Assignment 4

Computergrafik 1

MMTBECGIL – WS 2023

Overview

The assignment consists of 3 tasks

A. shader, uniforms, diffuse lighting

B. texture mapping

C. displacement mapping

The idea is that you start at task A and only continue to the next task once it‘s solved, using your solution as base. All of the tasks are things that we talked about in class before christmas, **and there is sample code available for each task on the wiki**.

Hand in the tasks separately, i.e. in folders named TaskA/TaskB/TaskC.

Make sure you put all shader files / textures into your repository.

Overview

As I explained in class, due to the fact that each of us has a slightly different GPU, it‘s possible that you write shader code that does in fact not work on my machine.

To be safe, you should test your code on the Lab PCs in room 351, to make sure your shaders work there. If your shaders don‘t work on my machine, I‘ll go and try them on the Lab PCs as well. This way, you will also know if you‘ve added all necessary shader files and textures to your repository.

Task A

General Shader Programming, Uniforms, Transformations

First, Fix Vertex Shader & Uniform Variables

When you start the project, you won‘t see the scene, and you‘ll get a bunch of error pop-up messages. The problem is with the shaders used for rendering:

- The vertex shader doesn‘t work properly (it doesn‘t do its main job…)
- Exactly one uniform variable is misspelled in the C++ code. (Its name doesn‘t match the name used in the shader)
- Exactly one uniform variable is spelled correctly, but the wrong datatype is used when trying to set it.
- You also get some pop-up messages because some uniforms are unused, **due to the vertex shader not doing its job**. Unused uniforms are optimized away by the compiler, so they generate a „this uniform doesn‘t exist“ message when they‘re accessed in C++. Once you‘ve fixed the vertex shader, these uniforms will actually be used by the shader & the related pop-ups will go away.

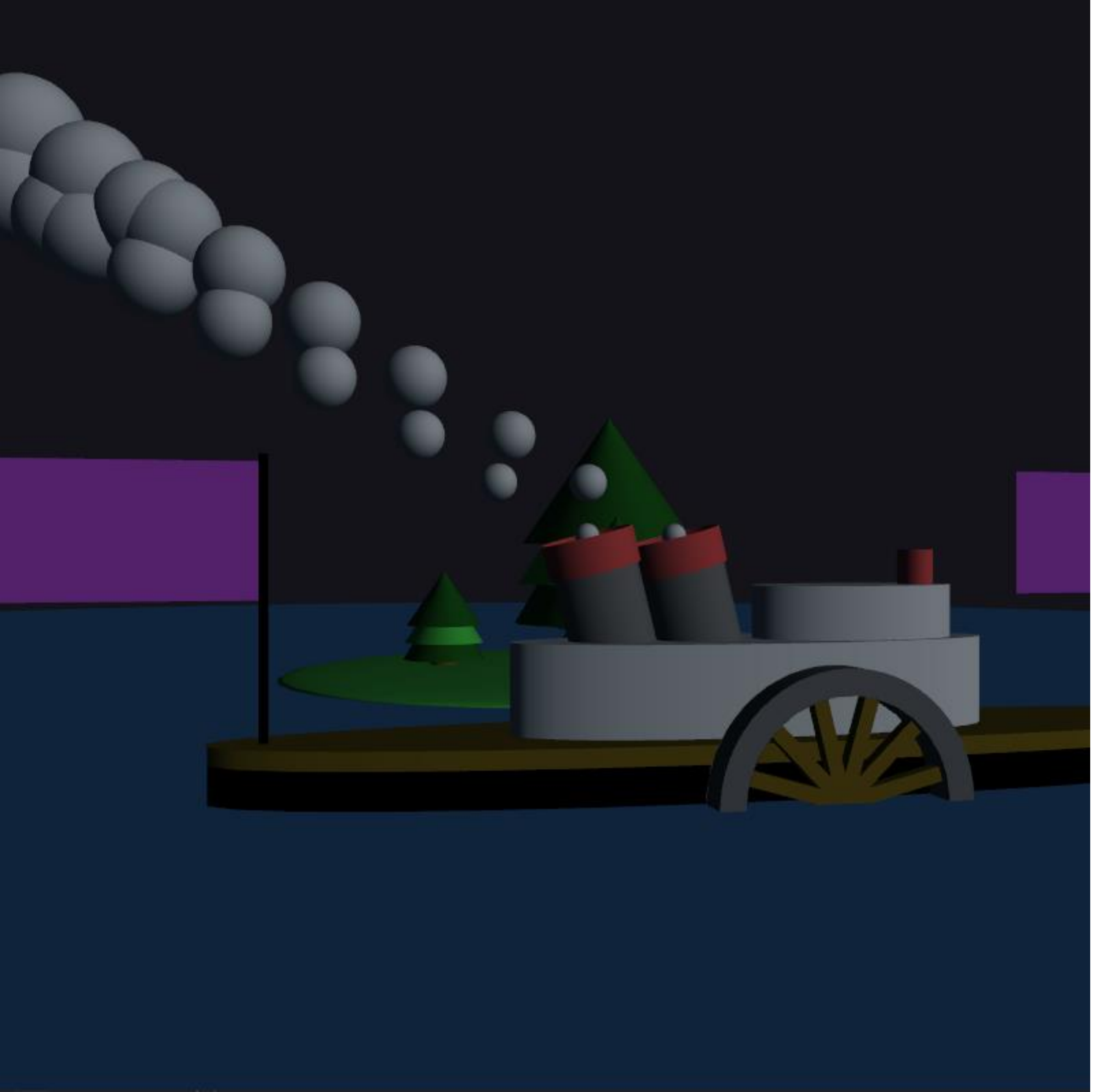
First, check the vertex shader and correctly transform the vertex position.

Then, check the uniforms used in the vertex & fragment shader and fix the uniform related errors.

There are 2 places in the C++ code where uniforms are set:

- When the OpenGL program object is activated, scene specific uniforms such as light color, ambient color, camera matrices etc. are set
- When a specific object is drawn, object specific uniforms such as color and model/normal matrix are set

Always use the proper function for setting uniforms, i.e. setUniformMat4 when setting a mat4, setUniformMat3 when setting a mat3, setUniformVal when setting a float etc.

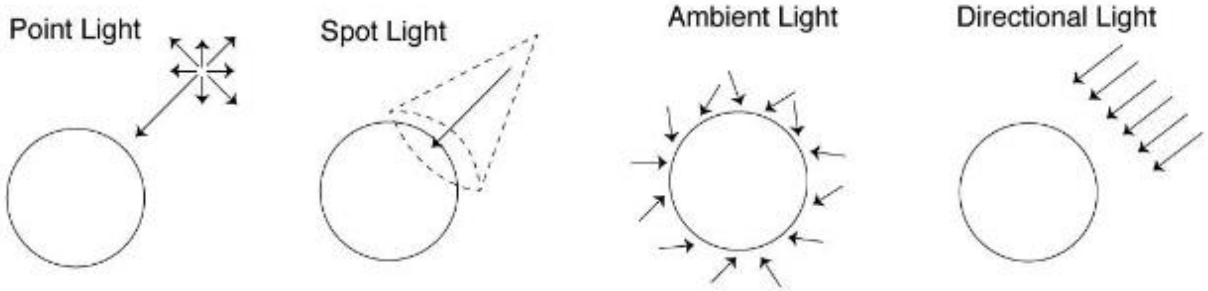


The program output, once you‘ve fixed the vertex shader & the uniform related errors. Before you fix these things, you‘ll see a single quad in the middle of the screen.

Let‘s Add A Spot Light

There is a little red cylinder on top of the ship. Its transform is called **shipHeadLight**. On top of this cylinder, we want to place a spotlight that shines forward (→ in front of the ship) – and a little bit downward. The fragment shader currently codes diffuse shading for a directional light + an ambient light. We want to keep these two light sources, and add the spot light as third light source.

Recap about light sources:



A spot light is basically a point light, meaning that **the light direction varies for each fragment**, while a directional light has a constant light direction, and an ambient light has no light direction at all.

Unlike a point light, a spot light only illuminates fragments within a certain light cone, the „spot cone“ (see graphic above). To find out if a fragment is within the spot cone, you need 2 more parameters:

- A „spot direction“ – this controls where the spotlight shines / how the cone is oriented
- A „spot angle“ – this controls how wide the cone is.

