PROFESSOR:Renata Muniz do Nascimento	
CURSO: Ciência da Computação	
DISCIPLINA: Cálculo I	
TURMA: 1 CCOMP	DATA: 13/10/2025
ALUNOS(AS): Lucas de Freitas Soares, Emilly Oliveira dos Santos, Guilherme Belcastro de Medeiros, Kaike Cavalcante dos Santos	

Cálculo I - Modelagem Matemática e Funções Aplicadas ao Jogo

Projeto 4 – Quarto Fobo

No ramo da programação a matemática e sua lógica está presente em diversos fatores, como em cálculos simples ou até mesmo em um sistema de verificação. No ramo do desenvolvimento de jogos digitais a matemática permanece estando presente, sendo um dos fatores mais essenciais para o funcionamento da lógica presente nos scripts e no próprio funcionamento de partes do jogo, indo desde questões comuns como a movimentação de um personagem até questões mais complexas como montar um sistema complexo, por exemplo um inventário, se baseando nos cálculos e na lógica matemática.

Dessa forma, visando demonstrar o funcionamento prático da lógica matemática aplicada a programação de desenvolvimento de jogos, este presente relatório descreverá os cálculos utilizados para a criação do jogo Quarto Fobo, jogo este que está em desenvolvimento e apresenta o conceito de um jogo Escape Room 3D com visão geométrica e uma temática voltada para o psicológico, possuindo em seus scripts a lógica da área de exatas para o funcionamento da rotação da sala, do sistema de inventário e afins. No desenvolvimento deste jogo foi utilizado a linguagem de programação C# através da plataforma Unity, plataforma essa que utiliza os cálculos matemáticos de forma “oculta”, ou seja, ele não demonstra diretamente qual o cálculo que foi feito, mas sim uma demonstração mais direta da aplicação do mesmo. Sendo assim, os cálculos serão demonstrados na forma como é apresentado na Unity, assim como em sua forma “natural”.

Script - Rotação de Sala

Start()

Nesse trecho do script obtivemos o código:

```
currentSnapAngle = 90f;
targetRotation = Quaternion.Euler(0, currentSnapAngle, 0);
```

Sendo ele um determinante para a rotação de um objeto através de ângulos, determinando estes ângulos pelos múltiplos de 90º convertendo ângulos de Euler (graus) para um quaternion (x, y, z, w). Analisando de forma matemática, este código trata o seguinte cálculo:

$$x = \sin(\frac{0}{2}) \cdot \cos(\frac{90}{2}), \quad y = \sin(\frac{90}{2}) \cdot \cos(\frac{0}{2}), \quad z = 0, \quad w = \cos(\frac{0}{2}) \cdot \cos(\frac{90}{2})$$

Desta forma ele transforma ângulos comuns em números que representam a rotação tridimensional, definindo o quanto o objeto está girando “de lado”, garantindo que o objeto só gira no plano horizontal (sem inclinar) e determinando pelo w o “peso” ou equilíbrio da rotação calculada.

OnMouseDown()

Nesse trecho do script obtivemos o código:

*float rounded = Mathf.Round(transform.eulerAngles.y / 90f) * 90f;*

Esta parte utiliza o transform.eulerAngles.y / 90f que divide o ângulo Y atual em “blocos” de 90º graus. Também está presente o Mathf.Round(transform.eulerAngles.y / 90f), que, por sua vez arredonda a o ângulo para o múltiplo de 90° mais próximo. Por fim, multiplica por 90f (f sendo o indicador de números reais para a Unity) novamente para trazer o valor de volta à escala angular real.

Com isso, temos o cálculo matemático que aplica o arredondamento do valor, sendo sentenciado a ir para o ângulo múltiplo de 90º mais próximo. Utilizando o valor 184 como exemplo de valor angular, o cálculo ficará da seguinte forma:

$$184/90 = 2.04 \Rightarrow \text{Round}(2.04) = 2 \Rightarrow 2 \times 90 = 180^\circ$$

Assim, seguindo o exemplo apresentado, a Unity entende que a rotação atual está mais próxima de 180° e aplica esse valor ao valor final da rotação.

