Computação Gráfica

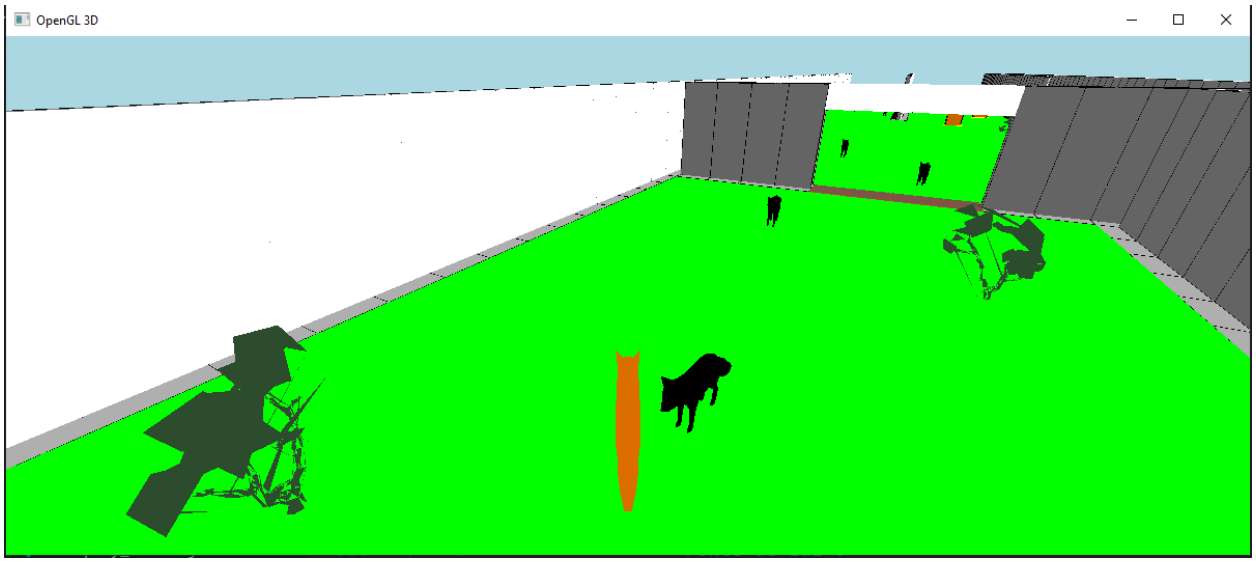
Trabalho 2 – Computação Gráfica 3D

Artur Zanette Marcon e Lucas Martins Weiss

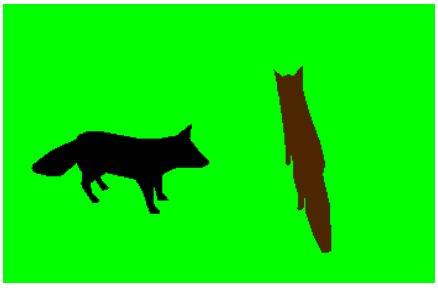
PUCRS- 2024/1

JOGO

O jogo consiste em movimentar um personagem num plano 3d com obstáculos e tentar sobreviver o máximo possível. A cada frame que o jogador anda ele gasta uma quantidade de energia e o objetivo é coletar cápsulas de energia pelo mapa para reabastecer e continuar jogando. Enquanto o jogador faz isso, inimigos vão correr perseguindo o jogador e a cada colisão eles retiram mais um pouco de energia, dificultando o jogo e garantindo que não se pode apenas ficar parado.

