

PROFESSOR: Renata Muniz do Nascimento	
CURSO: Ciência da Computação	
DISCIPLINA: Cálculo I	
TURMA: 1 CCOMP	DATA: 10/11/2025
ALUNOS(AS): Lucas de Freitas Soares, Emily Oliveira dos Santos, Guilherme Belcastro de Medeiros, Kaike Cavalcante dos Santos	

## ***Cálculo I - Aplicação de Derivadas no Jogo***

### **Projeto 4 – Quarto Fobo**

No ramo da programação a matemática de derivadas é muitas vezes vista de forma sutil, sendo aplicada em diversos sistemas, como sanidade, Spawn de entidades, entre outros. No ramo do desenvolvimento de jogos digitais a área de derivadas permanece estando presente como um fator essencial para o funcionamento da lógica de variações constantes dentro do jogo.

Dessa forma, visando demonstrar o funcionamento prático da lógica de derivadas aplicada a programação de desenvolvimento de jogos, este presente relatório descreverá os cálculos das mesmas 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”.

A derivada representa a taxa de variação instantânea de alguma grandeza, aparecendo no jogo sempre que algo é alterado continuamente ao longo do tempo. A derivada corresponde à inclinação da curva que descreve algum valor (sanidade, rotação, efeitos visuais). No código Unity essa “inclinação” é modelada através de variações por frame multiplicadas por `Time.deltaTime`, que funciona como uma derivada discreta:

$$\frac{dX}{dt} \approx \Delta X = \text{velocidade} \times dt$$

### ***Script - SanityManager***

#### **IEnumerator LoseSanity()**

Nesse trecho do script obtivemos o código:

```
sanitySlider.value -= 2f * difficulty * Time.deltaTime * 10f;
```

O valor do `sanitySlider` diminui a uma taxa constante, utilizando do `Time.deltaTime` para efetuar a derivada do cálculo contínuo, em forma matemática, o cálculo pode ser retratado da seguinte forma:

$$\frac{d(\text{Sanidade})}{dt} = -20 \cdot \text{difficulty}$$

Desta forma, o valor da sanidade desce gradativamente, mudando a curva da dificuldade continuamente. No jogo, é perceptível ao observar a sanidade reduzir gradativamente

### **Script – RotacaoDaSala**

#### **Update() - Quaternion.RotateTowards()**

Nesse trecho do script obtivemos o código:

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

Esta parte faz uma velocidade angular, em cálculo ele é representado como:

$$\frac{d\theta}{dt} = \text{smoothRotationSpeed}$$

Assim a rotação muda continuamente de acordo com a taxa angular. Durante o jogo, ela é perceptível ao girar a sala principal, vendo que ela faz uma rotação suave e não trocas bruscas de direção.

### **Script – SanityFXManager**

#### **Update()**

Nesse trecho do script obtivemos o código:

```
vignette.intensity.value = Mathf.Lerp(0f, maxVignette, inverted);
```

Em cálculo ele é representado como:

$$\text{intensidade}(t) = f(\text{sanidade}(t))$$

Através desse cálculo, a própria sanidade já é função do tempo (com derivada).

$$\frac{d(\text{intensidade})}{dt} = \frac{d(\text{intensidade})}{d(\text{sanidade})} \cdot \frac{d(\text{sanidade})}{dt}$$

Durante a gameplay, os efeitos visuais aumentam gradualmente, ou seja, um cálculo de derivada transforma uma variável no tempo em outra variável dependente.

#### **Update() - wobble**

Nesse trecho do script obtivemos o código:

```
float wobble = Mathf.Sin(Time.time * wobbleSpeed) * wobbleIntensity * inverted;
```

O movimento ondulante é definido (cosseno). Matematicamente, ele é representado como:

por uma derivada contínua (seno e

$$x(t) = A \sin(\omega t)$$

$$\frac{dx}{dt} = A\omega \cos(\omega t)$$

No jogo, o jogador sente um efeito de “respiração/instabilidade”, pois a derivada do seno produz uma oscilação suave.

