OpenGL 3

March 7, 2018

Computer Graphics

Shaders and Texture Mapping

Please note that you will need to (partly) re-use your code from the previous OpenGL assignments.

In this assignment you will implement animation. The first part will be the most common form of animation present in most games and other interactive media. In the second part we experiment with customising vertex shaders to achieve interesting effects such as animating water.
Important: You may submit both parts in 1 project or submit two Qt projects in different `Code` folders: `CodeAnimation` and `CodeWater`.

1 Common animation (4 points)

In this part you will be using the basic concept of movement: $x_{t+1} = x_t + t * \Delta x$
Or in words: the next position can be calculated by taking the current position and adding the distance travelled in a certain time period (speed $\times$ time).

Up until now your application behaved like this:

1. change setting/parameter/rotation etc.
2. redraw view

Using `QTimer` we will be able to iteratively call a certain function which may change/update settings/position and then render the scene.
Your application already has a `QTimer` available under the name `timer`. Verify that the constructor of `MainView` still contains the following line (or add it if it's not there):
`connect(&timer, SIGNAL(timeout()), this, SLOT(update()));`
At the end of `initializeGL()` you should add:
`timer.start(1000.0 / 60.0);`
Which will start calling your `paintGL()` function 60 times per second.

Rotating a model (1 point)

After the last assignments you should have a (textured) model available.

1. Create a float data member which stores the current rotation. The hints later in this assignment will describe a better way of storing this information.
2. Before setting the model transformation matrix, add a small value (in degrees) to the rotation variable. **Note:** your value will be updated 60 times each second, keep this in mind when choosing a value!
3. Apply your transformations and the additional rotation.
4. When you run the application your model should be rotating.

Experiment with different values, and also with translations (using a QVector3D as position and some speed). When you are confident, you can proceed with this part.

Animating a complete scene (3 points)

Now you should be able to render a complete animated scene. You may create your own scene, so be creative! It should satisfy the following guidelines:

- Load at least two different meshes (may use your own, but also see the notes below)
- Load at least two different texture images (may use your own, make sure that the sides are a power of 2 (256, 512, 1024, etc.))
- Draw at least two instances of each mesh (so at least 4 objects in total)
- All four instances should be animated in a different way (different speed, rotation, scaling etc.)
- The camera should be able to rotate around the screen either using the rotation dials or the mouse and keyboard. The scale slider could be used as zoom (expand the entire scene from its center), but this is not required.

Some examples:

- Four objects bouncing on the ground with different speeds and directions.
- A solar system with a rotating star, 2 planets orbiting the star and some Tesla (or other object) orbiting one of the planets.
- A fish-bowl with different fish constructed from spheres and a cat preying on them.

Some important notes and hints:

- Define functions for doing things you do often if you have not done this already! Examples: creating a VAO with a VBO and uploading data to buffers, loading textures, updating object positions etc.
- When using your own objects, make sure that the files are formatted in triangles. This settings is called `triangulate faces` in most software (such as Blender).
- Look at the Lecture slides on Transformation and Projections, slides 57 until 62. You should use three matrices: the projection, view and model matrix. The projection matrix is still the same matrix compared to the previous exercises. Apply your rotations of the camera on the view matrix, and use the model matrix to transform your instance to the desired places.

- **Important!** When doing lighting calculations, make sure that all the coordinates and vectors are either in world coordinates or camera coordinates!
- Create a struct (or class) storing the location/orientation, speed, VAO ID (use an unsigned int), number of vertices and a texture ID (use an unsigned int) to keep track of all instances and their properties.
- Usually your `paintGL()` looks like this:
  1. clear screen
  2. update positions/orientation of all objects
  3. bind shader
  4. set global uniforms (projection, view matrix, light position etc.)
  5. for each object:
     (a) bind vao of object
     (b) bind texture of object
     (c) set uniform values for the current object (model matrix, material etc.)
     (d) call `glDrawArrays` (or similar)
  6. end of `paintGL()`

Optional extension

Make it a small interactive game. Example: dodging falling spheres with a user-controlled cat.

2 Advanced vertex manipulation: water shader (4 points)

Animating water can use thousands of particles. Implementing and animating this on the CPU requires a lot of computational power. In this part you will be implementing a basic water shader on the GPU.
To refresh your memory, a sine wave has the following parameters:

$$\text{height} = \text{amplitude} * \sin(2\pi * (\text{frequency} * \text{position} + \text{phase} + \text{time})) \tag{2.1}$$

Height map (1 point)

Let's start with transforming the flat plane to a wave-like shape.

