Water simulation in OpenGL

Field of study: Media informatics
Semester: 4

Date: Juni 22, 2018

Authors:
Fabian Niehaus
Tuyet Nguyen

Contents

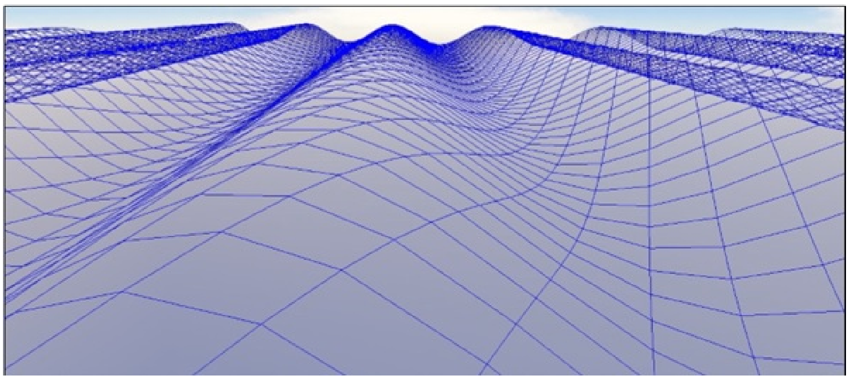
List of Figures

1 Abstract

In this project we will build a C++ application to simulate circular waves on a 3D mesh surface. We will also simulate reflection using the Image Source Method. We will use QT Creator for programming, QT for window management and OpenGL for 3D rendering.

2 Wave theory

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.
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 ÏĘ. 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 =gÂůl 2ÏĂ
f = c/l
k = 2ÏĂ /l
with the wavelength l and the gravitational acceleration g = 9,81m / s2.



3 Realisation

3.1 Creating a basic 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 QOpenGLWidget is placed and will be used as a placeholder for a new custom class inheriting QOpenGLWidgets 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 InitLightingAndProjection[1] are used.

[1] Taken from Prof. Dr. Martin Hering-Bertrams OpenGL_Example

3.2 Creating the data structure

The data structure is separated in different classes.
The basic class "Waves" contains the 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.

3.3 Creating the surface 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

3.4 Calculating 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);
```

Figure 1: Determining adjacent quads (pseudo code)

