I15
Informatics
Technical University of Munich

Eexam
Place student sticker here

Note:

- During the attendance check a sticker containing a unique code will be put on this exam.
- This code contains a unique number that associates this exam with your registration number.
- This number is printed both next to the code and to the signature field in the attendance check list.

Practical: Realtime Computer Graphics

Exam:	IN0039 / Mock Exam	**Date:**	XX XX0th , 2023
Examiner:	Westermann	**Time:**	XX00:00 – XX00:00

	P 1	P 2	P 3
I			

Working instructions

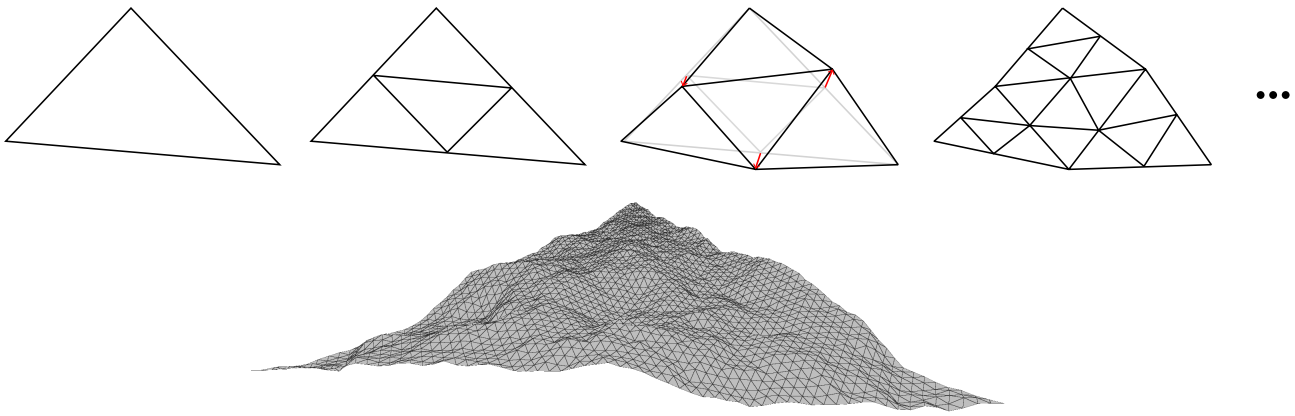
- This exam consists of **12 pages** with a total of **3 problems**.
  Please make sure now that you received a complete copy of the exam.
- The total amount of achievable credits in this exam is 46 credits.
- Detaching pages from the exam is prohibited.
- Allowed resources:
  - one **analog dictionary** English ↔ native language
- Subproblems marked by * can be solved without results of previous subproblems.
- **Answers are only accepted if the solution approach is documented.** Give a reason for each answer unless explicitly stated otherwise in the respective subproblem.
- Do not write with red or green colors nor use pencils.
- Physically turn off all electronic devices, put them into your bag and close the bag.

Left room from _____ to _____ / Early submission at _____

Problem 1 Geometric Modelling (11 credits)

In this assignment, your task is to write a CPU program that recursively subdivides and displaces a triangular geometry to generate a terrain. The algorithm consists of the following steps.

1. Add a base triangle to the triangle list at level 0. Set $\ell = 0$.
2. Iterate over all edges of triangles at level ℓ and subdivide the edges by adding a new vertex in the middle of each edge.
3. Displace the new vertices by a value drawn from the distribution $\mathcal{N}(0, \frac{\sigma^2}{2^\ell})$.
4. Triangulate the old and new vertices as seen in the image below to form the new triangles for level $\ell + 1$. Set $\ell = \ell + 1$ and stop once a given number of iterations is reached, or go to 2 otherwise.



