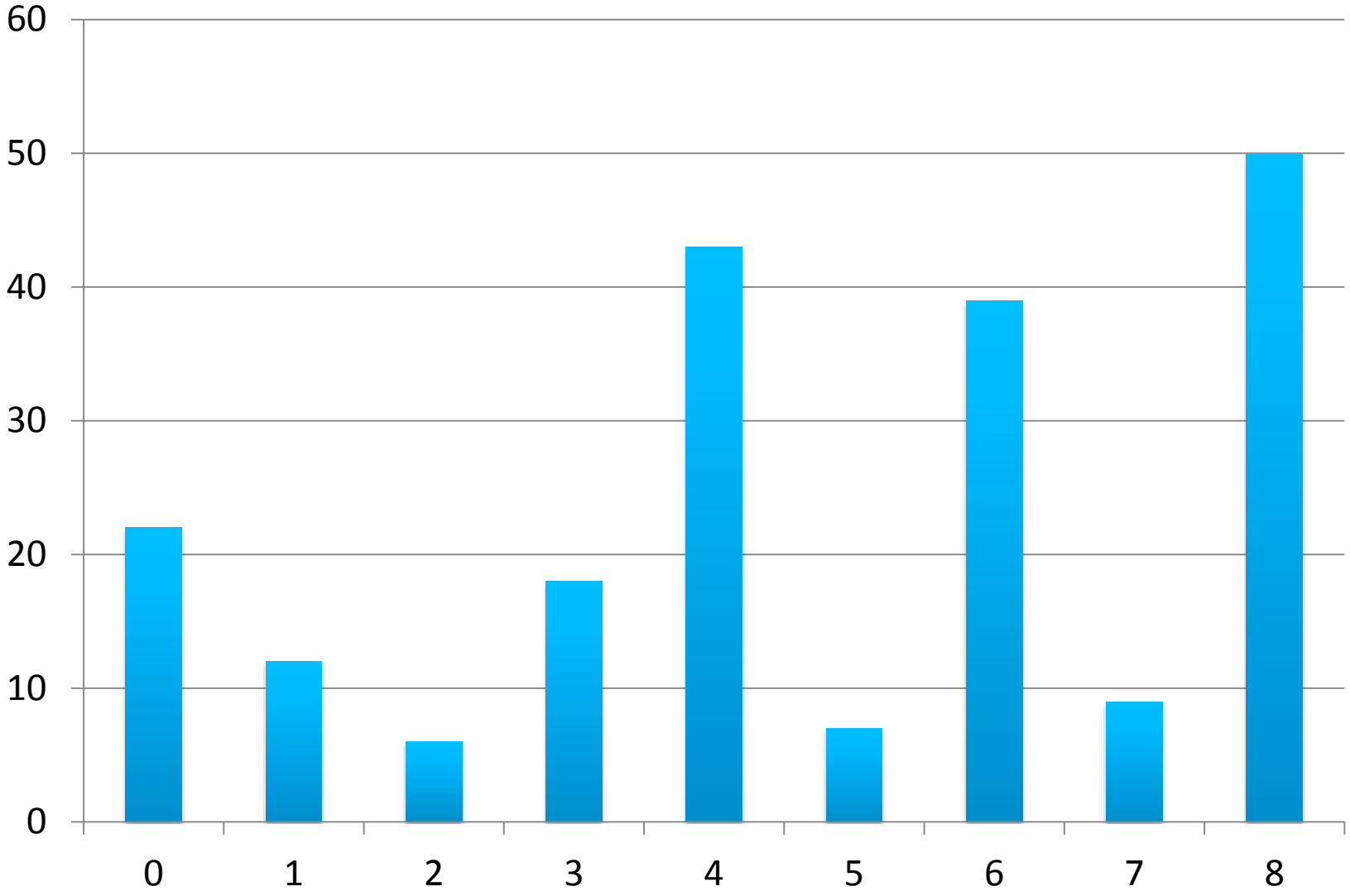
Computergraphik

Vorlesung im Wintersemester 2013/14
Kapitel 6ü: Clipping, Teil 2

Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie

Ergebnisse Übungsblatt 7

- Punkteverteilung:



Ergebnisse Übungsblatt 7

Häufige Fehler:

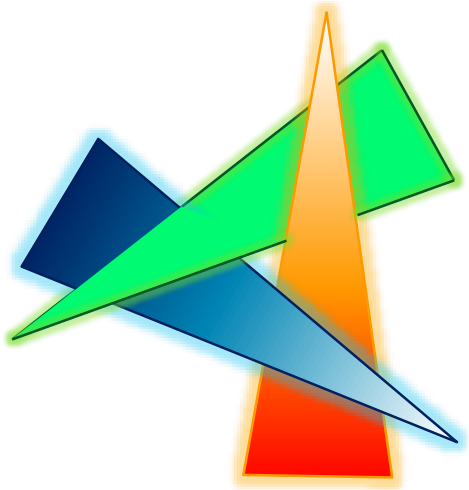
- Kein Clamping der MipMap-Level
  - führt zu Absturz

