# P5 Simulación de Ondas con Mapas de Altura


## AUTORES: Camilo Enguix y Pablo Gómez


## ÍNDICE

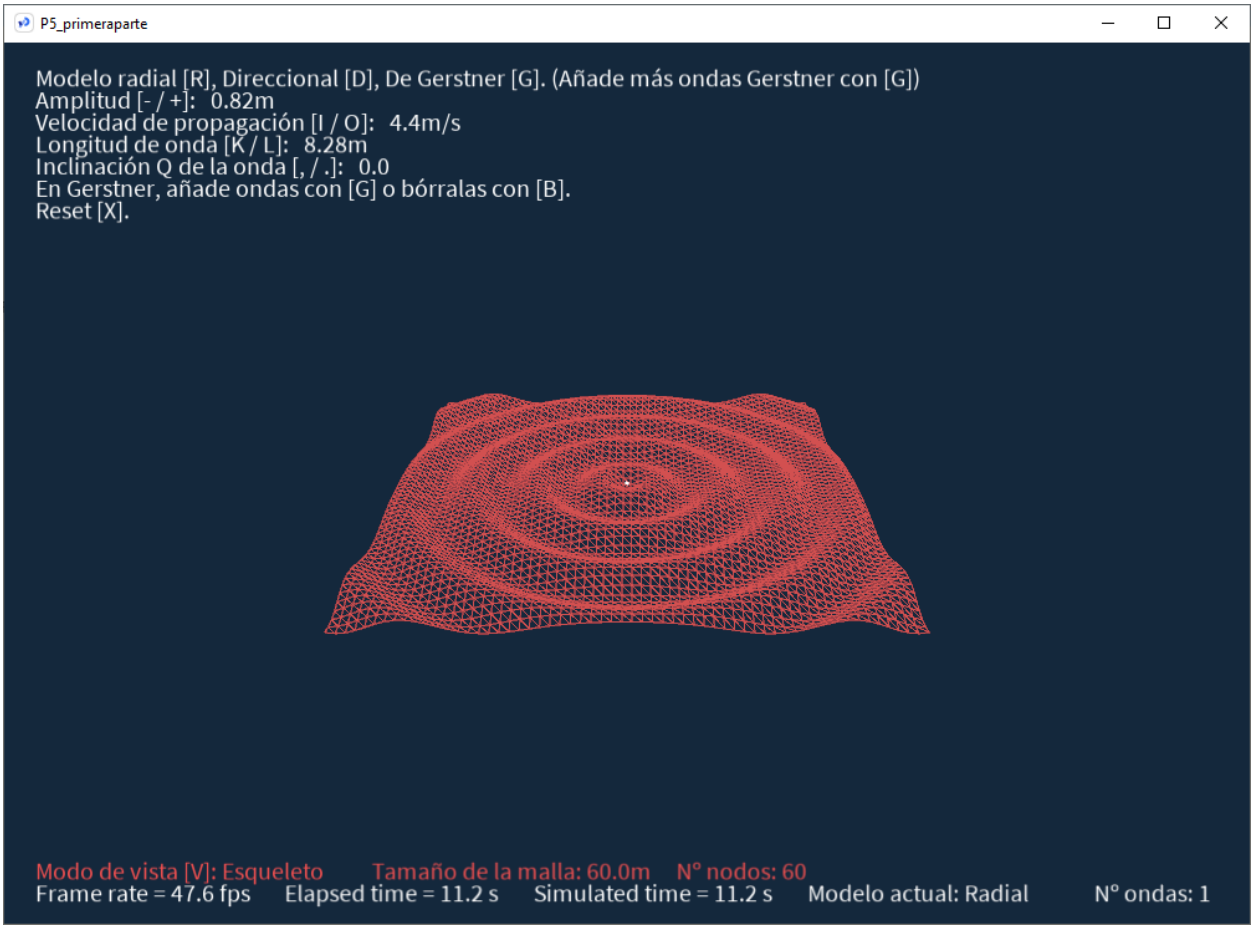


## 1. Introducción.

En esta práctica hemos desarrollado la simulación de tres tipos de ondas distintos a partir de la estructura de un mapa de alturas. Un mapa de alturas consiste en un conjunto de tres coordenadas en un plano. En esta práctica en concreto hemos creado un plano XZ variando el eje y. Eso ocurre en las ondas radial y direccional, mientras que las ondas Gerstner varia las coordenadas de los 3 puntos. En este caso, con un parámetro Q se varia la XZ.

