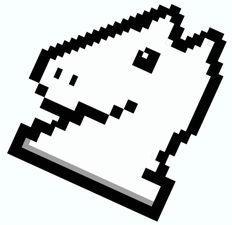
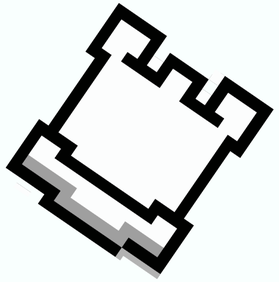
Octubre 2030

CHESS
WARS

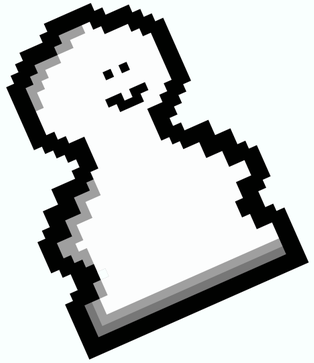





Logan Guerrero Diaz
Rogelio Guerrero Reyes

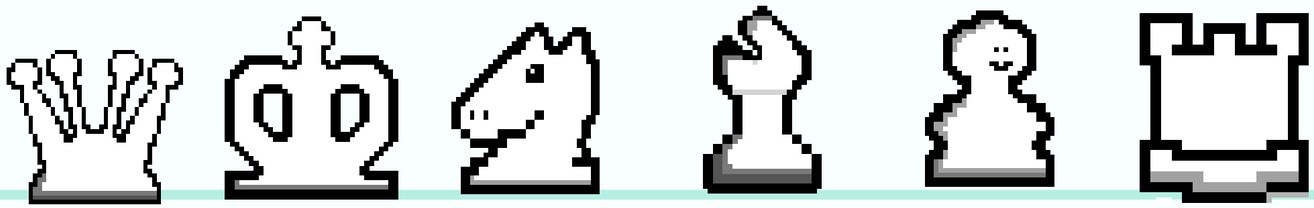
Colonia

La colonia consiste en dos bandos enemigos de ajedrez, entre piezas del mismo bando no se atacan y su objetivo es perdurar y vencer a la otra, y tu como jugador puedes incentivar las deciciones de las piezas

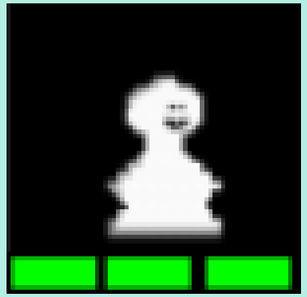


PERSONAJES

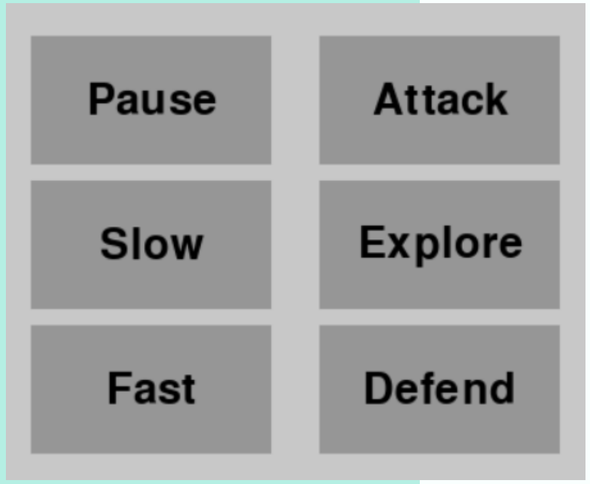
- Peon: Avanza una o dos casillas come en diagonal
- Caballo: Se mueve como caballo
- Alfil: En diagonal con un rango limitado de 3
- Torre: En linea rectta con un rango limitado de 3
- Reina: Como torre y alfil
- Rey: Como rey más la capacidad de crear piezas con recursos obtenidos



INFORMACION



Barra de estamina, los personajes ocupan energía para moverse, deciden si es necesario o no hacerlo



Botones de interacción de usuario:

