

## Aufgabe 1: Farben und Farbwahrnehmung

### Teilaufgabe 1a: Chromatizitätsdiagramm

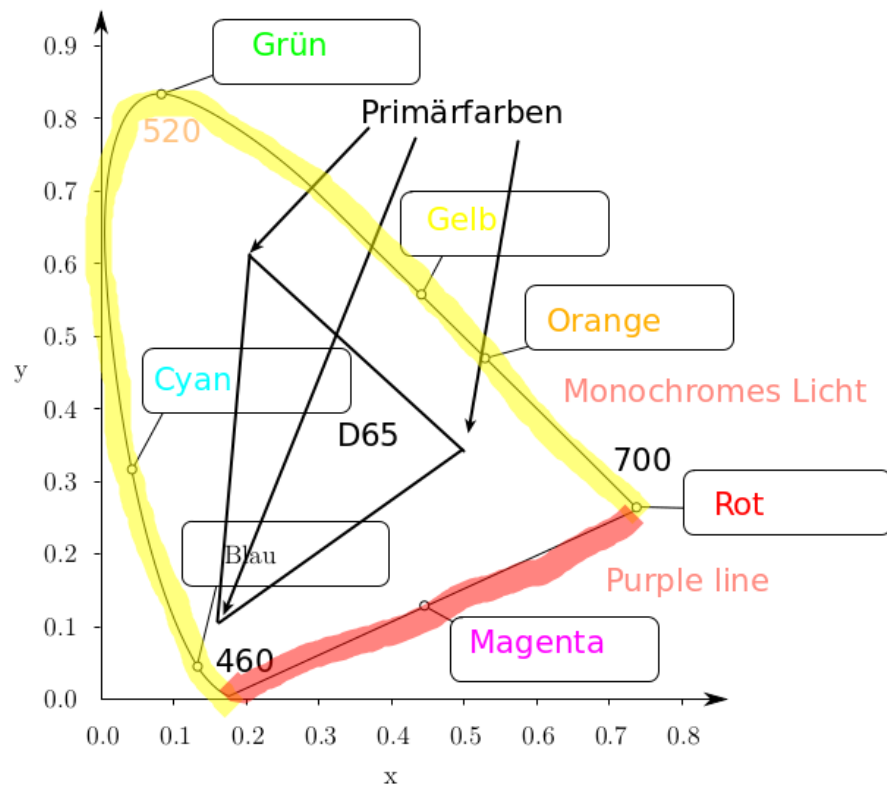


Abbildung 1: Aufgabe 1a

### Teilaufgabe 1b

Welcher der Farbeindrücke aus Aufgabe a) lässt sich nicht durch monochromatisches Licht erzeugen?

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

### Teilaufgabe 1c

$$x = \frac{X}{X + Y + Z} \quad (1)$$

$$y = \frac{Y}{X + Y + Z} \quad (2)$$

### Teilaufgabe 1d

(2) < (3) < (1), also

RGB < Raum aller Farben die durch 100 monochromatische Leuchtdioden darstellbar sind < XYZ

### Teilaufgabe 1e

#	Aussage	Wahr	Falsch	Begründung
1	Den Weißpunkt eines Farbraums bezeichnet man auch als Tristimuluswert.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Die RGB-Werte sind die Tristimulus-Werte. Der Weißpunkt heißt üblicherweise $D[\text{Zahl}]$ , wobei die Zahl die Temperatur angibt. D65 hat eine Farbtemperatur von ca. 6504K.
2	Die subjektiv empfundene Stärke von Sinneseindrücken ist proportional zum Logarithmus ihrer Intensität.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
3	Jeder Farbeindruck für den Menschen kann mit drei Grundgrößen beschrieben werden.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1. Graßmannsches Gesetz

## Aufgabe 2: Whitted-Style Raytracing

### Teilaufgabe 2a-d

Siehe Abbildung 2.

