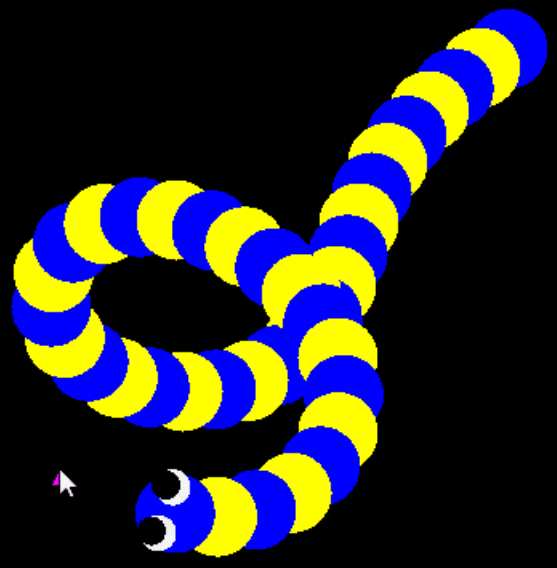


Cobrinha Slither.io: Passo-a-passo para o trabalho

por Rossana B Queiroz

Código Base

- Triângulo que segue o mouse, feito na aula do dia 05/10
  - https://github.com/fellowsheep/FCG2024-2/tree/main/HelloTriangle%20-%20Cobrinha

Recapitulando

- Estrutura **Geometry**
  - Estrutura para armazenar informações das geometrias da cena
  - Campos da struct (como se fossem atributos de uma classe, mas todos públicos):
    - VAO – identificador do Vertex Array Object
    - Posição – coordenadas de posição do objeto
    - Ângulo – rotação no eixo z, em pi radianos
    - Dimensões – largura e altura do objeto (para escalar)
    - Cor – cor para o desenho
    - nVertices – número de vértices contidos no buffer

Recapitulando

- Estrutura **Geometry**

```
// Estrutura para armazenar informações sobre as geometrias da cena
struct Geometry {
    GLuint VAO;         // Vertex Array Geometry
    vec3 position;      // Posição do objeto
    float angle;        // Ângulo de rotação
    vec3 dimensions;    // Escala do objeto (largura, altura)
    vec3 color;         // Cor do objeto
    int nVertices;      // Número de vértices a desenhar
};
```

Função drawGeometry

- A função **drawGeometry** desenha o objeto na tela

```
void drawGeometry(GLuint shaderID, GLuint VAO, int nVertices, vec3
position, vec3 dimensions, float angle, vec3 color, GLuint drawingMode
= GL_TRIANGLES, vec3 axis = vec3(0.0, 0.0, 1.0));
```

- Ela recebe como parâmetros:
  1. **shaderID**: identificador do shader, para enviarmos as informações da geometria
     - assume-se que já foi chamado o glUseProgram(shaderID)

Função drawGeometry

- Ela recebe como parâmetros (continuação):
  2. Campos **VAO**, **nVertices**, **posição**, **dimensões**, **ângulo** e **cor** da geometria a ser desenhada
     - **SUGESTÃO**: poderia se criar uma sobrecarga dessa função passando uma variável do tipo Geometry ao invés de todos os campos separados (foi feito assim para poder passar infos que não estejam em um Geometry também – compatibilidade com códigos anteriores)

Função drawGeometry

- Ela recebe como parâmetros (continuação):
  3. **drawingMode** – o tipo de primitiva de desenho desejada (por padrão GL_TRIANGLES)
     - No caso dos círculos, chamaremos com GL_TRIANGLE_FAN
  4. **axis** – como nossa cena é 2D, provavelmente vamos rotacionar sempre no eixo z, mas é possível alterar se precisar

Atualização da posição e ângulo do objeto em relação ao Mouse

- Cálculo da direção e nova posição – Matemática
  - O vetor direção $\vec{d}$ é o vetor que aponta da posição atual do triângulo até o ponto de destino (neste caso, a posição do mouse). Esse vetor é normalizado para ter um comprimento de 1, ou seja, apenas representando a direção.

