

# Trabajo Práctico de Algoritmos 1 - 2022

21 de Junio de 2022

---



## Integrantes:

- ❖ Mercedes Conde
- ❖ Denise Martin
- ❖ Alejandro Nava

---

## Precuela:

Para conocer el primer juego de carpincha: "Capybara adventure - Carpincha tu Propia Aventura", hecho en assembler 8086, se puede descargar de github:

<https://github.com/Enghellcraft/Assembly-8086---Game---Carpincha-Tu-Propia-Aventura>

Allí nos encontramos como un pequeño carpincho que al despertarse un día, encuentra una masiva construcción hecha, frente de su lugar de descanso. Pero la insistente criatura, decide intentar llegar al otro lado del edificio y reencontrarse con sus pares. Lleno de peligros y malas decisiones la historia cuenta con 2 finales: o es atrapado por los viles humanos o logra llegar a su destino deseado.

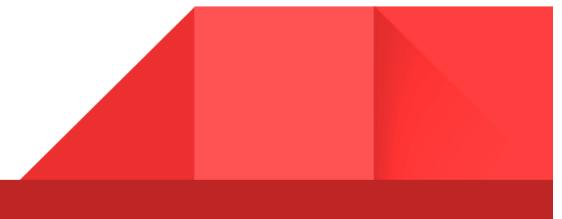
## Historia Capybara adventure - Carpinchos en Fuga:

Continuación de "Capybara adventure - Carpincha tu Propia Aventura". Para ambos finales, concluiremos que finalmente hubo un escape de los humanos (si fue atrapado en el juego anterior) o que llegan nuevos humanos que desean desalojarlos de su hábitat natural (si lograron el final de encuentro con sus otros carpinchos).

Nuevamente nuestro querido capybara deberá sortear desafiantes retos en pos de huir de los nuevos malvados humanos y poder vivir una vida despreocupada.

## Idea General:

En los tres niveles, capybara encontrará paisajes nuevos , obstáculos que esquivar, además de humanos y animales de los cuales debe escapar, para no perder vida. Para ayudarlo, encontrará a unos viejos amigos: botellas de los distintos países donde viajó, que le permitirán ir más rápido (Tequila de México) o lento (Cerveza de Alemania), o quizás darle un poquito de vida (Birkir de Islandia). Deberá recolectar las llaves necesarias para poder ir a casa a encontrarse con sus amigos.



---

## Niveles Destacados:

**Nivel Construcción:** primero deberá escapar de las edificaciones construidas por los humanos (paredes) y de sus garras (villanos constructores), recolectar 4 llaves y salir por el agujero de la pared.

**Nivel Pradera:** segundo deberá huir entre los obstáculos (rejas) y escaparle al “control de plagas” para llegar rápido al lago. En el camino deberá recolectar 6 llaves y escapar por la reja.

**Nivel Lago:** por último, esquivando los árboles talados (troncos), podrá escapar de los depredadores naturales (caimanes y serpientes) para reencontrarse con sus amigos. En el trayecto deberá recolectar 6 llaves y llegar a la madriguera.

## El Juego:

<https://github.com/picsfrunk/wollok-game-Capybara-adventure---Carpinchos-en-Fuga>

## ¿Qué usamos?:

**Polimorfismo:** las instancias producidas en el generador.wlk comparten el entendimiento de los mismos mensajes debido a su creación desde la superclase enemy de enemies.wlk y objetos, obstáculos y llaves de objects.wlk: llamando a la posibilidad de crearse, borrarse, responder a gravedad, etc. Los valores booleanos descritos en el siguiente punto cumplen con el fin de poder ser invocados por distintas clases y objetos para que los demás elementos del código puedan filtrarlos más adelante.

