Jocs Electronics

How to make a game

About learning how to make games

- It is a **HUGE** field that requires lots of transversal competencies and disciplines (programming, design, art, storytelling, project management, AI, networking, UX, human-computer interaction, composition, …)
- Game development is quite hard, people expect something better every time so it requires even more complex algorithms and pipelines.
- It is important to have good technical skills, but it also requires high amounts of creativity, time-management, team playing and problem-solving.
- Beyond the technical aspects, it has a strong artistic component.
- It must comply with the tendencies and spectations.

Making simple games is easy,
Making good games is hard

Game history

The 8-bit era

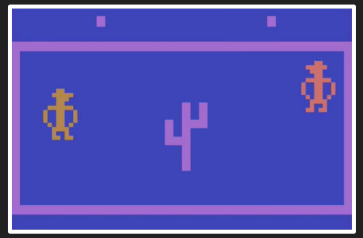
70s and 80s

Atari 2600 (1978)

CPU: MOS 6507 @ 1,19 MHz

128 Bs RAM y 4KBs de ROM

30 millones vendidos



Commodore 64 (1982)

CPU: MOS 6510 @ 0,985 MHz

64KBs RAM y 20KB de ROM

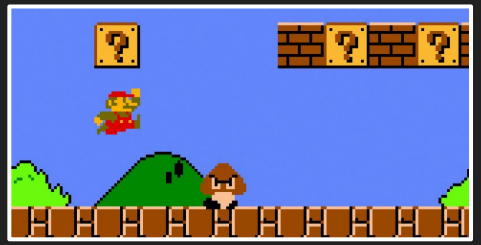
17 millones vendidos

NES (1983)

CPU: MOS 6502 @ 1,79 MHz

2KBs RAM + 2KBs VRAM y 32KBs de ROM

61.9 millones vendidos



90s

486 (1990)

CPU: Intel 20MHz hasta 120MHz

RAM: 20MBs y 120MBs de HDD

Playstation (1994)

CPU: R3000 @ 33.8688 MHz

RAM: 2 MB RAM, 1 MB VRAM

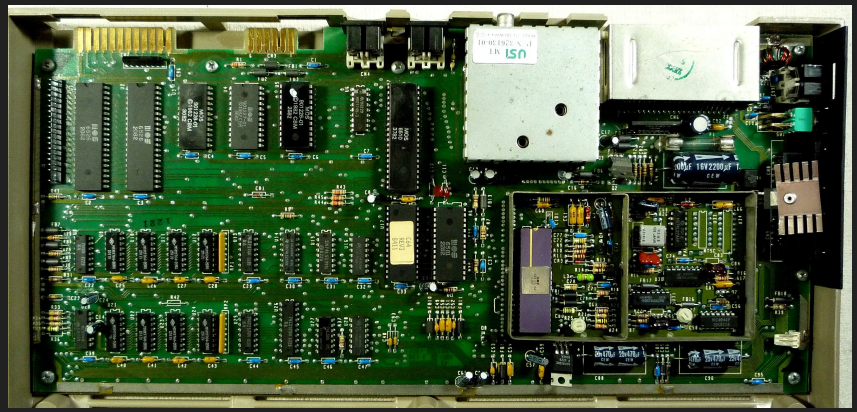
First GPU

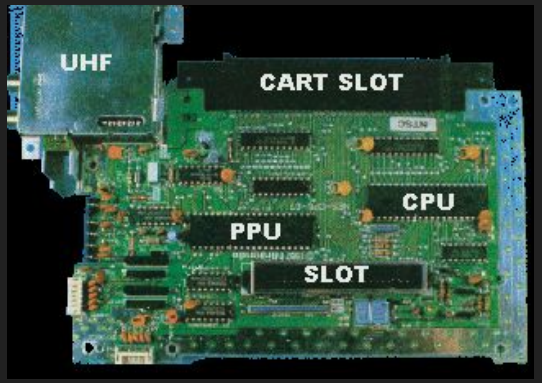
102.49 Millones vendidas

How did they make games in that thing?

- Games coded in Assembler (later in C)
- Different render modes (more colors but less resolution) No access to framebuffer.
- PPU: Video output created using registers to control sprites
- SPU: Audio created controlling an internal synthesizer
- Required perfect timing

Check this video or this video to know how complex developing a game was.





ASM

Programming in ASM could seem hard by our current standards but it was much simpler in terms of what could be done, there was just a set of documented instructions and memory addresses.

There were no Operative System, no libraries, no middlewares, just plain memory access.

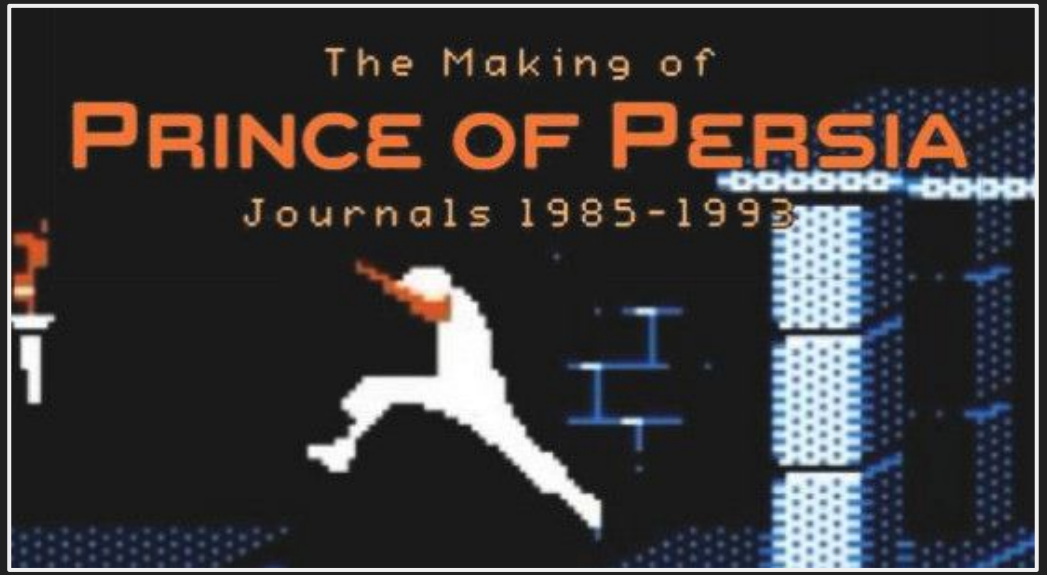
This allow people to have full control of the system and encourage them to create better games.

