1. **Abtract**

In this project we will build a C++ application to sumulating and rendering waves on water surfaces. It will also be able to…. We will use QT Creator for programming, QT for window management and OpenGL for 3D rendering.

1. **The function we use**

An obvious way to represent waves is through sine or cosine functions. If one chooses the parameters wavelength, amplitude and wave direction with a certain variance around specified basic values and superimposes a number of these waves, this results in at least one water-like wave train. The following formula results for a sum of sine waves, which are visualized by the Heigth Field H. The water is at rest in the x / z plane of a coordinate system.

y += wave.a \* cos(wave.k \* distanceToOrigin + phi) / (distanceToOrigin + 1);

y = \* cos(k \*

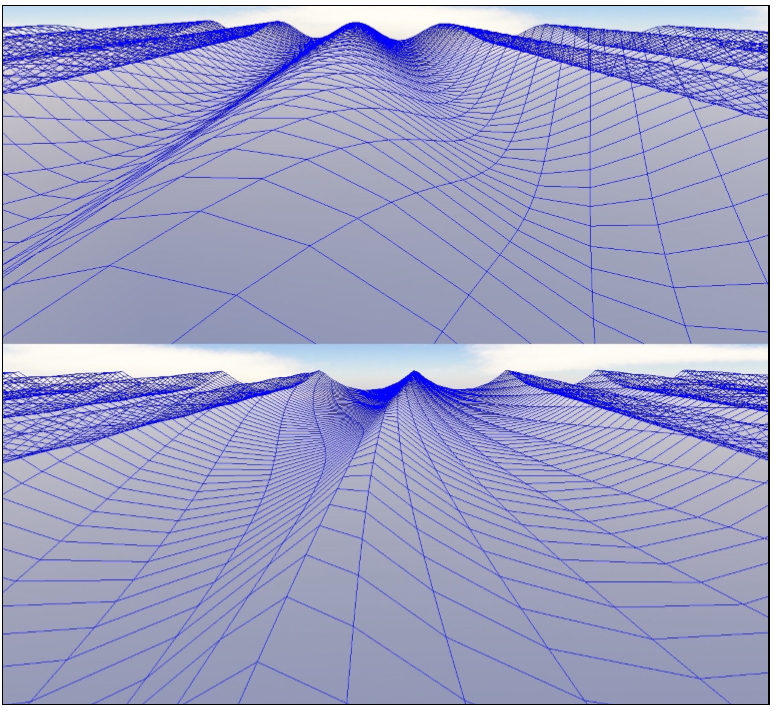
this results in an altitude value in the height field for each position (x, z), with the amplitude a, the direction vector D and a phase shift φ. D stands parallel to the stationary water surface and perpendicular to the associated wave train and the phase velocity c, the frequency f and the so-called wave number k are defined by

c =g·l 2π

f = c/l

k = 2π /l

with the wavelength l and the gravitational acceleration g = 9,81m / s2.



1. **Realisation**

*2.1 Creating an interface with QT*

First oﬀ, we create a new QT widget application. This allows us to use QT Creators design feature to set up our application’s interface. A new OpenGLWidget is placed and will be used as a placeholder for a new custom class inheriting OpenGLWidgets functionality. This class, called OGLWidget, needs to implement the following methods: initializeGL (for setting up OpenGL), paintGL (for doing the actual rendering), resizeGL (for handling resizes of the display window). Additionally, the functions stepAnimation, SetMaterialColor and InitLightingAndProjection1 are used.

*2.2 Creating the data structure*

The data structure is separated in diﬀerent classes. The basic class "Waves" contains the information of the waves like sine waves, Height-Field, coordinate, direction vector, phase velocity, the frequency and the wave number.

The class "Wavesurface" contains the wavesfunction . Logic and data regarding the computation of quad meshes is stored in a seperate class, as are Bezier surfaces and rotational sweep surfaces. In order to allow for easier use of a two dimensional matrix of vertices, a wrapper class containing a two dimensional vector of vertices is introduces.

1. **Mesh**

After creating the required data structure, a method to make a mesh for the waves.

Custom data structure

2-dimensional vector of QVector3D

Dimensions: 50 x 50 -> Best result

1. **Rendering as a wireframe**

Depending on the desired way of rendering the object, diﬀerent draw methods are implemented. These methods are then being called from the paintGL() function.

* 1. *Rendering as a solid*

After drawing the object as a wireframe we want to draw it as a solid cube with lighting. This is being achieved in the method drawQuads() which once again iterates over the list of quads. This time using GL\_Quads, the four vertices of a quad are connected and the area inbetween is ﬁlled. The normal vector for this is calculated using the cross product of the two diagonals vectors.