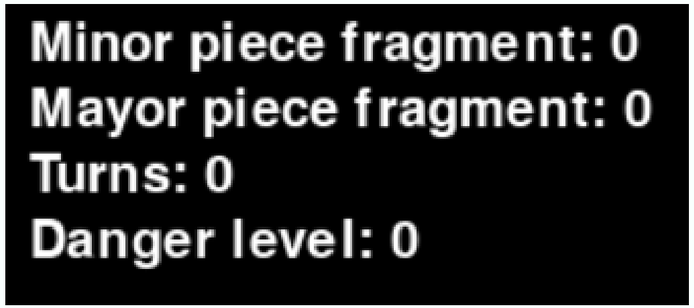
- Velocidad:

Pausa, lento, rápido

- Actitud:

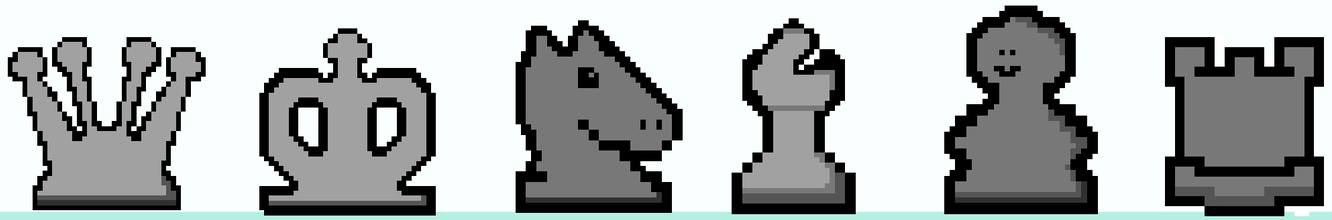
Atacar, explorar, defender

Panel con información general básica



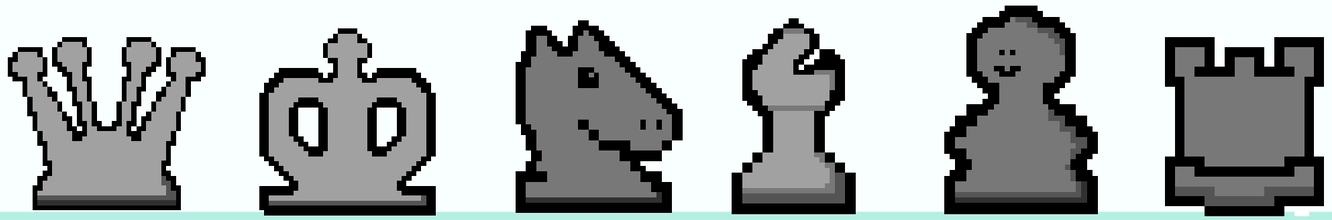
RECURSOS

Las piezas grandes o chicas sueltan fragmentos que son los recursos del juego, tanto como los blancos y los negros pueden obtener este tipo de recurso.



EVENTOS

El rey negro tiene una “IA” enemiga (otro algoritmo de decisión/ dequeue) en la cual va a ejercer precion sobre el rey blanco, decidira atacar, estar defensivo o curioso.



ORDENES

El usuario tendrá la opción de dos opciónes:

ATACAR

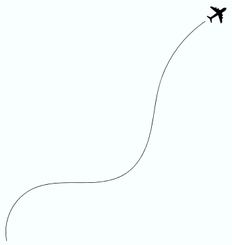
Las piezas tendrán un tono agresivo, priorizaran atacar, y acercarse al rival