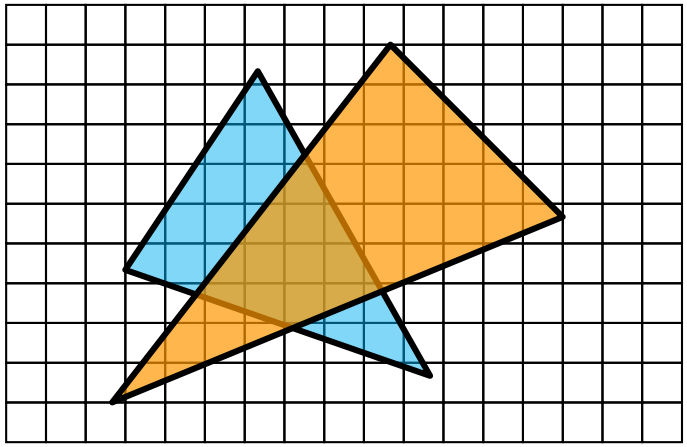
Übungsblatt 8 – Aufgabe 2

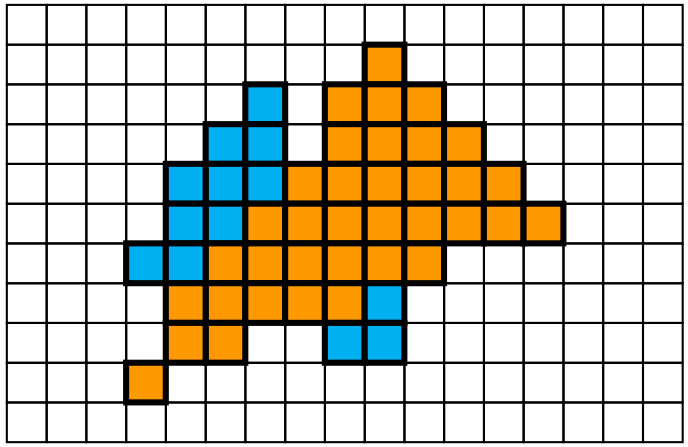
- Welche Probleme können durch die Interpolation von Normalen bei der Berechnung von Reflexionsrichtungen auftreten?
  Betrachten Sie dazu die Extremfälle, die beispielsweise bei einer groben Tesselierung einer Kugel auftreten.
- Beim Aufbau der BVH werden die Objekte/Dreiecke immer wieder sortiert. Fällt Ihnen eine Möglichkeit ein, die Anzahl der Sortiervorgänge zu reduzieren?

Inhalt

- Clipping von Linien und Polygonen (in der **Übung**)
  - Cohen-Sutherland und α-Clipping
  - Sutherland-Hodgeman







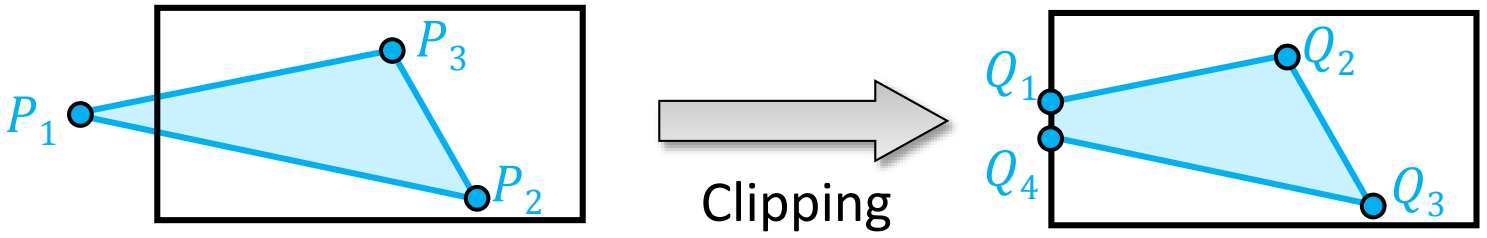
Line clipping (Wiederholung)

- Algorithmus für Clipping einer Linie an einem Rechteck:
  - Bestimme Window Edge Coordinates und Outcodes
  - Prüfe trivial accept / trivial reject
  - Setze $\alpha_{min} = 0, \alpha_{max} = 1$
  - Für alle Kanten mit unterschiedlichen Outcode-Bits in Endpunkten:
    - Bestimme Lage des Schnittpunkts α_S durch
      $$\alpha_s = \frac{WEC_E(P_1)}{WEC_E(P_1) - WEC_E(P_2)}$$
    - Falls P_1 außerhalb, $\alpha_{min} = \max(\alpha_{min}, \alpha_S)$
    - Sonst $\alpha_{max} = \min(\alpha_{max}, \alpha_S)$
    - Falls $\alpha_{max} < \alpha_{min}$ Linie außerhalb -> Ende
  - Zeichne Linie zwischen $\mathbf{P}_1 + \alpha_{min}(\mathbf{P}_2 - \mathbf{P}_1)$ und $\mathbf{P}_1 + \alpha_{max}(\mathbf{P}_2 - \mathbf{P}_1)$

- Und was machen wir bei Polygonen?