Update()

Nesse trecho do script obtivemos o código:

Vector3 delta = Input.mousePosition - lastMousePosition;
*float rotationAmount = -delta.x * dragRotationSpeed * Time.deltaTime;*

Esta parte utiliza o delta.x, que calcula a diferença horizontal do mouse em pixels, ou seja, o movimento na horizontal do mouse. Também é aplicado o dragRotationSpeed, se trantando do fator de conversão de pixels para graus por segundo. Por fim o Time.deltaTime, que calcula o tempo entre frames.

Com isso, temos o cálculo matemático que faz a verificação do movimento do mouse de acordo com o eixo solicitado, sendo este o eixo x, o cálculo ficará da seguinte forma:

$$\textbf{graus\_girar} = -(\textbf{pixels\_movidos}) \times (\textbf{velocidade}) \times (\textbf{tempo})$$

Exemplificando, Se o mouse moveu 20 pixels e dragRotationSpeed for 100, então, num frame de 0.0167 s seria representado:

$$-20 \times 100 \times 0.0167 = -33.4°$$

Observação: quando o valor está em negativo, ele determina que o mouse se moveu para a esquerda, visto que ele se baseia no plano cartesiano para poder calcular o movimento do mouse.

Limitação angular

Nesse trecho do script obtivemos o código:

float minAngle = (currentSnapAngle - 90f + 360f) % 360f;
float maxAngle = (currentSnapAngle + 90f) % 360f;
float proposedY = (currentY + rotationAmount + 360f) % 360f;

Essas três expressões controlam a faixa angular permitida (±90° a partir da posição atual). Assim, o código (x + 360f) % 360f; funciona de modo a normalizar o ângulo, garantindo que ele fique sempre entre 0° e 359°, evitando que ângulos negativos ou acima de 360° causem erros. Lembrando que nos códigos em C#, a expressão % se trata do valor

que resta ao fazer uma divisão, buscando resultados sempre em números inteiro, por exemplo, na expressão 3/2 o valor restante seria o número 1.

Com isso, temos o cálculo matemático que aplica este funcionamento determinado uma angulação sempre de números positivos e com valores entre 0 e 359º, o cálculo ficará da seguinte forma, utilizando o -45 como valor base:

$$(-45+360)\%360=315°$$

Assim, seguindo o exemplo apresentado a Unity entende que 315° e -45° são a mesma rotação.

Verificação de intervalo permitido

Nesse trecho do script obtivemos o código:

allowRotation = proposedY >= minAngle && proposedY <= maxAngle;

Esta parte trata de uma comparação matemática pura, verificando se o ângulo proposto (proposedY) está dentro dos limites definidos (entre minAngle e maxAngle). Na qual o cálculo matemático aplicado é justamente o próprio código
Exemplificando, o calor do allow Rotation vai verificar se o movimento do mouse convertido em ângulo atinge os limites propostos, se o ultrapassar, ele retornará para o limite mais próximo:

allowRotation = 45 >= 0 && 45 <= 359;

Assim, o resultado seria justamente o próprio ângulo 45º.

Atualização incremental da rotação

Nesse trecho do script obtivemos o código:

transform.Rotate(0f, rotationAmount, 0f);

Esta parte faz a verificação do quanto a rotação será feita, analisando a posição atual com a quantidade da rotação aplicada, sendo assim, o cálculo ficará da seguinte forma:

$$\text{novaRotaçãoY} = \text{rotaçãoAtualY} + \text{rotationAmount}$$

Entretanto, neste estilo a Unity aplica isso como um produto de matriz de rotação, não como soma direta de graus, internamente, ela atualiza o quaternion multiplicando pelo incremento de angular equivalente.

