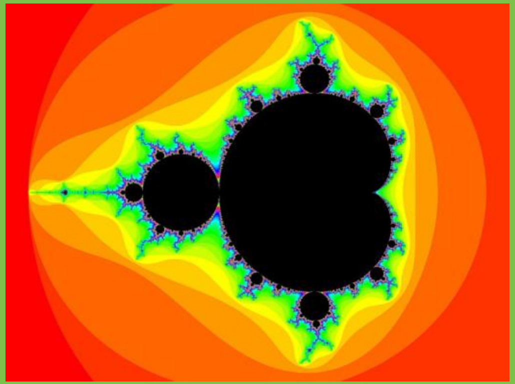
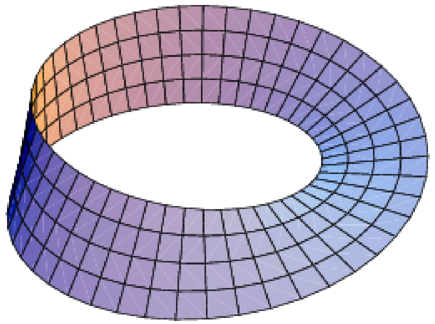
Visual Computing – Grundlagen 2D-/3D-Grafik

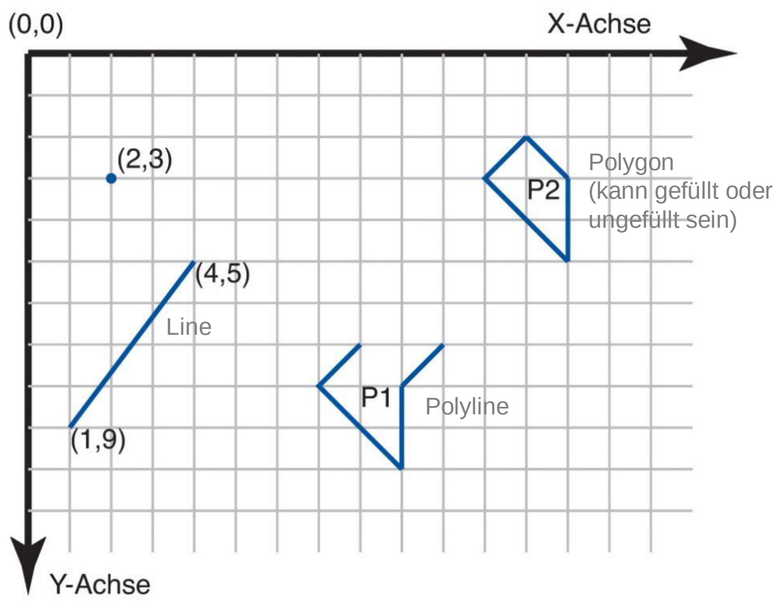


Yvonne Jung



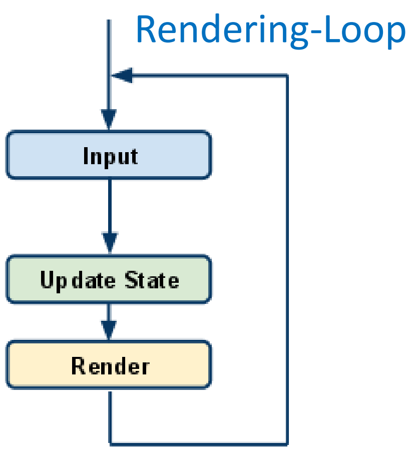
2D-Graphik mit Qt

MouseEvents und PaintEvent



Graphische Ausgabe

- Anwendung muss für sinnvolles Update sorgen
  - Nur sie weiß, was geschehen soll, wenn keine Standard-Widgets verwendet werden (z.B. bei Malprogramm oder 3D-Renderer)
    - Betriebssystem kann Ausschnitte, die auf Desktop sichtbar werden, nur primitiv rekonstruieren (indem Pixel kopiert werden)
    - Wird z.B. nötig beim Verschieben oder Vergrößern von Fenstern
- Anwendung muss Inhalt selbst verwalten und aktualisieren
  - Betroffene Klassen müssen Zeichen-Methode neu implementieren
    - Bei Änderungen neu zeichnen (wichtig: zu zeichnende Objekte verwalten)
- Darstellung aktualisieren via *update()*
  - Zustandsänderungen i.d.R. durch Mouse-/KeyEvents oder Timer-gesteuerte Animationen
    - Danach Neuzeichnen (d.h. Rendern) triggern
    - Neuzeichnen in Qt kann/sollte nur zentral im *paintEvent()* geschehen



Zeichnen mit Qt

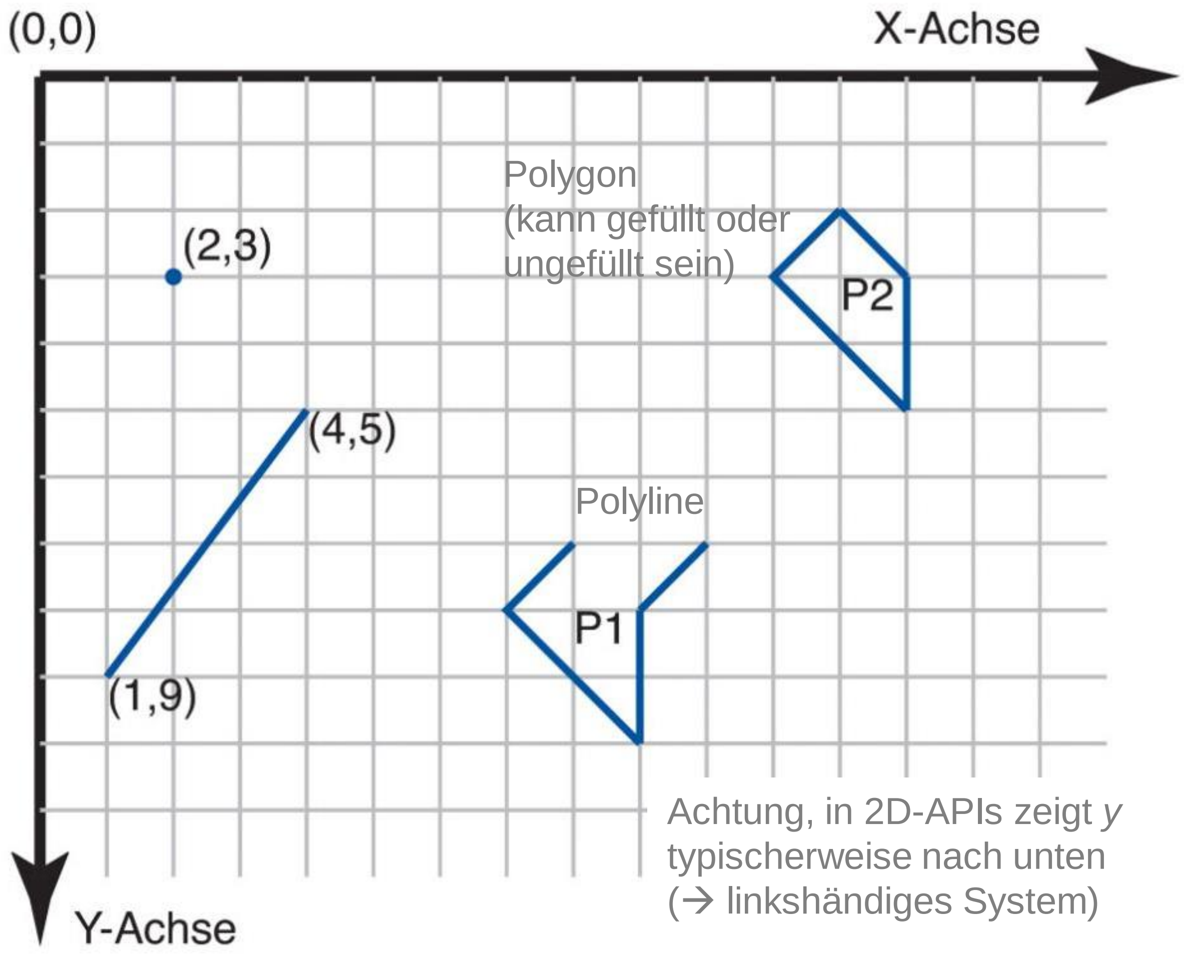
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Geschieht <u>immer</u> in Methode *paintEvent()*
- Zwei mögliche Herangehensweisen
  - Direkt mit Instanz von *QPainter*
    - QWidget ist (wie QImage) ein QPaintDevice
    - Kann damit über QPainter „bemalt“ werden
    - Zeichenbereiten QPainter für Widget erzeugen:
      - QPainter painter(this); // nur in paintEvent() deklarierbar
      - painter.setPen(Qt::red); // setzt Linienfarbe auf Rot
  - Mit Hilfe von *QPainter* und *QPainterPath*
    - QPainterPath für komplexere Zeichenoperationen
      - Hat Methoden moveTo() und lineTo(), wobei Parameter je 2D-Punkt vom Typ QPoint ist
      - painter.drawPath(*pp*); // Parameter ist zu zeichnender QPainterPath

QWidget ist Unterklasse von QPaintDevice

Abgeleitete Klassen können daher mit Hilfe von QPainter benutzerdefiniert zeichnen

2D-Objekte: Punkt, Linie, Polygon

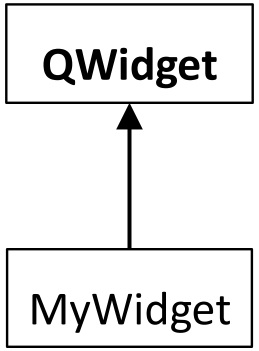


Wichtige Methoden der Klasse QPainter:

- void drawEllipse(const QPoint &center, int rx, int ry)
  - Sind Halbachsen *rx* und *ry* gleich, wird Kreis mit Radius *rx* um Punkt *center* gezeichnet
- void drawLine(int x1, int y1, int x2, int y2)
- void drawLine(const QPoint &p1, const QPoint &p2)
- void drawRect(int x, int y, int width, int height)
- void drawPolyline(const QPoint *points, int pntCnt)
  - Parameter *pntCnt* gibt Anzahl der zu betrachtenden Eckpunkte aus Array *points* an
- void drawPolygon(const QPoint *points, int pntCnt, Qt::FillRule fillRule = Qt::OddEvenFill)
- void drawPath(const QPainterPath &path)

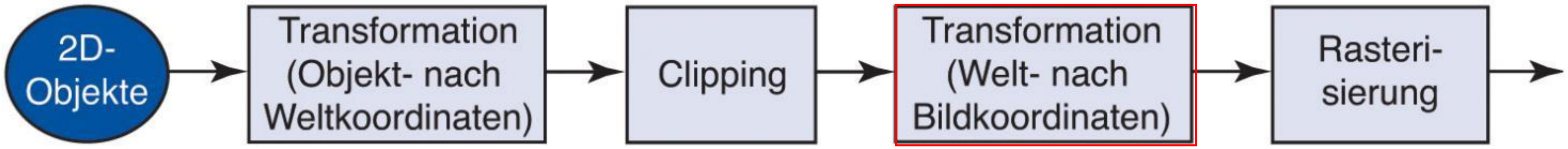
Grafik in Qt aktualisieren

```
void MyWidget::paintEvent(QPaintEvent *event) {
   // 2D-Bild basierend auf Mausinteraktion darstellen. Dazu in Mausevents jeweils
   // Mauspositionen merken und update() aufrufen, dann hier Bild aktualisieren
}
void MyWidget::mousePressEvent(QMouseEvent *event) {
   if (event->button() == Qt::LeftButton) {
      // Aktuelle Mausposition in event->pos(), diese ggfs. merken und aktualisieren:
      update();
   }
}
void MyWidget::mouseMoveEvent(QMouseEvent *event) {
   // Aktuelle Mausposition merken und aktualisieren
}
void MyWidget::mouseReleaseEvent(QMouseEvent *event) {
   // Aktuelle Mausposition merken und aktualisieren
}
```



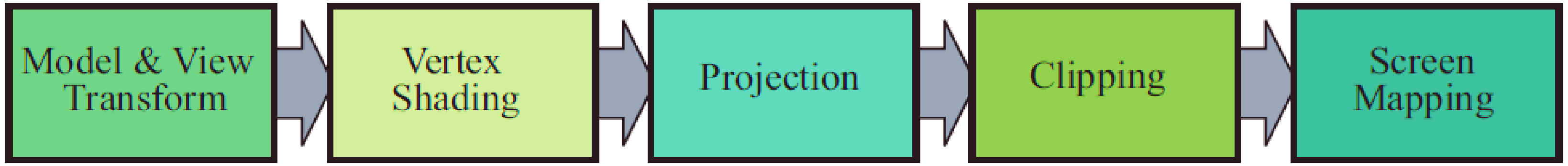
MyWidget sei eigene Klasse
und von *QWidget* abgeleitet

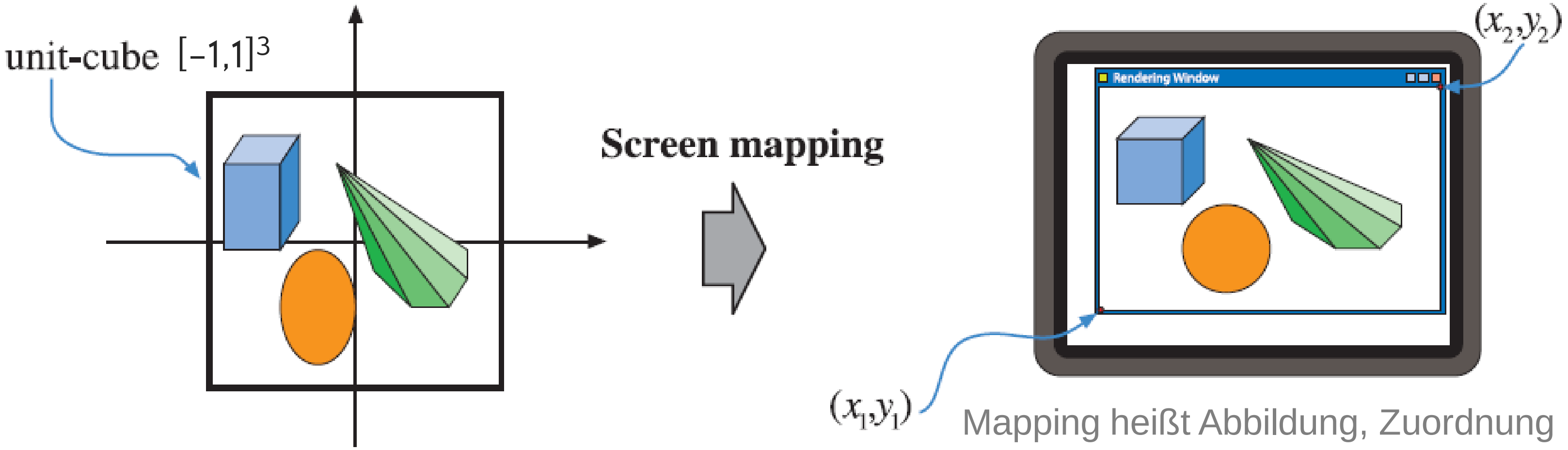
Die 2D-Rendering-Pipeline



Von Problemkoordinaten zu Gerätekoordinaten

Screen Mapping von Koordinaten



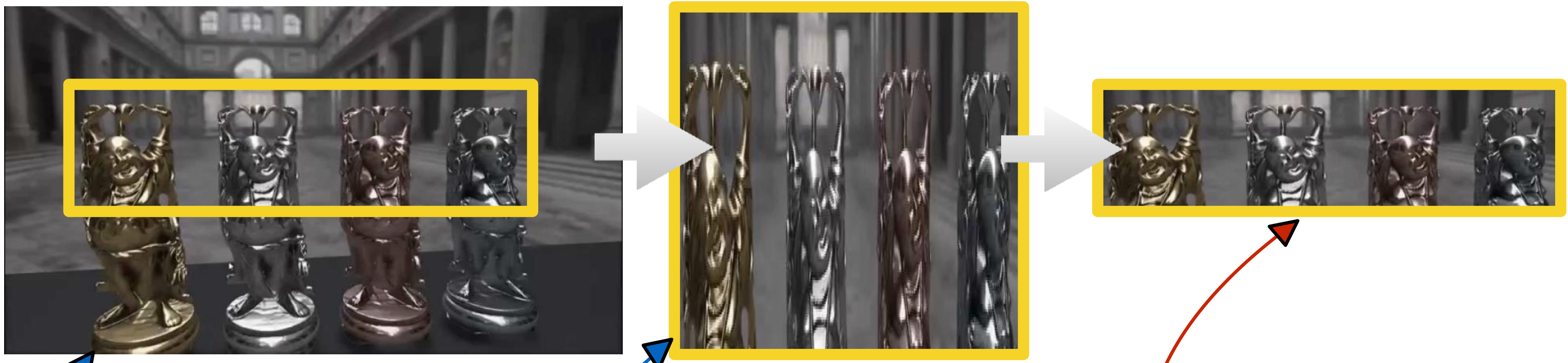


Mapping heißt Abbildung, Zuordnung

Ausblick: Einordnung in Verarbeitungspipeline bei 3D-Rendering

Screen Mapping

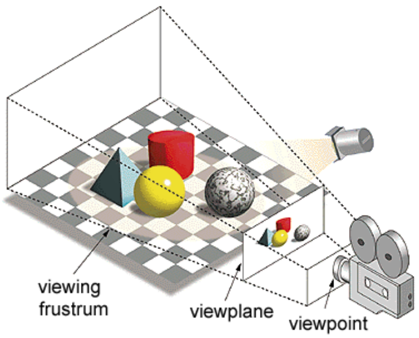
- Beispiel zur sog. Viewport-Transformation: von der 3D-Szene zum 2D-Bild



Definition des zu rendernden Sichtvolumens

Skalierung der Szene, so dass dieser Bereich im Einheitswürfel $[-1,1]^3$ liegt

Viewport-Transformation transformiert Grafik in Fensterkoordinaten



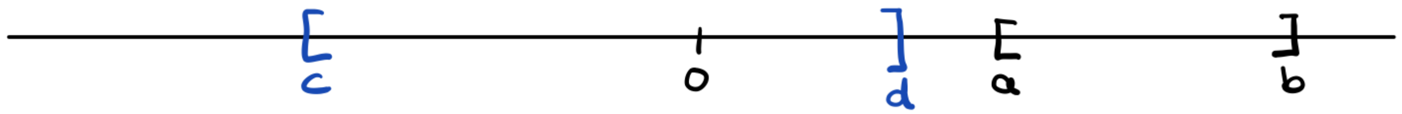
Auflösungsunabhängig

Abhängig von Auflösung des Ausgabemediums

Abbildungen / Transformationen

- Grundsätzliches Problem: Umrechnungen zw. unterschiedlichen Wertebereichen
- Bsp.: Abbildung von Bereich [a, b] auf [c, d]