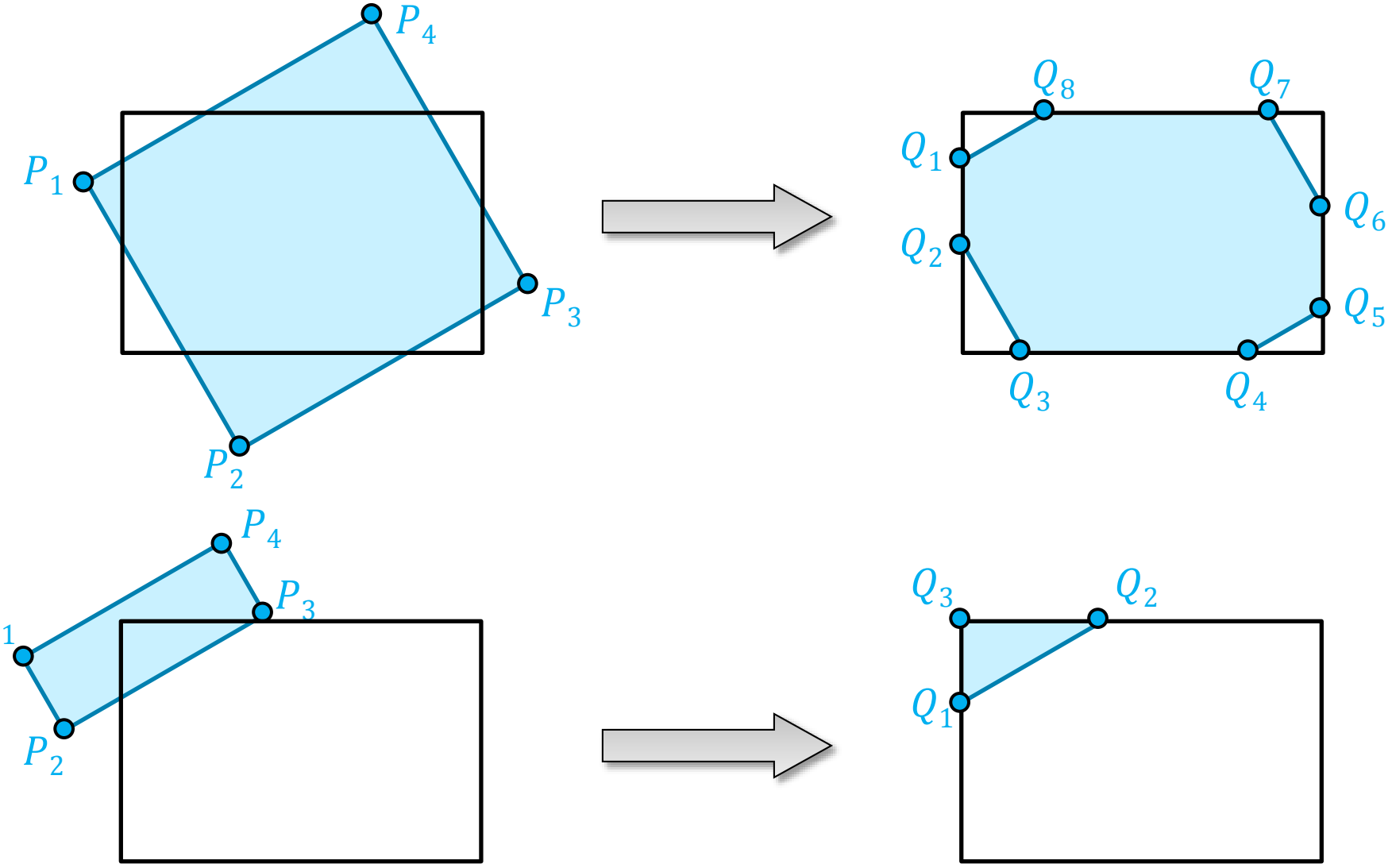
Clipping von Polygonen

- Polygon: Geordnete Liste von Punkten



Clipping von Polygonen

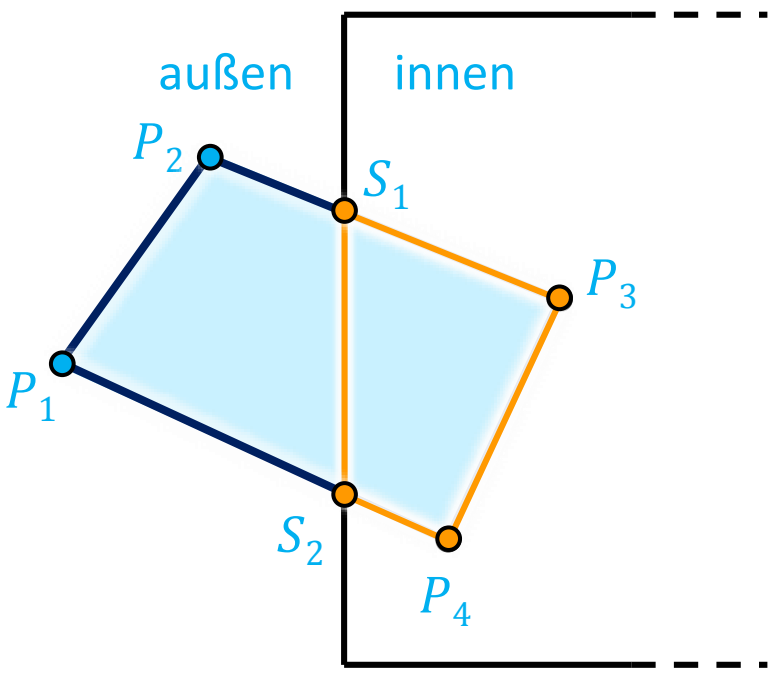
- wir brauchen mehr als nur die Liniensegmente, die durch Clipping der Polygonkanten entstehen → Algorithmus von Sutherland-Hodgeman (1974)



Sutherland-Hodgeman Polygon Clipping

Algorithmus für (konvexe) Viewports in 2D

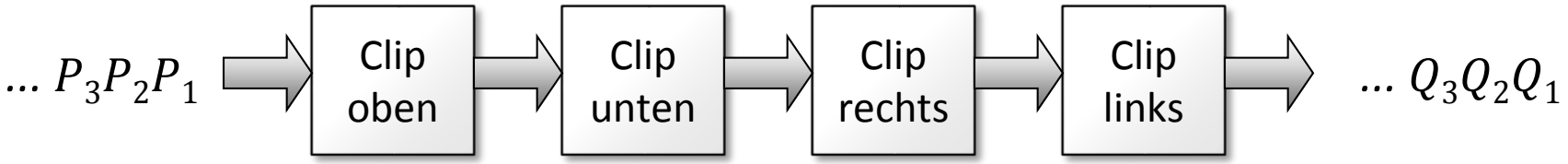
- Clipping gegen eine Kante nach der Anderen
- für jede Kante
  - Eingabe: Liste der Vertizes des Polygons
  - Ausgabe: Liste von Vertizes des resultierenden Polygons
  - Beispiel:
    - Eingabe: P_1, P_2, P_3, P_4
    - Ausgabe: S_1, P_3, P_4, S_2

