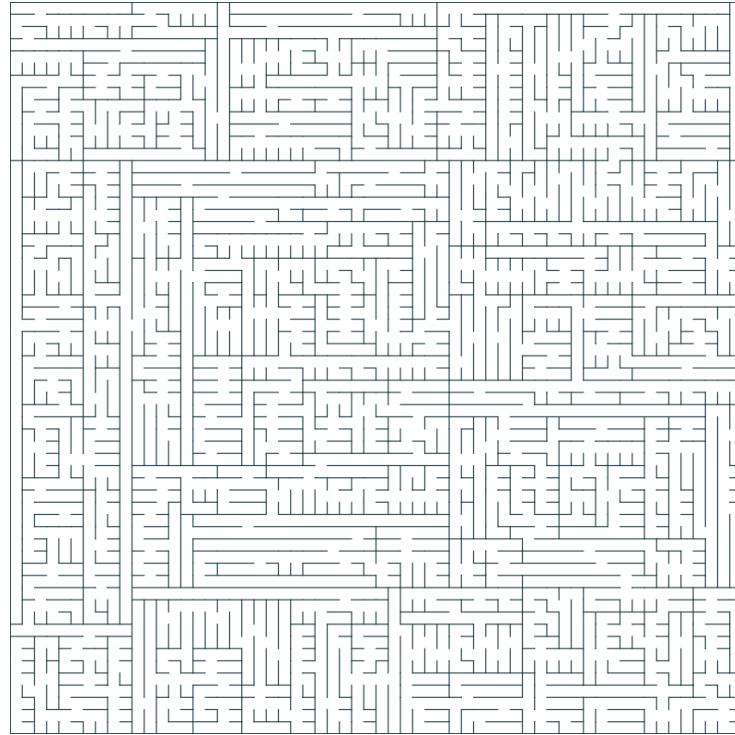
This keeps going until the maze is complete.



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.



The bias of Recursive Division Algorithm

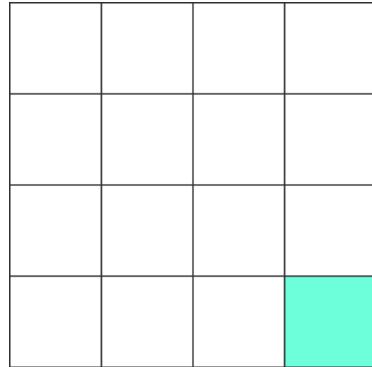
Let's see an example implementation of this algorithm:

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

## Binary Tree Algorithm

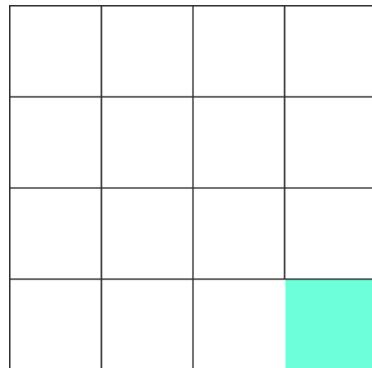
This is another very efficient “passage carver” algorithm: for each cell we carve a passage that either leads upwards or leftwards (but never both).

Let's see how the algorithm works: we start from the bottom-right cell of the maze (the last one).



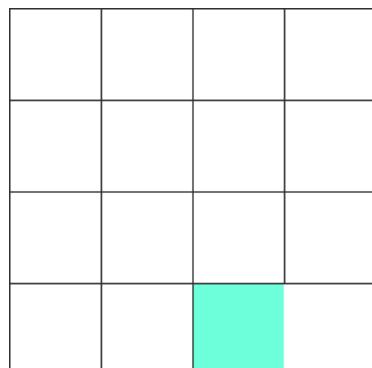
How the Binary Tree Maze generation works (1/6)

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.



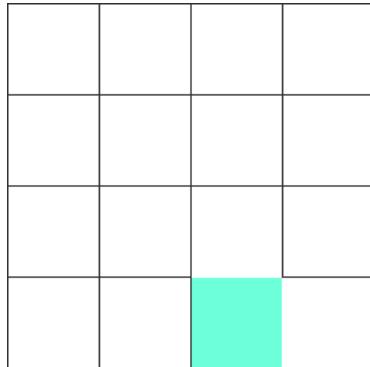
How the Binary Tree Maze generation works (2/6)

Now let's go to the second-to-last cell...



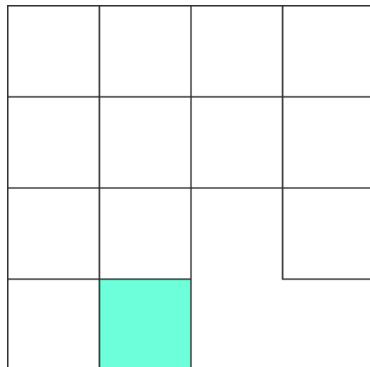
How the Binary Tree Maze generation works (3/6)

And again, decide randomly to carve a passage either upwards or leftwards, this time we chose upwards.



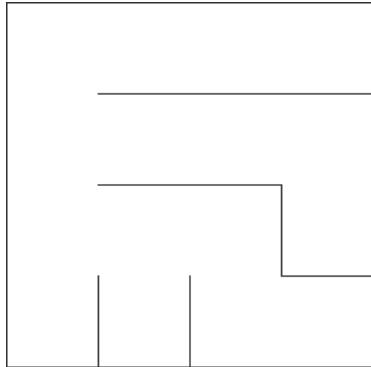
How the Binary Tree Maze generation works (4/6)

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.



How the Binary Tree Maze generation works (5/6)

Here is the result of the algorithm:



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]*

## 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”.



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”.



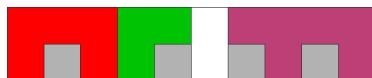
How Eller’s Maze Generation Algorithm Works (2/7)

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.



### 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.



### How Eller’s Maze Generation Algorithm Works (4/7)

After that we assign a new set to the remaining cells.



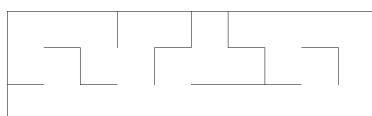
### How Eller’s Maze Generation Algorithm Works (5/7)

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).



### 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.



### 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]*

## Dungeon Generation

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

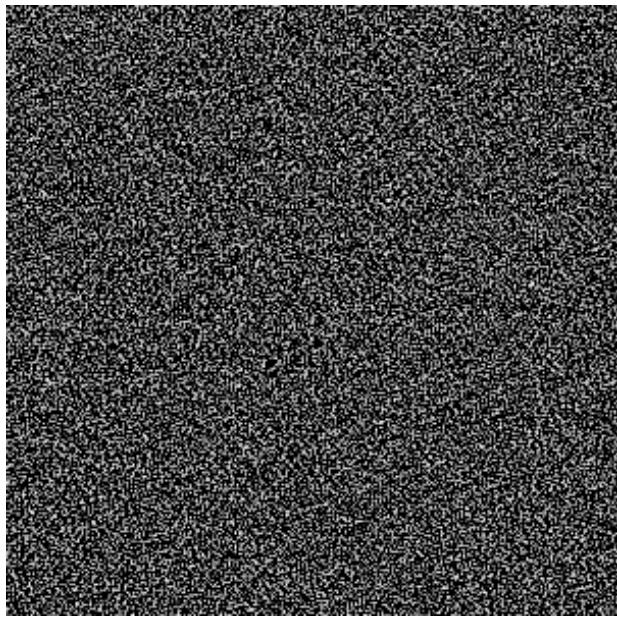
## Noise Generation

“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.

### 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:



Example of Random Noise

We can create some “TV-like” static with a few lines of code, like the following:

```
int WIDTH = 800;
    int HEIGHT = 600;
    // We create an empty texture
    Texture texture = Texture(WIDTH, HEIGHT);

        // Now we iterate through each pixel of the
texture
    for (each row in texture){
        for (each pixel in row){
            // We create a random gray color (0 is
black, 255 is white)
            int rand_gray_tone = random_int(0,255);
            // Most colors are made of Red Green and
Blue, by placing them at the
            // same value, we get a tone of gray
            Color rand_color = Color(rand_gray_tone,
rand_gray_tone, rand_gray_tone);
            pixel.setColor(rand_color);
        }
    }
```

Code: Example implementation of randomized noise

## **Perlin Noise**

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

# Procedural Content Generation

Science is what we understand well enough to explain to a computer. Art is everything else we do.

---

Donald Knuth

## 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:

```
function createProceduralWeapon() -> Weapon{  
    Weapon wp = new Weapon();  
    // Load some pre-defined components
```

```

        wp.loadBody("body0001.png");
        wp.loadScope("scope0051.png");
        wp.loadBarrel("barrel0045.png");
        wp.loadAmmoMagazine("mag0009.png");
        // set weapon damage to 45
        wp.setDamage(45);
        // set weapon range to 15
        wp.setRange(15);
        // set weapon spread to 23
        wp.setSpread(23);
        return wp;
    }
}

```

### Code: Example procedural weapon creation

This instead is a possible algorithm for a random weapon, for simplicity we assume that the pieces are all compatible:

```

function createRandomizedWeapon() -> Weapon{
    Weapon wp = new Weapon();
    // Load some randomized components

    wp.loadRandomBodyFrom("weaponBodies/shotguns");

    wp.loadRandomScopeFrom("weaponScopes/shotguns");

    wp.loadRandomBarrelFrom("weaponBarrels/shotguns");

    wp.loadRandomAmmoMagazineFrom("weaponMagazines/shotguns");
        // set weapon damage to a value between 35 and
50
        wp.setDamage(random(35, 50));
        // set weapon range to a value between 13 and
18
        wp.setRange(random(13, 18));
        // set weapon spread to a value between 20 and
30
        wp.setSpread(random(20, 30));
    return wp;
}

```

### Code: Example Randomized weapon creation

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.

## **Advantages and disadvantages**

As with everything, procedural and random generation has its advantages and disadvantages, which will be explained below.

### **Advantages**

#### **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.

#### **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.

## **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.

## **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.

## **Disadvantages**

### **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.

### **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.

### **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 patters are really easy to spot and can remove us from the game’s atmosphere and introduce us to one of our worst enemies: boredom.

This can become even worse if we try to “find the middle ground” and build our levels using hand-made “chunks”, joined together. If we want to avoid our player getting bored by repeating “pieces of level” we will need to build a lot of chunks that fit together to make something interesting and new almost every time.

### **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...

### **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.

## 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;
- **Difficulty Management:** Procedural generation can be involved into difficulty management by handing the enemy parameters and spawning to suit our needs.

## Procedural Generation and Difficulty Management

As stated above, we can use elements of procedural generation to aid us in managing the difficulty of our game, tweaking the challenge we are offering to our players.

### Static difficulty

Sometimes called “Algorithmic Difficulty”, is the kind of difficulty management seen in many games: elements and enemies are placed in a way that gives the player an ascending difficulty curve, with different parameters.

Such parameters build the abstract concept of “difficulty level” (beginner, normal, advanced, master, ...): each difficulty level contains a set of parameters that change how the game feels.

In practice usually you assign a “difficulty level” to an area and make the game handle the enemy spawn accordingly; sometimes such “area level” is used to make the game spawn randomized items with a certain power level, given a certain amount of displacement of its statistics.

## **Adaptive Difficulty**

Not really considered “procedural generation”, adaptive difficulty makes use of different algorithms (taken on a lease from AI programming) to tweak the game difficulty in reaction to how the player plays.

If the player progresses quickly, the game will become harder, instead if the player tends to lose lives, the game will become easier and less random.

The objective of adaptive difficulty is to create an “optimal experience” for everyone, by determining if the current game is “too easy” or “too hard” for the player.

## **Rubberbanding**

Rubberbanding (not to be confused with “network lag”, sometimes called “rubberbanding”) is an instance of adaptive difficulty, mostly used in racing games, where the opponents’ abilities (like speed in racing games) are “tweaked” to keep the game challenging (but not unfair).

This usually ends up with opponents getting faster the further in the lead the player is (sometimes even going over the maximum speed) or going slower when the player falls behind too much.

You can see as a virtual “rubber band” that ties your opponents to you, the more the rubber band is stretched, the more the opponents are “attracted” to you (or you to them, it can work both ways).

### **Random Trivia!**

“Rubberbanding” is not limited to racing games. For instance NBA Jam tweaks player skills to keep the game competitive and enjoyable.

## **Static vs. Adaptive Difficulty**

Each approach has its own advantages and shortcomings, which can make one or the other better suited for your game.

Static difficulty is easy to create and leaves choice to the players between varying levels of difficulty; maybe someone wants a more “relaxing experience” instead of being continuously challenged.

The biggest shortcoming is that each level of difficulty is an estimate of its difficulty, so an “easy mode” may be way too easy, while the “normal” mode may be too hard for someone. It’s hard to find the balance.

Also there’s all the work dedicated to program the parameters for each level of difficulty.

Adaptive difficulty is harder to code and sometimes can lead to great results, but it also completely invalidates the concept of “grinding” in an RPG game, for instance. If you try to become stronger by undertaking easier quests, you will find that the quests keep getting harder the stronger you get.

Adaptive difficulty doesn’t allow the player to “grind their way out” of a difficult part of the game.

# 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.

## General Purpose

### 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:

```
const float INVINCIBILITY_TIME = 0.75; // Seconds of
invincibility
    ...
    function update(float dt) {
        float inv_time = 0;
        ...
        if (inv_time <= 0){
            // Check for collision
            ...
            // Collision has been detected here, we
have a hit
            inv_time = INVINCIBILITY_TIME; // Start of
the invincibility period
        }else{
            // We are currently invincible
            inv_time = inv_time - dt; // We decrease
the invincibility time
        }
    }
```

Code: Example of I-Frames Implementation

## Scrolling Backgrounds and Parallax Scrolling

### Infinitely Scrolling Backgrounds

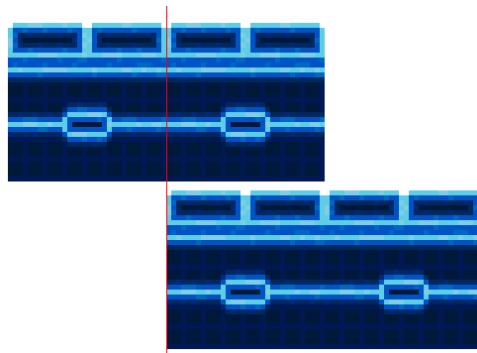
When doing any kind of game that features a scrolling background, you should construct your art accordingly, allowing for enough variety to make

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.



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.

```
float background_x_offset = 0.0; // The x offset of the background
                                const float BACKGROUND_X_SIZE = 512; // The
                                horizontal size of the background
                                const float LOOP_POINT = 256; // The horizontal
                                loop point of the image
                                const float DISTANCE_FACTOR = 0.5; // The
```

```

background moves at half the player speed

    function update(float dt) {
        // ...
        // In case we're moving right, the background
scrolls left slightly
        if (player.speed_x > 0){
            // Update player's position and state
            ...
            // Update the background position
            background_x_offset = background_x_offset -
player.speed_x * DISTANCE_FACTOR * dt;
            // If we passed the image's loop point
            if (background_x_offset <= - LOOP_POINT){
                // We reset the coordinates, keeping
note of the remainder
                background_x_offset =
background_x_offset % LOOP_POINT;
            }
        }
        // In case we're moving left, the background
scrolls right slightly
        if (player.speed_x < 0){
            // Update player's position and state
            ...
            // Update the background position
            background_x_offset = background_x_offset -
player.speed_x * DISTANCE_FACTOR * dt
            if (background_x_offset >= 0){
                // We reset the coordinates, keeping
note of the remainder, just backwards
                background_x_offset =
background_x_offset - BACKGROUND_X_SIZE;
            }
        }
    }

    function draw() {
        ...
        // Draw the background
        screen.draw(background, (background_x_offset,
0));
        ...
    }
}

```

Code: Example of an infinitely scrolling background implementation

## Parallax Scrolling

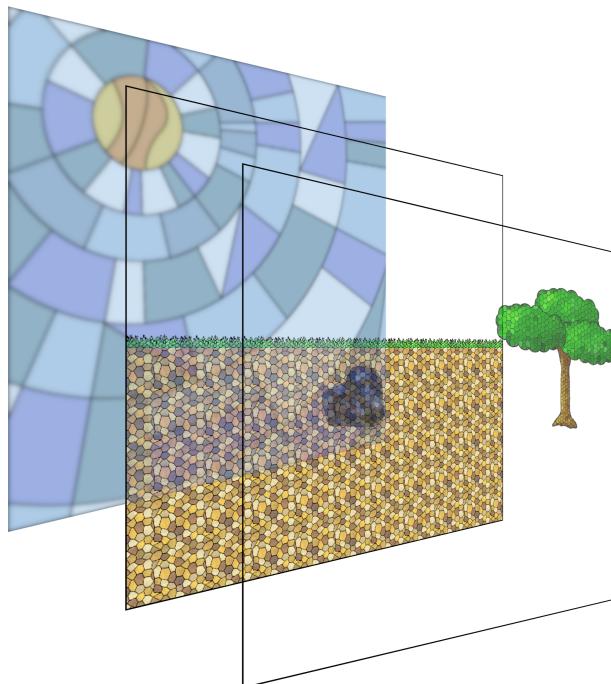
Parallax in games is an effect that can give more depth to our environment: it looks like objects farther away are moving much slower than the objects closer to us.