1. Download the `grid.obj` file from Nestor. It is a flat plane defined at $z = 0$ of 100×100 vertices connected in a square grid pattern. You will be using this as the base surface for our water shader.
2. Add the .obj file to your project's `resources.qrc` file and load the mesh.
3. Create a new shader program with two new shader source files.
4. Edit your fragment shader to get a `vec2` containing uv coordinates as input and output them as a color (`fColor = vec4(u, v, 0.0, 1.0)`)
5. Your vertex shader should provide inputs for at least vertex coordinates, normals and uv coordinates.

6. Store the current vertex position in a `vec3` variable in your vertex shader.
7. Apply the projection, view and model matrices as you did in the previous exercises, but using the variable instead of the input coordinates directly.
8. When you run the application you should see a single colored quad on your screen.
9. Change your vertex shader to change the z value of the stored position based on the u coordinate. (using sin for example, experiment with frequency, amplitude etc.)
10. When you run the application, you should see that the surface is no longer flat as seen in Figure 2.1.
11. Provide a screenshot of your surface in the `Screenshot` folder.

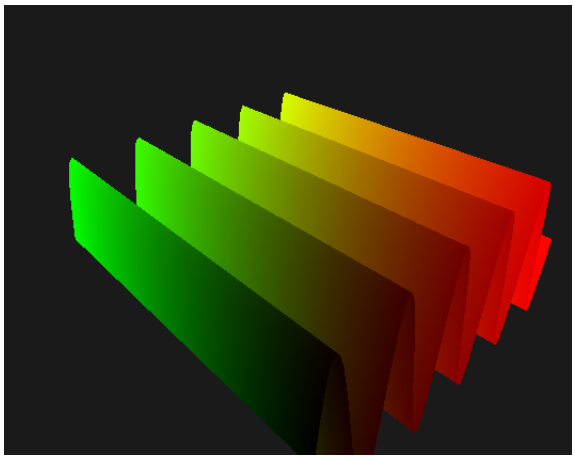

Figure 2.1: *A simple sine wave pattern in one direction colored by uv coordinates*

Normals (1 point)

The normals provided by the model are no longer correct, since the shape of the surface has been changed. However, it is quite easy to derive these since the shape of the surface is defined by a function.

1. Create an output in the vertex shader for the normal.
2. Calculate the value of the derivative of the function you used for your wave using the same u coordinate. **Hint:** The derivative of $\sin x$ is $\cos x$. However do keep the chain rule in mind!
3. The normal may then be calculated using:
   `vec3 normal = normalize(vec3(-dU, -dV, 1.0))`
   where `dU` and `dV` are the value of the derivative in that direction. Since we only consider u, `dV` is zero.
4. Edit your fragment shader to show the normals (mapped to $[0, 1]$) provided by the vertex shader.
5. When you run the application your results should look similar to Figure 2.2
6. Make a screenshot of it and store it in the `Screenshot` folder.

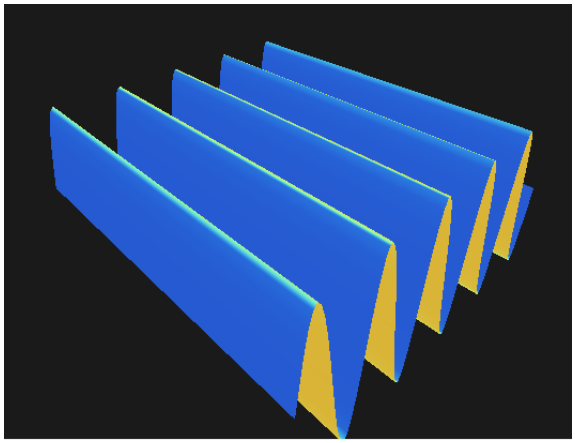

Figure 2.2: *Normals of the wave retrieved from the wave function*

Multiple waves (1 point)

Water consists of many waves. Results of the wave height function and the wave derivative function can simply be added together.