### Teilaufgabe 2e

$$\eta_i \sin \theta_i = \eta_t \sin \theta_t \quad (3)$$

$$1 \cdot \frac{4}{10} = 1.5 \sin \theta_t \quad (4)$$

$$\Leftrightarrow \sin \theta_t = \frac{4}{15} = \frac{2}{7.5} \quad (5)$$

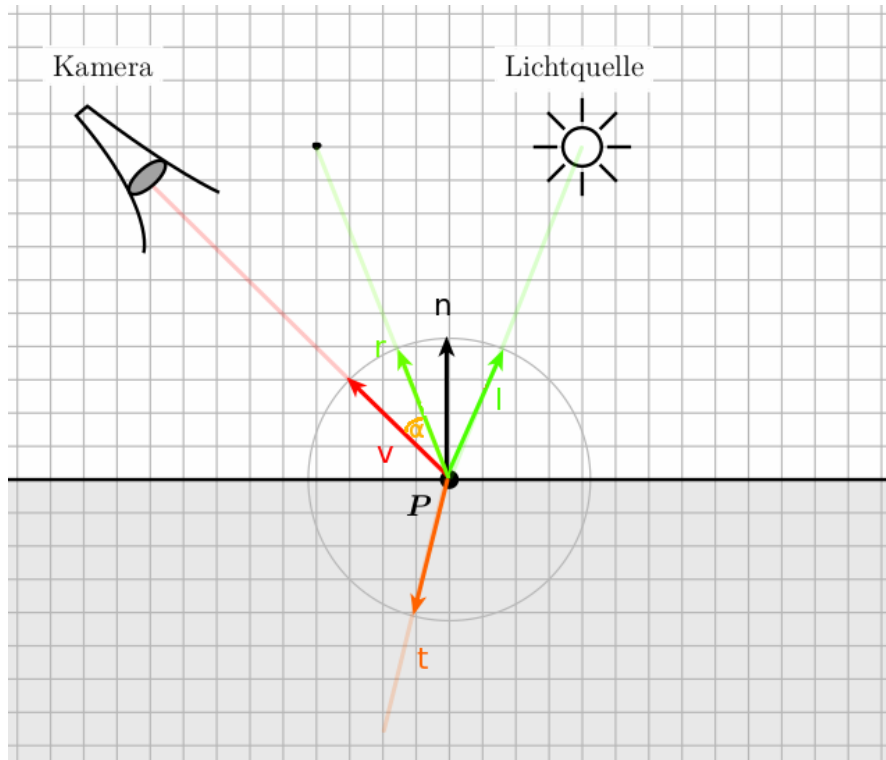


Abbildung 2: Aufgabe 2a-d;  $n_1 = 1, n_2 = 1.5$

### Teilaufgabe 2f

$$I_s = k_s \cdot I_L \cdot \cos^n \alpha \quad (6)$$

$$\alpha = r_L \cdot v \quad (7)$$

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  $t_x, t_y$  führen 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  $\theta$  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  $45^\circ$  gegen den Uhrzeigersinn.
- 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

Es müssen nur die Mittelwerte berechnet werden, also:

- Stufe 1: 5, 3, 8, 4
- Stufe 2: 4, 6
- Stufe 3: 5

### Teilaufgabe 4b

- oben: 2.8
- mitte: 1.8
- unten: 1.1

Siehe Abbildung 3 (vgl. Kapitel 4, Folie 58)

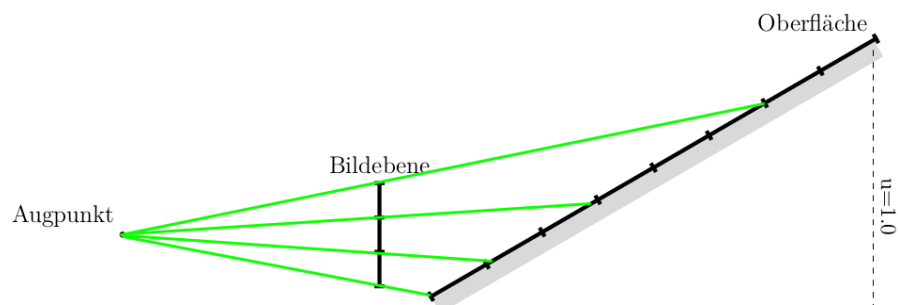


Abbildung 3: Aufgabe 4b; Der Footprint eines Bildpixels in der Textur wird ermittelt, indem man überprüft wie viele Texel diesen Bildpixel beeinflussen.