So in total you will need the following **four parameters** for the spot light: **color, position, direction, angle**. These values will be needed by the fragment shader, for the shading calculations, but you will also need them in the C++ code.

Let‘s Add A Spot Light

So we need 4 parameters: color, position, direction and angle.

Spotlight color & angle are typically constants. (Unless you maybe want to create a disco spotlight that changes color over time.) You can choose these values yourself, but make sure to choose meaningful values, i.e. don‘t make a black spotlight, or a spotlight with an angle of 0°, that‘s pointless. Do yourself a favor, and store the angle in radians instead of degrees.

Spotlight position & direction are not constants: as the ship moves around the island, both values change. Also, the position of the spotlight is linked to a geometric primitive.

@Spotlight Position:

We want to place the spotlight on top of the little cylinder called **shipHeadLight** (i.e. the cylinder that is rendered with this transform). The center of the cylinder geometry is the origin (0, 0, 0), and the top is (0, 0.5, 0) → so basically, we can get the position on top of the shipHeadLight cylinder by multiplying (0, 0.5, 0) with the **full model matrix** used to draw the cylinder.

Make sure to use the full transformation chain, including the matrices that contain the ships movements, otherwise the spotlight won‘t move with the ship.

@Spotlight Direction:

First, decide on a direction for the spotlight while the ship is standing still, e.g. something like (1, -0.5, 0) → this will make the spotlight shine slightly down. Then, apply the matrices that control the movement of the ship, to make the spotlight direction change in each frame.

Let‘s Add A Spot Light

Before actually coding the spotlight in the fragment shader, it‘s helpful to visualize it to make sure you have calculated the correct position / direction. We do this by drawing a **wireframe cone** at the spotlight position, oriented along the spotlight direction, and scaled so that it fits the spot angle.

Wireframe Mode

Activate before drawing the cone geometry via glPolygonMode(GL_FRONT_AND_BACK, GL_LINE) and switch back after drawing it via glPolygonMode(GL_FRONT_AND_BACK, GL_FILL).

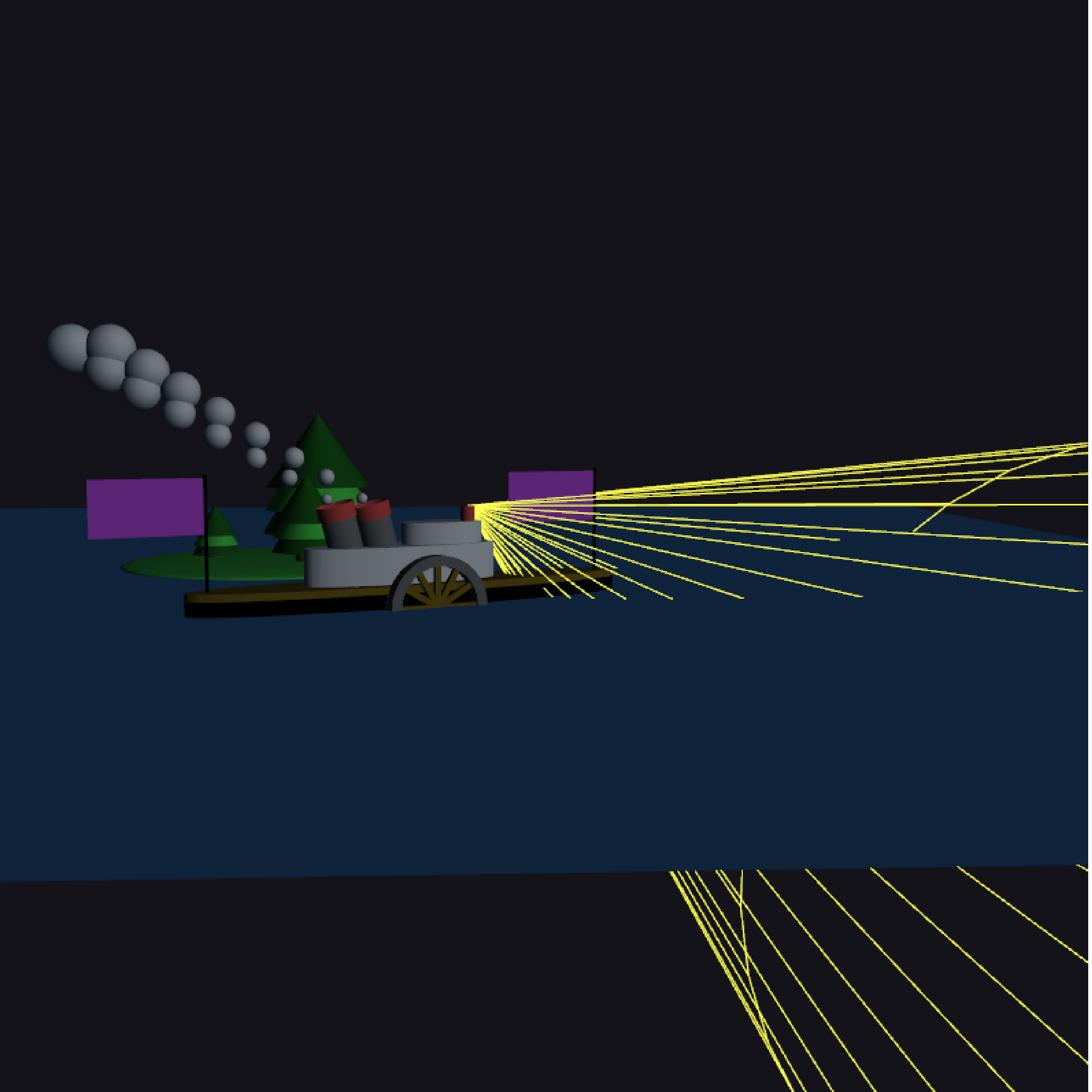
The cone object should not be shaded

It makes no sense to shade „debug geometry“ → add a second, super simple GlslProgram to your code. The vertex shader should just calculate gl_Position properly. The fragment shader should output a object-specific color. Load the vertex/fragment shader files in the App::init method, and use the new GlslProgram to render just the cone, with the spotlight color as the color of the geometry. **Once you have more than one GlslProgram in your code, be careful not to mix them up when setting uniforms.**

The cone model in model/object coordinates

You need to understand the cone geometry in oder to draw the „debug cone“ properly.

1. The cone has a height of 1, and its center is @the origin → the point at the top of the cone, which should be placed where the spotlight is, is (0, 0.5, 0).
2. The cone is oriented along the negative y-Axis → this axis should be aligned with the spotlight direction.
3. The cone model has a radius of 1. Since its height is also 1, this means that the cone angle is 45°. In general: tan(alpha) = radius / height. You can change the the radius by scaling the cone equally in X&Z → scale the cone so that the angle becomes equal to the spotlight angle.



At the end, your cone should look similar to this, and move with the ship. I used a spot angle of 30°, and a yellow spotlight color. I also upscaled the cone to make it bigger, so that it actually intersects with the water plane geometry.

