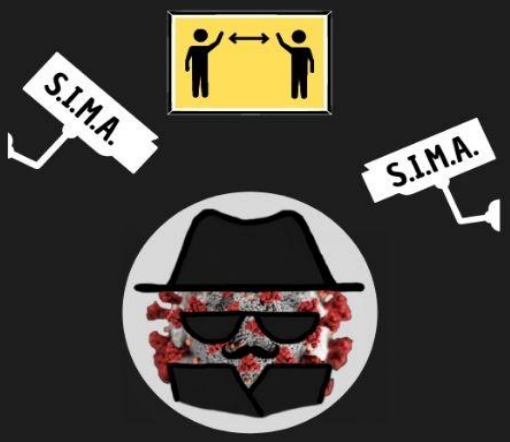
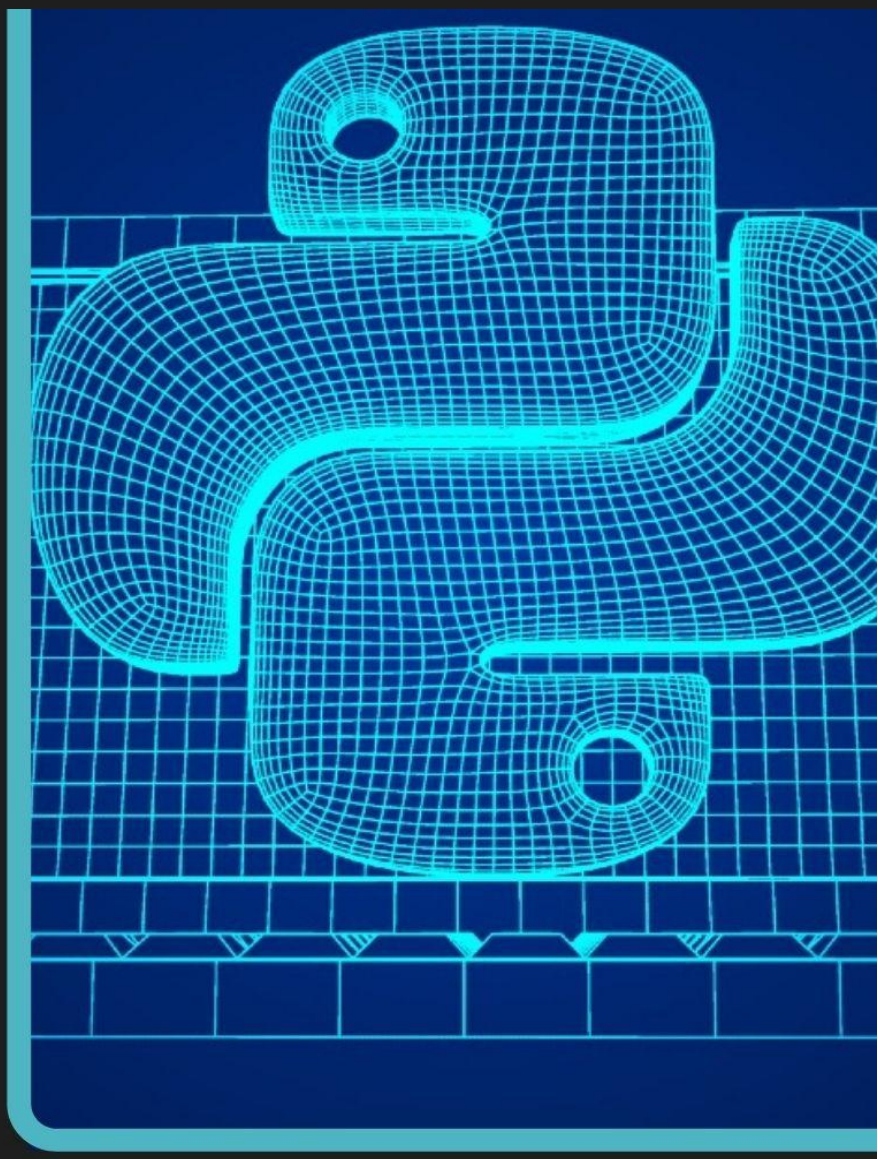


PROJETO S.I.M.A.

(Sistema Inteligente de Monitoramento de Aglomeração)



ESCOLA DE ENGENHARIA DE LORENA - EEL/USP

COMPUTAÇÃO CIENTÍFICA EM PYTHON (LOM3260)

PRINT (MANUAL S.I.M.A.)

EEL-USP

DEZEMBRO DE 2021

O Software

O Sistema Inteligente de Monitoramento de Aglomerações, S.I.M.A., é, como o nome sugere, um sistema que monitora a quantidade de pessoas em um determinado local, simulando essa situação de forma gráfica e interativa.

Objetivos

O objetivo principal para qual o programa foi criado é o de monitorar a possível formação de aglomerações em locais de uso público, tendo em vista a pandemia do COVID-19 e o retorno das aulas presenciais na faculdade.

Em teoria, seriam alimentados ao código dados sobre a quantidade de pessoas nos ambientes selecionados e o código então simularia a situação de cada um dos ambientes em tempo real. Seria possível para o usuário transitar por uma representação dessas salas e observá-las com uma quantidade de pessoas equivalente a real. Seria feita também a comparação entre a quantidade máxima de pessoas permitidas no ambiente e a quantidade presente no momento, avaliando se o risco de contaminação por vírus aéreos é alto ou não, para que o usuário possa, dessa forma, evitar esses ambientes sem ter que visitá-los pessoalmente.

O Código

2.1 Módulos

Visando a objetividade e a melhor estruturação para o código, o grupo tomou como parâmetro a utilização de módulos, o que fortifica a ideia de deixar o uso do software mais simples, implantando os seguintes módulos:

2.1.1 Mapa

O módulo criado para o mapa, tem como principal função a modelação do ambiente que será utilizado na simulação, com isso foram criadas 4 classes em sua composição, sendo elas: Porta, Sala, Corredor e Mapa.

Classe Porta:

O código consiste na modelação da parametragem das portas.

Classe Sala:

O código consiste na modelação da parametragem das salas.