### Teilaufgabe 4c

#### Teilaufgabe 4c (I)

TODO

#### Teilaufgabe 4c (II)

TODO

#### Teilaufgabe 4c (III)

TODO

### Teilaufgabe 4d

#	Aussage	Wahr	Falsch	Begründung
1	Texturkoordinaten müssen sich immer im Intervall $[0; 1]$ befinden.	<input type="checkbox"/>	<input type="checkbox"/>	
2	Texturkoordinaten können als Attribute der Eckpunkte (Vertizes) übergeben werden und werden als solche interpoliert.	<input type="checkbox"/>	<input type="checkbox"/>	
3	Texturkoordinaten müssen für die Darstellung wie Eckpunktkoordinaten der Model-View-Transformation unterzogen werden.	<input type="checkbox"/>	<input type="checkbox"/>	

## Aufgabe 5: Vorgefilterte Environment-Maps

### Teilaufgabe 5a

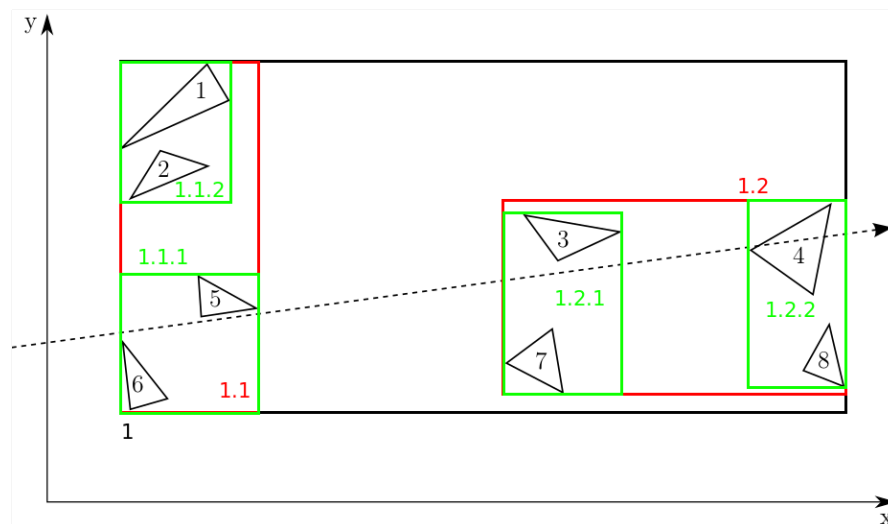
TODO

### Teilaufgabe 5b

TODO

## Aufgabe 6: Hierarchische Datenstrukturen

### Teilaufgabe 6a



### Teilaufgabe 6b

Inklusive Schnitttests 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
1	Beim Traversieren eines kD-Baums müssen immer beide Kinder in Betracht gezogen werden.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	vgl. Folie 103
2	Das Traversieren einer Hüllkörperhierarchie mit achsenparallelen Boxen (Bounding Volume Hierarchy, BVH) erfordert Mailboxing, um mehrfache Schnittpunkte mit einem Dreieck zu verhindern.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
3	Der Speicheraufwand einer BVH hängt logarithmisch von der Anzahl der Primitive ab.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
4	kD-Bäume sind eine Verallgemeinerung von BSP-Bäumen.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Es ist genau anders herum. kD-Bäume müssen Achsenparallele Trennebenen haben, BSP-Bäume jedoch nicht.
5	BSP-Bäume sind adaptiv und leiden nicht unter dem „Teapot in a Stadium“-Problem.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	

### Teilaufgabe 6d

#	Aussage	BVH	Octree	kD-Baum	Gitter
1	Die Datenstruktur partitioniert den Raum.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
2	Der Aufwand für den Aufbau der Datenstruktur ist linear in der Anzahl der Primitive.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	Eine effizientere Traversierung wird erreicht, wenn die Surface Area Heuristic bei der Konstruktion verwendet wird.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
4	Die Datenstruktur eignet sich am besten für Szenen, in denen die Geometrie gleichmäßig verteilt ist und kaum leere Zwischenräume vorhanden sind.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

## Aufgabe 7: Rasterisierung und OpenGL

### Teilaufgabe 7a

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

### 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

- **Tangentenbedingung:**  $c_0c_1$  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.
- **Variationsreduktion:** Eine Bézierkurve  $F$  wackelt nicht stärker als ihr Kontrollpolygon  $B$  ( $\sharp(H \cap F) \leq \sharp(H \cap B)$ ).
- **Affine Invarianz**

