Labb 3:

Vi börjar med vår väderkvarn!
De tre första delarna av väderkvarnen (väggen, stegen och taket) använder alla samma translationsmatris för att placeras korrekt med varandra. Dessa ritas ut genom att iterera över en array på detta vis:

```
// We draw the windmill.
trans = T(0, 0, 0);
glUniformMatrix4fv(glGetUniformLocation(program, "tranMatrix"), 1, GL_TRUE, trans.m);
for (int i = 0; i < modelno-1; i++) {
	DrawModel(windmill[i], program, "in_Position", "in_Normal", "inTexCoord");
}
```

Sedan ska bladen placeras och roteras kring den korrekta axeln. Det fick bli lite gissningslek för att heta en translation som såg bra ut, men efter att den funnits så roteras de med en och samma tidsvariabel. De placeras med en offset på $\pi/2$ * j rad, där j ökas för varje blad:

```
//After that, we place the blades.
trans = T(4.5, 9.25, 0);
for (int j = 0; j < 4; j++) {
	rot = Rx((t / 1000) + (j*3.1415/2));
	glUniformMatrix4fv(
		glGetUniformLocation(program, "tranMatrix"),
		1,
		GL_TRUE,
		Mult(trans, rot).m
	);
	DrawModel(windmill[modelno - 1], program, "in_Position", "in_Normal",
"inTexCoord");
}
```

Sedan tittar vi på styrningen av kameran, och detta görs i funktionen getCameraCoord() i lab3-2, och senare i updateCameraCoord() i lab3-4
Vi måste börja med att skapa två variabler för att bestämma hur mycket användaren kan röra sig / vända sig:

```
GLfloat turnValue = frameFreq / (10000 * pi), moveValue = frameFreq / 2000.0;
```

Framefreq är då uppdateringsfrekvensen (fps) för programmet.
Sedan har vi två exempel på hur input hanteras och konverteras till rörelse:

```
if (glutKeyIsDown('q')) {
	camDir = MultVec3(Ry(turnValue), camDir);
}

(…)

if (glutKeyIsDown('w')) {
	camPos = VectorAdd(camPos, ScalarMult(camDir, moveValue));
}
```

Den ena representerar då en rotation, medan den andra representerar en translation.
I slutet av funktionen så genereras kameramatrisen utifrån input från användaren:

```
// Then we create and return the camera coordinates.
vec3 focusPoint = VectorAdd(camDir, camPos);
mat4 camera = lookAt(
	camPos.x, camPos.y,  camPos.z,
	focusPoint.x, focusPoint.y, focusPoint.z,
	0, 1, 0
);
```

Vi använder funktionen lookAt() rakt av, och tar fram kamerapositionen och fokuspunkten sedan innan.

Sedan implementerar vi en skybox, genom att låta en texturerad låda ”följa med” kameran men inte rotera. Detta implementeras lite bakvänt, genom att lådan roteras med kameran men inte roteras. Detta är för att kameraobjektet inte är vad som förflyttas, utan världen i relation till den (likt en forntida uppfattning kring himlavalvet).
Matrisomvandlingarna som görs visas här nedan:

```
// We reverse the camera matrix
trans = InvertMat4(getCameraCoord());

// Then we let the skybox translate with the camera instead, but keep the rotation
trans = MultMat4(trans, MultMat4(getCameraCoord(), T(camPos.x, camPos.y -1.0,
camPos.z)));
glUniformMatrix4fv(glGetUniformLocation(noShaProgram, "tranMatrix"), 1, GL_TRUE,
trans.m);
```

En viktig sak att komma ihåg med ens skybox är att den både måste ritas ”bakom” världen, samt ej använda någon skuggning. För att uppnå detta så ritar vi ut skyboxen först av allt (kan också ritas ut mot slutet, men att den ignorerar där det redan finns värden i fragmentet); vi använder också en separat fragment shader just för skyboxen, som bara skickar vidare texturen:

```
void main(void)
{
	out_Color = texture(skyTex, texCoord);
}
```