Here is a video about the architecture of a Gameboy.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK imp	ORA inx	*KIL*	*SLO inx*	*NOP zp*	ORA zp	ASL zp	*SLO zp*	PHP imp	ORA imm	ASL akk	*ANC imm*	*NOP abs*	ORA abs	ASL abs	*SLO abs*
1x	BPL abs	ORA iny	*KIL*	*SLO iny*	*NOP zpx*	ORA zpx	ASL zpx	*SLO zpy*	CLC imp	ORA aby	*NOP imp*	*SLO aby*	*NOP abx*	ORA abx	ASL abx	*SLO abx*
2x	JSR abs	AND inx	*KIL*	*RLA inx*	BIT zp	AND zp	ROL zp	*RLA zp*	PLP imp	AND imm	ROL akk	*ANC imm*	BIT abs	AND abs	ROL abs	*RLA abs*
3x	BMI abs	AND iny	*KIL*	*RLA iny*	*NOP zpx*	AND zpx	ROL zpx	*RLA zpy*	SEC imp	AND aby	*NOP imp*	*RLA aby*	*NOP abx*	AND abx	ROL abx	*RLA abx*
4x	RTI imp	EOR inx	*KIL*	*SRE inx*	*NOP zp*	EOR zp	LSR zp	*SRE zp*	PHA imp	EOR imm	LSR akk	*ALR imm*	JMP abs	EOR abs	LSR abs	*SRE abs*
5x	BVC abs	EOR iny	*KIL*	*SRE iny*	*NOP zpx*	EOR zpx	LSR zpx	*SRE zpx*	CLI imp	EOR aby	*NOP imp*	*SRE aby*	*NOP abx*	EOR abx	LSR abx	*SRE abx*
6x	RTS imp	ADC inx	*KIL*	*RRA inx*	*NOP zp*	ADC zp	ROR zp	*RRA zp*	PLA imp	ADC imm	ROR akk	*ARR imm*	JMP ind	ADC abs	ROR abs	*RRA abs*
7x	BVS abs	ADC iny	*KIL*	*RRA iny*	*NOP zpx*	ADC zpx	ROR zpx	*RRA zpx*	SEI imp	ADC aby	*NOP imp*	*RRA aby*	*NOP abx*	ADC abx	ROR abx	*RRA abx*
8x	*NOP imm*	STA inx	*NOP imm*	*SAX inx*	STY zp	STA zp	STX zp	*SAX zp*	DEY imp	*NOP imm*	TXA imp	*XAA imm*	STY abs	STA abs	STX abs	*SAX abs*
9x	BCC abs	STA iny	*KIL*	*AHX iny*	STY zpx	STA zpx	STX zpy	*SAX zpy*	TYA imp	STA aby	TXS imp	*TAS aby*	*SHY abx*	STA abx	*SHX*	*AHX aby*
Ax	LDY imm	LDA inx	LDX imm	*LAX inx*	LDY zp	LDA zp	LDX zp	*LAX zp*	TAY imp	LDA imm	TAX imp	*LAX imm*	LDY abs	LDA abs	LDX abs	*LAX abs*
Bx	BCS abs	LDA iny	*KIL*	*LAX iny*	LDY zpx	LDA zpx	LDX zpy	*LAX zpy*	CLV imp	LDA aby	TSX imp	*LAS aby*	LDY abx	LDA abx	LDX aby	*LAX aby*
Cx	CPY imm	CMP inx	*NOP imm*	*DCP inx*	CPY zp	CMP zp	DEC zp	*DCP zp*	INY imp	CMP imm	DEX imp	*AXS imm*	CPY abs	CMP abs	DEC abs	*DCP abs*
Dx	BNE abs	CMP iny	*KIL*	*DCP iny*	*NOP zpx*	CMP zpx	DEC zpx	*DCP zpx*	CLD imp	CMP aby	*NOP imp*	*DCP aby*	*NOP abx*	CMP abx	DEC abx	*DCP abx*
Ex	CPX imm	SBC inx	*NOP imm*	*ISC inx*	CPX zp	SBC zp	INC zp	*ISC zp*	INX imp	SBC imm	NOP imp	*SBC imm*	CPX abs	SBC abs	INC abs	*ISC abs*
Fx	BEQ abs	SBC iny	*KIL*	*ISC iny*	*NOP zpx*	SBC zpx	INC zpx	*ISC zpx*	SED imp	SBC aby	*NOP imp*	*ISC aby*	*NOP abx*	SBC abx	INC abx	*ISC abx*

No tools available

Developers must create their own



The making of prince of persia (examples)
The making of Another World

Planning & Roles

Expectations vs Reality

At the end creating a game is more an exercise of doing what you can with the resources you have available, more than knowing all the possible tools and techniques.