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.

For the sake of clarity, we will re-use an image presented earlier to explain the “painter’s algorithm”:



## How we can split our game into layers

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.

## 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.

## Sprite Stenciling/Masking

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

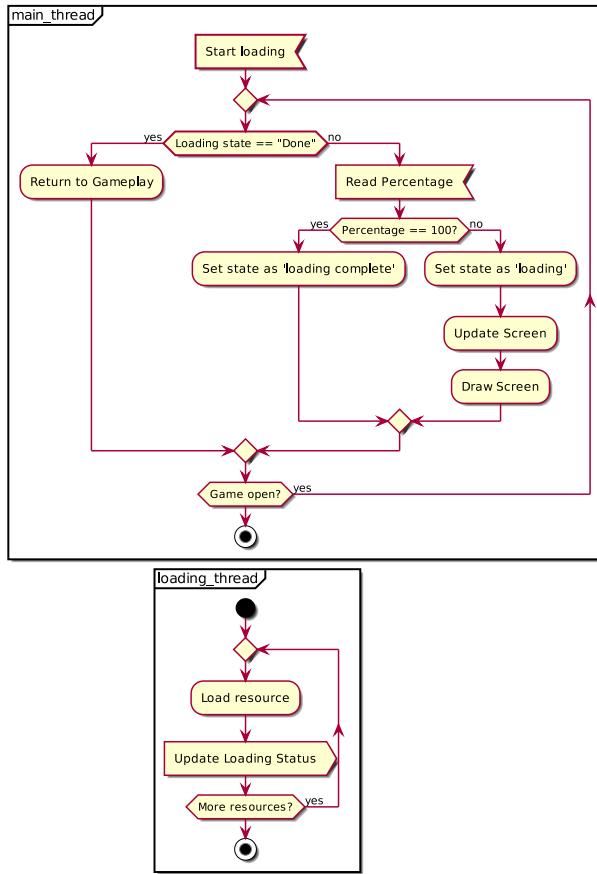
### Loading screens

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.

We can represent the two “loops” in the following UML diagram:



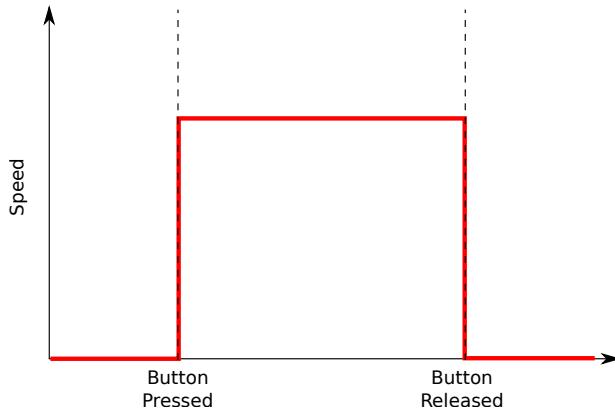
Rough UML diagram of a multithreaded loading screen

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

## Simulating Inertia

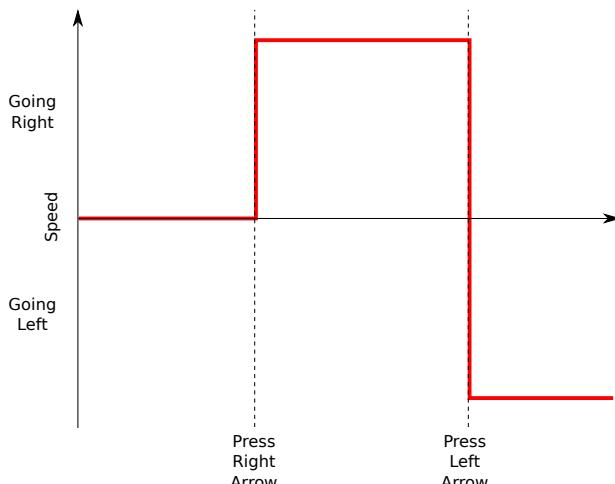
After learning how to move something on a screen, the next step is making the movement less “jarring” by introducing inertia. Before throwing solutions around, let’s see what the problem is.

When we press a button, without inertia, our character starts moving at full speed towards the direction we defined, and it will stop as soon as we let go of the button. We can represent such behaviour with the following chart:



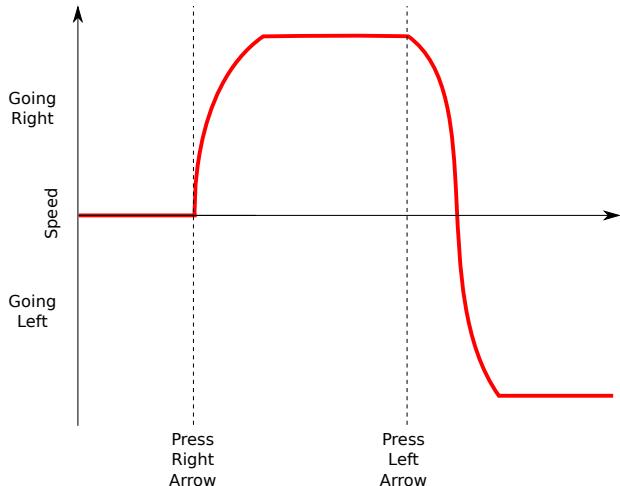
Example chart of how movement without inertia looks

This could be jarring on its own, but the situation gets more serious the higher the speed difference: one thing is having a character being able to run from a standstill and stop immediately, but it feels all the more jarring if a character can turn 180 degrees on its path without the slightest hint of inertia (we assume that positive speed means “going right”, while negative speed means “going left”):



Example chart of how movement without inertia looks: reversing directions

If this is connected to a [fully-tracking camera system](#) your player is in for a ride rivaling the deadliest rollercoasters in the world. What we want is the speed curve to behave more like the following (here too we assume that positive speed means “going right”, while negative speed means “going left”):



Example chart of how movement with inertia looks

A “softer” transition between directions can be a good way to avoid nausea as well as making the game behave more realistically, the change of direction can be also coupled with a skidding animation to make it even more convincing.

### Random Trivia!

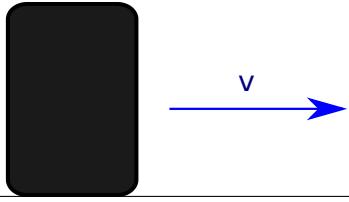
Inertia is so important (and common) that even the famous “Super Mario Bros.” (1983) for the NES features it, as well as a “skidding animation”.

In this section we will look at how to simulate inertia in a 1-dimensional space, where we can only move left or right.

When simulating inertia, the first things we need to know are:

- The top speed
- The acceleration rate
- The deceleration rate
- The direction we’re going
- The direction we want to go

Let’s think of the following situation, our character is running rightwards at a velocity  $v$ , measured in pixels per frame:

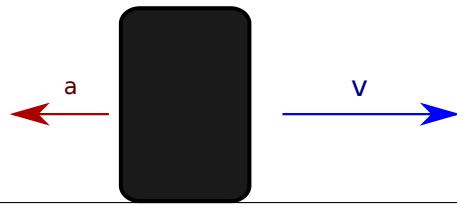


Example of character running

This means that the character's  $x$  coordinate is moving every frame using the formula:

$$x_{n+1} = x_n + v$$

Now we suddenly want the player to start walking leftwards: we need to apply an acceleration  $a$  in that direction:

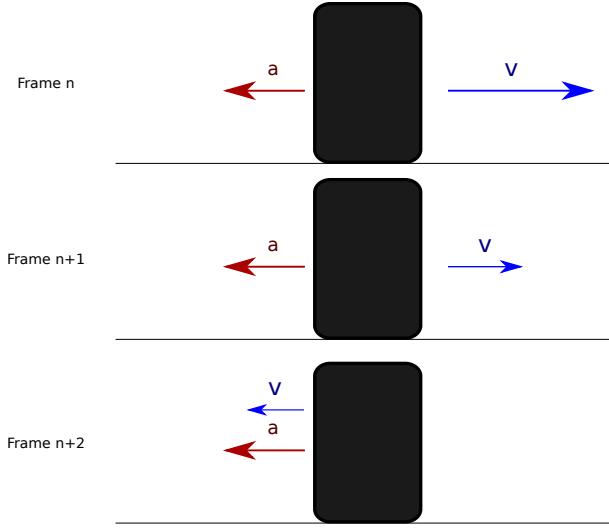


Applying an acceleration to a character running

The new acceleration will influence the velocity, frame by frame, with the formula

$$v_{n+1} = v_n + a$$

Since velocity and acceleration have opposite directions, the acceleration we're applying will start “eating away the velocity” frame by frame, until our character starts moving leftwards. This “eating away” phase is what gives the feeling of inertia.



Applying an acceleration frame by frame leads to the feeling of inertia

Being acceleration and velocity both vectors, we can apply an acceleration both in a one-dimensional way (like a 2D platformer) or a 2-dimensional way (like a space shooter) and the formulas will still be valid.

Deceleration is a special case of what we've seen so far, with the exception that the acceleration will always have direction opposite to velocity and as soon as velocity reaches zero, we stop applying it.

Now we can start writing some code:

```
// ...
class Player{
    const float MAX_SPEED = 50.0; // Maximum speed
    const float ACCEL = 15.0; // The acceleration
rate
    const float DECEL = 30.0; // The deceleration
rate
    Vector2 input_accel = Vector2(); // Defines
the direction we are accelerating
    Vector2 velocity = Vector2(); // Defines the
direction and magnitude of our speed
    Vector2 position = Vector2(); // Defines our
current position, in (x,y) coordinates
    bool is_moving = False; // Tells us if we're
moving

    function handle_input(){
        // First of all, we need to zero the
```

```

input_accel, or we'll be working on "residual data"
    input_accel = Vector2.ZERO;
    // Now we can handle movement
    if (KEYBOARD.Left_Arrow_Pressed) {
        input_accel.x = input_accel.x - 1;
    }
    if (KEYBOARD.Right_Arrow_Pressed) {
        input_accel.x = input_accel.x + 1;
    }
    if (KEYBOARD.Down_Arrow_Pressed) {
        input_accel.y = input_accel.y + 1;
    }
    if (KEYBOARD.Right_Arrow_Pressed) {
        input_accel.y = input_accel.y - 1;
    }
    // If any component of the acceleration
vector is not zero, we are moving
    if (input_accel != Vector2.ZERO) {
        is_moving = True;
    }
}

function handle_movement(float dt) {
    if (is_moving){
        // Vectors will take care of summing
forces
            velocity = velocity + ACCEL * dt *
input_accel;
            // We need to clamp the speed, to avoid
going too fast
            velocity.clamp(MAX_SPEED);
    }else{
        // We are stopping, let's subtract the
deceleration
            float velocity_value =
velocity.length() - DECEL * dt;
            if (velocity_value < 0){
                // If, After decelerating, we have
a negative value, we need to make it zero or the object will
start moving backwards
                velocity_value = 0;
            }
            // We are just changing the length of
the vector, so we can just clamp its length
            velocity.clamp(velocity_value);
    }

    // Now it's time to move the object
}

```

```
        position = position + velocity;  
    }  
}
```

Code: Code for simulating inertia

## 2D Platformers

### 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.

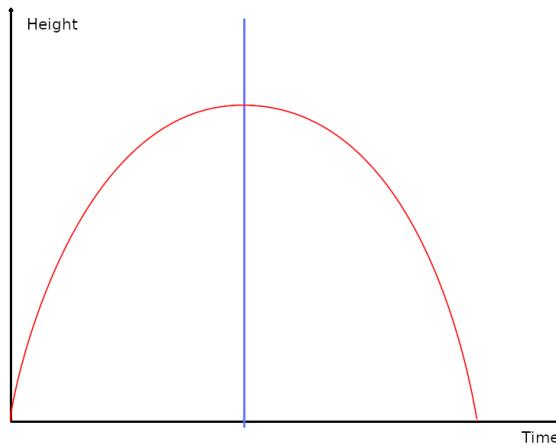
```
const int GRAVITY_ACCELERATION = 10;  
const float MAX_FALL_VELOCITY = 500;  
// ...  
// Apply Gravity  
speed_y = speed_y + GRAVITY_ACCELERATION;  
// Cap the fall speed  
if (speed_y > MAX_FALL_VELOCITY){  
    speed_y = MAX_FALL_VELOCITY;  
}
```

```
// ...
```

Code: Code for applying gravity to an object

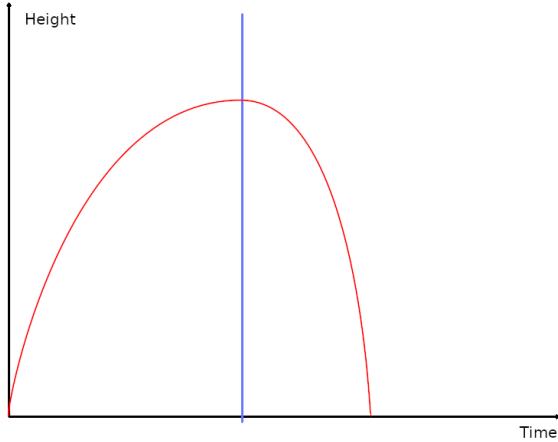
## Avoiding “Floaty Jumps”

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:



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:



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:

```
const int GRAVITY_ACCELERATION = 10;
        const float MAX_FALL_VELOCITY = 500;
        const float GRAVITY_FALL_MULTIPLIER = 1.5;
        // ...
        // Are we jumping?
        if (speed_y < 0){
            // Apply Gravity Normally
            speed_y = speed_y + GRAVITY_ACCELERATION;
        }else{
            // We're falling, enhance gravity
            speed_y = speed_y + GRAVITY_ACCELERATION *
GRAVITY_FALL_MULTIPLIER;
        }
        // Cap the fall speed
        if (speed_y > MAX_FALL_VELOCITY){
            speed_y = MAX_FALL_VELOCITY;
        }
        // ...
```

Code: Code for jump with enhanced gravity while falling

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.

## Ladders

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

## Walking on slanted ground

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

## Stairs

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

## Jump Buffering

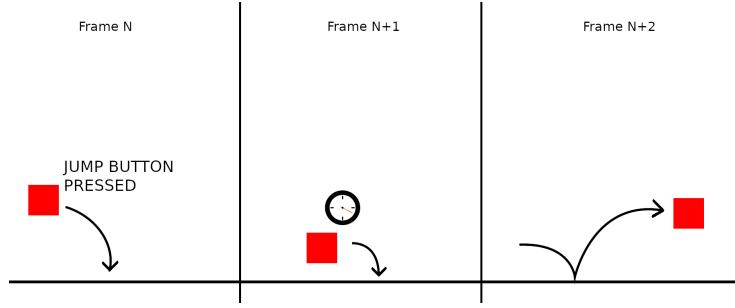
A nice trick used mostly in 2D platformers to allow for smoother gameplay is “jump buffering”, also known as “input buffering”.

Normally when a character is mid-air, the jump button does nothing, in code:

```
function update(float dt) {
    // ...
    if (controls.jump.isPressed()) {
        if (player.on_ground) {
            // Jump
        }
    }
    // ...
}
```

Code: Code for jumping without buffering

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.