You are allowed to use Java code or Java-style pseudo-code. In case you use pseudo-code, you may deviate from the Java syntax, but your code still needs to be understandable for an exam corrector familiar with Java or C++. You can use the function `rand.nextGaussian()` to draw a floating point value from $\mathcal{N}(0, 1)$. The following classes are given for vector operations and storing edge data. The class `TriTerrain` has some not yet implemented functions.

```
class vec3 {
    public float x, y, z;
    public vec3() { x = 0.0f; y = 0.0f; z = 0.0f; }
    public vec3(float _x, float _y, float _z) { x = _x; y = _y; z = _z; }
    public void add(vec3 other) { x += other.x; y += other.y; z += other.z; }
    public void mul(float scalar) { x *= scalar; y *= scalar; z *= scalar; }
}

class Edge {
    public Edge(int _vi0, int _vi1) { this.vi0 = _vi0; this.vi1 = _vi1; }
    public int vi0, vi1; // edge vertex indices -> array 'vertices'
    public int ne0, ne1; // new edge indices after subdivision -> array 'edges'
};

class TriTerrain {
    private final float variance; // sigma^2
    private Random rand;
    ArrayList<vec3> vertices;
    ArrayList<Edge> edges;
    ArrayList<Integer> triEIs; // triangle edge indices - three entries form a triangle

    public TriTerrain(vec3 v0, vec3 v1, vec3 v2, float variance) {
        this.variance = variance; // sigma^2
        rand = new Random();
        vertices = new ArrayList<vec3>();
        edges = new ArrayList<Edge>();
        triEIs = new ArrayList<Integer>();

        vertices.add(v0); vertices.add(v1); vertices.add(v2);
```

```
        // Pass indices into 'vertices'.
        edges.add(new Edge(0, 1));
        edges.add(new Edge(1, 2));
        edges.add(new Edge(2, 0));

        // Store indices into 'edges'. Three consecutive values belong to one triangle.´
        triEIs.add(0); triEIs.add(1); triEIs.add(2);
    }

    public void subdivideIterative(int numIts) { /* ... */ }
    void subdivideEdges(ArrayList<Edge> oldEdges, float scale) { /* ... */ }
    void connectNewTriangles(ArrayList<Integer> oldTriEIs, ArrayList<Edge> oldEdges) { /* ... */ }

    void writeVertexPositions(PrintWriter writer) { /* ... */ }
    void writeTriangleIndices(PrintWriter writer) { /* ... */ }
    public void writeToFile(String filename) { /* ... */ }
}
```

a)* What are the new subdivided vertex positions after one subdivision step for the following input points? Give the coordinate vectors.

0
1

```
vec3 v0 = new vec3(0.0, 0.0, 0.0);
vec3 v1 = new vec3(1.0, 0.0, 0.0);
vec3 v2 = new vec3(0.0, 1.0, 0.0);
```

b) Write the missing code for `subdivideIterative`. This function iteratively creates the triangle vertices and triangles by calling `subdivideEdges` and `connectNewTriangles`. The number of subdivision levels is controlled by the function input parameter. `scale` is the value with which we will later multiply `rand.nextGaussian()` to get a value from the distribution $\mathcal{N}(0, \frac{\sigma^2}{2^\ell})$.

0
1
2
3

```
public void subdivideIterative(int numIts) {
    // float scale = ...;
```

```
    for (int it = 0; it < numIts; it++) {
        ArrayList<Edge> oldEdges = edges;
        edges = new ArrayList<Edge>();
        subdivideEdges(oldEdges, scale);

        ArrayList<Integer> oldTriEIs = triEIs;
        triEIs = new ArrayList<Integer>();
        connectNewTriangles(oldTriEIs, oldEdges);

        // Update scale.
```

```
    }
}
```

0 1 2 3 4 5 6 7

c) Write the code for `subdivideEdges` to realize the following steps: Retrieve the vertices belonging to the old edge, add a new interpolated vertex, offset the vertex by a random displacment, add the two new edges to the edge list, and update the values of ne0 and ne1 of the old edge to reference the two new edges by their array index. ne0 and ne1 are later used in the function `connectNewTriangles` for re-triangulation, which you do NOT need to implement. `connectNewTriangles` is also responsible for adding the missing three edges within the subdivided triangle.

```
void subdivideEdges(ArrayList<Edge> oldEdges, float scale) {
    for (int i = 0; i < oldEdges.size(); i++) {
```

```
    }
}
```

Problem 2 Gouraud Shading and Texture Mapping (14 credits)

In this exercise, your task is to implement Gouraud Phong shading and texture mapping. GLSL version and floating point precision statements (like `#version 300 es` or `precision highp float;`) can be omitted from the code, but apart from that, your shaders should be able to compile. Compute the shading color using Gouraud shading and use the input texture coordinates to sample from a texture `baseTexture`, which is modulated with the Gouraud shading color in the fragment shader. Use uniform variables for specifying the position of the point light source `pointLightPosition` and its color `lightColor`.

a) Implement the vertex shader.