Obviously, the more you know the better, but your aspirations should always match your ability to fulfill them, otherwise you would never reach your goals.

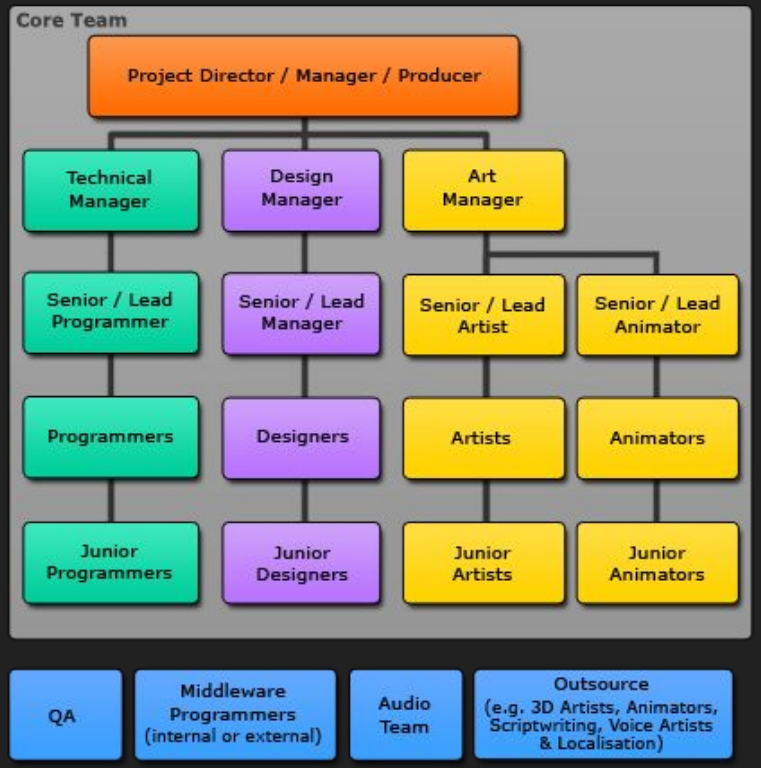
One must know when to dream big and when to compromise, an excess of ambition usually leads to the lost of motivation and the project failure.

Game development

Developing a game is a complex task that involves many disciplines and roles:

- Technical roles: people in charge of programming every aspect of the game, like engine, shaders, network, and game logic.
- Designer roles: people in charge of the gameplay, level design, story
- Art roles: character designer, animators, texture artist, sound designers, etc.
- Project management: people in charge of assigning task, reaching deadlines.

Without this roles it is very unlikely that a game idea could become a reality.



Planning

As any other kind of project, video games require a careful planning, because it is easy to get lost once your start coding, so the best approach is always to visualize the game on paper, and once it makes sense, start developing it.

Developers are usually forced to write a document with the full game description (levels, enemies, game mechanics, etc) before they are allowed to start development.

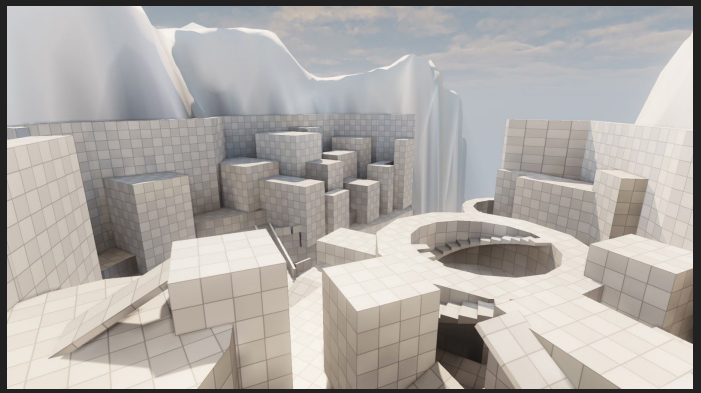
Prototyping

It is hard to know if a game mechanic will work (it is fun) before you have played it by yourself, so unless you are copying an existing game mechanic, your must test it to be sure.

Ideas that seem very fun in theory could be boring once tested and the opposite.

So before commiting to an idea, game developers usually make a very simple version of the game, with placeholder art, that showcases the main game mechanic, and once it works, they develop it completely.

Zelda Breath of the Wild prototype



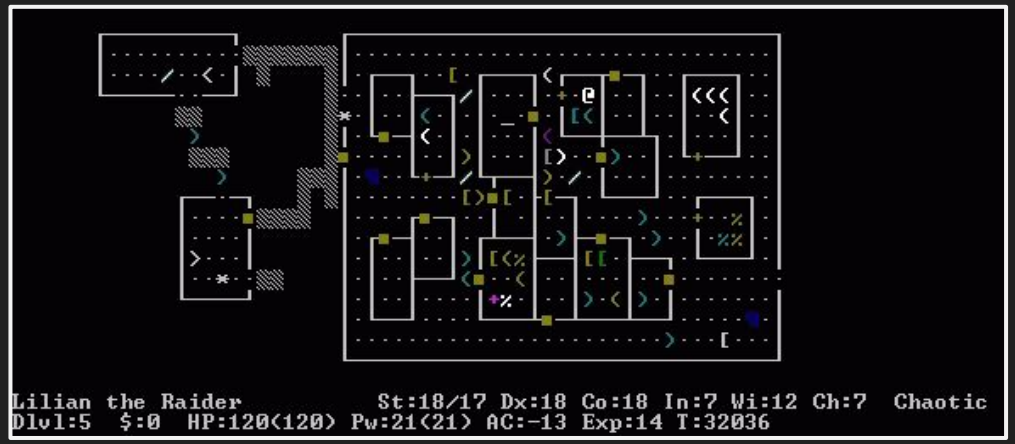
Game programming

The technical part

Programming a game

Games can be make in any programming language, even if you only have a text-based terminal.

