Exploração do algoritmo Fast Inverse Square Root em renderizações 3D

Computação Visual - Projeto N2

https://github.com/BrunoEAH/FISR-3D-Rendering

Professor André Kishimoto

Bruno E. A. Hayek - R.A: 10389776
Gustavo Cunha Ciola - R.A:
Caio Alexandre V. B. de Andrade - RA: 10298313
Nicolas Fernandes Melnik - RA: 10402170

❖ Contextualização

O algoritmo Fast Inverse Square Root (FISR) é famoso por sua implementação no jogo Quake III Arena, em que os desenvolvedores estavam se esforçando ao máximo para programar o jogo no hardware de 1990, o qual era menos potente em comparação ao de atualmente. Era necessário um cálculo de normalização dos vetores, para assim renderizá- los em 3D, porém, o cálculo de uma inversa de uma raiz quadrada $\left(\frac{1}{\sqrt{x}}\right)$ era muito custoso para o hardware da época, no entanto, o programador John Carmack conseguiu criar o algoritmo FISR, o qual normalizava o vetor muito mais rapidamente.

❖ Normalização de vetores

Em ambientes com renderização 3D, como jogos, programas de simulação e design, cada figura de três dimensões possui um vetor Para aplicar iluminação, reflexão e leis da física ao objeto 3D, deve-se aplicar a normalização ao vetor dele, pois em vários casos de renderização 3D, a magnitude do vetor ser de no máximo 1 pode auxiliar em cálculos e na representação da física, além disso, em vários casos a direção do vetor irá desempenhar um papel mais importante do que a magnitude e ao aplicar a normalização, conservaremos a direção dele.

Vamos tomar o exemplo de um vetor $\vec{v}$. Para calcular a magnitude de $\vec{v}$, devemos aplicar a norma euclidiana, em que calculamos a raiz quadrada da soma dos quadrados dos componentes do vetor. Em gráficos 3D, o vetor $\vec{v}$ teria as componentes x, y e z, logo sua magnitude seria:

$$||\vec{v}|| = \sqrt{x^2 + y^2 + z^2}$$

O processo de normalização pode ser descrito como um jeito de deixar os dados padronizados, no caso de vetores nós assumimos que um vetor padrão tem magnitude de 1. Então, para normalizar um vetor, devemos fazer com que ele continue apontando para a mesma direção e dividir cada componente por sua magnitude, tornando assim em um vetor unitário. No exemplo do vetor $\vec{v}$, sendo $\hat{v}$ o vetor unitário, teríamos:

$$\hat{v} = \frac{\vec{v}}{||\vec{v}||}$$

$$\hat{v} = \frac{\vec{v}}{\sqrt{x^2 + y^2 + z^2}}$$

Portanto, essa é a operação que deve ser feita para a normalização de um vetor. O problema emerge quando o computador deve realizar tal operação inúmeras vezes, no processo de renderização 3D, a divisão de números de ponto flutuante é muito custosa, já a operação de raiz quadrada é mais ainda. Em software gráficos, precisamos de *real-time graphics*, logo tais operações devem ser performar rapidamente e com o maior nível de precisão possível.

Deve-se levar em consideração que na época em que Quake III havia sido lançado, isto é 1990, o *hardware* não era tão poderoso quanto o de atualmente, logo havia uma grande necessidade de otimização de cálculos, em especial este normalização de vetores em espaço 3D.

❖ Método de Newton-Raphson

Em análise numérica, existem diversos métodos para a resolução de equações, como o método prático de Briot Ruffini, método Cardano-Tartaglia, método da bissecção e entre outros. O método de Newton-Raphson se destaca por

ser um dos mais poderosos e mais conhecidos métodos, que inclusive foi utilizado no FISR. Este método só necessita de um ponto para iniciar o processo de aproximação para se obter a raiz. Ele se inicia com a aproximação inicial p_0 e posteriormente gera a sequência $\{p_0\}^{\infty}_{n=0}$, utilizando a seguinte equação:

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \; para\, n \geq 1.$$

Como o método de Newton funciona como um esquema iterativo, é necessário fornecer uma tolerância ε. A partir desta tolerância, existem diversas alternativas para critérios de parada, em que a partir de $\varepsilon > 0$ construímos $p_1,...p_N$ até:

$$\left|p_N - p_{N-1}\right| < \varepsilon,$$

$$\frac{\left|p_N - p_{N-1}\right|}{\left|p_N\right|} < \varepsilon, p_N \neq 0,$$

$$\left|f(p_N)\right| < \varepsilon.$$

Exemplo proposto pelo professor Jamil Kalil Naufal Junior, para a aproximação da raiz da seguinte equação:

$$f(x) = x^3 + 4x^2 - 10, \; com\, o\, critério\, de\, parada\, sendo$$

$$\left|f(p_N)\right| < \varepsilon, \; em\, que\, \varepsilon = 0.1$$

Temos:

$$f(x) = x^3 + 4x^2 - 10$$

$$f'(x) = 3x^2 + 8x$$

Sendo p a aproximação da raiz, temos:

$$f(p) = p^3 + 4p^2 - 10$$

$$f'(p) = 3p^2 + 8p$$

Com base na equação do método de Newton temos:

$$p_n = p_{n-1} - \frac{{p_{n-1}}^3 + 4{p_{n-1}}^2 - 10}{3{p_{n-1}}^2 + 8p_{n-1}}, para\ n \geq 1.$$

Logo, supondo que $p_0 = 1$, temos a seguinte tabela para o método de Newton:

n-1	p_{n-1}	$f(p_{n-1})$	$f'(p_{n-1})$	$f(p_{n-1})/f'(p_{n-1})$	p(n)	$f(p_n)$	ε
0	1,00	-5,00	11,00	-0,454545	1,454545	1,540195	1,540195
1	1,454545	1,540195	17,983471	0,085645	1,368900	0,060720	0,060720
2	1,368900	0,060720	16,572868	0,003664	1,365237	0,000109	0,000109

$$Raiz = 0,13689 \cdot 10^{-1}$$
$$\varepsilon = 0,607 \cdot 10^{-1} < 0,1$$

O algoritmo FISR utiliza o método de Newton-Raphson em seus cálculos, tal que sua aplicação será elaborada mais a frente.

❖ Representação de números de ponto flutuante

A representação em ponto flutuante permite que computadores armazenem e processem números reais de forma eficiente, mesmo com limitações de memória e precisão. Isso é feito por meio de um sistema de numeração em ponto flutuante, que utiliza a notação F(β, p, m, M) Onde:

- **β** é a base do sistema (geralmente 2 ou 10),
- **p** é a precisão (número de dígitos significativos na mantissa),
- **M** e **m** são os limites superior e inferior do expoente.

Exemplo proposto pelo professor Rex Medeiros, para escrever o número $(-3,625)_{10}$ no sistema F(10,5,-3,3):

1. Devemos primeiramente transformar o número da sua base 10 para a base 2, que é a base de numeração do sistema
   $(-0,3625)_{10} = (-11,101)_2$

2. Colocar o número na forma normalizada $(-0,11101 \times 10^2)$

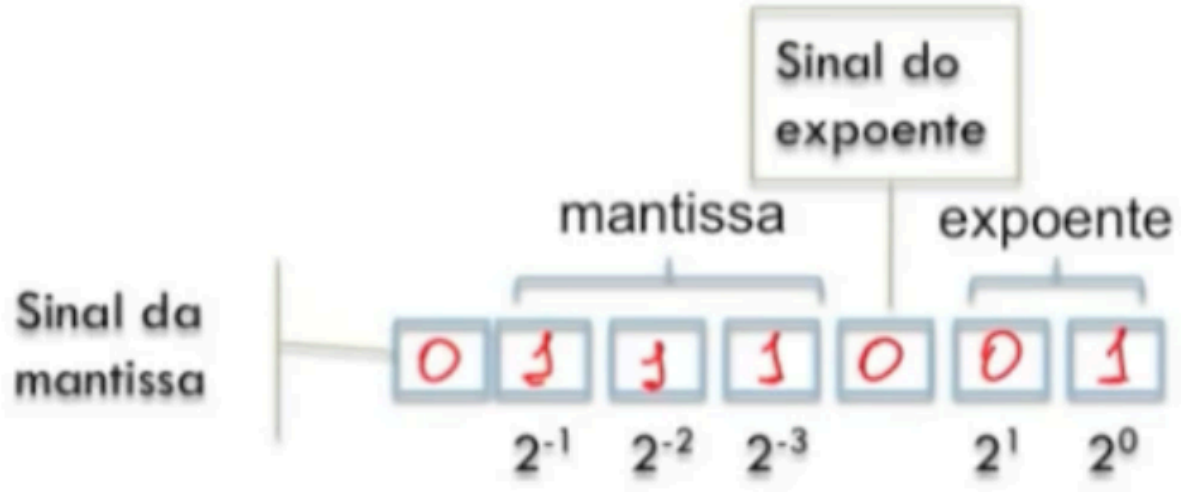
3. Verificar se o número de dígitos na mantissa é menor, igual ou maior do que a precisão do sistema de ponto flutuante. Neste exemplo, o número de dígitos da mantissa (5) é exatamente igual à precisão (5) do sistema, então o número já está perfeitamente representado neste sistema de ponto flutuante