- **Valores Booleanos** -> siguiendo la línea del polimorfismo y para evitar los “if”s , podemos encontrar en objects.wlk el método “isObstacle()” para poder identificar qué objetos serán obstáculos que no permitirán pasar a capybara cuando lleguen a y=0:

```
class DefaultObject {  
    method isObstacle() = false  
,
```

```
class Obstacle inherits VisualObject{  
    const damage = 10  
    override method isObstacle() = true  
    . . .  
,
```

```
method puedeSaltar(_position) {  
    return game.getObjectsIn(_position).  
        all({visual => ! visual.isObstacle() } )  
,
```

Otro caso es el método enCurso(). Este método comienza en “false” en la clase Nivel pero toma el valor “true” según el nivel del juego en que nos encontremos:

```
class Nivel inherits DefaultObject {  
    var property nivel  
    var property enCurso = false  
    . . .  
,
```

```
method cargar() {  
    enCurso = true  
}
```

```
method levelUp(){  
    self.resetPosition()  
    if (nivel3.enCurso()) {  
        (nivelActual.is()).pista().stop()  
        self.win()  
    }  
    else {nivelActual.is().terminar()  
}  
}
```

**Colecciones:** en *generador.wlk*, el generador de objetos, tiene establecido un valor límite para no inundar la pantalla de enemigos; estos objetos se guardan en una lista para luego calcular su cantidad total:

```
class ObjectGenerator {  
    var property max  
    const genObjects = []  
  
method haveToGenerate() = genObjects.size() <= max
```

Allí también podemos ver una lista con números para utilizar como sufijos para permitir aleatoriedad en las distintas imágenes de enemigos y obstáculos:

```
class Factory {  
    const property suf2 = (1 .. 2)  
    const property suf3 = (1 .. 3)  
  
object humanFactory inherits Factory{  
    override method build() = new Human (sufijo=suf3.anyOne(), position=self.random())  
}  
  
object predatorFactory inherits Factory{  
    override method build() = new Predator(sufijo=suf2.anyOne(), position=self.randomLeft())  
}  
  
class Human inherits Enemy{  
    var sufijo  
    method image() = "human_worker" + sufijo.toString() + ".png"  
    override method borrar(){  
        humanGenerator.borrar(self)  
    }  
}
```

En *capybara.wlk*, puede observarse otra lista que acumulará las llaves recolectadas en cada nivel para satisfacer la condición de ganar de cada nivel (establecida dentro del nivel correspondiente en *niveles.wlk*):

```
object capybara inherits objects.DefaultObject {  
    var property position = game.at(1,0)  
    var property sufijo = "inicial"  
    var property life = 100  
    var property nextPosition = game.at(0,0)  
    var property keys = []  
    var property keysForWin = 0  
    const property maxLife = 100
```

```
method addKey(key){  
    game.removeVisual(keychain)  
    keys.add(key)  
    if (self.keyscount() == keysForWin)  
        nivelActual.is().showExit()  
    game.addVisual(keychain)  
}  
method keyscount() = keys.size()
```

**Clases:** creadas en *enemies.wlk* para los distintos humanos y animales; en *niveles.wlk* para los tres niveles, pantallas de inicio, pantallas de finalización y carga de pantalla de nivel; en *objects.wlk* para las botellas, obstáculos, salidas, llaves, contadores de vida, llaves y tiempo; y en *generador.wlk* para los enemigos. Nos permite tener varias instancias únicas que se comportan de igual o similar manera.

La necesidad de las clases surge a partir de la gran cantidad de elementos que responden a las mismas definiciones que debimos crear en cada nivel.

**Instanciación:** en el *generador.wlk*, utilizamos objetos “factory” para la creación de nuevas instancias/personajes que debíamos generar aleatoriamente: humanos, animales, obstáculos, botellas, llaves, etc. provenientes de las clases en *enemies.wlk* y *objects.wlk*.

```
object beerFactory inherits Factory{  
    method buildBottle() = new Beer(position=self.random())  
}
```

**Herencias:** en *generador.wlk*, todos los objetos “factory” heredan de la clase *Factory*, donde se encuentra un método que permite que las nuevas instancias aparezcan únicamente en espacios vacíos del tablero. También los obstáculos, enemigos y llaves heredan sus características de la clase *ObjGeneratorWithGravity*, que permite el establecimiento de tiempo de creación y gravedad, y la capacidad de modificación de las mismas por los eventos de recolección de botellas que modifican los mismos.

En *enemies.wlk*, los distintos humanos y animales heredan de la clase *Enemy* que a su vez hereda de *VisualObjects*.