## 2. Primera parte - Simulación base.

En esta primera parte, esta es nuestra interfaz:

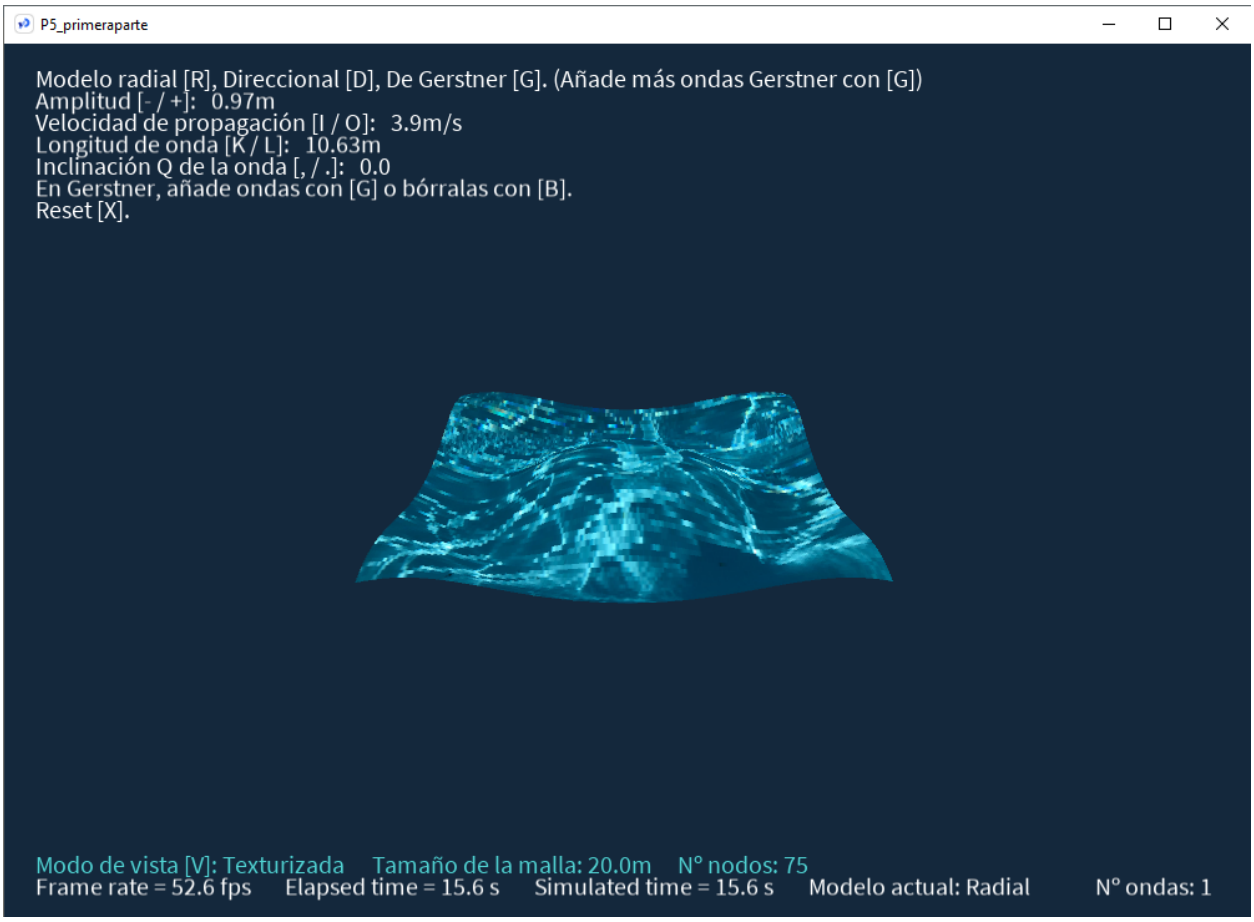



Interfaz del proyecto.

Como podemos ver en la interfaz, podemos variar los tipos de ondas con las teclas R(Radial), la D(Direccional) y la G(Gerstner) con la cual podremos añadir mas ondas para darle un efecto de mar que veremos mas adelante.

Podemos aumentar la amplitud a partir de las teclas + y -, la velocidad de propagación con las teclas I y O, la longitud de onda con la K y L, en el caso de Gerstner, la inclinación Q de la onda con , y . , y por último, como ya he mencionado antes, se pueden añadir mas ondas Gerstner con la G o borrarlas con la B.

Con la letra V podríamos añadirle o quitarle la textura. La malla con la textura se ve de la siguiente forma:




### Clase HeightMap

Es la clase que contiene la malla con los puntos representado el mapa de alturas y la función para dibujar esta misma malla.

El constructor inicializa las variables y llama a las funciones init y draw.

```
HeightMap(float size, int nodes){
    _size = size;
    _nodes = nodes;
    _tamCel = _size/_nodes;
    posini = _size/-2.0;

    init();
    draw();


    //waves
    waves = new ArrayList<Wave>();
  }
```

