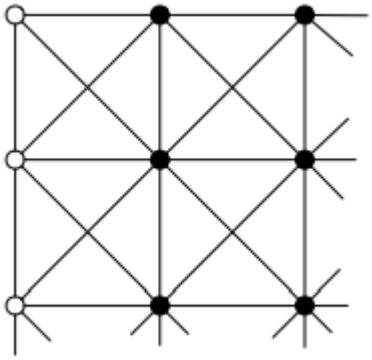
Tecido INF1608

Objetivo

O Objetivo desse projeto é fazer uma simulação de um pedaço de Tecido 3D representado por uma grade de partículas com barras de de restrição.



Pode-se considerar um objetivo secundário fazer uma simulação de uma corda em 2D seguindo o mesmo princípio.

Teoria

Vamos usar o *Método de Verlet* para controlar essa simulação, o sistema se baseia na posição corrente, na posição anterior e um somatório de forças para calcular a proxima posição da partícula em questão.

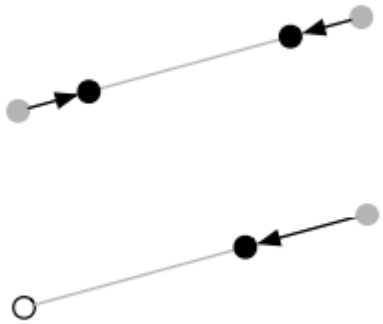
```
X[i+1] = 2*x[i] + X[i-1] + (h*h)/m * f[i]

X[i+1] -> Posição futura
X[i] -> Posição corrente
X[i -1] -> Posição passada
h -> Passo de integração
m -> Massa da Partícula
f[i] -> Somatório de forças atuantes na particula naquele momento
```

Com isso conseguimos encontrar a posição da particula e alterar a sua visualização, mas para que elas se mantenham conectadas é necessário um sistema de restrição, vamos usar o sistema de restrições de barras rigidas.

Serão criadas barras que conectam 2 partículas, iterando por todas as barras é possível ver se os partículas estão com um distância maior que o *comprimento da barra*. Se for esse o caso, é necessário alterar a posição das partículas para as posições corretas, para que a distância entre ela seja igual a barra que as conecta.

Nenhuma partícula bloqueada pode ser movida. Como ilustrado na figura.



Código

Foi usado para esse trabalho a linguagem C#, pois o motor gráfico Unity foi usado para fazer a visualização e C# é a linguagem deste ambiente. Os códigos que serão mostrados nesse relatório são focados exclusivamente na Simulação, porém no código existem códigos auxiliares da plataforma para tornar possível a vizualização em tempo real.

Classe Point

Poucas informações são necessárias para definir um ponto/partícula nesse sistema. Aqui vemos a posição corrente, a passada, sua massa, o somatório de forças e se está presa ou não.

```
public class Point
{
    public Vector3 position, prevPosition;
    public float mass;
    public bool locked;
    public Vector3 sumForce;

    public Point(Vector3 position, float mass, bool locked, Vector3 sumForce)
    {
        this.position = position;
        this.prevPosition = position;
        this.mass = mass;
        this.locked = locked;
        this.sumForce = sumForce;
    }
```

```
    public void ChangeForceX(float amount)
    {
        sumForce.x = amount;
    }

    public void ChangeForceY(float amount)
    {
        sumForce.y = amount;
    }

    public void ChangeForceZ(float amount)
    {
        sumForce.z = amount;
    }
}
```

Classe Bar

Um barra aponta para dois pontos A e B, ela tem um comprimento. Para facilitar alguns cálculos existe um atributo que indica se esta é uma barra diagonal ou não.

```
public class Bar
{
    public Point pointA, pointB;
    public float length;
    public bool diagonal;

    public Bar(Point pointA, Point pointB, float length)
    {
        this.pointA = pointA;
        this.pointB = pointB;
        this.length = length;
        this.diagonal = false;
    }
}
```

Classe Rope e Cloth

As classes que contém as listas de pontos que coordenam a simulação.

