**Universidade Federal Rural de Pernambuco**

**Disciplina:** Computação Gráfica

**Semestre:** 2018.1

**Professor:** Ícaro Cunha

**Projeto desenvolvido por:** Danilo Moraes, Fagner Barros, Ihago Santos

**Introdução**

O presente relatório se refere ao projeto desenvolvido para a disciplina de Computação Gráfica, no 1º semestre de 2018. O  projeto consiste em um labirinto 3D, no qual é possível prosseguir a partir de uma entrada, usando as setas do teclado, até alcançar a saída.

Todos os arquivos do projeto podem ser visualizados no repositório do gitHub (<https://github.com/FagnerPulca/projetoCG>).

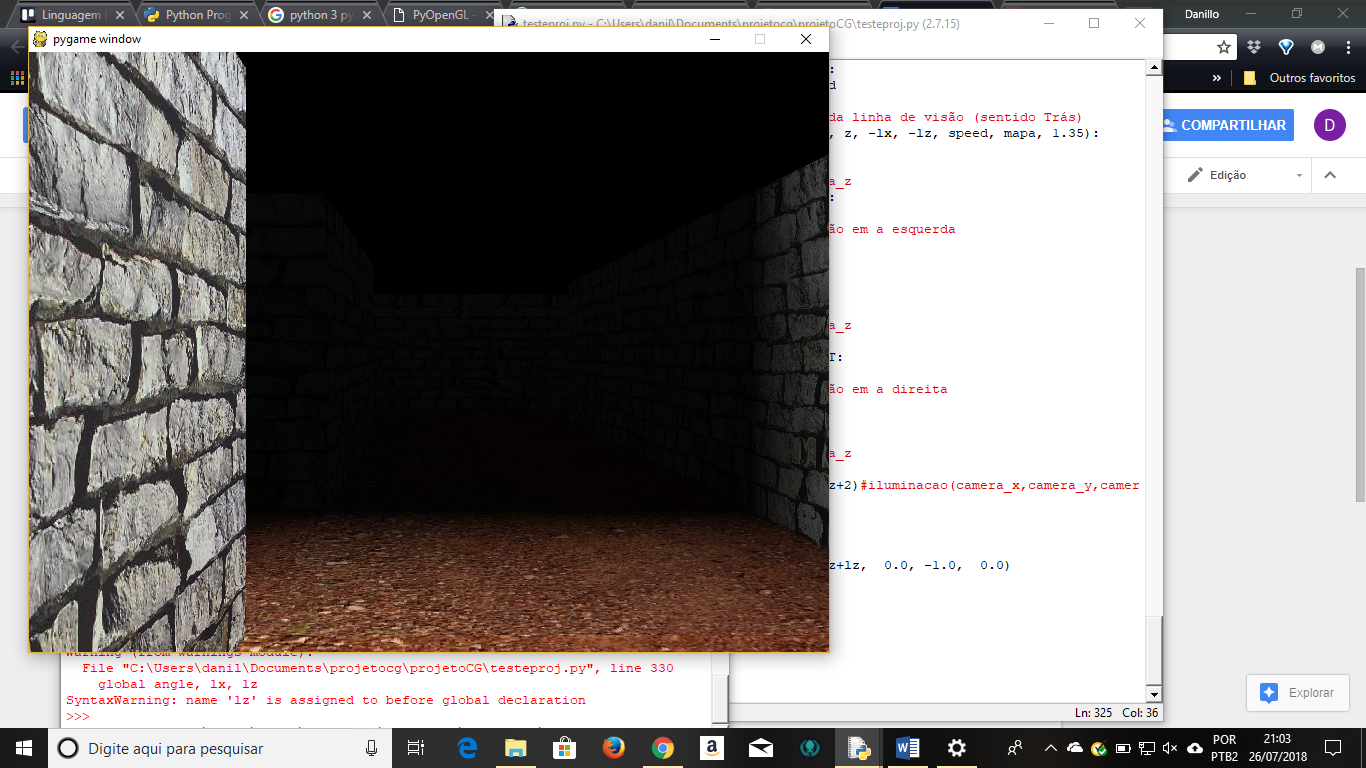
**Jogabilidade**

Com as setas do teclado o jogador movimenta a câmera para frente, para trás e girando para os lados. As setas foram configuradas da seguinte forma:

* Seta para cima: movimenta a câmera para frente;
* Seta para baixo: movimenta a câmera para trás;
* Seta para esquerda: gira a câmera no sentido anti-horário;
* Seta para direita: gira a câmera no sentido horário.