Geg.: $a, b, c, d \in \mathbb{R}$



Ges.: $[a, b] \xrightarrow{\text{Umrechnung ?}} [c, d]$

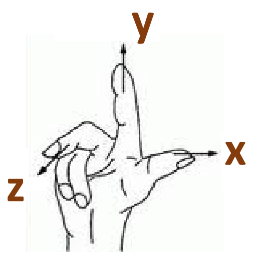
Lösungsansatz: je die Grenzen betrachten

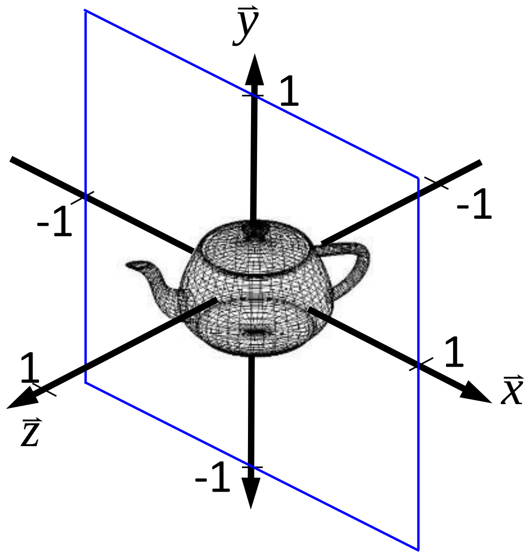
$$[a, b] \xrightarrow{-a} [0, b-a] \xrightarrow{:(b-a)} [0, 1]$$

$$\xrightarrow{\cdot(d-c)} [0, d-c] \xrightarrow{+c} [c, d]$$

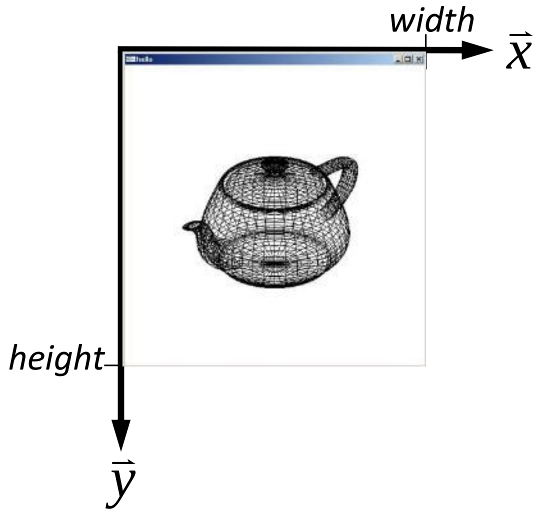
Transformation von 2D-Welt-/Problem-koordinaten in Bild-/Gerätekoordinaten

- Anwendungen beschreiben 2D-/ 3D-/ *n*D-Daten in anwendungsspezifischen Einheiten (z.B. Meter)
  - Y-Achse zeigt nach oben



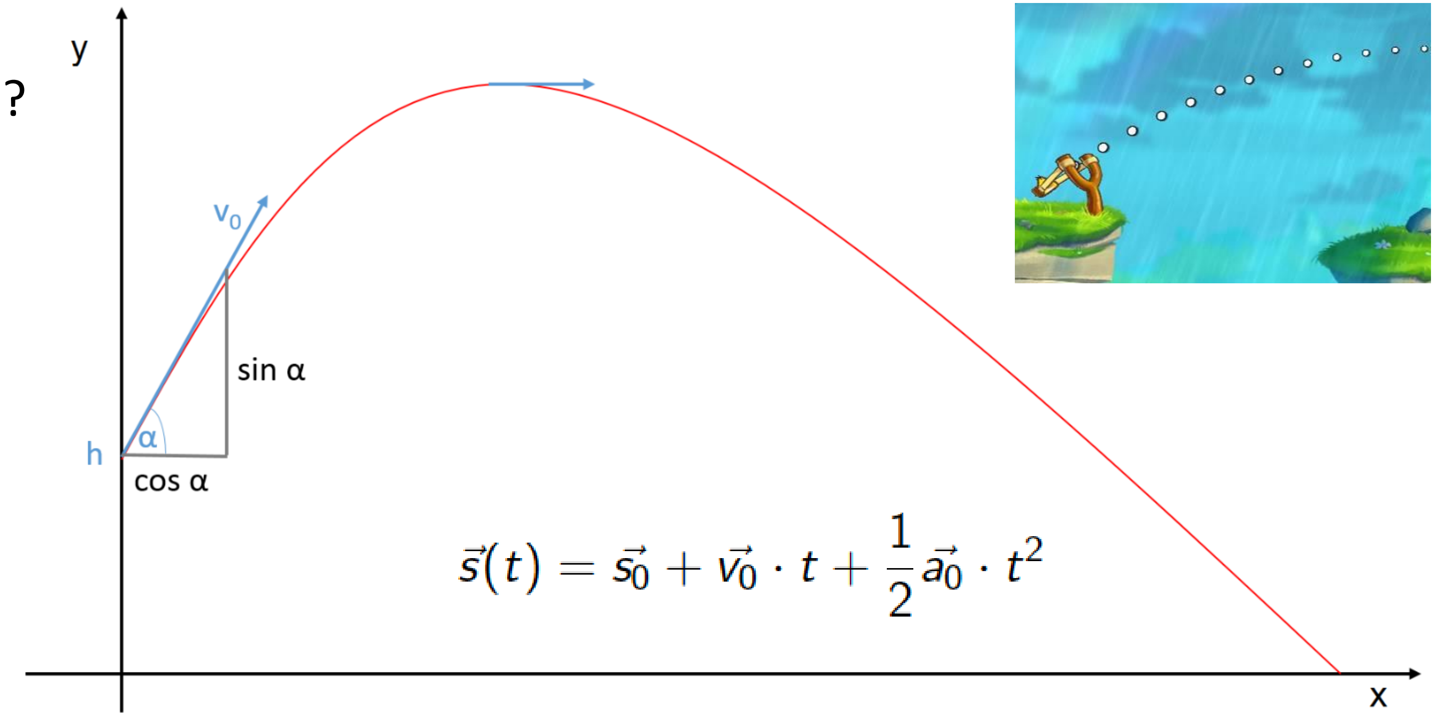


- Ausgabegeräte arbeiten aber nur auf 2D-Pixelkoordinaten
  - Dabei Wechsel von rechts- zu linkshändigem Koordinatensystem
  - Y-Achse zeigt nach unten



Von Welt- zu Bildkoordinaten?

- Beispiel Wurfparabel (schräger Wurf)
  - Abschuss/Wurf von Raketen, Bällen, Vögeln…
  - Modellierung mit Hilfe der Weg-Zeit-Funktion
    - Darstellung auf Bildschirm?

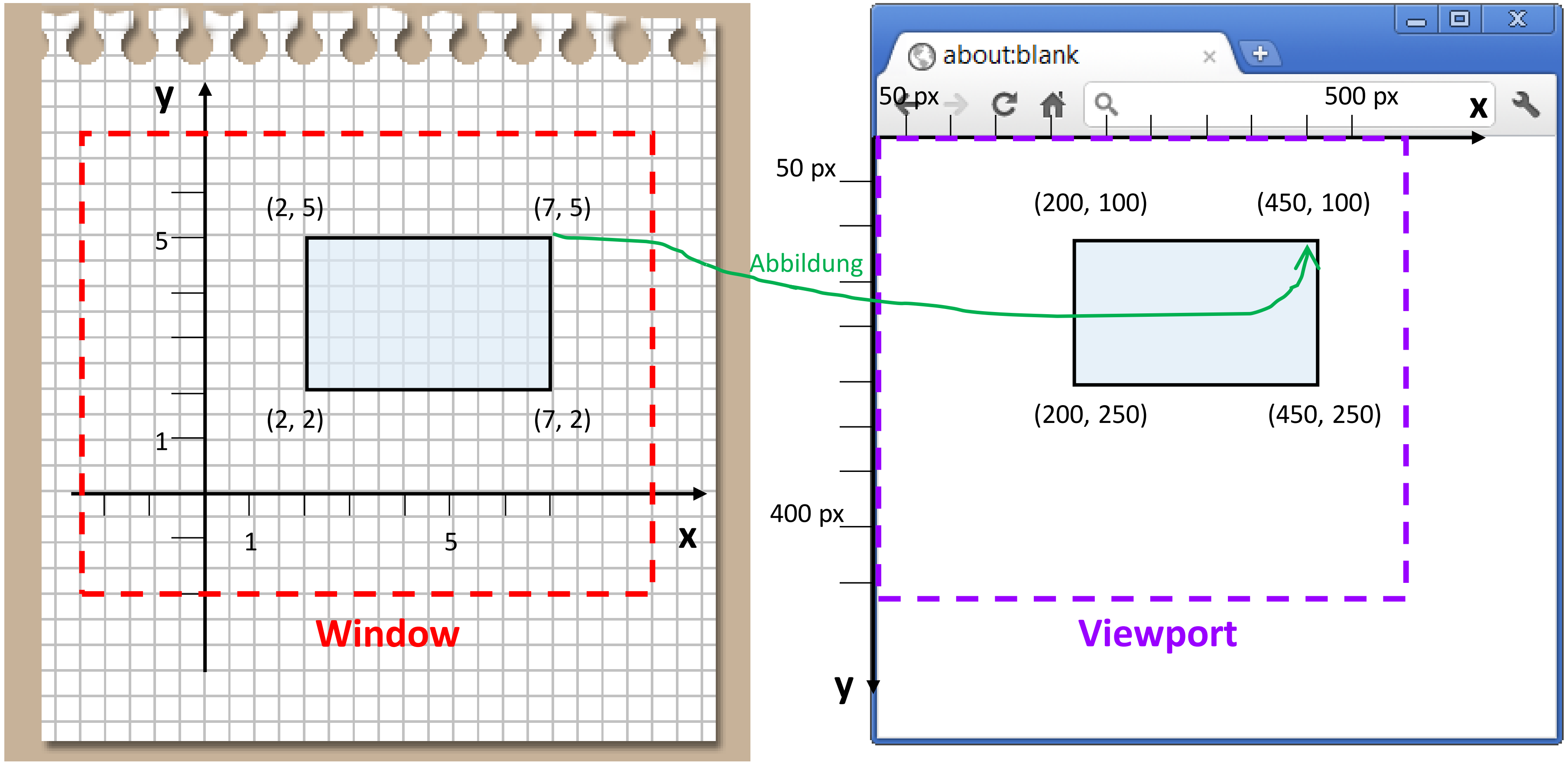


$$\vec{s}(t) = \vec{s_0} + \vec{v_0} \cdot t + \frac{1}{2}\vec{a_0} \cdot t^2$$

Window-Viewport Transformation

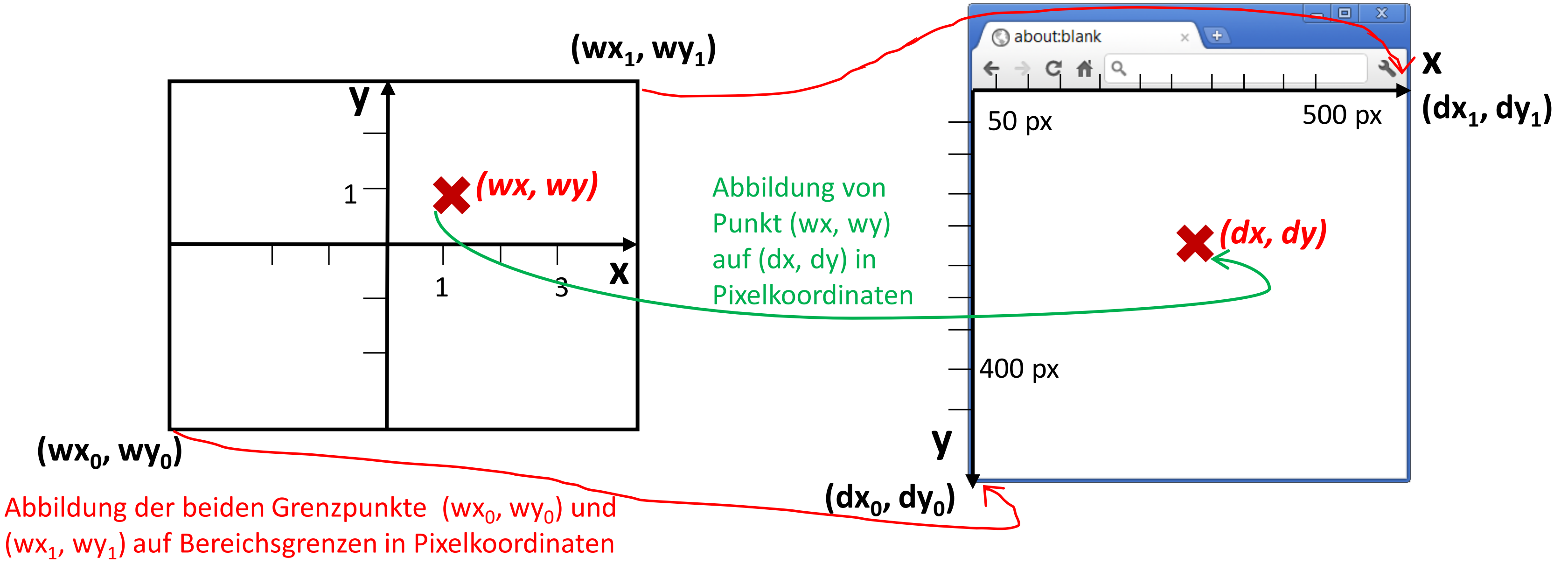
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Idee: Abbildung zweier rechteckiger Bereiche aufeinander
- Seitenverhältnisse von Window und Viewport i.A. verschieden



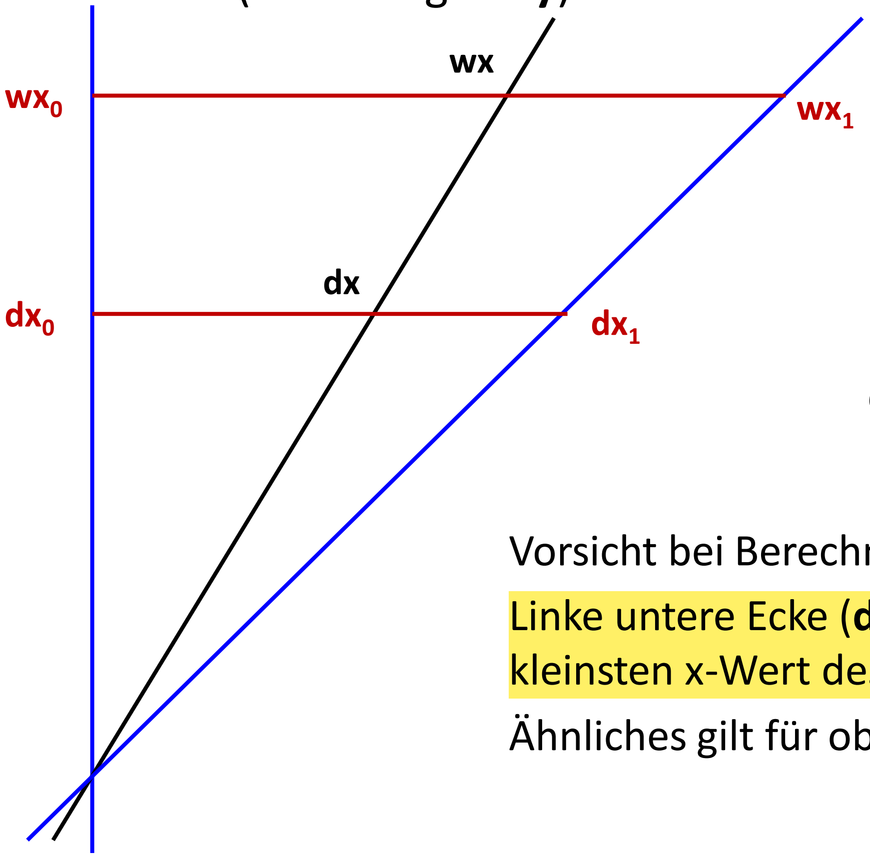
Window-Viewport Transformation

- Window-Viewport Transformation bildet Problem-Koordinaten (geg. in anwendungsspezifischen Einheiten und Maßstäben) ab auf Geräte-Koordinaten (in Integer-wertige Pixel-Koordinaten)
  - Achtung: dies bedeutet i.d.R. Wechsel von Rechts- auf Linkshändiges Koordinatensystem
  - Eigenschaften: achsparallele Transformationen, aber Verzerrungen möglich
- Beispiel (für *x*): Bereich **[wx_0, wx_1]** soll abgebildet werden auf **[dx_0, dx_1]**
  - Analog für y (Vorsicht, **dy_0** > **dy_1** wegen geflippter y-Koordinate)



Window-Viewport Transformation

- Gegeben: $\mathbf{dx_0}$, $\mathbf{dx_1}$, $\mathbf{wx_0}$, $\mathbf{wx_1}$ und **wx**
- Gesucht: **dx** (→ analog für **y**)



Herleitung z.B. mittels Strahlensatz

$$\frac{wx - wx_0}{wx_1 - wx_0} = \frac{dx - dx_0}{dx_1 - dx_0}$$

$$dx = \frac{wx - wx_0}{wx_1 - wx_0} \cdot (dx_1 - dx_0) + dx_0$$

Vorsicht bei Berechnung von **dy**:

Linke untere Ecke ($\mathbf{dx_0}$, $\mathbf{dy_0}$) des Viewports in Geräte- bzw. Pixelkoordinaten hat zwar kleinsten x-Wert des betrachteten Ausschnitts, aber größten y-Wert!