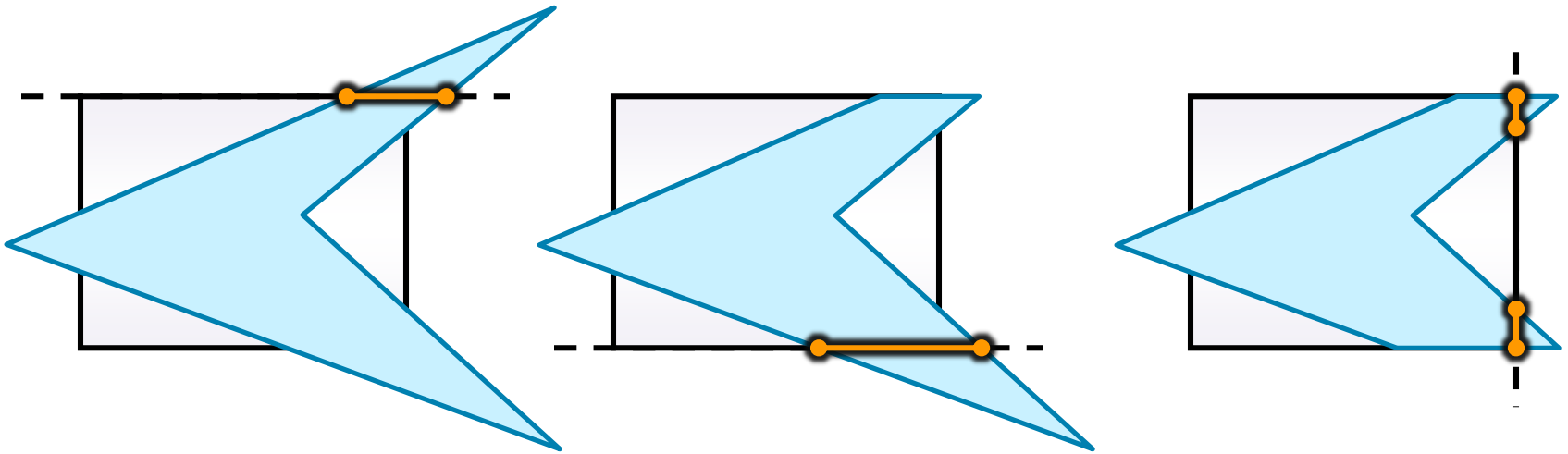


Sutherland-Hodgeman Polygon Clipping

Algorithmus für (konvexe) Viewports in 2D

- Clipping gegen eine Kante nach der Anderen
  - nach jeder Kante erhält man ein geschlossenes Polygon
  - erlaubt Pipelining (wichtig für Hardware-Umsetzungen)

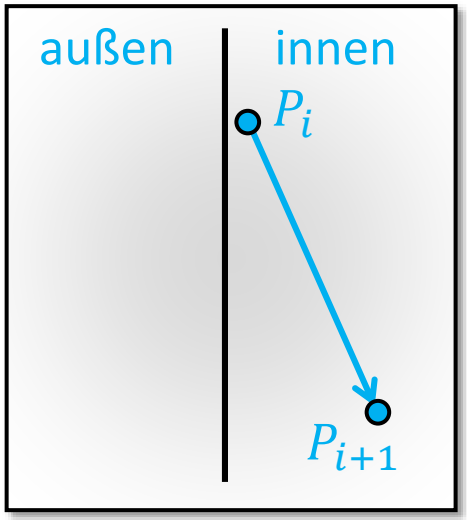




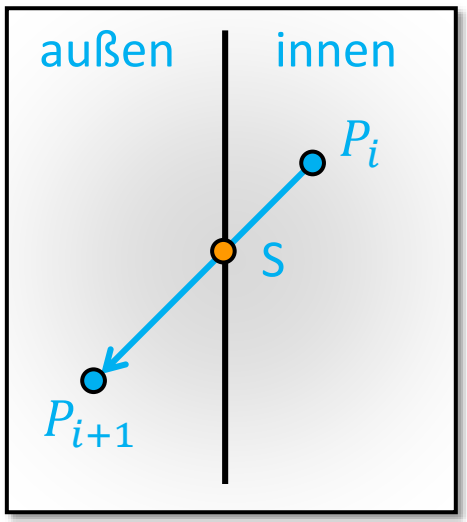
Sutherland-Hodgeman Polygon Clipping

Clipping gegen eine Kante

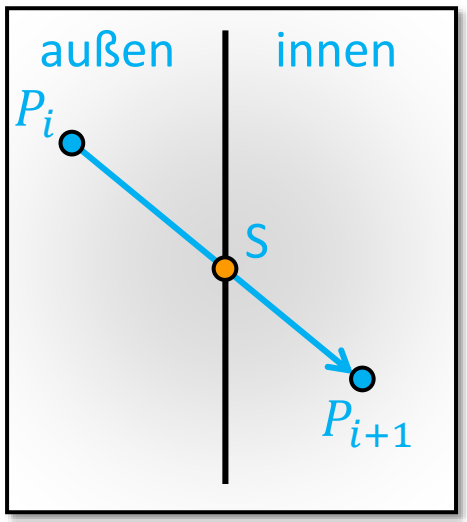
- geg. Liste von Eingabe-Vertizes P_1, P_2, P_3, P_4
- betrachte eine Kante P_iP_{i+1} nach der anderen
- klassifiziere P_i bzw. P_{i+1} als innen/außen → 4 Fälle sind zu unterscheiden



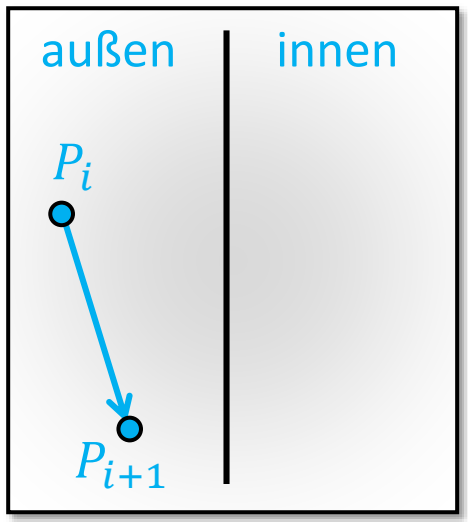
innen → innen:
füge P_{i+1} zur Ausgabe hinzu