Classe Corredor:

O código consiste na modelação da parametragem do corredor

Classe Mapa:

O código consiste na modelação da parametragem do mapa em geral, ou seja, basicamente une todas as classes, a fim de fazer a interação entre os códigos.

```
class Porta:                            # Cria a classe das Portas
    def __init__(self, x, y, largura, altura, id=-1):  # Define as características básicas das Portas
        self.x = x
        self.y = y
        self.altura = altura
        self.largura = largura
        self.id = id # Identificação do destino da Porta

class Sala:                             # Cria a classe das Salas
    def __init__(self, x, y, largura, altura):  # Define as características básicas das Salas
        self.x = x
        self.y = y
        self.altura = altura
        self.largura = largura
        self.nmax = 0
```

```python
class Mapa:                             # Cria a classe dos Bots
    def __init__(self, start_x, start_y, id): # Define as características básicas dos Mapas
        self.start_x = start_x
        self.start_y = start_y
        self.id = id
        self.salas = []
        self.corredores = []

    def addSala(self, sala):                # Adiciona a sala
        self.salas.append(sala)

    def addCorredor(self, corredor):        # Adiciona o corredor
        self.corredores.append(corredor)

def DesenhaMapa(Mapa, screen):          # Desenha o mapa atual com suas salas e corredores
    """
    Função responsável por desenhar as salas, corredores e portas do Mapa Atual na tela "screen"
    """
    for sala in Mapa.salas:          # Desenha as salas do mapa
        pygame.draw.rect(screen, (211, 211, 211), (sala.x, sala.y, sala.largura, sala.altura))

    for corredor in Mapa.corredores: # Desenha os corredores da sala
        pygame.draw.rect(screen, (169, 169, 169), (corredor.x, corredor.y, corredor.largura, corredor.altura))
        for porta in corredor.porta: # Desenha as portas dos corredores
            pygame.draw.rect(screen, (211, 211, 211), (porta.x + corredor.x, porta.y + corredor.y, porta.largura, porta.alt
```

```python
def MudaMapa(player, mapa_atual, bot_atual, sala_atual, TMapas, TBots, TSalas): # Muda o Mapa atual
    """
    * Suporta um Mapa com vários corredos, mas utilizou-se apenas 1 por Mapa

    Função responsável por mudar o mapa atual após o player passar pela porta.

    Recebe como parâmetros:

    player: O personagem jogável, para executar a função "VerificaPorta" e mudar a posição do player
    mapa_atual: Para verificar os corredos do Mapa
    bot_atual: Para manter os bots caso não passe de Mapa
    sala_atual: Para manter a sala caso não passe de Mapa
    TMapas: Lista completa com todas os Mapas, para identificação
    TBots: Lista completa com todas os Bots, para identificação
    TSalas: Lista completa com todas as Sala, para identificação

    Funcionamento:
    - Para cada corredor do mapa atual, executa "VerificaPorta" para identificar o Id da porta
    que o jogador colidiu, caso não haja um Id (não houve colisão), mantém a Sala, Mapa e Bot atual.
    - Caso ocorra a colisão, é definido um novo mapa, sala e Bot. Além disso, como o ID da porta representa o seu destin
    (Id_porta = 3, leva para o Mapa 3). Ademais, o Id oficialmente usado é (porta_id - 1), pois os IDs da porta começam
    em 1, mas as listas, por definição, começam em zero.
    - Por fim, verifica-se se o If da porta é maior ou menor que o da sala atual, isso representa se ocorrerá o avanço
    ou retrocesso da sala, mudando o local em que o player apareceria.
    """
```

```python
for corredor in mapa_atual.corredores:
    porta_id = corredor.VerificaPorta(pygame.Rect((player.x, player.y, player.raio, player.raio))) # Fornece o Id da
    # porta colidida
    if porta_id != None:            # Isto é, caso ocorra colisão
        id = porta_id - 1
        mapa_novo = TMapas[id]      # Atualiza o mapa atual
        bot_novo = TBots[id]        # Atualiza a lista de bot atual
        sala_nova = TSalas[id]      # Atualiza a sala atual

        # Muda a posição do player ao avançar um mapa
        if mapa_atual.id < porta_id:
            player.x = 1200 - player.x

        # Muda a posição do player ao retornar um mapa
        if mapa_atual.id > porta_id:
            player.x = 1200 - (player.raio + 5) - player.x

    else: # Mantém o mapa, bot e sala atual, caso não ocorra colisão
        mapa_novo = mapa_atual
        bot_novo = bot_atual
        sala_nova = sala_atual

    return[mapa_novo, bot_novo, sala_nova]
```