Determinação do próximo “snap”

Nesse trecho do script obtivemos o código:

```
float angleDiff = Mathf.DeltaAngle(currentSnapAngle, transform.eulerAngles.y);
if (Mathf.Abs(angleDiff) >= 45f)
{
   nextSnapAngle = (angleDiff > 0)
      ? (currentSnapAngle + 90f) % 360f
      : (currentSnapAngle - 90f + 360f) % 360f;
}
```

Esta parte utiliza o Mathf.DeltaAngle(a, b) para calcular a menor diferença entre dois ângulos, podendo ser negativa (para esquerda) ou positiva (para direita). Com issoo cálculo ficará da seguinte forma:

$$delta=((b-a+540)\%360)-180$$

Assim, isso garante que o resultado sempre fique entre -180° e +180°, Mathf.Abs(angleDiff) >= 45f determina quando o usuário arrasta mais de metade de 90°, o sistema já decide qual será o próximo “snap” de rotação.

Exemplificando, se a = 90° e b = 270°:

$$((270-90+540)\%360)-180=(720\%360)-180=0-180=-180°$$

Rotação suave

Nesse trecho do script obtivemos o código:

```
transform.rotation = Quaternion.RotateTowards(
    transform.rotation,
    targetRotation,
    smoothRotationSpeed * Time.deltaTime
);
```

Esta parte calcula o ângulo entre as duas rotações:

$$\theta = \arccos\left(2(q_a \cdot q_b)^2 - 1\right)$$

Move transform.rotation uma fração limitada por:

$$t = \frac{\text{maxDegreesDelta}}{\theta}$$

Calcula nova rotação usando interpolação esférica:

$$q' = \frac{\sin\left((1-t)\theta\right)}{\sin\left(\theta\right)} q_a + \frac{\sin\left(t\theta\right)}{\sin\left(\theta\right)} q_b$$

Com isso, é criado um movimento suave e contínuo.

Verificação de final de rotação

Nesse trecho do script obtivemos o código:

```
if (Quaternion.Angle(transform.rotation, targetRotation) < 0.1f)
```

Esta parte faz a comparação ente o ângulo atual com o ângulo múltiplo de 90º mais próximo. Com issoo cálculo ficará da seguinte forma

$$\text{angle} = \arccos\left(2(q_a \cdot q_b)^2 - 1\right) \times \frac{180}{\pi}$$

Se o resultado for menor que 0.1°, o código considera que chegou no destino e para a rotação.

Atualização visual das paredes em relação ao ângulo

Nesse trecho do script obtivemos o código:

float angle = Mathf.Round(transform.eulerAngles.y) % 360;

Esta parte utiliza o Mathf.Round() para arredondar o ângulo para o inteiro mais próximo. Enquanto o % 360 garante que o resultado esteja entre 0° e 359°. Funcionando muito próximo ao método do “OnMouseDown()”, porém aplicando de forma visual nas paredes da sala

Exemplificando:

$$\text{Se rotação} = 719.6^\circ \Rightarrow \text{Round}(719.6) = 720 \Rightarrow 720\%360 = 0^\circ$$

Assim, seguindo o exemplo apresentado, a Unity consegue identificar qual parede está mais próxima da câmera.

Script – Click Manager

Raycast e clique do mouse (interação)

Nesse trecho do script obtivemos o código:

```
if (Physics.Raycast(ray, out hit, clickRange))
{
    Interactable interactable = hit.collider.GetComponent<Interactable>();
    if (interactable != null)
    {
        interactable.Interact();
    }
}
```

Esta parte faz uma criação do raio (Ray ray), através do código:

Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

Com isso, a Unity converte coordenadas da tela (2D) para coordenadas no espaço 3D da câmera. Sendo assim, o cálculo ficará da seguinte forma:

$$P(t) = P_{\text{origem}} + t \cdot \vec{d}$$

- P_{origem} = posição da câmera
- $\vec{d}$ = direção normalizada do raio passando pelo ponto do mouse
- t = distância ao longo do raio

Assim o Physics.Raycast(ray, out hit, clickRange): irá testar se existe interseção entre o raio e algum collider dentro da distância máxima clickRange. O clickRange atua como limite máximo de t na equação acima. A Unity calcula internamente a interseção de linhas com caixas, esferas e malhas. O GetComponent<Interactable>() e null check: irá verificar que não há contas matemáticas aqui, apenas verificação de referência.