innen → außen:
berechne Schnittpunkt S, gebe S aus



außen → innen:
berechne Schnittpunkt S, gebe S und P_{i+1} aus

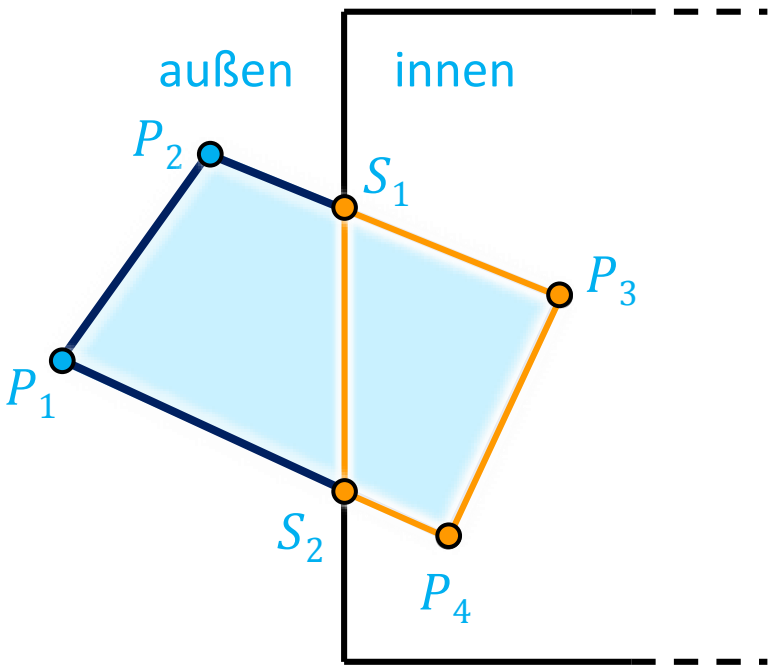


außen → außen:
weiter mit nächster Kante

Sutherland-Hodgeman Polygon Clipping

Beispiel

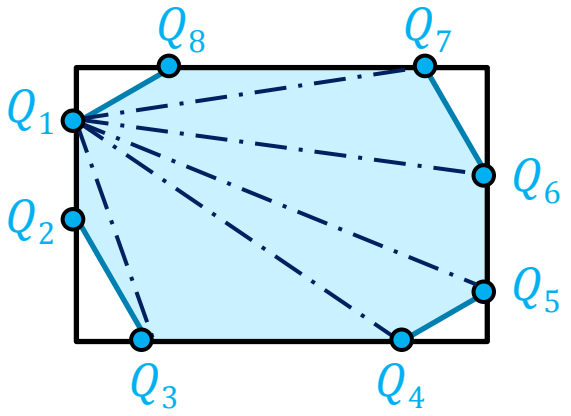
- Eingabe: P_1, P_2, P_3, P_4
- getestete Kanten: $P_1P_2, P_2P_3, P_3P_4, P_4P_1$ (Schließen des Kantenzugs)
- Ausgabe:
  - $P_1P_2 \rightarrow$ nichts
  - $P_2P_3 \rightarrow S_1$ und P_3
  - $P_3P_4 \rightarrow P_4$
  - $P_4P_1 \rightarrow S_2$
  - Kante S_2S_1 schließt das Polygon



Sutherland-Hodgeman Polygon Clipping

Clipping mit Attributen

- Vertex-Attribute (z.B. Texturkoordinaten oder Farben) werden mit und für die Schnittpunkte berechnet
- ist das Eingabepolygon konvex, dann auch das Ausgabepolygon
  - kann bei Bedarf einfach in Dreiecke zerlegt werden:
    Q_1, Q_i, Q_{i+1} mit $1 < i < n$



Clipping in 3D

- Clipping gegen ein konvexes Sichtvolumen
  - statt Clipping gegen Kanten: Clipping gegen Ebenen
  - α-Clipping in homogenen Koordinaten

OpenGL

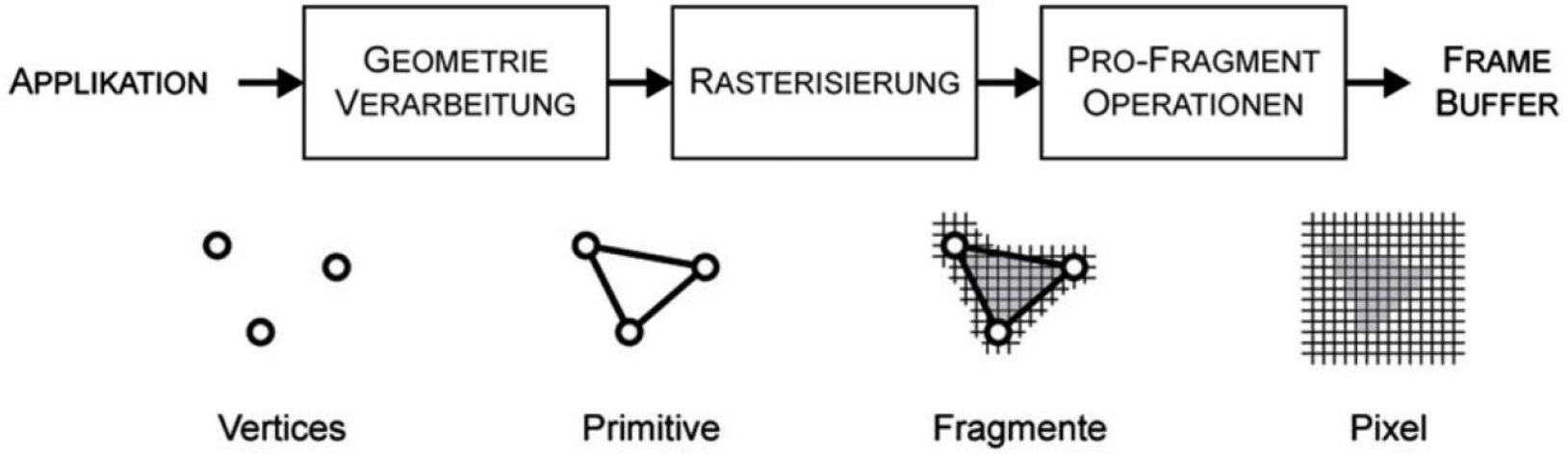
- Plattformunabhängige API zur Programmierung von Graphikhardware
- Statemachine
- Klassisches vs. modernes OpenGL

- Window- und Event-Handler: GLUT, GLFW, Qt, ...
  - Fenster erzeugen, Benutzereingaben behandeln, ...