Ähnliches gilt für obere rechte Ecke des Viewports…

Window-Viewport Transformation

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Umformen zeigt, dass Formel aus Skalierung (also Streckung oder Stauchung) und Translation (also Verschiebung) besteht

$$dx = \underbrace{\frac{dx_1 - dx_0}{wx_1 - wx_0}}_{\text{Skalierung}} \cdot wx + \Bigg(\underbrace{dx_0 - \frac{wx_0(dx_1 - dx_0)}{wx_1 - wx_0}}_{\text{Translation}}\Bigg)$$

$$dx = S \cdot wx + T$$

  – Skalierung *S* und Translation *T* i.d.R. für *x* u. *y* je verschieden (→ damit i.A. nicht uniforme Skalierung; Verzerrungen also möglich)

- Allgem. vektorielle Darstellung
  – Translation um Vektor $(t_x, t_x)^T$:

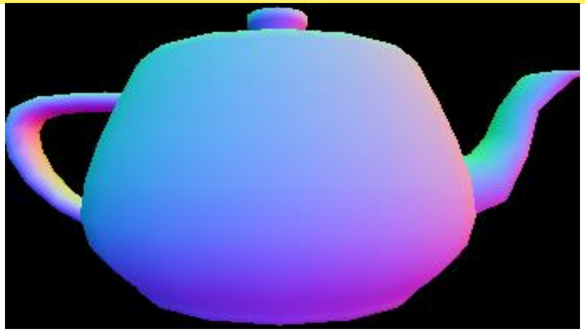
$$\begin{pmatrix} x_{neu} \\ y_{neu} \end{pmatrix} = \begin{pmatrix} x_{alt} \\ y_{alt} \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = \begin{pmatrix} x_{alt} + t_x \\ y_{alt} + t_y \end{pmatrix}$$

  – Skalierung um Faktoren s_x, s_y:

$$\begin{pmatrix} x_{neu} \\ y_{neu} \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x_{alt} \\ y_{alt} \end{pmatrix} = \begin{pmatrix} s_x x_{alt} \\ s_y y_{alt} \end{pmatrix}$$

Beispiel: Abbildungen im Bildraum

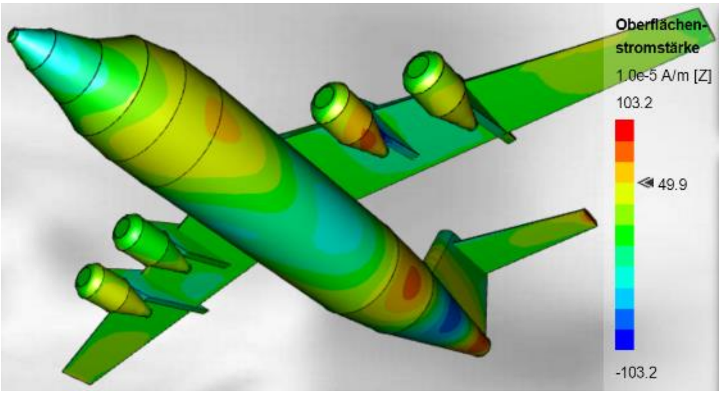
- Oberflächen-Normalen als Farbe anzeigen (für VFX oder Debugging)



$$[-1; 1]^3 \longrightarrow [0; 1]^3$$

Voraussetzung: Normalenvektoren sind normiert, x-/y-/z-Komponente liegt also zwischen -1 und 1

- Transferfunktion bzw. Lookup-Table (LUT)



z.B. Temperatur als Farbe

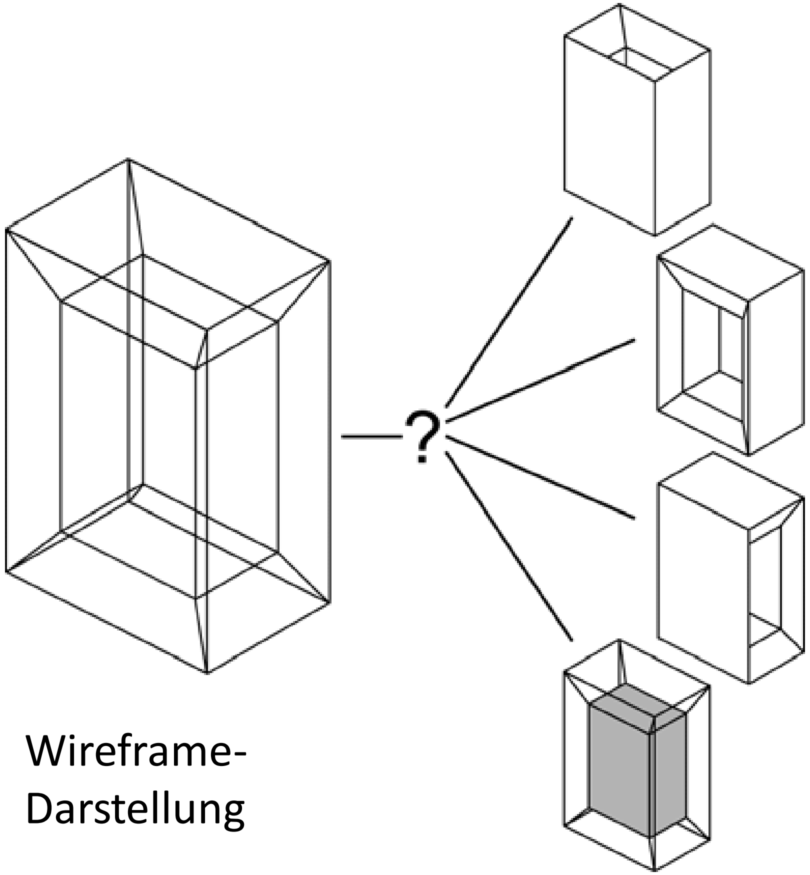
$$[a; b] \longrightarrow [0; 1]$$

Wert als Texturkoordinate nehmen → in 1D-Textur

Farbcodierung: Farben als "Skala" für physikalische Werte

Übung 1

- Sie möchten Bereich [-1, 1] abbilden auf [0, 1], um Richtungsvektoren als RGB-Farben abzuspeichern
  - Wie würden Sie rechnen, auf welchen Wert würde Vektor $(0.2, 0.4, -0.8)^T$ abgebildet werden?
  - Wie würden Sie für einen vorzeichenlosen 8-Bit Datentypen das Ergebnis noch umrechnen?
- Sie möchten den Problemkoordinatenbereich von (-2, -4) bis (10, 4) abbilden auf Pixelkoordinaten zwischen (40, 40) und (640, 440)
  - Wie lauten die Pixelkoordinaten für Punkt P(3, -1)?
- Ist auch Rücktransformation der Window-Viewport-Transformation nötig?



Wireframe-
Darstellung

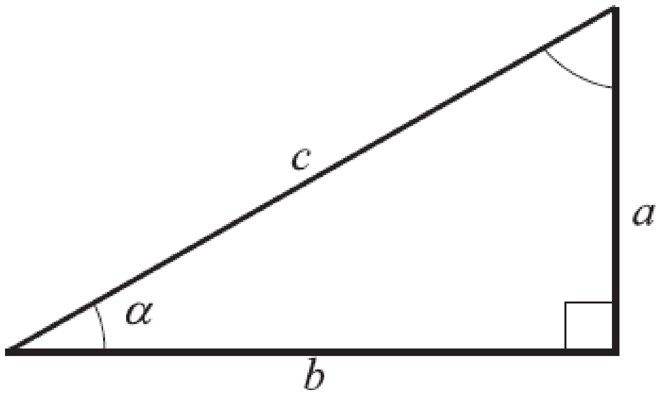
Geometrie-Erzeugung

Geometrische Formen erzeugen

Wdh.: Trigonometrie

$$\sin a = \frac{a}{c}$$

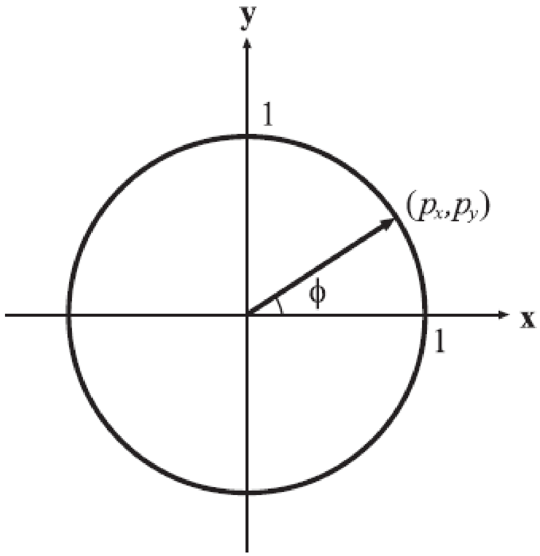
$$\cos a = \frac{b}{c}$$

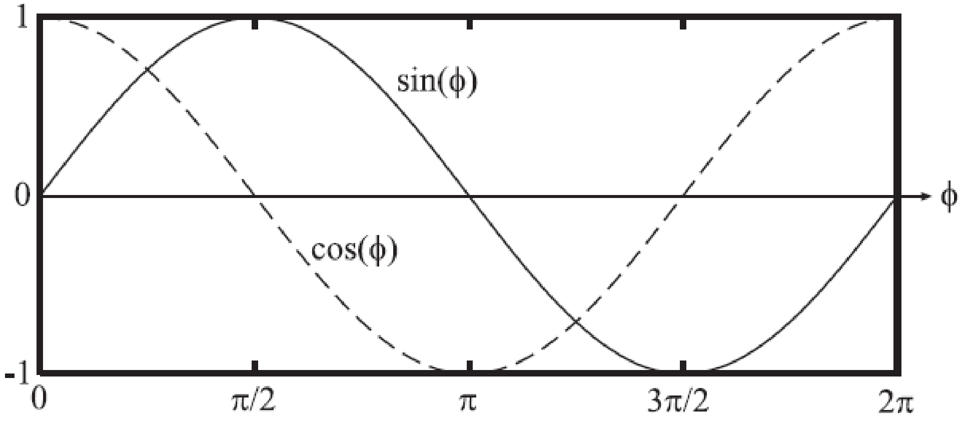


$$\cos(\alpha) = \sin(\alpha + \frac{\pi}{2})$$

$$\cos(\alpha) = \cos(-\alpha)$$

$$\sin(-\alpha) = -\sin(\alpha)$$





Funktionsplotter für Sinuskurve

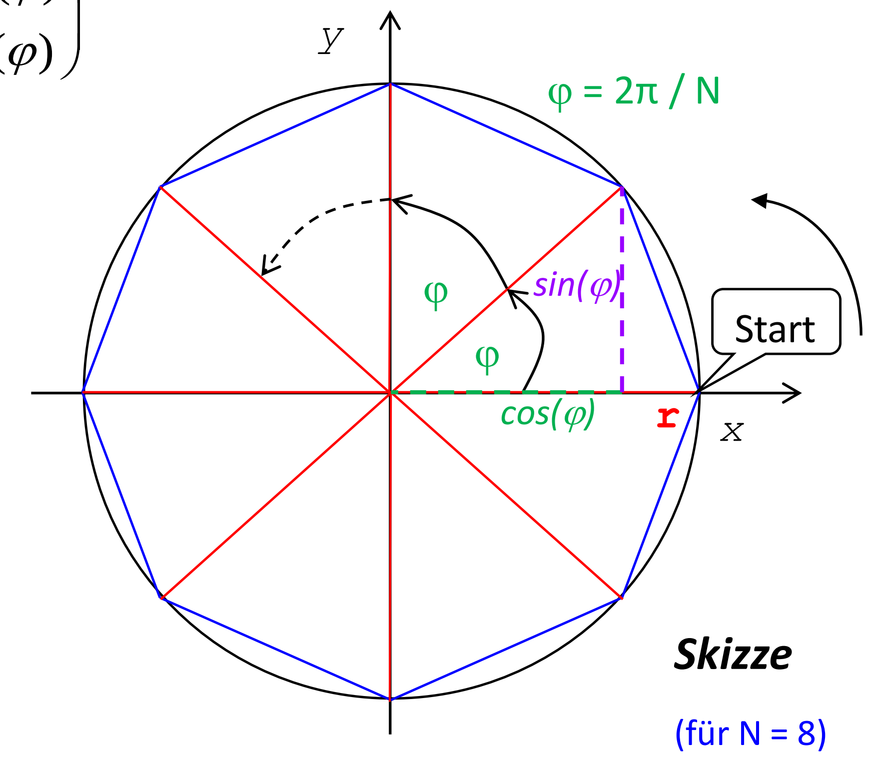
```
QPainter painter(this);
QPainterPath pp;
QPoint dc;

const int N = 32;
float x, y, dx;

x = wMin.x();
y = sin(x);

dc = WC_to_DC(QPointF(x, y));
pp.moveTo(dc);
```

```
dx = (wMax.x() - wMin.x()) / N;

for (int i = 0; i < N; i++) {
    x += dx;
    y = sin(x);

    dc = WC_to_DC(QPointF(x, y));
    pp.lineTo(dc);
}

painter.setPen(Qt::red);
painter.drawPath(pp);
```

Kreiskurve in Vektordarstellung

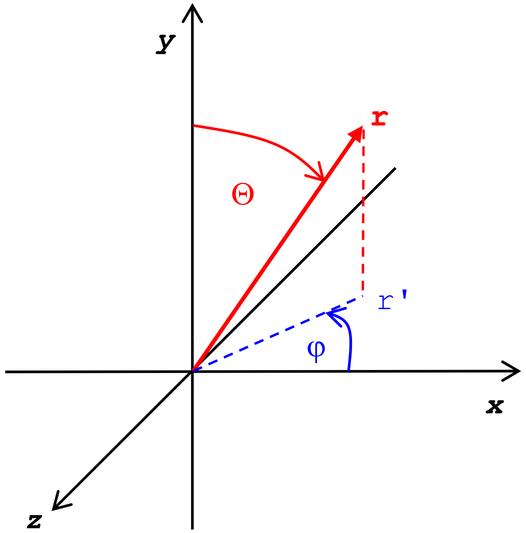
- Kreis mit Radius *r* um Punkt *p* mittels Sinus und Kosinus zeichnen
  - Kreisgleichung: $(x - p_x)^2 + (y - p_y)^2 = r^2$
  - Parameterdarstellung für Kreis mit Radius r: $\vec{s}(\varphi) = r \cdot \begin{pmatrix} \cos(\varphi) \\ \sin(\varphi) \end{pmatrix}$

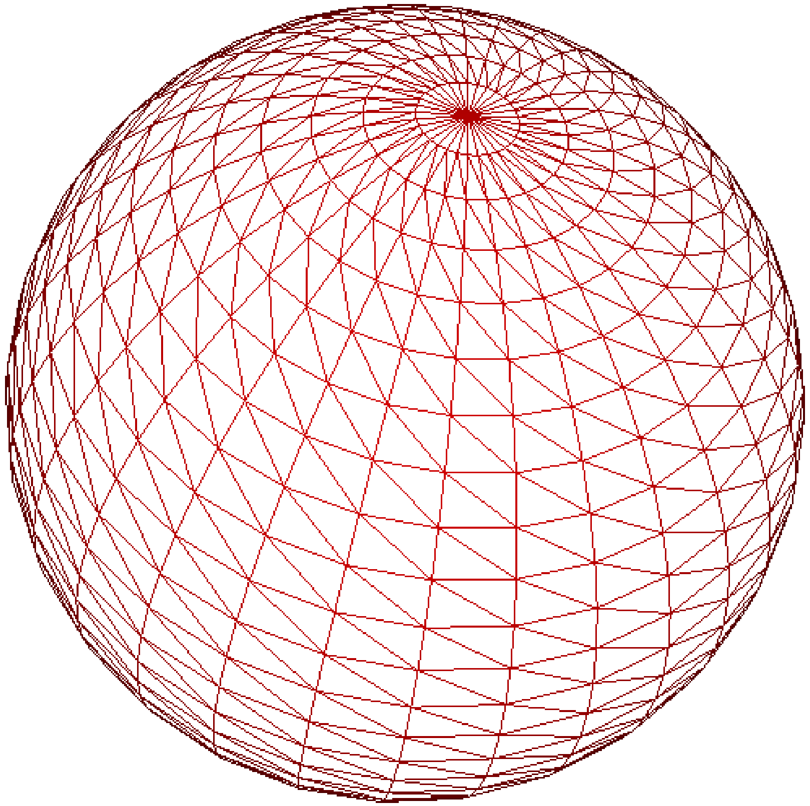
```
for (i=0; i<=N; i++) {
    phi = i * (2 * M_PI / N);
    x = radius * cos(phi) + p.x();
    y = radius * sin(phi) + p.y();

    //convert to pixel coords
    dc = WC_to_DC(QPointF(x, y));

    if (i == 0)
        pp.moveTo(dc);
    else
        pp.lineTo(dc);
}
```



Von 2D zu 3D: Kugel

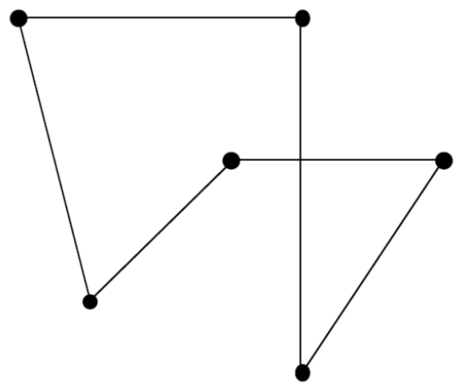
- Berechnung (analog zu Kreis) über Kugelkoordinaten (mit Radius *r*)
  - $$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = r \cdot \begin{pmatrix} \cos\varphi \cdot \sin\Theta \\ \cos\Theta \\ \sin\varphi \cdot \sin\Theta \end{pmatrix}$$
  - Komplexere graphische Objekte werden nur einmalig berechnet oder geladen
  - Gespeichertes Vertex-Array mit Adjazenz-Information wird zum Rendern genutzt



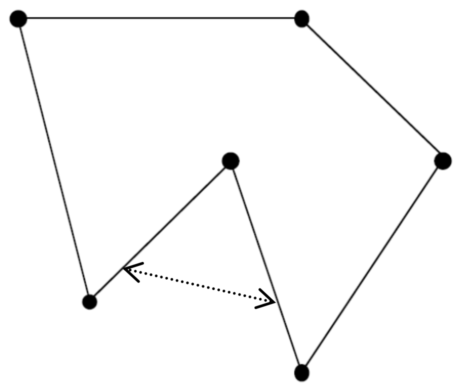


Polygone

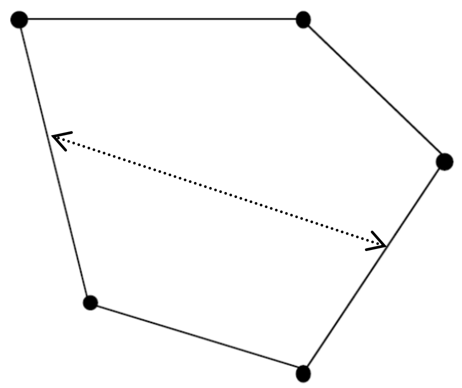
Polygon (*n*-Eck)



konkaves Polygon



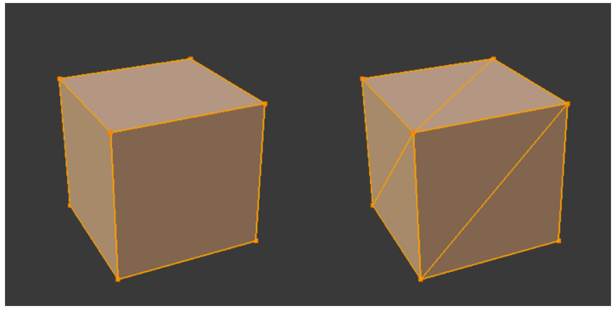
konvexes Polygon

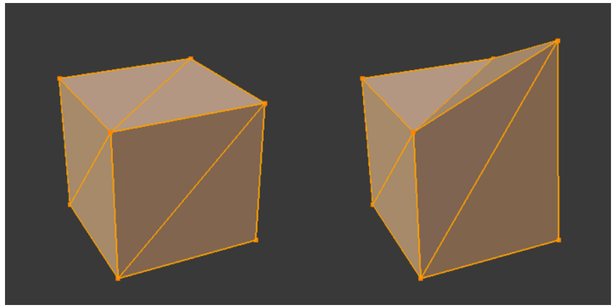


- Einfaches Polygon: Kanten schneiden sich **nicht** (→ gewünschte Form)
- Konvexes Polygon: Verbindungsstrecke zwischen zwei Punkten innerhalb der Fläche liegt ebenfalls in Fläche
  - Bzw. jeder Innenwinkel ≤ 180° (Aufpassen bei Kollinearität)
  - Konkav: nicht alle Linien zwischen Punkten der Fläche liegen in Fläche