DEFENDER

Las piezas tendrán un tono pasivo, priorizaran estar de su lado y protegerse

EXPLORAR

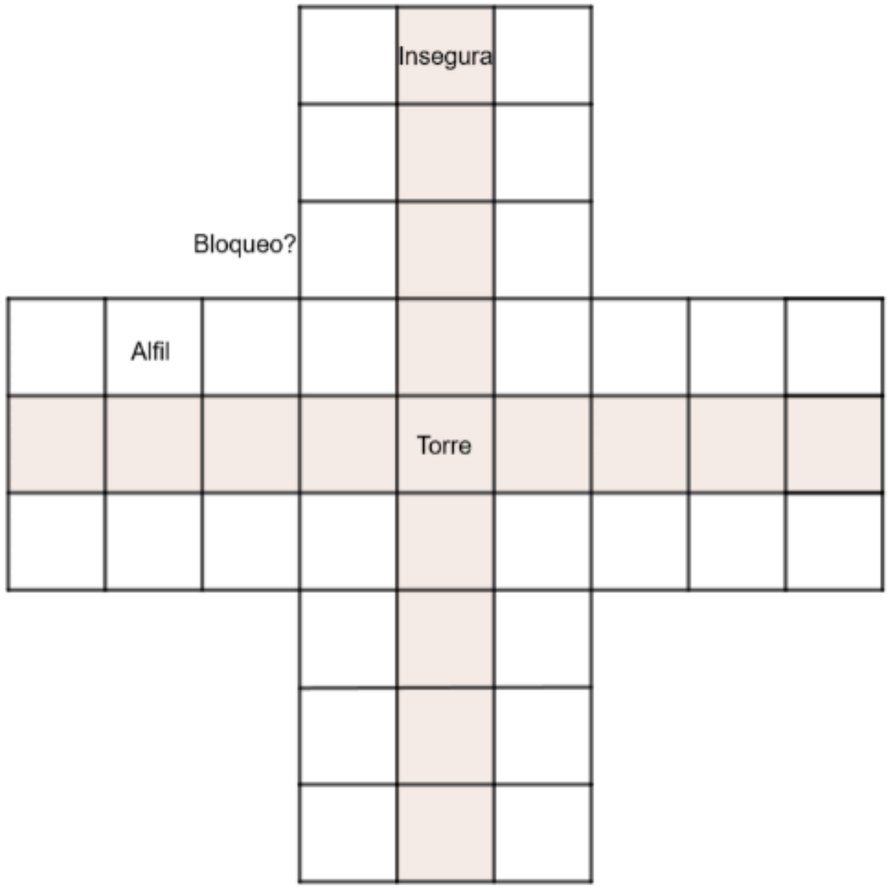
Las piezas tendran iniciativa de recorrer el mapa



Una vez capturada la orden , esta pasa por GUI → grid → Pieces
Para que afecte en la toma de desiciónes

ALGORITMO DE DECISION

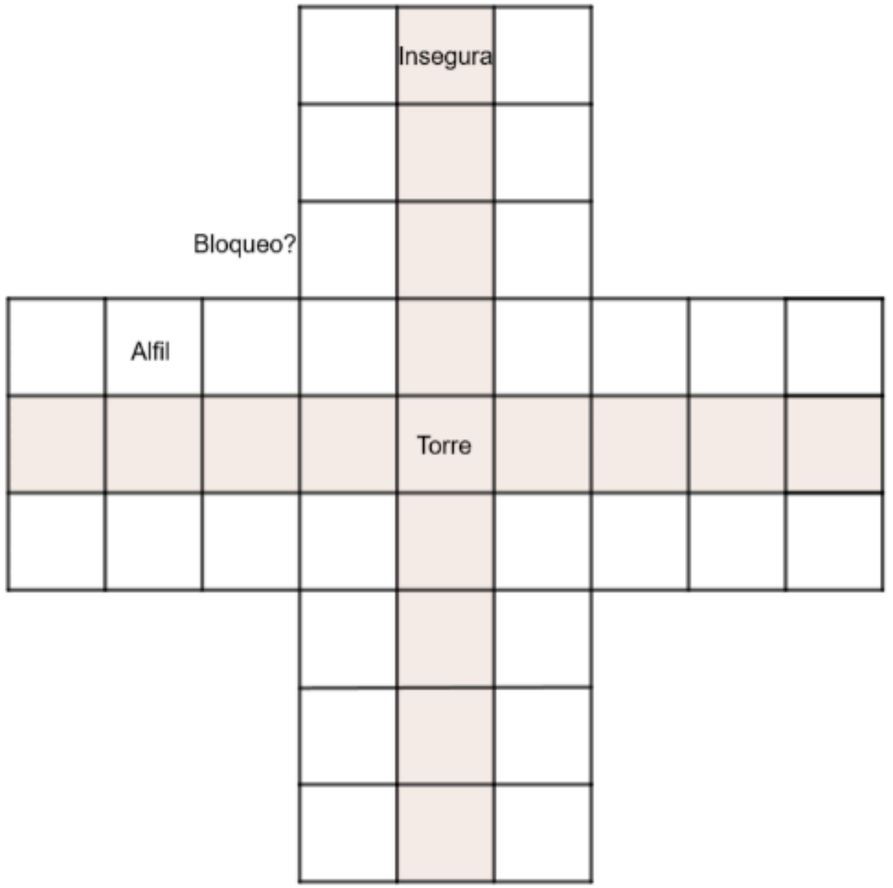
Cada pieza tiene un movimiento único
y una visión única alrededor
de sus movimientos