Example of how jump buffering would work

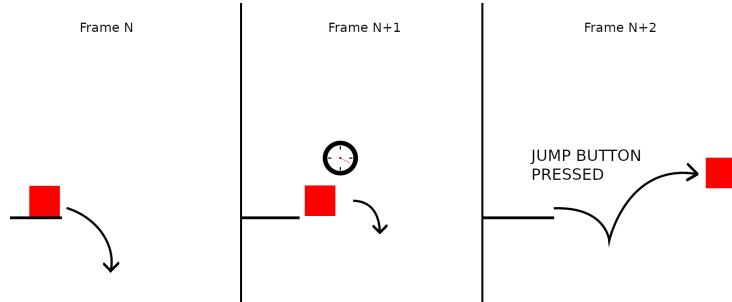
Jump buffering usually is put into practice using a timer, in a fashion similar to the following:

```
// ...
    const float jumpBufferTime = 5.0;
    // ...
    function update(float dt) {
        // ...
        if (controls.jump.isPressed()) {
            player.hasBufferedJump = True;
            player.jumpBufferCountdown =
jumpBufferTime;
        }
        // Take note on how this piece is outside the
        "jump is pressed" section
        if (player.hasBufferedJump) {
            player.jumpBufferCountdown =
player.jumpBufferCountdown - dt;
        }
        if (player.on_ground) {
            if (player.jumpBufferCountdown > 0.0) {
                // Jump
                player.jumpBufferCountdown = 0.0;
                player.hasBufferedJump = False;
            }
        }
        // ...
    }
```

Code: Jump buffering example

## 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.



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.

```
class Player{
    bool coyote_time_started = False;
    float coyote_time = 0;
    bool onground = False
    bool has_jumped = False
    // ...
    function update(float dt){
        // ...
        if (onground){
            // Do stuff when player is on ground
            // ...
        }else{
            if (not has_jumped){
                // Player is not on ground and has
                not jumped, the player's falling
                if (not coyote_time_started){
                    coyote_time_started = True;
                    coyote_time = 5;
                }else{
                    coyote_time = coyote_time - dt;
                }
            }
        }
    }
}
```

```

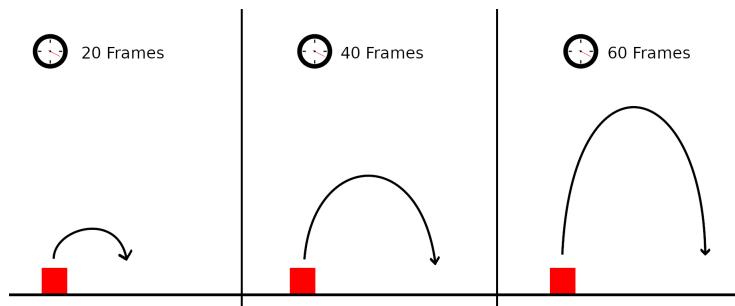
function jump() {
    // This function takes care of jumping
    // ...
    if (coyote_time > 0) {
        // Do Jump
    }
    // ...
}

```

Code: Coyote time code example

## 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.



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.

```

class Player{
    const float JUMP_VELOCITY = -12.0;
    float y_speed;
    // ...
    function onJumpKeyPressed(){
        /* The jump key has just been pressed

```

```

        (doesn't account the jump key being
            pressed from previous frames) */
        y_speed = JUMP_VELOCITY;
    }

    function onJumpKeyReleased() {
        // The jump key was just released, cut the
y_speed so the jump is lower
        if (y_speed < JUMP_VELOCITY / 2){
            // The speed is higher than the cutoff
speed (in absolute value)
            y_speed = JUMP_VELOCITY / 2;
        }
    }
}

```

Code: Example code of how timed jumps work

## Screen Wrap

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

## Top-view RPG-Like Games

### Managing height

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

## Rhythm Games

### The world of lag

Welcome to the world of rhythm games, as with all new experiences we shall start with... the final boss: Lag.

Lag will be one of the most problematic things you will have to account for: things are not as easy as you may imagine when it comes to implementing a

rhythm game. Let's see how to account for it, and eventually how to limit its effect on the player experience.

## Input Lag

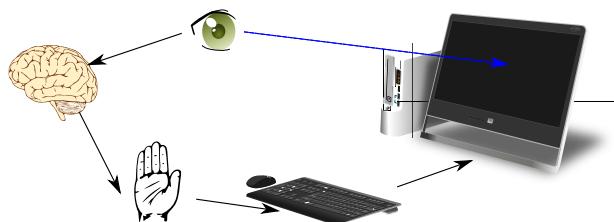
First of all: the ever-present “input lag”: there is a certain time window between the moment the user presses a button and the moment the game receives such input. In the middle we find electrons running at breakneck speed through our keyboard circuitry, through the cable, to the motherboard, then the CPU, input abstraction layers in our OS, and finally the input system in our game.

And we didn't reach the game update stage yet.

Also we are not even accounting for the reaction time (about one second) from when the player sees something on screen and when they react.

Input lag is something that we cannot avoid, but there are countermeasures, as we will see below.

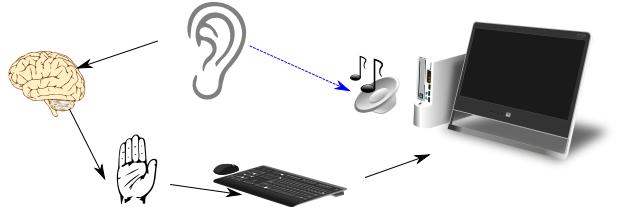
## Video Lag



Reference image for video lag

As with the input lag, there is also a not-negligible video lag. The game has to prepare the image, send it to the video card, the card has to render it, apply effects and then send it to the screen, where the liquid crystals (or whatever technology we will have in the future) will have the re-align to create the colored pixels on screen.

## Audio Lag



Reference image for audio lag

When the audio doesn't exactly match with the video, we talk about "audio lag", this has to be accounted for if you want to have a good rhythm game. In that case, there is a need to compensate for the audio lag, by starting each sound effect (or music) earlier or later by a well-defined amount of milliseconds.

## Lag Tests

When it comes to lag, it is really difficult to estimate how the computer will react to our game, so we need a metric that will tell us what corrections we need to apply.

Such corrections are estimated comparing video and audio to the input: this way we will keep everything synchronized to the player input, making the game feel tighter.

First kind of test is done "video vs. input", the player has to push a button when something on the screen happens (like pushing rhythmically with a dot changing color), this way we can account for the video lag, compared to the input. This means we will obtain a  $(\text{video} + \text{input})$  lag measurement.

The second test done is the "audio vs. input" one, the player has to push a button when a sound cue happens on their speakers/headphones (like pushing rhythmically with a beep), this way we can account for the audio lag, compared to the input. This way we will obtain a  $(\text{audio} + \text{input})$  lag measurement.

By simple math we can account for the "video vs. audio" lag, like follows:

$$(\text{video} + \text{input}) - (\text{audio} + \text{input})$$

$video + input - audio - input$

$video + \underline{input} - audio - \underline{input}$

$video - audio$

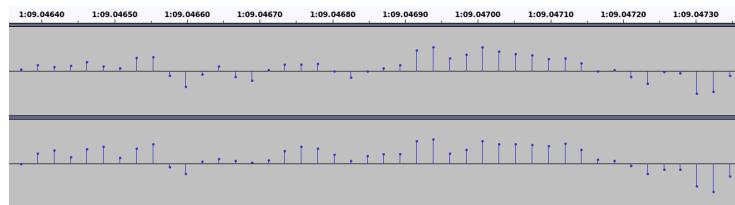
*[This section is a work in progress and it will be completed as soon as possible]*

## Synchronizing with the Music

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

### Time domain vs. Frequency Domain

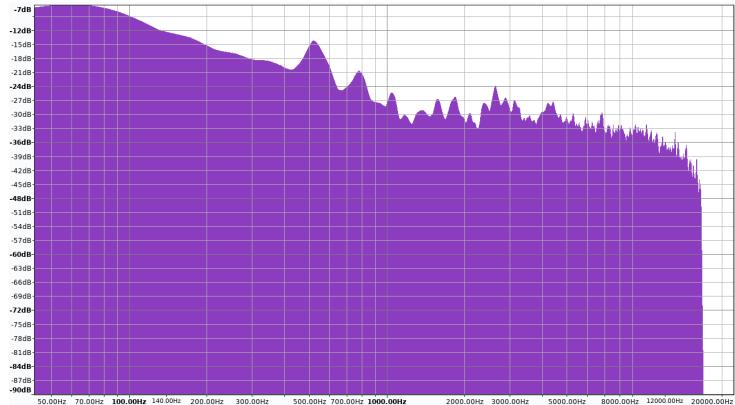
When we listen to music, we are essentially streaming a bunch of numbers as the time goes forward, so we can plot the amplitude of our waveform against time, as follows:



Plotting amplitude against time

In this case, when the time is the “independent variable” that we use to base our work, it’s said we’re working in *time domain*.

When we are working with games, we don’t really care about what will happen (music-wise) 5 minutes from now, instead we care about other things that are happening now. In that case, it may be interesting to work in *frequency domain*, which can look something like this:



Plotting frequency domain

We can switch back and forth between the two domains with “transforms”, the most used is the Fourier Transform, and one of the most used algorithms to do it on computer is “FFT” (Fast Fourier Transform).

## The Fast Fourier Transform

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

## Beat Detection

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

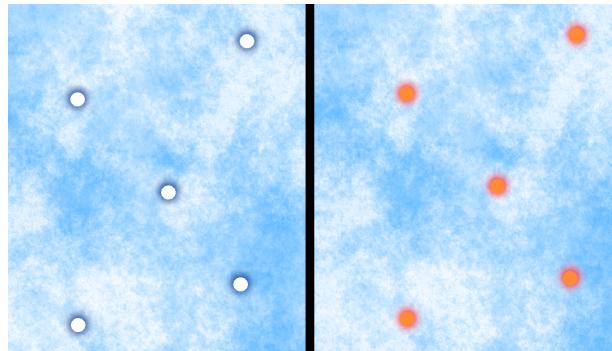
## “Bullet Hell” Style Games

The common definition of a “bullet hell” game is usually the one of a scrolling (usually space-themed) shooter with a very high level of difficulty and lots of enemy bullets on screen (hence the name).

## Bullets

When it comes to this kind of game, it is vital that the enemy bullets are **well visible** (as stated in the [shooters section](#) in the “game design” chapter), this

usually means that their color is brighter and has a lot of contrast with the background and the sprites on screen.



Example of how to better “highlight” bullets

Having “evident” enemy bullets makes the situation easier to assess, even when the situation becomes really chaotic. If you zoom out (or get your reading support farther from your eyes) you can see that the “non-highlighted” bullets (on the left side) tend to “blend in”, while the “highlighted” (on the right side) version stay visible.

To highlight bullets, you can use “complementary colors”, as shown in the [use contrast to your advantage](#) section.

### Tip!

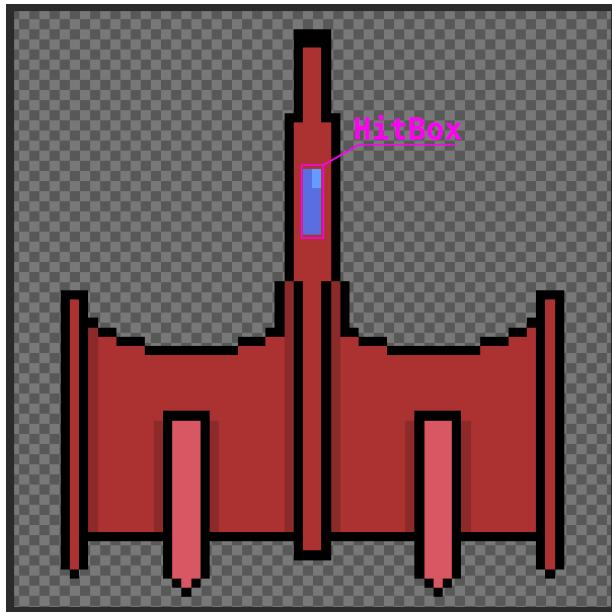
Bullet visibility is so important that in many games bullets are the last thing to be drawn before the player: this means they’re drawn over explosions, other enemies and your own bullets too.

If you let players lose sight of bullets by drawing graphical effects over them, the game will feel unfair.

## The Ship Hitbox

In the bullet hell genre usually the player ship’s (or character of some kind) hitbox is usually much smaller than the visible sprite, this makes the game a

little bit “easier than it seems”, but at the same time it doesn’t mean that the game is easy either.



An example of a Bullet Hell ship hitbox

In this image, the ship’s hitbox is limited to the cockpit, some games prefer some area that could be considered the “ship engine” while others just have a “core” of some sort.

Many games of the genre even make the hitbox a single pixel!

## Screen-clearing bombs

Another mechanic used in bullet hell games are “screen-clearing bombs”: these are used to rid the screen of the gigantic number of bullets on it, to give the player some breathing room.

In some games bombs may be also used to destroy small enemies and damage bigger ones. The screen clearing move can happen in many ways: the most common is just making the bullets disappear, but other games prefer turning the “destroyed bullets” into small collectibles that can give the player points.

## Clearing bullets on pattern changes

Some bullet hell games feature multi-phase bosses, where the boss changes attack strategy, and thus their bullet pattern and speed, at certain points of the fight (usually when reaching a certain amount of health left). This may create some issues to the player, since the new bullets may cover all “escape routes” willingly left by the previous bullets, thus making it impossible to not die.

A simple and effective strategy is clearing the screen of the enemy bullets automatically when the boss changes phase (sometimes transforming the bullets into collectibles for score), this will allow for a quick breather to the player, as well as a somewhat smooth transition to the new phase.

## **Find other chances to clear some bullets**

Some games find creative ways to clean up a screen cluttered with bullets: for instance some bullets can turn into collectibles when a pickup is touched by the ship.

### **Random Trivia!**

In ZenoDyne R powerups are real, “physical” objects, and as such they block incoming bullets, so they can be strategically used as a “shield”, and then pick them up at the last second.

Some games like to clear the screen (without giving out collectibles) at the beginning of a boss fight, to give a “clean slate” to start the boss with.

## **Turn enemy bullets into collectibles at the end of a boss fight**

An interesting form of bonus that is often present in bullet hell games is turning all the boss’s bullets on screen into collectibles at the end of a boss fight.

Since this genre of game gets progressively harder the more bullets are on screen, this small trick rewards players for being good at dodging, while

players who used screen-clearing bombs will have a smaller bonus.

## The “Chain Meter”

This is a mechanic used in many bullet hell games: the chain meter is a meter that gains value according to the number of enemies you kill in a certain amount of time and giving a score multiplier according to it.

This meter will automatically discharge with time, making it hard to keep up a high score multiplier, adding challenge to the player and rewarding them for being good at destroying enemies in large numbers fast.

Usually the meter has 5 levels, starting from level 1, when the meter is full, the meter “gains a level” and a score multiplier is applied accordingly. For instance we can have:

- Level 1: 1x multiplier (normal score)
- Level 2: 2x multiplier (double score for each killed enemy)
- ...
- Level 5: 5x multiplier

### Tip!

You can code the “discharge” so it is faster at higher levels. This will bring even more challenge at keeping the level high.

When the player dies, the counter gets completely emptied and thus the multiplier gets reset to 1x.

### Tip!

Alternatively, you can halve the level of the meter on player’s death.

## Managing the player's death

It is very common in the “bullet hell” genre to punish the player’s death with a strong cut at the ship’s power.

This has a problem: a player dying may spiral into a fully-fledged game over because the ship is now extremely underpowered compared to the stage the player died in.

A solution often used in this genre of games is having a dying player’s ship have a random chance of releasing a random number of powerups and bomb pickups on death, thus allowing the now-weakened player to “regain some strength” and continue their game.

## The Enemy AI

Probably the hardest part to develop in a “bullet hell” style game is the enemy AI and how to make the enemy bullets form a pattern that is hard but not impossible to dodge.

Since the gameplay is very hard to balance, this genre seldom sees a “procedural game” (an exception that comes to my mind is “Task force Kampas”, which features procedural levels, but handmade bosses).

A way to make the game feel more fair is programming the AI so it doesn’t shoot “on the way out” of the screen. Each enemy essentially has 3 phases in its AI:

1. Enter the screen
2. Fight (usually by shooting a single pattern or a continuous stream of the same pattern)
3. Exit the screen (or die)

When the enemy exits the screen, it should stop shooting and just orderly leave.

**Tip!**

If your game features enemy turrets, they should stop shooting when they are behind the player's ship: the player is already busy enough handling shots from the front. Shooting from behind makes the game unfair.