The important thing is to use the most adequate tool for the kind of game you want to make.



Using the proper tools

You can make a game in any programming language, but there are already tools and frameworks more suited for it. Here is a list to consider:

Language	Pros	Cons	Libraries
C/C++	Very fast, very powerful Can be ported easily	Complex to code Errors crash	SDL, SFML, GLFW, raylib
C#	Fast and safer	Used GC	MonoGame, XNA
Javascript	Very portable (works in any browser) Do not need to download anything Easy to distribute (just a link)	Not as fast Garbage collected Soft typed	Pixi.js, phaser
LUA	Simple and mature	Not as fast	LÖVE, pico-8
Game Maker RPG Maker	Complete tool	Not free	

Creating an interactive application

Creating a game is the same as creating an interactive application.

We call an interactive application to those applications that ensure the next points:

- It draws at least 30 frames per second so the user perceives the movement.
- It reads the user input (keyboard, mouse, gamepad)
- It updates the scene according to the user input and internal logic

If we can ensure all this points, then we have an interactive application.

Main loop

If we want to create a very simple interactive application let’s start by ensuring those 3 points by creating an application like this.

The loop will keep executing till the must_loop equals to false, then it will exit the loop and finish the application.

```
init();
while( must_loop )
{
    draw_image();
    read_user_input();
    update_scene();
}
exit(1);
```

Drawing images

To draw images we need to fill the pixels that represent the image we want to show.

Depending on what we want to draw this process could be simple and fast or very complex and slow. But because we want to generate at least 30 images per second, we need to be sure that drawing the image takes less than 1/30 seconds.

The faster we draw, the more elements we can draw in our image to make it more pleasant, so it is important always to keep in mind if we are doing things in the best way possible.

Drawing one frame

Usually to draw one frame we start by clearing the framebuffer.

Then we draw the whole scene.

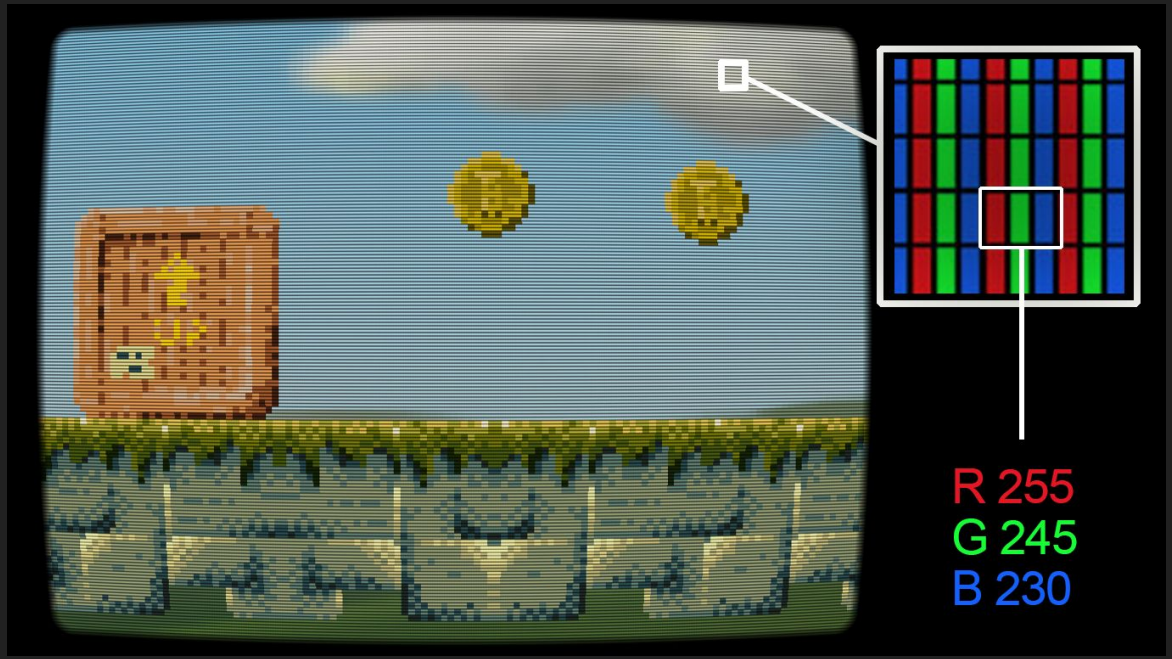
And finally we swap the current framebuffer on the screen with the one with just draw so the image is visible to the user only when it is finished.

```
void draw_image()
{
    clear_buffer();
    draw_scene();
    swap_buffers();
}
```

The Framebuffer

The framebuffer is the area of the memory that contains the color intensity of every pixel.

Pixels are expressed using three numbers, that determine the intensity of its RED, GREEN and BLUE channel.