Till sist ska vi visa hur vi implementerat Phong Shading. Här kikade vi lite på LearnOpenGL’s guide[1] samt i boken. På klientsidan så sker utritningen inte

[1] https://learnopengl.com/Lighting/Basic-Lighting?fbclid=IwAR3yLzuBPLBlwkM9zgRqLEYTbsfkdHkxXVDKk3gLI8V1_fHGK4kQ6g5wnCM

så annorlunda ut (förutom den kod som givits av labben). Den enda skillnaden är att man skickar in ett reflektionsvärde per objekt:

```
trans = T(10, 0.5, 3);
glUniformMatrix4fv(glGetUniformLocation(program, "tranMatrix"), 1, GL_TRUE, trans.m);
glUniform1f(glGetUniformLocation(program, "specularExponent"), specularExponent[1]);
DrawModel(bunny, program, "in_Position", "in_Normal", "inTexCoord");
```

Den stora skillnaden finnes istället i våra shaders.
Först så tar vertex shader fram fragmentets position och skickar denna till fragment shader:

```
fragPosition = vec3(tranMatrix * vec4(in_Position, 1.0));  //The fragment's position in
world space
```

Sedan i fragment shader så går fragmentet igenom en del steg per ljuskälla. Det första steget är att kolla om ljuskällan är placerad i scenen eller inte, och korrigerar ljusets riktning därefter:

```
// We check if the light is directional or not, and calculate the vector accordingly.
if(isDirectional[i]) {
	lightDir = normalize(lightSourcesDirPosArr[i]);
}
else {
	lightDir = normalize(lightSourcesDirPosArr[i] - fragPosition);
}
```

Sedan tar vi fram reflektionen. För detta behöver vi vinkeln från kameran samt reflektionsvinkeln från ljuskällan. Dessa beräknas då innan:

```
// We calculate the vectors needed for the specular light (view- and reflection
direction).
viewDir    = normalize(camPos - fragPosition);
reflectDir = reflect(-lightDir, norm); // LightDir is inverted as the vector must point
away from the surface.
// After that, we calculate the specular component.
spec     = pow(max(dot(viewDir, reflectDir), 0.0), specularExponent);
specular = specularStrength * spec * lightSourcesColorArr[i];
```

Därefter tar vi fram den diffusa delen av ljuset, d.v.s. den delen av ljuset som är oberoende av betraktningsvinkel:

```
// Then we calculate the diffuse light, which is the dot product between the previous
vector and the normal.
// As we don't want any negative value, we have 0.0 as the lowest possible value.
diff = max(dot(norm, lightDir), 0.0);
diffuse = diff * lightSourcesColorArr[i];
amb = ambStr * lightSourcesColorArr[i];
```

Till sist så slås dessa färger ihop. När detta är gjort för alla ljuskällor så multipliceras det med texturen, så har vi värdet på vårt fragment:

```
// Lastly, we combine the ambient, diffuse and specular light and add it to the result.
result = result + vec4((diffuse + amb + specular), 1.0);

(…)

// Before we're done, we multiply the light with the texture.
vec4 tex  = texture(texUnit, texCoord);
out_Color = result * tex;
```

Labb 4:

Normalen räknas ut genom att använda de omkringliggande ytorna för att ta fram två lutningar (vi tittar på lokala höjdskillnaden i x- och z-led):

```
// https://stackoverflow.com/questions/49640250/calculate-normals-from-heightmap
// vec3 normal = vec3(2*(R-L), 2*(B-T), -4).Normalize();
uint16_t L = x > 0 ?
             tex->imageData[( (x-1) + z * tex->width) * (tex->bpp/8)] : 0;
uint16_t R = x < tex->width - 1 ?
             tex->imageData[( (x+1) + z * tex->width) * (tex->bpp/8)] : 0;
uint16_t B = z > 0 ?
             tex->imageData[(x + (z-1) * tex->width) * (tex->bpp/8)] : 0;
uint16_t T = z < tex->height - 1 ?
             tex->imageData[(x + (z+1) * tex->width) * (tex->bpp/8)] : 0;

float sl = L / 100.0;
float sr = R / 100.0;
float sb = B / 100.0;
float st = T / 100.0;

vec3 comb = { 2.*(sr-sl), 4 ,2. * (sb-st) };
vec3 normal = Normalize(comb);
```

