Nomes: Gustavo Willian Martins da Silva e Vítor Aguirre Caús

Computação Gráfica

Trabalho 2 - Robô Articulado Atirador

Neste trabalho, buscou-se implementar um cenário 3d onde o usuário é capaz de controlar um veículo atirador articulado, capaz de disparar projéteis para destruir um paredão posicionado no centro do espaço onde ele se localiza. As articulações desenvolvidas permitem a rotação do tanque em torno do eixo Y e a subida/descida de seu canhão em relação ao eixo Z, de modo a posicionar a trajetória do disparo conforme desejado, também sendo possível controlar a força do tiro do canhão, de modo a alcançar alvos mais distantes. Abaixo, são descritas as implementações realizadas em relação ao código

- **Desenho do Paredão, Piso e Texturas:**

Inicialmente, é desenhado um piso de tamanho 25X50 no ponto (-20,-1,-10), a partir do qual, em seguida, é desenhada a parede de tamanho 25X15, que sofre transformações de translação e de rotação no eixo Z para que fique posicionada no centro do piso. No momento de criação desses dois planos, são aplicadas suas texturas (grama e tijolos), mediante auxílio de variáveis que guardam os valores X e Y de cada imagem para mapear corretamente um trecho das texturas a cada polígono do piso e da parede.

```
void DesenhaPiso()
{
    imgCoordX1 = 0.0;
    imgCoordX2 = 0.0;
    imgCoordY1 = 0.0;
    imgCoordY2 = 0.0;
    srand(100); // usa uma semente fixa para gerar sempre as mesma cores no piso
    glPushMatrix();
    glTranslated(CantoEsquerdo.x, CantoEsquerdo.y, CantoEsquerdo.z);
    for(int x=0; x<50;x++)
    {
        imgCoordY1 = x * 0.020;
        imgCoordY2 = imgCoordY1 + 0.020;
        glPushMatrix();
        for(int z=0; z<25;z++)
        {
            imgCoordX1 = z * 0.040;
            imgCoordX2 = imgCoordX1 + 0.040;
            DesenhaLadrilho(MediumGoldenrod, rand()%40);
            glTranslated(0, 0, 1);
        }
        glPopMatrix();
        glTranslated(1, 0, 0);
    }
    glPopMatrix();
}

void DesenhaParede()
{
    imgCoordX1 = 0.0;
    imgCoordX2 = 0.0;
    imgCoordY1 = 0.0;
    imgCoordY2 = 0.0;
    glPushMatrix();
        glTranslatef(5,0,0);
        glRotatef(90,0,0,1);
        glTranslatef(19,0,0);
        srand(100); // usa uma semente fixa para gerar sempre as mesma cores no piso
        glTranslated(CantoEsquerdo.x, CantoEsquerdo.y, CantoEsquerdo.z);
        for(int y=0; y<15;y++){
            imgCoordY1 = y * 0.060;
            imgCoordY2 = imgCoordY1 + 0.060;
            glPushMatrix();
            for(int z=0; z<25;z++){
                imgCoordX1 = z * 0.040;
                imgCoordX2 = imgCoordX1 + 0.040;
                if(quadradosParede[z][y]){
                    DesenhaLadrilho(MediumGoldenrod, rand()%40);
                }
                glTranslated(0, 0, 1);
            }
            glPopMatrix();
            glTranslated(1, 0, 0);
        }
    glPopMatrix();
}
```

```
void DesenhaLadrilho(int corBorda, int corDentro)
{
//    defineCor(corDentro);// desenha QUAD preenchido
    glColor3f(1,1,1);
    glBegin ( GL_QUADS );
        glNormal3f(0,1,0);
        glTexCoord2f(imgCoordX1, imgCoordY1);
        glVertex3f(0.0f,  0.0f, 0.0f);
        glTexCoord2f(imgCoordX2, imgCoordY1);
        glVertex3f(0.0f,  0.0f,  1.0f);
        glTexCoord2f(imgCoordX2, imgCoordY2);
        glVertex3f( 1.0f,  0.0f,  1.0f);
        glTexCoord2f(imgCoordX1, imgCoordY2);
        glVertex3f( 1.0f,  0.0f, 0.0f);
    glEnd();

    //defineCor(corBorda);
//    glColor3f(0,1,1);

    glBegin ( GL_LINE_STRIP );
        glNormal3f(0,1,0);
        glVertex3f(0.0f,  0.0f, 0.0f);
        glVertex3f(0.0f,  0.0f,  1.0f);
        glVertex3f( 1.0f,  0.0f,  1.0f);
        glVertex3f( 1.0f,  0.0f, 0.0f);
    glEnd();


}
```