## Be fair to the player, but also to the computer

The title may be a bit awkward: how can you “be fair to a computer”?

Computers don't have feelings, but players do. And letting the players kill the enemy before the AI activates takes away all the challenge from the game itself: the enemies become cannon fodder when the player's weapons have enough “power” to instantly kill most of the enemy forces.

So to apply this, you should probably make the enemies invincible until they're fully on screen: this way the player sees them and doesn't kill them beyond the top of the game area.

## Inertia

Control is everything in a game where a pixel can be the difference between life and death of your ship/character. This means that heavy inertia does not play well with the “bullet hell” genre.

This also doesn't mean that you can't apply any, just be careful and don't go overboard. When a player dies because their ship went too far due to inertia, they will get mad at the game, and by transitive property, at the devs.

## Some examples

There are games that make the most of the “bullet hell” mechanics to give player more challenge, or risk/reward choice.

One game is “Touhou”, which has a “grazing” mechanic: if a bullet slightly grazes (but does not hit) your hitbox, you will see some sparks and get a bonus in points.

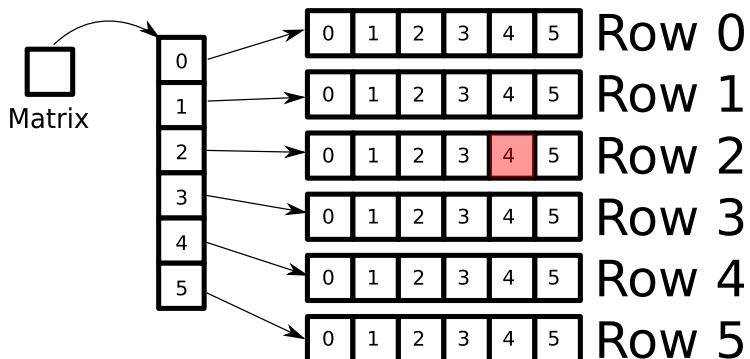
Another title that makes the most of giving the player a “risk vs reward” choice is Ikaruga, with its “polarity” mechanic. Your ship has two sides: black and white, each side is able to absorb (and so is also immune) to the bullets of the same color, but also does more damage to the enemies of the opposite color.

## Match-x Games

### 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.



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.

## 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:

```
function findHorizontalMatches() -> Tile[] {
    int matchLength = 0;
    const int minMatchLength = 3;
    Tile[] matches = new Tile[];
    for (each row in matrix) {
        Tile lastMatchingTile = null;
        for (each column in row) {
            Tile currentTile = matrix[row]
```

```

    [column].tile;
        if (currentTile == lastMatchingTile){
            matchLength = matchLength + 1;
        }else{
            if (matchLength >= minMatchLength){
                // We need to memorize all the
                tiles involved in the match
                for (each tile from matrix[row]
                [column-matchLength] to matrix[row] [column])){
                    matches.add(tile);
                }
            }else{
                // No matches, reset the
                counter and set the current tile as last matching
                matchLength = 1;
                lastMatchingTile = currentTile;
            }
        }
        // We need to account for the right-
        hand border corner case
        if (column == size(matrix[row])){
            if (matchLength >= minMatchLength){
                // We need to memorize all the
                tiles involved in the match
                for (each tile from matrix[row]
                [column-matchLength] to matrix[row] [column])){
                    matches.add(tile);
                }
            }
        }
    }
    return matches;
}

```

Code: Finding horizontal matches in a match-3 game

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:

```
function findVerticalMatches() -> Tile[]{
    int matchLength = 0;
    const int minMatchLength = 3;
    Tile[] matches = new Tile[];
    for (each column in matrix) {
        Tile lastMatchingTile = null;
        for (each row in column) {
            Tile currentTile = matrix[row]
[column].tile;
            if (currentTile == lastMatchingTile) {
                matchLength = matchLength + 1;
            } else{
                if (matchLength >= minMatchLength) {
                    // We need to memorize all the
tiles involved in the match
                    for (each tile from matrix[row-
matchLength] [column] to matrix[row] [column]) {
                        matches.add(tile);
                    }
                } else{
                    // No matches, reset the
counter and set the current tile as last matching
                    matchLength = 1;
                    lastMatchingTile = currentTile;
                }
            }
            // We need to account for the bottom
border corner case
            if (row == size(matrix[column])) {
                if (matchLength >= minMatchLength) {
                    // We need to memorize all the
tiles involved in the match
                    for (each tile from matrix[row-
matchLength] [column] to matrix[row] [column]) {
                        matches.add(tile);
                    }
                }
            }
        }
    }
    return matches;
}
```

## Code: Finding vertical matches in a match-3 game

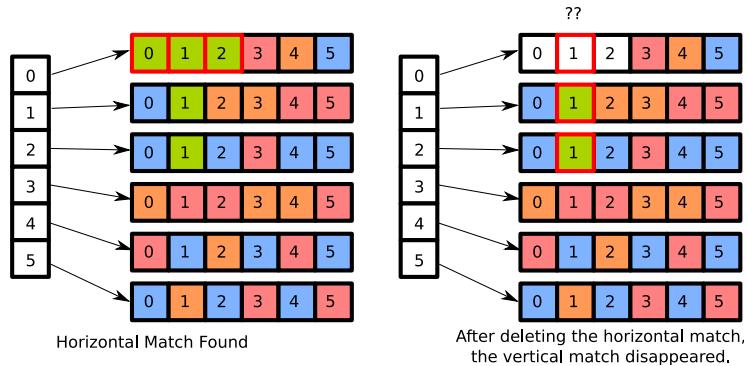
Both algorithms run in  $O(n)$ , where “n” is the number of tiles on the screen.

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).

### 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:



### 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.

## **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 multipier 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]*

## **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.

## **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]*

## **Scripted Cutscenes**

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

# **Part 6: Refining your game**

# 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.

## 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.

## 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.

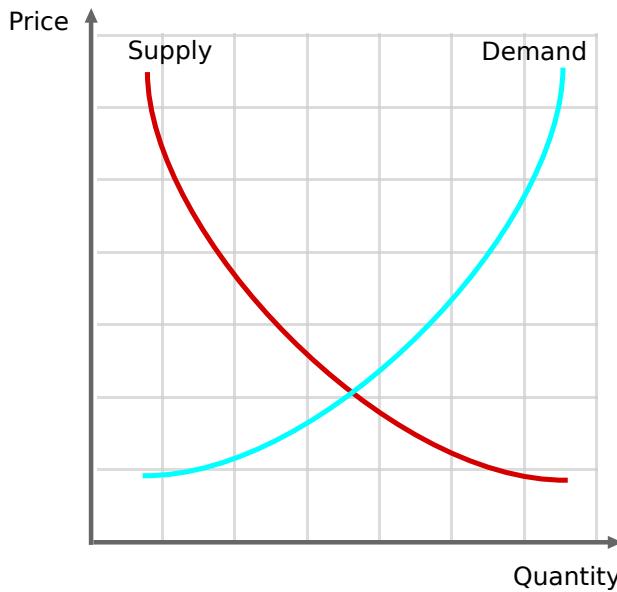
## Economy

Some games (not only MMOs) feature an “economy” side to their gameplay: this can prove to be something really difficult to balance without creating a virtual financial disaster.

This section will give you some basics to get things right.

## Supply and Demand

Every economy is (at least in part) governed by the laws of supply and demand, which can be graphically represented in the following graph:



A simplified vision of supply and demand

### Note!

This is an oversimplification of how the market (and the economy in general) works, just enough to keep you far away from the most common issues.

We can get some takeaways from such graph:

- If the demand is low (noone wants the product), suppliers will try to “boost it” by lowering prices;
- If the demand is high (many want the product), suppliers will try to earn more by boosting prices;

- If the supply is low (the product is rare), people will value it more (the price will be higher);
- If the supply is high (the product is really common), people will value it less (paying it less).

This also shows that artificially keeping the supply low will make the product feel more valuable, allowing to ask for higher prices.

Another thing to remember: money is a good too, and is subject to the same laws.

## **Money sources and sinks**

Any artificial economy is usually composed by 3 “components”:

- **Money Sources:** they create money from nothing, these can be quest givers, treasure chests and the like;
- **Money Sinks:** places that “destroy money”, these are NPC salesmen at the market (that create items from nothing), fortune machines, anything that takes or exchanges money for something else.

Sources and sinks are extremely important and should be carefully balanced, since an imbalance in the quantity of money created and destroyed can have catastrophic effects. Among those, uncontrolled inflation and deflation are the most prominent.

## **Inflation**

Inflation is a phenomenon where prices usually rise uncontrollably: this means that money “lost its value”.

This is usually due to the massive presence of money in the economy, so in a source/sink view, the money sources emit much more money than what the sinks can consume.

As a consequence fixed-price operations (like if you put “repair a weapon” at a fixed 50 golds) become incredibly cheap, while products in the market become prohibitively expensive.

In a supply/demand perspective, there is big supply of money which triggers little demand for it (since it's so common), while there is a big demand for products (thus raising the prices).

This may end with people having loads of money and noone accepting them for trades. Bartering may arise as an alternative to money.

## Deflation

Deflation is a phenomenon where prices usually have a drop: this means that money has “too much value”.

This has the exact opposite causes of inflation: there is too little money in the economy, so the money sources don't emit enough money and there are too many sinks that can consume it.

As a consequence fixed-price operations become extremely expensive (if you have 100 gold, paying 50 gold to repair a weapon may seem a lot), while products in the market become extremely cheap.

Again, in a supply/demand perspective, there is a low supply of money (making it more valuable), while demand is really high.

This can trigger “money hoarding” thus freezing the economy, sometimes bartering can arise as an alternative way to exchange goods without involving the “precious precious money”. Some operations that require a minimum amount of money may even get locked because of the little amount of money circulating.

## 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.

This section will just give a primer on the types of cheating we can find, since knowledge of something is the best weapon against it; so questions like “how to cheat” (or “how to hack”) are outside the scope of this book.

## **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.

## **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.

## **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.

## **How cheating influences gameplay and enjoyability**

### **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](#).

## Multplayer

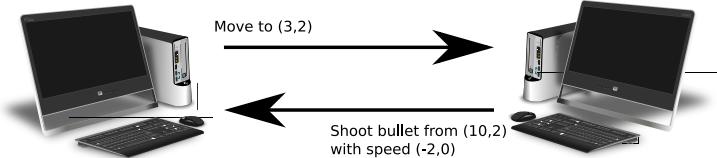
When it comes to multiplayer and “leaderboards”, cheating can be create some major damage to the game’s enjoyability. It is honestly disheartening seeing a level that has been completed in 0 seconds on top of the leaderboard, totally unreachable with normal gameplay.

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.

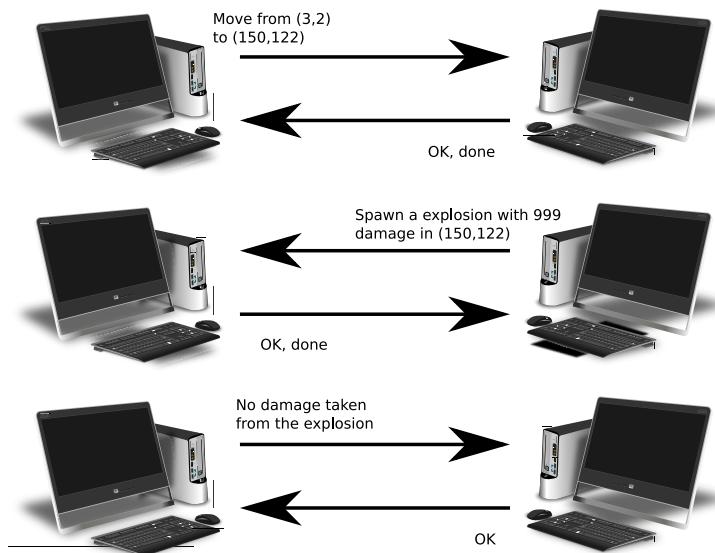
### P2P

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.



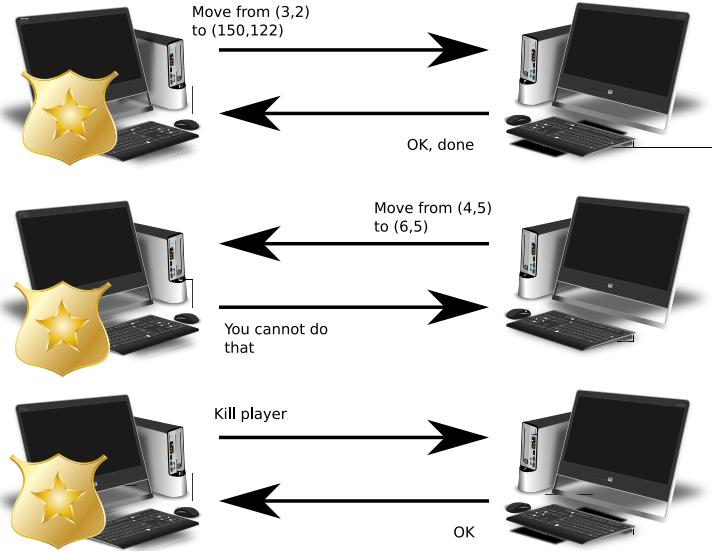
Example of a P2P connection

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.



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.

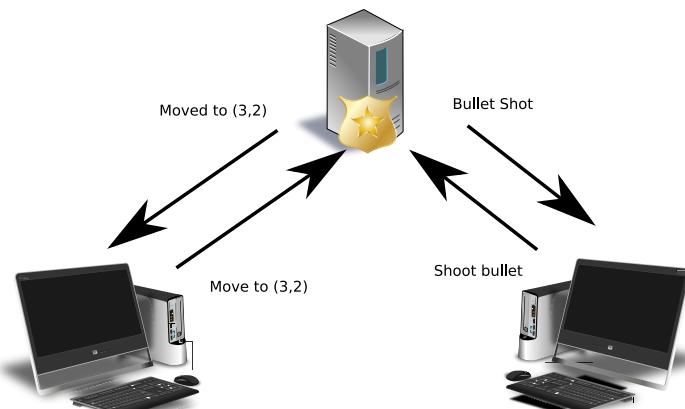


What would happen if one of the Peers had the authoritative game state

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.

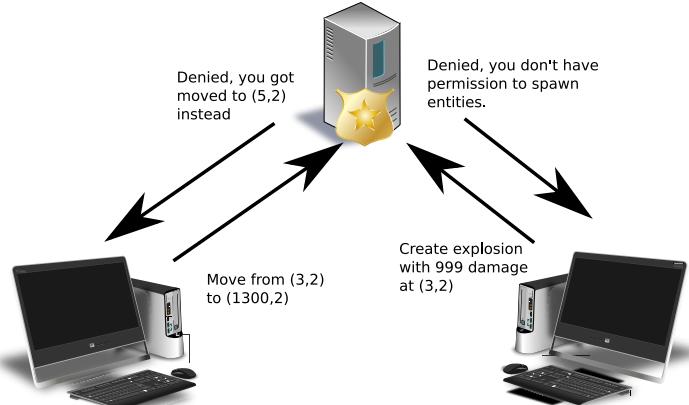
## Dedicated Servers

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”.



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.



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.

## Cheating protection

Protecting your game, leaderboards and community from cheating is hard and there are many ways to prevent it. Dedicated servers are one of the many ways, cheat engine detection (there are many commercial solutions) but none of these solutions is cheater-proof.

If your game has 1 million players (I hope it does!) and cheating has a 0.0001% probability of people cheating at it, you have a statistical certainty that there will be a cheater among them. Cheating will happen and usually cannot be completely prevented, but that shouldn't discourage you.

It should be hard to cheat at a game but, most of all, it should be harder to do it undetected.

So there are some other tools in your toolbox that you can use, for instance you can save a “lightweight” replay of the game session (if your game has a leaderboard), that way anyone who wants to enter the leaderboard will also send a replay of their gaming session.

This has multiple advantages: from the community side you have players can “learn tricks from the best”, but also can report who evidently cheats, because the replay would show it.

Usually these “lightweight” replays are done by recording the position of the player and its state, as well as the initial state of the game. Add the fact that the game is deterministic and you have the equivalent of a recording of the gameplay.

Even better, you can record the inputs of the player and see if “a simulation” done with those inputs validates against the positions and actions recorded on the replay: this way if someone modified their game to make themselves invincible or faster, the simulation would fail and the replay wouldn’t validate.