Aqui podemos ver a criação de uma corda, cria-se e se posiciona os ponto, e no caso o ponto 0 começa bloqueado, para que a corda não caia.

```
public class Rope
{
    public List<Point> points;
    public List<Bar> bars;
```

```
    public int pointsNum;
    public int numBarInterations;
    public float barLength;
    public Vector3 forceOnPoints;

    public Rope(int pointsNum, int numBarInterations, float barLength, GameObject
originalPointObject, GameObject originalLineObject)
    {
        this.points = new List<Point>();
        this.bars = new List<Bar>();
        this.pointsNum = pointsNum;
        this.numBarInterations = numBarInterations;
        this.barLength = barLength;
        this.forceOnPoints = new Vector3(0, -15, 0);
        /////
        Initialize();
    }

    public void Initialize()
    {
        if (this.bars.Count > 0) this.bars.Clear();
        if (this.points.Count > 0) this.points.Clear();
        root = new GameObject("Rope");
        //
        int barCount = 0;
        for (int i = 0; i < pointsNum; i++)
        {
            Point newPoint = new Point(new Vector3(i * barLength, i * -barLength,
0) + Vector3.left * 10f, 1, i == 0, forceOnPoints);
            this.points.Add(newPoint);
        }

        for (int i = 0; i < pointsNum; i++)
        {
            if (i != pointsNum - 1)
            {
                bars.Add(new Bar(points[i], points[i + 1], barLength));
            }
        }
    }

    public void ChangeForce(float x, float y, float z)
    {
        foreach (Point point in points)
        {
            point.ChangeForceX(x);
            point.ChangeForceY(y);
            point.ChangeForceZ(z);
        }
    }
...
}
```

Simulação

Aqui podemos ver a simulação em si. Com o passo de integração h, conseguimos calcular a proxima posição baseada na força, massa, posição corrente e anterior.

Após isso, se fazem N iterações do processo de restrição de barras rígidas, esse N é possível se ser alterado e diferentes números de iteração causam resultados mais ou menos convincentes.

```
    public void Simulate(float h)
    {
        // nextPos = currentPosition + (1- amort)(currentPosition - lastPosition)
+ h*h / mass * sumForce;
        foreach (Point p in points)
        {
            if (!p.locked)
            {
                Vector3 nextPosition;
                nextPosition = p.position + (p.position - p.prevPosition) + (h * h
/ p.mass) * p.sumForce;
                p.prevPosition = p.position;
                p.position = nextPosition;
            }
        }

        for (int i = 0; i < numBarInterations; i++)
        {
            foreach (Bar bar in bars)
            {
                Vector3 centerBar = (bar.pointA.position + bar.pointB.position) /
2;
                Vector3 dirBar = (bar.pointA.position -
bar.pointB.position).normalized;
                if (!bar.pointA.locked)
                {
                    bar.pointA.position = centerBar + dirBar * bar.length / 2;
                }
                if (!bar.pointB.locked)
                {
                    bar.pointB.position = centerBar - dirBar * bar.length / 2;
                }
            }
        }

        foreach (Point p in points)
        {
            p.gameObject.transform.position = p.position;
        }
        foreach (Bar bar in bars)
        {
            bar.UpdateLine();
        }
    }
```

Aqui podemos ver a criação de um tecido, o príncipio é o mesmo na criação de pontos e barras, porém mais complexos pois existem mais conexões entre pontos.

```
public class Cloth
{
    public Point[,] pointMatrix;

    public List<Bar> bars;

    public int rowNum;
    public int columnNum;
    public int numBarInterations;
    public float barLength;
    public bool useDiagonal;
    public bool renderLines;
    public float CPUTime;
    public LineRenderer line;

    private Vector3 forceOnPoints;

    public Cloth(int rowNum, int columnNum, int numBarInterations, float
barLength, GameObject originalPointObject, GameObject originalLineObject)
    {
        this.rowNum = rowNum;
        this.columnNum = columnNum;
        this.numBarInterations = numBarInterations;
        this.barLength = barLength;
        this.bars = new List<Bar>();
        this.forceOnPoints = new Vector3(165.0f, -15, 36.0f);
        this.sw = new Stopwatch();
        ///////
        Initialize();
    }

    public void Initialize()
    {
        pointMatrix = new Point[rowNum, columnNum];
        int barCount = 0;
        float sqrtTwo = Mathf.Sqrt(2);
        if (this.bars.Count > 0) this.bars.Clear();
        //
        for (int i = 0; i < rowNum; i++)
        {
            for (int j = 0; j < columnNum; j++)
            {
                Point newPoint = new Point(new Vector3(i * barLength, j * -
barLength, 0), 1, j % 4 == 0 && i == 0, forceOnPoints);
                newPoint.name = $"Point {i}|{j}";
                pointMatrix[i, j] = newPoint;
            }
        }
```

```
        for (int i = 0; i < rowNum; i++)
        {
            for (int j = 0; j < columnNum; j++)
            {
                if (j != columnNum - 1)
                {
                    bars.Add(new Bar(pointMatrix[i, j], pointMatrix[i, j + 1],
barLength));
                    barCount++;
                }

                if (j != 0 && i != rowNum - 1) // Diagonal
                {
                    Bar newBar = new Bar(pointMatrix[i, j], pointMatrix[i + 1, j -
1], barLength * sqrtTwo);
                    newBar.diagonal = true;
                    bars.Add(newBar);
                    barCount++;
                }

                if (i != rowNum - 1)
                {
                    bars.Add(new Bar(pointMatrix[i, j], pointMatrix[i + 1, j],
barLength));
                    //
                }

                if (j != columnNum - 1 && i != rowNum - 1) // Diagonal
                {
                    Bar newBar = new Bar(pointMatrix[i, j], pointMatrix[i + 1, j +
1], barLength * sqrtTwo);
                    newBar.diagonal = true;
                    bars.Add(newBar);
                    barCount++;
                }
            }
        }

        ...
```

O motor gráfico dispõe o acesso á váriável **Time.deltaTime**, que foi o tempo em segundo entre o frame anterior e o frame corrente. Com isso é possível usa-lo como passo de integração para chegar na posição futura.