Las funciones init() y draw()

```
void init(){
    _pos = new float[_nodes][_nodes][3];        //Tamaño del vPosición de cada punto de la malla
    _posdefault = new float[_nodes][_nodes][3]; //Por defecto

    _tex = new float[_nodes][_nodes][2];
    /* Inicialmente, los puntos de la malla tienen un valor en
    x y en z, según su desplazamiento lateral y profundidad,
    respectivamente. Pero la altura se establece en 0. */
```

```
    for(int i = 0; i < _nodes; i++){ // Desplazamiento lateral
      for(int j = 0; j < _nodes; j++){ // Profundidad
        _pos[i][j][idX] = posini + _tamCel/2f + j * _tamCel; //x = desplazamiento lateral
        _pos[i][j][idY] = 0;                    //y = altura
        _pos[i][j][idZ] = posini + _tamCel/2f + i * _tamCel; //z = profundidad

        _tex[i][j][0] =  (float)((j * _tamCel) /_size )* img.width;
        _tex[i][j][1] =  (float)((i * _tamCel) /_size )* img.height;
      }
      _p = new PVector();
    }
    _posdefault = _pos;
  }

  void draw(){
    if(display){
      //Dibujar la malla (con textura)
      noStroke();
      for(int i=0; i<_nodes-1; i++){
        beginShape(QUAD_STRIP);
        texture(img);
        for(int j=0; j<_nodes-1; j++){

          PVector posactual = new PVector(_pos[i][j][idX], _pos[i][j][idY], _pos[i][j][idZ]);
          PVector posnext = new PVector(_pos[i+1][j][idX], _pos[i+1][j][idY], _pos[i+1][j][idZ]);
          vertex(posactual.x, posactual.y, posactual.z, _tex[i][j][0], _tex[i][j][1]);
          vertex(posnext.x, posnext.y, posnext.z, _tex[i+1][j][0], _tex[i][j][1]);
        }
        endShape();
      }
    }else {
      //Draw mesh (sin textura)
      stroke(210,80,80);
      strokeWeight(1);
      fill(255);
      for(int i=0; i<_nodes-1; i++){
        for(int j=0; j<_nodes-1; j++){
          line(_pos[i][j][idX], _pos[i][j][idY], _pos[i][j][idZ], _pos[i+1][j][idX], _pos[i+1][j][idY], _pos[i+1][j][idZ]);
          line(_pos[i][j][idX], _pos[i][j][idY], _pos[i][j][idZ], _pos[i][j+1][idX], _pos[i][j+1][idY], _pos[i][j+1][idZ]);
          line(_pos[i][j][idX], _pos[i][j][idY], _pos[i][j][idZ], _pos[i+1][j+1][idX], _pos[i+1][j+1][idY], _pos[i+1][j+1][idZ]);
        }
        //Unir los puntos de la última columna de la matriz (desplazamiento lateral) (x máxima)
        line(_pos[_nodes-1][i][idX], _pos[_nodes-1][i][idY], _pos[_nodes-1][i][idZ], _pos[_nodes-1][i+1][idX], _pos[_nodes-1][i+1][idY], _p
        //Unir los puntos de la última fila de la matriz (z máxima)
        line(_pos[i][_nodes-1][idX], _pos[i][_nodes-1][idY], _pos[i][_nodes-1][idZ], _pos[i+1][_nodes-1][idX], _pos[i+1][_nodes-1][idY], _p
      }
    }
  }
```

La función addWave(Wave wave) añade una onda a el Array de ondas.

```
void addWave(Wave wave){
    this.waves.add(wave); //Añadir una onda
  }
```

La funcion update() es la encargada de ir actualizando las posiciones dependiendo de el modelo que se este utilizando.

```
void update(){
    PVector delta;
    for(int i = 0; i < _nodes; i++){
      for(int j = 0; j < _nodes; j++){

        _p = new PVector(_pos[i][j][idX], _pos[i][j][idY], _pos[i][j][idZ]);
         _posdefault[i][j][idX] = posini + _tamCel/2f + j * _tamCel; //x = desplazamiento lateral
         _posdefault[i][j][idY] = 0;                    //y = altura
         _posdefault[i][j][idZ] = posini + _tamCel/2f + i * _tamCel; //z = profundidad
         delta = new PVector(0, 0, 0);
        for(int k = 0; k < waves.size(); k++){
          delta.add(waves.get(k).deltaWavePoint(_p));

          _pos[i][j][idY] = delta.y;

          /*Para Gerstner*/
          PVector aux = new PVector(_posdefault[i][j][idX], _posdefault[i][j][idY], _posdefault[i][j][idZ]);
          if(mode ==2 ){
```

```
            _pos[i][j][idX] = aux.copy().x + delta.x;
            _pos[i][j][idZ] = aux.copy().z + delta.z;

          }
        }
      }
    }
  }
```

### Clase Wave

Es la encargada de implementar los tres tipos. Las 3 clases de ondas derivan de esta.

```
Wave(float amp, float lam, float vel, PVector dir, PVector centro){
        _A = amp;
        _lambda = lam;
        _vp = vel;
        _epi = centro;

        _f = 1/_T;
        _w = 2 * PI/_T;
        _k = 2 * PI/_lambda;
        //_vp = _w/_k;
        _phi = _vp * _w;

        _dirN = dir.copy().normalize();
        _dist = db(dir.copy(), _epi);
        delta = new PVector();
        //println("A: " + _A + " lambda: " + _lambda + " vp: " + _vp + " Módulo dist: " + _dist + " dir normalizada: " + _dirN);
    }
```

Mas todas las funciones get de todas las variables para poder recuperarlas en las diferente clases de las ondas.

```
float getDist(){
        return this._dist;
    }
    float getA(){
        return this._A;
    }
    float getLambda(){
        return this._lambda;
    }
    float getVp(){
        return this._vp;
    }
    PVector getDirN(){
        return this._dirN;
    }
    float getQ(){
        return this._Q;
    }
```