0 1 2 3 4 5 6 7 8 9 10

```
// Uniform and in and out variables.

// Model matrix
uniform mat4 mMatrix;
// View matrix
uniform mat4 vMatrix;
// Projection matrix
uniform mat4 pMatrix;

// Vertex position in object space coordinates
in vec3 vertexPosition;
// Surface normal at the vertex in object space coordinates
in vec3 vertexNormal;
// Texture coordinates at that vertex
in vec2 vertexTextureCoordinates;
```

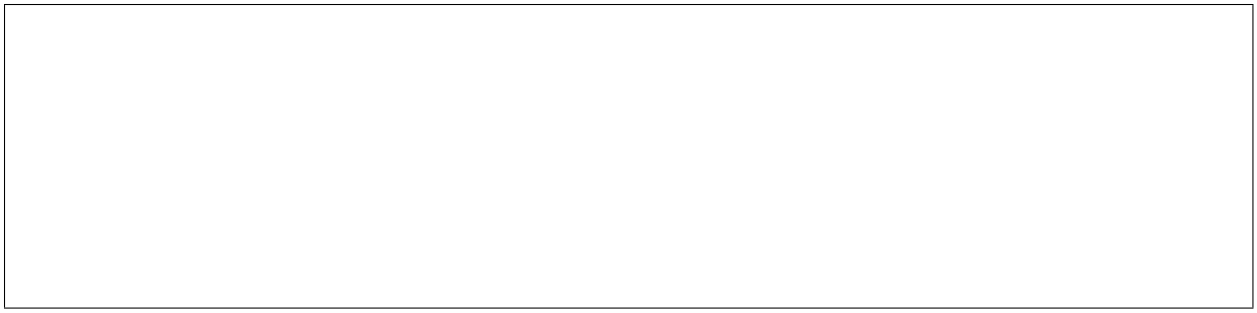
```
void main() {
   const float kA = 0.5; // Ambient factor
   const float kD = 0.5; // Diffuse factor
   const float kS = 0.8; // Specular factor
   const float s = 20.0; // Specular exponent
   const vec3 baseColor = vec3(1.0, 0.0, 0.0); // ambient and diffuse Phong base color
   const vec3 specularBaseColor = vec3(1.0, 1.0, 1.0); // specular Phong base color
```

```
}
```

0 1 2 3 4

b) Implement the fragment shader.

```
// Uniform and in and out variables.
```



```
void main() {
```

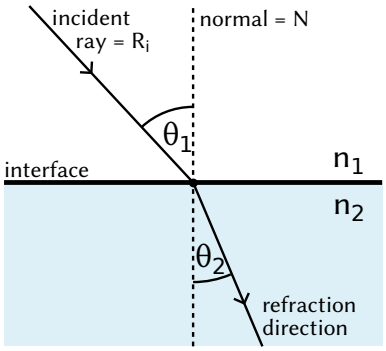
```
}
```

Problem 3 Whitted-style Ray Tracing (21 credits)

In this exercise, your task is to implement a Whitted-style ray tracer. Unlike discussed in the exercise, you need to consider two types of object materials.

1. A purely diffuse reflecting material. The Blinn-Phong model is used to simulate the reflection, with a shadow ray sent towards the light source.
2. A semi-transparent material with both a reflection and a refraction. Thus, at each ray-object intersection, one reflection and one refraction ray are cast into the scene. A refraction occurs when light travels from one medium (like air) to another medium (like glass or water), and the path may be redirected at the interface. This effect is described by Snell's law.

Given the indices of refraction of two media n_1 and n_2, Snell's law states regarding the angle of incidence θ_1 and the angle of refraction θ_2 that the equality $\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$ holds.