### Random Trivia!

It was recently discovered that the game “Trackmania” records inputs as well as the position of the vehicle, this allows the game to validate the replays against the most common forms of cheating

We can be sure that the simulation would be equal to a video because a game (as well as any program) is deterministic: given the same initial state and inputs, the game will always end up the same way. This is true even if random numbers are involved (that is why they’re called “pseudorandom”), see [“random numbers on computers are not really random”](#).

# Accessibility in videogames

Accessible design is good design - it benefits people who don't have disabilities as well as people who do. Accessibility is all about removing barriers and providing the benefits of technology for everyone.

---

Steve Ballmer

## What accessibility is and what it is not

Let's start with what accessibility is not: accessibility is not making a game playable by people with disabilities. To be precise, it is not only that.

Accessibility is an inclusive practice where you make sure there are no (or at least the fewest number of) barriers that prevent someone with interacting with your game.

Many times “good design” and “customization options” bleed into “accessibility” too, as we can see from the sections below.

## UI and HUD Scaling

This is one of those “customization options” that bleeds well into “accessibility”. Some people may like smaller Heads-Up Displays (HUDs) and general UI, to give more space to gameplay, while others may want to have bigger UI because of some visual impairment.

UI and HUD scaling is a great customization for any player, not only the ones who suffer from some kind of impairment or disability.

## Subtitles

This feature is really useful if you can't afford to keep the volume too high without disturbing someone (and don't have headphones). Another great use for subtitles is to expand the game's audience to non-native english speakers, who may have a bit of trouble with pronunciation.

Some games further expand their demographic by using speech in a certain language and enabling support for subtitles in another language, other times it can be used as a story ploy: in Rayman 2 the characters speak "Raymanian" (sometimes called Wandayē), a fantasy language which is subtitled in the language of choice.

Another use of subtitles is to describe world sounds for hearing-impaired users.

## Mappable Buttons

This is another one of those "customization options" that falls into the "accessibility features" too. Giving the player the possibility of changing their button mapping is essential to guarantee that a larger number of players is able to enjoy the game you created.

Modern engines usually include input handlers that already support mappable buttons, you just need to "transpose" such functionality in your own game by creating the right menus.

## Button Toggling

In some games, a player may have necessity to keep a button pressed to perform a certain action (like sprinting).

Some games allow buttons to work as "toggles", so one press enables sprinting, another press disables it. This can be both a functionality for "comfort" as well as a feature that allows the game to be more accessible to players with difficulty when it comes to muscle control.

## Dyslexia

One often overlooked issue when it comes to text is dyslexia. In this small section we will take a quick look at some accessibility options that can help with this problem.

## **Text Spacing**

A good start would be having the possibility of customizing the space between lines of text: this way the possibility of mixing up different lines of text is greatly reduced. This also helps other people who prefer a smaller UI, but at the same time find difficulty in reading very “condensed” text.

## **Fonts**

Another great tool could be font customization: there are specialty fonts dedicated to greatly reduce letter-swapping and confusion, thanks to letters that have a thicker bottom (to give a clearer baseline) and different letter shapes.

One of such specialty fonts is “OpenDyslexic” (available at <https://www.opendyslexic.org/>), which is free to use, being licensed under the SIL-OFL license.

## **“Slow Mode”**

Having an option that allows to slow down the game is an authentic boon for anyone who wants to enjoy a videogame but has some issues when it comes to reaction time or muscle control, but this makes the game more enjoyable for audiences who may not appreciate a high level of challenge but still want to play your game.

## **Colorblind mode**

When it comes to color, everyone sees colors differently but others may not be able to distinguish between certain colors at all: that’s where a colorblind mode comes into play.

Having the game palette adapt so it's easier to distinguish colors for colorblind users is definitely one of those accessibility options that may require a bit more work but could make some players really happy too.

## No Flashing Lights

When it comes to photosensitivity, (very dangerous) seizures are not the only outcome possible: some people may get really strong headaches from seeing intense visual effects and flashing lights.

Having an option to disable or tone down flashing lights and intense (sometimes even overbearing) visual effects can make a game a lot more enjoyable to someone who has some kind of sensitivity to light.

## No motion blur

There are some people like motion blur, some who don't and some who can't physically stand it: making motion blur a toggle can be both a quality of life improvement as well as something that can help people who have issues with sight.

## Reduced Motion

Some people may be particularly sensitive to cameras or texture patterns that involve lots of motion, this could cause discomfort in the player or even full-blown motion sickness. Having an option to reduce motion (like avoiding certain camera rotation mechanics or making some patterns static) can really help users enjoy your game without having the game "hurt" them.

### Random Trivia!

Sonic 4: Episode 2 has a level (precisely "Death Egg. Mk. II") which has a really cool mechanic where the camera follows the ground you're riding on, rotating with it. This is a really cool effect, but this is one of

the effects that could cause motion sickness. A trigger to disable the camera rotation (and maybe take a slight hit in “realism”) could help a lot, even though it could introduce some control scheme challenges.

## Assisted Gameplay

Having an “aim assist” option that allows for the crosshair to “snap to an enemy” when it is “close enough” it’s great for people who have problems with coordination, but it’s also great for players who prefer to play with a controller instead of a mouse.

Any kind of “assisted gameplay” can help both people with and without disabilities.

## Controller Support

Supporting only one certain type of “control device” is not great for accessibility; some people may prefer or need a controller of some sort, be it a gamepad or something more sophisticated, having support for “alternative inputs” is always a great idea.

## Some special cases

A special mention goes to the VR game “Moss”, where a character (named Quill) communicates both emotionally and gives clues on the puzzles using the American Sign Language (ASL).

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

# Testing your game

The bitterness of poor quality remains long after the sweetness of meeting the schedule has been forgotten.

---

Anonymous

When you want to assure the quality of your product, you need to test it. Testing will 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!

## When to test

### 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.

## 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.

## 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.

## 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.

## 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.

## 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.

### 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):
    // This function takes two numbers and sums
    them
    return a + b

function test_sum():
    // This function tests the sum function
    int result = sum_two_numbers(2, 2)
    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.

## 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.

## 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.

## 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.

## 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 integration and delivery) to execute them at each push or at regular intervals.

## Playtesting

Automated testing won't be able to help with how a game "feels" to the player, for that you need a thorough play testing strategy. Here we will talk a bit of different strategies that can be mixed and matched to get the best out of it.

### In-House Testing

The first sessions of play testing should be done in-house, with a dedicated play testing team that has great reporting capabilities, and the product should already include tools that allow for quick reporting of bugs and issues that arise, as well as a good logging system set to its `DEBUG` level for maximum detail.

Close collaboration with the testing team is vital for a good game to be released, instead of seeing them as "the ones that give you more work", try looking at them as "the ones that will ensure your game gets a lot of praise".

### Closed Beta Testing

Happening after the in-house testing, usually done with multiplayer games, the "Closed Beta" phase is done with a selected group of players that try the game and report each and every issue with it, as well as bugs.

The product should have an easy way to directly report bugs and issues from inside the game itself, with the possibility of attaching a detailed log with the "ticket".

### Open Beta Testing

Differently from the “Closed Beta” phase, and usually coming after that, the “Open Beta” phase doesn’t have a hard limit on the number of players that can take part to the beta testing.

The product should have the same characteristics that are used in the “closed beta” phase, plus possibly a higher degree stability.

Open Beta is usually done to test the robustness of the network system at higher loads, and having a possibly large (maybe larger than expected) player base can be a real test for the infrastructure.

## A/B Testing

A/B testing is a particular kind of testing that doesn’t involve the “solidity of the game”, as much as the enjoyability of some of its features. In A/B testing users are randomly presented with one of two versions of a certain feature; with a slight difference (usually a single variable that affects the experience is changed) between each of the two “versions”.

A/B testing is normally used in an user experience research setting, but it can be used in game development too, to see which version (usually called *variant*) is better suited for the game, or even prepare a “segmented strategy” where one of the two variants could find place in a certain situation (for example a “simplified control scheme” vs a “full control scheme”).

# 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

## Profiling your game

### 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.

### 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
- 90 FPS: Used mostly in VR games
- 144 FPS: Used in more modern, high-refresh rate monitors
- 240 FPS: Used in the most recent high-end games and monitors

## **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]*

## **First investigations**

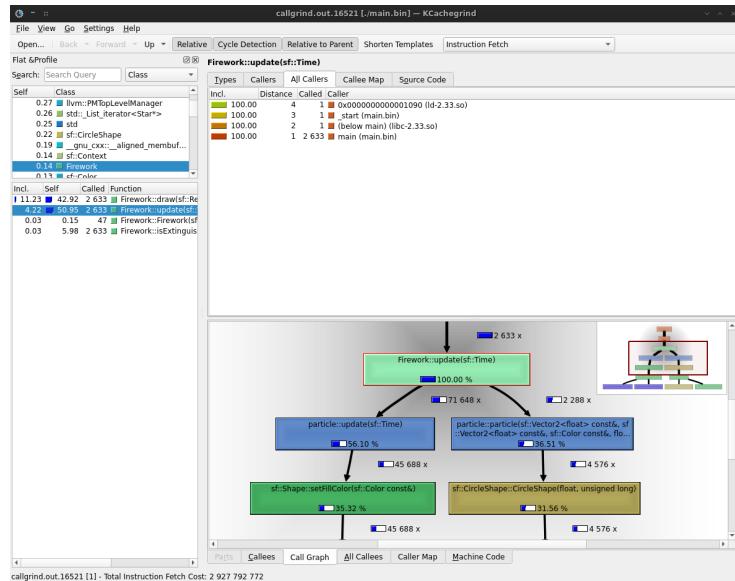
First of all, we need to understand what is the bottleneck of your game: check your task manager and see how your game is performing.

### **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.



Using Valgrind's Callgrind tool and Kcachegrind we can see what is bogging down our game

## Is your game overloading your GPU?

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.

## Is your game eating up more and more RAM as it's running?

Your game starts well enough, but after just a few minutes it starts slowing down and becomes choppy. Your may have a memory problem at hand.

If your game supports windowed mode, keep your task manager (or “top”/“htop”/“bpytop” if you're on Linux) open and look at your game's process: does the memory used by your game increase as you're playing it?

If so, you may be having a so-called *memory leak*: somewhere during its running cycle, your game forgets to clean up something, which stays resident in memory until your game closes. The result, after creating and

deleting a lot of entities and leaving a lot of *garbage* behind is that the total memory used increases.

This is especially common in languages like C++, where there is no automatic “garbage collecting” and having cases of so-called *unreachable memory* can really mess up your memory usage.

### Pitfall Warning!

Some people call unreachable memory cases “*dangling pointers*”, but technically they are two different (and opposite) things.

Check the glossary for more information.

If you suspect a memory leak, you may want to take a look at these sections:

- [Entity Cleanup and Memory Leaks](#)
- [Using memory analyzers to detect leaks](#)
- [Resource Pools](#)

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

## 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.

## Working with 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.

## 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]*

## Reduce the calls to the Engine Routines

Some engines have routines that introduce sanity checks, logic optimizations and more, and calling such routines more than necessary can burden your game's performance, even worse when you're calling them per-frame.

If you want to move a character diagonally both up and right, don't do this:

```
function update(float dt) {
    Vector vector_up = (0, -1);
    Vector vector_right = (1, 0);
    // ...
    characterController.Move(vector_up * dt);
    characterController.Move(vector_right * dt);
    // ...
}
```

Code: Double Engine Movement Call

As all the sanity checks in the `Move` function will be executed twice per frame (since we're in the "Update" function). Instead you should get the resulting movement vector first, and then use the `Move` function only once:

```
function update(dt) {
    Vector vector_up = (0, -1);
    Vector vector_right = (1, 0);
    // ...
    Vector total_movement = vector_up +
vector_right;
    characterController.Move(total_movement * dt);
    // ...
}
```

## Code: Single Engine Movement Call

This way instead we're doing sanity checks and related operations only once, moving the character in its final position without wasting resources.

## Entity Cleanup and Memory leaks

One of the biggest scourges in software development (and an even bigger one in game development) are memory leaks: the program allocates memory but doesn't release it properly.

Memory management (as well as any kind of “resource management”) can be summarized in 3 phases:

- Acquisition;
- Usage;
- Release.

This is especially annoying when languages that don't have automatic garbage collection (like C++) are involved, but it can affect any programming language. Memory management is hard, and we should always release any resource that we acquire as soon as we're done using it, but that's not always easy: for instance when loading and unloading levels is involved.

As mentioned before, this problem affects all languages, since some resources may be acquired by some “active code” that is actually never running, thus preventing the garbage collector from working as it should.

Besides “being careful” with your resource management, you can check for memory leaks by using specific tools.

## Using analyzers to detect Memory Leaks

When developing a game, there are a lot of tools that allow you to inspect your game and find possible memory leaks. Some are “static scanners” while other (usually called “dynamic testing tools”) require the game to be running.

## Static Scanners

These tools analyze the code without running it, checking the style and common bugs that can be inserted by mistake. An example of these static tools are “linters” (or linting tools).

Most of these tools are included in IDEs but some (like LLVM’s scan-build) are standalone.

```
scan-build: Analysis run complete.  
scan-build: 5 bugs found.  
scan-build: Run 'scan-view /tmp/scan-build-2021-01-17-201803-11375-1' to examine bug reports.
```

An example screen from LLVM’s scan-build

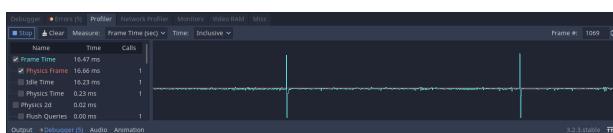
## Dynamic testing tools

Some tools require the game to be running, some general-purpose ones are used to find memory leaks (like Valgrind), while others have more specific purposes and are usually integrated into the engine.

```
penaz@PenazMW2: ~/lazur ➤ valgrind --leak-check=yes ./lazur  
==12912= Memcheck, a memory error detector  
==12912= Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.  
==12912= Using Valgrind-3.17.0 and LibEXE; rerun with -h for copyright info  
==12912= Command: ./lazur  
==12912=  
==12912== HEAP SUMMARY:  
==12912==     in use at exit: 864,094 bytes in 14,262 blocks  
==12912==   total heap usage: 387,765 allocs, 373,583 frees, 194,735,329 bytes allocated  
==12912==  
==12912== 24 bytes in 1 blocks are definitely lost in loss record 132 of 2,301  
==12912==    at 0x483FF3F: operator new(unsigned long) (vg_replace_malloc.c:417)  
==12912==    by 0x12A714: viewEditItem::viewEditItem(User const*, MultimediaItem*, QWidget*) (viewEditItem.cpp:120)  
==12912==    by 0x125988: mainWindow::mainWidget(User const*, std::vector<listUser>, std::all_of<listUser> const*) (mainWindow.cpp:38)  
==12912==    by 0x1263FC: mainWindow::dologin(User const*) (mainwindow.cpp:38)  
==12912==    by 0x134E96: mainWindow::qt_static_metacall(QObject*, QMetaObject::Call, int, void**) (mainWindow.h:134)  
==12912==    by 0x58FB3AF: ??? (in /usr/lib/libQt5Core.so.5.15.2)  
==12912==    by 0x134985: loginWidget::loginExecuted(User const*) const (moc_loginWidget.cpp:170)  
==12912==    by 0x125413: loginWidget::checkUser() const (loginWidget.cpp:57)  
==12912==    by 0x134694: loginWidget::qt_static_metacall(QObject*, QMetaObject::Call, int, void**) (loginWidget.h:134)  
==12912==    by 0x58FB3AF: ??? (in /usr/lib/libQt5Core.so.5.15.2)  
==12912==  by 0x4ACE4E2: QAbstractButton::clicked(bool) (in /usr/lib/libQt5Widgets.so.5.15.2)  
==12912==  by 0x4ACE76B: ??? (in /usr/lib/libQt5Widgets.so.5.15.2)  
==12912==
```

A screenshot from Valgrind, looks like we have a memory leak here

These more specific tools can track the FPS, memory as well as the calls done to each function, allowing you to track down what is bogging down your game.



A screenshot from Godot’s profiler

## Resource Pools

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

## Lookup Tables

Inside older games, where CPU cycles were at a premium, a widely used trick to gain performance were “lookup tables”.