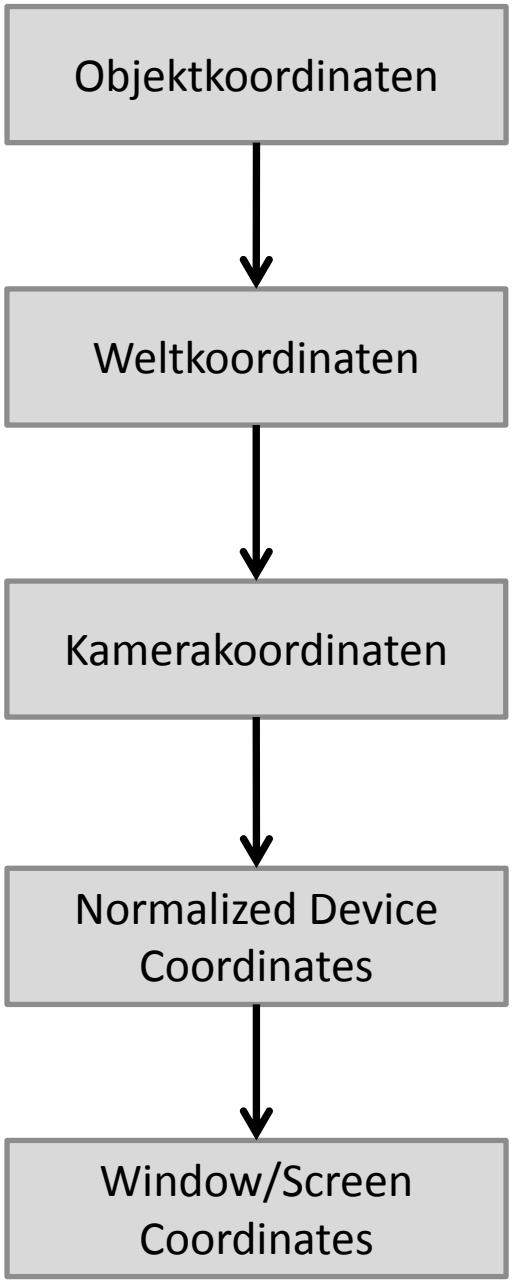
- Extension Manager: GLEW
  - Hilft beim Umgang mit dem OpenGL-Extension Mechanismus

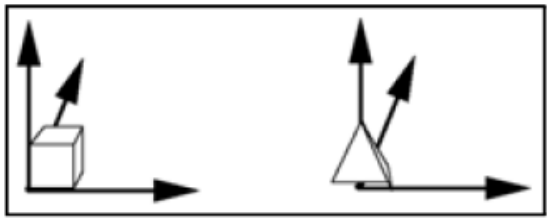
Rasterisierungspipeline

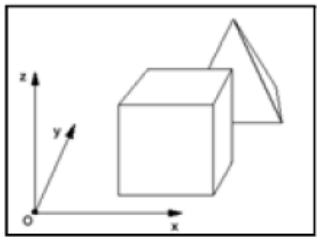
- Abbilden der Geometrie (Dreiecke) auf 2D Bildschirmkoordinaten
- Rasterisierung (siehe 1. Übungsblatt)
- Interpolation von Vertex-Attributen
- Tiefentest und andere Fragmentoperationen

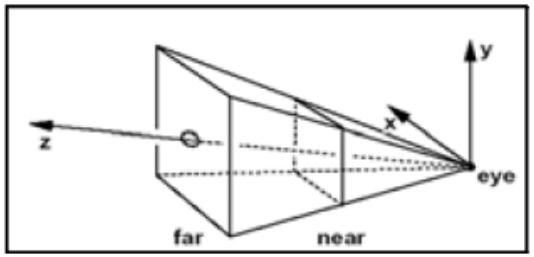


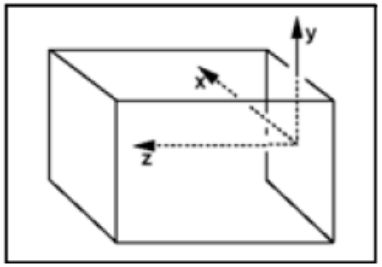
Koordinatensystem Pipeline











OpenGL

- Schematische Hauptschleife

```
display() {
        clearBuffers();
        setupCamera();
        drawScene();
        swapBuffers();
}


drawScene() {
        for each object {
                stateSetup();
                drawObject();
                stateClean();
        }
}
```

OpenGL – Immediate mode

- Matrix Stack
  - z.B. für die Model-View Transformtionen

```
glMatrixMode(GL_MODELVIEW);

glPushMatrix();
glTranslatef(40, 0, 30);
drawObject1();
glPopMatrix();

glPushMatrix();
glTranslatef(40, 0, -30);
drawObject2();
glPopMatrix();
```

OpenGL – Immediate mode

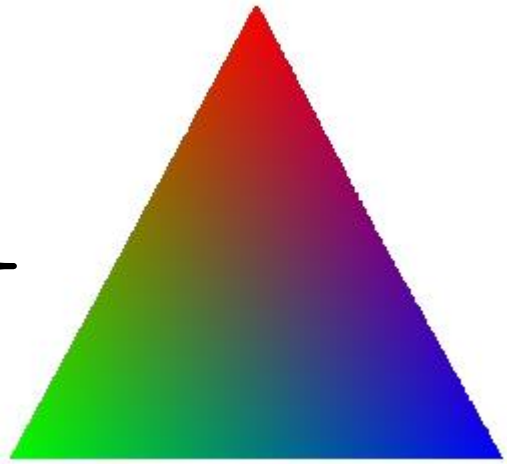
- Geometriedaten
  - Layout: GL_TRIANGLE, GL_TRIANGLE_STRIP, ...

```
drawObject() {
      glBegin(GL_TRIANGLES);
      glVertex3f(0, 0, 0);
      glVertex3f(1, 0, 0);
      glVertex3f(1, 1, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(1, 1, 0);
      glVertex3f(0, 1, 0);
      glEnd();
}
```