The code skeleton is given below. In the following tasks, you need to implement the functions `reflect`, `refract` and `Code Section (1)` and `Code Section (2)`.

```
// ...
struct Ray {
   vec3 origin; // Ray origin.
   vec3 dir; // Ray direction.
   float f; // Ray contribution factor to final color. I.e., we can add f * rayColor to the final color.
};

/* Returns true if the ray hits an object. Then hitObjectID contains the ID of the first object that
   has been hit by the ray and can be used as index into 'objectMaterialIsDiffuse'. */
bool traceRay(vec3 origin, vec3 dir, out vec3 hitPos, out vec3 hitNormal, out uint hitObjectID){/*...*/}
// Returns shaded surface color computed using the Blinn-Phong model. Casts a shadow ray towards light.
vec4 blinnPhong(vec3 worldPos, vec3 viewDir, vec3 surfaceNormal, uint hitObjectID) { /* ... */ }
// Returns amount of light reflected (compared to refracted). For Ri, N see figure above. eta = n2/n1.
float fresnel(vec3 Ri, vec3 N, float eta) { /* ... */ }
// ... functions reflect, refract ...

void main() {
  // ...
  vec3 viewOrigin = ...;
  vec3 viewDirection = ...;
  vec3 finalColor = vec3(0.0);
  float n1 = ..., n2 = ...; float eta = n1 / n2;

  const int MAX_STACK_SIZE = 16;
  int stackSize = 0;
  Ray stack[MAX_STACK_SIZE];
  // Maps object ID to whether it has a diffuse or reflective/refractive material.
  const bool objectMaterialIsDiffuse[] = { ... };

  // CODE SECTION (1)

  // CODE SECTION (2)

  fragColor = vec4(finalColor, 1.0);
}
```

0 1 2 a) Implement the function `reflect`, which computes the reflection direction of a ray direction *Ri* at a surface point with a normal *N* in GLSL. Do NOT use the built-in GLSL function `reflect`.

```
vec3 reflect(vec3 Ri, vec3 N) {
```

```
}
```

0 1 2 3 4 5 6 b) Implement the function `refract`, which computes the refraction direction of a ray direction *Ri* at a surface point with a normal *N* in GLSL. `eta` is the ratio of the indices of refraction $\frac{n_1}{n_2}$. Do NOT use the built-in GLSL function `refract`. HINT: Make use of the formula $v_{refract} = \frac{n_1}{n_2} Ri + (\frac{n_1}{n_2} \cos\theta_1 - \cos\theta_2)N$, and Snell's law stating $\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$. For these equations $\cos\theta_1 > 0$ must hold, which is true if *N* points toward the side where the ray comes from. Otherwise, use the negative normal and the inverse of `eta` in the equations. Also check for the case of total internal reflection, which is indicated by a negative radicand when solving for $\cos\theta_2$. In this case, just return `vec3(0.0)`.

```
vec3 refract(vec3 Ri, vec3 N, float eta) {
```

```
}
```

c) Implement `Code Section (1)`, which is responsible for pushing the first ray position and direction onto the stack with ray contribution factor 1. We manage our own stack, as GLSL does not support recursion.

0 1 2

d) Implement `Code Section (2)`. This section processes all rays on the stack until no rays are left or a maximum number of iterations is reached. It computes the intersection of the current ray with the objects in the scene using the given function `traceRay`, and checks if a hit was found (i.e., the function returns true). If the hit object is diffuse, it uses the Blinn-Phong model via the function `blinnPhong` to compute the object color and adds it to the `finalColor` modulated by the ray contribution factor. If the hit object is not diffuse, push a reflection and refraction ray onto the stack if the stack is not yet completely filled. The ray contribution factors of the new rays are computed as the product of the ray contribution factor of the old ray times the Fresnel factor kr (reflection ray) or one minus the Fresnel factor kr (refraction ray) computed using the function `fresnel`.
HINT: Only push the refraction ray onto the stack if no total internal reflection occurred (see task b)).

0 1 2 3 4 5 6 7 8 9 10 11

```
const int MAX_ITERATIONS = 64;
int iteration = 0;
while (stackSize > 0 && iteration < MAX_ITERATIONS) {
```

```
  iteration++;
}
```

Additional space for solutions–clearly mark the (sub)problem your answers are related to and strike out invalid solutions.