These tables would store the result values for certain expensive functions, given certain inputs, thus replacing the expensive operation with a lookup inside a certain data structure (which is usually really fast).

This has a tradeoff: you’re trading CPU time for Memory space, since the lookup tables are meant to stay into RAM.

In modern games instances of lookup tables are as rare as hens’ teeth, but it’s an interesting historical view over some older forms of optimization.

## Memoization

Memoization (sometimes known as “tabling”) is an optimization technique that consists in saving the result of an expensive function, as well as the function’s arguments for later calls: this way when the same arguments are passed to the function, you can return the stored value instead of performing the calculation again.

This is due to the fact that functions are deterministic, so if you have the same inputs you will always receive the same outputs: this allows to minimize expensive computations at the expense of memory.

Obviously this technique can’t really be applied to functions that make use of pseudo-random numbers and connected functionalities, because memoization would completely void such randomization.

Memoization is usually implemented via decorators that check if the arguments passed are inside a defined data structure (usually a hash table): if there is a hit, the result is returned immediately, if not the original (expensive) function is run and its result is memorized in said structure.

Memoization should be used only on functions that are:

- Expensive
- Called often with the same arguments

If we start using this technique on all functions, we may end up with a software that occupies a lot of memory without any significant speedup.

## Approximations

Many times when developing games we don't need to have a value that is precise to the 10th decimal digit, that's where approximation comes into hand.

A prime example of approximation was used in Quake III Arena, via the algorithm known as "Fast Inverse Square Root". Back in 1999 calculating the inverse square root of a number was an expensive calculation for the CPU, so the developers decided to create an algorithm that would calculate an approximation quickly.

This was done by playing around with the floating point low-level structure and using a "magic constant" (`0x5f3759df`) to create a good "first guess", after that a single iteration of the [Newton-Raphson Method](#) is applied to refine the guess.

This proved to be faster than directly calculating a normalized vector (which uses a square root and a division, expensive at the time) and also faster than using a lookup table. The algorithm proved to be slower (and less precise) than the dedicated SSE instruction in the newer x86 CPUs.

## Eager vs. Lazy Evaluation

Lazy objects are yet another possibility when it comes to optimization, with some drawbacks: you create an object but the calculations related to its state are performed when the object is first used, instead of when it is constructed.

This can be really useful when you have a great quantity of items that you are iterating through, one at a time, but don't need the whole collection at hand at once. When it comes to collections, lazy objects help saving memory at the cost of more CPU cycles while the game is running.

In some languages, this concept is abstracted in a language feature (like “generator expressions” in Python), while in others you'll have to work a little bit harder to get them.

Let's take an example, we have a custom object that contains a reference to a list of numbers: when we iterate through this object, we want it to return the numbers saved, halved.

### Note!

What follows is just a didactic example, but should be simple enough to understand the difference between “eager” and “lazy” objects.

## Eager approach

The eager approach is to take the list of numbers, create a second list inside our object with the numbers halved: this will make sure that the values are always ready and readily available, but will consume more memory. Here's the example:

```
class EagerObject{
    int[] halved_numbers;

    constructor(int[] numbers) {
        // Prepares the halved numbers list
        for (number in numbers) {
```

```

        halved_numbers.append(number / 2);
    }
}

getObject(int index) -> int{
    // Returns the pre-calculated object at
the requested index
    return halved_numbers[index];
}
}

```

Code: An eager object

## Lazy approach

If we know that we are working with millions of values, and we are going through them kind of rarely, saving all the halved values in RAM may not be a good idea. This is where lazy evaluation comes into play: instead of memorizing the value in RAM, we calculate it on-demand. Here's the example:

```

class LazyObject{
    int[] numbers_reference;

    constructor(int[] numbers){
        // Saves the original list (possibly as a
reference)
        numbers_reference = numbers;
    }

    getObject(int index) -> int{
        // Calculates the halved number on-demand
        return numbers_reference[index] / 2;
    }
}

```

Code: A lazy object

## Tips and tricks

## Be mindful of your “updates”

It is a common mistake among new game developers of putting the whole game logic inside the engine’s `update()` method: this will eventually bog down the game and create inconsistencies when the framerate varies.

Input should be handled in your engine’s event-based input system (very rarely you will need to check the keyboard status inside the `update()` method), also you should absolutely take advantage of your engine’s facilities when it comes to managing how the game updates.

For instance, Unity offers 3 update functions:

- `FixedUpdate()`
- `Update()`
- `LateUpdate()`

`FixedUpdate()` is executed with the Physics engine, so here is where you should apply forces, torques and any other physics-related function. Being run with the physics engine, this function may be called zero, one or more times per frame.

`Update()` is your run of the mill update function, it is always executed once per frame, without fail. This is used for other kinds of updates, if you do physics operations here the results may be inconsistent (since it doesn’t run in sync with the physics engine). You can still move objects that are not tied to physics.

`LateUpdate()` is a utility function that is run once per frame, after the `Update()` function. This is useful for all kinds of operations that would require the `update()` calculations to be completed.

## Dirty Bit

Not all entities in your game need to have their state updated all the time. Continuously updating all entities’ internal state can be really costly in terms of game performance.

A quick way to make your game lighter on resources (and thus more performing) can be putting a boolean check at the beginning of the update function, checking if the object really needs to have its internal state updated.

A possible example could be the following:

```
class Player{
    Vector speed = Vector(0, 0);
    bool needs_update = False;
    // ...
    function input() {
        // ...
        if (right_key.is_Pressed) {
            speed = speed + Vector(1, 0); // Move
right
            needs_update = True;
            // ...
        }
        if (up_key.is_Pressed) {
            speed = speed + Vector(0, -100); // Move up (jump)
            needs_update = True;
        }
        // ...
    }

    function update(float dt) {
        if (needs_update) {
            // Do Update instructions
            // ...
        }
    }
}
```

Code: Example on how to optimize entities with a dirty bit

If your code is well-done, you won't have issues like animations freezing, because those will be separated from the "update routine", since the animator will chug along its frames when requested by the `draw` function.

## Far-Away entities (Dirty Rectangles)

Another way to optimize your game performance is not updating entities way off screen: this is also a technique used in the game Minecraft, where entities are frozen when you are far away from them, to save on resources.

A possible idea would be having an “updatable rectangle” (sometimes called “Dirty Rectangle”), bigger than the screen, and only the entities inside such rectangle will be updated.

This could create some issues when it comes to games that have their challenge deriving from entities updating in sync with each other, thus if we implement this “updatable rectangle” one or more entities would fall “out of sync”, possibly making beating the level impossible.

In that case we may just put out an exception (where certain entities are updated no matter what) or divide our level into smaller “rooms” that are instead entirely updated all the time.

## **Tweening is better than animating**

Animators and animation frames are performance-hungry and should absolutely not be used in all those situations where you can instead use inbetweening techniques.

This means that frame-by-frame animations should not be used when taking care of moving UI parts: if you want to slide a piece of UI (take for instance a drop-down terminal from a “computer hacking” game) from the top, you can just tween its position and save a lot of memory.

Remember that Tweening doesn’t apply only to positions, you can tween any property of a game object.

So a quick way you can optimize your game, is removing all the unnecessary animations and replace them with tweening, your game will surely benefit from that.

## **Remove dead code**

There are many definitions for “dead code”, some use the “unreachable code” definition (for instance code placed after a “return statement”) some use a more extensive definition.

I like to think of dead code as “wasted code”, which is:

- Anything that happens to be written after a “return statement” in a function: return statements are used to give control of the program back to the caller of a function, so this code will never be executed;
- Unused variables: variables are allocated in memory, require calculations and CPU cycles, if not used that’s just a waste;
- Unused code: complete functions that are never called are a waste of memory (because they may be loaded in RAM) and of disk space, making the executables bigger;
- Debug code: sometimes we need to write code to debug other code, this code may end up being part of a “release version” and weigh it down.

You should be careful when optimizing out dead code, even more when you are dealing with functions which result is not used: those functions may change some global state (or change stuff by usage of *side effects*<sub>g</sub>).

## Non-Optimizations

In this small section we take a look at some alleged “optimizations” that actually do nothing (or close to nothing) for our game’s performance.

### “Switches are faster than IFs”

Some people allege that using “switch” statements instead of “if” statements is bound to optimize the game. This is an overstatement, and we can prove it with a simple test.

Let’s create two C++ listings, like follows:

```
#include <iostream>
using namespace std;
```

```
int main() {
    for (int i = 0; i < 10000000; i++) {
        int x = rand() % 5;
        if (x==1) {
            cout << "One" << endl;
        }else if (x==2) {
            cout << "Two" << endl;
        }else if(x==3) {
            cout << "Three" << endl;
        }else if(x==4) {
            cout << "Four" << endl;
        }else if(x==5) {
            cout << "Five" << endl;
        }
    }
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 0; i < 10000000; i++) {
        int x = rand() % 5;
        switch(x) {
            case 1:
                cout << "One" << endl;
                break;
            case 2:
                cout << "Two" << endl;
                break;
            case 3:
                cout << "Three" << endl;
                break;
            case 4:
                cout << "Four" << endl;
                break;
            case 5:
                cout << "Five" << endl;
                break;
        }
    }
}
```

```
    return 0;  
}
```

These pieces of code will be compiled without any optimization, using G++, using the following command:

```
g++ -Wall -Wextra -Werror -O0 filename.cpp -o filename.bin
```

Where “filename” is replaced by the source name, then each file will be executed using the “time” linux command, like follows:

```
time ./filename.bin
```

Below we can see the results for both the codes:

```
Two  
One  
./if.bin 3.94s user 14.39s system 95% cpu 19.166 total
```

Time taken by the IF code

```
Four  
Two  
One  
./switch.bin 3.84s user 14.22s system 95% cpu 18.915 total
```

Time taken by the Switch code

We can see a difference of just around 0.25 seconds, over 10 Million iterations. If you changed an equivalent IF statement for a Switch statement, you would earn a quarter of a second every 46 hours of gameplay at 60fps.

The right choice is the simply choose the structure that lets you have the most readable code: the more your code is readable, the easier it is to understand; the easier to understand, the lower the probability that there is a bug in there (or a performance hog of some sort).

## Blindly Applying Optimizations

There rarely is something more wrong you can do when optimizing than blindly applying optimizations without considering your application context.

Using resource pooling in an environment with limited memory (but plenty of CPU power) can prove a disaster: it's better to instantiate and destroy objects in such case.

Sometimes animators can be faster than LERPing/Tweening, mostly when you have to tween objects with multiple children: tweening would create a lot of CPU-bound calculations for the new position and size that will make the whole thing bog down and get choppy.

The only thing you can do is think first and try later: this book can give you some suggestions, but nothing should be taken at face value. Remember the context your game is working in and **do not treat all platforms like they're the same**: WebGL is different than Console which is different than Mobile.

# **Part 7: Marketing and Communities**

# Marketing your game

Here's my whole marketing idea: treat people the way you want to be treated.

---

Garth Brooks

## An Important Note: Keep your feet on the ground

Let's be honest: the "indie success stories" you see everywhere are going to give you a false sense of hope, that "there is space for everyone in the industry".

I'm going to break it to you: if you're in for the money, you already lost.

The reality is that many game developers are horribly underpaid and many projects failed and many others are going to fail.

If you're in for the sake of creativity, to make something you want to make, seeing your project perform "under par" will hurt less, if at all. Your first project won't be a best-seller, but doing something you enjoy will make it better.

This book is meant to teach the basics of game development as a way to channel your creativity, and to help you understand the pitfalls and mistakes that can keep you from showing your best product, in the best possible way.

## 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, 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 genre 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.

## 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.

### 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.

### 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.

### 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.

## 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.

## 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.

## **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.

## **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”).

## 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.

### 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.

## 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”的 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.

## 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<sub>g</sub>*.

There have been cases of DRM systems installing themselves as *rootkits<sub>g</sub>*, 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.

## **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.

## Free-to-Play Economies and LootBoxes

Let's face the most controversial topic, here we go.

### 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.

## **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.

## **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.

## 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.

## 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.

### 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.

## **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.

## **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.

## **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.

## **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, ...)

## **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.

# Engagement vs Fun

In the games industry we can sometimes come across the terms “fun” and “engagement” in a way that makes them seem like synonyms, but they definitely aren’t.

Let’s start with the hardest to define: fun. Fun is really subjective, as humans each one of us has a radically different concept of “fun”. The Oxford English dictionary comes to help us, in a way:

Light-hearted pleasure, enjoyment, or amusement; boisterous joviality or merrymaking; entertainment.

This is a really broad definition, but the concept of “fun” itself is really broad and complex, and cannot become more than a few common points that we can define across games, making “designing fun” a form of art, instead of an act of engineering.

“Player engagement” is completely different, it has a well-defined metric: play time.

It is possible to artificially extend the players’ play time by tweaking the difficulty so that a player becomes frustrated enough to keep trying, but not enough to drop the game, occasionally adding a prize to keep the player gaming on (no matter how well-tweaked your experience is, without payoff players will leave sooner or later).

Is the player having fun when the experience is “tweaked” as such? Probably not, but the game is tailored so that the play time is longer.

The fun goes down, but the play time goes up. Fun and engagement are not the same thing.

“Player engagement” (and thus “play time”) is one of the metrics that is used by big publishers to analyze and hook players into in-game economies and lootboxes, the more the player engages with your game, the more you can “advertise” lootboxes and in-game economies. These metrics take a bigger role when the game takes the form of a “live service”.

# Streamers and Content Creators

When you're developing or publishing a videogame, you should not underestimate the power of so-called “content creators”: streamers and youtube stars, small or big, can turn your game from a niche project right into a cash buster.

Let's take some examples of games that, even though great on their own, would not be the same without streamers:

- **Minecraft:** this game is great on its own, but having a big number of streamers covering it definitely helped its growth, so much that Mojang got bought by Microsoft Corporation;
- **Fortnite:** even though pushed by a big company like Epic Games, this game benefited from the streaming community a lot;
- **Fall Guys:** published by Devolver Digital, this game cashes in on the “free for all/battle royale” genre, while giving a quirky and funny twist to it. Being a quirky game on its own right makes it an ideal game to stream, which definitely helped its sales.

I want to bring up one last example before delving into the nitty-gritty of the Streamers/Developers symbiotic relationship: **Among Us**.

“Among us” is a game that went from “unknown” to “cash buster” in a few weeks thanks to streaming. It got its “prime time” in 2020, after streamers brought it to light: it’s a multiplayer, “Town of Salem”-like experience based on deception.

The thing is, this game was released in 2018, and it was not doing too well on its own, until streaming came in the picture. Being a very interactive game, this made it ideal for collaborations between streamer friends, striking a nice balance between suspect, suspense and strategy. After streamers came into play, the sales exploded.

The Content Creator/Developer symbiotic relationship is one of the best win/win situations that can happen in the videogame development, let's see how.

## The game developers' side

For a game developer, having a game covered by a streamer has one big advantage: **free advertisement**.

Some big streamers can gather between 15.000 and 35.000 viewers *alone*, giving your game a huge boost in coverage. Most of the times the game is bought by the streamer themselves, with their own money, so you already sold a copy, but its worth is a lot more than meets the eye.

This boost in advertisement has consequences, if the game looks fun people will be encouraged to buy it, resulting in a surge in sales. The more the people who know about the game, the more it spreads via “word of mouth”, furthering your sales.

There is no “traditional advertising” that can bring that much attention to your game.

## The Streamers' side

For a streamer, having a new game is definitely advantageous: it's fresh new content.

Fresh content keeps a streaming channel interesting, engaging the public, which in turn generates advertising revenue.

Such revenue can be spent by the streamer for better gear, but also new games which can bring even more fresh content to the channel, thus keeping viewer engagement high.

## Other entities and conclusions

This “symbiosis” doesn't limit itself to game developers and streamers: advertisers earn because their products are exposed, other game developers earn money by having their products bought to bring new content, gear manufacturers (keyboard, mice, monitors, cameras, lights, green screens,

...) get revenue from streamers looking to improve the quality of their streams. This is sector economic growth at its finest.

On October 22nd, 2020 the Creative Director at Google Stadia threw out the idea of making streamers pay a fee (or a revenue share) to game developers (or publishers) for the content they stream. Given what we just saw, and the great advantages it brings, having a “streamer tax” would remove content creators from the video game environment, nullifying all the good that comes from them.