Resultado: $(-0,11101 \times 10^2)$

O IEEE (Institute of Electrical and Electronic Engineers) publicou o relatório *Binary Floating Point Arithmetic Standard 754-1985*, que especifica formatos para precisão simples, dupla e estendida. Esse documento ficou conhecido como o Padrão IEEE 754 (ou Padrão IEEE para Ponto Flutuante) e passou a ser adotado por fabricantes de hardware e desenvolvedores de software que lidam com cálculos em ponto flutuante.

Comumente, utilizamos dois tipos desse padrão: **float** (ponto flutuante de precisão simples utilizando 32 bits para representar um número) e **double** (ponto flutuante de precisão dupla utilizando 64 bits para representar um número).

No exemplo proposto pelo professor Rex Medeiros , temos como é representado o número $(1,75)_{10}$ na memória de um computador que trabalha com o sistema de ponto flutuante F(2,3,-3,3):



Representação em memória do número $(1,75)_{10} = (0,111 \times 2^{1})_{2}$

No entanto, em sistemas de base binária, é comum que números que não sejam potências de 2 sofram erros de arredondamento ao serem representados. Embora muitas vezes imperceptíveis, esses erros podem se acumular ao longo de múltiplas operações, comprometendo a precisão dos resultados. Em muitos casos, o valor real é cortado após um certo número de dígitos, resultando em truncamento, o que agrava ainda mais a perda de exatidão. Por isso, é fundamental controlar esses desvios, especialmente em contextos que envolvem um grande número de cálculos ou algoritmos numéricos sensíveis.

Um exemplo prático pode ser observado na linguagem C:

```
    float a = 0.1f;
    float b = 0.2f;
    float soma = a + b;
    printf("Resultado: %.20f\n", soma); // imprime com alta
precisão
```

Resultado: 0.30000001192092895508

Embora esperássemos 0.3, o resultado mostra uma leve imprecisão devido à representação interna dos números em ponto flutuante. Esse comportamento é

típico e esperado, e demonstra claramente os efeitos do arredondamento e truncamento introduzidos pelo padrão IEEE 754.

Integrando com o tema do projeto: Assim como discutido no artigo “Origin of Quake3’s Fast InvSqrt”, nesses casos a performance era priorizada em detrimento de uma precisão matemática absoluta. A genialidade da função está em explorar diretamente a representação binária dos números de ponto flutuante, definida pelo padrão IEEE 754, junto com uma constante empírica ***(0x5f3759df)*** para gerar um chute inicial extremamente eficiente da raiz quadrada inversa.

Esse tipo de otimização é um exemplo clássico de como compreender a estrutura interna dos dados e o funcionamento de baixo nível pode levar a ganhos significativos de desempenho, o que dialoga diretamente com os objetivos do nosso projeto ao buscar eficiência computacional em ambientes com restrições de tempo ou recursos.

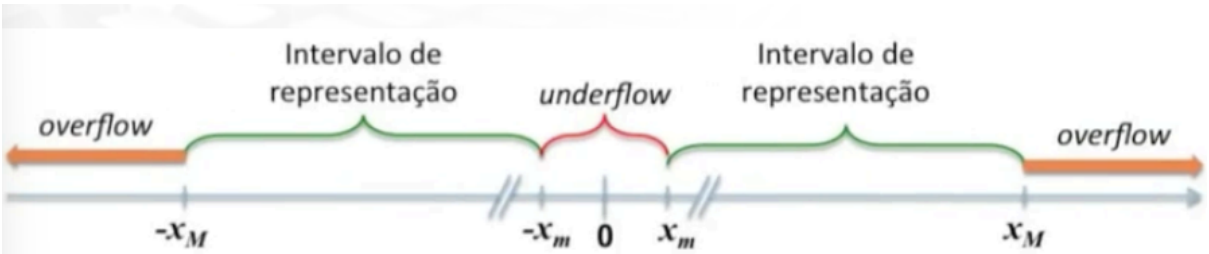
Apêndice – Limitações da representação em ponto flutuante: Overflow e Underflow

A representação em ponto flutuante tem limites definidos pelo intervalo do expoente. Quando esses limites são ultrapassados, ocorrem dois problemas clássicos:

- **Overflow:** Ocorre quando o valor excede o maior número representável pelo sistema. Nestes casos, o resultado é tratado como infinito positivo ou negativo (±∞), o que pode comprometer a lógica do programa.