Flächenmodelle

- Beschreibung der Oberfläche eines 3D-Objekts als Polygonnetz (Mesh)
  - Sollte geschlossen u. zusammenhängend sein → Boundary Representation (B-Reps)
- Meshes bestehen aus…
  - Geometrie: Position/Lage der Eckpunkte
    - Gerade (oder selten krummlinige) Verbindung benachbarter Punkte
  - Topologie: Graph
    - Nachbarschaftsbeziehungen von Punkten (Vertices), Kanten (Edges), Flächen (Faces)
    - Links: Gleiche Geometrie, unterschiedliche Topologie (mehr Kanten)
    - Rechts: Gleiche Topologie, unterschiedliche Geometrie (kein Würfel)

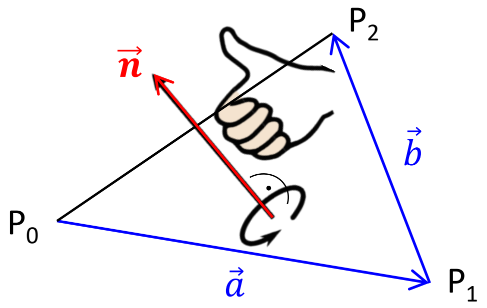


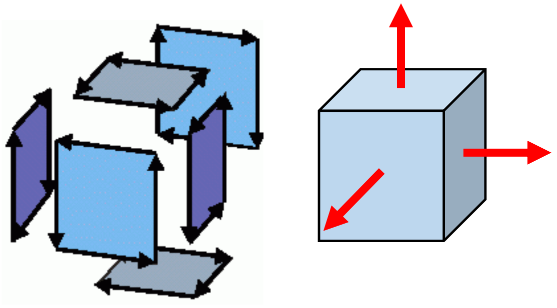


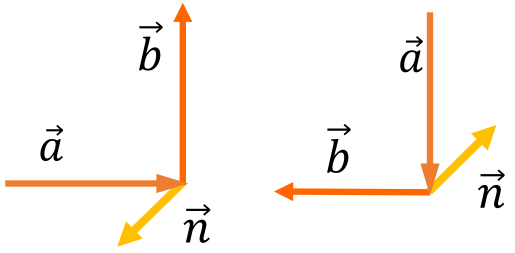
Oberflächennormalen

- Reihenfolge, in der Punkte in Polygon durchlaufen werden, ist relevant (Umlaufsinn)

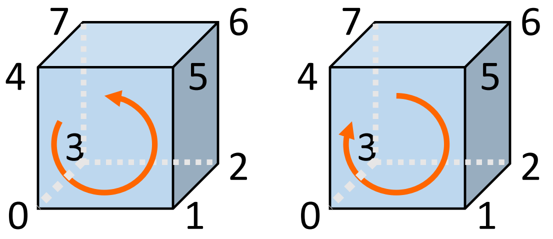
$\vec{a}$ = P_1 - P_0
$\vec{b}$ = P_2 - P_1
$\vec{n}$ = $\vec{a} \times \vec{b}$



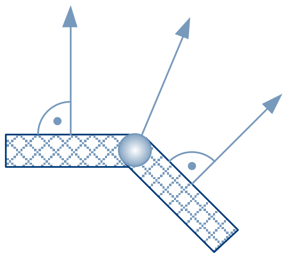


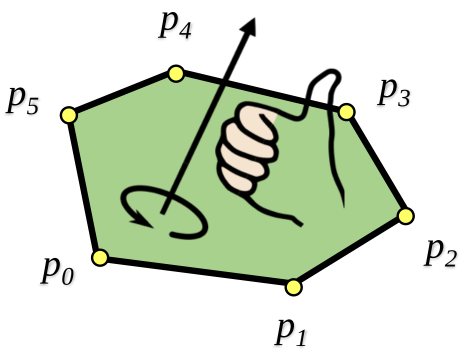


- Berechnung der Normalen
  - Z.B. über Kreuzprodukt aufeinander folgender Kanten
    - $\vec{n} = \vec{a} \times \vec{b}$
      - Rechtssystem (Rechte-Hand-Regel)
    - Achtung: $\vec{b} \times \vec{a} = -\vec{n}$
  - Garantiert Orientierung der Flächennormalen nach *außen*



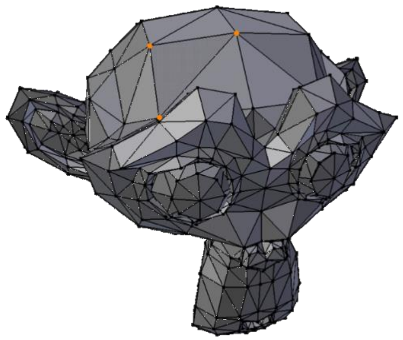
Polygone zur Körpermodellierung

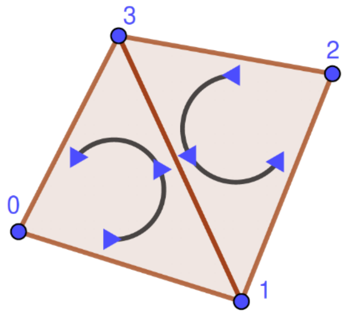




- Ebene Objekte besonders einfach zu handhaben
  - Aus genügend vielen Objekten lassen sich auch runde Formen annähern → Polygon als Basisobjekt für 3D
- Polygon
  - Aneinanderhängende Folge von Kanten (Edges), die durch je 2 Punkte (Vertices) definiert werden:

    $((p_0, p_1), \ldots, (p_{n-1}, p_n))$
  - Eingeschlossene Fläche auch als Face bezeichnet



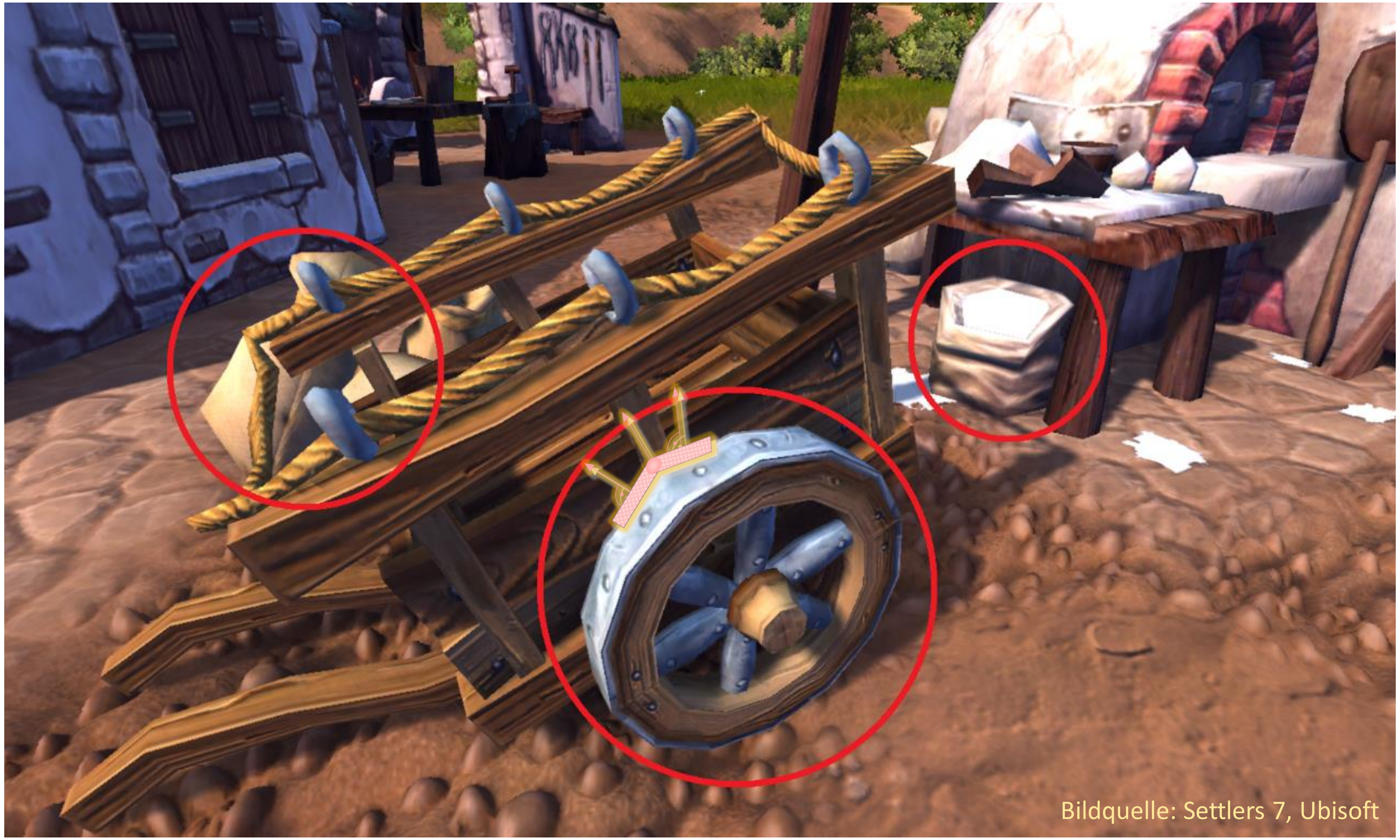


(0,1,3) and (2,3,1)
consistently oriented

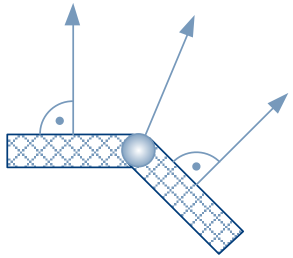
- Geforderte Eigenschaften
  - Geschlossen: Anfangspunkt entspricht Endpunkt → formal: $p_0 = p_n$
  - Einfach: Polygon schneidet sich selbst nicht
  - Eben: alle Kanten liegen in einer Ebene
  - Kanten benachbarter Polygone haben gegenläufige Orientierung (damit gleiche Normalenrichtung)

Gekrümmte Flächen: Quality vs. Speed

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES



Glättung eines Low-Poly-Modells durch Mittelung der Normalen von an Eckpunkt angrenzenden Flächen
→ Speicherung als Vertexnormale

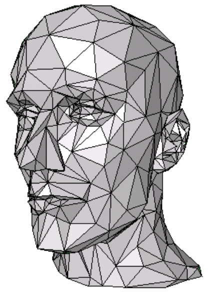


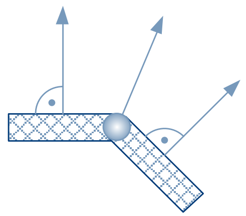
```
struct Vertex {
  Vec3 position;
  Vec3 normal;
  Vec2 texCoord;
};
```

Definition von Meshes

- Vertices definieren Attribute von Eckpunkten
  - Einfaches Beispiel:

```
struct Vertex {
  Vec3 position; // Position im 3D-Raum
  Vec3 normal;   // Interpolierte Flächennormale
  Vec2 texCoord; // Texturkoordinate (für Texturen)
};
```

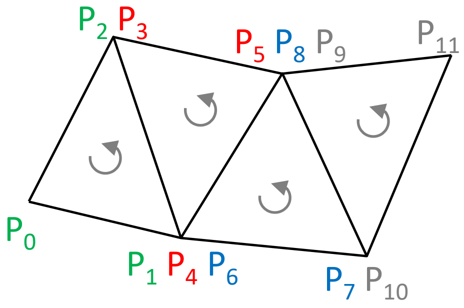




- Vertex-Reihenfolge kann Topologie beschreiben
  - Nicht-indiziertes Dreiecksmesh (naiver Ansatz):

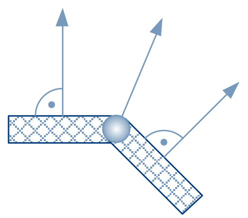
```
Vertex nonIndexedTriangleMesh[NUM_TRIS][3];
```

    - Hier im Bsp.: $P_0P_1P_2$ $P_3P_4P_5$ $P_6P_7P_8$ $P_9P_{10}P_{11}$
  - Problem: Redundanz
    - Mehrfaches Auftreten gleicher Vertices (inkl. zugehöriger Attribute wie Vertex-Normalen etc.)



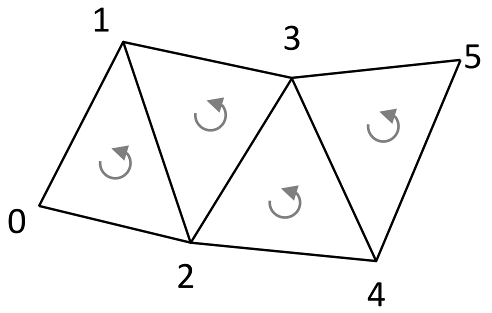
Definition von Meshes

- Lösung des Problems: indizierte Beschreibung
  - Arrays mit Positionen (u. ggfs. weiteren Attributen wie Vertex-Normalen)
  - Array mit Indizes, welche die Vertices (Eckpunkte) adressieren
    - Spart Speicher, nutzt Vertex Cache besser und vermeidet Rasterisierungsfehler (z.B. Löcher zwischen angrenzenden Kanten durch Floating-Point-Ungenauigkeiten)

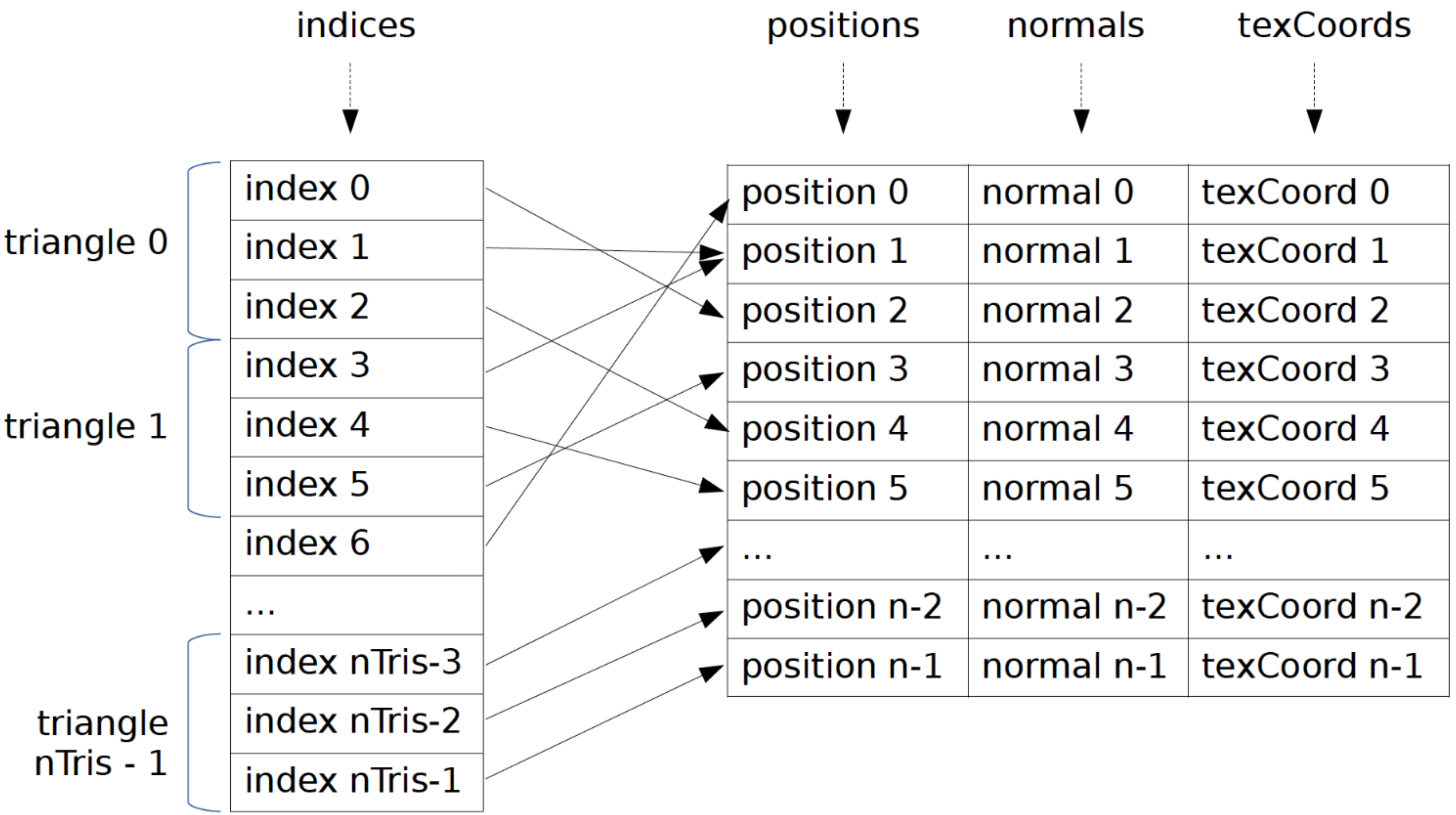


- GPU erwartet i.d.R. Vertex-Attribute in je separaten Arrays
  - Meist eindimensional gespeichert: x-, y-, z-Koordinaten folgen je aufeinander
  - Indiziertes Dreiecksmesh

```
struct IndexedTriangleMesh {
  Vec3 positions[NUM_VERTS];
  Vec3 normals[NUM_VERTS];
  Vec2 texCoords[NUM_VERTS];
  unsigned indices[NUM_TRIS * 3];
};
```