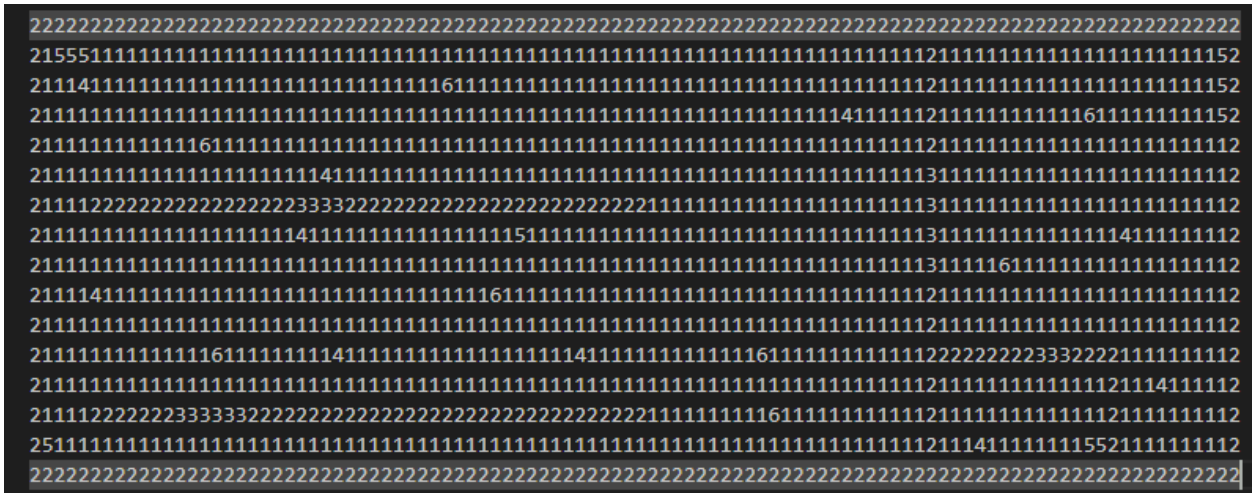


Conforme o jogador vai perdendo energia a cor do personagem vai se esvaindo até ele se tornar corrupto, igual a seus inimigos.



LABIRINTO E OBJETOS

Para a construção do labirinto, foi solicitado que o mapa fosse gerado a partir da leitura de uma matriz lida por um arquivo. Esta matriz deve conter entradas que indicam a localização de diferentes elementos do labirinto como paredes, portas, inimigos, energias e posição inicial do jogador. Este método permite uma maior flexibilidade na hora de modificar o labirinto caso seja necessário.



Para a construção do labirinto foi utilizado a função ‘DesenharPiso()', que foi fornecida no código junto ao material de apoio. Essa função foi adaptada para realizar uma checagem dos valores da matriz. Caso um valor corresponda a um valor que não representa um pedaço do chão, o método modifica o comportamento para substituir o chão pelo elemento apropriado.

Originalmente a função era responsável apenas pela criação do plano XZ, porém agora depois de melhorada também constroi as paredes seguindo as medidas pré determinadas pelo professor.

```python
def DesenharMapa():
    glEnable(GL_LIGHTING)
    global mapa

    glPushMatrix()
    glTranslated(0, -1, 0)
    for x in range(len(mapa)):
        glPushMatrix()
        for z in range(len(mapa[x])):
            if mapa[x][z] == '1' or mapa[x][z] == '5' or mapa[x][z] == '6':
                DesenhaLadrilho(Green, Green)
            elif mapa[x][z] == '2':
                alturaParede = 2.7
                glPushMatrix()
                glTranslatef(0, alturaParede / 2, 0)
                SetColor(Gray)
                gira = False

                if z-1 > 0 and z+1 < len(mapa[0]):
                    if mapa[x][z-1] == '2' or mapa[x][z-1] == '3':
                        gira = True

                if gira:
                    glRotatef(90, 0, 1, 0)

                glScalef(1, alturaParede, 0.25)
                glutSolidCube(1)

                glColor3f(0, 0, 0)
                glScalef(1, 1, 1)
                glutWireCube(1)

                glPopMatrix()
                DesenhaLadrilho(Black, Gray)
            elif mapa[x][z] == '3':
                alturaParede = 2.7
                alturaPorta = 2.1
                espessuraParede = 0.25
                larguraPorta = 1
                SetColor(DarkBrown)
                glPushMatrix()
                glTranslatef(x, 0 + alturaPorta / 2, z)
                glPopMatrix()
                glPushMatrix()
                glTranslatef(0, 0 + alturaPorta + (alturaParede - alturaPorta) / 2, 0)
```

Para o instanciamento da posição do jogador, posição de inimigos, energias e outros objetos adicionais, foi implementada uma varredura inicial na matriz. Essa varredura é realizada no início do código e tem como objetivo instanciar todos esses elementos com base em suas posições especificadas na matriz. Durante esse processo, o código verifica cada célula da matriz para identificar os diferentes tipos de objetos a serem instanciados.