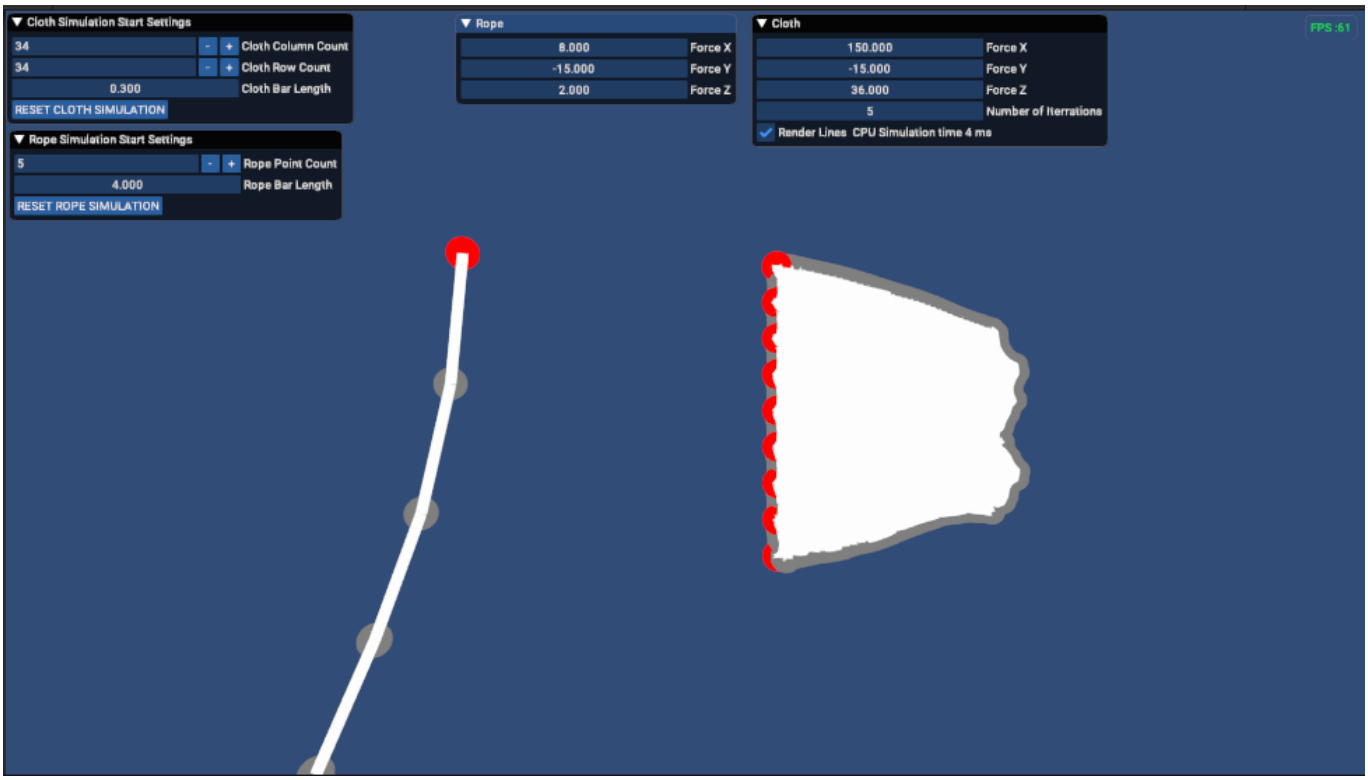
A simulação é chamada todo o frame para atualizar a posição em tempo real.