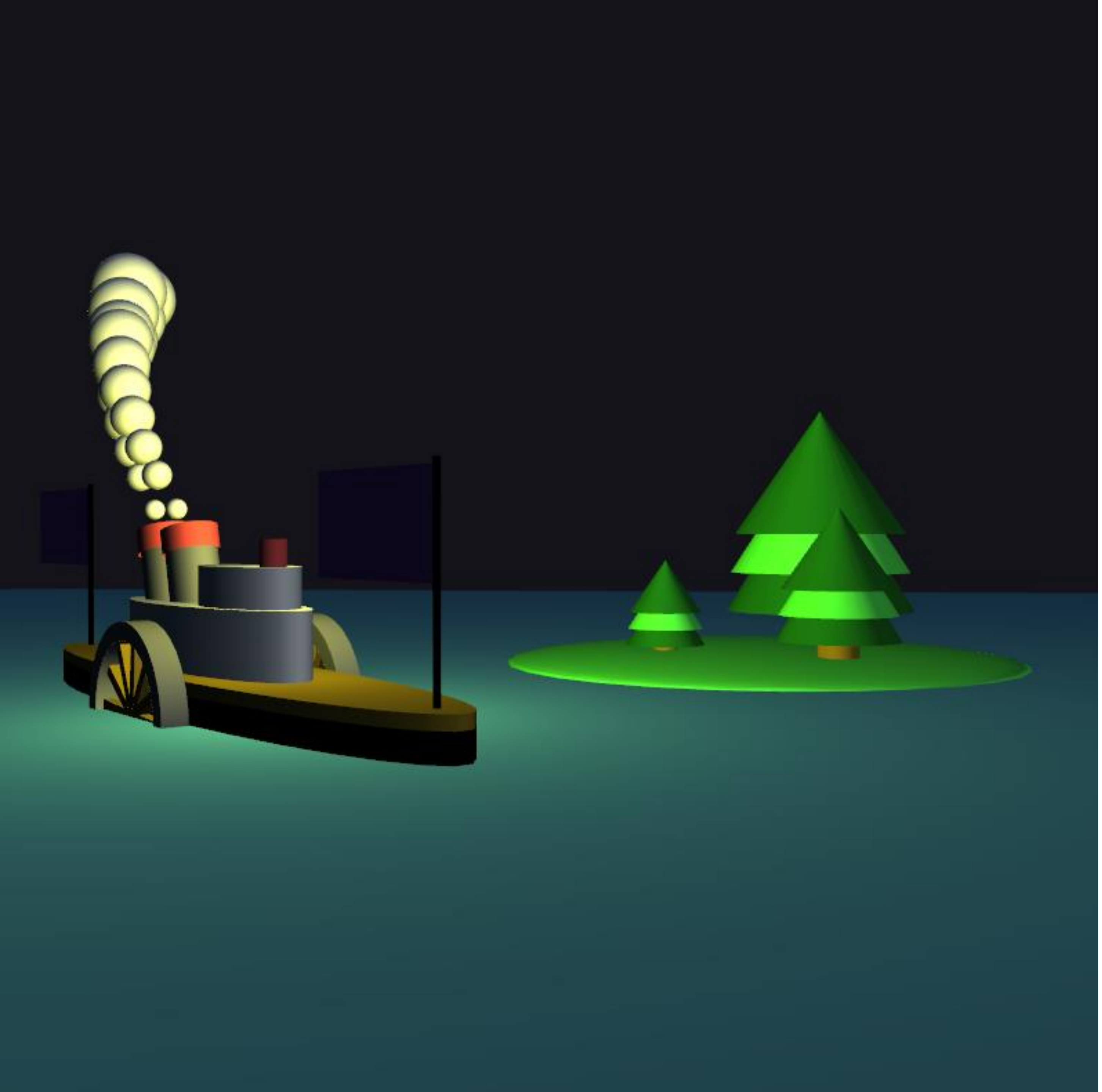
Tip for calculating the transformation matrices: **Before** you apply any other transformation to the cone, translate the cone so that the cone top (0, 0.5, 0) is moved into the origin. This can save you a lot of trouble. After all, you want to place the top where the spotlight is.

Let‘s Add The Spot Light In The Shader

At this point, the 4 spotlight parameters color, position, direction and angle are defined and we have a debug cone to show us where exactly the spotlight is, and which areas are illuminated.

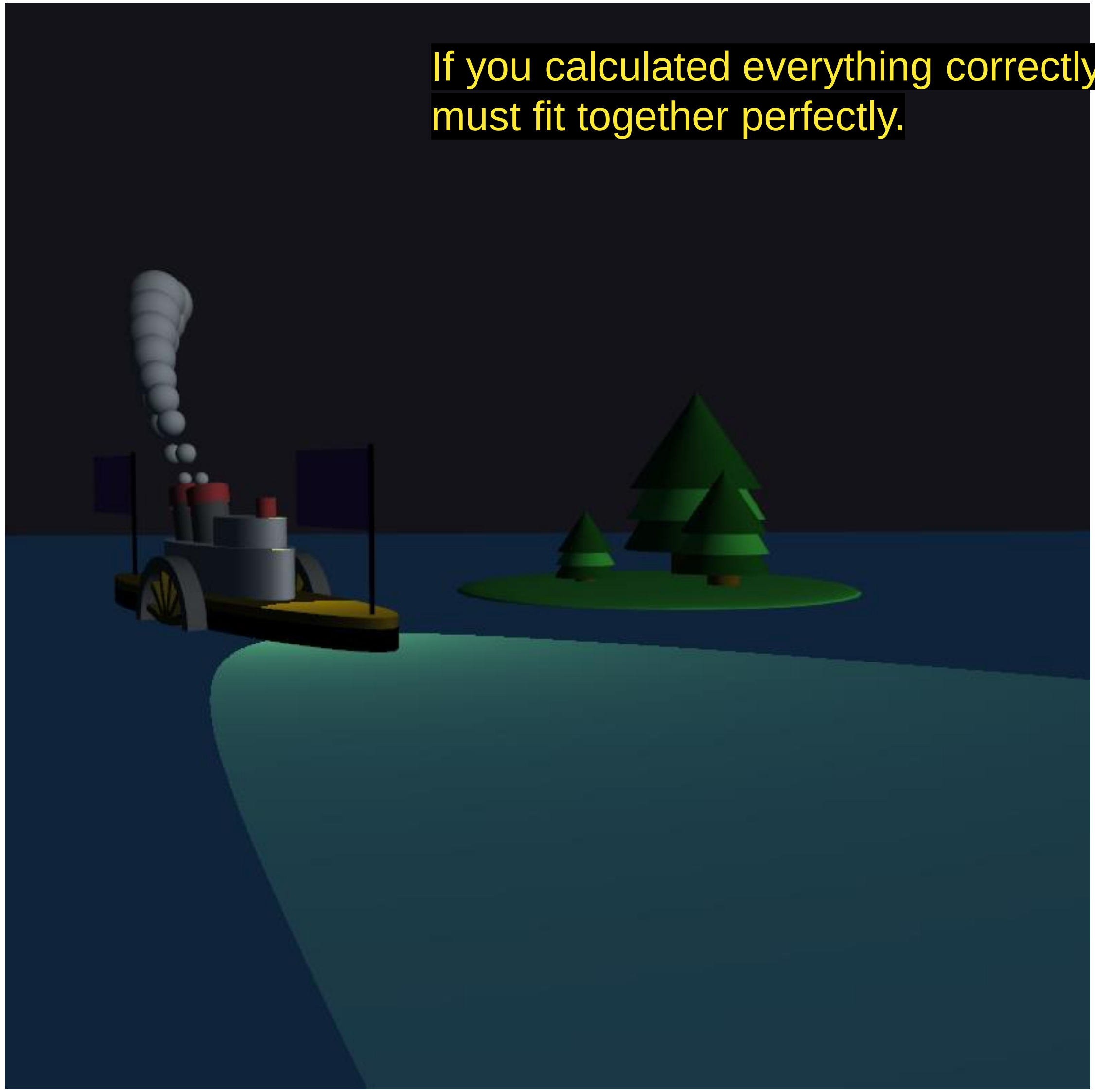
We can now add the spotlight to the fragment shader. In the end, if all calculations are correct, those pixels inside the yellow cone should receive light from the spotlight, and those outside should not receive any light

