Aufgabe 1: Farben und Farbwahrnehmung

Teilaufgabe 1a: Chromatizitätsdiagramm

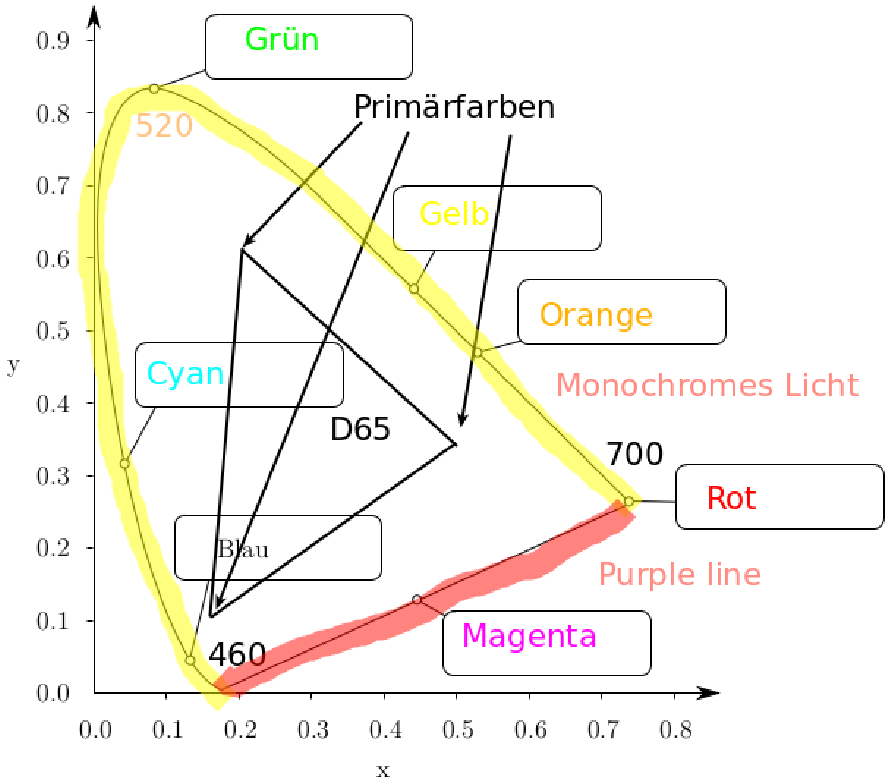


Abbildung 1: Aufgabe 1a

Teilaufgabe 1b

Alles auf der Purple line. Also insbesondere **Magenta**.

Teilaufgabe 1c

$$x = \frac{X}{X+Y+Z} \tag{1}$$

$$y = \frac{Y}{X+Y+Z} \tag{2}$$

Aussage	Wahr	Falsch	Begründung
Den Weißpunkt eines Farbraums bezeichnet man auch als Tristimulus-wert.	☐	☑	Die RGB-Werte sind die Tristimulus-Werte. Der Weißpunkt heißt pblicherweise D[Zahl], wobei die Zahl die Temperatur angibt. D65 hat eine Farbtemperatur von ca. 6504K.
Die subjektiv empfundene Stärke von Sinneseindrücken ist proportional zum Logarithmus ihrer Intensität.	☑	☐	
Jeder Farbeindruck für den Menschen kann mit drei Grundgrößen beschrieben werden.	☐	☑	vgl. 1 (b)

Teilaufgabe 1d

(2) < (3) < (1), also
RGB < Raum aller Farben die durch 100 monochromatische Leuchtdioden darstellbar sind < XYZ

Teilaufgabe 1e

Aufgabe 2: Whitted-Style Raytracing

Teilaufgabe 2a-d

Siehe Abbildung 2.