- Fórmulas: $\vec{d} = \dfrac{p_{\text{mouse}} - p_{\text{triângulo}}}{\left| p_{\text{mouse}} - p_{\text{triângulo}} \right|}$

$$p_{\text{novo}} = p_{\text{atual}} + v \cdot d$$

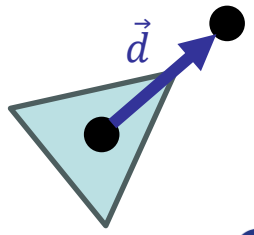
Onde:

- $\vec{d}$: vetor direção
- p_{mouse} : posição do mouse
- $p_{\text{triângulo}}$: posição do triângulo (ou outra geometria)
- v: : velocidade (rapidez) do deslocamento
- p_{novo} :nova posição
- p_{atual}: posição atual da geometria

Atualização da posição e ângulo do objeto em relação ao Mouse

- Cálculo da direção – Código
  - No código, usamos a função normalize da GLM para normalizar o vetor direção
  - Depois disso, somamos essa direção (escalada por alguma velocidade, como por exemplo 0.5) à posição para obtermos a nova posição do objeto

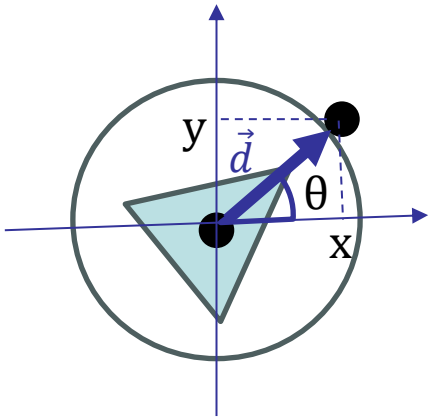
```
dir = normalize(vec3(mousePos, 0.0) - position);
position = position + 0.5f * dir;
```



Atualização da posição e ângulo do objeto em relação ao Mouse

- Cálculo do Ângulo: Matemática
  - Para calcular o ângulo de rotação do triângulo, usamos a função inversa da tangente (arco tangente) para obter o ângulo a partir das componentes $\mathbf{x}$ e $\mathbf{y}$ do vetor direção $\vec{d}$. Isso nos dá o ângulo θ que o triângulo deve rotacionar para alinhar-se com o vetor direção.
    - Fórmula:

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$



Atualização da posição e ângulo do objeto em relação ao Mouse

- Cálculo do Ângulo: Código
  - A função `atan2(y,x)` é usada em vez de `atan(y/x)` porque considera os sinais de x e y, retornando o ângulo no intervalo correto $[-\pi,+\pi]$, o que evita ambiguidades sobre o quadrante em que o vetor se encontra.

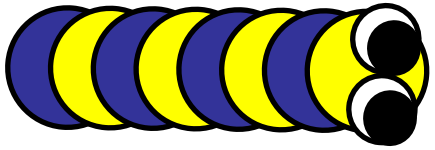
```
float lookangle = atan2(dir.y, dir.x);
```

Cobrinha: estrutura de dados

- Vector de Geometry (Cabeça e Segmentos)
  - A estrutura Geometry é utilizada para armazenar a cabeça e cada um dos segmentos do corpo da cobrinha.
  - O vector cobrinha contém todos os segmentos, começando pelo elemento de índice 0 (a cabeça).
  - A cabeça e os segmentos são tratados de forma semelhante, mas com cores alternadas para distinguir os segmentos.

Cobrinha: estrutura de dados

- Vector de Geometry (Cabeça e Segmentos)
  - Código:



    - Variável global

```
// Vetor que armazena todos os segmentos da cobrinha,
// incluindo a cabeça
std::vector<Geometry> cobrinha;
```

    - Código de inicialização (pode ser no main)

```
//Criação da Cabeça
Geometry head = createSegment(0, dir);
cobrinha.push_back(head);
```

Cobrinha: estrutura de dados

- Olhos (variável da classe Geometry)
  - A geometria dos olhos foi definida como um objeto Geometry à parte, representando as escleras e as pupilas.
  - Os olhos são desenhados na mesma posição da cabeça e rotacionados de acordo com a direção da mesma.
  - O objeto eyes contém os dados para desenhar dois círculos para cada olho (esclera e pupila).