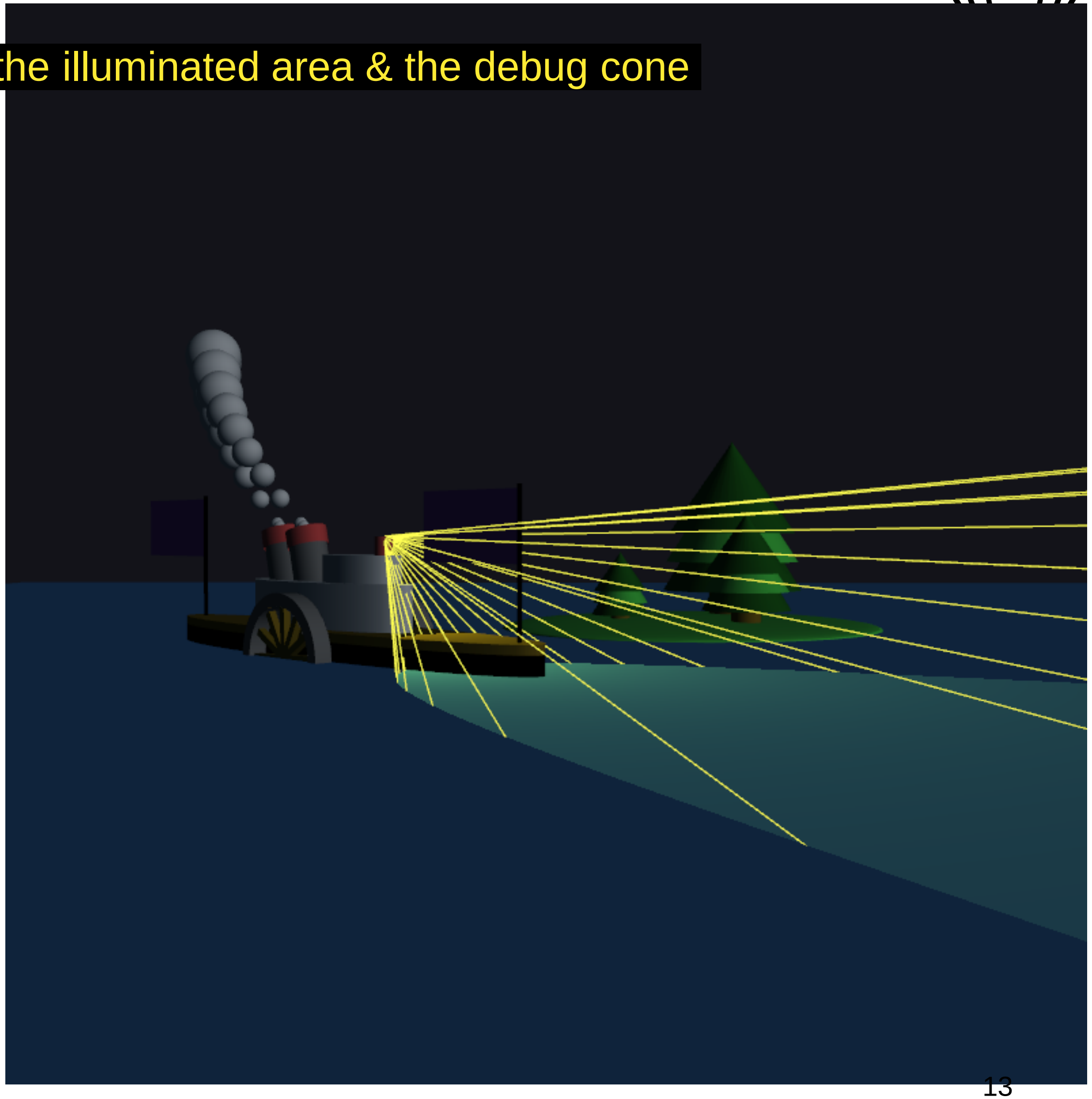
1. Pass color, position, direction and angle into the fragment shader. Make sure that you use the correct datatypes, and that you don‘t have any typos in the uniform names. **Check which setUniform methods are available in the GlslProgram class.**
2. The fragment shader already calculates the contribution of the ambient and the directional light, and the final fragment color is the sum of their contributions. Calculate the contribution of the spotlight using the diffuse shading formula, and just add it to the final fragment color.
3. Since with a spotlight, the light vector L varies for each fragment, you will need the **world space position of the fragment**. → You need to pass the world space position of each vertex to the fragment shader. Then, you can calculate L as normalize(lightPosition – worldPosition).
4. **In case you have trouble with the shading, remember that I already showed you how to calculate diffuse shading for a point light – and that there is samplecode on the wiki.**
5. Finally, you need to check **if the fragment is inside of the spotlight cone**. If it‘s outside, the spotlight contributes no illumination to the fragment. You can check this by calculating the angle between L and the spotlight direction, and by comparing the calculated value to the spotlight angle.



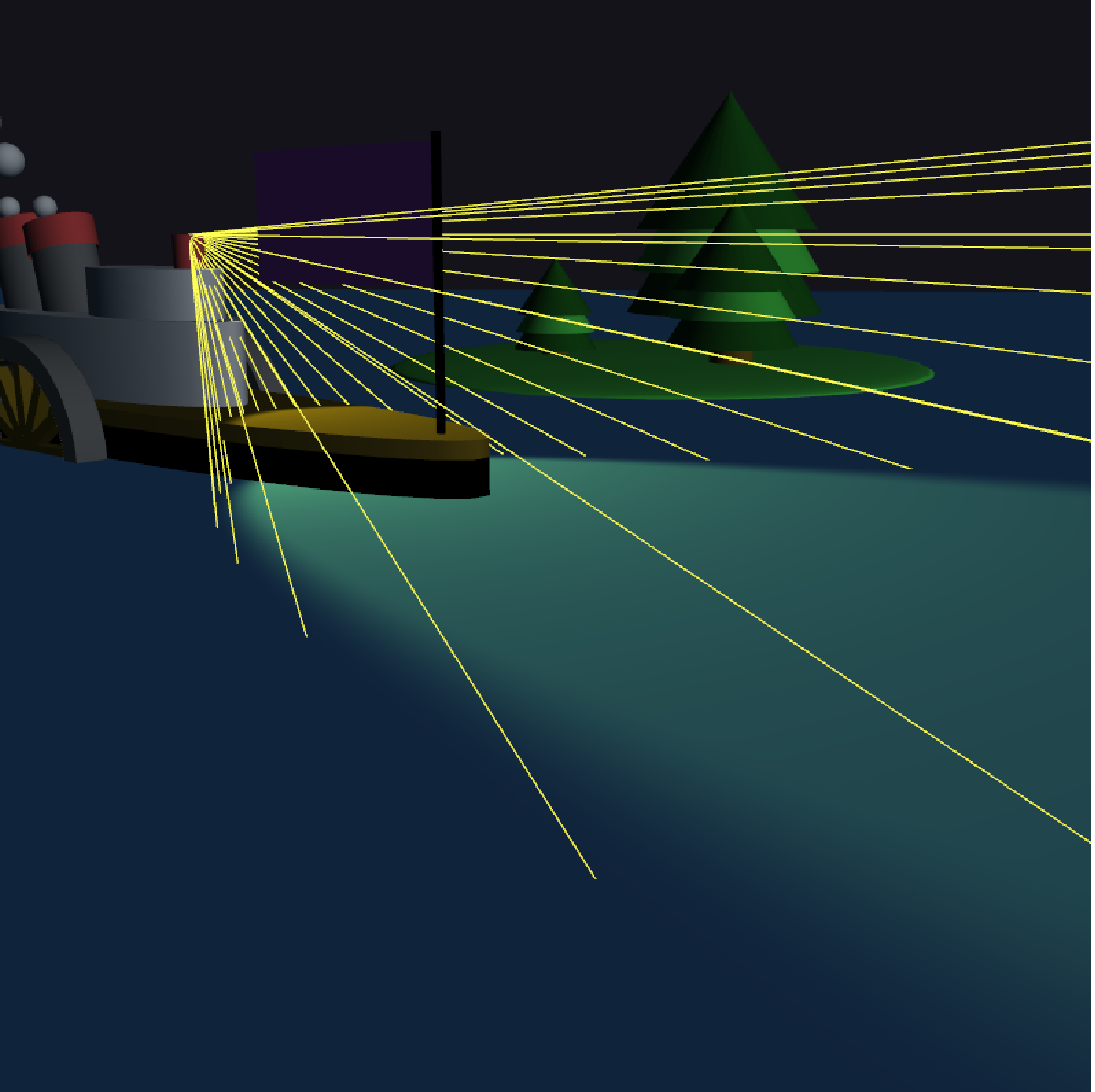
Without the „cone check“, the light source is basically just a point light that follows the ship.

Output, Without / With The Debug Cone





If you calculated everything correctly, the illuminated area & the debug cone must fit together perfectly.



Finally, we use the GLSL function smoothstep https://registry.khronos.org/OpenGL-Refpages/gl4/html/smoothstep.xhtml to create a smooth transition between lit & unlit areas – compare this image to previous slide, where the transition between lit & unlit areas is binary (→ ugly).

Image that there is a smaller cone located inside the outer spotlight light cone; with a cone angle that‘s 80° of the outer cone angle.

- Every fragment within the inner cone is fully lit, i.e it receives 100% of the spotlight‘s contribution.
- Every fragment outside of the outer spotlight cone is not lit at all.
- Fragments that fall within the outer cone, but not within the inner cone receive only part of the light – the spotlight contribution is **attenuated** using the smoothstep function.