2.1.2 Estrutura

Responsável por criar a parametragem das salas como valores em metros, realizando o cálculo e devolvendo, a partir do valores de distanciamento, a quantidade de pessoas que podem permanecer, ou seja , a capacidade total de comportamento de pessoas.

```
# Densidade Demográfica - quantidade de pessoas na sala
def DDemo(Largura, Altura, Dist):
    NMax = ((Largura* Altura)//(Dist*2))//375 # Número máximo de pessoas
    if NMax == 0:
        NMax = 1                              # Define o mínimo com 1
    return(NMax)
```

```
if modo == 0: # Modo de criação manual
    NMapa = int(input("Informe o número de salas: "))
    for n in range(1, NMapa+1): # Cria [NMapa] Mapas (Por opção, decidiu-se utilizar uma Sala por mapa, por isso
        # NMapa = NSala)
        xcorredor = 25        # Fixa a posição X do corredor
        ycorredor = 410       # Fixa a posição Y do corredor
        if n == 1 and NMapa > 1: # Primeira Sala
            mapa = Mapa(50, 50, n)
            largura, altura, NBot, Dist, NMax  = Valores()
            salax = 700 - (largura/2)
            salay = ycorredor - altura - 15
            sala = Sala(salax, salay, largura, altura)
            sala.nmax = NMax
            mapa.addSala(sala)

            corredor = Corredor(xcorredor, ycorredor, 1150, 150)
            mapa.addCorredor(corredor)
            corredor.addPorta(Porta(1150, 0, 25, 150, 2))
            ListaB = ListaBot(NBot, Dist, sala)
            player.dist = Dist
            TBots.append(ListaB)
            TMapas.append(mapa)
            TSalas.append(sala)
```

```python
# Define valores: largura e altura da sala, distanciamento e quantidade de pessoas
def Valores():
    """
    Função responsável por definir os valores de comprimento( ou largura), altura (ou profundidade),
    o distânciamento e o número de pessoas(Bots) na sala
    O usuário pode inserir os valores desejados para a simulação e o programa calcula um número máximo de
    pessoas a serem escolhidas, todos esse valores são posteriormente utilizados no programa
    """
    largura = int(input("Informe o comprimento da sala, entre 15m e 90m: "))*10
    while largura < 150 or largura > 900:
        largura = int(input("Insira um valor entre 15m e 90m: "))*10

    altura = int(input("Informe a profundidade da sala, entre 15m e 35m: "))*10
    while altura < 150 or altura > 350:
        altura = int(input("Insira um valor entre 15m e 35m: "))*10

    Dist = float(input("Informe o distânciamento desejado, entre 1m e 4m: "))*10
    while Dist < 10 or Dist > 40:
        Dist = int(input("Insira um valor entre 1m e 4m: "))*10

    NMax = DDemo(largura, altura, Dist)

    NBot = int(input(f"Informe o número de pessoas dentro da sala respeitando o limite de {NMax} pessoas: "))
    while NBot > NMax or NBot < 1:
        NBot = int(input(f"Deve-se conter pelo menos uma pessoa e no máximo {NMax} : "))

    return[largura, altura, NBot, Dist, NMax]



def CriaMapa(player):
    """
    Função responsável por criar todos os mapas e organizar os seus respectivos Bots.
    Inicialmente, o usuário escolhe o modo que deseja utilizar, caso escolha o modo Manual,
    o programa pedirá a quantidade de salas e, em seguida, os parâmetros escolhidos. Assim, os Mapas
    são criados ao mesmo tempo que os dados são fornecidos
    """
    # Saudações iniciais
    print(f"Ola, bem-vindo ao Simulador Inteligente de Monitoramento de Aglomeração, denominado S.I.M.A.\n"
    "Nosso programa busca simular fluxos de pessoas em ambientes limitados, a fim de indicar possíveis aglomerações \n"
    "indesejadas em cenários pandêmicos de patógenos transmitidos por via aérea,\n"
    "como o Sars-Cov-2")

    modo = int(input("Para começar, você deseja utilizar salas prontas? (Sim = 1/Não = 0) "))
    while modo != 1 and modo != 0:
        modo = int(input("Digite um valor válido (Sim = 1/Não = 0) "))

    print("Preparando simulação...")
    TBots = []
    TMapas = []
    TSalas = []
```

```
if modo == 0: # Modo de criação manual
    NMapa = int(input("Informe o número de salas: "))
    for n in range(1, NMapa+1): # Cria [NMapa] Mapas (Por opção, decidiu-se utilizar uma Sala por mapa, por isso
        # NMapa = NSala)
        xcorredor = 25        # Fixa a posição X do corredor
        ycorredor = 410       # Fixa a posição Y do corredor
        if n == 1 and NMapa > 1: # Primeira Sala
            mapa = Mapa(50, 50, n)
            largura, altura, NBot, Dist, NMax  = Valores()
            salax = 700 - (largura/2)
            salay = ycorredor - altura - 15
            sala = Sala(salax, salay, largura, altura)
            sala.nmax = NMax
            mapa.addSala(sala)

            corredor = Corredor(xcorredor, ycorredor, 1150, 150)
            mapa.addCorredor(corredor)
            corredor.addPorta(Porta(1150, 0, 25, 150, 2))
            ListaB = ListaBot(NBot, Dist, sala)
            player.dist = Dist
            TBots.append(ListaB)
            TMapas.append(mapa)
            TSalas.append(sala)


    elif n == 1 and NMapa == 1: # Sala Única
        mapa = Mapa(50, 50, n)
        largura, altura, NBot, Dist, NMax  = Valores()
        salax = 700 - (largura/2)
        salay = ycorredor - altura - 15
        sala = Sala(salax, salay, largura, altura)
        sala.nmax = NMax
        mapa.addSala(sala)

        corredor = Corredor(xcorredor, ycorredor, 1150, 150)
        mapa.addCorredor(corredor)
        ListaB = ListaBot(NBot, Dist, sala)
        player.dist = Dist
        TBots.append(ListaB)
        TMapas.append(mapa)
        TSalas.append(sala)
```

```
elif n > 1 and n < NMapa and NMapa != 1: # Salas intermediárias
    mapa = Mapa(1200 - player.x, 600 - player.y, n)
    largura, altura, NBot, Dist, NMax  = Valores()
    salax = 700 - (largura/2)
    salay = ycorredor - altura - 15
    sala = Sala(salax, salay, largura, altura)
    sala.nmax = NMax
    mapa.addSala(sala)

    corredor = Corredor(xcorredor, ycorredor, 1150, 150)
    mapa.addCorredor(corredor)
    corredor.addPorta(Porta(-25, 0, 25, 150, n - 1))
    corredor.addPorta(Porta(1150, 0, 25, 150, n + 1))
    ListaB = ListaBot(NBot, Dist, sala)
    player.dist = Dist
    TBots.append(ListaB)
    TMapas.append(mapa)
    TSalas.append(sala)
```

```
elif n == NMapa: # Ultima Sala
    mapa = Mapa(1200 - player.x, 600 - player.y, n)
    largura, altura, NBot, Dist, NMax  = Valores()
    salax = 700 - (largura/2)
    salay = ycorredor - altura - 15
    sala = Sala(salax, salay, largura, altura)
    sala.nmax = NMax
    mapa.addSala(sala)

    corredor = Corredor(xcorredor, ycorredor, 1150, 150)
    mapa.addCorredor(corredor)