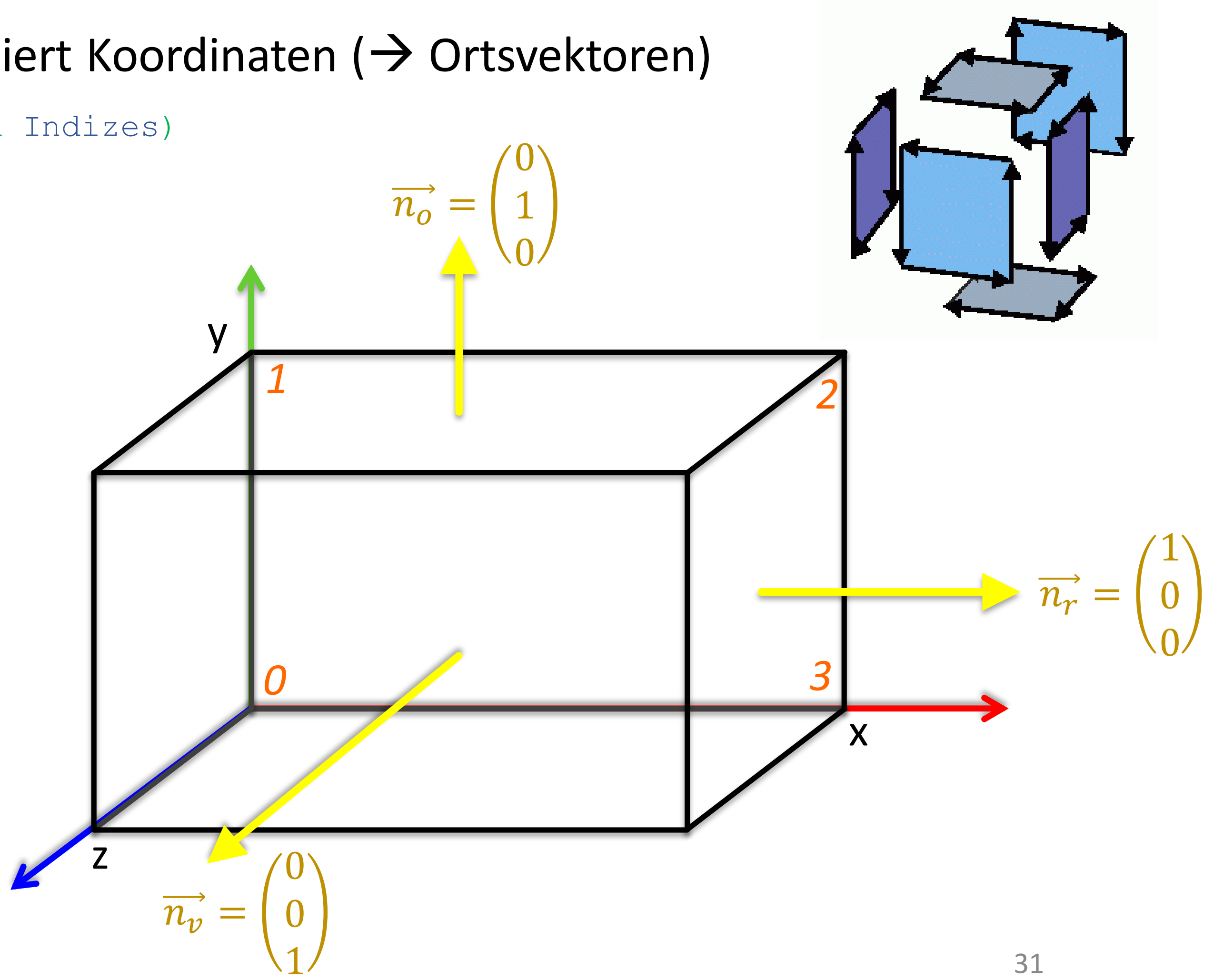
Indizierte Mesh-Beschreibung



Beispiel: Box im Eigenbau

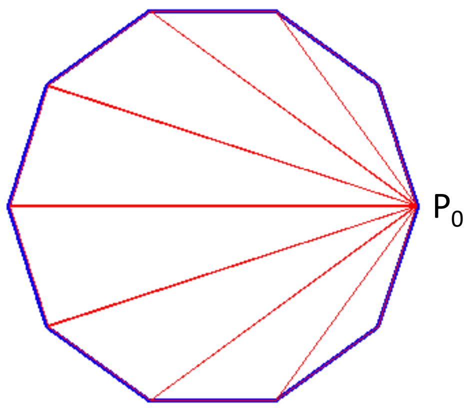
- Lage der Eckpunkte zum lokalen Ursprung definiert Koordinaten (→ Ortsvektoren)

```
// Dreiecksnetz (Positionen sowie Flächennormale und Indizes)
0,0,0,  0,y,0,   x,y,0,  x,0,0,   // hinten
          0, 0,-1  ( 0, 1, 2,   2, 3, 0)
0,0,z,  0,y,z,   x,y,z,  x,0,z,   // vorne
          0, 0, 1  ( 4, 7, 5,   5, 7, 6)
0,0,0,  0,0,z,   0,y,z,  0,y,0,   // links
         -1, 0, 0  ( 8, 9,10,  10,11, 8)
x,0,0,  x,0,z,   x,y,z,  x,y,0,   // rechts
          1, 0, 0  (12,14,13,  14,12,15)
0,y,0,  0,y,z,   x,y,z,  x,y,0,   // oben
          0, 1, 0  (16,17,18,  18,19,16)
0,0,0,  0,0,z,   x,0,z,  x,0,0    // unten
          0,-1, 0  (20,22,21,  22,20,23)
```



Zerlegung in Dreiecke

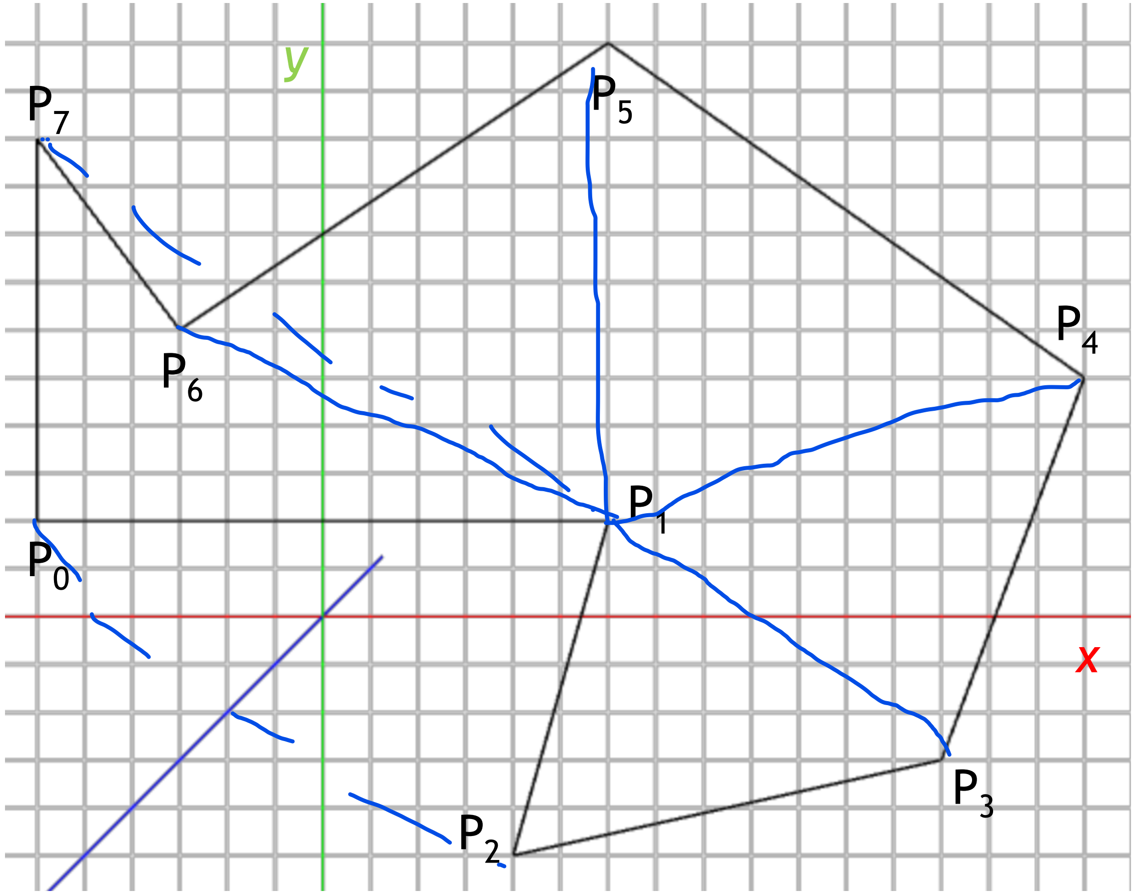
- Geg.: alle Punkte einer Fläche (z.B. in Array)
- Wähle beliebigen Start-Eckpunkt P_i
- Erzeuge Dreieck aus Eckpunkt P_i und dessen beiden Nachfolgern
- Teste, ob Innenwinkel < 180° und keine Schnittpunkte mit Kanten
  - *Ja:* Verwende Dreieck u. streiche mittleren Punkt weg; führe Verfahren fort, bis nur noch 2 Punkte existieren – behalte ursprünglichen Punkt je bei
  - *Nein:* Verwerfe Dreieck und setze Verfahren beim nächstmöglichen, noch nicht gestrichenen Eckpunkt neu an



Übung 2

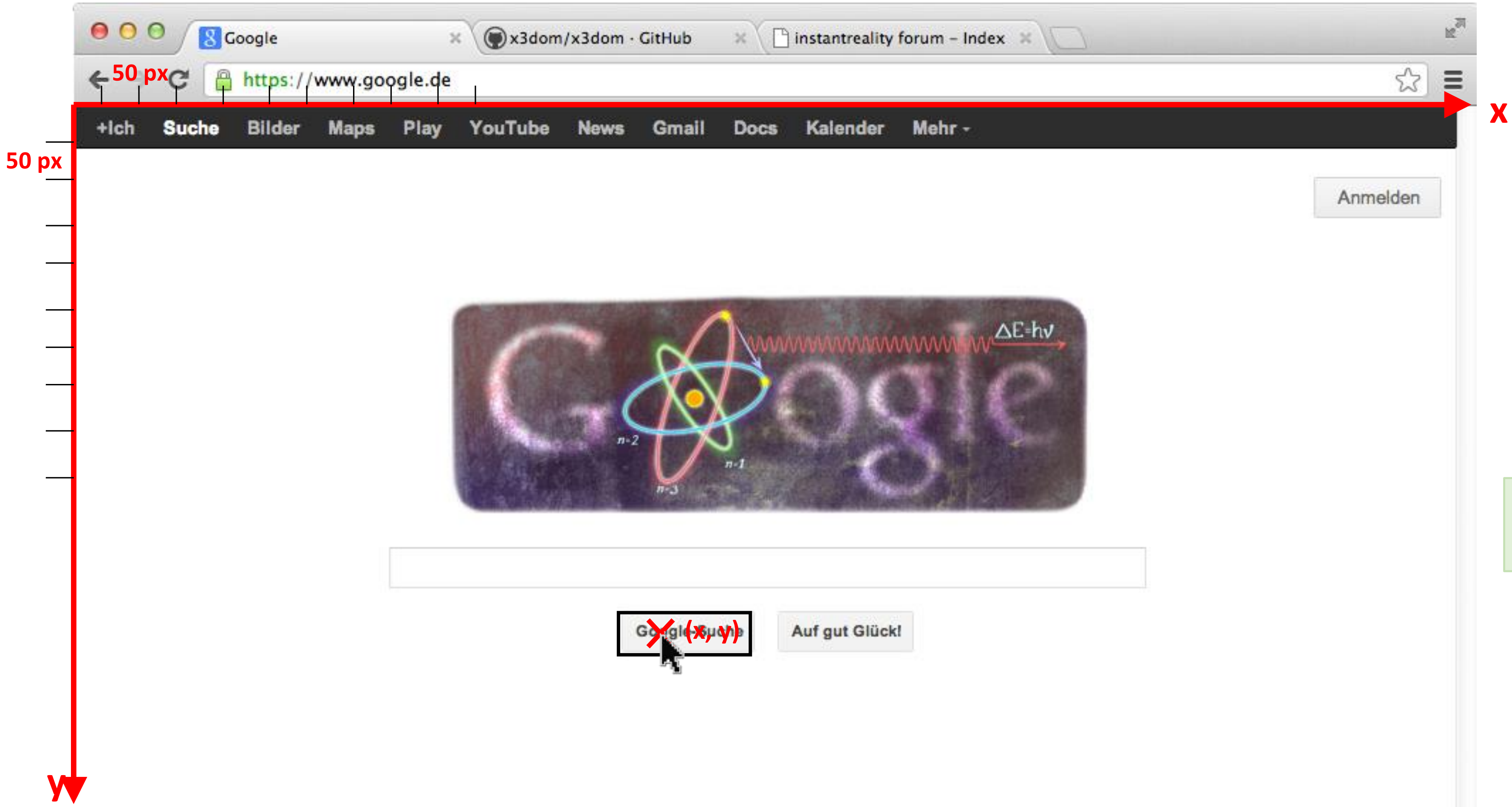
- Zerlegen Sie das gezeigte 2D-Polygon in Dreiecke!

```
float positions[] = {
   -3,    1,
    3,    1,
    2,   -2.5,
    6.5, -1.5,
    8,    2.5,
    3,    6,
   -1.5,  3,
   -3,    5,
};

unsigned short indices[] = {
   0, 1, 2, 3, 4, 5, 6, 7
};
```



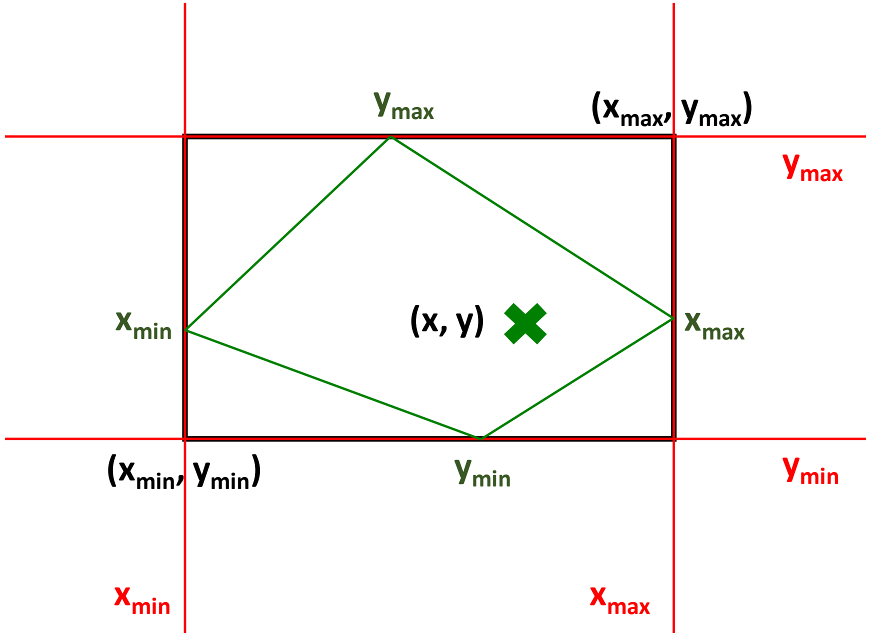
Einfache Interaktionen

Bounding Box und Inside Test



Achtung, in 2D-APIs zeigt *y*-Achse meist nach unten

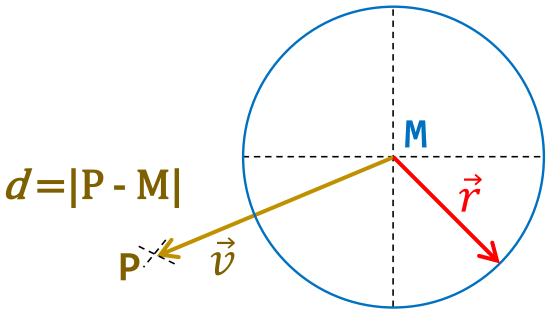
Bounding Box und Inside Test



- 2D-Selektion am einfachsten mit Rechtecken
  → daher Grundform von Buttons
- Implementierung:

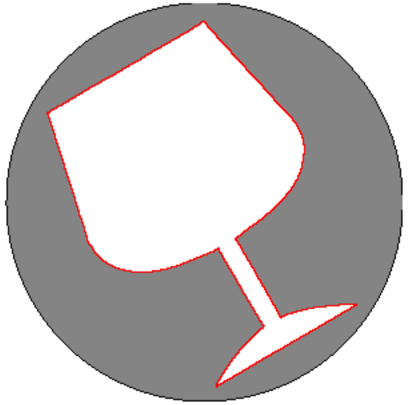
```
bool inside(float x, float y) {
    return xmin <= x && ymin <= y &&
           x <= xmax && y <= ymax;
}
```

- Analog für Kreis: Punkt P (x, y) liegt innerhalb, wenn Distanz *d* zwischen P und M kleiner als Radius *r* ist

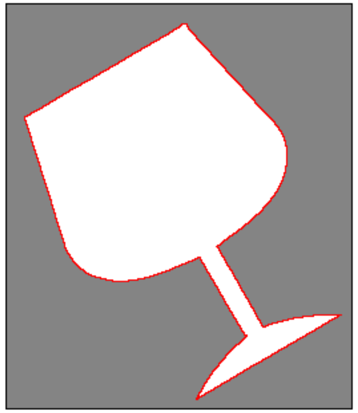


Bounding Volumes (1)

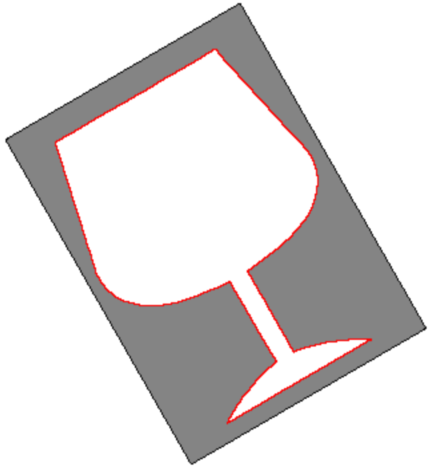
- Müssen einfach sein, damit sich Schnitttests mit anderen „Primitiven“ (für Sichtvolumen, Sehstrahl, Kollision) leicht berechnen lassen



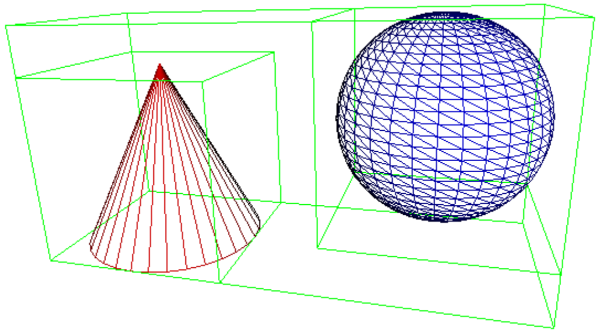
Sphere



Axis-Aligned BBox (AABB)

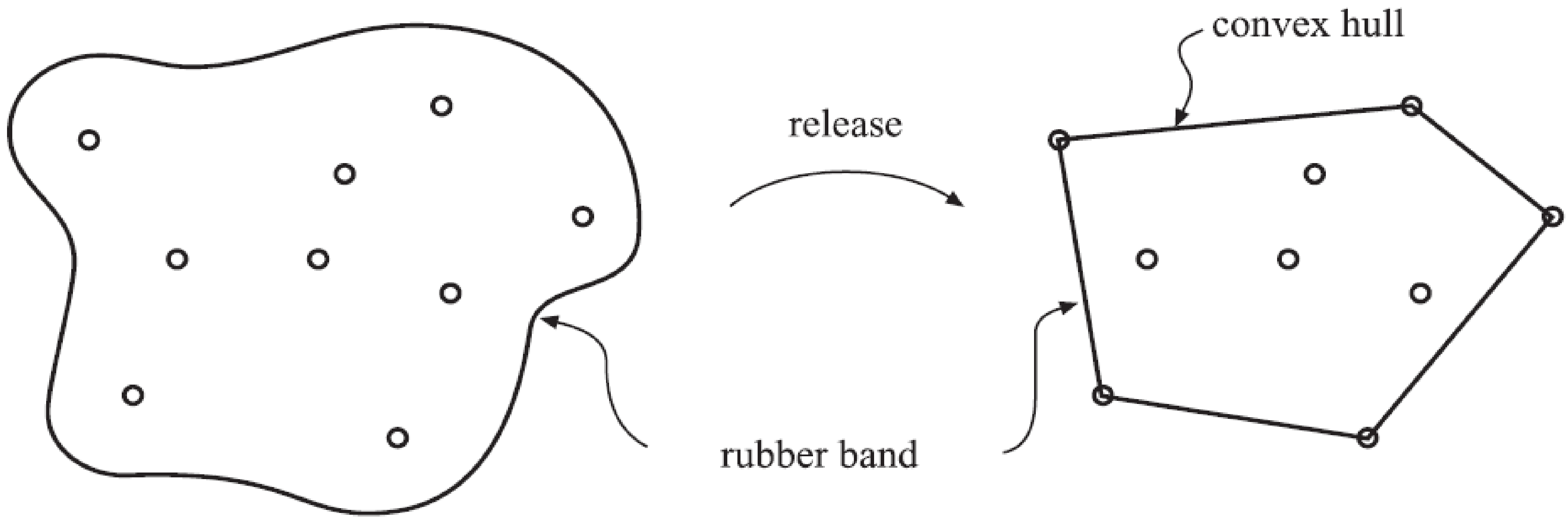


Object-Oriented BBox (OBB)



Bounding Volume Hierarchy möglich

Bounding Volumes (2)



- Konvexe Hülle: kleinstes konvexes Polygon (bzw. Polyeder in 3D), das Objekt enthält
- *Übung:* Geben Sie die konvexe Hülle des in voriger Übung gezeigten Polygons an!