1. **At**
2. **Recalculate Mesh**

for(unsigned int i = 0; i < mesh.size(); i++){

for(unsigned int j = 0; j < mesh.at(0).size(); j++){

QVector3D \* vec = mesh.at(i).at(j);

double y = calculateWaveHeight(vec->x(), vec->z(), time);

1. vec->setY(y);
2. **Calculate the wave Height**

double distanceToOrigin = QVector2D(x,z).distanceToPoint(wave.O);

double phi = -2 \* wave.pi \* wave.f \* (time + wave.timeOffset);

y += wave.a \* cos(wave.k \* distanceToOrigin + phi) / (distanceToOrigin + 1);

F = exp( i\*abs(X\*2\*pi/8.5))

freq = 10;

for t=0:.001:1

phi = -2\*pi\*freq\*t

F1 = exp( i\* phi) \* F;

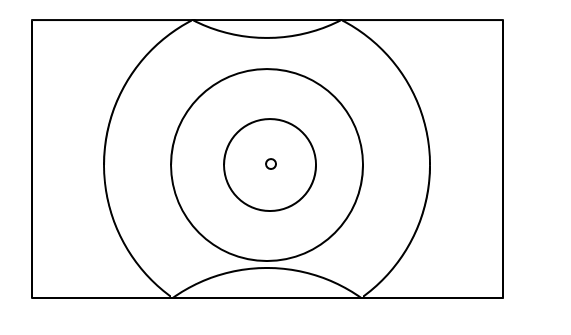
plot( X, real(F1));

axis([-50 50 -5 5]);

drawnow

end

1. **Reflexion**



* Reflection using Image Source Method
* Placing 8 additional waves in the neighbouring ‘tiles’
* Simulating reflection by overlapping waves

// Benachbarte Ursprungspunkte berechnen

QVector2D n1 = QVector2D(-1 \* meshDim\_X - sourceX, meshDim\_Z - sourceZ); // links oben

QVector2D n2 = QVector2D(sourceX, meshDim\_Z - sourceZ); // oben

QVector2D n3 = QVector2D(meshDim\_X - sourceX, meshDim\_Z - sourceZ); // rechts oben

QVector2D n4 = QVector2D(-1 \* meshDim\_X - sourceX, sourceZ); // links

QVector2D n5 = QVector2D(meshDim\_X - sourceX, sourceZ); // rechts

QVector2D n6 = QVector2D(-1 \* meshDim\_X - sourceX, -1 \* meshDim\_Z - sourceZ); // links unten

QVector2D n7 = QVector2D(sourceX, -1 \* meshDim\_Z - sourceZ); // unten

QVector2D n8 = QVector2D(meshDim\_X - sourceX, -1 \* meshDim\_Z - sourceZ); // rechts unten

// Urpsungspunkt ausgeben für Debugging

cout << "N1: " << n1.x() << " | " << n1.y() << endl;

cout << "N2: " << n2.x() << " | " << n2.y() << endl;

cout << "N3: " << n3.x() << " | " << n3.y() << endl;

cout << "N4: " << n4.x() << " | " << n4.y() << endl;

cout << "N5: " << n5.x() << " | " << n5.y() << endl;

cout << "N6: " << n6.x() << " | " << n6.y() << endl;

cout << "N7: " << n7.x() << " | " << n7.y() << endl;

cout << "N8: " << n8.x() << " | " << n8.y() << endl;

// Wellen mit entsprechenden Ursprungspunkten einfügen

waveSurface->addWave(Wave (amplitude, wavelength, 0.0, n1));

waveSurface->addWave(Wave (amplitude, wavelength, 0.0, n2));

waveSurface->addWave(Wave (amplitude, wavelength, 0.0, n3));

waveSurface->addWave(Wave (amplitude, wavelength, 0.0, n4));

waveSurface->addWave(Wave (amplitude, wavelength, 0.0, n5));

waveSurface->addWave(Wave (amplitude, wavelength, 0.0, n6));

waveSurface->addWave(Wave (amplitude, wavelength, 0.0, n7));

waveSurface->addWave(Wave (amplitude, wavelength, 0.0, n8));

1. **Dynamic Parameters**

Sliders to determine origin, amplitude and wavelength

Checkbox to toggle reflection

Using QTs Signal - Slot mechanism

Repopulating the array of waves on every parameter change

void OGLWidget::**updateWaveX**(int sourceX)

{

const Wave \* w = waveSurface->getWave(0);

double sourceZ = w->O.y();

double amplitude = w->a;

double wavelength = w->l;

updateWaves(amplitude, wavelength, (double) sourceX, sourceZ);

}

void OGLWidget::**updateWaveZ**(int sourceZ)

{

const Wave \* w = waveSurface->getWave(0);

double sourceX = w->O.x();

double amplitude = w->a;

double wavelength = w->l;

updateWaves(amplitude, wavelength, sourceX, (double) sourceZ);

}

void OGLWidget::**updateWaveA**(int amplitude)

{

const Wave \* w = waveSurface->getWave(0);

double sourceX = w->O.x();

double sourceZ = w->O.y();

double wavelength = w->l;

updateWaves((double) amplitude, wavelength, sourceX, sourceZ);

}

1. **Final Demonstration**
2. **References References**

[1] Prof. Dr. Martin Hering-Bertram, Lecture CG18\_1, HSB, 2018.

[2] Prof. Dr. Martin Hering-Bertram, Lecture CG18\_2, HSB, 2018.