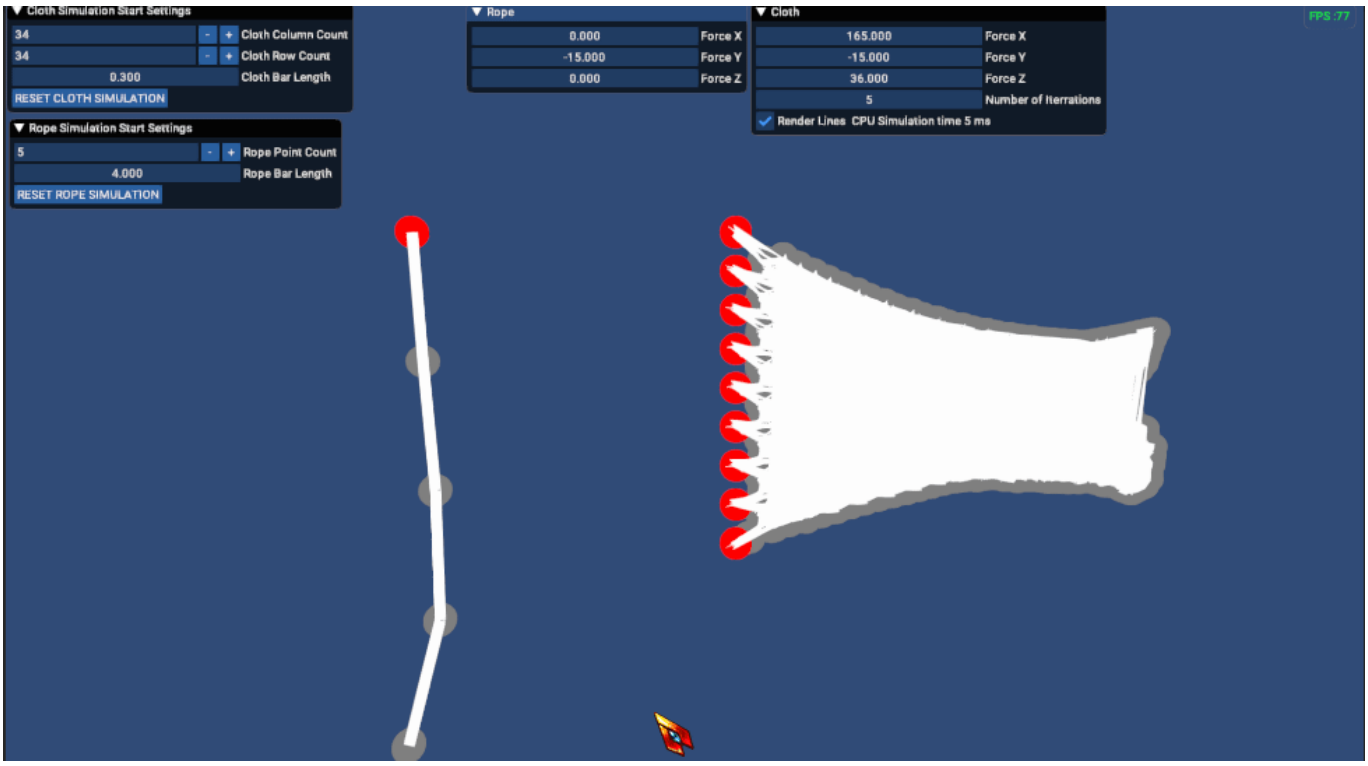
```
    if (simulationStartedOnce)
    {
        cloth.Simulate(Time.deltaTime);
```

```
            rope.Simulate(Time.deltaTime);
        }
```

Resultados

> Gifs estão disponíveis para visualização na página do github:
> https://github.com/nicopaes/Tecido_INF1608





É possível verificar nesse gif o resultado da simulação. É possível clickar nos pontos para bloquea-los e mudar os parâmetros da simulação.

Toda a simulação é feita em tempo real em 3D.

Respondendo algumas das questões colocadas no enunciado do trabalho.

> Quantas iterações, em média, foram necessárias para o relaxamento das barras a fim de se obter resultados convincentes?

Entre 4 em 10 Iterações foram necessárias para chegar em um bom resultado.

> Qual o desempenho da sua simulação? Ele roda em tempo real?

Podemos acompanhar em tempo real o desempenho da simulação, e o seu desempenho varia drasdicamente com a quantidade de pontos, principalmente na simulação do tecido.

Normalmente um bom desempenho para esse motor gráfico é de 60 *Frames por Segundo*, ou seja, o espaço entre um frame e outro, usado como passo de integração em media é 16 ms.

Caso, o tempo da simulação cresça para além disso, a quantidade de frames cairá para compensar, mas a visualização é prejudicada.

Obviamente, existem os fatores gráficos além da simulação em si, mas pelo tempo variado entre um frame e outro, a simulação sempre terá um tempo menor que o passo de integração.

> Sua implementação é genérica para simular qualquer configuração massa-barra de partículas?

Sim, qualquer configuração de um conjunto de pontos e barras nessas configurações pode ser simuladas usando essa implementação.

Interatividade

É possível mudar a direção e intensidade das forças atuantes nas párticulas, bloquear pontos com um clique e mover a câmera livremente pelo espaço.

Para acessar a demonstração:

DEMO V1.0

Créditos

Motor Gráfico - Unity

Interface de Usuário - UImGUI