Task B

Texture Mapping

Create A Separate GlslProgram For The Flags

There are currently 2 purple flags on the ship, one in front and one in the back. We‘re now going to add texture mapping to the flags.

The first step is to add another GlslProgram to the C++ code – on that is going to be used just for the flags. However, the shaders should of course also include the diffuse shading from Task A, **so make sure you only start Task B once Task A is completely done** – this way, you can just make a copy of the vertex and fragment shader files, rename them and load them into a new GLSLProgram ☺.

You may now wonder if this is really the best way of doing things, and if maybe it would be better to just have a single shader, where we decide via a boolean uniform variable if we want to use textures or not. In general, so-called „uber-shaders“ that try to do everything are considered a bad idea, and your going to learn why that is so in CG2. A single uniform boolean to decide about textures would be ok, but in Task C, you‘re going to modify the shaders even more, so it‘s better to just create a new GlslProgram now.

Once you have the new GlslProgram, make sure you set all the necessary uniforms – currently the same ones as the regular diffuse shader uses – and draw the 2 flags using that shader. As long as no textures are being used, the results should look exactly the same as before.

Add Texture To The Flags

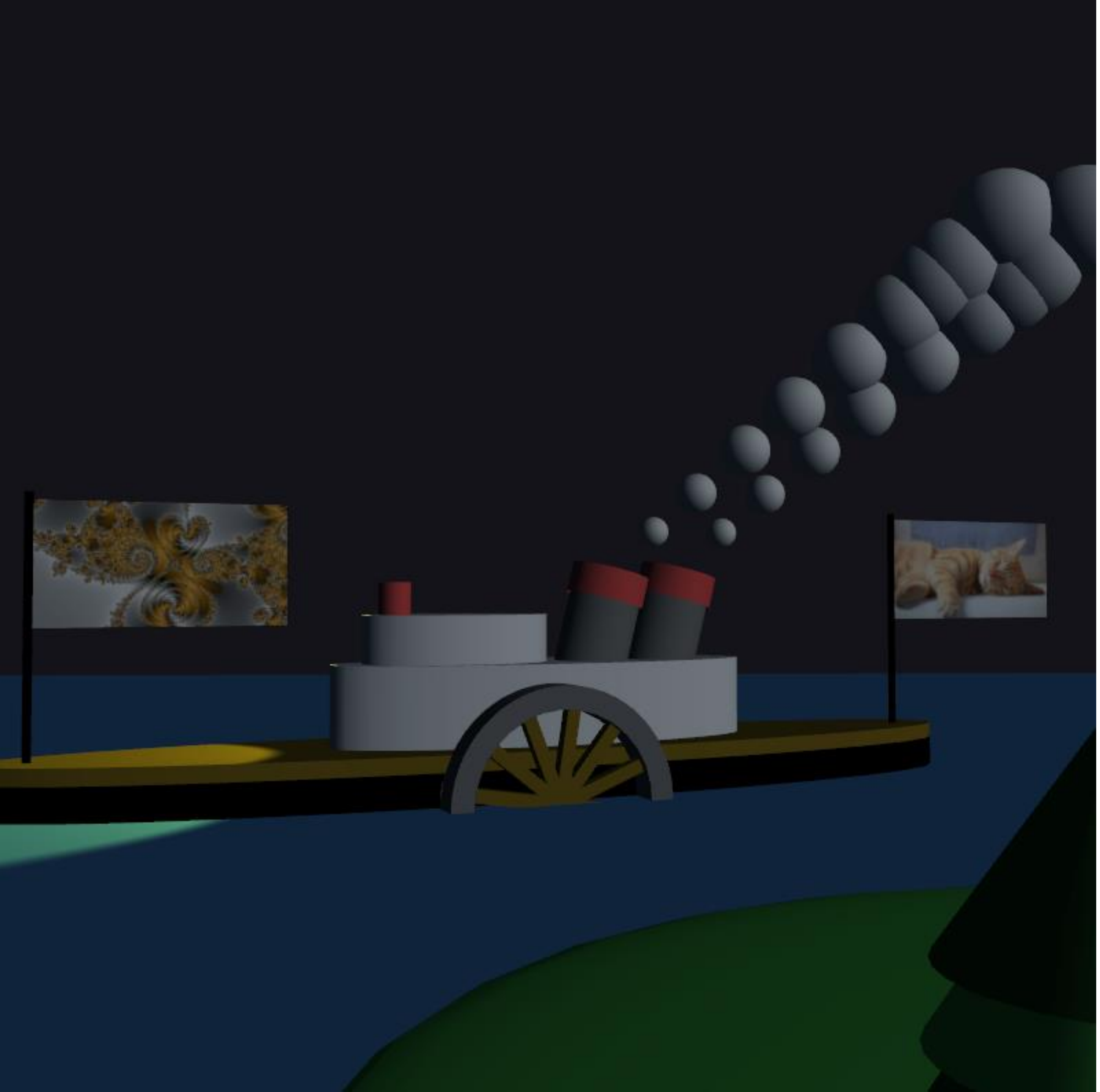
Look for 2 different images that you can uses as textures for the flags. Use simple image formats (png, bmp or jpg) Since the flags are small, there is no point in using high-res textures - you‘re just going to end up with a very bad case of texture minification otherwise ☺. Use linear or trilinear filtering (+ mipmaps) to make sure the texture looks nice.

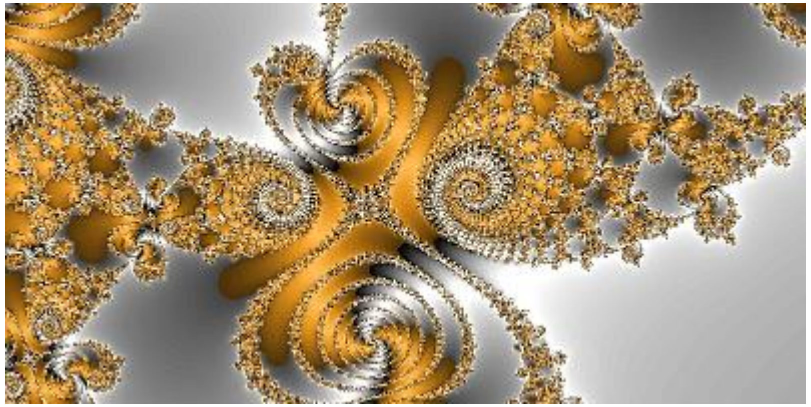
I advise you to use texture where the width is a multiple of 4 – maybe resize your texture to ensure this – to avoid optical errors.

Again, there is sample code available (on the slides and also as executable code) that shows you how to load the images using STBI, how to upload them to the GPU, how to configure the filters and how to use them in a shader. Create 2 texture objects, and make them available when rendering the flags (i.e. bind texture 1 when rendering the first flag, and texture 2 when rendering the second flag).

In the shader, declare the texture as sampler2D, and obtain the diffuse color by reading the texture. (→ you don‘t need kD any more, it‘s now the color value from the texture). You‘ll need to pass the texture coordinates from the vertex to the fragment shader – **remember that the uv coordinates are bound to attribute location 4 in the vertex shader.**

The modifications for the shader are pretty simple in this task, it‘s about loading & using 2 different textures.