**Do Desenvolvimento**

O labirinto foi desenvolvido com a linguagem Python 2.7, em conjunto com as bibliotecas gráficas Pygame e PyOpenGL. A biblioteca Pygame foi usada para exibição dos elementos em uma tela 800x600 pixels e para a captura dos eventos de key down e carregamento das imagens de textura que são utilizadas pelo PyOpenGL, enquanto a biblioteca PyOpenGL foi usada para a criação dos elementos do labirinto (as paredes, o chão, a iluminação, a câmera e as texturas).



**Figura 1: O labirinto, ressaltando a diferença de iluminação na área próxima da câmera da área mais distante**

Com a linguagem Python foram definidos os cálculos que regeram a movimentação da câmera para que a mesma se concretizasse com o PyOpenGL, assim como a posição inicial de cada parte das paredes (representada por cubos), do chão do labirinto e do cálculo da colisão.

**Dos elementos do labirinto**

* Textura

A textura é aplicada no chão e nas paredes através da função loadTexture, que recebe como parâmetro o nome do arquivo que contém a textura que será usada.

|  |
| --- |
| **Função loadTexture** |
| def loadTexture(textura):         textureSurface = pygame.image.load(textura) #carrega imagem da textura         textureData = pygame.image.tostring(textureSurface,"RGBA",1)         width = textureSurface.get\_width()         height = textureSurface.get\_height()         glEnable(GL\_TEXTURE\_2D)#habilita textura 2D         texid = glGenTextures(1)#ID da textura         glBindTexture(GL\_TEXTURE\_2D,texid)         glTexImage2D(GL\_TEXTURE\_2D,0,GL\_RGB,width,height,0,GL\_RGBA,GL\_UNSIGNED\_BYTE,textureData)         glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT)         glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT)         glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST)         glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST)         return texid |

* Paredes

As paredes são desenhadas pela classe Cube, onde cada objeto (cubo) é instanciado com uma posição (x,y,z) como parâmetro, e seus vértices são definidos com base na posição passada. O ponto central de cada cubo também é calculado baseado no ponto médio da diagonal entre dois vértices opostos (que fazem parte de uma das diagonal que intersecta o ponto central do cubo) para auxiliar na identificação de colisões.

|  |
| --- |
| **Classe Cube** |
| class Cube(object):     def \_\_init\_\_(self, posicao):         self.posicao = posicao         self.vertices = (               (-1+self.posicao[0],-1+self.posicao[1],1+self.posicao[2]),             (1+self.posicao[0],-1+self.posicao[1],1+self.posicao[2]),             (1+self.posicao[0],1+self.posicao[1],1+self.posicao[2]),             (-1+self.posicao[0],1+self.posicao[1],1+self.posicao[2]),             (1+self.posicao[0],-1+self.posicao[1],-1+self.posicao[2]),             (-1+self.posicao[0],-1+self.posicao[1],-1+self.posicao[2]),             (-1+self.posicao[0],1+self.posicao[1],-1+self.posicao[2]),             (1+self.posicao[0],1+self.posicao[1],-1+self.posicao[2])               )         self.centro\_massa = ((self.vertices[0][0]+self.vertices[len(self.vertices)-1][0])/2,                         (self.vertices[0][1]+self.vertices[len(self.vertices)-1][1])/2,                         (self.vertices[0][2]+self.vertices[len(self.vertices)-1][2])/2)       def desenhar(self):             self.edges = (             #tras             (4,5),             (6,7),             #frente             (0,1),             (2,3),             #lado direito             (1,4),             (7,2),               #lado esquerdo             (5,0),             (3,6),             #cima             (3,2),             (7,6),             #baixo             (5,4),             (1,0)               )         texture\_vertices = (             (0,0),             (1,0),             (1,1),             (0,1)             )           i = 0             glBegin(GL\_QUADS)         for edge in self.edges:             for vertex in edge:                 glTexCoord2f(texture\_vertices[i][0], texture\_vertices[i][1])                 glVertex3fv(self.vertices[vertex])                 i += 1                 if (i > 3):                     i = 0         glEnd() |