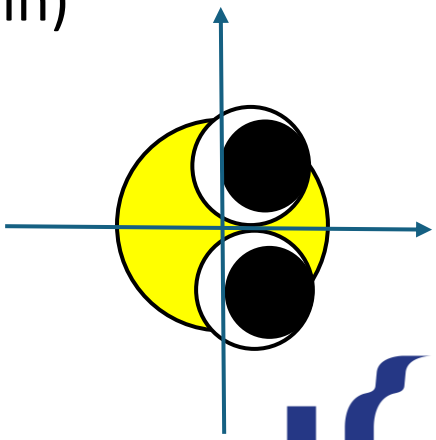
Cobrinha: estrutura de dados

- Olhos (variável da classe Geometry)
  - Código de criação:
    - Variável global:

```
// Objeto Geometry que representa os olhos da cabeça da cobrinha
Geometry eyes;
```

- Código de inicialização (pode ser no main)

```
    //Criação da geometria dos olhos
    eyes.VAO = createEyes(32, 0.25);
    eyes.nVertices = 34;
    eyes.position = vec3(400, 300, 0);
    eyes.dimensions = vec3(50, 50, 1.0);
    eyes.angle = 0.0;
    eyes.color = vec3(1.0, 1.0, 1.0);
```



Cobrinha: estrutura de dados

- Função Create Segment
  - A função createSegment cria os segmentos da cobrinha.
    - O segmento de índice 0 é a cabeça, que começa no centro da tela.
    - Os segmentos subsequentes são posicionados a uma distância mínima do segmento anterior.
  - As cores dos segmentos alternam entre azul e amarelo, dependendo do índice.

Cobrinha: estrutura de dados

- Função CreateSegment – Código (parte 1)

```
// Cria um segmento da cobrinha, retornando um objeto Geometry com a posição e cor apropriadas
// i: Índice do segmento (0 para a cabeça, índices maiores para os segmentos do corpo)
// dir: Vetor direção indicando a direção inicial do segmento
Geometry createSegment(int i, vec3 dir)
{
    std::cout << "Criando segmento " << i << std::endl;

    // Inicializa um objeto Geometry para armazenar as informações do segmento
    Geometry segment;
    segment.VAO = createCircle(32, 0.5);  // Cria a geometria do segmento como um círculo
    segment.nVertices = 34;               // Número de vértices do círculo

    // Define a posição inicial do segmento
    if (i == 0) {  // Cabeça
        segment.position = vec3(400.0, 300.0, 0.0);  // Posição inicial no centro da tela
    } else {
        // Ajusta a direção com base na posição dos segmentos anteriores para evitar sobreposição
        if (i >= 2)
            dir = normalize(cobrinha[i - 1].position - cobrinha[i - 2].position);
        // Posiciona o novo segmento com uma distância mínima do segmento anterior
        segment.position = cobrinha[i - 1].position + minDistance * dir;
    }
```

Cobrinha: estrutura de dados

- Função CreateSegment – Código (parte 2)

```
    // Define as dimensões do segmento (tamanho do círculo)
    segment.dimensions = vec3(50, 50, 1.0);
    segment.angle = 0.0;  // Ângulo inicial (sem rotação)

    // Alterna a cor do segmento entre azul e amarelo, dependendo do índice
    if (i % 2 == 0) {
        segment.color = vec3(0, 0, 1);  // Azul para segmentos de índice par
    } else {
        segment.color = vec3(1, 1, 0);  // Amarelo para segmentos de índice ímpar
    }

    return segment;
}
```

Cobrinha: estrutura de dados

- Função CreateEyes
  - A função createEyes define os vértices para as escleras e pupilas dos olhos.
  - São criados dois círculos para as escleras (um para cada olho).
  - Depois, são criados mais dois círculos menores para as pupilas.