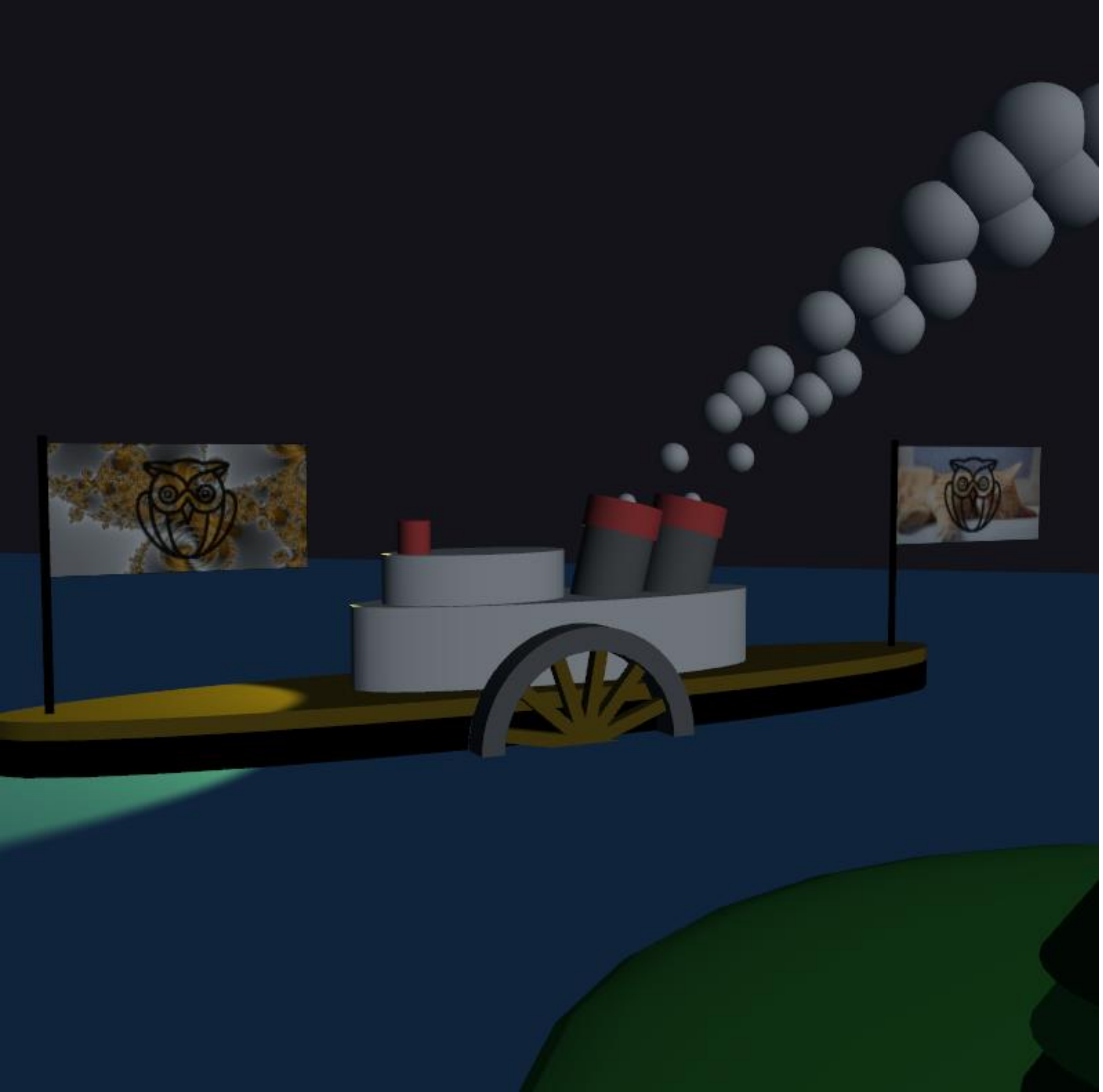
If you can‘t see the textures because the scene is overall rather dark, you can try turning up the directional light little.
Btw., the front flag does not receive light from the the spotlight. This is because the incoming light from ths spot light is nearly parallel to the falg surface.

Add Multitexturing

Currently, we use two different textures – but not at the same time: We need one texture for the first flag, and the other one for the second flag, but they should each be bound to the same **texture unit** when you use them.

Now, look for a **black & white** texture – you can use the FH Salzburg Owl – and make this available in the flag shader as well. Multiply the color of the new texture with the other texture color.

Now, you can no longer use a single texture. You have to bind two textures at the same time → you have to pay attention to the texture units.



Feel free to use a different texture if you don‘t like the owl ;-)

Task C

Vertex Displacement

Vertex Displacement

We‘re now going to add displacement in the vertex shader to the flags, to make it look as if they‘re waving in the wind.

We discussed how to do it before christmas –**there is (again) sample code available on the wiki** / on the slides.

In class, we used a displacement in both X and Y, to create fake water. Now, we‘re just going to displace the vertices in X. Remember:

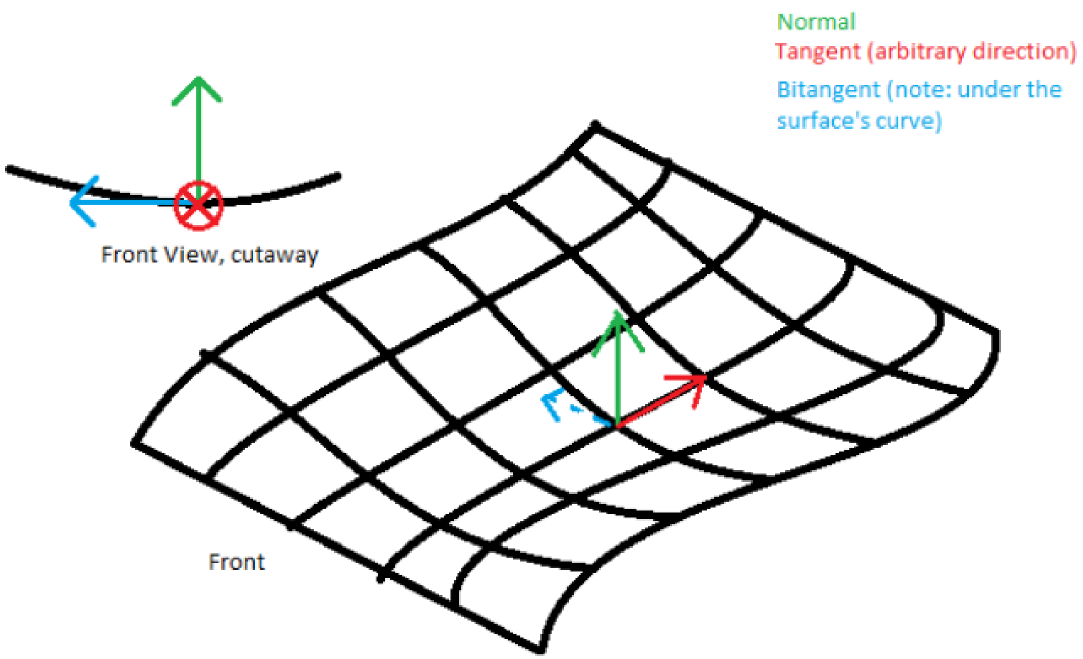
$$displacement = amplitude * cos(phase + frequency * pos.x)$$

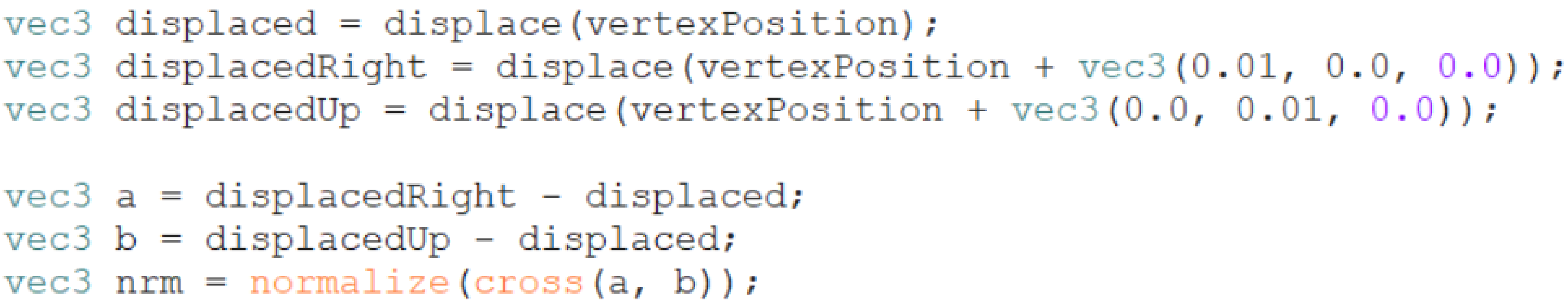
$$pos = pos + displacement * normal$$

Where phase is the elapsed time in seconds (→ pass to shader as uniform), and frequency and amplitude are values you can choose. The displacement is then added to each vertex in direction of the vertex normal. **All of this is done in object space, i.e. before any matrices are applied to the vertex**.

For the amplitude, a very small value (something between 0.05 and 0.1) is ok. For the frequency, a low value around 10 should work.