Small streamers would not be able to pay such “tax”, bigger ones would be willing to change their content (moving away from videogaming) to not pay such “tax” and not suffer many losses: people are watching streams because they like the streamer’s behaviour, reactions and jokes, as well as gameplay style and technique. If people watched streamers and content creators for the games they play, views and subscriptions would not be as stable as they actually are.

It would be changing a system that works for one that will, in the most absolute way, not work.

One of the best things you can do as a game developer is harnessing the power of the streaming community: reach out to some small streamers, give out free keys, offer review copies, give out preview copies of DLCs, be friendly and supportive. Small streamers may become big ones one day, and supporting mutual growth is a really satisfying experience.

# Keeping your players engaged

There is no power for change greater than a community discovering what it cares about.

---

Margaret J. Wheatley

Having a solid story and great game mechanics is only part of your objective: you want your players to talk about your game or play it for as long as possible, this can be done in two main ways: **communities** and **replayability**.

## Communities

Coding a game is just part of the whole game development ecosystem, engaging your community and having a fruitful exchange of ideas and knowledge can be an amazing resource to keep your game inside players' minds.

## Forums

Forums (also known as “boards”) are websites where people (in our case players) engage in discussions on certain matters (fittingly called “topics”).

Forums can be places where our players can share their fan work, discuss about the game and give suggestions.

Forums can be really useful a makeshift “service desk”, where players can report bugs and any kind of weirdness your games may present to them.

## Wikis

Wikis are a collaborative knowledge source where people write different pages about a certain matter.

In our case having a wiki about our game can contain various pages talking about:

- Characters
- Levels
- Bosses
- Enemies
- Walkthroughs

The great thing having many players collaborating on the same matter is that someone may be able find something someone else missed, helping to build a better understanding of the game.

Searching for more information about a game can be a kind of game itself, building and strengthening a community.

## Update Previews

Beta channels and update previews can be an incredible way to engage your most faithful fans as well as:

- **Give you more testing ground:** having many players test your game can give you further testing than any small team and also let you test server load;
- **Give you some market insight:** having some different point of view can give you some insight on whether your players will like the new features you're preparing or not.

## Speedrunning

When your game has a decent success (or even if it doesn't!), there may be someone who wishes to challenge themselves into completing it as fast as they possibly can, sometimes with the aid of glitches.

The speedrunning community can really change your game from “obscure” to “well known”, showing it at events and gatherings, making a fun show out of it, for better or for worse (hey, not all donuts come with a hole!):

even when your game is “not so good” it can still be a fun oddity that people may want to speedrun.

Giving your own game a “speedrun mode” can prove beneficial, lowering the entrance bar to speedrunners and letting them know that you are thinking about them and would like them to speedrun your game.



What a simple “speedrun mode” may look like

What could such “speedrun mode” contain? More features than you may think about:

- **A game-wide chronometer:** this will allow to register how long beating the game takes;
- **A time for each section:** usually called “splits”, they can feature the difference with older recorded times for each single level or section of the game;
- **Automatic Cutscene Skip:** cutscenes are usually just a waste of time in speedrunning, some may appreciate the fact that in this mode cutscenes don’t play;
- **Automatic chronometer stop:** the chronometer can automatically stop at the end of the game, and the game itself can identify when a section is finished, giving a consistent way of evaluating speedruns;
- **Automatic save of best times:** this will help speedrunners with the management of their times and remember their “PB” (personal best);
- **An input monitor:** this will show the buttons pressed by the speedrunner, it may prove to be an invaluable tool if a new strategy is discovered.

Here's how a more complex and advanced "speedrun mode" screen could look like:



What a more advanced "speedrun mode" may look like

Some game developers went to the extent of not fixing some glitches for the sake of speedrunning.

## Replayability

### Modding

A great way to gather some more tech-savvy fans of your game is opening your game's structure to editing, which means enabling people to "mod" your game.

Mods are a great way to engage your community, and there are just so many types of mods that can be made:

- **Data Packs:** These add new user-created content to the game, such as:
  - New Maps/Levels;
  - Upgraded Textures;
  - Upgraded Models (if the game is 3D);
  - Upgraded Sounds;
- **Feature Mods:** These add new features to the game, such as:
  - New game modes;
  - HUD Modifications or upgrades (mini maps for instance);
  - New options and settings;
  - Additional/tweaked difficulty levels;

- **Total Conversions:** These take the engine of a game (like the first DOOM) and make a completely new game out of it, an example is “Castlevania: Simon’s Destiny”.

Refining and giving out your own map editor can be a really great way to make your game more “open to editing” and allow modders to enjoy working on it, after that, if your game structure allows it, you could document the engine and allow people to create their own masterpieces, maps and fan-made stories.

## Fan games

If your game becomes so famous that you manage to get a full franchise out of it, fan games are bound to happen: small games that take inspiration from your storyline and take different directions, according to the creators’ will. This is a great opportunity to test how your current game is liked, and how people would like for its story to develop.

Some developers prefer to squash fan games under the Digital Millennium Copyright Act (DMCA) and get them removed, to avoid “tarnishing the franchise”, some others instead prefer turning a blind eye, and some even go to hire the fan game creators to make a fully fledged official game.

In my humble opinion, fan games are good, they signal the community’s desire for more, and you should probably play them too, new ideas come from comparing your own thoughts with other peoples’.

## Randomizing

In recent years, a certain kind of hacks for old games (mostly built around the Legend Of Zelda and Metroid series) that mix up the gameplay by mixing up the item locations, with a certain logic to avoid unbeatable games.

If your game has a somewhat linear story and no “roguelike” elements, you may be able to extend its replay value by adding a “randomized mode”. This new mode would have some logic implemented to avoid “soft locking” a

savefile and mixes up the items and powerups, usually making the game a bit more challenging.

## New Game+

A good way to lengthen the life span of your game is adding a “New Game+” mode. Let’s see what it is.

New Game+ is a mode that gets unlocked when you finish a game on a certain savefile, this mode allows you to re-play the same story and game with some additions (hence the “plus”), those additions may be multiple, like:

- **Keep your inventory/equipment:** this is especially handy in story-driven, multiple-path/multiple-ending games. This allows the player to gloss over the fighting and instead concentrate more on “taking a different path” in the story. Let’s not underestimate the willingness of a player to play a “power fantasy” by mowing through enemies in a fighting-heavy game, though;
- **Keep your statistics:** this is very useful, in a similar way to the “keep your equipment” bullet point, just more specific to RPG-style games;
- **Unlock new items:** this makes the game a bit different by allowing to use brand new, previously locked, equipment in the new “run”;
- **Make the game more challenging:** this makes every new run a harder challenge to the player, by making enemies stronger, more resistant or just by adding more enemies in the various areas. A good idea is also making enemies from mid and end-game appear earlier.

## Transmogrification

We all know that looks matter, and there are people who are willing to “pay extra” to look how they want. This is especially true in MMO games, where (sadly) peer pressure tends to be quite high.

Say hello to “Transmogrification” (sometimes called “transmog”, “tmog”, “xmog” or just “mog”): it’s a system that allows players to change the look

of a weapon, armor or anything to make it look like another weapon, armor or whatever.

In short, transmog solves the issue that is represented by the following sentence:

| This thing looks so cool! But its stats are horrible...

Using transmog you can have both the power of your best armor (or weapon, or mighty stick) with the looks of the (less powerful) armor (or weapon, and don't forget the stick) that just looks so cool.

Transmogrification can be implemented quite easily by allowing each instance of an object have a customizable sprite, instead of locking it behind the item itself. You just have to remember to save the skin used by each object's skin in your inventory/savefile.

Design-wise, you should make transmogrification quite challenging to get, since it's a "quality of life" enhancement, but at the same time it shouldn't be so frustrating that the players just give up trying.

Any decision that will be frustrating to the player (maybe to "encourage" them to make use of microtransactions) will always come back to bite you.

Remember: you are not a AAA developer or publisher, **you can't afford to take the hit of these decisions**: being too greedy can completely destroy a good game's performance.

# **Part 8: Learning from others and putting yourself out there**

# Dissecting games: two study cases

I'm a writer, so I like dissecting things.

---

Hal Sparks

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”.

## A bad game: Hoshi wo miru hito

### 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).

### Balance Issues

#### 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.

## **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.

## **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.

## **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.

## **Bad design choices**

### **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.

### **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.

## **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.

## **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 or flat-out unfair mechanics is just bad design.

## **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.

### **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.

### **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.

### **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.

## **Confusing Choices**

### **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.

### **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.

### **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.

### **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.

## **Inconveniencing the player**

### **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.

### **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.

## **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)

## **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.

## Bugs and glitches

### 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.

### 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.

## 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”.

## 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.

## A good game: Dark Souls

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

# Project Ideas

In theory, theory and practice are the same. In practice, they are not.

---

Albert Einstein

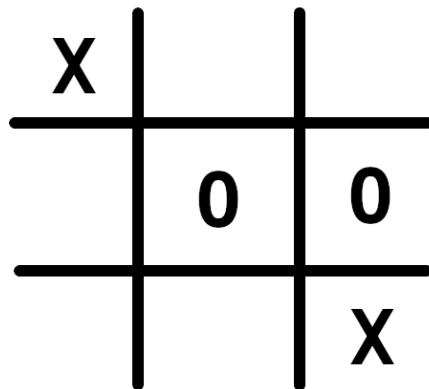
This section tries to give you some ideas for some small projects you can try to do by yourself.

The projects will be put in order of (perceived) difficulty and each one will use a larger set of skills you have learned. Each project will have three levels of completion:

- **Basic:** This is the baseline to consider a project “complete”, this is also the set of requirements the projects are sorted by;
- **Advanced:** This is more of a challenge, requiring more advanced skills and techniques. If you complete this level, you can consider yourself comfortable with most of the matters treated in the project;
- **Master:** This can be a real challenge, requiring skills and techniques that may not be taught in this book, but such problems can be solved with a bit of planning. If you complete this level successfully, pat yourself on the back and keep up the great work!

A title screen for each game is not required or necessary, but if you want to make one, feel free to do so.

## Tic-Tac-Toe



Example picture of Tic-Tac-Toe

Tic-Tac-Toe is a “finite” game, there are a finite number of choices and strategies but that makes it ideal for a simple project.

The objective of the game is to score 3 of your own symbol (either an x or an o) in a row or in diagonal, in a 3x3 grid; the players take turns and put their symbol on the game board in an empty spot.

## Basic Level

Make a simple Tic-Tac-Toe style of game, where the mouse commands both players (so it alternatively switches between x and o), the game should be able to detect winning conditions for a certain symbol or a draw.

Skills required:

- Drawing to a screen;
- Event handling;
- Mouse Events;
- Winning/Losing Conditions.

## Advanced Level

Make a computer-controlled player, by making the mouse write only the x symbol, while the computer will randomly put the o symbol onto a blank

spot. The game should still be able to detect winning conditions for each player, as well as a draw.

Furthermore, the game should enforce turns, so that the human player is not able to put their own symbol when it is the computer's turn.

Further skills required:

- Random number generation;
- Coding basic game logic.

## Master Level

Improve your computer-controlled player by making it actually seek for a way to win against the human player. The AI should do these checks, in order:

1. “Can I make a 3-in-a-row in a single move?”: this means that the AI has 2-in-a-row with an empty space available. The AI should put their symbol in the empty spot to win.
2. “Is my adversary about to make a 3-in-a-row?”: this means the human player has 2-in-a-row with an empty space available. The AI reaction should be to put their symbol in the empty spot.
3. If none of those cases happen, then make the AI fall back to a “random choice”.

The AI does not need to be perfect, just mildly challenging.

### Tip!

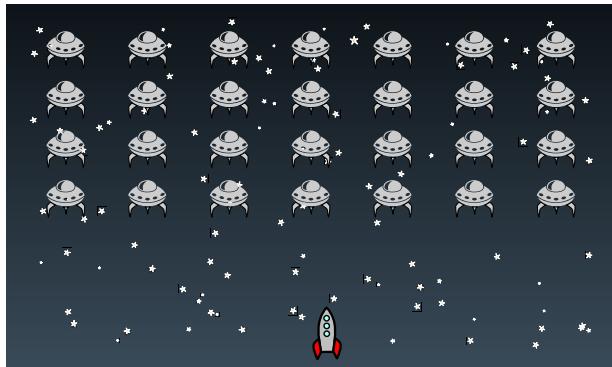
To some people, points 1 and 2 may be problematic, in the case that two among rows, columns or diagonals satisfy one of those conditions.

If both satisfy condition 1, there is actually no problem, the AI would win anyway: the AI can choose randomly.

In case 2, the AI should check if there is a way to block both “potential wins” in one move, but it is a perfectly acceptable solution to consider the case a loss for the AI and just choose randomly.

## Space Invaders

Manufactured by Taito in 1978, the arcade Space Invaders is probably one of the most known games around for its historical value.



Example of a “Space Invaders”-style game

Your objective is to prevent a horde of aliens from landing, by shooting them with your monochrome laser cannon. The concept is deceptively simple, but the implementation can be really complex if you look more thoroughly. The more aliens are killed, the more the remaining ones descend faster, there is a bonus ship that pops out at random intervals, the aliens shoot back at you, there are destructible “shields” to give you some defense from the alien bullets, the win and lose mechanics...

### Random Trivia!

The aliens getting faster wasn’t initially intended in Space Invaders, the position of the aliens gets updated every frame, but the hardware couldn’t process all the entities fast enough. The more aliens got killed, the less entities had to be processed per frame, making the aliens move faster.

Instead of coming up with a solution, this “bug” was kept as a challenging “feature”

## Basic Level

In the basic level, we will just create a static, unarmed horde of aliens menacing our base (although being static it won’t be much of a threat), we can shoot out projectiles at them and they disappear when they’re hit, adding some points to our score.

Since our unwanted alien guests are not much of a threat, we won’t need to put up any shields to defend against bullets. When all the aliens are dead, we win the game.

Skills Required:

- Drawing on screen;
- Collision Detection;
- Keyboard Input;
- Bullets;
- Score management;
- Winning conditions;
- Managing entities.

## Advanced Level

For the advanced level, we will make our alien foes more menacing by introducing movement and making the shoot at random intervals towards our laser cannon. The aliens do not need to accelerate as in the original game (but it’s not really difficult to implement).

When an alien touches the ground, we will lose the game. Thus making the game a bit more difficult to manage (as well as to play).

We will also introduce a bonus ship that appears on top of the alien horde, floating from left to right, awarding a bonus when successfully hit.

## Tip!

To make the bonus ship, you may be tempted to create an entirely new object, but at the same time a “bonus ship” IS AN “alien ship”, which would call for subclassing.

If you separate behaviour from the objects well, with methods, you can use subclassing to “change the ship’s behaviour” by overriding the `update()` method.

Further Skills Required:

- Timers (for the ship movement, and if you want, the bonus ship);
- Random number generation (for the bonus ship and the alien bullets);
- Subclassing;
- Losing conditions.

## Master level

For the master level we will make the shields, which are complex to code in the way of the original game, so we will “cheat”.

Our shields will be “force fields” which can withstand 3 shots before getting disabled: such shots can come from the aliens or our ship. After 5 to 10 seconds, the shields will come back online and ready to stop bullet again.

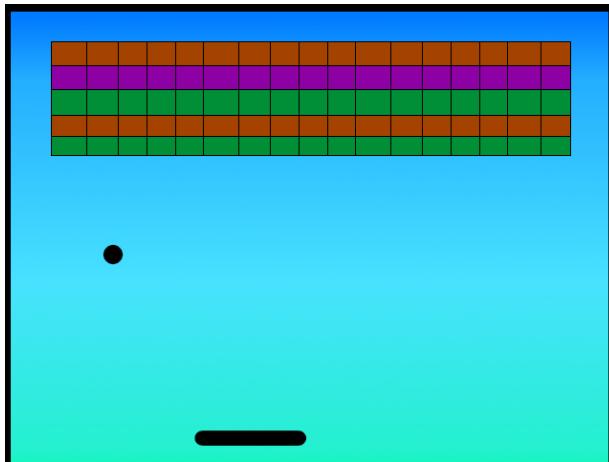
Like the original game, the shields can block both our and our enemy’s bullets, making for a more challenging gameplay.

For an even more challenging gameplay, we can (rarely) make a random ship detach from the horde to try and attack us directly: on such event, the ship will start shooting more often and move towards our laser cannon, before going back into position (a bit like galaxian).