    corredor.addPorta(Porta(-25, 0, 25, 150, n - 1))
    ListaB = ListaBot(NBot, Dist, sala)
    player.dist = Dist
    TBots.append(ListaB)
    TMapas.append(mapa)
    TSalas.append(sala)
```

```
elif modo == 1: # Modo pré-definido
    player.dist = 15
    mapa_1 = Mapa(50, 50, 1)                              # Cria mapa 1
    sala_1_mapa_1 = Sala(250, 45, 900, 350)               # Cria Sala 1
    mapa_1.addSala(sala_1_mapa_1)                         # Add Sala 1 no mapa 1
    sala_1_mapa_1.nmax = 15

    corredor_1_mapa_1 = Corredor(25, 410, 1150, 150)      # Cria corredor 1
    mapa_1.addCorredor(corredor_1_mapa_1)                 # Add corredor 1 no mapa 1

    corredor_1_mapa_1.addPorta(Porta(1150, 0, 25, 150, 2)) # Add Porta para o mapa 2 no mapa 1

    ##############################

    mapa_2 = Mapa(1200 - player.x, 600 - player.y, 2)
    corredor_1_mapa_2 = Corredor(25, 410, 1150, 150)
    corredor_1_mapa_2.addPorta(Porta(-25, 0, 25, 150, 1))
    corredor_1_mapa_2.addPorta(Porta(1150, 0, 25, 150, 3))

    mapa_2.addCorredor(corredor_1_mapa_2)
    sala_1_mapa_2 = Sala(250, 45, 600, 350)
    mapa_2.addSala(sala_1_mapa_2)
    sala_1_mapa_2.nmax = 15
```

```
    ##############################

    mapa_3 = Mapa(1000 - player.x, 600 - player.y, 3)
    corredor_1_mapa_3 = Corredor(25, 410, 1150, 150)
    corredor_1_mapa_3.addPorta(Porta(-25, 0, 25, 150, 2))

    mapa_3.addCorredor(corredor_1_mapa_3)
    sala_1_mapa_3 = Sala(250, 45, 750, 350)
    sala_1_mapa_3.nmax = 15
    mapa_3.addSala(sala_1_mapa_3)

    ##############   Listas   #############

    ListaB1 = ListaBot(15, 15, sala_1_mapa_1)
    ListaB2 = ListaBot(10, 15, sala_1_mapa_2)
    ListaB3 = ListaBot(14, 15, sala_1_mapa_3)

    TBots = [ListaB1, ListaB2, ListaB3]                   # Lista com todas as listas de bots
    TMapas = [mapa_1, mapa_2, mapa_3]                     # Lista com todos os mapas
    TSalas = [sala_1_mapa_1, sala_1_mapa_2, sala_1_mapa_3] # Lista com todos as salas

return[TBots, TMapas, TSalas]
```

```
# Executa a estrutura do programa com os valores estabelecidos
def Estrutura(TMapas, TBots, TSalas, player):
    """
    Função essencial para a simulação, após os dados serão definidos pela função "Valores()", "Estrutura()"
    terá o papel de executar e gerencias a simulação

    O que faz?
    - __init__() : Define as bases do programa e recebe os parâmetro básicos de Tela

    - initSima (): Cria a tela, define a fonte e o tempo

    - Interface(): Em construção, visa gerar um menu inicial para o programa

    - SimaMain(): Executa as funções de Evento, Update e Renderização, além de ser a responsável por manter
    a execução do programa, juntamento com o seu encerramento

    - SimaEvent(): Verifica se "Quit" ou "Esc" foram precionados para informar à "SimaMain()" o fim da simulaçao

    - SimaUpdate(): Realiza a atualização da simulação, Muda o mapa atual, atualiza os Bots e o Player, calcula
    o número de pessoas e define o estado da porta

    - SimaRender(): Atualiza a tela, Desenha e Renderiza os corredos, salas, mapas, portas, bots e players.
    Permite a visualização do que está ocorrendo

    """



class SIMA():
    def __init__(self, screenSize = (1200, 600), fps=60, title='S.I.M.A', icon=None):

        self.SimaRunning = True                 # Status do programa (True = Em execução)
        self.screenSize = screenSize            # Tamanho da tela
        self.title = title                      # Título
        self.icon = icon                        # Icon
        self.fps = fps                          # FPS

        self.initSima()

        self.mapa_atual = TMapas[0]             # Define o Mapa atual como o primeiro da lista geral
        self.bot_atual = TBots[0]               # Define o conjunto de Bots atual como o primeiro da lista geral
        self.sala_atual = TSalas[0]             # Define a sala atual como a primeira da lista geral
```

```python
def initSima(self):
    # Define a Tela,  a fonte do texto e "relógio" da simulação
    self.screen = pygame.display.set_mode(self.screenSize)

    pygame.display.set_caption(self.title)

    if(self.icon != None):
        pygame.display.set_icon(self.icon)

    pygame.font.init()

    self.start = False

    self.SimaClock = pygame.time.Clock()
    self.SimaFont = pygame.font.SysFont('Arial', 25)

def SimaMain(self):
    # Limpa/ atualiza a tela, marca o tempo e realiza as etapas da simulação
    while self.SimaRunning:
        self.deltaTime = self.SimaClock.tick(self.fps)

        for event in pygame.event.get():
            self.SimaEvent(event)
        self.SimaUpdate()
        self.SimaRender()

        pygame.display.update()

    pygame.display.quit()
```

```python
def SimaEvent(self, event):
    # Verifica se o "QUIT" ou "ESC" foram precionados, se sim, fecha a simulação
    if(event.type == pygame.QUIT):
        self.SimaRunning = False
    if(event.type == pygame.KEYDOWN):
        if(event.key == pygame.K_ESCAPE):
            self.SimaRunning = False

def SimaUpdate(self):
    # Retorna a tecla precionada
    keys = pygame.key.get_pressed()

    # Muda o Mapa atual
    self.mapa_atual, self.bot_atual, self.sala_atual = MudaMapa(player, self.mapa_atual, self.bot_atual, self.sala_atual,

    # Nº de pessoas na sala
    self.NP = NPessoas(self.bot_atual, player, self.sala_atual)

    # Atualiza os Bots que estão na tela
    Atualiza(self.bot_atual, self.sala_atual, self.deltaTime, player, keys, self.screen)

    # Fecha a porta e permite o usuário usar a função "Restaura"
    self.sala_color = Restaura(player, self.sala_atual, self.NP, keys)
```

```python
def SimaRender(self):
    # Limpa a Tela
    self.screen.fill((70,130,180))

    # Desenha o mapa atual com suas salas e corredores
    DesenhaMapa(self.mapa_atual, self.screen)

    # Porta
    pygame.draw.rect(self.screen, self.sala_color ,(((self.sala_atual.x + (self.sala_atual.largura/2)))-75, 395, 150, 15))

    # Desenha os bots atuais
    DesenhaBots(self.bot_atual, self.screen)

    # Desenhando personagem
    pygame.draw.circle(self.screen, player.color, (player.x, player.y), player.raio)

    # Mostra o nº de pessoas na sala
    N = self.SimaFont.render(f'Nº de pessoas na sala: {self.NP}', True, (255, 255, 255))
    self.screen.blit(N, (10, 5))

    # Mostra o nº máximo de Bots na Sala
    NM = self.SimaFont.render(f'Nº máximo de pessoas na sala: {int((self.sala_atual).nmax)}', True, (255, 255, 255))
    self.screen.blit(NM, (880, 5))

    # Exibe os alertas de
    AlertaAglo(self.screen, self.NP, self.sala_atual, player)
```

```python
    Simulação = SIMA()
    return(Simulação.SimaMain())
```