Teilaufgabe 2e

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t \tag{3}$$

$$1 \cdot \frac{4}{10} = 1.5 \sin\theta_t \tag{4}$$

$$\Leftrightarrow \sin\theta_t = \frac{4}{15} = \frac{2}{7.5} \tag{5}$$

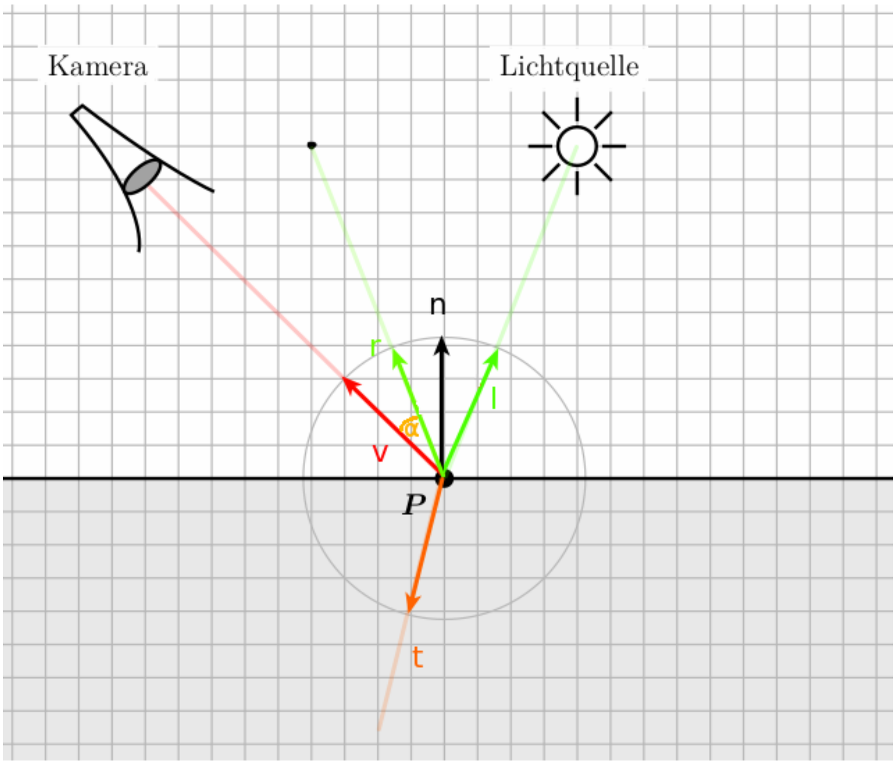


Abbildung 2: Aufgabe 2a-d; $n_1 = 1, n_2 = 1.5$

Teilaufgabe 2f

$$I_s = k_s \cdot I_L \cdot \cos^n \alpha \tag{6}$$

$$\alpha = r_L \cdot v \tag{7}$$

wobei k_s ein Materialparameter und I_L die intensität der Lichtquelle ist. n wird der Phong-Exponent genannt (TODO: woher kommt der?)

Teilaufgabe 2g

Snellsches Brechungsgesetz

$$\eta_i \sin \theta_i = \eta_t \sin \theta_t$$

Aufgabe 3: Transformationen

$$\begin{pmatrix} s_x & h_x & t_x \\ h_y & s_y & t_y \\ a & b & c \end{pmatrix}$$

- Die Parameter s_x, s_y skalieren in Richtung der x bzw. y Achse.
- Die Parameter h_x, h_y scheeren in Richtung der x bzw. y Achse.
- Die Parameter t_x, t_y füren eine Translation in x bzw. y Richtung aus.
- Die Parameter a, b, c skalieren.

Die Matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

rotiert um θ um den Ursprung (gegen den Uhrzeigersinn.)

- Bild 1: Translation um 1 in x und 3 in y-Richtung.
- Bild 2: Scherung im -2 in y-Richtung.
- Bild 3: Rotation um 45° gegen den Urzeigersinn.
- Bild 4: In x-Richtung um $1/2$ stauchen, in y-Richtung um 3 Strecken und dann um 4 nach rechts verschieben.
- Bild 5: Projektion auf die zur x-Achse parallele Gerade durch $(0, 3)$.

Aufgabe 4

Teilaufgabe 4a

TODO

Teilaufgabe 4b

TODO

Teilaufgabe 4c

Teilaufgabe 4c (I)

TODO

Teilaufgabe 4c (II)

TODO

Teilaufgabe 4c (III)

TODO

Teilaufgabe 4d

Aussage	Wahr	Falsch	Begründung
Texturkoordinaten müssen sich immer im Intervall $[0; 1]$ befinden.	☐	☐	
Texturkoordinaten können als Attribute der Eckpunkte (Vertizes) übergeben werden und werden als solche interpoliert.	☐	☐	
Texturkoordinaten müssen für die Darstellung wie Eckpunktkoordinaten der Model-View-Transformation unterzogen werden.	☐	☐	