* Chão

Cada parte do chão do labirinto tem formato quadrado e assim como os cubos que formam as paredes, o chão também recebe uma posição (x,y,z) como parâmetro e seus vértices são criados com base nos valores do parâmetro posição.

|  |
| --- |
| **Classe Ground** |
| class Ground(object):     def \_\_init\_\_(self, posicao):         self.posicao = posicao       def desenhar(self):         chao = (             (-1+self.posicao[0],1+self.posicao[1],1+self.posicao[2]),             (1+self.posicao[0],1+self.posicao[1],1+self.posicao[2]),             (1+self.posicao[0],1+self.posicao[1],-1+self.posicao[2]),             (-1+self.posicao[0],1+self.posicao[1],-1+self.posicao[2]),         )         texture\_vertices = (             (0,0),             (1,0),             (1,1),             (0,1)             )         i = 0         glBegin(GL\_QUADS)         for vertice in chao:             glTexCoord2f(texture\_vertices[i][0], texture\_vertices[i][1])             glVertex3fv(vertice)             i += 1         glEnd() |

* Colisão

A função verificarColisao trata a câmera como se fosse um cubo e trata o ponto central da câmera como ponto central desse cubo, com base nisso ele verifica a distância entre o ponto central da câmera com o ponto central de todos os cubos, caso a distância seja menor que tam\_lado\_cubo, a função indica que houve colisão.

|  |
| --- |
| **Função verificarColisao** |
| def verificarColisao(pos\_x,pos\_z, vet\_x ,vet\_z, speed, mapa, tam\_lado\_cubo):     nova\_x = (pos\_x + vet\_x\*speed )     nova\_z = (pos\_z + vet\_z\*speed )     #print len(mapa.cubos)     colisao = True     for cubo in mapa.cubos:         colisao = True         x, y, z = cubo.centro\_massa         d = math.sqrt((x-nova\_x)\*\*2 + (z-nova\_z)\*\*2)         #print "dist", d         #print "x", x, "---", "z",z         #if(abs(x-nova\_x) > 2 and abs(z - nova\_z) > 2 or (pos\_x == 0.0 and pos\_z == -2.0)):         if(d >= tam\_lado\_cubo):             colisao = False         else:             return colisao       return colisao |

* Mapa

A classe Map gera um conjunto de de cubos de pedaços do chão baseado na matriz mapa, que define no labirinto o que é parede (mapa[x][y] = 1) e o que é chão (mapa[x][y] = 0), e coloca a textura em cada um desses elementos com a função loadTexture.