Alltså beräknar normalerna samt vertex-positionerna utifrån vår givna heightmap. If-satserna ser också till att kanterna på bilden inte ställer till med problem.

För att sedan ta fram höjden av en given punkt på landskapet så använder vi Barycentriska koordinater med linjär interpolering där emellan. Vi tar alltså fram tre skalärer (l1, l2, l3) som multipliceras med y-värdena från triangelns hörn (p1, p2, p3). Summan av detta blir till sist höjden vi söker. (ox och oz är den offset som råder från det nedre vänstra hörnet av vår kvadrant).

```
float l1 = lambda1(p1, p2, p3, ox, oz);
float l2 = lambda2(p1, p2, p3, ox, oz);
float l3 = 1 - l1 - l2;
float intz = l1*p1.y + l2*p2.y + l3*p3.y;
```

Dessa är funktionerna som används för att ta fram l1 och l2:

```
//
https://stackoverflow.com/questions/36090269/finding-height-of-point-on-height-map-triang
les
float lambda1(vec3 p1, vec3 p2, vec3 p3, float x, float z) {
	float num = (p2.z - p3.z)*(x - p3.x) + (p3.x - p2.x) * (z-p3.z);
	float den = (p2.z - p3.z)*(p1.x - p3.x) + (p3.x - p2.x)*(p1.z-p3.z);
	return num / den;
}

float lambda2(vec3 p1, vec3 p2, vec3 p3, float x, float z) {
	float num = (p3.z - p1.z)*(x - p3.x) + (p1.x - p3.x)*(z-p3.z);
```

```
	float den = (p2.z - p3.z)*(p1.x - p3.x) + (p3.x - p2.x)*(p1.z-p3.z);
	return num / den;
}
```

Multitextureringen i programmet är ganska simpel, och är beroende på fragmentets höjd i y-led:

```
float a = height / (maxheight*2.);
float dist =  1. / length(lightPos - FragPos);
outColor = vec4(diffuse, 1.0) *
           mix(texture(tex, texCoord), texture(snowTex, texCoord), a) *
           vec4(vec3(dist)*2.0, 1.0);
```

Som utökning av programmet valde vi att rendera många objekt (en skog) som anpassar sig efter terrängens lutning. Vi återanvänder samma modell för alla träd, men representerar vardera träd med en transformationsmatris som sparas i arrayen positions[]. Det som syns nedan är då först variablerna för skogen, därefter är det uträkningen av deras transformationsmatriser och till sist är det utritningen av dem:

```
const int amt = 50000;

(…)

GLfloat positions[amt*16];

(…)

for(i = 0; i < amt; i++) {
	int x = (rand() / (float)RAND_MAX) * (float)ttex.width;
	int z = (rand() / (float)RAND_MAX) * (float)ttex.height;
	float height = getValue(&ttex, x, z, heightScale);

	float px = (float) x / xScale;
	float py = height;
	float pz = (float) z / zScale;


//https://stackoverflow.com/questions/15101103/euler-angles-between-two-3d-vectors
	float nx = tm->normalArray[(x + z * ttex.width)*3 + 0];
	float ny = tm->normalArray[(x + z * ttex.width)*3 + 1];
	float nz = tm->normalArray[(x + z * ttex.width)*3 + 2];

	vec3 normal = { nx, ny, nz };
	vec3 upVector = {0, -1, 0};
	vec3 axis = CrossProduct(upVector, normal);
	float lm = sqrt(pow(nx, 2) + pow(ny, 2) + pow(nz, 2));
	float angle = acosf(DotProduct(normal, upVector) / lm);
	mat4 m = Mult(T(px, py, pz), ArbRotate(axis, angle));


	for(j = 0; j < 16; j++) {
		positions[i*16 + j] = m.m[j];
	}
}
```

```
(…)

mat4 treeMdl = IdentityMatrix();
glUniformMatrix4fv(
	glGetUniformLocation(sphereProgram, "mdlMatrix"),
	1,
	GL_TRUE,
	treeMdl.m
);
glUseProgram(treeProgram);
glUniform1f(glGetUniformLocation(sphereProgram, "iTime"), t/1000);
glUniformMatrix4fv(
	glGetUniformLocation(treeProgram, "viewMatrix"),
	1,
	GL_TRUE,
	camMatrix.m
);
glBindVertexArray(vao);
glDrawElementsInstanced(GL_TRIANGLES, tree->numIndices, GL_UNSIGNED_INT, 0L, amt);
```