Aufgabe 5: Vorgefilterte Environment-Maps

Teilaufgabe 5a

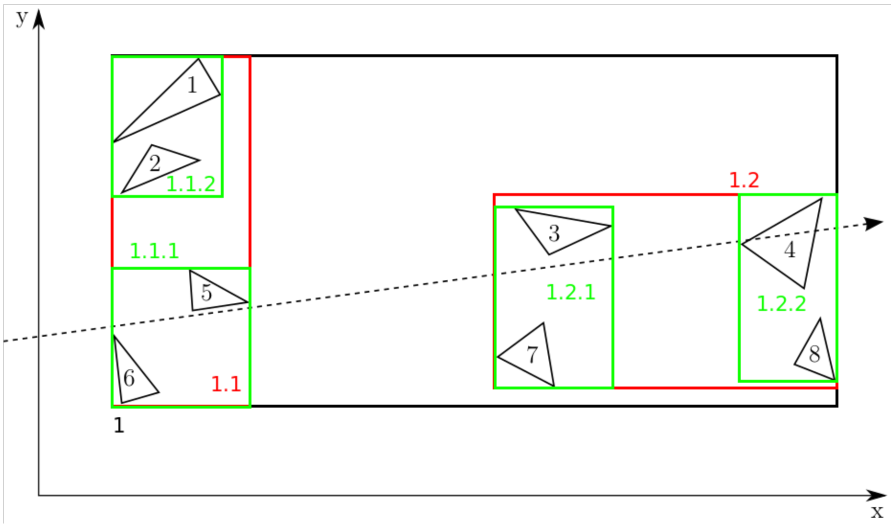
TODO

Teilaufgabe 5b

TODO

Aufgabe 6: Hierarchische Datenstrukturen

Teilaufgabe 6a



Teilaufgabe 6b

Inklusive Schnittests der AABB Hüllkörper:

1. 1
2. 1.1
3. 1.1.1
4. 5, 6
5. 1.1.2
6. 1.2
7. 1.2.1
8. 3, 7
9. 1.2.2
10. 4, 8

Teilaufgabe 6c

Aussage	Wahr	Falsch	Begründung
Beim Traversieren eines kD-Baums müssen immer beide Kinder in Betracht gezogen werden.	☐	☑	vgl. Folie 103
Das Traversieren einer Hüllkörperhierarchie mit achsenparallelen Boxen (Bounding Volume Hierarchy, BVH) erfordert Mailboxing, um mehrfache Schnitttests mit einem Dreieck zu verhindern.	☑	☐	
Der Speicheraufwand einer BVH hängt logarithmisch von der Anzahl der Primitive ab.	☑	☐	
kD-Bäume sind eine Verallgemeinerung von BSP-Bäumen.	☐	☑	Es ist genau anders herum. kD-Bäume müssen Achsenparallele Trennebenen haben, BSP-Bäume jedoch nicht.
BSP-Bäume sind adaptiv und leiden nicht unter dem „Teapot in a Stadium“-Problem.	☑	☐	

Teilaufgabe 6d

Aussage	BVH	Octree	kD-Baum	Gitter
Die Datenstruktur partitioniert den Raum.	☐	☑	☑	☑
Der Aufwand für den Aufbau der Datenstruktur ist linear in der Anzahl der Primitive.	☐	☐	☐	☐
Eine effizientere Traversierung wird erreicht, wenn die Surface Area Heuristic bei der Konstruktion verwendet wird.	☐	☐	☑	☐
Die Datenstruktur eignet sich am besten für Szenen, in denen die Geometrie gleichmäßig verteilt ist und kaum leere Zwischenräume vorhanden sind.	☐	☐	☐	☑

Aufgabe 7: Rasterisierung und OpenGL

Teilaufgabe 7a

Aussage	Wahr	Falsch	Begründung / Quelle
In der OpenGL-Pipeline wird View Frustum Clipping vor der perspektivischen Division durchgeführt.			
Vertex-Shader können auf Texturen zugreifen.			
Bei Gouraud-Shading muss man die Normale im Fragment-Shader erneut normalisieren.			
Gouraud-Shading mit dem Phong-Beleuchtungsmodell kann im Geometry-Shader implementiert werden.			
Phong-Shading kann man alleine mit einem Vertex-Shader und einem Geometry-Shader implementieren; letzterer gibt dann die Farbe aus.			
Bei beliebig feiner Tessellierung ist kein Unterschied zwischen Gouraud- und Phong-Shading erkennbar.			
Selbst wenn der Tiefentest für ein Fragment fehlschlägt, kann der Stencil-Puffer verändert werden.			
Instanziierung von Geometrie kann man sowohl mit dem Vertex- als auch dem Geometry-Shader durchführen.			

Teilaufgabe 7b

Warum zieht man das Tiefenpuffer-Verfahren (Z-Buffering) dem Sortieren von Dreiecken vor? Nennen Sie drei Gründe!

- Dreiecke können nicht sortierbar sein (wenn ein Dreieck ein andere schneidet)
- TODO
- TODO

Aufgabe 8: OpenGL-Primitive

(a) `GL_TRIANGLE_STRIP`: Ganz links ist (1), (2) ist rechts unten davon, (3) ist rechts oben von (1). Dann im Zick-Zack-Muster weiter.
(b) `GL_TRIANGLE_FAN`: Der mittlere Knoten ist (1), dann wird von ganz links gegen den Uhrzeigersinn nummeriert.