Cobrinha: estrutura de dados

- Função CreateEyes (Código parte 1)

```
// Cria a geometria dos olhos da cabeça da cobrinha, retornando o identificador do VAO
// nPoints: Número de pontos usados para aproximar os círculos que compõem os olhos
// radius: Raio das escleras dos olhos
int createEyes(int nPoints, float radius) {
    // Vetor para armazenar os vértices dos olhos (escleras e pupilas)
    std::vector<GLfloat> vertices;

    // Ângulo inicial e incremento para cada ponto do círculo
    float angle = 0.0;
    float slice = 2 * Pi / static_cast<float>(nPoints);

    // Posições iniciais para os círculos dos olhos (escleras e pupilas)
    float xi = 0.125f;  // Posição inicial X das escleras
    float yi = 0.3f;    // Posição inicial Y das escleras
    radius = 0.225f;    // Raio das escleras
```

Cobrinha: estrutura de dados

- Função CreateEyes (Código parte 2)

```
    // Olho esquerdo (esclera)
    vertices.push_back(xi);    // Xc
    vertices.push_back(yi);    // Yc
    vertices.push_back(0.0f);  // Zc
    for (int i = 0; i < nPoints + 1; i++) {
        float x = xi + radius * cos(angle);
        float y = yi + radius * sin(angle);
        float z = 0.0f;

        vertices.push_back(x); // Coordenada X
        vertices.push_back(y); // Coordenada Y
        vertices.push_back(z); // Coordenada Z

        angle += slice; // Incrementa o ângulo para o próximo ponto
    }
```

Cobrinha: estrutura de dados

- Função CreateEyes (Código parte 3)

```
    // Olho direito (esclera)
    angle = 0.0;
    vertices.push_back(xi);    // Xc
    vertices.push_back(-yi);   // Yc
    vertices.push_back(0.0f);  // Zc
    for (int i = 0; i < nPoints + 1; i++) {
        float x = xi + radius * cos(angle);
        float y = -yi + radius * sin(angle);
        float z = 0.0f;

        vertices.push_back(x); // Coordenada X
        vertices.push_back(y); // Coordenada Y
        vertices.push_back(z); // Coordenada Z

        angle += slice;
    }
```

Cobrinha: estrutura de dados

- Função CreateEyes (Código parte 4)

```
    // Olho esquerdo (pupila)
    radius = 0.18f;   // Raio das pupilas
    xi += 0.09f;      // Ajuste de posição para as pupilas
    angle = 0.0;
    vertices.push_back(xi);    // Xc
    vertices.push_back(yi);    // Yc
    vertices.push_back(0.0f);  // Zc
    for (int i = 0; i < nPoints + 1; i++) {
        float x = xi + radius * cos(angle);
        float y = yi + radius * sin(angle);
        float z = 0.0f;

        vertices.push_back(x); // Coordenada X
        vertices.push_back(y); // Coordenada Y
        vertices.push_back(z); // Coordenada Z

        angle += slice;
    }
```

Cobrinha: estrutura de dados

- Função CreateEyes (Código parte 5)

```
// Olho direito (pupila)
angle = 0.0;
vertices.push_back(xi);    // Xc
vertices.push_back(-yi);   // Yc
vertices.push_back(0.0f);  // Zc
for (int i = 0; i < nPoints + 1; i++) {
    float x = xi + radius * cos(angle);
    float y = -yi + radius * sin(angle);
    float z = 0.0f;

    vertices.push_back(x); // Coordenada X
    vertices.push_back(y); // Coordenada Y
    vertices.push_back(z); // Coordenada Z

    angle += slice;
}
```