```
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
```

Figure 2: Determining adjacent quads (pseudo code)

3.5 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.

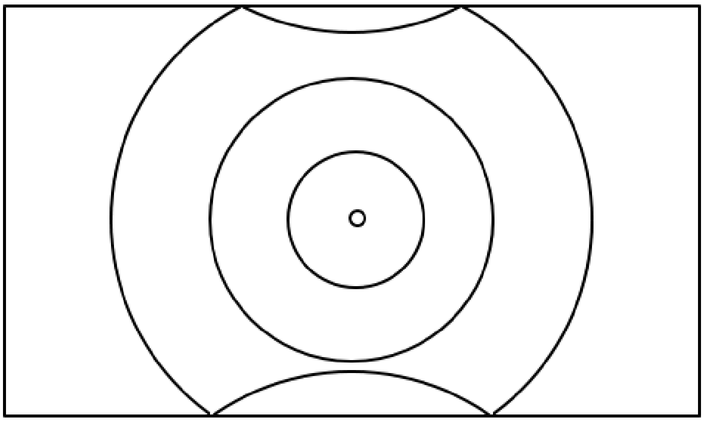
3.6 Rendering as an opaque surface

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.

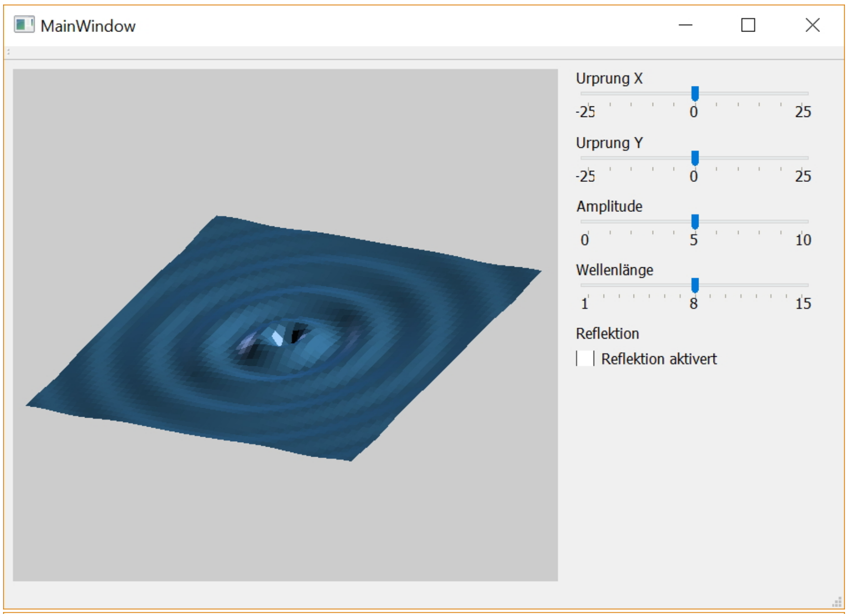
3.7 Reflection

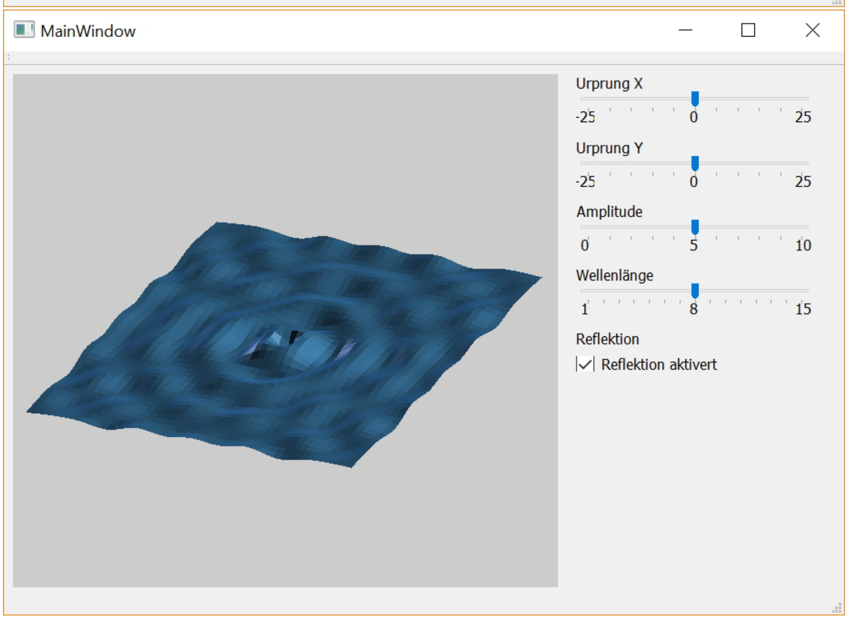
```
// Benachbarte Ursprungspunkte berechnen
        QVector2D n1 = QVector2D(-1 * meshDim_X - sourceX, meshDim_Z - sourceZ
    ); // links oben
        QVector2D n2 = QVector2D(sourceX, meshDim_Z - sourceZ); // oben
        QVector2D n3 = QVector2D(meshDim_X - sourceX, meshDim_Z - sourceZ); //