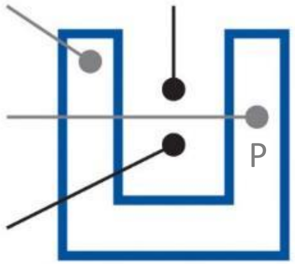
Inside-Test bei Polygonen (in 2D)

- Test, ob Punkt *P* innerhalb oder außerhalb Fläche *F*
  - 3D: ggfs. Projektion auf eine der 3 Koordinatenebenen

Dazu Achse wählen, die zu größter Projektion führt

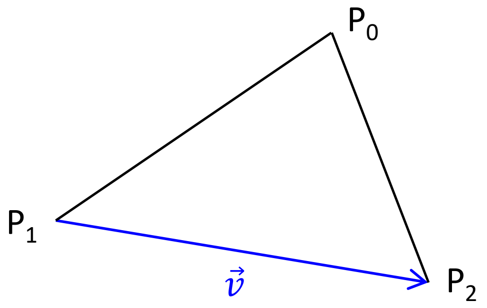
Koordinate, deren Koeffizient in Ebenengleichung größten Absolutbetrag hat, auf Null setzen

- Strahl *r* durch *P* legen (z.B. entlang x- oder y-Achse)
  - Strahlkonstruktion: $r(t) = P + t \cdot \binom{1}{0}$ mit $t > 0$
- Anzahl *n* der Schnittpunkte mit Kanten bestimmen
  - Ist *n* gerade, so liegt *P* außerhalb, sonst innerhalb, d.h. $P \in F$
  - LGS am einfachsten lösbar, wenn konstruierter Strahl entlang *x*- oder *y*-Achse geht
- Alternativen:
  - Ausgehend von *P* auf Eckpunkte blicken und Summe *S* aller Innenwinkel berechnen
    - $S = 360° \;\Rightarrow\; P \in F$
  - Zuerst in Dreiecke zerlegen und dann auf gültige baryzentrische Koordinaten testen



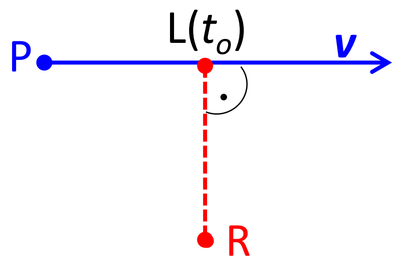
Linien und Kanten

- Geg.: Zwei Punkte $P_1(x_1, y_1)$ und $P_2(x_2, y_2)$
  - Strecke zwischen zwei Punkten P_i und $P_{(i+1)\%n}$ eines *n*-Ecks heißt Kante
- Parametrische Gleichung der zugehörigen Geraden
  - $L(t) = P_1 + t\,(P_2 - P_1) = P_1 + t\,\mathbf{v}$
- Wichtige Daten für Graphik sind hier i.d.R. Vektor, Punkt und Kante
  - $t \in [0; 1] \rightarrow$ Kante/Strecke; $t \in \mathrm{IR} \rightarrow$ Gerade; $t \in \mathrm{IR}_0^+ \rightarrow$ Strahl



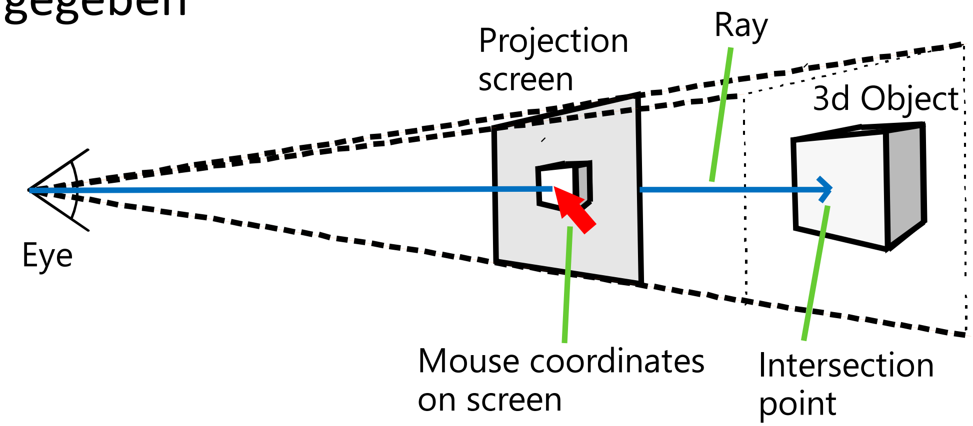
- Welcher Punkt auf Geraden $L(t) = P + t\,\mathbf{v}$ liegt am nächsten zu Punkt R?
  - Verwendung des Skalarprodukts: $(R - L(t_0)) \cdot \mathbf{v} = 0$
  - $\Leftrightarrow (R - (P + t_0\,\mathbf{v})) \cdot \mathbf{v} = 0$
  - $\Leftrightarrow (R - P) \cdot \mathbf{v} - t_0 \cdot \mathbf{v} \cdot \mathbf{v} = 0$
  - $\Leftrightarrow (R - P) \cdot \mathbf{v} = t_0 \cdot |\mathbf{v}|^2 \quad \rightarrow \quad \boldsymbol{t_0} = (R - P) \cdot \mathbf{v} \,/\, |\mathbf{v}|^2$

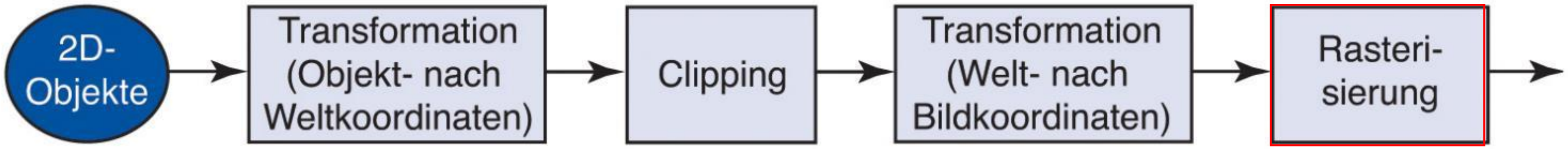
$L(t_0)$ liegt auf Strecke P_1P_2, wenn $t_0 \in [0, 1]$



Selektion in 3D-Szenen (Picking)

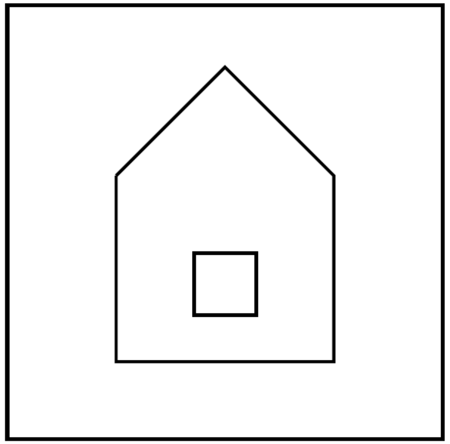
- (Maus-) Interaktion mit 3D-Objekten
- Über Strahl, von Auge durch Mausposition bzw. Fingerspitze
  - Is-over: Strahl trifft Objekt (Raycasting)
  - Trigger: User klickt Button
- Von 2D nach 3D durch „Umkehroperation“ der Projektion
  - Problem: Mausposition (x, y) nur 2-dimensional gegeben
  - Kein eindeutiger z-Wert möglich,
    2D-Position definiert nur Strahl:
    → Strahltest nötig



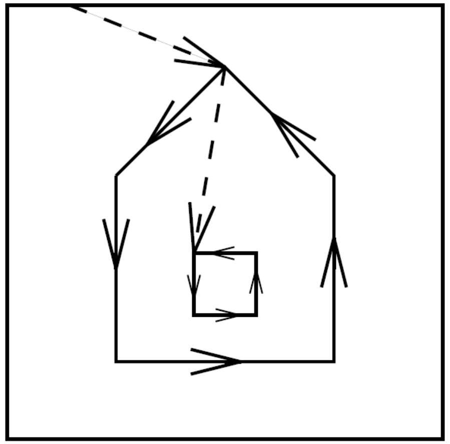


Rasterisierung

Vektor- vs. Rasterverfahren

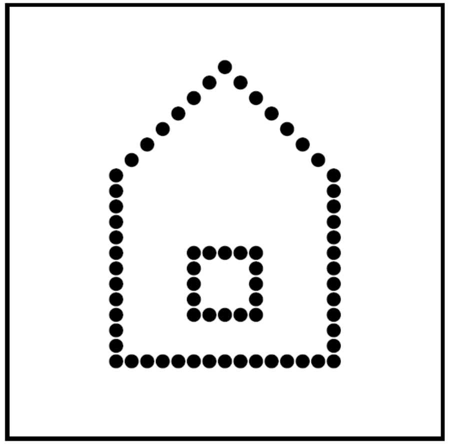


(a) ideal line drawing

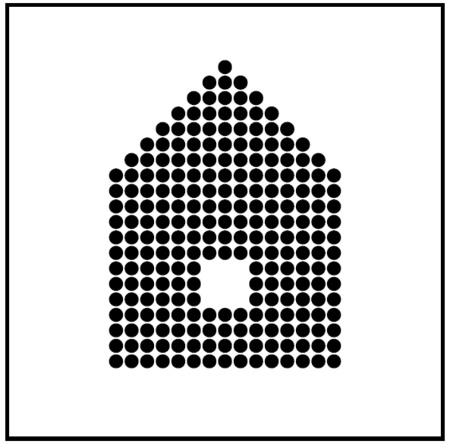


(b) Vector scan

Beispielformat:
SVG



(c) Raster scan with outline primitives

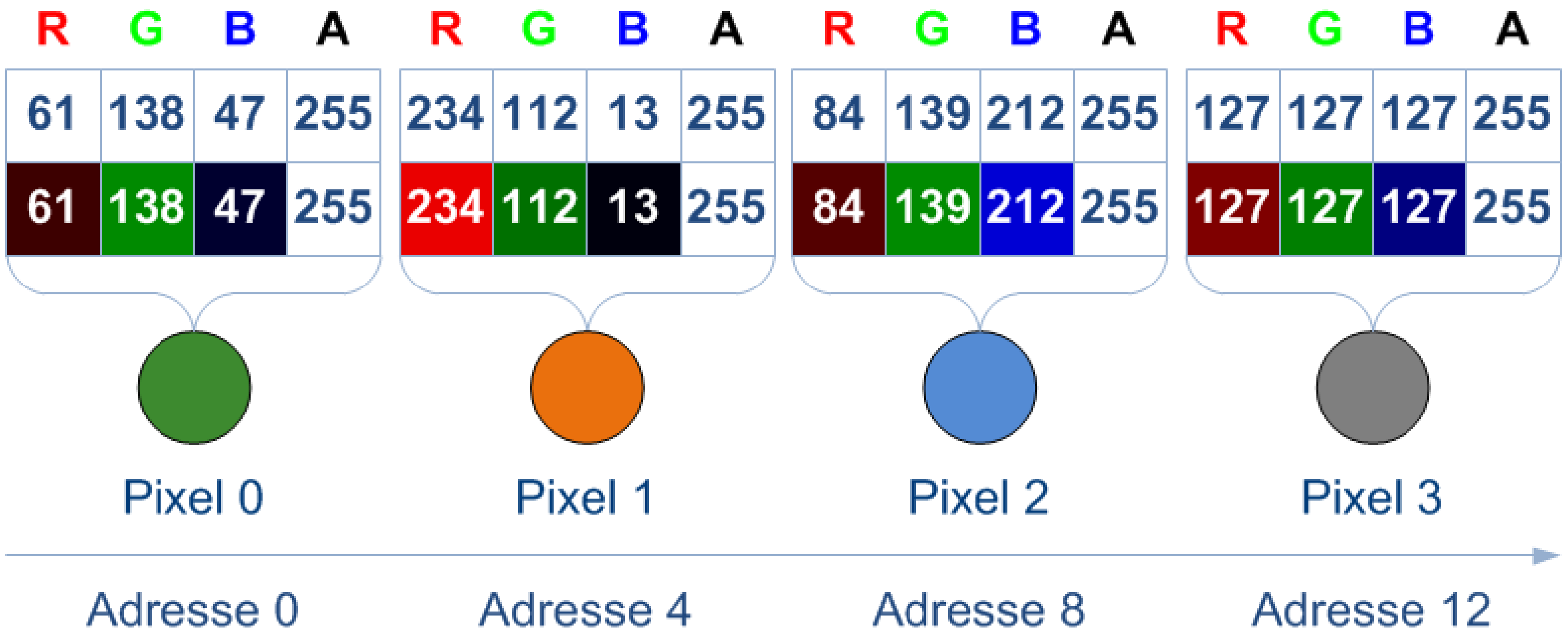


(d) Raster scan with filled primitives

Beispielformat:
BMP

Pixelspeicherung

- Videospeicher linear in horizontalen Pixelreihen (sog. Scanlines) ausgerichtet
  - Grafikkarte speichert pro Pixel einen 4 Byte großen Farbwert
  - Farbwert setzt sich aus vier Kanälen zusammen (RGBA)



Bresenham-Algorithmus

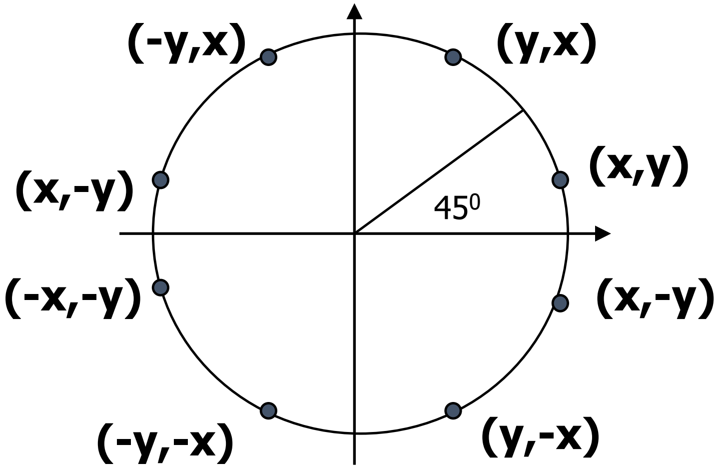
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Approximation einer Linie im kontinuierlichen Raum durch Punktmenge im diskreten Raum
  - Zum Rastern von Linien auf Rasterdisplays
- Nur Integer-Arithmetik (Addition, Subtraktion und Linksshift, d.h. Multiplikation mit 2)
  - Da bereits 1965 veröffentlicht…
- Jeder Punkt wird nur einmal erzeugt und hat kürzesten Abstand zur idealen Gerade
  - Inkrementelles Verfahren (→ x++)
  - Inkrement von y durch Betrachtung von Fehler ε bei Steigung $m = {}^{dy}/_{dx}$

Herleitung

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \tan \alpha$$

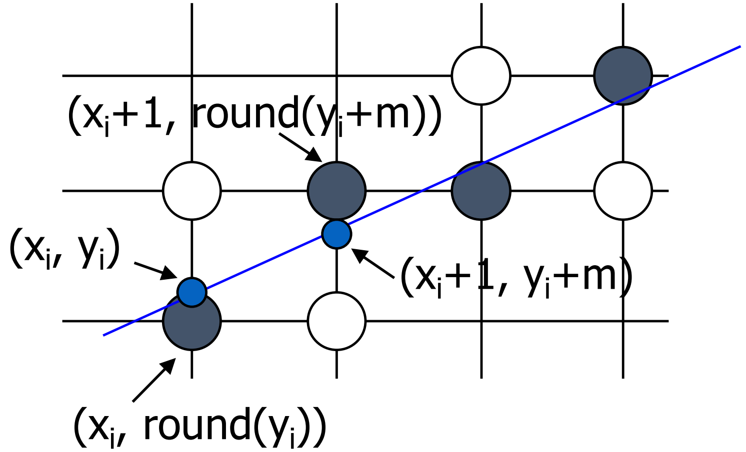
- Zugrunde liegende Idee:
  - DDA-Algorithmus (Digital Differential Analyzer)
  - Differentialquotient für Gerade $y = m \cdot x + b$ ist Steigung $m = \frac{dy}{dx}$
  - Anfangs- und Endpunkt liegen auf Raster
- Vereinfachende Annahme:
  - Nur 1. Oktanten betrachten, d.h. $0 \leq m \leq 1$
    - Vereinfachung: y nach oben
    - 0 < dy <= dx <= 1 → dx >= dy
  - Für negative Steigung y negieren
  - Für Steigungen > 45° x und y vertauschen