- **Underflow:** Acontece quando o valor é muito próximo de zero, menor do que o menor valor representável. Nesses casos, o número é arredondado para zero, causando perda de precisão.

  imagem retirado do material do professor Rex Medeiros:



❖ Algoritmo Fast Inverse Square Root

O algoritmo Fast Inverse Square Root (FISR) original do jogo Quake III Arena, com as mesmas variáveis e comentários, é o seguinte:

```
float Q_rsqrt( float number )
{
      long i;
      float x2, y;
      const float threehalfs = 1.5F;

      x2 = number * 0.5F;
      y  = number;
      i  = * ( long * ) &y;                       // evil floating
point bit level hacking
      i  = 0x5f3759df - ( i >> 1 );               // what the f?
      y  = * ( float * ) &i;
      y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//    y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration,
this can be removed

      return y;
}
```

Linhas 1,2 e 3: Declaração de variáveis

Nas três linhas são declaradas as variáveis do **i** do tipo *long*, **x2** e **y** do tipo *float* e **threehalfs** como uma constante do tipo *float*, recebendo o valor de 1,5.

Linha 4: Atribuição a **x2**

Na quarta linha, é atribuído um valor a **x2**, o valor do parâmetro multiplicado por 0,5. Esse valor é utilizado na iteração do método de Newton-Raphson, sendo metade do *input*. O valor é computado antes do cálculo da iteração na linha X, pois ao realizar tal operação antes se torna uma multiplicação a menos a ser escrita durante a iteração.

Linha 5: Atribuição a **y**

Nesta linha é feita a atribuição do valor do *input* a **y**, a qual será utilizada na próxima linha.

Linha 6: Manipulação a nível de bit - Type Punning

Type Punning é uma técnica de programação de baixo nível em que se trata um pedaço de memória como se fosse diferente do que declarado anteriormente. Na sexta linha os valores originais do *float* **y** são reinterpretados como tipo *integer*, para realizar cálculos na mantissa e no expoente. O **&y** capta o endereço de memória da variável y, que utiliza 32 bits por ser um *float*, **(long *)&y** "transforma" o *ponteiro float** em um ponteiro *long**, "enganando" o compilador a interpretar os bits do jeito que desejamos.

Linha 7: A linha mágica

A principal justificativa para a sexta linha se encontra na linha atual, pois a representação IEEE-754 de *floats* permite que os bits estejam dispostos de tal forma que, pode-se manipular a representação do *integer* de maneira efetiva, logo haveria uma estimativa muito do cálculo da inversa de uma raiz quadrada.

Ao fazer a operação de **(i >> 1)**, estamos dividindo pela metade a seção do expoente, deslocando um bit para a direita, após isso subtraímos a "constante mágica" **0x5f3759df**, obtida empiricamente. Ao final, é fornecida uma aproximação rápida da inversa da raiz quadrada.

Linha 8: Voltando à interpretação de inteiro

Nesta linha, fazemos novamente um *Type Punning*, agora com **i** estando com o padrão de bits de um número de ponto flutuante, fazendo com que seja atribuído no **y** e ele receba a primeira aproximação inicial da inversa de uma raiz quadrada.

Linha 9: Método Newton-Raphson

O cálculo que o FISR visa resolver pode ser reescrito como:

$$y = \frac{1}{\sqrt{x}} \Rightarrow \frac{1}{y^2} - x = 0$$

Essa equação pode ser reformulada para a seguinte função: $f(y) = \frac{1}{y^2} - x$

Aplicando o método de Newton-Raphson para essa função, temos que a fórmula iterativa para encontrar uma aproximação de y_n é:

$Com\ base\ em\ f(y)\ ,\ f'(y)\ =\ -\ \frac{2}{3y^3}\ e\ no\ método\ de\ Newton,\ concluímos\ que$:

$$y_n = y_{n-1} \cdot \left(\frac{3}{2} - \frac{xy_{n-1}^{\ 2}}{2}\right)$$

Este cálculo de aproximação de y_n é representado nesta linha, utilizando a constante **threehalfs** e as variáveis **y** e **x2**.