GLOpenGL – Immediate mode

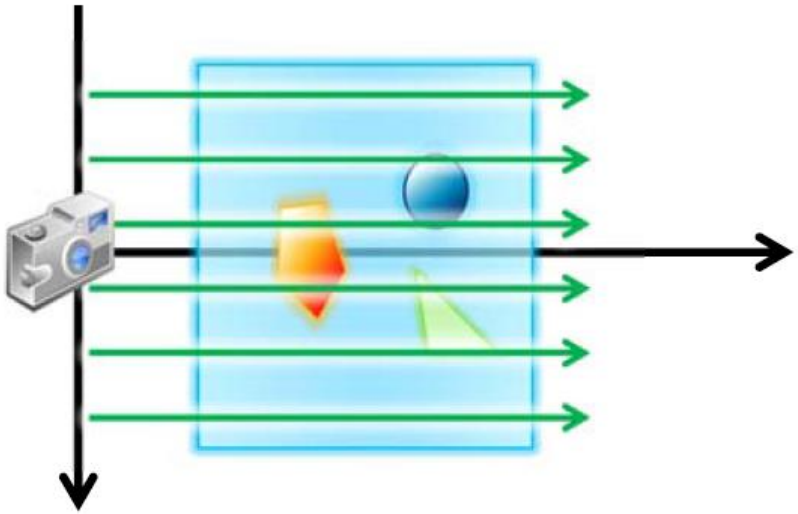
- Vertex-Attribute
  - Normale, Farbe, Texturkoordinate, ...

```
drawObject() {
      glBegin(GL_TRIANGLES);
      glColor3f (1, 0, 0);
      glNormal3f(0, 0, 1);
      glVertex3f(1, 2, 0);
      glColor3f (0, 1, 0);
      glNormal3f(0, 0, 1);
      glVertex3f(0, 0, 0);
      glColor3f (0, 0, 1);
      glNormal3f(0, 0, 1);
      glVertex3f(2, 0, 0);
      glEnd();
}
```

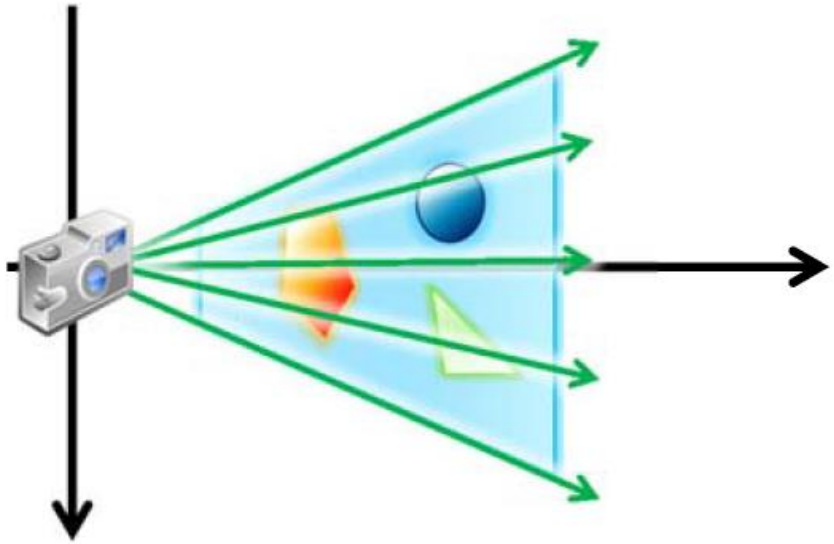


Projektionen

- Abbilden der Geometrie (Dreiecke) auf 2D Bildschirmkoordinaten
- Mehrere Möglichkeiten
  - Orthographisch
  - Perspektivisch



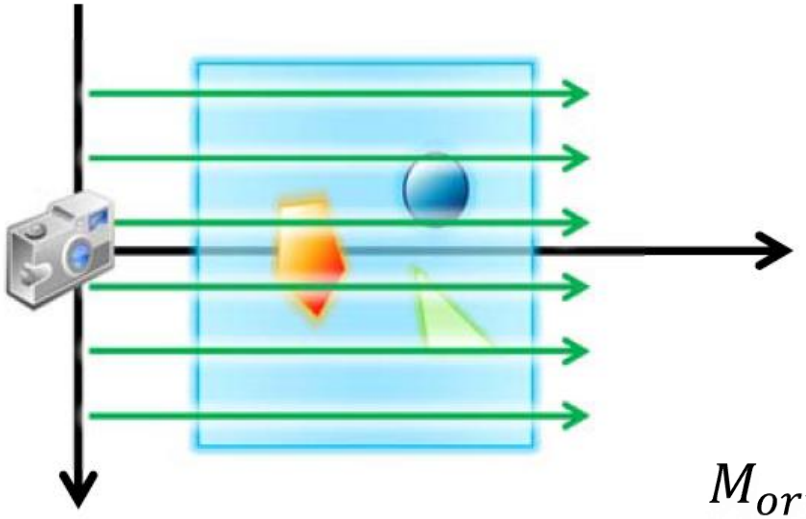
Orthographische Kamera:
Sichtvolumen ist ein Quader



Perspektivische Kamera:
Sichtvolumen ist ein Pyramidenstumpf

Projektionen

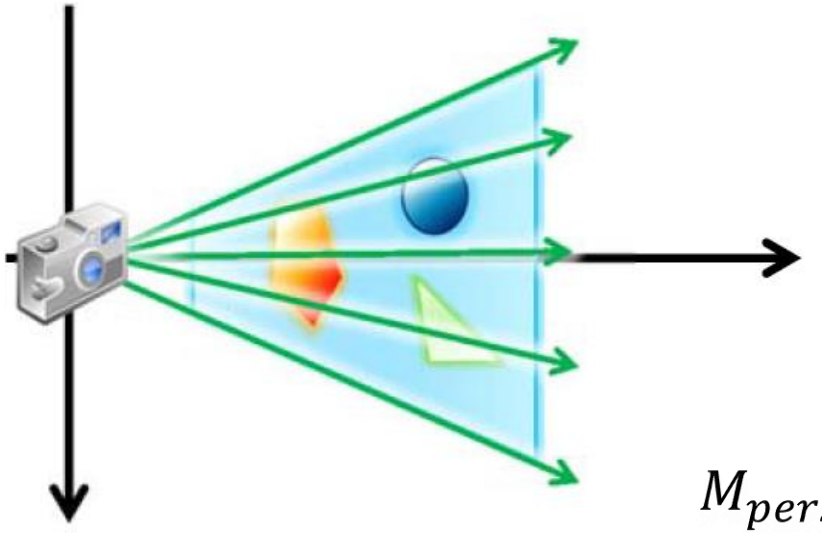
- Abbilden der Geometrie (Dreiecke) auf 2D Bildschirmkoordinaten
- **Zwischenschritt:** Abbildung des Sichtvolumens auf den Einheitswürfel
- Orthographisch: Einfach!
- Sichtvolumen: $[r, l] \times [t, b] \times [n, f] \rightarrow [-1,1]^3$