Cobrinha: estrutura de dados

- Função CreateEyes (Código parte 6)

```
    // Identificadores para o VBO (Vertex Buffer Object) e VAO (Vertex Array Object)
    GLuint VBO, VAO;

    // Geração do identificador do VBO e vinculação
    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // Envia os dados do vetor de vértices para a GPU
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(GLfloat), vertices.data(), GL_STATIC_DRAW);

    // Geração do identificador do VAO e vinculação
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);

    // Configuração do ponteiro de atributos para os vértices
    // layout (location = 0) no Vertex Shader, 3 componentes por vértice (x, y, z)
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
    glEnableVertexAttribArray(0);

    // Desvincula o VBO e o VAO para evitar modificações acidentais
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);

    // Retorna o identificador do VAO, que será utilizado para desenhar os olhos (escleras e pupilas)
    return VAO;
}
```

Cobrinha: atualização da cabeça

- Para atualizar a cabeça da cobrinha, basta usar a posição e orientação que tínhamos calculado para o código do triângulo que segue o mouse:

```
// Atualiza a posição e ângulo da cabeça e dos olhos
cobrinha[0].position = position;
cobrinha[0].angle = lookangle;
eyes.position = position;
eyes.angle = lookangle;
```

Cobrinha: atualização dos segmentos do corpo

- Cada segmento da cobrinha segue o anterior, mantendo uma distância mínima e máxima.
- O movimento é suavizado utilizando a função `mix` do GLM para interpolação linear.
- Interpolação linear:

$$p_{\text{novo}} = (1 - \alpha) \cdot p_{\text{atual}} + \alpha \cdot p_{\text{alvo}}$$

Onde:

- α : é o fator de interpolação, ou seja, o valor que determina a suavidade (0 a 1)
- p_{atual} : é a posição atual
- p_{alvo} : é a posição alvo (a posição do segmento anterior)
- p_{novo} : é a nova posição interpolada

Cobrinha: atualização dos segmentos do corpo

- Código

```
// Atualiza a posição dos segmentos da cobrinha para seguir a cabeça
for (int i = 1; i < cobrinha.size(); i++)
{
    vec3 dir = normalize(cobrinha[i - 1].position - cobrinha[i].position);
    float distance = length(cobrinha[i - 1].position - cobrinha[i].position);

    // Calcula a nova posição do segmento com suavidade, respeitando as distâncias mínima e máxima
    vec3 targetPosition = cobrinha[i].position;
    float dynamicSmoothFactor = smoothFactor * (distance / maxDistance);

    if (distance < minDistance)
    {
        targetPosition = cobrinha[i].position + (distance - minDistance) * dir;
    }
    else if (distance > maxDistance)
    {
        targetPosition = cobrinha[i].position + (distance - maxDistance) * dir;
    }

    // Interpolação suave para a nova posição do segmento
    cobrinha[i].position = mix(cobrinha[i].position, targetPosition, dynamicSmoothFactor);
}
```

Adicionando um novo segmento

- No trabalho, cada segmento deve ser adicionado quando houver colisão da cabeça com alguma comidinha
- Para fins de teste, recomenda-se fazer essa adição ao pressionar alguma tecla
  - Por exemplo, na função de callback de teclado, ao pressionar a tecla SPACE

```
// Adiciona um novo segmento à cobrinha quando a tecla Espaço é pressionada
if (key == GLFW_KEY_SPACE && action == GLFW_PRESS)
{
    addNew = true;
}
```

Adicionando um novo segmento

- O novo segmento é posicionado a uma distância mínima do último segmento
  - Para a direção, usa-se a direção contrária àquela que calculamos para mover a cabeça da cobrinha

```
// Adiciona novos segmentos à cobrinha quando solicitado
if (addNew)
{
    cobrinha.push_back(createSegment(cobrinha.size(), -dir));
    addNew = false;
}
```

Desenhando a Cobrinha

- Ordem de Desenho dos Segmentos:
  - A função percorre o vetor cobrinha de trás para frente (i = cobrinha.size() - 1 até i >= 0), desenhando os segmentos do corpo do último até o primeiro.
  - Isso garante que os segmentos mais recentes (a cabeça) sejam desenhados por último, ficando sempre à frente dos outros na tela.