- **Modelagem de veículo, articulações e movimento:**

O veículo atirador é modelado com dois cubos: o primeiro é transformado com uma escala de (3,1,2), enquanto o segundo é transladado para o ponto central do cubo da base (com incremento de 0.75 em Y) e então é desenhado com escala (2.0, 0.5, 0.5).

Com as teclas “a” e “d” é realizado, respectivamente, o incremento e o decremente do ângulo do veículo em 5°, que é utilizado, durante seu desenho, para rotacionar a base do tanque e o canhão em torno do eixo Y. Já com as teclas “c” e “C”, o ângulo do canhão em torno do eixo Z é, respectivamente, aumentado e reduzido em 1°. A movimentação do canhão só é realizada quando o tanque tem ângulo de rotação igual a 0°, ou seja, quando ele está direcionado de frente para o paredão.

Para o movimento do tanque, as teclas “w” e “s” foram definidas para movê-lo para frente e para trás, de modo a alterar as posições da base do tanque e de seu canhão, além dos pontos do observador e alvo da visão de primeira pessoa.

```
void desenhaTanque()
{

    glPushMatrix();

        glDisable(GL_TEXTURE_2D);
        defineCor(DarkOliveGreen);
        RotacionaAoRedorDeUmPontoY(anguloVeiculo,Tanque);
        glTranslatef(PosicaoVeiculo.x, PosicaoVeiculo.y, PosicaoVeiculo.z);
        glRotatef(anguloVeiculo, 0, 1, 0);
        glScalef(3,1,2);
        glutSolidCube(1);
    glPopMatrix();

    glPushMatrix();
        defineCor(ForestGreen);
        RotacionaAoRedorDeUmPontoY(anguloVeiculo,Tanque);
        glTranslatef(PosicaoCanhaoEstrutura.x, PosicaoCanhaoEstrutura.y, PosicaoCanhaoEstrutura.z);

        glRotatef(anguloVeiculo, 0, 1, 0);
        glRotatef(anguloCanhao, 0, 0, 1);
        glTranslatef(0.75,0,0);
        glScalef(2.0, 0.5, 0.5);
        glutSolidCube(1);
    glPopMatrix();
}
```

```
case 'w':

        Tanque = Tanque + (VetTanqueFrente)*0.2;
        olharFrente = olharFrente + (VetTanqueFrente)*0.2;
        PosicaoVeiculo = PosicaoVeiculo + (VetTanqueFrente)*0.2;
        PosicaoCanhaoEstrutura = PosicaoCanhaoEstrutura + (VetTanqueFrente)*0.2;
        PosicaoCanhao = PosicaoCanhao + (VetTanqueFrente)*0.2;
        break;
case 'a':
        aux = VetTanqueFrente;
        aux.rotacionaY(anguloVisao*2);
        olharFrente = Tanque + aux;
        anguloVeiculo += 5;
        if(anguloVeiculo >= 180){anguloVeiculo = 0;}
        configDisparo = true;
        break;
case 's':
        Tanque = Tanque - (VetTanqueFrente)*0.2;
        olharFrente = olharFrente - (VetTanqueFrente)*0.2;
        PosicaoVeiculo = PosicaoVeiculo - (VetTanqueFrente)*0.2;
        PosicaoCanhaoEstrutura = PosicaoCanhaoEstrutura - (VetTanqueFrente)*0.2;
        PosicaoCanhao = PosicaoCanhao - (VetTanqueFrente)*0.2;
        break;
case 'd':
        aux = VetTanqueFrente;
        aux.rotacionaY(-anguloVisao*2);
        olharFrente = Tanque + aux;
        anguloVeiculo -= 5;
        if(anguloVeiculo <= -180){anguloVeiculo = 0;}
        configDisparo = true;
        break;

         case 'c': // sobe canhao
             if(anguloVeiculo == 0.0 && anguloCanhao < 90.0){
                 anguloCanhao += 1.0;
                 configDisparo = true;
             }
             break;
         case 'C':// baixa canhao
             if(anguloVeiculo == 0.0 && anguloCanhao > 0.0){
                 anguloCanhao -= 1.0;
                 configDisparo = true;
             }
             break;
```