2.1.3 Colisão

Como o principal objetivo do software é o controle de fluxo, desenvolveu-se uma mecânica dentro da simulação de colisão, basicamente após a criação dos mapas e quantidade de NPCs (bots) baseados no distanciamento imposto pelo usuário do software, os NPCs se movimentam de forma *randômica* pelas salas quando se chocam há uma mudança de cor (vermelha), para demonstrar que está acontecendo uma quebra de protocolo de distanciamento, o mesmo acontece na interação do usuário com NPCs.

```
class GCollider:
    """
    Classe que determina os parâmetro de colisão
    """
    # flag que indica se ocorreu colisão
    has = False
    # informa o último objeto a colidir
    last_id = -1
    # se é um objeto de swap (faz a troca da velocidade)
    swap = False
    # guarda a direção em relação ao último objeto colidido (overlay)
    dir = [0, 0]

  def Colisao(ListaBots): # Função que viabiliza a colisão
      """
      Função que executa a colisão e altera o movimento
      """
      colliderLogic(ListaBots) # Obtém-se os Bots colididos

      # Verifica as colisões e faz a troca das velocidades
      for Bot1 in ListaBots:
          # Se há colisão
          if(Bot1.collider.has):
              # Pega o ultimo objeto que foi colidido
              Bot2 = getCircleById(Bot1.collider.last_id, ListaBots)

              # Remove a sobreposição
              overlapLogic(Bot1, Bot2)
```

```
        # Verifica se os dois objetos são swap, se sim, faz a troca
        # Se não for swap, simplesmente reverte sua movimentação
        if(Bot1.collider.swap and Bot2.collider.swap):
            # swap das velocidades
            v_x = Bot1.v_x
            v_y = Bot1.v_y

            Bot1.v_x = Bot2.v_x
            Bot1.v_y = Bot2.v_y

            Bot2.v_x = v_x
            Bot2.v_y = v_y
        elif(Bot1.collider.swap):
            v = np.sqrt(np.power(Bot1.v_x, 2) + np.power(Bot1.v_y, 2))
            # reverte a movimentação
            Bot1.v_x = Bot1.collider.dir[0] * v
            Bot1.v_y = (-1) * Bot1.collider.dir[1] * v

        elif(Bot2.collider.swap):
            v = np.sqrt(np.power(Bot2.v_x, 2) + np.power(Bot2.v_y, 2))
            # reverte a movimentação
            Bot2.v_x = Bot2.collider.dir[0] * v
            Bot2.v_y = (-1) * Bot2.collider.dir[1] * v

        # Limpa os dados da colisão para verificar a próxima no próximo loop
        Bot2.collider.has = False
        Bot1.collider.has = False

        Bot2.collider.last_id = -1
        Bot1.collider.last_id = -1


def checkColision(Bot1, Bot2): # Verifica se houve colisão
    # Calcula a distância entre os dois circulos
    d = np.sqrt(np.power(Bot2.x - Bot1.x, 2) + np.power(Bot2.y - Bot1.y, 2))
    # Verifica se a distância entre os dois é menor que a soma dos dois raios + o distanciamento, se sim, ocorreu
    #  uma colisão
    if(d < (Bot1.raio + Bot2.raio + Bot1.dist)):
        return True
    # Caso não tenha colisão, retorna falso
    return False
```

```python
        def colliderLogic(ListaBots): # Armazena quem colidiu
            """
            Função responsável por armazenar as colisões entre os Bots
            """
            for Bot1 in ListaBots:
                for Bot2 in ListaBots:
                    if(Bot1.id != Bot2.id and Bot1.collider.has == False):
                        if(checkColision(Bot1, Bot2)):
                            # Os dois guardam os ids um do outro
                            Bot1.collider.last_id = Bot2.id
                            Bot2.collider.last_id = Bot1.id

                            # Indica para os dois que houve colisão
                            Bot1.collider.has = Bot2.collider.has = True
```

```python
def overlapLogic(Bot1, Bot2): # Evita sobreposição
    """
    Função responsável por evitar a colisão e impedir o travamento
    """
    # Salva Bot1 e Bot2
    BotA = Bot1
    BotB = Bot2

    # Verifica se BotB é maior que BotA e se BotA pode trocar, se sim, faz a troca (move-se sempre o menor raio)
    if(BotA.raio < BotB.raio and Bot1.collider.swap):
        BotA = Bot2
        BotB = Bot1

    # Cria um vetor que liga A e B b Vetor = Ponto(B) - Ponto(A)
    v = [BotB.x - BotA.x, BotA.y - BotB.y]

    # Normaliza-se o vetor para saber a direção que se deve mover os Bots
    v_mod = np.sqrt(np.power(v[0], 2) + np.power(v[1], 2))
    v[0] = v[0] / v_mod
    v[1] = v[1] / v_mod

    # Com o vetor normalizado, multiplica-se pelo raio A + raio B + distanciamento
    # Depois, soma-se com a coordenada do A (o outro circulo ficará tangente ao circulo maior ou igual)
    BotB.x = BotA.x + v[0] * (BotB.raio + BotA.raio + Bot1.dist) * 1.05
    BotB.y = BotA.y + (-1)*v[1] * (BotB.raio + BotA.raio + Bot1.dist) * 1.05
```

```python
def overlapLogic(Bot1, Bot2): # Evita sobreposição
    """
    Função responsável por evitar a colisão e impedir o travamento
    """
    # Salva Bot1 e Bot2
    BotA = Bot1
    BotB = Bot2

    # Verifica se BotB é maior que BotA e se BotA pode trocar, se sim, faz a troca (move-se sempre o menor raio)
    if(BotA.raio < BotB.raio and Bot1.collider.swap):
        BotA = Bot2
        BotB = Bot1

    # Cria um vetor que liga A e B b Vetor = Ponto(B) - Ponto(A)
    v = [BotB.x - BotA.x, BotA.y - BotB.y]

    # Normaliza-se o vetor para saber a direção que se deve mover os Bots
    v_mod = np.sqrt(np.power(v[0], 2) + np.power(v[1], 2))
    v[0] = v[0] / v_mod
    v[1] = v[1] / v_mod

    # Com o vetor normalizado, multiplica-se pelo raio A + raio B + distanciamento
    # Depois, soma-se com a coordenada do A (o outro circulo ficará tangente ao circulo maior ou igual)
    BotB.x = BotA.x + v[0] * (BotB.raio + BotA.raio + Bot1.dist) * 1.05
    BotB.y = BotA.y + (-1)*v[1] * (BotB.raio + BotA.raio + Bot1.dist) * 1.05
    # Será movido o que tem o menor raio

    # Salva-se o vetor direção nos dois circulos que corresponde a direção em relação aos circulos
    BotA.collider.dir = v
    BotB.collider.dir = v


    # Retorna o objeto referente ao id passado

def getCircleById(id, ListaBots): # Pega o ultimo Bot colidido
    """
    Função responsável por informar qual Bot colidiu
    """
    for Bot in ListaBots:
        if(Bot.id == id):
            return Bot
    return None
```