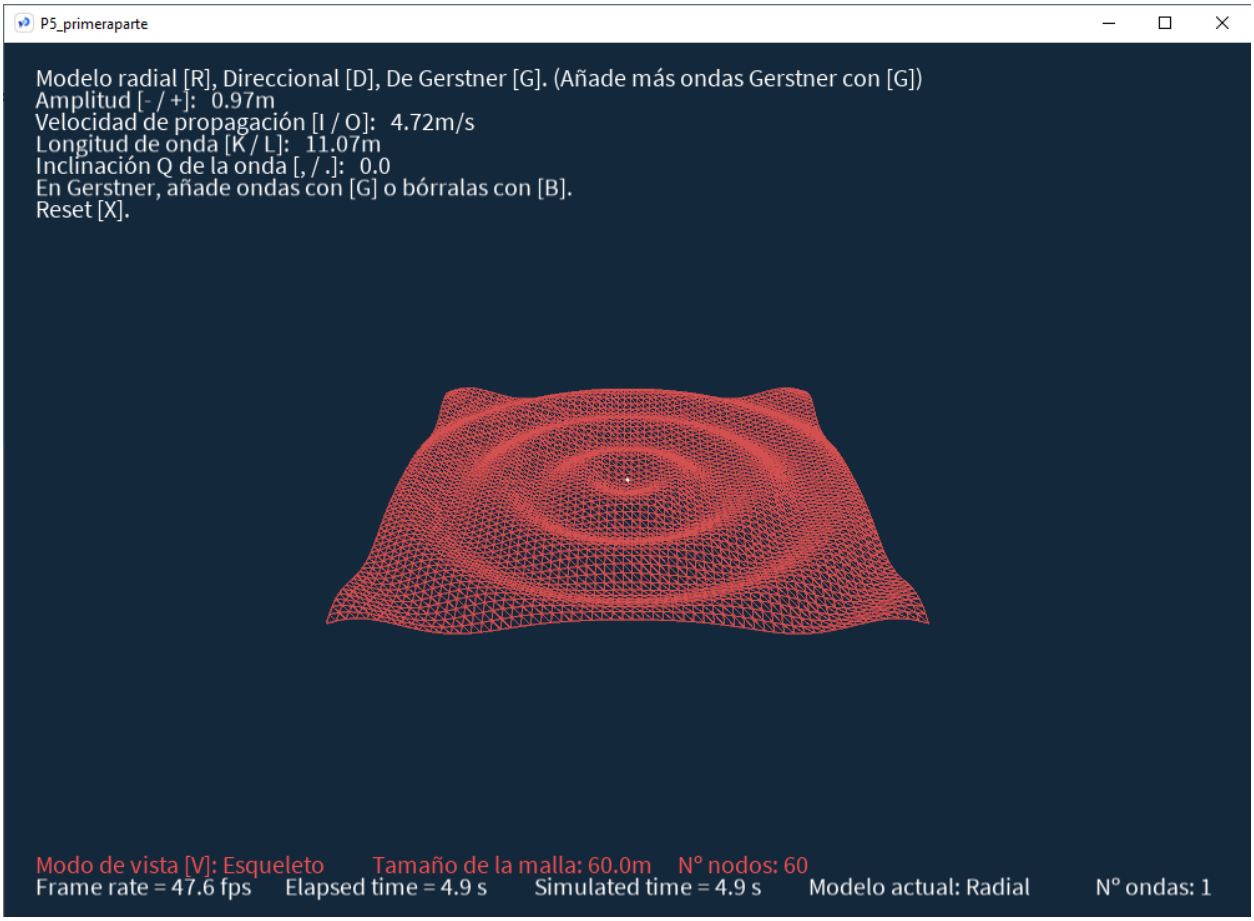
### Clase RadialWave

Implementa la onda radial en la malla.

```
class RadialWave extends Wave{

  PVector epi;
  float s;
  RadialWave(float amp, float lam, float vel, PVector dir, PVector centro){
    super(amp, lam, vel, dir, centro);
    epi = centro;
    setEpi(epi);
  }

  public PVector deltaWavePoint(PVector p){
```

```
    PVector d = PVector.sub(epi, p);
    s = d.mag();

    delta.x = 0;
    delta.y = _A * cos(2*PI/getLambda() * (s - getVp() * _simTime));
    delta.z = 0;

    return delta;
  }
}
```

El resultado final es el siguiente:



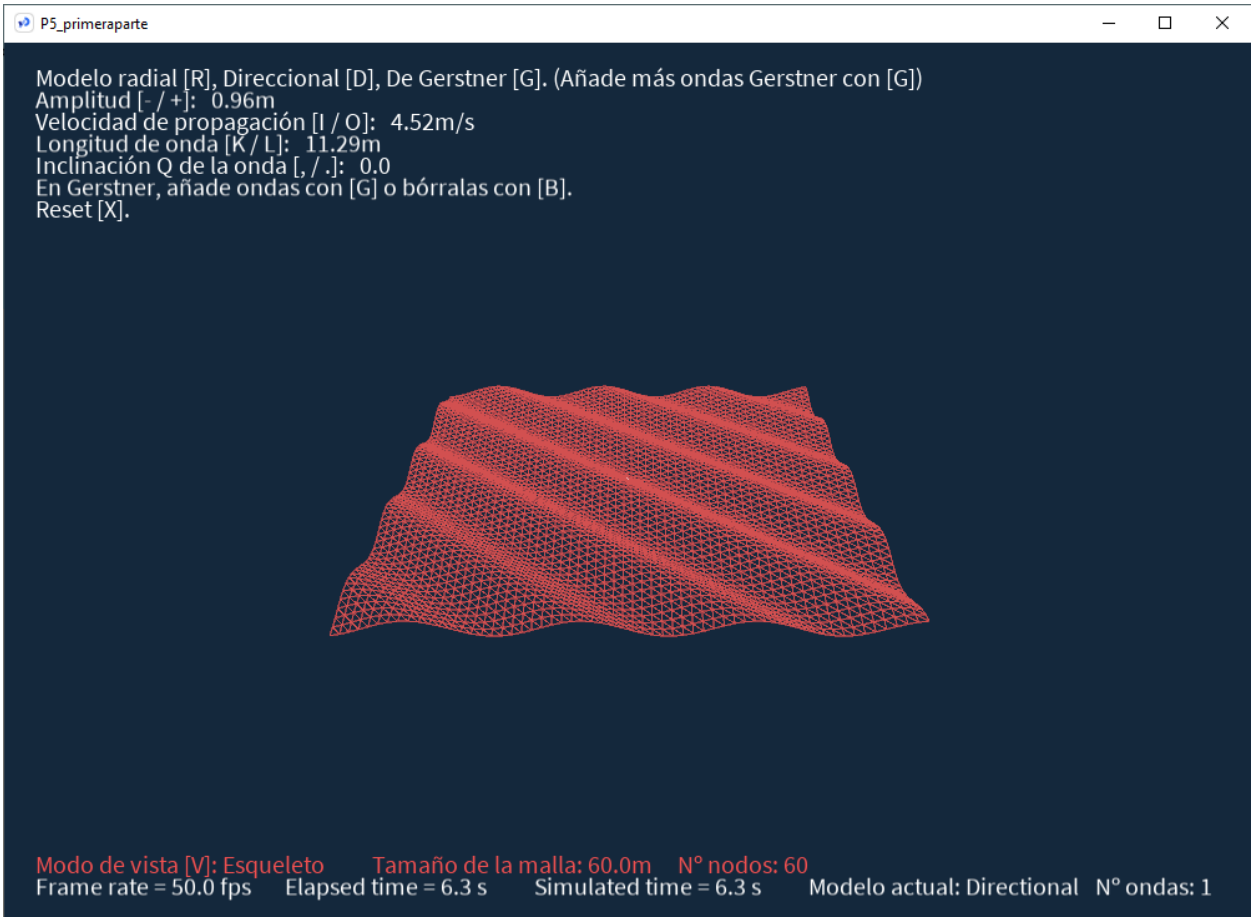

### Clase DirectionalWave

Implementa la onda direccional en la malla.

```
class DirectionalWave extends Wave{

  DirectionalWave(float amp, float lam, float vel, PVector dir, PVector centro){
    super(amp, lam, vel, dir, centro);
  }

  public PVector deltaWavePoint(PVector p){
    //Sacar el vector normalizado de propagación
    delta.x = 0;
    delta.y = _A * sin(2*PI/getLambda() * (getDirN().dot(p) + (getVp() * _simTime)));
    delta.z = 0;

    return delta;
```

```
  }
}
```

El resultado final es el siguiente:

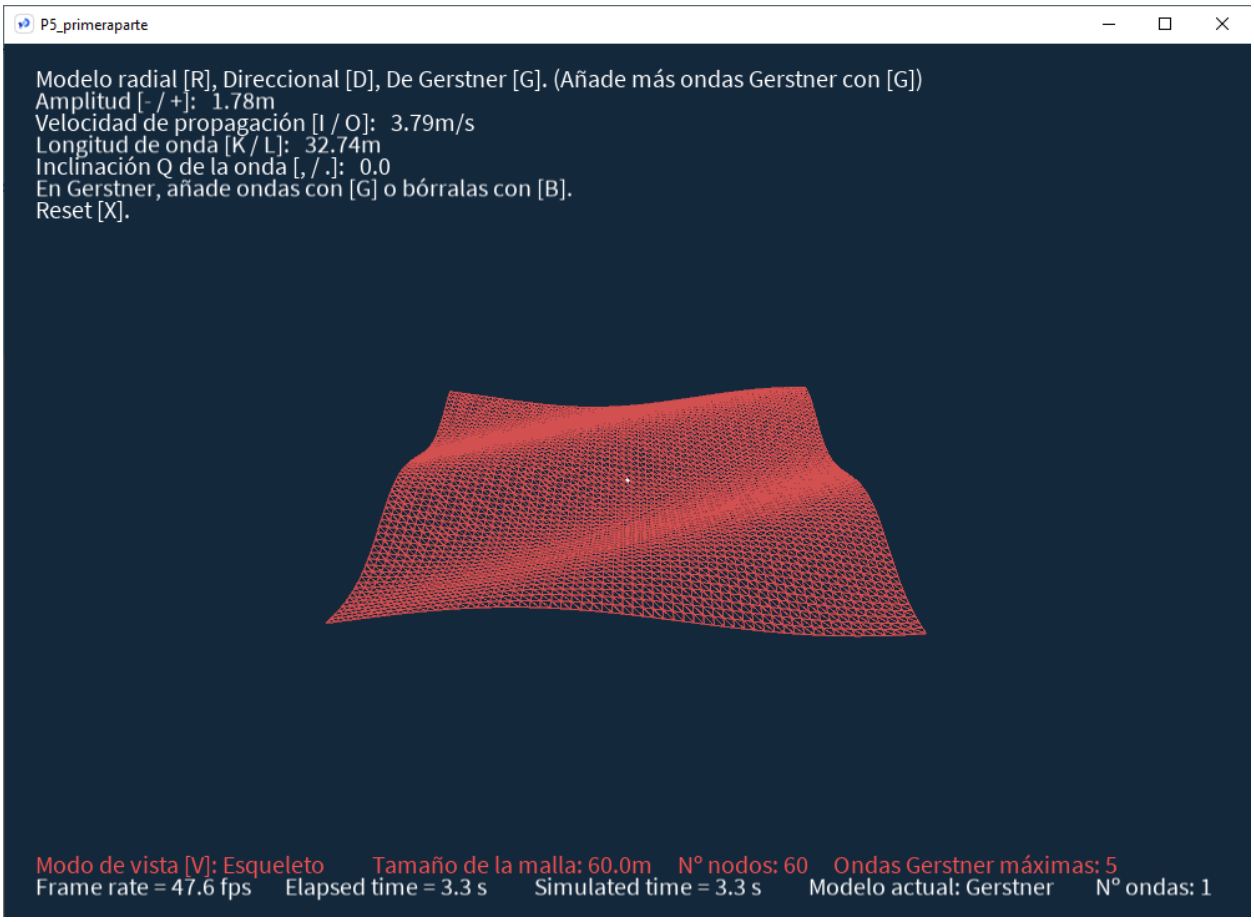



### Clase GerstnerWave

Implementa la onda gertsner en la malla.

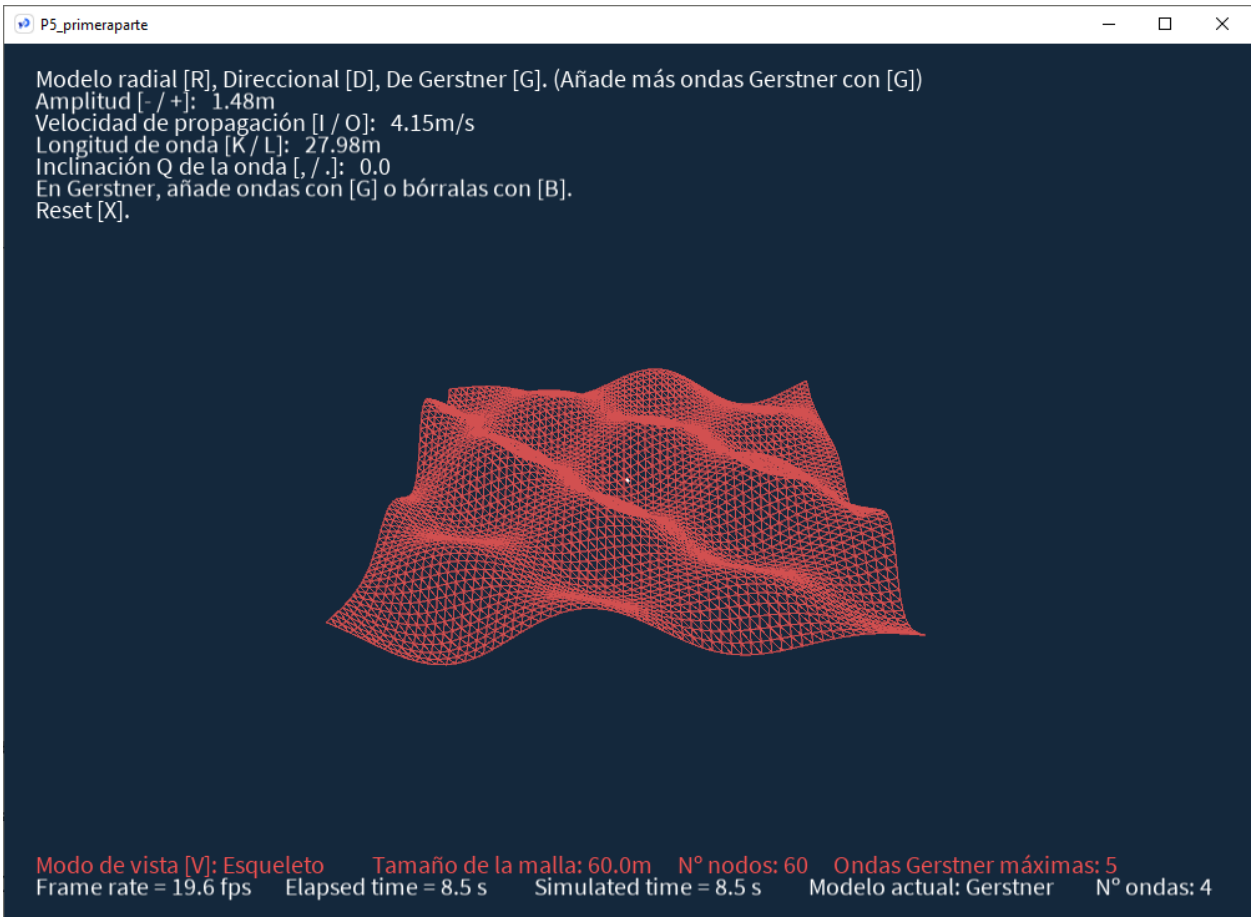
```
class GerstnerWave extends Wave{

  GerstnerWave(float amp, float lam, float vel, PVector dir, PVector centro){
    super(amp, lam, vel, dir, centro);
  }

  public PVector deltaWavePoint(PVector p){

    delta.x = _Q * _A * getDirN().x * cos(2*PI/getLambda() * (getDirN().dot(p) + (getVp() * _simTime)));
    //delta.x = 0;
    delta.y = _A * sin(2*PI/getLambda() * (getDirN().dot(p) + (getVp() * _simTime)));
    //delta.z = 0;
    delta.z = _Q * _A * getDirN().z *  cos(2*PI/getLambda() * (getDirN().dot(p) + (getVp() * _simTime)));

    return delta;
  }
}
```