Linha 10: Segunda iteração do Newton-Raphson

Apesar dessa segunda iteração fornecer mais precisão, no contexto do Quake III Arena esse passo seria custoso, levantando novamente a questão de velocidade por acurácia, tornando este passo como opcional.

Linha 11: Final

Na última linha, após o "truque de mágica com bits " e iteração do método de Newton-Raphson, o valor aproximado do cálculo é retornado.

❖ Implementação

Para este projeto, adaptamos um visualizador do mapa do Quake utilizando OpenGL, feito pelo programador Johan Gardahage, em que o código sem nossas alterações se encontra no seguinte repositório: https://github.com/johangardhage/opengl-quake .

Nossas alterações foram de implementar o algoritmo FISR, um algoritmo tradicional e um algoritmo de séries de Taylor para o cálculo normalização de vetores, posteriormente foi feito um benchmark para comparar cada método.

Pelo fato do projeto original ser feito em inglês, todos os comentários que adicionamos ao código estão tanto em português como em inglês, porém a tela que mostrará o *benchmark* dos métodos e as instruções está em inglês.

❖ Método tradicional e séries de Taylor.

O método tradicional se resume a utilizar a standard library do C++ para somente fazer o cálculo da inversa de uma raiz, aplicando assim **1.0f/sqrt(number)**.

O outro método utilizado para a comparação foi o de utilizar as séries de Taylor para conseguir uma estimativa de uma raiz quadrada. A fórmula do método é a seguinte:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Para $\sqrt{x}$, temos que cada termo da aproximação será:

$$\frac{f^{(n)}(1)}{n!}(x-1)^n$$

A iteração para aproximação utiliza 20 termos, sendo um método muito lento para o cálculo da inversa de uma raiz quadrada, feito exclusivamente para aplicar o *benchmark*.

❖ Conclusão

Atualmente, com o *hardware* poderoso que possuímos atualmente, não há mais a necessidade de utilizar o FISR ou outra função que calcule a inversa de uma raiz quadrada, visto que nos processadores modernos já existe uma instrução de código Assembly que faz o cálculo de uma inversa de uma raiz quadrada. Essa instrução mostra a sua velocidade na implementação feita por nós, que prova que o método tradicional possui um desempenho semelhante ou até melhor que o FISR.

Apesar disso, o algoritmo FISR demonstra a criatividade dos programadores Quake III Arena, ao implementarem um algoritmo meticulosamente arquitetado para realizar um cálculo que faz parte de um fator de extrema relevância para gráficos 3D.

Ademais, o algoritmo *Fast Inverse Square Root* demonstra a importância do domínio de conhecimentos matemáticos, computação gráfica e baixo nível que um programador deve ter, para assim provar a sua genialidade resolvendo problemas de maneira criativa.

❖ Referências:

https://mathworld.wolfram.com/UnitVector.html#:~:text=A%20unit%20vector%20is%20a,as%20the%20(finite)%20vector%20.

https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-vectors/a/vector-magnitude-normalization

https://www.lighthouse3d.com/tutorials/glsl-tutorial/vertex-shader/

https://stackoverflow.com/questions/10002918/what-is-the-need-for-normalizing-a-vector

https://youtu.be/ZTywc8v9uU8

https://stackoverflow.com/questions/19299155/normalize-a-vector-of-3d-coordinates-to-be-in-between-0-and-1

Material de aula de Análise Numérica, ministrada pelo Professor Jamil Kalil Naufal Júnior.

Livro Numerical Analysis de Richard L. Burden e J.Douglas Faires.

https://www.inf.ufes.br/~zegonc/material/Introducao_a_Computacao/Aula_Zegonc_Ponto_Flutuante_NEW.pdf

https://www.beyond3d.com/content/articles/8/

https://www.geeksforgeeks.org/fast-inverse-square-root/

https://raw.org/book/algorithms/the-fast-inverse-square-root-algorithm/

https://youtu.be/p8u_k2LIZyo

https://scribe.rip/%40edouard.courty/fast-inverse-square-root-a-masterpiece-of-coding-10da9cdf7929

https://randomascii.wordpress.com/2012/01/23/stupid-float-tricks-2/

https://www.youtube.com/watch?v=9PhvKOraCHM

https://en.wikipedia.org/wiki/Taylor_series

https://en.wikipedia.org/wiki/Square_root_algorithms#Taylor_series