Remember to Recalculate the Displaced Normals

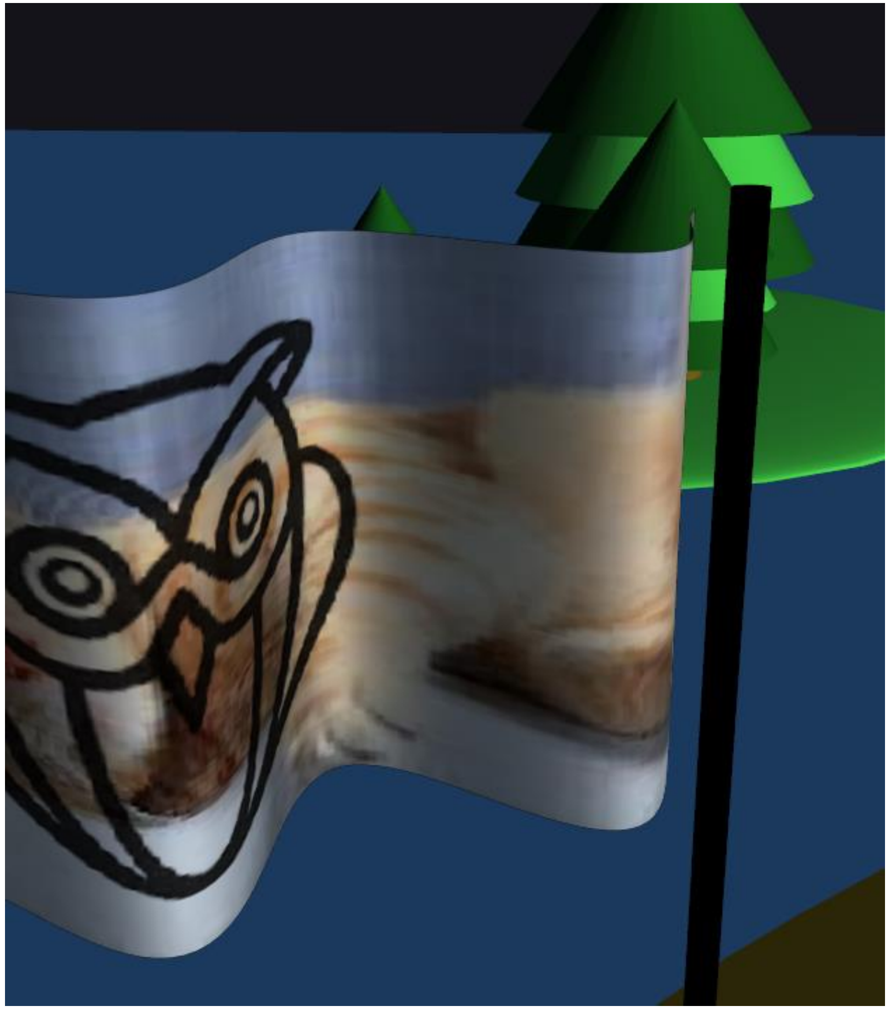

Normal
Tangent (arbitrary direction)
Bitangent (note: under the surface's curve)
Front View, cutaway
Front



```
vec3 displaced = displace(vertexPosition);
vec3 displacedRight = displace(vertexPosition + vec3(0.01, 0.0, 0.0));
vec3 displacedUp = displace(vertexPosition + vec3(0.0, 0.01, 0.0));

vec3 a = displacedRight - displaced;
vec3 b = displacedUp - displaced;
vec3 nrm = normalize(cross(a, b));
```

What‘s Happening Here?

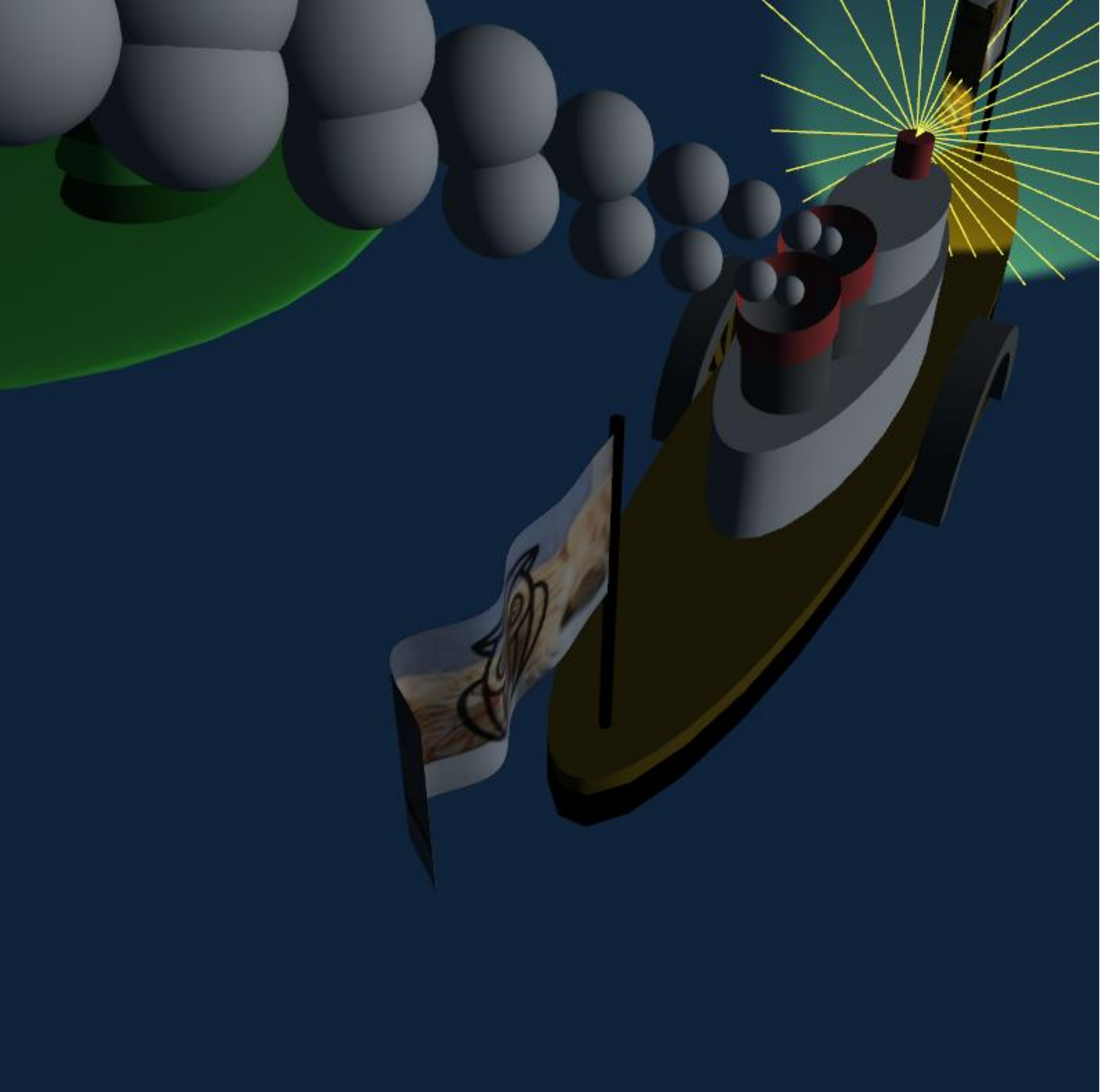


(I turned up the light a bit to show this) The flag is floating freely in the air and is no longer connected to the flagpole. This is a result of the amplitude we use to calculate the displacement, which is the same for every vertex in the flag geometry.

Here is what you really need to do: The closer the flag vertices are to the flagpole, the more you want to scale the amplitude down. (Since the amplitude directly controls the magnitude of the displacement!) The vertices that are directly connected to the flagpole should not get displaced at all, i.e. the amplitude should be 0 where the flag touches the flagpole.