## Teilaufgabe 10b

```
shader.vert
1 uniform mat4 matrixMVP; // Model-View-Projection-Matrix
2 in vec3 position; // Koordinaten des Eingabe-Vertex
3 uniform vec3 b[12]; // Array der Kontrollpunkte
4 uniform float time; // Zeitpunkt für die Animation in [0;3)
5
6 // bezier3(..) soll die Bezier-Kurve an der Stelle s
7 //   auswerten und das Resultat als vec3 zurückgeben.
8 //   Sie können die Bernstein-Polynome oder den
9 //   Algorithmus von de Casteljau verwenden.
10 vec3 bezier3(float s, // Parameter s in [0;1)
11             const vec3 b0, const vec3 b1, // Kontrollpunkte b0, b1, b2, b3
12             const vec3 b2, const vec3 b3) {
13     // Fügen Sie Ihren Code hier ein.
14
15     // Lösung mit Bernstein-Polynomen
16     vec3 result = vec3(0.);
17     result += b0 * (1. - s) * (1. - s) * (1. - s) * 1.;
18     result += b1 * (1. - s) * (1. - s) * s * 3.;
19     result += b2 * (1. - s) * s * s * 3.;
20     result += b3 * s * s * s * 3.;
21     return result;
22 }
23
24 vec3 bezier3(float s, // Parameter s in [0;1)
25             const vec3 b0, const vec3 b1, // Kontrollpunkte b0, b1, b2, b3
26             const vec3 b2, const vec3 b3) {
27     // Algorithmus von de Casteljau
28     vec3 b01 = mix(b0, b1, s);
29     vec3 b11 = mix(b1, b2, s);
30     vec3 b21 = mix(b2, b3, s);
31     vec3 b02 = mix(b01, b11, s);
32     vec3 b12 = mix(b11, b21, s);
33     return mix(b02, b12, s);
34 }
35
36 // bezierspline3(..) soll die Auswertung des Bezier-Splines an der
37 //   Stelle t als vec3 zurückgeben.
38 //   Verwenden Sie dazu die Funktion bezier3(..)!
39 vec3 bezierspline3(float t) {
40     // Fügen Sie Ihren Code hier ein.
41     int i = int(t);
```

```
42     float s = fract(t);
43     return bezierspline3(s, b[3*i], b[3*i+1], b[3*i+2], b[3*i+3]);
44 }
45
46 void main() {
47     vec3 offset = bezierspline3(time);
48     vec3 newpos = position + offset;
49     gl_Position = matrixMVP * vec4(newpos,1.0);
50 }
```

---

## Aufgabe 11: Wasseroberfläche mit GLSL

### Teilaufgabe 11a

---

```
1  shader.frag
2  vec3 determineIntersection(in vec3 P, in vec3 r, out int index)
3  {
4      // Ermitteln Sie hier den Schnittpunkt mit der nächsten Gefäßfläche
5      // und geben Sie ihn zurück. Zusätzlich muss 'index' auf den Index
6      // der entsprechenden Seitenfläche gesetzt werden.
7
8      bool intersects = false;
9      float t_min;
10
11     for (int i = 0; i <= 5; i++) {
12         float t;
13         if (intersect(i, P, r, t) && t > 0.) {
14             if (!intersects || t < t_min) {
15                 t_min = t;
16                 index = i;
17                 intersects = true;
18             }
19         }
20     }
21
22     return P + t_min * r;
```

---

## Teilaufgabe 11b

---

```
1  shader.frag
2  vec2 determineTextureCoordinate(in vec3 S, in int index)
3  {
4      vec2 UV;
5      switch(index)
6      {
7          // Vervollständigen Sie die Fälle entsprechend der Aufgabenstellung
8          case 0:
9              UV = P.yz;
10             break;
11          case 2:
12          case 3:
13              UV = P.xy;
14              break;
15          case 4:
16          case 5:
17              UV = P.xz;
18              break;
19      }
20      // Fügen Sie ggf. notwendige weitere Anweisungen hier ein
21      UV = UV * .5 + .5;
22      return UV;
23  }
```

---