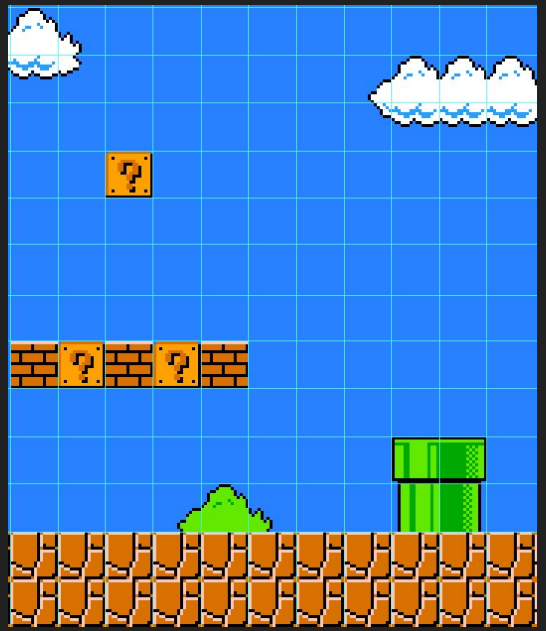
History of framebuffer

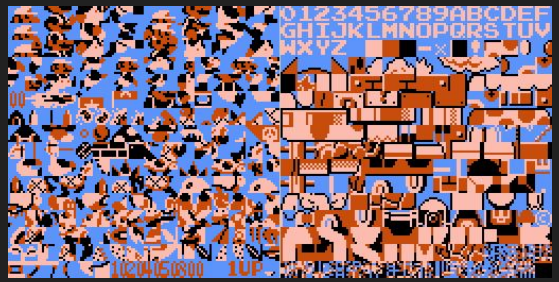
The first computers and consoles (70s-80s) didn't have a framebuffer, memory chips were very expensive and CPUs were too slow to paint the frame pixel by pixel.

Instead, they allow to draw a finite number of sprites (tiny images usually of 8x8 pixels) to the screen per frame, by assigning its position into some special registers or memory addresses. For instance, to create the backgrounds they use tilesets and assign in memory which tile goes into every position.

Programmers used tricks like changing those registers while the screen was drawing every row of the image, to be able to draw more sprites or do interesting effects.

For further info watch this video or this talk





Color tables during the 90s

During the 90s computers were faster and had more memory, this opened the option to work with framebuffers, but not in a regular way.

Instead of using RGB values per pixel they used indexed color tables. This meant that the framebuffer contained a single number per pixel (instead of three), and this number pointed to the color value in a color table (called palette).

Some artist used tricks like modifying the palette to get cool animations. For more info check this talk from Mark Ferrari and this interactive gallery of his work

Using sprites today

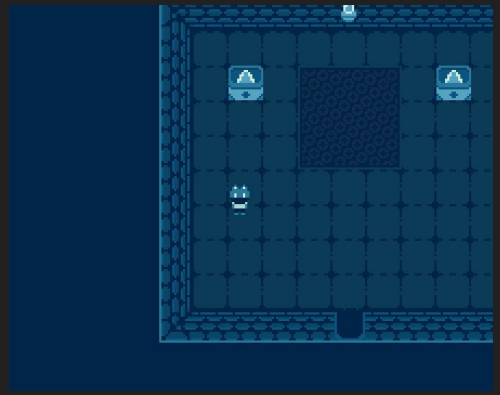
Today we do not have big limitations of memory or CPU so games can use as many sprites as they want, with all the colors and of any size.

2D games use spritesheets (images that contain all the sprites in one single image) for convenience.

And also use tilesets because it makes creating maps easier for artist and game designers.

Tricks with sprites

Use tilesets to create the maps, or directly construct a big image with the whole map



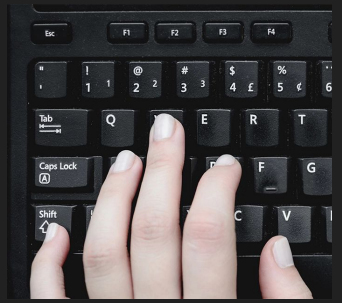
Move all the sprites to give the sense of a camera panning

Parallax (move different sprites at different speeds to create the sense of depth

Tricks: faking perspective

Some games used tricks to make games look like they have perspective. Like pseudo-3D or the mode7 for racing or piloting games, raycasting used to create First Person View games or Voxel Space for flying games. This technics require lots of maths but the results are interesting.

Reading user input



Reading user input shouldn’t be hard.

In one side we should have a way to query the current state of the devices (which keys or buttons are pressed, the position of the mouse or the axis of a gamepad).

But also we require to have a way to detect events that happen when the user interacts, otherwise if the user clicks a key too fast maybe we will miss it.

To do so we need that the OS will send us events when they happen, but to avoid interruption problems, we use an events queue where all the user events are stored and we just need to process the queue to get them.

Accessing the hardware

The problem requesting hardware devices their state is that these devices are controlled by the operative system, and we do not have direct access to them.

We need to ask the OS API to get access to those devices, but the APIs are different on every OS (and very complex in some of them).

The solution is to use a middle-layer, a library that wraps the OS calls into a more generic calls.

There are many libraries to do that: SDL, SFML, GLFW, Raylib

Using this middle-layer we ensure our code can be easily ported to any OS.

Updating the scene

By updating the scene we mean to change the current game state according to the user input and the internal behaviour of the game.

Moving the player, moving the enemies, moving the bullets, detecting collisions, etc.

But we want to be sure that no matter how fast a computer runs, the game runs at the same speed, otherwise the game will change according to the hardware where it runs.

To ensure this we must take into account how much time has passed since the last update, and use that delta_time as a factor for all the data modifications affected by time.

Updating with delta_time

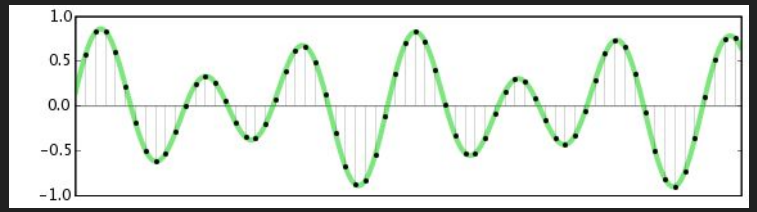
By multiplying factors by the time, we ensure that no matter how fast a game runs, the same amount of change will happen during the same amount of time.