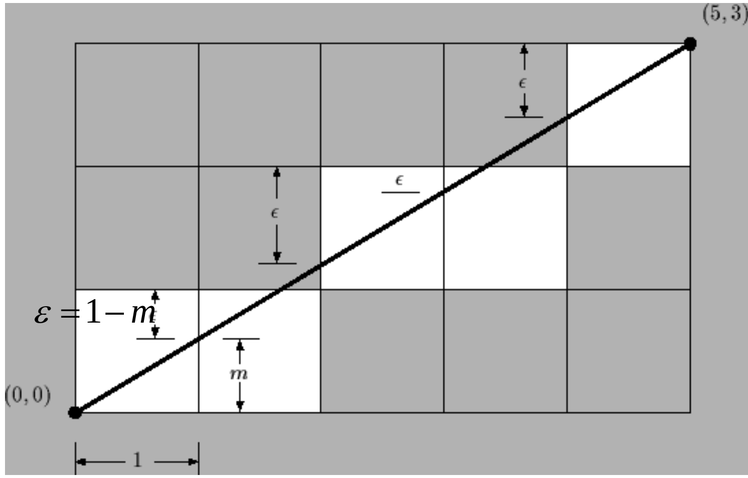
Fehlerbetrachtung

- Algorithmus (V1, Simple DDA)

```
int dx = x1 - x0, dy = y1 - y0;
float m = dy/(float)dx, y = y0;
for (int x=x0; x<=x1; x++) {
  setPixel(x, round(y));
  y += m;
}
```



  - Problem: Floats & Rundungen
- Verbesserung
  - Welcher Pixelmittelpunkt liegt näher zur Geraden?
  - y-Inkrement in separater Variable ε summieren
  - y nur erhöhen, falls Fehler ε näher zu y+1 liegt, also ε>0.5



Fehlerbetrachtung

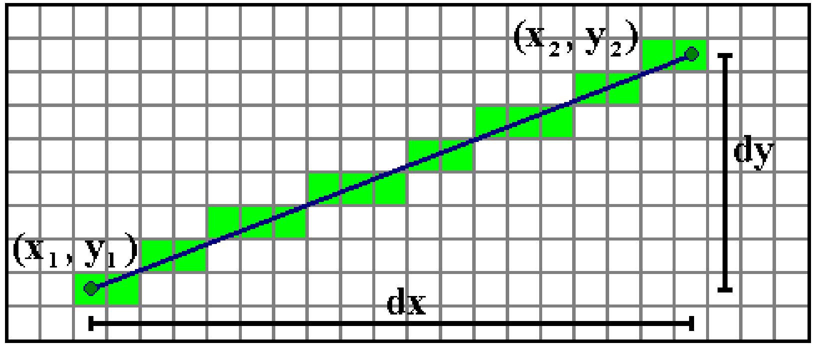
```
int dx = x1 - x0, dy = y1 - y0;
float e = 0, m = dy / dx, y = y0;

for (int x = x0; x <= x1; x++) {
  setPixel(x, y);

  e += m;
  if (e > .5f) {
    y++;
    e--;
  }
}
```

Problem:
Immer noch 2 Floats (→ *m* und 0.5)
Floating-Point-Operationen waren teuer und sind ungenau

Lösung:
Tatsächlicher Wert von ε ist unwichtig, nur Abschätzung ist interessant
→ mit 2 * dx multiplizieren

Bresenham-Algorithmus

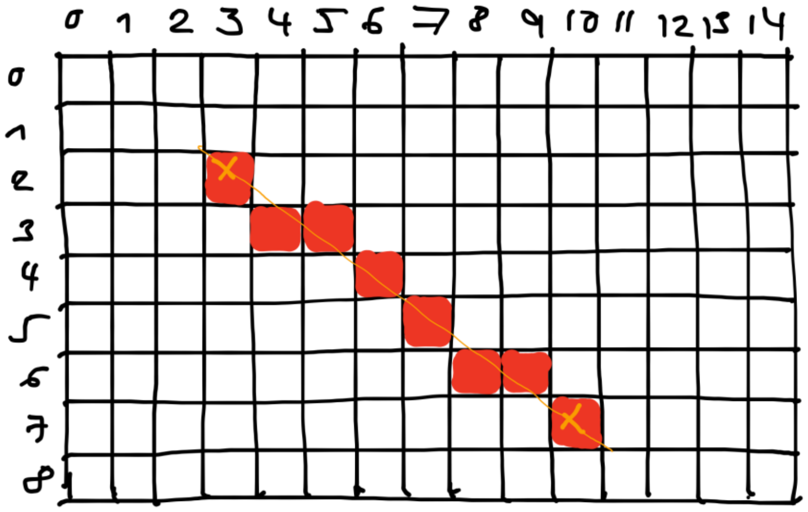
```
void line(int x0, int y0, int x1, int y1) {
  int dx = x1 - x0, dy = y1 - y0;
  int e = 0, dx2 = dx << 1, dy2 = dy << 1;

  for (int x = x0, y = y0; x <= x1; x++) {
    setPixel(x, y);

    e += dy2;
    if (e > dx) {
      y++;
      e -= dx2;
    }
  }
}
```



Bresenham am Beispiel

- Geg.: Pixel-Gitter des Bildschirms
  - Startpunkt der Linie in (3, 2), Endpunkt in (10, 7)
    - x0 = 3, y0 = 2;
      dx = 10 – 3 = 7 (≥ 0)
      dy = 7 – 2 = 5 (≥ 0)
    - dx ≥ dy und m = dy / dx = 5 / 7 (≤ 1)
- Berechnungsschritte für alle Pixel auf Linie:

Schritt	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
e	0	-4	6	2	-2	-6	4	0
x	3	4	5	6	7	8	9	10
y	2	3	3	4	5	6	6	7

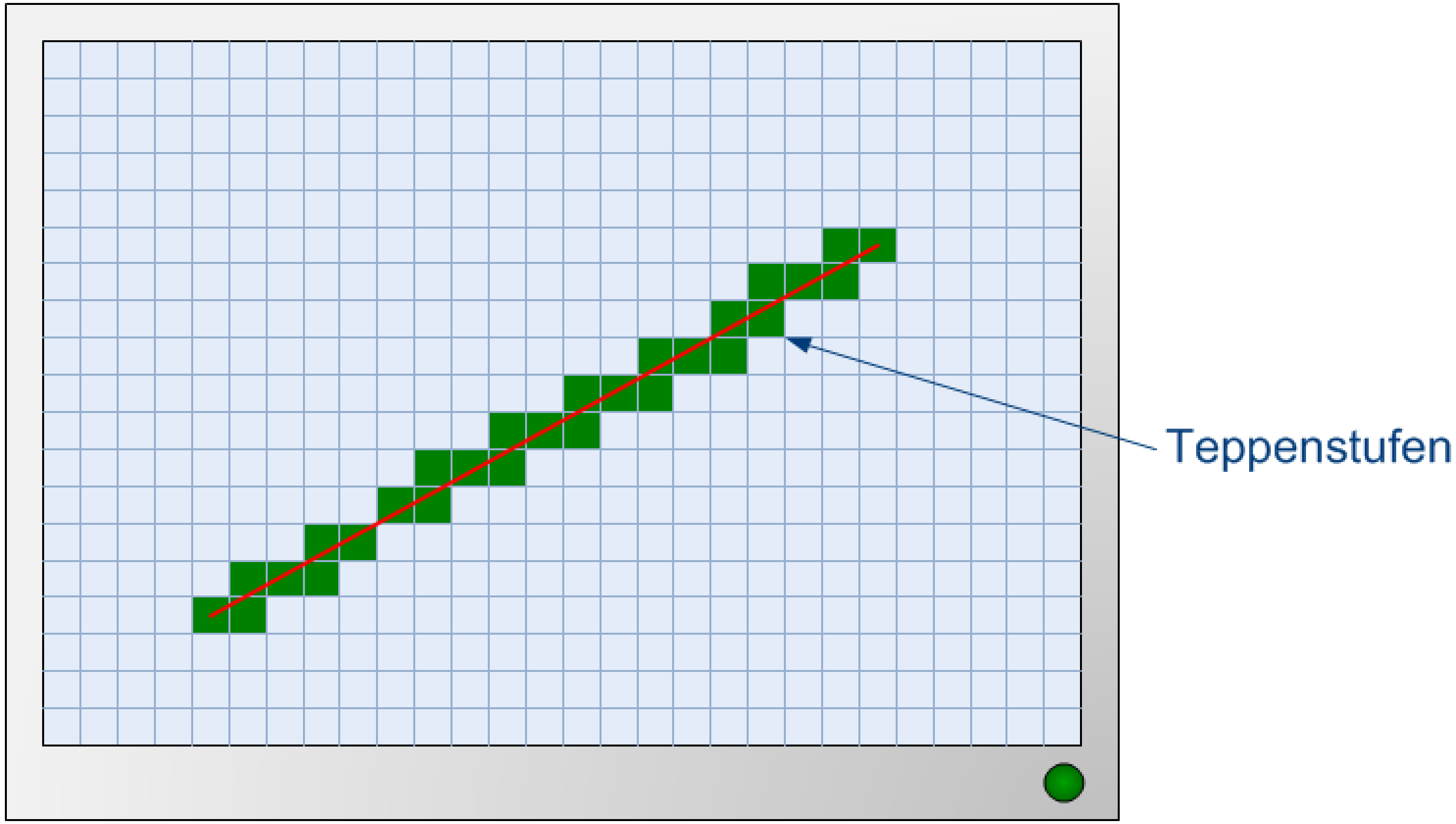


Ergebnis: Linie, die nur einen Pixel pro Zeile benötigt – bzw. pro Spalte, falls Steigung *m* > 1

Exkurs: Kantenglättung

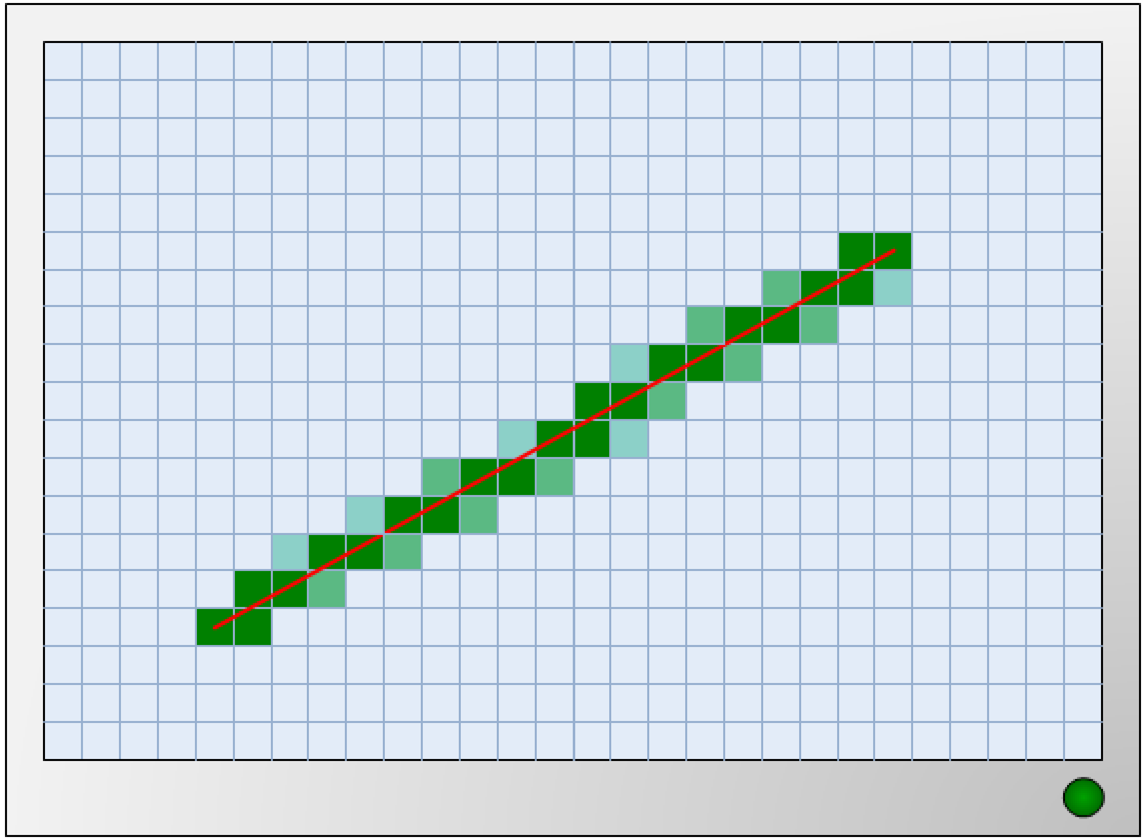
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Auflösung des Bildschirms begrenzt $\Longrightarrow$ perfekte Linie nicht möglich: Treppenstufen

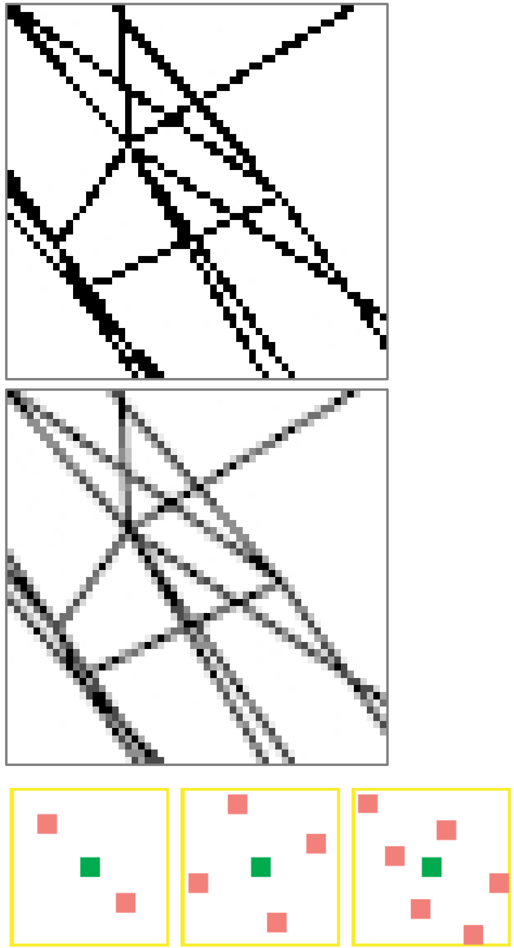


Exkurs: Kantenglättung

- Aufweichung der Stufen durch Übergangsfarben/ -intensitäten (je nach Abstand zur Originallinie)



Exkurs: Antialiasing



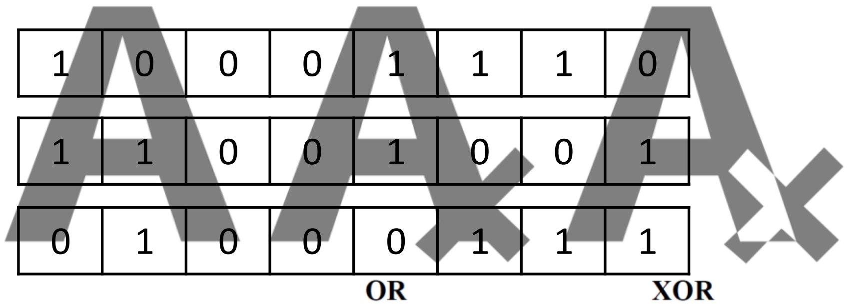
- Monitor-Auflösung vergleichsweise gering
  - Besonders an Kanten u. Linien einzelne Pixelsprünge sichtbar (Aliasing)
  - Anti-Aliasing versucht dies durch Farbmischung abzuschwächen

- Multisampling
  - Bild wird in höherer (z.B. 2x / 4x / ...) Auflösung berechnet und z.B. 2x2 bzw. 4x4 Pixel gemittelt
  - Analytisch: für jeden Pixel wird berechnet, welcher Anteil vom Primitiv bedeckt ist
  - Anteil wird als Wichtungsfaktor zur Mittelung von Primitivfarbe und vorhandener Farbe verwendet
  - MSAA (Multisample Antialiasing) Sampling Patterns
    (grün ist eigentliches Pixel, rot sind Samples)

Einschub: Bit Block Transfer

- Blitting ist Operation, bei der Pixelblock bitweise entsprechend geg. RasterOp kombiniert wird
  - Möglichkeiten: Constant, Copy, AND, OR, XOR, NOT
  - XOR für Rubberband oder Mauscursor eingesetzt
    - → Zweimal XOR stellt alten Wert wieder her: $(S \oplus D) \oplus D = S \oplus (D \oplus D) = S \oplus 0 = S$
- Zeichenmodus
  - Bsp.: Logische Verknüpfung mit XOR

Source-Pixel:	1	0	0	0	1	1	1	0
Destination-Pixel:	1	1	0	0	1	0	0	1
Ergebnis:	0	1	0	0	0	1	1	1

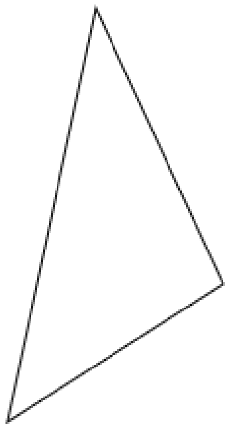


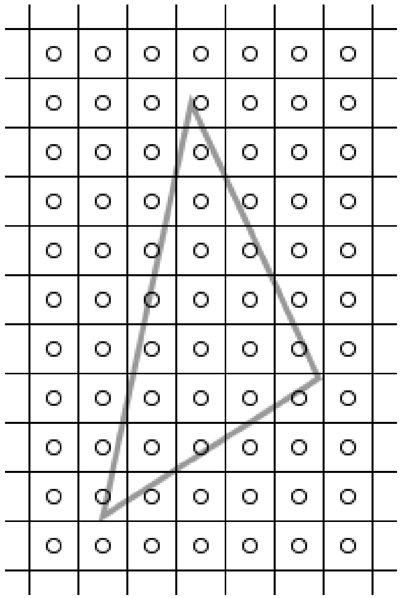
  - Heute meist durch sog. Alpha Blending ersetzt