El resultado final es el siguiente:

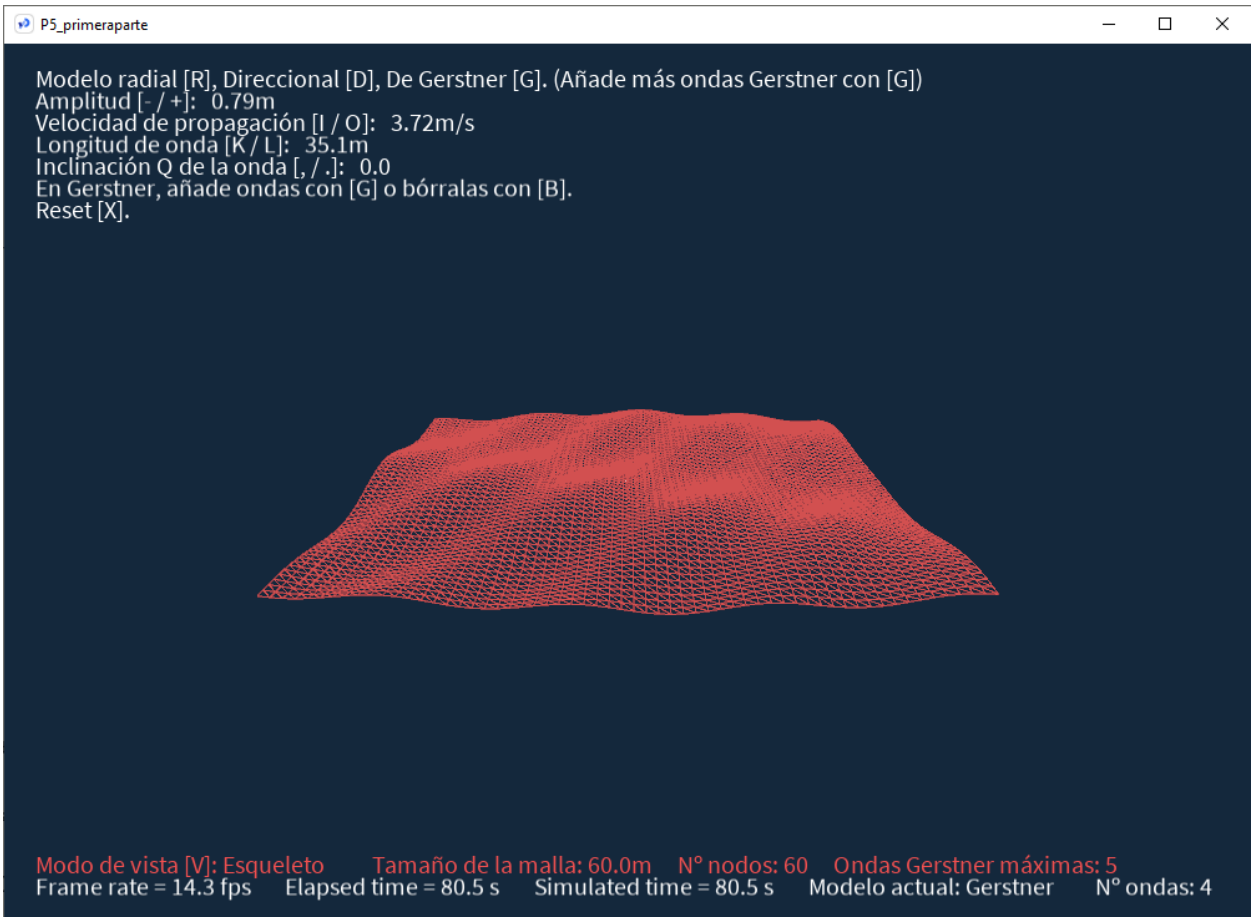



## 3. Segunda parte - Simulación del mar.

Para la simulación del mar lo que podemos hacer con nuestro proyecto es añadir con la tecla G mas ondas Gerstner. Con 4 queda una simulación estable y realista del mar. Una ilustración:



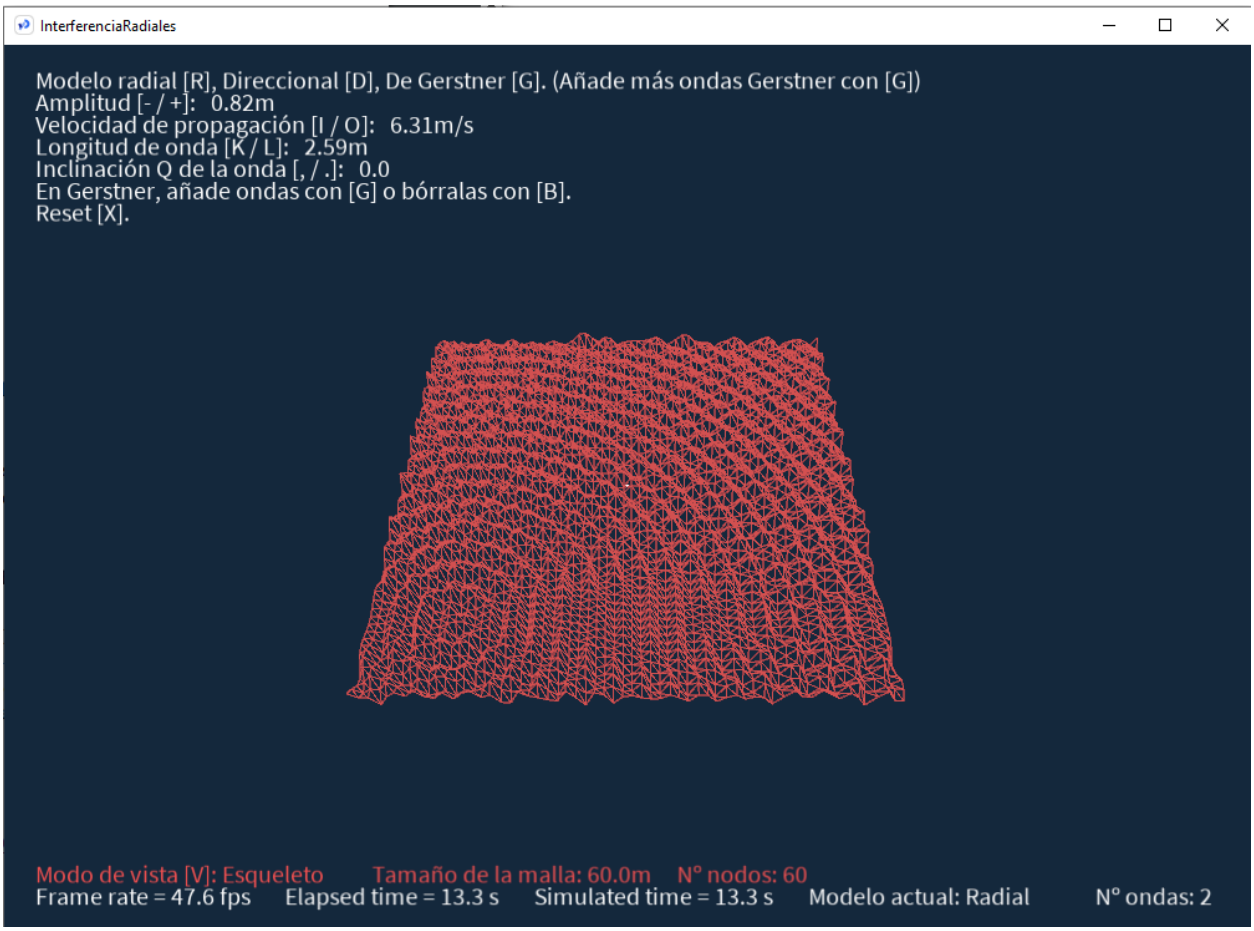

Asi de primeras parece un mar alterado, como si estuviera en una tormenta. Si bajamos la amplitud con la tecla - podemos admirar la simulación del mar como un mar en calma, como si fuera el agua de una playa:




## 3. Segunda parte - Simulación de interferencia de dos ondas radiales.

En esta simulación hemos generado dos ondas radiales en dos puntos paralelos para poder ver la interferencia entre las dos ondas generadas. Nos hemos inspirado en este video: https://www.youtube.com/watch?v=Jqm4f55soJQ&ab_channel=AllRealityVideo

Una captura del resultado es la siguiente:




## 5. Videos.

Primera parte:

https://web.microsoftstream.com/video/ffa137cd-04c4-42dd-9df3-16fcedccdb10

Segunda parte - Mar:

Mar ajetreado: https://web.microsoftstream.com/video/4e8fe56c-4cc1-4356-a60c-ad74f88e93ec

Mar calmado: https://web.microsoftstream.com/video/0fe2779e-ca57-4235-896a-b4846a8fe9d2

Segunda parte - Interferencia de dos ondas radiales: https://web.microsoftstream.com/video/840c898e-e419-448a-a34e-dbb4fef6343d