In old computer the loop will be executed less times per second but the dt will be bigger so it should compensate.

And just to be sure, always test your game in different computers, old and new.

```
void update( float dt )
{
    if( isButtonPressed() )
    {
        player.x += 10 * dt;
    }
}
```

Audio



Doing audio is very similar to drawing images, but instead of pixels we have samples that represent the waveform to play out in the speakers.

At the beginning computers didn’t have enough memory to play digitized samples, so instead they used hardware synthesizers.

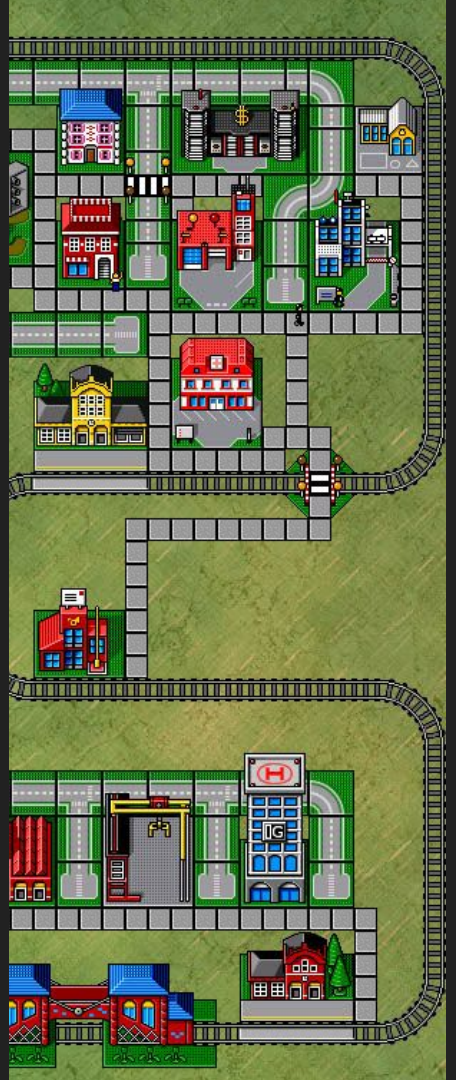
Programs will control the frequency and amplitude of every oscillator in the synthesizer and this will produce sound in the analog output of the chip.

But creating audio using this approach required perfect timing.

Check this MOS6581 chip emulator to see how game music was decomposed.

Game design

The creative part



Game design

The hardest part to make a game is not the technical aspects, but the design ones, and by design I mean game logic and gameplay. Here are some common goals that all games should achieve:

- A game must be FUN, and to be fun it requires to have a good game design behind.
- The game must be **challenging**, too easy and people get bored, too hard and people get frustrated.
- A game should offer something new, otherwise, why to play it?
- The player must feel he gets better while playing it, otherwise it feels random
- The game should have an ***ending***, some conclusion.

But there are many games that do not follow this points and still are successful

Rules to make a game engaging

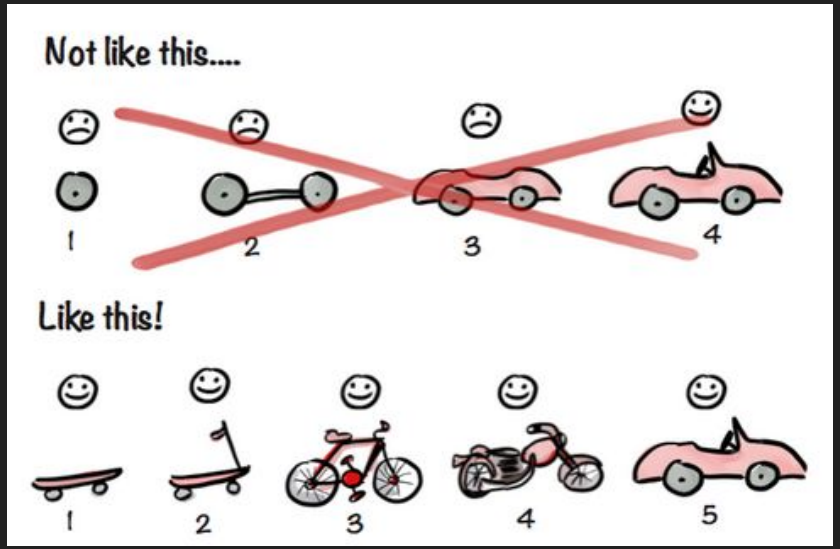
Now, if we want the user to enjoy our game, here are some basic rules:

- Try to tell the player a story, give him some context about the game world and the characters (use some intro images and text). Let him be inspired so he will fill the gaps in your game with his own ideas.
- Make clear to the user which is the goal, what should I do?
- Give feedback to the user about what is he doing
- Show him the progress, how much has accomplished and how much is still remaining.
- Once the user completed the goal, reward him with something
- Subvert the expectations, surprise the player whenever possible
- Make the game *more personal*, add details to reflect yourself as a creator

Tricks to develop a game faster

- Use very simple art, small images, small framebuffer, limited palette, two-frame animations, lots of texts. But respect the aesthetics.
- Reuse the same art as much as possible (draw the same tree many times to create a forest, instead of creating a city where every building is different).
- Focus in one system. Instead of a regular game with common traveling mechanics, focus in one specific stage of the game, like a boss fight, a puzzle or a conversation.
- Prioritize, spend more time in the parts where the user is going to spend more time. If one item only appears once in the game, then paint it fast, but spend time in the player character.
- Be resourceful, use extra images not only for sprites but also to define properties of the map (p.e. White pixels are crossable, black ones no).
- Use sprites and tilesets from other games while prototyping.
- Be AGILE, make something simple but complete and if there is more time improve it.