---

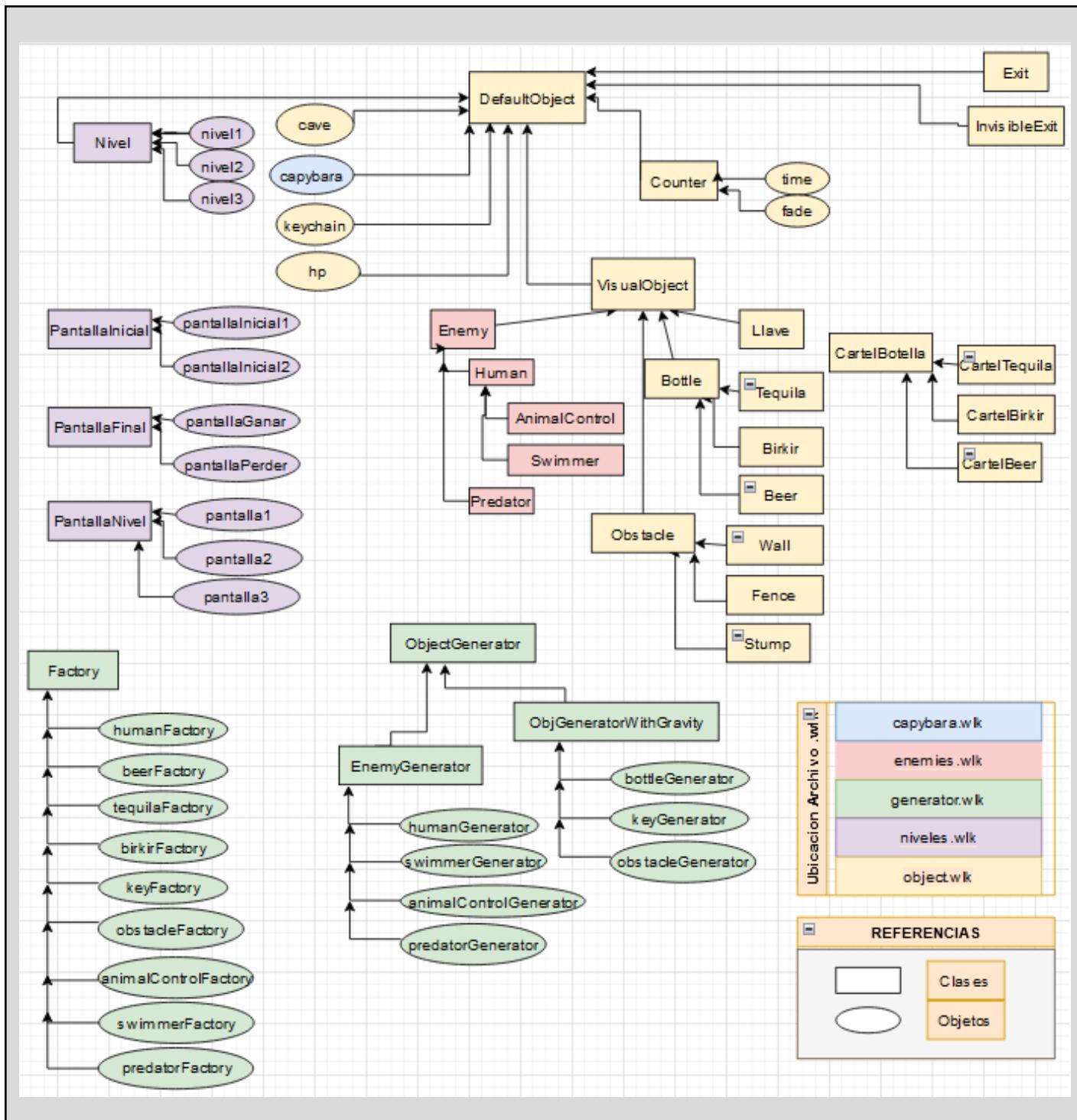
Lo mismo sucede en *niveles.wlk*, donde cada nivel toma como base la clase Nivel. La pantalla principal y las de instrucciones, lo hacen de PantallaInicial; y las pantallas de ganar o perder, de PantallaFinal. Por otro lado, las inter pantallas de nivel, heredan de PantallaNivel.

En *capybara.wlk*, el objeto capybara hereda de la clase DefaultObjects en *objects.wlk*, la cual se usa para definir si un objeto es considerado un obstáculo.