Quando uma parte da matriz contém o valor que representa a posição do jogador, inimigo, energia ou qualquer outro objeto adicional, o código instancia o objeto correspondente na posição especificada. Após a criação do objeto, o valor na matriz é redefinido para corresponder a um ladrilho do chão. Dessa forma, quando a função ‘DesenharMapa()’ for chamada, ela constroi o labirinto garantindo que os objetos estejam corretamente posicionados sobre o chão.

```
for line in range(len(mapa)):
    for i in range(len(mapa[0])):
        if mapa[line][i] == '0':
            mapa[line][i] = '1'
            fox = Ponto(line, -1, i)
        if mapa[line][i] == '5':
            movelPos.append((line, 1, i))
        if mapa[line][i] == '6':
            enemyPos.append((line, -1, i))
```

CÂMERAS

O jogador pode escolher entre 3 tipos de visão, primeira pessoa, terceira pessoa e panorâmica. Para alterar basta apertar ‘f’ e a câmera muda em tempo real.

```
def PosicUser():

    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(60,AspectRatio,0.01,50) # Projecao perspectiva
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    if view == 0:

        distancia = 3


        obsX = fox.x  - vetorAlvo.x * distancia
        obsY = fox.y + distancia
        obsZ = fox.z - vetorAlvo.z * distancia

        gluLookAt(obsX, obsY, obsZ,
                  fox.x, fox.y + 1.5, fox.z,
                  0.0, 1.0, 0.0)
    elif view == 1:

        gluLookAt(10, 40, 50,
                  10, 0, 50,
                  1, 0.0, 0)

    if view == 2:

                gluLookAt(fox.x, fox.y + 1, fox.z,
                  fox.x + vetorAlvo.x, fox.y + 1 + vetorAlvo.y, fox.z + vetorAlvo.z,
                  0.0, 1.0, 0.0)
```

Para implementar esses requisitos, utilizamos o comando ‘gluLookAt()’ da biblioteca do OpenGL, que permite definir a posição da câmera e a direção para a qual ela está apontando.

Para a primeira visualização, configuramos uma distância de 3 metros para deslocar a câmera para uma posição acima e atrás do personagem. Isso resulta em uma

perspectiva em terceira pessoa, onde o jogador pode ver o personagem e o ambiente ao seu redor, oferecendo uma visão ampla e permitindo uma navegação mais intuitiva.

Para a segunda visualização, posicionamos o observador no centro do mapa a uma altura de 40 metros. A câmera é direcionada para a mesma posição do observador, mas com o eixo Y igual a zero. Isso cria uma visão panorâmica que abrange todo o mapa, proporcionando uma visão geral da disposição dos elementos no ambiente.

Por último, projetamos uma câmera em primeira pessoa, posicionando o observador a 1 metro acima do objeto. Isso oferece uma perspectiva imersiva, permitindo que o jogador veja o ambiente diretamente dos olhos do personagem, criando uma sensação de presença e envolvimento mais profundos no jogo. A combinação dessas três visualizações oferece uma experiência diversificada e abrangente, atendendo a diferentes necessidades e preferências dos usuários.

JOGADOR

Nosso jogador foi feito como sendo um ponto que possui um modelo 3d de uma raposa.

Não conseguimos passar os valores da textura, portanto no jogo ela acabou ficando toda laranja.

Para movimentar a raposa se usa as setas do teclado e a barra de espaço para controlar se ela anda ou fica parada. A ideia é que a raposa anda automaticamente

para frente e o jogador altera a direção que ela olha e pode escolher se ela fica parada ou não.

```
def arrow_keys(a_keys, x, y):
    global foxRotacao, foxVetorRotacao
    if a_keys == GLUT_KEY_UP:
        pass
    if a_keys == GLUT_KEY_RIGHT:
        foxRotacao += 12.0
        foxVetorRotacao -= 12.0
        if foxRotacao >= 360.0:
            foxRotacao -= 360.0
    elif a_keys == GLUT_KEY_LEFT:
        foxRotacao -= 12.0
        foxVetorRotacao += 12.0
        if foxRotacao < 0.0:
            foxRotacao += 360.0
    if a_keys == GLUT_KEY_RIGHT:
        pass

    glutPostRedisplay()
```

INIMIGOS

Os inimigos foram feitos de forma semelhante ao jogador, eles possuem o mesmo modelo mas com uma coloração diferente, a ideia é que a raposa principal vai lentamente se transformando em um desses inimigos caso ela não colete energia.