2.1.4 Bots&Player

Nessa etapa, ocorre a construção dos NPCs e do usuário, com cores randômicas, exceto a vermelha, que é um alerta de quebra de distanciamento. Possuindo 2 classes para montagem:

- **Parte Inicial do código:**

```python
# Bots&Player.py
import pygame
import numpy as np
import random as rd
import ColisaoT as C

FPS = 60

Angle = (np.pi / 4)

def Randc():                                    # Randomiza uma Cor
    return rd.uniform(0, 255)
```

- **Classe BOT:**

Responsável pela criação dos bots, como a velocidade de movimento e as cores de cada NPC.

```
class BOT:                                                  # Cria as características da classe Bot
    def __init__(self, x, y, velocidade_i, distanciamento, id):
        self.raio = 15
        self.dist = distanciamento

        self.x = x
        self.y = y

        # Define as velocidades
        self.velocidade_i = velocidade_i
        self.v_x = 0
        self.v_y = 0
        self.lv_x = self.v_x
        self.lv_y = self.v_y

        self.color = (Randc(), Randc(), Randc())    # Randomiza a cor do Bot
        self.color_o = self.color                   # Armazena a cor original do Bot
        self.color_a = (255, 0, 0)                  # Define a cor vermelha como a cor de alerta

        self.collider = C.GCollider()               # Define as características e permite a colisão
        self.collider.swap = True                   # Permite a troca de velocidade entre Bots
        self.id = id                                # Identificação do Bot


        # Define a direção do Bot
        self.dir = 1
        if(rd.uniform(0, 1) >= 0.5):
            self.dir = -1

        # Randomiza o ângulo
        randAngle = rd.uniform(-Angle, Angle)

        # Randomiza velocidade
        self.randVelocidade = rd.uniform(self.velocidade_i/1.2, self.velocidade_i*1.2)
        self.v_x = self.dir * self.randVelocidade * np.cos(randAngle)
        self.v_y = self.randVelocidade * np.sin(randAngle)
```

- **Classe Player:**

Responsável pela criação do player (usuário), velocidade, tamanho, dentre outras características.

```
class Player:                                           # Cria as características da classe Player
    def __init__(self, x, y, r):
        self.x = x
        self.y = y
        self.raio = r

        self.color = (255,20,147)                       # Define a cor do player
        self.color_o = self.color                       # Armazena a cor original
        self.color_a = (255, 0, 0)                      # Define Vermelho como a cor de alerta
        self.id = 0                                     # Define a identificação do player como 0

        self.collider = C.GCollider()                   # Define as características e permite a colisão
        self.collider.swap = False                      # NÃO Permite a troca de velocidade do player
        self.dist = 0                                   # Distanciamento incial de 0

        self.last_x = 0                                 # Armazena a velocidade no eixo x
        self.last_y = 0                                 # Armazena a velocidade no eixo y
        self.dir_x = 0                                  # Armazena a direção no eixo x
        self.dir_y = 0                                  # Armazena a direção no eixo y
```