|  |
| --- |
| **Classe Map** |
| class Map():     def \_\_init\_\_(self):         self.mapa =[[0,0,1,1,1,1,1,1,1,1,1,1,1,1,1],                     [1,0,1,0,1,1,0,0,1,0,0,0,0,0,1],                     [1,0,0,0,0,0,0,0,1,0,1,1,1,1,1],                     [1,1,1,1,1,1,1,0,1,0,1,0,0,0,1],                     [1,1,0,0,0,0,0,0,0,0,0,0,1,0,1],                     [1,0,0,1,1,1,1,1,1,1,1,1,1,0,1],                     [1,0,1,1,1,1,0,0,1,0,0,0,0,0,1],                     [1,0,1,1,1,1,1,0,1,1,1,0,1,1,1],                     [1,0,0,1,1,1,1,0,1,0,0,0,0,0,1],                     [1,0,0,1,1,1,0,0,1,0,1,1,1,0,1],                     [1,1,0,1,1,1,0,0,1,0,1,0,1,0,1],                     [1,1,0,0,1,1,0,0,1,0,1,0,0,0,1],                     [1,1,0,0,0,0,0,0,1,0,1,1,1,1,1],                     [1,1,0,0,0,0,0,0,1,0,0,0,0,0,1],                     [1,1,1,1,1,1,1,1,1,1,1,1,1,0,1]]         self.cubos = []         self.ground = []           for i in range(len(self.mapa)):             for j in range(len(self.mapa[i])):                 if (self.mapa[i][j] == 1):                     cubo = Cube((i\*2,0,j\*2))                     self.cubos.append(cubo)                 if (self.mapa[i][j] == 0):                     ground = Ground((i\*2,0,j\*2))                     self.ground.append(ground)     def desenhar(self):         #desenha as paredes com a textura         tex0 = loadTexture('textura\_parede.jpeg')         glEnable(GL\_TEXTURE\_2D)         glBindTexture(GL\_TEXTURE\_2D,tex0)         for cubo in self.cubos:             cubo.desenhar()         glDisable(GL\_TEXTURE\_2D)         #desenha o chão com a textura         tex1 = loadTexture('textura\_chao.jpg')         glEnable(GL\_TEXTURE\_2D)         glBindTexture(GL\_TEXTURE\_2D,tex1)         for ground in self.ground:             ground.desenhar()         glDisable(GL\_TEXTURE\_2D) |

* Iluminação

A função iluminação é responsável por gerar a luz que seguirá a câmera simulando uma lanterna, como é utilizada no jogo Amnesia, por exemplo. A chamada glShadeModel(GL\_SMOOTH) informa que será utilizado um sombreamento suave. Após isso, é ativado o teste de profundidade que atualiza o buffer de profundidade. Utilizamos dois tipos de efeitos de luz (especular e difusa).

|  |
| --- |
| **Função iluminacao** |
| def iluminacao(camera\_x,camera\_y,camera\_z):          glShadeModel(GL\_SMOOTH)     glEnable(GL\_DEPTH\_TEST)     glEnable(GL\_LIGHTING)          lightZeroPosition = [camera\_x,camera\_y,camera\_z, 1.]      lightZeroColor = [0.5, 0.5, 0.5, 1]     lightEspecular = [2., 4., 10., 1.]          glLightfv(GL\_LIGHT0, GL\_POSITION, lightZeroPosition)     glLightfv(GL\_LIGHT0, GL\_DIFFUSE, lightZeroColor)     glLightfv(GL\_LIGHT0, GL\_SPECULAR, lightEspecular)     glLightf(GL\_LIGHT0, GL\_CONSTANT\_ATTENUATION, 0.1)     glLightf(GL\_LIGHT0, GL\_LINEAR\_ATTENUATION, 0.05)          glEnable(GL\_LIGHT0) |

* Main

Na função main é definida a resolução da tela (800px por 600px) com projeção perspectiva e as variáveis usadas para translação e rotação da câmera (*speed*, *angle*, *x*, *z*, *lx* e *lz*) através da função gluLookAt (optamos por fazer a translação e rotação da câmera através dessa função porque nós precisávamos movimentar somente a câmera, o que não seria possível se fosse utilizado as funções de translação e rotação do PyOpenGl).

Cada vez que um comando de uma das setas do teclado é recebido, as variáveis são atualizadas de acordo com o movimento, se for a seta para cima ou para baixo (para frente e para trás), as variáveis *x* e *z* (posição atual da câmera) são atualizadas com os valores dos cálculos entre as variáveis *lx* e *lz* (direção da câmera) com *speed* (velocidade do movimento). Se a seta for para esquerda ou para direita, a variável *angle* é modificada de acordo com a direção da rotação, e os valores das variáveis *lx* e *lz* são calculados a partir de seno e cosseno de *angle*.