Para movimentação foi usado um loop dentro da classe ‘UpdatePositions()’ que todo frame atualiza a posição de todas as entidades, incluindo inimigos.

```python
def UpdatePositions():
    global fox, foxRotacao, foxSpeed, walk, mapa, vetorAlvo, energia, enemies

    # Calculate the direction vector from each enemy towards the player
    player_pos = Ponto(fox.x, fox.y, fox.z)

    for enemy in enemies:
        posInimigo = Ponto(enemy.x, enemy.y, enemy.z)
        vetorX = fox.x - posInimigo.x;
        vetorZ = fox.z - posInimigo.z;
        anguloRad = math.atan2(vetorZ, vetorX);
        anguloGraus = anguloRad * 180.0 / math.pi;
        enemy.rotacao = -anguloGraus;
        enemy.vetorRotacao = -anguloGraus;
        rad = enemy.rotacao * math.pi / 180.0
        EproxX = enemy.x + math.cos(rad) * enemy.speed
        EproxZ = enemy.z - math.sin(rad) * enemy.speed
        enemy.set_position(EproxX, enemy.y, EproxZ)
        if round(posInimigo.x) == round(fox.x) and round(posInimigo.z) == round(fox.z):
            enemy.set_position(random.randint(0, len(mapa) - 1), -1, random.randint(0, len(mapa[0]) - 1))
            energia += 1
            if energia > 100:
                os._exit(0)
```

A única diferença da movimentação dos inimigos para o da raposa é que eles não possuem colisão com obstáculos (“*Os inimigos devem ser representados por objetos 3D que se movem de forma lógica, tentando colidir com o jogador, e sem colidir com obstáculos. A velocidade de deslocamento dos inimigos ser constante em 20 m/s.”)*

Os inimigos sempre olham para o jogador e andam sempre em direção reta e não podem ser derrotados.

MODELAGEM 3D

Para os modelos usamos o método ‘mLoader’, para ler os vértices e passar para os métodos de desenho.

```python
def mLoader(filename):
    vertices = []
    faces = []

    try:
        with open(filename, 'r') as obj_file:
            for line in obj_file:
                if line.startswith('v '):
                    vertices.append(list(map(float, line.strip().split()[1:])))
                elif line.startswith('f '):
                    face = [int(vertex.split('/')[0]) - 1 for vertex in line.strip().split()[1:]]
                    faces.append(face)
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return None

    return vertices, faces
```

Esta função `Desenhafox` renderiza um modelo de raposa em OpenGL e altera sua cor com base no valor de `energia`. A cor transita de laranja para preto à medida que a energia diminui. Aqui está uma explicação detalhada da função:

1. **Carregando o modelo OBJ**:
    - A função `mLoader` é chamada para carregar os vértices e faces do modelo `Lowpoly_Fox.obj`. Se o modelo não puder ser carregado (`obj_vertices` é `None`), a função sai cedo.
2. **Interpolação de Cor**:
    - Infelizmente não conseguimos implementar.

```python
def Desenhafox():
    global fox, foxVetorRotacao, energia

    # Example loading an OBJ model named 'Lowpoly_Fox.obj'
    obj_vertices, obj_faces = mLoader('Lowpoly_Fox.obj')

    if obj_vertices is None:
        return

    # Calculate color based on energy (transition from orange to black)
    # Example of linear interpolation (lerp) from orange to black
    initial_color = [1.0, 0.5, 0.0]  # Orange
    final_color = [0.0, 0.0, 0.0]    # Black
    lerp_factor = energia / 100.0    # Assuming energia ranges from 0 to 100

    # Interpolate color
    current_color = [
        initial_color[0] * (1 - lerp_factor) + final_color[0] * lerp_factor,
        initial_color[1] * (1 - lerp_factor) + final_color[1] * lerp_factor,
        initial_color[2] * (1 - lerp_factor) + final_color[2] * lerp_factor
    ]

    glPushMatrix()
    glTranslatef(fox.x, fox.y, fox.z)
    glRotatef(foxVetorRotacao + 90.0, 0.0, 1.0, 0.0)

    glScalef(0.009, 0.009, 0.009)

    # Disable lighting to ensure solid color rendering
    glDisable(GL_LIGHTING)

    # Apply calculated color
    glColor3f(current_color[0], current_color[1], current_color[2])

    # Render the player's body from the loaded OBJ model
    glBegin(GL_TRIANGLES)
    for face in obj_faces:
        for vertex_id in face:
            vertex = obj_vertices[vertex_id]
            glVertex3f(vertex[0], vertex[1], vertex[2])
    glEnd()

    glPopMatrix()

    # Re-enable lighting if needed for other objects
    glEnable(GL_LIGHTING)
```