- **Lançamento do projétil:**

Com base na formulação sugerida para cálculo dos pontos da trajetória, foram definidos dois pontos, “alcanceAux” e “alcanceFinal”, para tratar o deslocamento do tiro do canhão. Esses dois pontos também são rotacionados conforme o ângulo do veículo, a fim de alinhar corretamente a mira do canhão com sua estrutura quando o tanque gira para um dos lados. Dados os 2 pontos mencionados e a posição do canhão, é calculada uma curva Bèzier para definir o trajeto do objeto disparado (ativado pela tecla “m”), cuja força é incrementada e decrementada com as teclas “f” e “F”, sendo limitada de 1 a 50. Só é possível disparar um único projétil por vez, até que este atinja a parede ou saia dos limites definidos para seu cálculo e renderização.

```
if(configDisparo)
{
    direcaoCanhao = Ponto (1,0,0);
    direcaoCanhao.rotacionaZ(anguloCanhao);
    direcaoCanhao.rotacionaY(anguloVeiculo);
    configDisparo = false;
}

    alcanceAux = PosicaoCanhao + direcaoCanhao * forca;
    distancia = 2 * forca * cos(anguloCanhao * 3.14/180);
    alcanceFinal = PosicaoCanhao + Ponto(distancia, 0, 0);

    Ponto vetAuxCanhao = alcanceAux - PosicaoCanhao;
    vetAuxCanhao.rotacionaY(anguloVeiculo);
    alcanceAux = PosicaoCanhao + vetAuxCanhao;

    Ponto vetDistCanhao = alcanceFinal - PosicaoCanhao;
    vetDistCanhao.rotacionaY(anguloVeiculo*2);
    alcanceFinal = PosicaoCanhao + vetDistCanhao;
```

```
void trajetoria(float t)
{
    float aux = 1-t;
    Ponto parteP0 = PosicaoCanhao * pow(aux,2);
    Ponto parteP1 = alcanceAux * (2*aux*t);
    Ponto parteP2 = alcanceFinal * pow(t,2);
    projetil = parteP0 + parteP1 + parteP2;

    glPushMatrix();
        glColor3f(0.5,0.5,0.5);
        glTranslatef(projetil.x,projetil.y,projetil.z);
        glutSolidSphere(0.3,15,15);
    glPopMatrix();

    if(projetil.y < -1 || projetil.x > 50 || projetil.x < -30 || projetil.z > 30 || projetil.z < -40)
    {
        disparo = false;
        valorT = 0.0;
    }
    if(colisaoParede()){
        disparo = false;
        valorT = 0.0;
    }
}
```

- **Detecção de colisão com a parede e reconfiguração:**

Para o mapeamento dos polígonos da parede, foi definida uma matriz 25 X 15 com valores booleanos inicializada como ***true*** em todas as posições, a fim de permitir o desenho dos polígonos do paredão apenas se suas respectivas posições na matriz indicarem que eles existem. A partir disso, durante o cálculo da trajetória do projétil, é verificado se ele alcançou ou atravessou o paredão (caso sua coordenada X seja maior ou igual à do paredão) e, em caso afirmativo, a posição do projétil é mapeada para sua respectiva posição na matriz de booleanos da parede, de modo a adquirir o valor que define ou não se há um polígono no ponto atingido.