1. Create three uniform arrays of floats (e.g: `uniform float amplitude[(num waves)]`): for amplitude, frequency and phase in your vertex shader. Start with 2 waves, and add more when your method works.
2. Fill these arrays with values in your C++ code (using `void glUniform1fv(GLint location, GLsizei count, const GLfloat *value);`)
3. Create two functions in your vertex shader: `float waveHeight(waveIdx, uvalue)` and `float waveDU(waveIdx, uvalue)`.
4. Implement both functions where `waveIdx` is an offset into the three uniform arrays.
5. In main you should be able to loop over all waves and sum the results in two floats variables. Then use that to calculate the final wave height and wave normal.
6. Your results may look something like Figure 2.3.

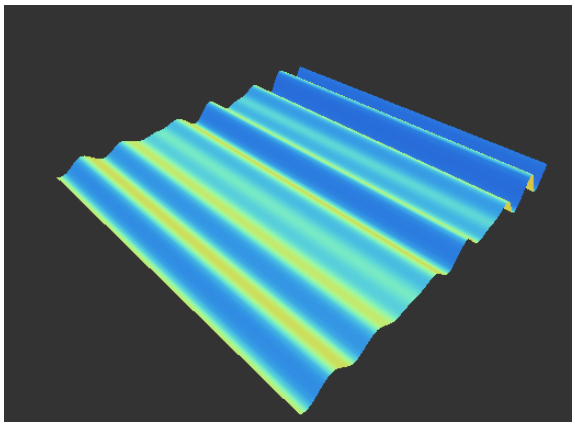

Figure 2.3: *Normals of a complex wave created with 8 simple waves. Uses small (< 0.1) amplitudes to get nice results.*

7. Re-implement Phong shading. Define the material constants (ambient, diffuse and specular) yourself. The color of the material must be derived from the final wave height. Use `smoothstep` to set the boundaries and use `mix` with the value from the `smoothstep` function to interpolate between two colors. The results should look similar to Figure 2.4.

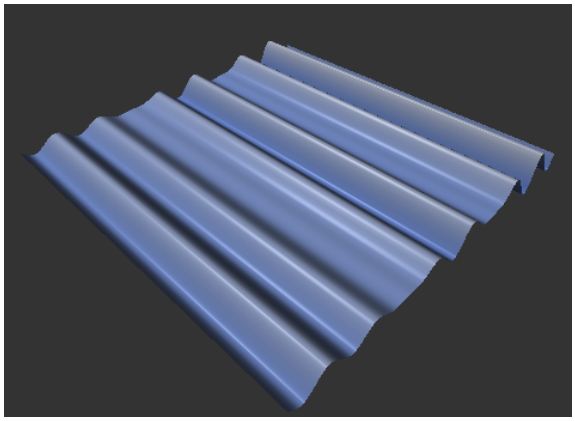

Figure 2.4: *Phong shaded complex wave created with 8 simple waves using a blue color for low parts and white for higher parts.*

Animating water (1 point)

Your water is standing still at this moment. Add an extra `uniform float` for time in your vertex shader and corresponding data members in `MainView` to be able to set the uniform. In your `C++` code the time should be increased by a small value (e.g. 1.0/60.0) for each frame. Adjust your `waveHeight` and `waveDU` functions to take this uniform time into account. Using the timer from the first part of this assignment, your water should now be animated with moving waves.

Optional extensions

Possible extensions include:

- Create radial waves in both the u and v direction: simulate something falling into the water.
- Apply waves on a sphere instead of a flat surface (using the normal at the vertex to change the height)

Deadline

See Nestor (*Time Schedule*). Details on how to submit your work can also be found on Nestor (*Lab Assignments*).

Assignment submission

Please use the following format:

- Main directory name: `Lastname1_Lastname2_OpenGL_3`, with the last names in alphabetical order, containing the following:

- Sub-directory named Code, containing the modified Qt framework (please do **NOT** include executables or build directories)
- Sub-directory named Screenshots with your screenshots or videos.
- README, a plain text file with a short description of the modifications/changes to the framework along with user instructions. We should not have to read your code to figure out how your application works!

The main directory and its contents should be compressed (resulting in a **zip** or **tar.gz** archive) which is the file that should be submitted (using the *Assignment Dropbox*). An example of a file to be submitted associated with the second OpenGL assignment would be: `Kliffen_Talle_OpenGL_3.tar.gz`

Assessment

See Nestor (*Assessment & Rules* → Assignment assessment form).