Após a criação de ambos, deve-se criar ou incumbir funções para os NPCs (bots), para que assim possa realizar as interações entre NPCs e Player.

```
def ListaBot(n, dist, local):     # Cria uma lista de [n] Bots com um distanciamento de [dist] em um determinado [local]
    lista = [] # Cria uma lista inicialmente vazia para os Bots
    for i in range(n):

        # Cria n bots com valores randomizados
        Bot = BOT(
        rd.randint(local.x + 25, local.x - 25 + local.largura),
        rd.randint(local.y + 25, local.y - 25 + local.altura),
         (1 / FPS) * 3,
          dist,
           i+1)

        if len(lista) >= 1: # Se já houver um Bot na lista
            for Bot2 in lista: # Pega outro Bot
                if Bot != Bot2: # Caso esses Bots sejam diferentes
                    D = np.sqrt((Bot.x - Bot2.x)**2 +(Bot.y - Bot2.y)**2) # Calcula a distância entre eles
                    while D <= ((2 * Bot.raio) + (2* Bot.dist)): # Enquanto houver sobreposição, muda o x e y
                        Bot.x = rd.randint(local.x + 25, local.x - 25 + local.largura)
                        Bot.y = rd.randint(local.y + 25, local.y - 25 + local.altura)
                        D = np.sqrt((Bot.x - Bot2.x)**2 +(Bot.y - Bot2.y)**2) # Recalcula a distância
            lista.append(Bot) # Adiciona o Bot na Lista
        else:
            lista.append(Bot) # Adiciona o Bot na Lista
    return lista
```

```
def ListaAlerta(ListaBots):              # Cria uma lista de Rects dos Bots de [ListaBots] para alerta
    ListaDist = [] # Cria uma lista inicialmente vazia para Rects de Alerta
    for Bot in ListaBots:
        # Cria um Rect um pouco maior que o Bot, com base no Distanciamento
        b = pygame.Rect(((Bot.x - Bot.raio) - Bot.dist, (Bot.y - Bot.raio) - Bot.dist, (2*Bot.dist) + 2*Bot.raio, (2*
        ListaDist.append(b) # Adiciona o Rect na Lista
    return ListaDist

def AlertaB(ListaBots, ListaA):                # Verifica o Distanciamento dos Bots de [ListaBots] e aciona o  Alert
    for Rect in ListaA:
        colisao = Rect.collidelist(ListaA)      # Verifica se a colisão ocorreu e retorna o indíce do objeto colidido
        Bot1 = ListaBots[ListaA.index(Rect)]    # ListaB[ListaD.index(Rect)] acha o objeto oqual o Rect corresponde
        Bot2 = ListaBots[colisao]               # ListaB[colisao] acha qual foi o objeto colidido
        if Rect != ListaA[colisao]:             # Se os Rects pertencerem a objetos diferentes
            if colisao != -1:                   # Se ocorrer colisão
                Bot1.color = Bot1.color_a       # Muda para a cor de alerta
                Bot2.color = Bot2.color_a
        else:
            Bot1.color = Bot1.color_o           # Volta para a cor original
            Bot2.color = Bot2.color_o
```

```
    def DesenhaBots(ListaBots, screen):    # Desenha [ListaBots] em uma tela [screen]
        for Bot in ListaBots:
            pygame.draw.circle(screen, Bot.color, (Bot.x, Bot.y ), Bot.raio)

    def MovLimite(local, ListaBots, time): # Define o limite do movimento [ListaBots] em certo [local]
        # com base em um certo [tempo]
        for Bot in ListaBots:
            Bot.x += (Bot.v_x) * time                           # Movimenta o Bot
            Bot.y += (Bot.v_y) * time

            if(Bot.y < local.y + Bot.raio):                     # Colisão superior
                Bot.y = local.y + Bot.raio
                Bot.v_y = (-1)*Bot.v_y

            if(Bot.y > (local.y + local.altura) - Bot.raio):  # Colisão inferior
                Bot.y = (local.y + local.altura) - Bot.raio
                Bot.v_y = (-1)*Bot.v_y

            if(Bot.x < local.x + Bot.raio):                     # Colisão lateral (Esquerda)
                Bot.x = local.x + Bot.raio
                Bot.v_x = (-1)*Bot.v_x

            if(Bot.x > (local.x + local.largura) - Bot.raio): # Colisão lateral (Direita)
                Bot.x = (local.x + local.largura) - Bot.raio
                Bot.v_x = (-1)*Bot.v_x
```

Agora, basta criar a interação entre os NPCs e o Player, sendo elas a movimentação do Player pelo mapa, a quebra de distanciamento quando acontecer etc.

```
#### Funções Player + Bots ####

def AlertaP(player, ListaBots, ListaA): # Verifica o Distanciamento do Bot de [ListaBots] com o player e aciona o  Alerta
    rectPlayer = pygame.Rect(((player.x - player.raio) - 8, (player.y - player.raio) - 8, player.raio*2 + 16, player.raio*2
    colisao = rectPlayer.collidelist(ListaA)  # Verifica se a colisão ocorreu e retorna o indíce do objeto colidido
    Bot = ListaBots[colisao]                  # ListaB[colisao] acha qual foi o objeto colidido
    if colisao != -1:                         # Se ocorrer colisão
        player.color = player.color_a         # Muda para a cor de alerta
        Bot.color = Bot.color_a
    else:
        player.color = player.color_o         # Volta para a cor original
        Bot.color = Bot.color_o

def NPessoas(ListaBots, player, sala_atual):  # Conta nº de pessoas (Bot + player) na [sala_atual]
    RectSala = pygame.Rect(((sala_atual.x, sala_atual.y, sala_atual.largura, sala_atual.altura)))

    if RectSala.collidepoint(player.x, player.y): # Se o centro do player está na sala
        N = len(ListaBots) + 1
    else:
        N = len(ListaBots)
    return N
```

```
def MovePlayer(player, keys, time, screen):                 # Movimenta o Player por meio de WASD e salva sua direção

    # Calcula a direção de movimento
    player.dir_x = 0
    player.dir_y = 0

    if keys[pygame.K_d]:                                    # Movimento o player para a direita ao pressionar "D"
        player.dir_x += 1

    if keys[pygame.K_a]:                                    # Movimento o player para a esquerda ao pressionar "A"
        player.dir_x -= 1

    if keys[pygame.K_w]:                                    # Movimento o player para a cima ao pressionar "W"
        player.dir_y -= 1

    if keys[pygame.K_s]:                                    # Movimento o player para a direita ao pressionar "S"
        player.dir_y += 1

    if(player.dir_x != 0 and player.dir_y != 0):            # Normaliza o vetor direção
        tmp_normal = np.sqrt(np.power(player.dir_x, 2) + np.power(player.dir_y, 2))

        player.dir_x = player.dir_x / tmp_normal
        player.dir_y = player.dir_y / tmp_normal
```

```
####  Colisão por pixel  ####

color = screen.get_at((             # Fornece a cor do pixel a frente do player
                        int(player.x + player.dir_x * player.raio*1.2),
                        int(player.y + player.dir_y * player.raio*1.2)
                    ))
if(color != (70,130,180)):          # Se a cor do pixel for diferente da proibida, o Player se move
    if keys[pygame.K_d]:
        player.x += time*0.3

    if keys[pygame.K_a]:
        player.x -= time*0.3

    if keys[pygame.K_w]:
        player.y -= time*0.3

    if keys[pygame.K_s]:
        player.y += time*0.3


#### Colisão por posição (Análise)  ####

def Atualiza(ListaBots, local, time, player, keys, screen): # Realiza todas as atualizações de [ListaBots] e [Player] em um determinado [local] com base em um certo [time]
    ListaD = ListaAlerta(ListaBots)            # Cria uma Lista de Rects para alerta baseada em uma lista de Bots
    ListaGeral = ListaBots + [player]          # Cria uma lista geral de pessoas (Bots + player)

    AlertaB(ListaBots, ListaD)                 # Executa a função de Alerta entre os Bots
    AlertaP(player, ListaBots, ListaD)         # Executa a função de Alerta entre o Player e os Bots
    C.Colisao(ListaGeral)                      # Executa a colisão entre as pessoas (Bots + Player)

    MovLimite(local, ListaBots, time)          # Limita o movimento dos Bots dentro de um determinado local
    MovePlayer(player, keys, time, screen)     # Executa o movimento do player e define a colisão por cores

def Restaura(player, sala_atual, NPessoas, keys):          # Fecha a porta e permite a mudança de posição caso o limite ultrapasse
    # Cria um rect da Sala para colisão
    RectSala = pygame.Rect(((sala_atual.x, sala_atual.y, sala_atual.largura, sala_atual.altura)))

    # Se ultrapassar o limite máximo e o player estiver na sala
    if NPessoas >= (int((sala_atual).nmax)) and RectSala.collidepoint(player.x, player.y):
        porta_color = (70,130,180)             # Fecha a porta

        if keys[pygame.K_r]:                   # Permite a restauração da posição
            player.x = 50
            player.y = 480

    # Se ultrapassar o limite máximo, mas o player não está la
    elif NPessoas >= (int((sala_atual).nmax)):
        porta_color = (70,130,180)             # Simplismente fecha a porta

    else: # Caso contrário deixa aberta
        porta_color = (210, 180, 140)

    return(porta_color)
```

```python
def AlertaAglo(screen, NP, sala_atual, player):             # Mostra o Alerta de aglomeração e o aviso da restauração
    # Cria um rect da Sala para colisão
    RectSala = pygame.Rect(((sala_atual.x, sala_atual.y, sala_atual.largura, sala_atual.altura)))

    aviso = pygame.image.load('Imagens/ATENÇÃO.png').convert()        # Importa a imagem de aviso (atenção)

    avisoR = pygame.image.load('Imagens/ATENÇÃO_R.png').convert()     # Importa a imagem de aviso com restauração

    # Define quando o alerta e o aviso da função Restauradora devem ser acionados
    if NP >= int((sala_atual).nmax) and RectSala.collidepoint(player.x, player.y):
        # Renderiza o Alerta com o aviso da função Restauradora
        screen.blit(avisoR, (5 , 45))

    elif NP >= int((sala_atual).nmax):
        # Renderiza  somente o Alerta
        screen.blit(aviso, (5 , 45))
        pass
```