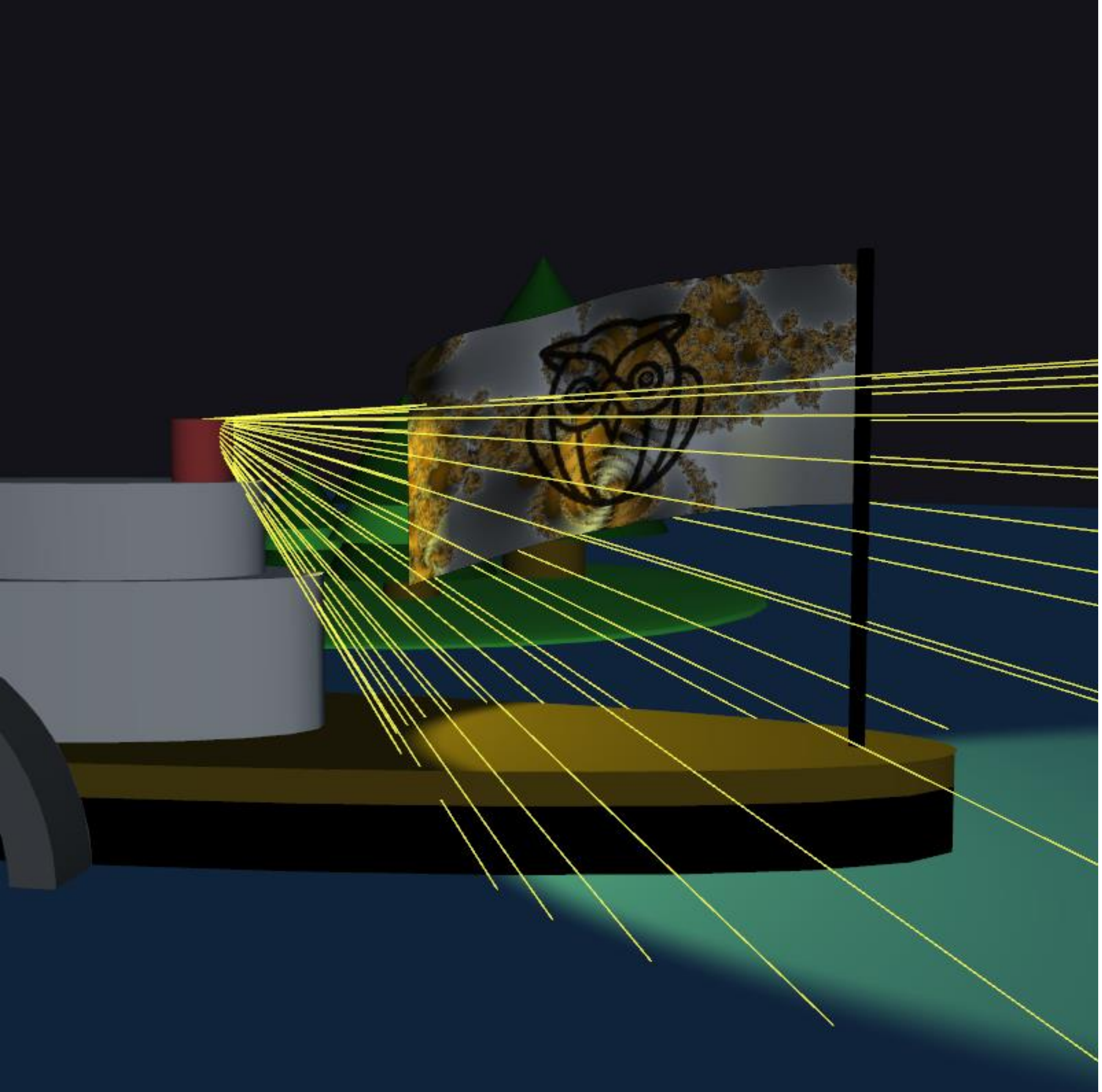
We can achieve this, since we know how the flag geometry is defined: it‘s a plane, with the X-coordinate in range [-1, 1]. The right-most vertices of the plane – where x == 1 – are connected to the flagpole. So basically, you have to remap the range of possible X-coordinates [-1, 1] to scaling factors in range [1, 0], and use these to scale the amplitude. That way, you will get no displacement where the flag touches the flagpole, and full displacement at the other side of the flag.

You can also try offsetting vertices down the Y-Axis just a little bit based on the same scaling factor – this can make it look as if gravity was affecting the flag.

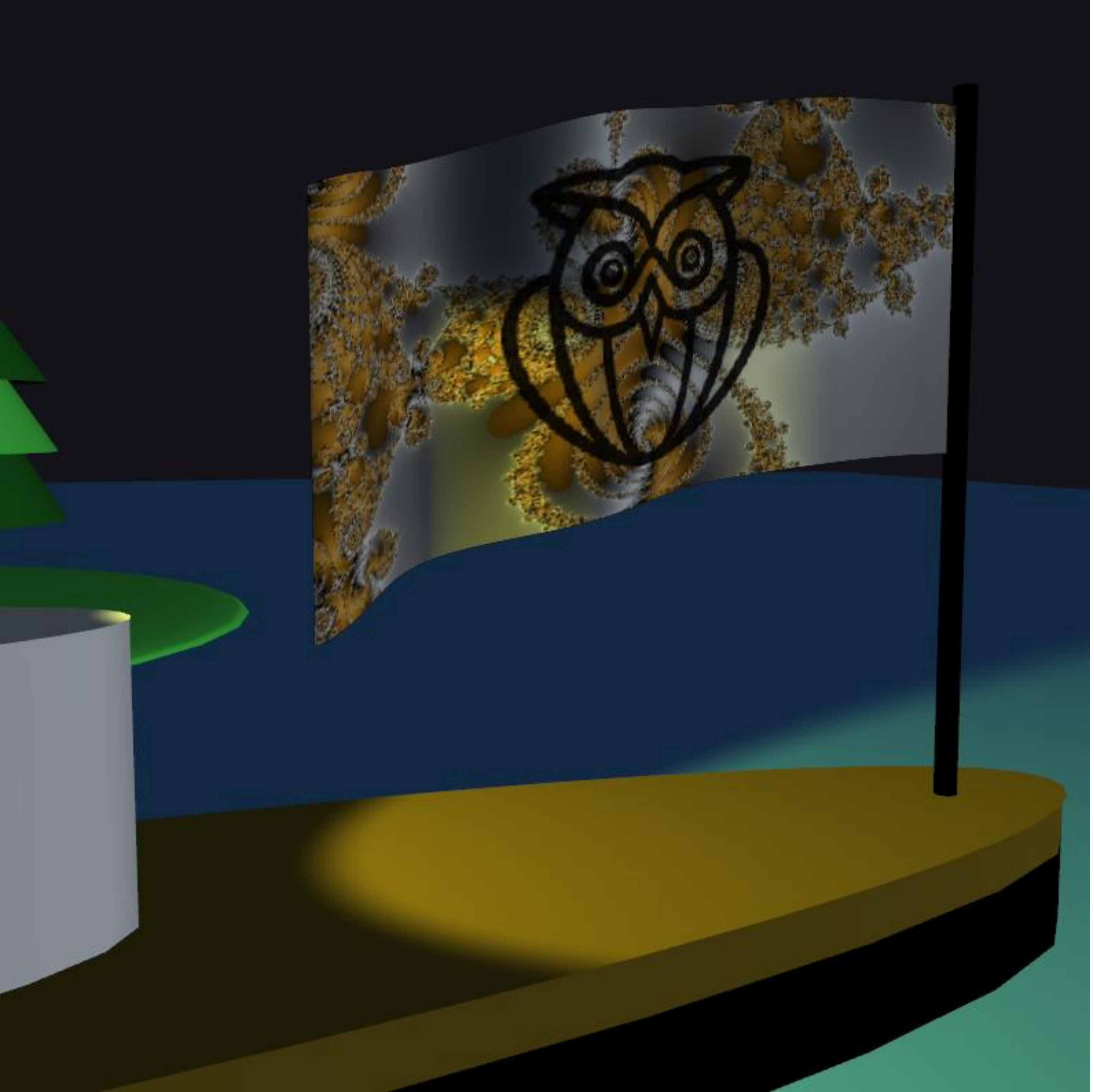


Here, you can see that there is no more displacement right where the flag touches the flagpole.

On the opposite side of the flag, the displacement factor is at it‘s maximum, 1.



And here you can see that the flags are no longer just quads, but that the seem to be affected by gravity a little bit. This is the result of offsetting the vertices down the Y-Axis just a little bit when they‘re far away from the fagpole.



Btw, part of the front flag now receives light from the spot light – due to the displacement, it‘s no longer parallel to the incoming light.