Desenhando a Cobrinha

- Desenho dos Olhos:
  - O objeto eyes (olhos) foi criado separadamente e recebe a posição e rotação da cabeça.
  - Para desenhar os olhos, a posição e o ângulo da cabeça são passados como parâmetros:

Desenhando a Cobrinha

- Código:

```
// Desenha os segmentos da cobrinha e os olhos
for (int i = cobrinha.size() - 1; i >= 0; i--)
{
    drawGeometry(shaderID, cobrinha[i].VAO, cobrinha[i].nVertices, cobrinha[i].position,
cobrinha[i].dimensions, cobrinha[i].angle, cobrinha[i].color, GL_TRIANGLE_FAN);

    if (i == 0)
    { // cabeça
        // Desenha as escleras dos olhos
        drawGeometry(shaderID, eyes.VAO, eyes.nVertices, eyes.position, eyes.dimensions, eyes.angle,
eyes.color, GL_TRIANGLE_FAN, 0);
        drawGeometry(shaderID, eyes.VAO, eyes.nVertices, eyes.position, eyes.dimensions, eyes.angle,
eyes.color, GL_TRIANGLE_FAN, 34);

        // Desenha as pupilas dos olhos
        drawGeometry(shaderID, eyes.VAO, eyes.nVertices, eyes.position, eyes.dimensions, eyes.angle,
vec3(0.0, 0.0, 0.0), GL_TRIANGLE_FAN, 2 * 34);
        drawGeometry(shaderID, eyes.VAO, eyes.nVertices, eyes.position, eyes.dimensions, eyes.angle,
vec3(0.0, 0.0, 0.0), GL_TRIANGLE_FAN, 3 * 34);
    }
}
```

Extra! createCircle

- Repetindo o código da função createCircle (que já estava no Github)
- OBS.: Para a criação dos olhos, o que basicamente se fez foi replicar a createCircle 4x (para as escleras e pupilas dos olhos esquerdo e direito)
  - PODERIA ter sido feita a chamada do createCircle e cada porção ter sido armazenada eu um Geometry diferente
  - OPTOU-SE em criar apenas um Geometry com os 4 círculos

Extra! createCircle

- Código (Parte 1)

```
int createCircle(int nPoints, float radius, float xc, float yc) {
    // Vetor para armazenar os vértices do círculo
    std::vector<GLfloat> vertices;

    // Ângulo inicial e incremento para cada ponto do círculo
    float angle = 0.0;
    float slice = 2 * Pi / static_cast<float>(nPoints);

    // Adiciona o ponto central do círculo (0.0, 0.0, 0.0)
    vertices.push_back(xc); // Xc
    vertices.push_back(yc); // Yc
    vertices.push_back(0.0f); // Zc

    // Calcula as coordenadas de cada ponto do círculo e adiciona ao vetor de vértices
    for (int i = 0; i < nPoints + 1; i++) {
        float x = xc + radius * cos(angle);
        float y = yc + radius * sin(angle);
        float z = 0.0f;

        vertices.push_back(x); // Coordenada X
        vertices.push_back(y); // Coordenada Y
        vertices.push_back(z); // Coordenada Z

        angle += slice; // Incrementa o ângulo para o próximo ponto
    }
```

Extra! createCircle

- Código (Parte 2)

```
    // Identificadores para o VBO (Vertex Buffer Object) e VAO (Vertex Array Object)
    GLuint VBO, VAO;

    // Geração do identificador do VBO e vinculação
    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // Envia os dados do vetor de vértices para a GPU
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(GLfloat), vertices.data(), GL_STATIC_DRAW);

    // Geração do identificador do VAO e vinculação
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);

    // Configuração do ponteiro de atributos para os vértices
    // layout (location = 0) no Vertex Shader, 3 componentes por vértice (x, y, z)
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
    glEnableVertexAttribArray(0);

    // Desvincula o VBO e o VAO para evitar modificações acidentais
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);

    // Retorna o identificador do VAO, que será utilizado para desenhar o círculo
    return VAO;
}
```