2.1.5 S.I.M.A

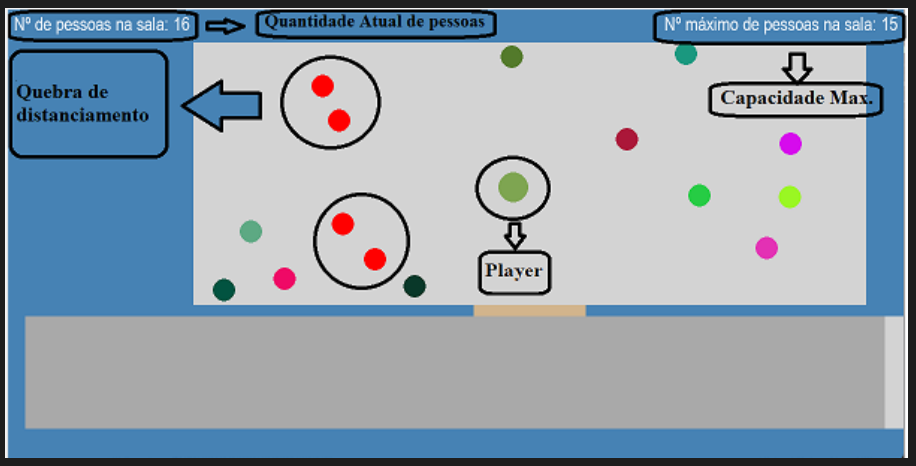
Na última etapa do software, criou-se um código que tem como função importar o módulo Estrutura e adicionar a interface de início da simulação. A partir disso, gera-se a simulação de controle de fluxo, denominada como S.I.M.A. A interface contém um botão de iniciar simples para dar início a simulação.

```
from tkinter import *
from Estrutura import *

###########  Iniciando o programa  ####
TBots, TMapas, TSalas = CriaMapa(player) # Inicia a função que define os parâmetros das salas




###########  Interface ################

def ChamarBotao():                      # Inicia Simulação
    """
    Define a função que determina a ação do botão quando o usuário clica nele
    Quando o usuário clica no botão, a função Estrutura() é chamada e inicia a simulação

    """
    Estrutura(TMapas, TBots, TSalas, player)

def Interface():                        # Define a função que cria a interface do programa

    # Cria a janela e importa a imagem de fundo
    janela = Tk()

    img = PhotoImage(file="Imagens/Projeto.png")

    # Define os parâmetros da janela
    janela.geometry("1300x700")
    janela.title("S.I.M.A")
    janela.configure(bg="black")

    # Cria e posiciona o botão (além de definir o comando que ele executará)
    botao = Button(janela, text="Iniciar", font=("Arialle, 20"), command=ChamarBotao)
    botao.grid(column=0, row=0, padx=0, pady=10)

    # Posiciona a imagem na tela
    label_image = Label(janela, image=img, border="0")
    label_image.grid(padx=30, pady=0)

    # Inicia a janela e mantem ela funcionando
    janela.mainloop()

Interface()                             # Executa a interface inicial do programa
```

Quando processado, o código retorna as seguintes interfaces para o usuário:

Tela de início

Mapa da simulação



Obrigado por utilizar o S.I.M.A.

Desenvolvedores:

Isabela Bruni Moraes

Lucas Rodini Amato

Luisa Kuymjian Belentani

Marcos Rafael da Silva

Miguel Ângelo Machado Rodrigues