Further Skills Required:

- Timers (this time for the shields);
- Minimal AI (for the “ship attack”);
- Managing object states.

## Breakout



Example picture of a breakout-style game

Breakout is a well-known brick-breaker style of game, where the player drives a paddle trying to keep a ball from getting to the bottom of the screen, while breaking all the blocks on the screen.

### Basic Level

The basic level is just making the basic game: make a single level that works. If the player loses their ball, they lose, if they manage to clear the screen they win.

The paddle should be moved with mouse or keyboard (you can choose to implement either or both) with sufficient speed to avoid useless deaths but slow enough to be usable. The ball should bounce like it bounces on a wall: keeping the angle with the wall without changing it.

Skills Required:

- Drawing to a screen;
- Vectors;
- Moving elements of the game;
- Event handling;
- Winning/Lose Conditions;
- Mouse/Keyboard Controls;
- Collision detection and reaction;
- Object management (creation and deletion);
- Score keeping.

## Advanced Level

In the advanced level you should implement a basic life system: each time the ball crosses the lower side of the screen, you lose a life; when all lives are lost, you lose the game.

To make the game a bit more interesting, it could be an idea to implement different block types:

- **Explosive Blocks:** When destroyed, these blocks explode, destroying the surrounding blocks;
- **Multiple-hit Blocks:** They simply require more than one hit to destroy;
- **Score Blocks:** When destroyed, these blocks drop score items that slowly descend toward the bottom of the screen. If you catch these items with the paddle, you get extra points.

Furthermore, the ball should progressively get faster with gameplay, you can either do it every few bounces or just every bounce with the paddle. The choice is yours.

Further Skills Required:

- Managing Object's states;
- Subclassing;
- Managing Game State.

## Master Level

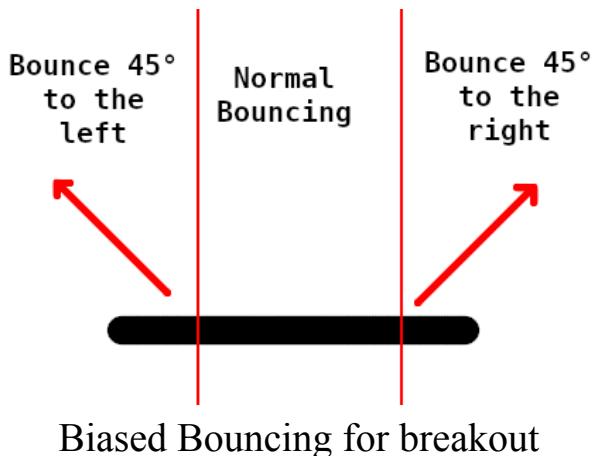
With the “master level” we are going to complete this game by adding powerups: they work in a similar way to the advanced level’s score blocks with a descending item that grants a certain status to either the ball or the paddle.

You can make it a random chance for each block destruction or make a dedicated block. Here are some suggestions for some powerups:

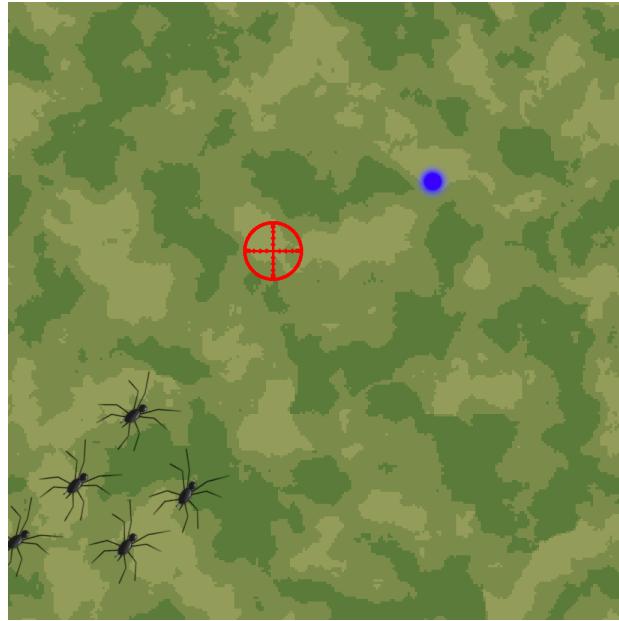
- Larger/Smaller paddle;
- Paddle that shoots to destroy blocks;
- Faster/Slower ball;
- Ball that goes through blocks, destroying them;
- “Sticky Paddle” that allows to stop and then release the ball;
- Multiball.

Furthermore, you can implement a sort of “biased bouncing” for the paddle: the further left or right on the paddle the ball touches, the more it will take a horizontal bias towards that direction.

This way the center of the paddle is “neutral”, keeping the normal bouncing mechanics, while the leftmost and rightmost sides allow the player to direct the ball the way they want.



## Shooter Arena



Possible Shooter Arena Game

This is a simple mouse-controlled arena shooting game, much in the style of “Crimsonland” and “R.I.P.”: the objective of the game is surviving for as long as possible.

## Basic Level

This level entails making the basic game: being a horde-based game, the AI can be really basic, following your movements and trying to touch you.

A single type of monsters will spawn from outside the screen and slowly make its way towards you, while the playable character will shoot bullets towards the mouse cursor.

If a bullet hits an enemy it will die and your score will be incremented, if an enemy hits you you will die.

Skills required:

- Drawing to a screen;
- Vectors;
- Basic AI;
- Collision detection and reaction;

- Projectiles;
- Lose Conditions;
- Mouse/Keyboard Controls;
- Spawning entities;
- Score Keeping.

## **Advanced level**

In the advanced level you should implement a life system for our playable character, so instead of dying our player will get hurt (thus reducing its health) and will have its life reduced. A health bar should be shown on screen.

The same should be done for each enemy, so you will need to be able to manage the state of each object (enemy) separately. Extra points if you show a small healthbar on top of a hit enemy, showing its current health.

You should also implement a way to easily code in new enemy types, this will require refactoring your code to support importing entities from data. Each enemy should at least have different speeds and health.

On each death, the enemy should have a random chance of dropping a medkit that will heal you when touched, such item should stay on screen for a limited amount of time, then disappear if not used.

Further Skills Required:

- Drawing an HUD;
- Managing an object's state;
- Coding entities as data;
- Random number generation;
- Timers.

## **Master Level**

In the most difficult level, you should start coding powerups, like new weapon types, temporarily increased walking speed, higher damage

projectiles, etc... Similarly to medkits, these powerups should disappear after a certain amount of time.

An interesting weapon to implement would be a “railgun”, with bullets that can go through enemies, this will be easier when you use ray casting (and some tricks for drawing), if you didn’t use it already.

You should also animate the characters, thus getting used to your favourite engine’s animator nodes/classes: this will make the game feel more complete.

On each death, the enemy should leave a blood (or if you prefer, goo) splatter that will disappear after a few seconds: this will make the game more messy and in its own way fun.

Further Skills Required:

- Animators;
- Ray Casting.

# 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.

## 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.

## 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.

## 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**.

## 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*<sub>g</sub> 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).

## 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.

## 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.

## **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.

## **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".

## 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 DLLs. 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!

# 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]

## Collections of different topics

## 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]

## Videos

- freeCodeCamp.org’s YT: <https://www.youtube.com/channel/UC8butISFwT-WI7EV0hUK0BQ> [F]
- 3DBuzz (Archived on the Internet Archive): <https://archive.org/details/3dbuzz-archive> [F]

## Multiple formats

- Daily “Free Learning” offer from PacktPub: <https://www.packtpub.com/free-learning> [F]
- “Awesome lists”: <https://github.com/sindresorhus/awesome> [F]
- Gamedevelop.io: <https://gamedevelop.io/> [F]

## Pixel Art

### Multiple Formats

- Pedro Medeiros’s Patreon Page: <https://www.patreon.com/saint11/> [D]

## Sound Design

### Multiple Formats

- EpicSound’s Sound Effects Tutorials: <https://www.epicsound.com/sfx/> [F]

# Game Design

## Books

- **100 Game Design Tips and Tricks** (*Wlad Marhulets*)  
<https://archive.org/details/100-design-tips-and-tricks/> [F]
- **A theory of fun for game design** (*Raph Koster*) [P]
- **The Art of Game Design: A Book of Lenses** (*Jesse Schell*) [P]

# Game Development

## Videos

- **OneLoneCoder's YT:** <https://www.youtube.com/channel/UC-yuWVUpIkJZvieEligKBkA> [F]

# References and Cheat Sheets

- **Easing functions Cheat Sheet:** <https://easings.net/> [F]

# **Appendices**

# 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.

## C

### Call by reference

Evaluation strategy where a function parameters are bound to a function by passing a reference to the arguments, this could cause *side effects* since the function would be able to change variables outside its local scope.

### Call by value

Evaluation strategy where a function parameters are bound to a function by making a copy of the values used as an argument.

## D

### Dangling Pointer

A dangling pointer is a memory pointer that references an area of memory that doesn’t contain a valid object. A dangling pointer usually happens when an object is deleted from memory forcibly, but the pointers referencing said object are not invalidated (usually by setting such pointers to a null value).

### 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).

### Information Hiding

Information hiding is one of the basic principles of programming: each part of a program (a “module”) should not expose its inner workings, but rather expose a stable “interface” to the outside world. This will

help separating modules from each other and avoid “snowball effects” when modifying the inner workings of one of them.

## 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.

## L

### Letter Notation

Also known as “letter music notation”, it’s a music notation system that uses letters A through G to write music.

## 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...)

### Memory Leak

A memory leak is usually the result of a programming error, where the memory is not correctly managed. This usually entails allocating and using memory without releasing it, thus the program will eat more and more memory as it keeps running.

### Modern music notation

The most common way to write music, using symbols to indicate the duration and type of note, while the symbol’s positioning in a 5-line staff defines its pitch.

# O

## Oscillator

An oscillator is a device (usually hardware) used to create an alternating current. Oscillators can be used in audio synthesis to create pitches.

## Out of order execution

Paradygm 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

## Pre-emption

See *preemptive multitasking*

## Preemptive multitasking

A multitasking environment where the operating system forcibly initiates context switches (saves the state and interrupts temporarily) between running processes, regardless whether their job is finished or not.

## 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

## Side Effect

In computer science a function is said to have a “side effect” when it changes variables outside its local environment, this can happen in languages which use *call by reference<sub>g</sub>* evaluation strategies.

## 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.

## Unreachable Memory

This is a phenomenon where some dynamically allocated memory has no more references pointing to it. This is a common cause of memory

leaks in programming languages without automatic garbage collection (like C and C++).

## 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.

## 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.

## **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.

## 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.

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**

---

**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.

## **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:**

<b>Windows</b>	<b>Linux</b>	<b>Mac OS</b>	<b>iOS</b>	<b>Android</b>	<b>Web-Based</b>
✓	✓	✓	✓	✓	

---

Löve is distributed under the ZLib/png license, which allows for both commercial and personal use, both in proprietary and open-source situations.

## **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:**

<b>C++</b>	<b>C</b>	<b>C#</b>	<b>Go</b>	<b>Java</b>	<b>Python</b>	<b>Ruby</b>	<b>Lua</b>	<b>Rust</b>	<b>JavaScript</b>
------------	----------	-----------	-----------	-------------	---------------	-------------	------------	-------------	-------------------

---

✓

## **Other Bindings:**

## **Proprietary Language   Visual Programming**

---

### **Platform Compatibility:**

<b>Windows</b>	<b>Linux</b>	<b>Mac OS</b>	<b>iOS</b>	<b>Android</b>	<b>Web-Based</b>
✓	✓	✓	✓	✓	✓

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.

## **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:**

<b>C++</b>	<b>C</b>	<b>C#</b>	<b>Go</b>	<b>Java</b>	<b>Python</b>	<b>Ruby</b>	<b>Lua</b>	<b>Rust</b>	<b>JavaScript</b>
✓	✓			✓	✓		✓		

### **Other Bindings:**

## **Proprietary Language   Visual Programming**

---

### **Platform Compatibility:**

---

<b>Windows</b>	<b>Linux</b>	<b>Mac OS</b>	<b>iOS</b>	<b>Android</b>	<b>Web-Based</b>
----------------	--------------	---------------	------------	----------------	------------------

---

✓	✓	✓
---	---	---

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 permissive MIT license.

## Panda3D

**Website:** <https://www.panda3d.org/>

**Price:** Free

Panda3D is a complete and open source 3D game engine.

**Relevant Bindings:**

---

<b>C++</b>	<b>C</b>	<b>C#</b>	<b>Go</b>	<b>Java</b>	<b>Python</b>	<b>Ruby</b>	<b>Lua</b>	<b>Rust</b>	<b>JavaScript</b>
------------	----------	-----------	-----------	-------------	---------------	-------------	------------	-------------	-------------------

---

✓	✓
---	---

**Other Bindings:**

---

<b>Proprietary Language</b>	<b>Visual Programming</b>
-----------------------------	---------------------------

---

**Platform Compatibility:**

---

<b>Windows</b>	<b>Linux</b>	<b>Mac OS</b>	<b>iOS</b>	<b>Android</b>	<b>Web-Based</b>
----------------	--------------	---------------	------------	----------------	------------------

---

✓	✓	✓
---	---	---

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.

## 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.

## 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.

## 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.

# **Some other useful tools**

## **Graphics**

### **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<sub>g</sub>*.

<https://www.aseprite.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/>

### **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/>

### **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/>

## **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/>

## **Laighter**

This is a great “name your own price” tool that allows you to create normal maps for 2D textures, to give your game a more “professional” feeling.

<https://azagaya.itch.io/laighter>

## **LibreSprite**

LibreSprite was created as a fork of Aseprite, but it keeps the previous GNU GPL 2 license.

<https://libresprite.github.io/>

## **NormalMap-Online**

This is a very nice tool that allows you to create normal maps automatically, for free. Good for a quick game jam or for starters.

<https://cpetry.github.io/NormalMap-Online/>

## **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/>

## **Pixelorama**

Pixelorama is a really interesting beast, since it is made using the Godot game engine and GDScript, it features tools, animation timeline, tool options, pattern filling, and many many more interesting features.

<https://orama-interactive.itch.io/pixelorama>

## **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**

### **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.

This program has been object of controversy due to licensing and telemetry, thus a fork has been made, called “Tenacity”, which is [described below](#)

<https://www.audacityteam.org/>

### **Bosca Ceoil**

Created by Terry Cavanagh (creator of VVVVVVV, Super Hexagon and Dicey Dungeons), this tool is geared towards beginners, containing chords

and scales as well as pre-made instruments. This allows to get something good going on straight away.

This tool is open source, made in Adobe AIR, available for Windows, Linux and Mac

<https://terrycavanagh.itch.io/bosca-ceoil>

## **Chiptone**

An alternative to SFXR, Chiptone is an online tool (made with HTML5, Haxe and OpenFL) that can be used to create chiptune-like sounds for your games.

It also features downloadable versions for Windows and Mac OS.

<https://sfbgames.itch.io/chiptone/>

## **FamiStudio**

FamiStudio is a DAW (Digital Audio Workstation) software that allows you to edit and create tracker files for the Nintendo Famicom/NES. Its interface is very intuitive and resembles a lot of DAW software you can find.

The software is available for Windows, MacOS and Linux.

<https://famistudio.org/>

## **Helio**

The Helio project poses itself as “an attempt to rethink a music sequencer to create a tool that feels right”. It strives for an extremely clean interface, throwing the idea of it being used as a tool to “grow as a composer”.

It is still in its early stages, but what’s there seems really good.

This software is available for Windows, Linux, MacOS, Android and iOS.

<https://helio.fm/>

## 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/>

## 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/>

## 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/>

## 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](http://www.drpetter.se/project_sfxr.html)

## Tenacity

Tenacity is a fork of the Audacity audio editor, mostly due to licensing woes and complaints about new telemetry functionalities and data collection.

<https://tenacityaudio.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 December 18th, 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 but now features 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.

## **CCMixter**

Website: <http://ccmixter.org/> and <http://dig.ccmixter.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:**

<b>Graphics</b>	<b>Music</b>	<b>Sound Effects</b>
-----------------	--------------	----------------------

---

✓

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:**

<b>Graphics</b>	<b>Music</b>	<b>Sound Effects</b>
-----------------	--------------	----------------------

---

✓

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)
- Sjofin (Fin)
- Nikita Ivanchenko (Nivanchenko)