En *objects.wlk* los objetos visibles y las salidas del juego invisibles heredan de DefaultObjects; los obstáculos, llaves y botellas heredan de VisualObjects. A su vez, las botellas heredan de una subclase: Bottle, y las paredes y rejas, de la subclase: Obstacles. Para mostrar el efecto de las botellas, cada cartel de distinto efecto hereda de una clase CartelBotella, y para los efectos del contador de tiempo y fade, se hereda de una clase Counter. Sin embargo la visualización de ambos en hp y keychain, se ven como herencia de DefaultObjects.

A continuacion se puede ver el cuadro de todas las clases creadas con aquellas clases u objetos que heredan de ellas:





**Redefinición y Super:** en `objects.wlk` existe el caso de las botellas, donde si bien existe una clase `Bottle` con un método “`taken`” que es abstracto, cada una de las subclases lo redefinirá a lo que necesite que realice ese método en su situación particular pero podrá seguir refiriéndose a la clase de la que hereda con `super()`:

```
class Bottle inherits VisualObject{
    var property cartel

    method taken(visual){
        if(! game.hasVisual(cartel)){
            game.addVisual(cartel)
            game.schedule(3000, {game.removeVisual(cartel)})
        }
    }
    override method borrar(){
        bottleGenerator.borrar(self)
    }
    override method crash(visual){
        visual.drinkBottle(self)
        game.removeVisual(self)
        self.borrar()
    }
}
```

```
class Beer inherits Bottle (cartel = cartelBeer){ // desacelera el tiempo osea sube el tiempo de gravedad
    var property timeDown = 100
    method image() = "beer.png"
    override method taken(visual){
        super(cartel)
        nivelActual.is().desacelerar(timeDown)
    }
}
class Tequila inherits Bottle (cartel = cartelTequila) { //acelera tiempo osea baja el tiempo de gravedad
    var property timeUp = 100
    method image() = "tequila.png"
    override method taken(visual){
        super(cartel)
        nivelActual.is().acelerar(timeUp)
    }
}
class Birkir inherits Bottle (cartel = cartelBirkir) { //aumenta la vida
    var property lifeUp = 10
    method image() = "birkir.png"
    method winLives(won){
        game.removeVisual(hp)
        const newLife = capybara.life() + won
        if (newLife > capybara.maxLife() )
            capybara.life(capybara.maxLife())
        else
            capybara.life(newLife)
        game.addVisual(hp)
    }
    override method taken(visual){
        super(cartel)
        self.winLives(lifeUp)
    }
}
```

En `niveles.wlk`, las pantallas de inicio y las pantallas de carga de nivel, también deben sobreescribir métodos de las clases de las que heredan, pero tomando algunos de sus elementos con `super()`:

```
class PantallaInicial {
    var property image
    var property pista
    var property siguiente

    method enterParaJugar() {
        keyboard.enter().onPressDo({ self.finalizar() })
    }
    method finalizar() {
        game.clear()
        self.siguiente().iniciar()
    }
}
```

```
object pantallaInicial2 inherits PantallaInicial
    (image = "amigosenemigos.jpg", pista = pistaInicial, siguiente = pantalla1){
        override method finalizar() {
            super()
            self.pista().stop()
        }
    }
```

```
class PantallaNivel {
    var property image

    method iniciar() {
        game.clear()
        game.addVisualIn(self, game.at(0,0))
    }
}
```

```
object pantalla1 inherits PantallaNivel (image = "nivell1.png") {
    override method iniciar() {
        super()
        game.schedule(1000, { game.clear()
            nivell1.cargar()
        })
    }
}
```

Aquí mismo se producen las mismas condiciones para los distintos niveles que heredan de la clase Nivel.

**Game:** además de las funciones básicas de game.\* , en el movimiento de capybara necesitamos usar las flechas del teclado con “keyboard” y “onPressDo()” en niveles.wlk:

```
keyboard.left().onPressDo( { capybara.mover(izquierda) } )  
keyboard.right().onPressDo( { capybara.mover(derecha) } )  
keyboard.up().onPressDo( { capybara.mover(arriba) } )  
keyboard.down().onPressDo( { capybara.mover(abajo) } )
```

- **onTick(milliseconds, name, action)** -> fue posible cronometrar la gravedad de los distintos elementos mediante el game.onTick() en generador.wlk (donde se encuentra el metodo) y niveles.wlk (donde se encuentra el tiempo).

```
method applyGravityAllCollection(){  
    genObjects.forEach( { obj => obj.gravity() } )  
}  
method onTickGenerator(){  
    game.onTick(timeTickGen, nameOnTickGenerator, { self.generate() })  
}  
method gravityOn(){  
    game.onTick(timeGravity, nameGravityOn, {self.applyGravityAllCollection()} )
```

```
override method initTimeGenerator() = 2000  
override method initTimeGravity() = 700
```