## Script – ItemRotator

### Update()

Nesse trecho do script obtivemos o código:

```
Vector3 delta = Input.mousePosition - lastMousePosition;
```

Esta parte faz uma análise da última posição do mouse, sendo um valor constantemente alternável, em cálculo ele é representado como:

$$\frac{d\text{posição do mouse}}{dt}$$

Assim, a rotação muda continuamente de acordo com a posição do mouse, influenciando a rotação do seguinte trecho:

```
rotationRoot.Rotate(Vector3.up, rotY, Space.World);  
rotationRoot.Rotate(Vector3.right, rotX, Space.World);
```

Portanto, para que a rotação do objeto ocorra, ele necessita da derivada constantemente para informar o novo valor da posição do mouse, influenciando no jogo em como o objeto pode ser rotacionado.

## Script – SanityManager

### IEnumerator LoseSanity()

Nesse trecho do script obtivemos o código:

```
sanitySlider.value -= 2f * difficulty * Time.deltaTime * 10f;
```

Esta parte faz uma a sanidade diminuir continuamente conforme o tempo passa, em cálculo ele é representado como:

$$\frac{d(\text{Sanidade})}{dt} = -20 \cdot \text{difficulty}$$

Assim, atenção aumenta de forma suave, criando **curva de dificuldade contínua**, não brusca.

### DynamicWait()

Nesse trecho do script obtivemos o código:

```
float speed = Mathf.Lerp(minTimeSpeed, maxTimeSpeed, sanityPercent);  
elapsed += Time.deltaTime * speed;
```

Esta parte faz a velocidade do tempo acelerar conforme a sanidade diminui:

$$\frac{dt}{dt} = \text{speed}(\text{sanidade})$$

Assim, o spawn da entidade acontece **cada vez mais rápido**, criando dificuldade crescente proporcional à derivada da sanidade.

## Script – EntidadeSanidade

### MoverAteAlvo()

Nesse trecho do script obtivemos o código:

```
transform.position = Vector3.MoveTowards(  
    transform.position,  
    alvo.position,  
    velocidadeAtual * Time.deltaTime  
);
```

Esta parte faz a A posição da entidade é alterada pela derivada da velocidade. Matematicamente temos:

$$\frac{d(\text{posição})}{dt} = \text{velocidadeAtual}$$

Desse modo, a aproximação é contínua e inteiramente dependente da taxa de insanidade → curva de dificuldade crescente.

### MoverAteAlvo()

Nesse trecho do script obtivemos o código:

```
float t = Mathf.InverseLerp(5f, 0.5f, distancia);  
float escala = Mathf.Lerp(1f, aumentoEscala, t);
```

Esta parte faz a entidade crescer mais rápido conforme se aproxima do jogador. Em cálculo, ficará:

$$\frac{d(\text{escala})}{dt} = \frac{d(\text{escala})}{d(\text{distância})} \cdot \frac{d(\text{distância})}{dt}$$

Assim, a entidade cresce mais rápido conforme se aproxima do jogador, tornando o clima mais tenso e permitindo uma ambientação mais voltada ao terror.



## **Script – EntitySpawn**

### **FuryMode()**

Nesse trecho do script obtivemos o código:

```
yield return new WaitForSeconds(5f);
```

Essa taxa depende diretamente da sanidade, sendo uma derivada implícita. Matematicamente ficando:

$$\frac{dN}{dt} = \text{teleportes por segundo}$$

Esta parte faz com que o teleporte da entidade aconteça repetidamente.

## **Conclusão**

Assim, podemos concluir que as derivadas estão presentes em diversos sistemas e aplicações e, apesar de não serem demonstradas diretamente, elas estão aplicadas através das funções do programa de desenvolvimento, sendo demonstradas de formas mais diretas, porém, com uma vasta gama de funcionalidade executada.