Caso se confirme que o projétil acertou um ponto onde há parte do paredão, a posição desse polígono é convertida para ***false*** na matriz, e o mesmo ocorre com os 8 quadrados ao seu redor, se existirem. Dessa forma, os polígonos que passam a ser definidos como inexistentes deixam de ser renderizados. A colisão e destruição do paredão garante 10 pontos ao usuário.

```
bool colisaoParede()
{
    if(projetil.x >= 6){
        if((projetil.z > -10 && projetil.z < 15)&& projetil.y < 15 && projetil.y >= 0)
        {
            int yAux = (int)projetil.y + 1;
            int zAux = (int)projetil.z + 10;
            if(quadradosParede[zAux][yAux])
            {
                quadradosParede[zAux][yAux] = false;
                if(yAux-1 >= 0){quadradosParede[zAux][yAux-1] = false;}
                if(zAux-1 >= 0){quadradosParede[zAux-1][yAux] = false;}
                if(yAux+1 < 15){quadradosParede[zAux][yAux+1] = false;}
                if(zAux+1 < 25){quadradosParede[zAux+1][yAux] = false;}
                if(yAux-1 >= 0 && zAux-1 >= 0){quadradosParede[zAux-1][yAux-1] = false;}
                if(yAux+1 < 15 && zAux+1 < 25){quadradosParede[zAux+1][yAux+1] = false;}
                if(yAux-1 >= 0 && zAux+1 < 25){quadradosParede[zAux+1][yAux-1] = false;}
                if(yAux+1 < 15 && zAux-1 >= 0){quadradosParede[zAux-1][yAux+1] = false;}
                pontuacao += 5;
                cout << "Pontuacao atual = " << pontuacao << endl;
                return true;
            }
        }
    }
    return false;
}
```

- **Exibição de objetos TRI:**

Com base nos códigos de exemplo para leitura, armazenamento e exibição de objetos TRI, modificou-se parte da leitura para incluir o cálculo dos vetores normais de cada face triangular, além de sua adição na função responsável por exibir os objetos. Entretanto, é possível observar que a qualidade da renderização depende da posição do observador em relação a cada objeto (identificado no vídeo de demonstração).

```
for (int i=1;i<nFaces;i++)
{
    // Le os vertices
    arq >> x >> y >> z; // Vertice 1
    faces[i].P1.set(x,y,z);
    arq >> x >> y >> z; // Vertice 2
    faces[i].P2.set(x,y,z);
    arq >> x >> y >> z; // Vertice 3
    faces[i].P3.set(x,y,z);

    Ponto norma1, norma2, normaResult;
    norma1 = faces[i].P3 - faces[i].P2;
    norma2 = faces[i].P1 - faces[i].P2;
    norma1.versor();
    norma2.versor();
    ProdVetorial(norma1, norma2, normaResult);
    faces[i].Normal.set(normaResult.x,normaResult.y,normaResult.z);

    // ler o RGB da face
    arq >> std::hex >> rgb;
    faces[i].rgb = rgb;
}
arq.close();
```

- **Teclas auxiliares:**
    - t = altera a visão entre primeira/terceira pessoa;
    - u/U = na terceira pessoa, rotacionam o alvo em torno do eixo Z, respectivamente, para cima e para baixo.

- **Critérios não alcançados durante a execução do trabalho:**
    - Colisão do veículo com as bordas do piso e com trechos desenhados da parede;

- Colisão do projétil e do veículo com objetos TRI (e consequentemente sua destruição, no caso do projétil);
- Parte do mapeamento de colisão do paredão exclui trechos errados ou não exclui nada mesmo com o projétil atingindo-o.

Link para o vídeo:

https://www.youtube.com/watch?v=dKGcaakYSZs