Aufgabe 9: OpenGL und Blending

Teilaufgabe 9a

Teilaufgabe 9a (I)

- Die Reihenfolge ist wegen des Tiefenpuffers egal: TODO
- Von hinten nach vorne: TODO
- Von vorne nach hinten: TODO

Teilaufgabe 9a (II)

```
glBlendFunc(TODO, TODO)
```

Teilaufgabe 9b

```
glBlendFunc(TODO, TODO)
```

Teilaufgabe 9c

```
glBlendFunc(TODO, TODO)
```

Aufgabe 10: Bézier-Kurven und Bézier-Splines

Teilaufgabe 10a

- Bézier-Kurven liegen innerhalb der konvexen Hülle ihrer Kontrollpunkte.
- c_0c_1 ist Tangential an die Bezierkurve am Anfang.
- c_2c_3 ist Tangential an die Bezierkurve am Ende.
- **Wertebereich**: Bézierkurven liegen innerhalb der konvexen Hülle, die durch die 4 Kontrollpunkte gebildet werden.
- **Endpunktinterpolation**: Bézierkurven beginnen immer beim ersten Kontrollpunkt und enden beim letzten Kontrollpunkt.

- **Variationsredukion**: Eine Bézierkurve F wackelt nicht stärker als ihr Kontrollpolygon B ($\sharp(H \cap F) \leq \sharp(H \cap B)$).

Teilaufgabe 10b

shader.vert

```
uniform mat4 matrixMVP; // Model-View-Projection-Matrix
in vec3 position; // Koordinaten des Eingabe-Vertex
uniform vec3 b[12]; // Array der Kontrollpunkte
uniform float time; // Zeitpunkt für die Animation in [0;3)

// bezier3(..) soll die Bezier-Kurve an der Stelle s
//    auswerten und das Resultat als vec3 zurückgeben.
//    Sie können die Bernstein-Polynome oder den
//    Algorithmus von de Casteljau verwenden.
vec3 bezier3(float s, // Parameter s in [0;1)
             const vec3 b0, const vec3 b1, // Kontrollpunkte b0, b1, b2, b3
             const vec3 b2, const vec3 b3) {
    // Fügen Sie Ihren Code hier ein.
    vec3 result = vec3(0.f, 0.f, 0.f);
    result += b0 * (1-u) * (1-u) * (1-u) * 1;
    result += b1 * (1-u) * (1-u) * u * 3;
    result += b2 * (1-u) * u * u * 3;
    result += b3 * u * u * u * 3;
    return result;
}

// bezierspline3(..) soll die Auswertung des Bezier-Splines an der
//    Stelle t als vec3 zurückgeben.
//    Verwenden Sie dazu die Funktion bezier3(..)!
vec3 bezierspline3(float t) {
    // Fügen Sie Ihren Code hier ein.
    int i=0;
    float s = t;
    while(s >= 1.0f) {
        s -= 1;
        i += 1;
    }
    return bezierspline3(s, b[3*i], b[3*i+1], b[3*i+2], b[3*i+3]);
}

void main() {
    vec3 offset = bezierspline3(time);
    vec3 newpos = position + offset;
    gl_Position = matrixMVP * vec4(newpos,1.0);
}
```

Aufgabe 11: Wasseroberfläche mit GLSL

Teilaufgabe 11a

shader.frag

```
vec3 determineIntersection(in vec3 P, in vec3 r, out int index)
{
    // Ermitteln Sie hier den Schnittpunkt mit der nächsten Gefäßfläche
    // und geben Sie ihn zurück. Zusätzlich muss 'index' auf den Index
    // der entsprechenden Seitenfläche gesetzt werden.

    bool intersects = false;
    float t, t_min;
    int index_min, index;

    for (int i=0; i<= 5; i++) {
        if (intersect(i, P, r, &t)) {
            if (!intersects || t < t_min) {
                t_min = t;
                index_min = i;
                intersects = true;
            }
        }
    }

    index = index_min;
    return t_min;
}
```

Teilaufgabe 11b

shader.frag

```
vec2 determineTextureCoordinate(in vec3 S, in int index)
{
    vec2 UV;
    switch(index)
    {
        // Vervollständigen Sie die Fälle entsprechend der Aufgabenstellung
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
        case 5: TODO
    }
    // Fügen Sie ggf. notwendige weitere Anweisungen hier ein
    return UV;
}
```