ALGORITMO DE DECISION

En base a esta información la pieza analiza diversos ataques o intersacciónes que el conozca, y las mide en base a sus pesos

```
# Weights for different situations
self.value = 10
self.support = 8
self.attacked = -15
self.recomended_x = ( 8 if self.team == 0 else GRID_WIDTH - 8)
self.initiative = 10
self.restore = 10
```



ALGORITMO DE DECISION

Fracción de código de toma de decisón

```
def tick(self):
            # Weight of position
            cells.add((x, y),0)

            if self.team == 0 and pos_x <= self.recomended_x:
                cells.change_priority((x,y), 10 * abs((x - pos_x)) + abs(y -pos_y))
            elif self.team == 1 and pos_x >= self.recomended_x:
                cells.change_priority((x,y), 10 * abs((x - pos_x)) + abs(y -pos_y))
            else:
                cells.change_priority((x,y), 5 * abs((x - pos_x)) + abs(y -pos_y))


    for x,y in vision:
        piece = grid.get_piece((x ,y))
        if piece:
            for u,v in piece.get_moves(vision,0):
                if (u,v) in moves:
                    # Weight of incoming attacks or support of other pieces
                    if piece.team != self.team:
                        cells.change_priority((u,v), self.attacked)
                    else:
                        cells.change_priority((u,v), self.support)

    #Weight of possibles attacks or support

    for x,y in moves:

        piece = grid.get_piece((x,y))

        #Direct attack pieces
        if piece:
            cells.change_priority((x,y), 2 * piece.value + 5)

        self.position = pygame.Vector2(x,y)
        moves2 = self.get_moves(vision, attack = False)

        for u,v in moves2:
            if (u,v) == (x,y):
                continue
            if (u,v):
                piece = grid.get_piece((u, v))
                if piece:
                    cells.change_priority((x ,y), self.initiative)
```

ALGORITMO DE DECISION

Las casillas a las que se puede mover son anexadas en un heap y se llama a la funcion change priority para actualizar los datos

```
# Weight of position
cells.add((x, y),0)
```

```
for u,v in piece.get_moves(vision,0):
    if (u,v) in moves:
        # Weight of incoming attacks or support of other pieces
        if piece.team != self.team:
            cells.change_priority((u,v), self.attacked)
        else:
            cells.change_priority((u,v), self.support)
```

ALGORITMO DE DECISION

La complejidad por entrada y manejo de
datos es en tiempo O(log n) por la estructura
de arbol binario del heap implementado

```
class Node:···

class Pqueue:
    def __init__(self, data = None):
        self.__root = None
        self._size = 0

        if data:
            try:
                for d, k in data:
                    self.add(d,k)
            except:
                self.add(data[0], data[1])

    def add(self, data, key):
        new_node = Node( data, key)
        if self.__root:
            self.insert_node(new_node)
            self.__bubble_up(new_node)
        else:
            self.__root = new_node
        self._size += 1
```

OTRAS ESTRUCTURAS DE DATOS

Una queue para manejar los turnos de la piezas, con operaciones para indexar o eliminar, las querys en tiempo de O(1) y removes en tiempo O(n)

```
self.deque[team].push_back(self)
```

```
class Node(Generic[T]):···


class Deque(Generic[T]):
    def __init__(self):
        self.__front = None
        self.__back = None
        self.__size = 0


    def front(self):
        if not self.__front:
            raise Exception("front() from empty deque")

        return self.__front.get_data()


    def push_front(self, data: T):
        new_node = Node(data)
        self.__size += 1
```

MEJORAS FUTURAS

01 Agregar estructuras, o piezas diferentes, (valla, cañon, etc.)

02 Añadir Super piezas que generen eventos como “Mega Torre”, “Peon Gigante”

03 Mejorar la calidad del dibujo y animaciones (GUI)

04 Añadir version premium de paga