A colisão é checada sempre que se pressiona a seta para cima ou para baixo, se a câmera estiver em colisão o movimento nessa direção é interrompido até a câmera sair desse estado, se movendo em uma direção diferente da que resultou em colisão.

Antes de aplicar a movimentação da câmera de fato, através da gluLookAt, nós reiniciamos as transformações através da função glLoadIdentity (que é responsável por carregar a matriz identidade).

|  |
| --- |
| **Main** |
| def main():     pygame.init()     display = (800,600)     pygame.display.set\_mode(display, DOUBLEBUF|OPENGL)       glMatrixMode(GL\_PROJECTION)     glLoadIdentity()     gluPerspective(45, (display[0]/display[1]), 0.1, 50)     #Carrega os audios     trilha\_sonora = pygame.mixer.Sound('trilha\_sonora.ogg')       glMatrixMode(GL\_MODELVIEW)     glLoadIdentity()     gluLookAt(0.0,0.0,-2.0,               0.0,0.0,4.0,               0.0,-1.0,0.0)     global angle, lx, lz, x, z, speed     #Variáveis global usadas para movimentos da camera     angle = 0.0     lx = 0.0     lz = 1.0     x = 0.0     z = -2.0       #Fração da movimentação na direção da linha de visão     speed = 0.2       #Habilita o z-Buffer     glEnable(GL\_DEPTH\_TEST)     #carrega Textura  ##    loadTexture()     #toca trilha sonora     trilha\_sonora.play()  ##    glutInitDisplayMode(GLUT\_DEPTH|GLUT\_DOUBLE|GLUT\_RGBA)     #iluminacao()       while True:         for event in pygame.event.get():             if event.type == pygame.QUIT:                 sys.exit(pygame.quit())           glClear(GL\_COLOR\_BUFFER\_BIT|GL\_DEPTH\_BUFFER\_BIT)         glPushMatrix()         mapa = Map()         mapa.desenhar()           glPopMatrix()             if event.type == pygame.KEYDOWN:             f = glGetDoublev(GL\_MODELVIEW\_MATRIX)             camera\_x = f[3][0]             camera\_y = f[3][1]             camera\_z = f[3][2]               #print camera\_x,",",camera\_z               if event.key == pygame.K\_UP:                 global lx, lz, x, z, speed, play\_x, play\_z                 #movimentação na direção da linha de visão (sentido Frente)                 #print mapa.mapa[0][3]                 if not verificarColisao(x, z, lx, lz, speed, mapa, 1.35):                     x += lx \* speed                     z += lz \* speed                   #print camera\_x,",",camera\_z                   #print play\_x,",",play\_z               if event.key == pygame.K\_DOWN:                 global lx, lz, x, z, speed                   #movimentação na direção da linha de visão (sentido Trás)                 if not verificarColisao(x, z, -lx, -lz, speed, mapa, 1.35):                     x -= lx \* speed                     z -= lz \* speed                 #print camera\_x,",",camera\_z             if event.key == pygame.K\_LEFT:                 global lx, lz, angle                 #Rotaciona a linha de visão em a esquerda                   angle -= 0.1                 lx = math.sin(angle)                 lz = math.cos(angle)                   #print camera\_x,",",camera\_z               if event.key == pygame.K\_RIGHT:                 global angle, lx, lz                 #Rotaciona a linha de visão em a direita                 angle += 0.1                 lx = math.sin(angle)                 lz = math.cos(angle)                 #print camera\_x,",",camera\_z             iluminacao(x+lx,camera\_y,z+ lz+2)#iluminacao(camera\_x,camera\_y,camera\_z + 10)           # Reset transformations         glLoadIdentity()  # Set the camera         gluLookAt(x, 0.0, z,  x+lx, 0.0, z+lz, 0.0, -1.0,  0.0)           pygame.display.flip()         pygame.time.wait(10) |