- **schedule(milliseconds, action)** -> permite programar que un elemento realice una acción/ocurra un evento en un momento determinado. En nuestro caso, se utiliza como modo de limitar la duración total del juego y para pasar de la pantalla inicial a la pantalla de comandos y entre las pantallas de nivel.

```
game.schedule(60000, { capybara.timeOver() })
```

- **keyboard.enter().onPressDo({game.start()})** -> lo utilizamos como modo de comenzar el juego una vez mostrados los comandos

```
method enterParaJugar() {  
    keyboard.enter().onPressDo({ self.finalizar() })  
}
```

- **Fondos del tablero** -> utilizamos una imagen de inicio fija de fondo, mientras que las demás están programadas para cargar según el nivel en el que esté el juego mediante un addVisualIn(). De la misma manera se manejan las imágenes de inicio, fin, entre-niveles, carteles y contadores visuales.

```
class Nivel inherits DefaultObject {  
    . . .  
  
method cargar() {  
    enCurso = true  
    time.resetCounter()  
    game.clear()  
    game.addVisualIn(self, game.at(0,0))  
    game.addVisual(capybara)  
    game.errorReporter(capybara)  
    game.addVisual(hp)  
    game.addVisual(time)  
    game.addVisual(keychain)  
  
object nivel1 inherits Nivel(image ="fondo_nivel1.jpg", nivel = 1, pista = musicaNivel1,  
                                imagenInicioNivel = "nivel1.png", exit = "wallcrack.png") {  
    override method cargar() {  
        capybara.keysForWin(4)  
        super()  
    }  
}
```

- **Música** -> en sonido.wlk se encuentran cargadas todas las pistas que decidimos utilizar, además del método musicaOnOFF() que permite pausarlas y reproducirlas. Luego, utilizando el mensaje play(), pudimos agregarle música a las distintas etapas del juego llamando cada pista desde niveles.wlk y removerlas antes de la siguiente pantalla:

```
const pistaInicial = game.sound("Simple Orange.mp3")
const musicaNivel1 = game.sound("China.mp3")
const musicaNivel2 = game.sound("Boat 14.mp3")
const musicaNivel3 = game.sound("Crazy black light.mp3")
const sonidoGanar = game.sound("win.mp3")
const sonidoPerder = game.sound("lose.mp3")
```

```
method musicaOnOff(pista) {
    keyboard.m().onPressDo({ if (pista.paused()) {
        pista.resume()
    } else {
        pista.pause()
    }
})
}
```

```
Keyboard.down().onPressDo( [ copybara ]
self.pista().play()
musicConfig.musicaOnOff(self.pista())
game.onCollision([copybara] [ comando ])
```

```
self.pista().stop()
```

**Randomizer:** la aleatoriedad con la que deben aparecer en pantalla los enemigos/botellas/llaves/etc. requirió un archivo especial para permitirlo: randomizer.wlk. Allí dentro podemos encontrar dos métodos: uno que define los límites del área donde deberán aparecer los objetos y otro que encuentra un espacio vacío dentro de dicha área donde cargar un objeto con el mensaje getObjectsIn(position).

```
import wollok.game.*  
object randomizer {  
  
    method position() {  
        return game.at( (1 .. game.width() - 2 ).anyOne(), game.height() - 2 )  
    }  
    method positionOnlyRight() {  
        return game.at( ( 10 .. game.width() - 2 ).anyOne(), game.height() - 2 )  
    }  
    method emptyPositionRight() {  
        const position = self.positionOnlyRight()  
        if(game.getObjectsIn(position).isEmpty()) {  
            return position  
        }  
        else {  
            return self.emptyPositionRight()  
        }  
    }  
    method positionOnlyLeft() {  
        return game.at( ( 1 .. game.width() - 5 ).anyOne(), game.height() - 2 )  
    }  
  
    method emptyPositionLeft() {  
        const position = self.positionOnlyLeft()  
        if(game.getObjectsIn(position).isEmpty()) {  
            return position  
        }  
        else {  
            return self.emptyPositionLeft()  
        }  
    }  
    method emptyPosition() {  
        const position = self.position()  
        if(game.getObjectsIn(position).isEmpty()) {  
            return position  
        }  
        else {  
            return self.emptyPosition()  
        }  
    }  
}
```