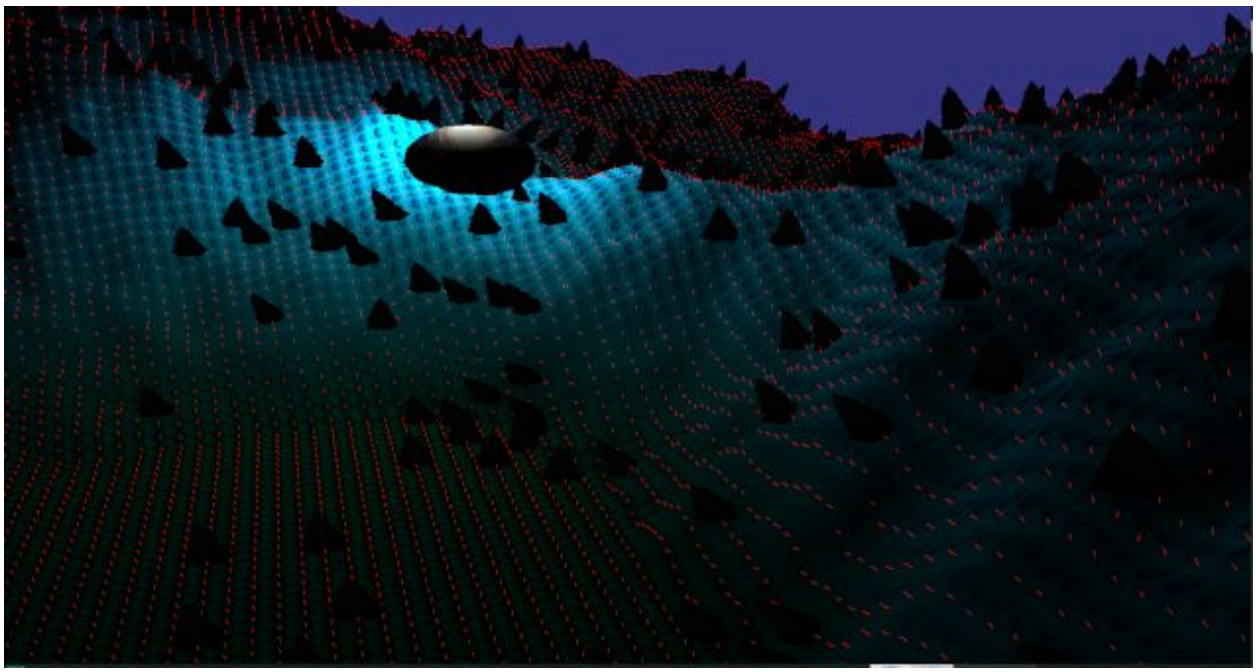

Bild på terräng med träd som antar terrängens lutning, och linjer för att debugga terrängens normaler.