$$M_{orth} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Projektionen

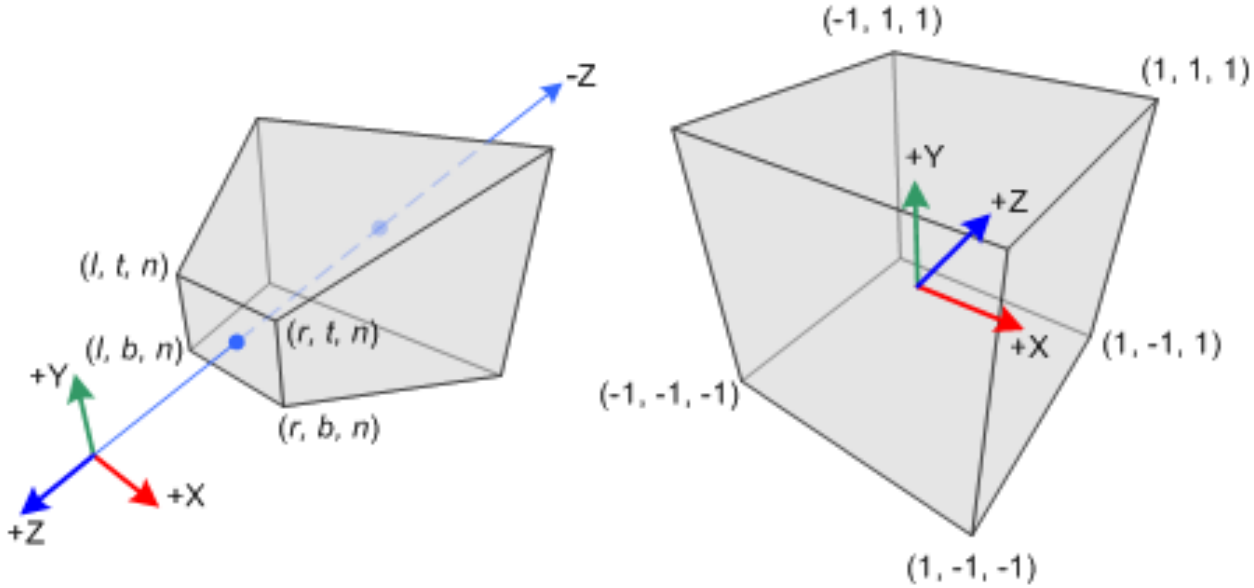
- Abbilden der Geometrie (Dreiecke) auf 2D Bildschirmkoordinaten
- **Zwischenschritt:** Abbildung des Sichtvolumens auf den Einheitswürfel
- Perspektivisch: Projektionszentrum im Ursprung
- Sichtvolumen definiert durch: $\mathrm{r}, l, b, t, n, f$



$$M_{pers} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Projektionen

- Multiplikation mit M_{pers}/M_{orth} und anschließender Dehomogenisierung beschreibt die Abbildung „Sichtvolumen → Kanonisches Volumen“
- Die eigentliche Projektion ist dann das Weglassen der z-Komponente
- Letzter Schritt: Viewport Transformation
  - $[-1,1]^2 \rightarrow [0, screenWidth] \times [0, screenHeight]$



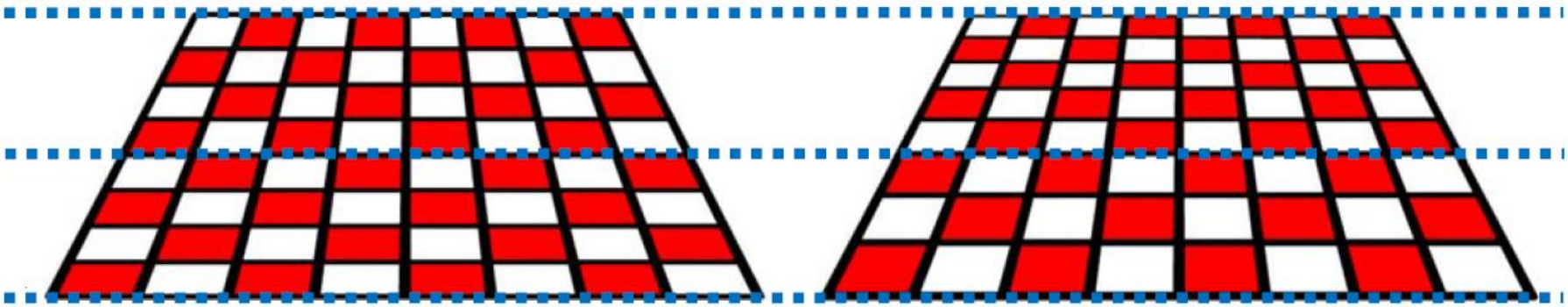
Perspektivisch-korrekte Interpolation

- Naive (affine) Interpolation im Bildraum führt zu falschen Ergebnissen bei perspektivischen Projektionen
- Richtige Tiefeninterpolation:

$$\frac{1}{z_t} = \left(s\frac{1}{z_1} + (1-s)\frac{1}{z_2} \right)$$

- Allgemein:

$$\frac{I_t}{z_t} = \left(s\frac{I_1}{z_1} + (1-s)\frac{I_2}{z_2} \right)$$



affin perspektivisch korrekt

Z-Buffer

- Entscheide Sichtbarkeit durch Speichern der Distanz des nahesten Objekts pro Pixel