---

**Errores:** frente a la posibilidad de interrumpir visualmente el juego, no se utilizaron detecciones de errores, sino más bien se trató de buscar la forma que no se llegasen a estos. Por ejemplo, en el caso de la pausa de la música, donde si una pista se pausa cuando ya está pausada o se la intenta correr cuando ya estaba corriendo, esto produce un error, se utilizó entonces un condicional, evitándolo:

```
object musicConfig {  
    method musicaOnOff(pista) {  
        keyboard.m().onPressDo({ if (pista.paused()) {  
            pista.resume()  
        } else {  
            pista.pause()  
        }  
    })  
}
```

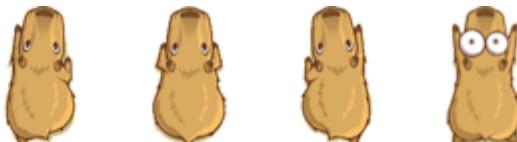
---

## Elementos Utilizados:

### IMÁGENES:

Editadas en Photoshop® del tamaño aproximado 70\*70 pixeles (dependiendo sus dimensiones originales) en formato png, excepto los fondos de los niveles y pantallas de cambio que tienen un tamaño mayor de 700\*700 pixeles.

- **Capybaras** (personaje principal. Una imagen distinta según la dirección)



- **Enemigos** (restan 5 de vida)

- *Nivel 1:* obreros



- *Nivel 2:* control animal



- *Nivel 3:* depredadores + nadadores



- 
- **Botellas** (actúan como pociónes de vida/poderes)

- *Birkir* (elixir de vida +10)



- *Tequila* (acelerar el tiempo)

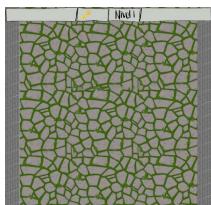


- *Cerveza* (desacelerar el tiempo)

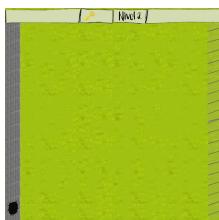


- **Paisajes**

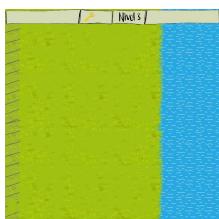
- *Nivel 1:* construcción + pared de ladrillo + agujero cuando recoge 4 llaves



- *Nivel 2:* reja con espacio cuando recoge 6 llaves



- *Nivel 3:* pasto y agua para distintos animales + madriguera cuando recoge 6 llaves



---

- **Obstáculos** (restan 10 de vida)

- *Nivel 1:* paredes



- *Nivel 2:* rejas



- *Nivel 3:* troncos



- **Llaves** (necesarias para desbloquear el siguiente nivel)



- Pantallas

Inicio	Enemigos	Comandos
Nivel 1	Nivel 2	Nivel 3
Ganaste	Perdiste	

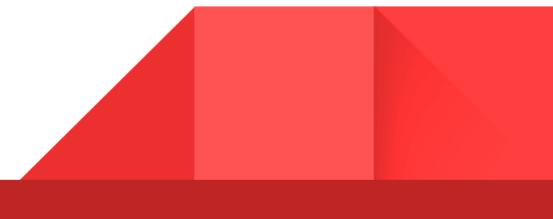
- 
- **Vida** ( con disminución de 5 en 5)



---

## NÚMEROS

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60									



---

## PISTAS

Pistas recortadas para minimizar la carga del juego y adecuadas al cronómetro de pantalla mediante Super Sound App® .

- **Pista inicial:** "Simple Orange" by Pics Frunk - Alejandro Nava ®
- **Nivel 1:** "China" by Pics Frunk - Alejandro Nava ®
- **Nivel 2:** "Boat 14" by Pics Frunk - Alejandro Nava ®
- **Nivel 3:** "Crazy black light" by Pics Frunk - Alejandro Nava ®
- **Ganar:** "Win" by The Romanovs - White Flag ®
- **Perder:** "Lose" by Lineage II game OST - Spring Awakening ®

Alejandro Nava: <https://picsfrunk.bandcamp.com/>

---

© Capybara Team