Be AGILE, never lose focus on the game



Have always a playable and conclusive version. Even if the player is a box and the goal is to touch another box into the screen. Then from there you start improving.

Things to avoid when developing games with short time

- Avoid narrative games (games with complex story), it requires lots of art to tell a story, otherwise is boring. Also, reading text gets boring unless you are a great writer.
- Avoid a game where the enemies have complex AI (just give them basic behaviour)
- Avoid using complex physics (bouncing stuff, collisions with platforms, etc)
- Avoid creating big worlds, everything takes longer than planned, focus in one room or small area, instead of lots of levels that require different art.
- Avoid creating a game that relies in some sort of complex algorithm.
- Avoid having too many game mechanics, focus in a single one (a boss fight)
- Avoid using an unknown technology that you never have used before
- Avoid creating an online game, networking is hard, too many cases

But the most important: use your common sense and have a plan B for everything

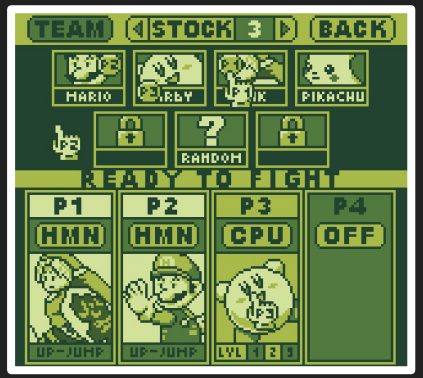
Things you can do that are easy and work well

- Add some cinematic intro (three images fading narrating a story)
- Use a grid-based world and movement, makes all way easier to develop.
- Add local multiplayer. Code with two players in mind, it doesn't make coding the game harder, it is more fun to play with other people than alone
- Add charismatic characters, make them say funny things
- Add emotions to the characters including the player (sad, happy, scared, angry, etc)
- Play with camera effects (panning, dramatic zoom, camera shake, …)
- Add a narrator that describes what is going on (just with subtitles)
- Add high-scores, let the players beat each other
- Add music to make scenes more dramatic or cheerful
- Localize the game to your audience, set the game in your city or around some current events (it makes the game feels closer)

Getting inspired

Ideas: old games

Commodore 64 games



Gameboy

NES games

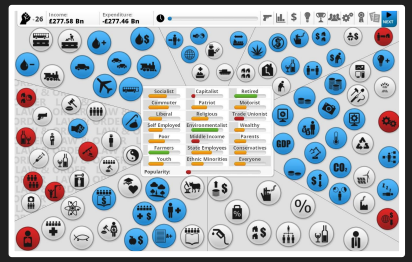
DOS games

How to get good ideas

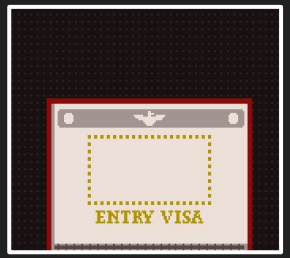
- Force yourself to come up with 10 different ideas, do not get attached to any of them, then rate them according to originality, complexity, and size.
- If there is a theme, think about all possible meanings for the word.
- **Dream big**, but then try to simplify the idea till a point it is feasible.
- Try to stay away from saturated topics, go to areas where not many games have been before, it is easy to innovate there.
- Get inspired by other media (books, news, TV, movies, comics)
- Make a game about *something you care about* (sports, politics, your teacher)
- Stay away from common game-mechanics, not all games are about jumping, shooting or require having good reflexes, a game can also be relaxing.
- Its ok to copy classic games, but at least add something original to it.

Ideas: genres and themes

There are more ideas for games than zombies, knights or space-soldiers...