     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 fÃr 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 einfÃgen
        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));
```

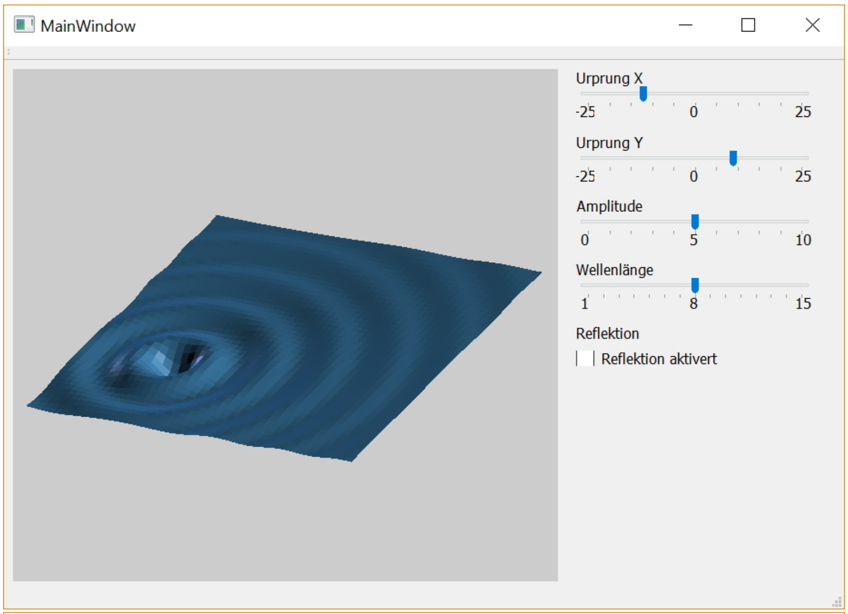
Figure 3: Determining adjacent quads (pseudo code)

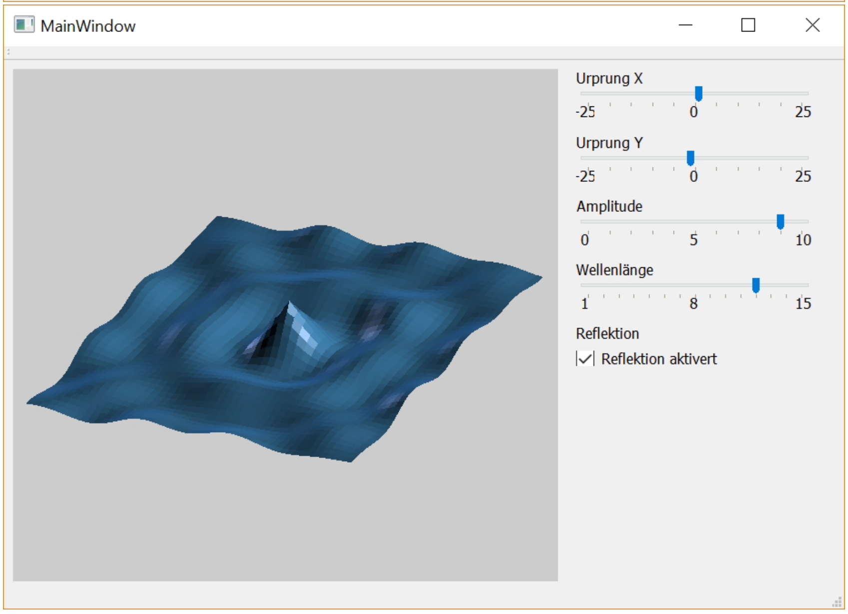


3.8 Fazit


MainWindow
Urprung X
-25
0
25
Urprung Y
-25
0
25
Amplitude
0
5
10
Wellenlänge
1
8
15
Reflektion
Reflektion aktivert


MainWindow
Urprung X
-25
0
25
Urprung Y
-25
0
25
Amplitude
0
5
10
Wellenlänge
1
8
15
Reflektion
Reflektion aktivert


MainWindow
Urprung X
-25
0
25
Urprung Y
-25
0
25
Amplitude
0
5
10
Wellenlänge
1
8
15
Reflektion
Reflektion aktivert


MainWindow
Urprung X
-25
0
25
Urprung Y
-25
0
25
Amplitude
0
5
10
Wellenlänge
1
8
15
Reflektion
Reflektion aktivert