Politics

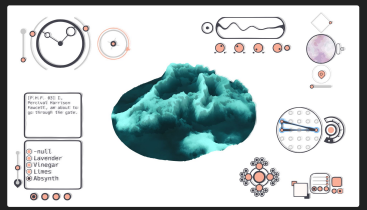


Burocracy

Engineering

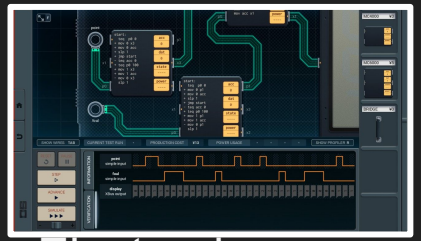
Management

Philosophy



Cartography

Farming



Electronics

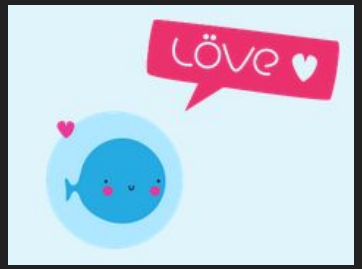
Ideas: amateur communities

Ludum Dare

itch.io

Pico-8 games & TIC-80

LÖVE



IGF

tigsource

Ideas: Choose a camera view

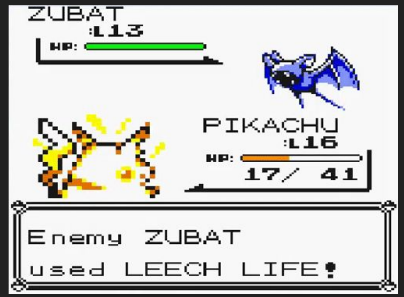
Side scroller

Isometric

First person

Cenital view

Top-down oblique

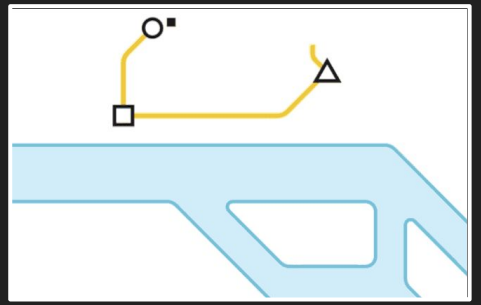


Fixed view

Ideas: Visual styles

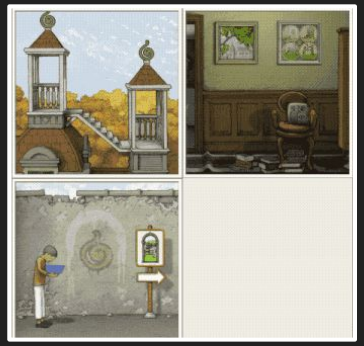
Pixel art

Hand-drawn



Vectorial

Minimalist



Illustrations

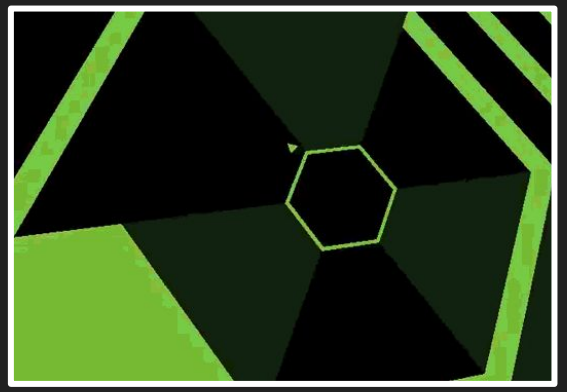
Using photos

Ideas: Choose a context

Ideas: successful indies

Baba is you

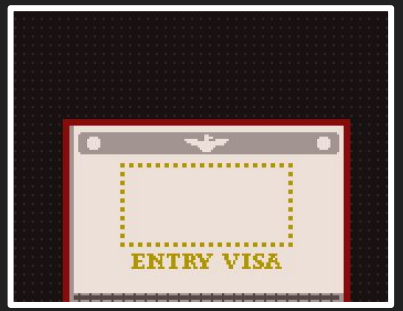
Undertale



superhexagon

Dwarf fortress

Kingdom



Papers, please

Ideas: get inspired by other fields

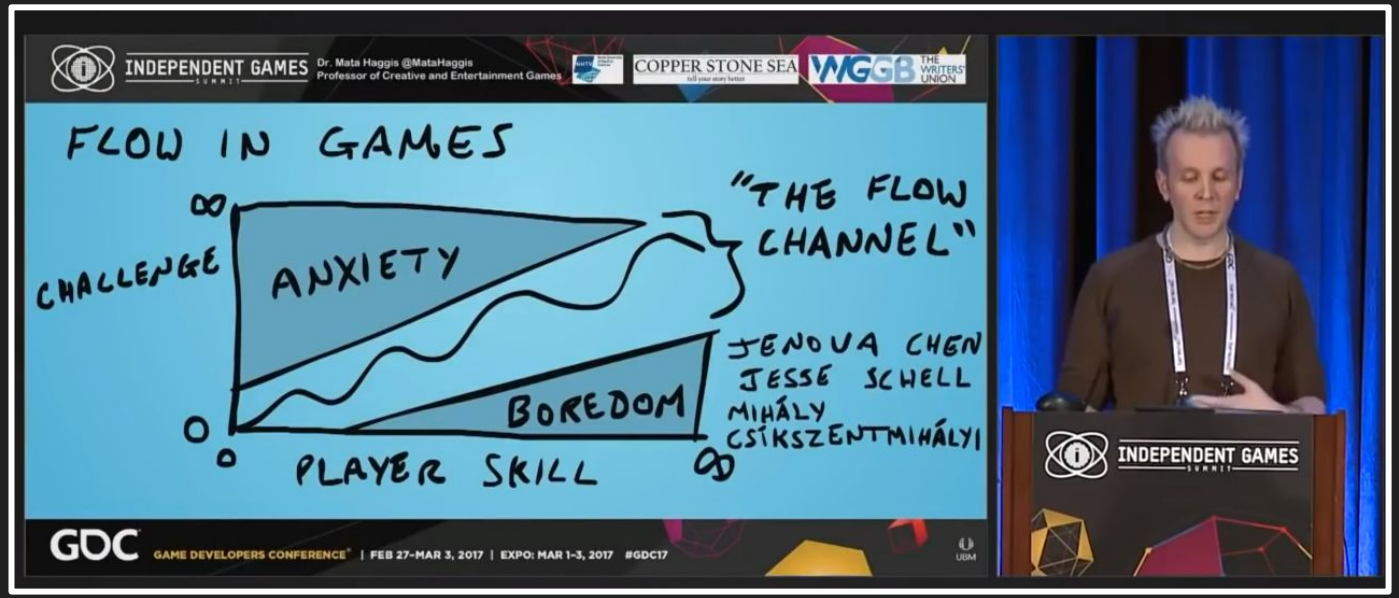
From design

From TV Shows

From boardgames

Examples of games made in 48 hours

History of Creativity



This video contains lots of useful information about video game creativity

and the most important...

“the game as a medium, not as an end”