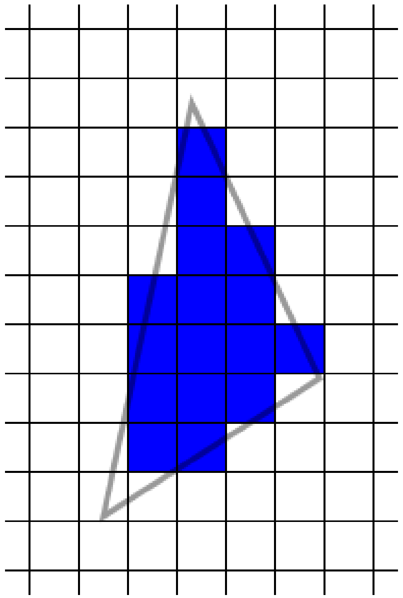
Polygonfüllen

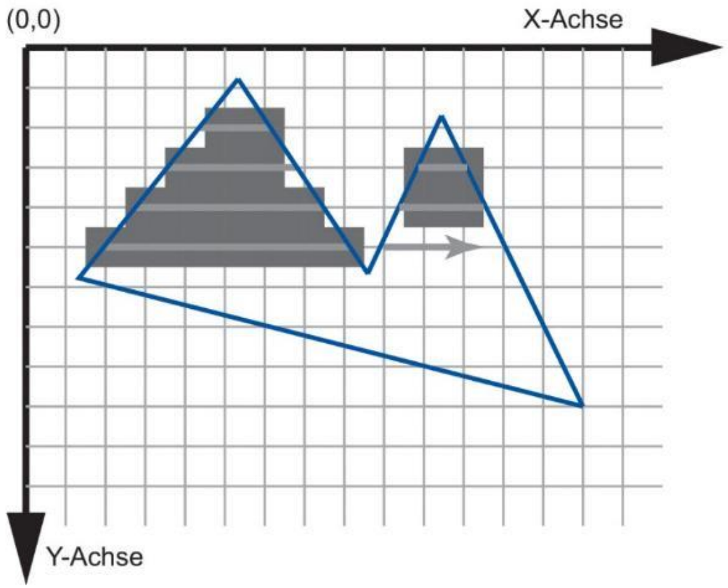
- Old-School auf CPU: Mit Scanline-Algorithmus
- Erst Polygon in Dreiecke zerlegen (Tessellierung)
  - Trivial bei konvexen Flächen: Eckpunkte eines Dreiecks bilden immer eine Ebene!
- Dann Flächen zeilenweise mit Farbverlauf füllen
  - Alternativ direkt Pixel mit ungerader Parität* einfärben

* Von links kommend Kanten zählen: Nur Füllen bei ungerader Kantenzahl





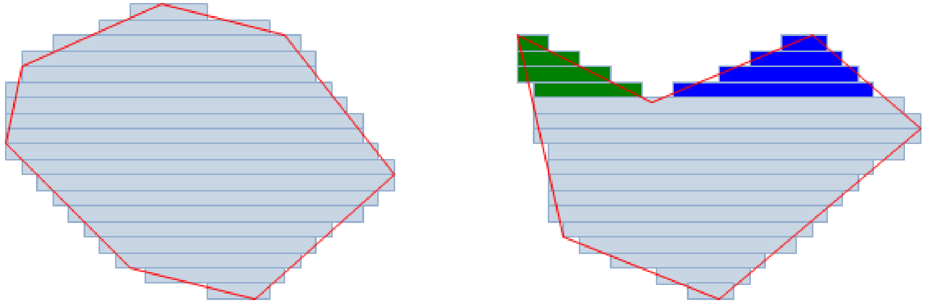




Dreiecke bevorzugt

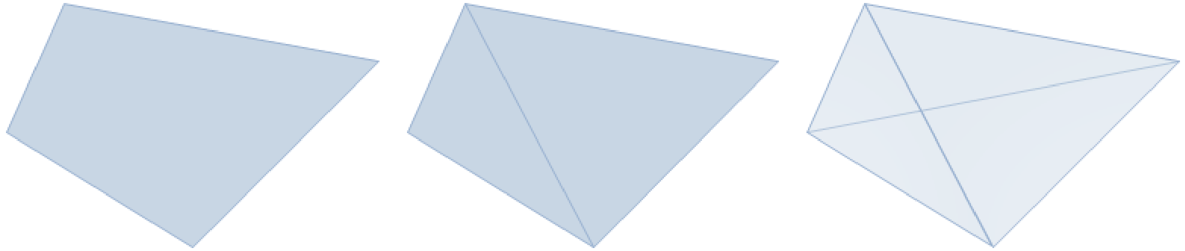
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Konvexe Flächen erlauben Füllen mit zusammenhängender waagrechter Pixelreihe pro Zeile



Konvexes Polygon Nicht-konvexes Polygon

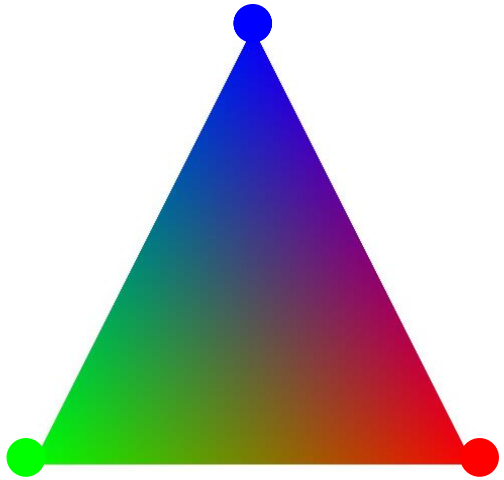
- Dreiecke sind konvex und Eckpunkte bilden immer Ebene



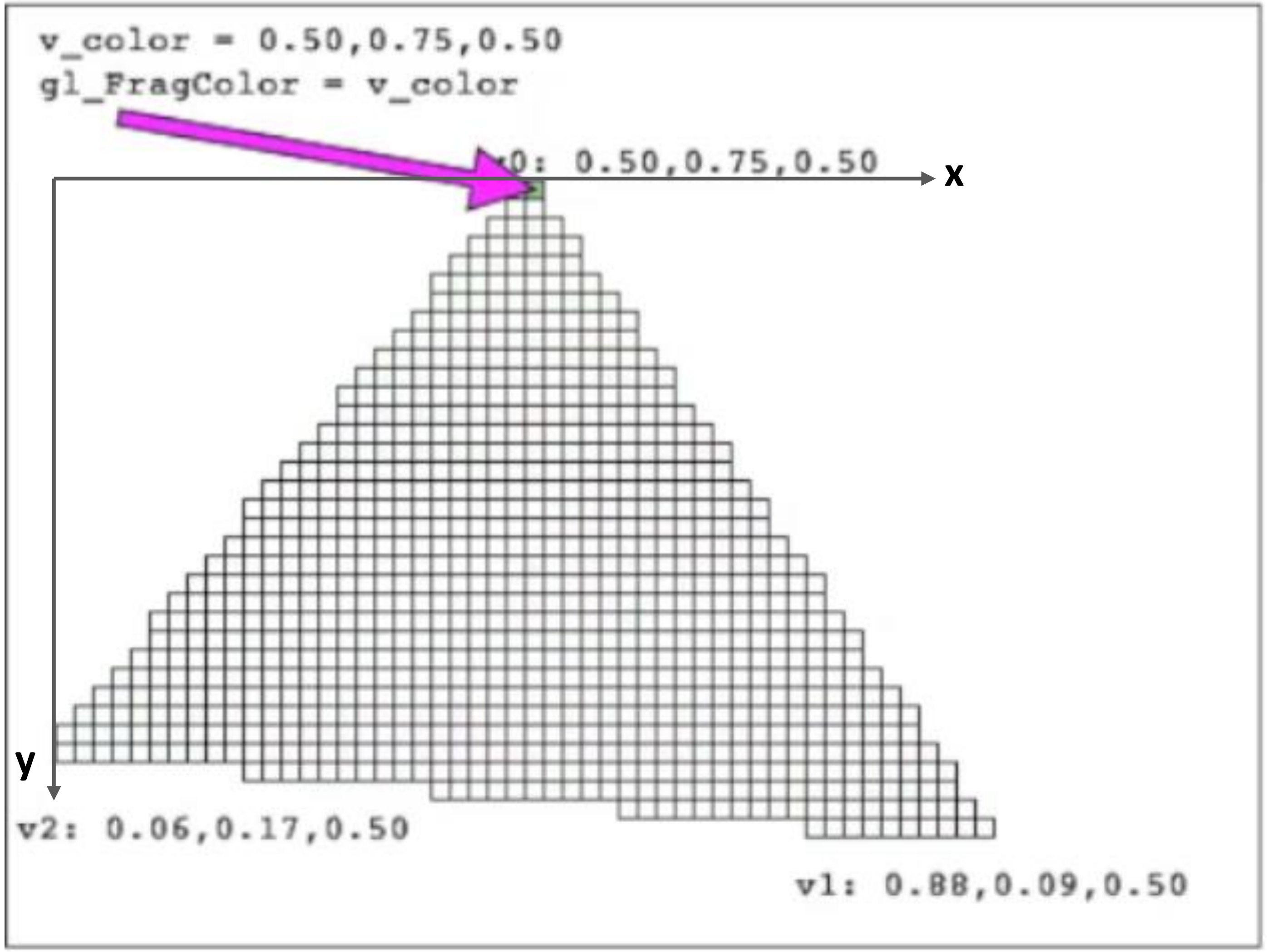
Eckpunkte eines n-Ecks müssen nicht auf einer Ebene liegen

Dreiecke bevorzugt

- Angabe einer Farbe pro Dreieck auf GPU nicht möglich
  - Farben können für GPU nur für gesamtes 3D-Objekt oder pro Eckpunkt definiert werden
  - Geometrie beinhaltet neben Farben u.a. auch Vertexnormalen und Texturkoordinaten
- Vertices haben damit evtl. unterschiedliche Farben
  - Mit welcher Farbe wird Dreieck gefüllt?
- Eckpunkte eines Dreiecks bilden glücklicherweise Ebene
- Lineare Interpolation pro Pixel zwischen Eckpunkten
  - Lineare Interpolation zw. Punkten P und Q für Interpolationswert $t \in [0, 1]$ ergibt sich aus: $R(t) = P + t \cdot (Q - P) = (1 - t) \cdot P + t \cdot Q$
  - Interpoliert damit entlang der Kante von P (für $t = 0$) nach Q ($t = 1$)
  - Geschieht auf GPU beim Rasterisieren



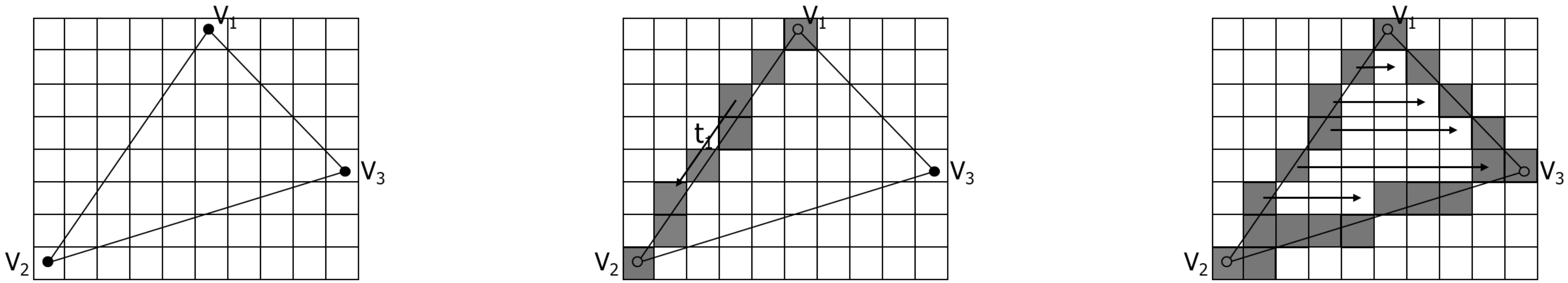
Bilineare Interpolation



- Interpolationsverfahren für Flächen
- $c = c_1 + \frac{c_2 - c_1}{x_2 - x_1} \cdot (x - x_1)$
- Erledigt GPU für uns ☺

Rasterisierung im Detail

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

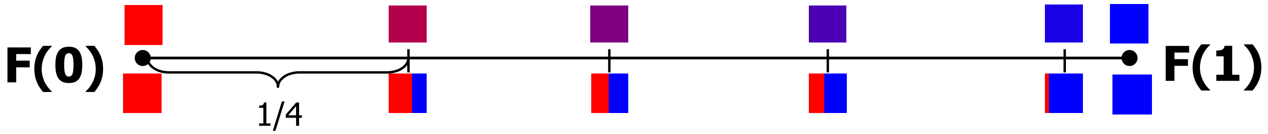


1. Bestimmung der Pixel-Koordinaten der Vertices
2. Interpolation aller Vertex-Attribute entlang der Dreieckskanten
   - Parameter t_1 geht von 0 bis 1: $R(t_1) = V_1 + t_1 \cdot (V_2 - V_1) = (1 - t_1) \cdot V_1 + t_1 \cdot V_2$
3. Interpolation der Pixelwerte entlang der einzelnen Rasterzeilen
   - Berechnet beim Füllen der Fläche bilinear interpolierte Vertex-Attribute

Lineare Farbinterpolation

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

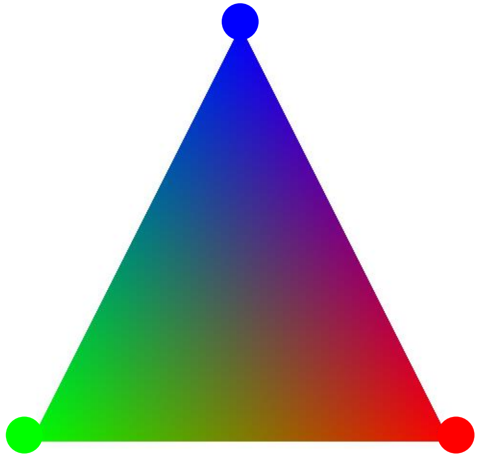
- Wie berechnet man den Farbverlauf innerhalb einer Linie oder eines Dreiecks, wenn jeder Eckpunkt (d.h. Vertex) eine andere (Vertex-)Farbe hat?



$$F(t) = (1-t)\cdot F(0) + t \cdot F(1)$$

$$F(t) = (1-\frac{1}{4})\cdot F(0) + \frac{1}{4} \cdot F(1)$$

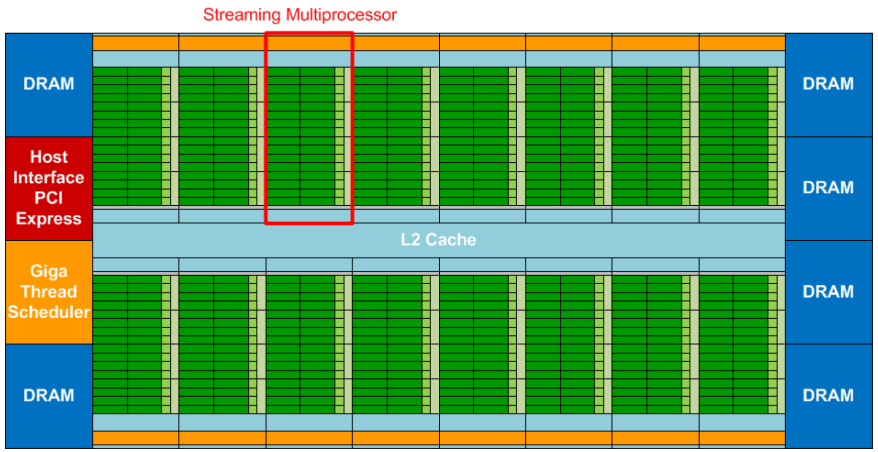
$$F(t) = \frac{3}{4}\cdot\begin{pmatrix}1\\0\\0\end{pmatrix} + \frac{1}{4}\cdot\begin{pmatrix}0\\0\\1\end{pmatrix} = \begin{pmatrix}\frac{3}{4}\\0\\\frac{1}{4}\end{pmatrix}$$

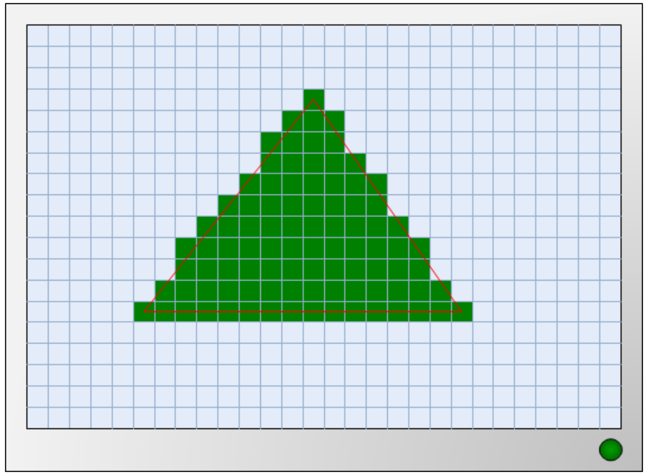


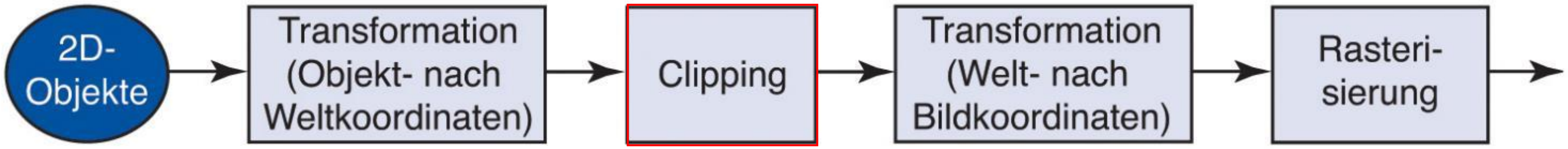
- Auf die gleiche Weise werden alle Vertex-Attribute interpoliert
- Zuordnung erfolgt durch gleichen Attributnamen in Vertex- sowie in Fragment-Shader

Exkurs: Parallelität

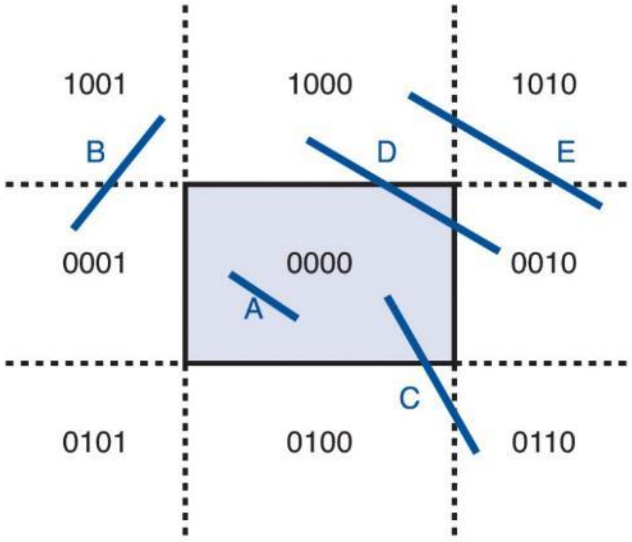
- GPU erlaubt massiv parallele Abarbeitung der gleichen einfachen Abläufe (nämlich u.a. Pixelfärben)
  - Wenige arithmetische Instruktionen, geringe Anzahl von Verzweigungen
- Füllen eines Dreiecks erfordert gleiche Instruktionen für alle Pixel (geschieht in Shader-Programm)
  - Geschieht heutzutage nicht mehr zeilenweise sondern parallel



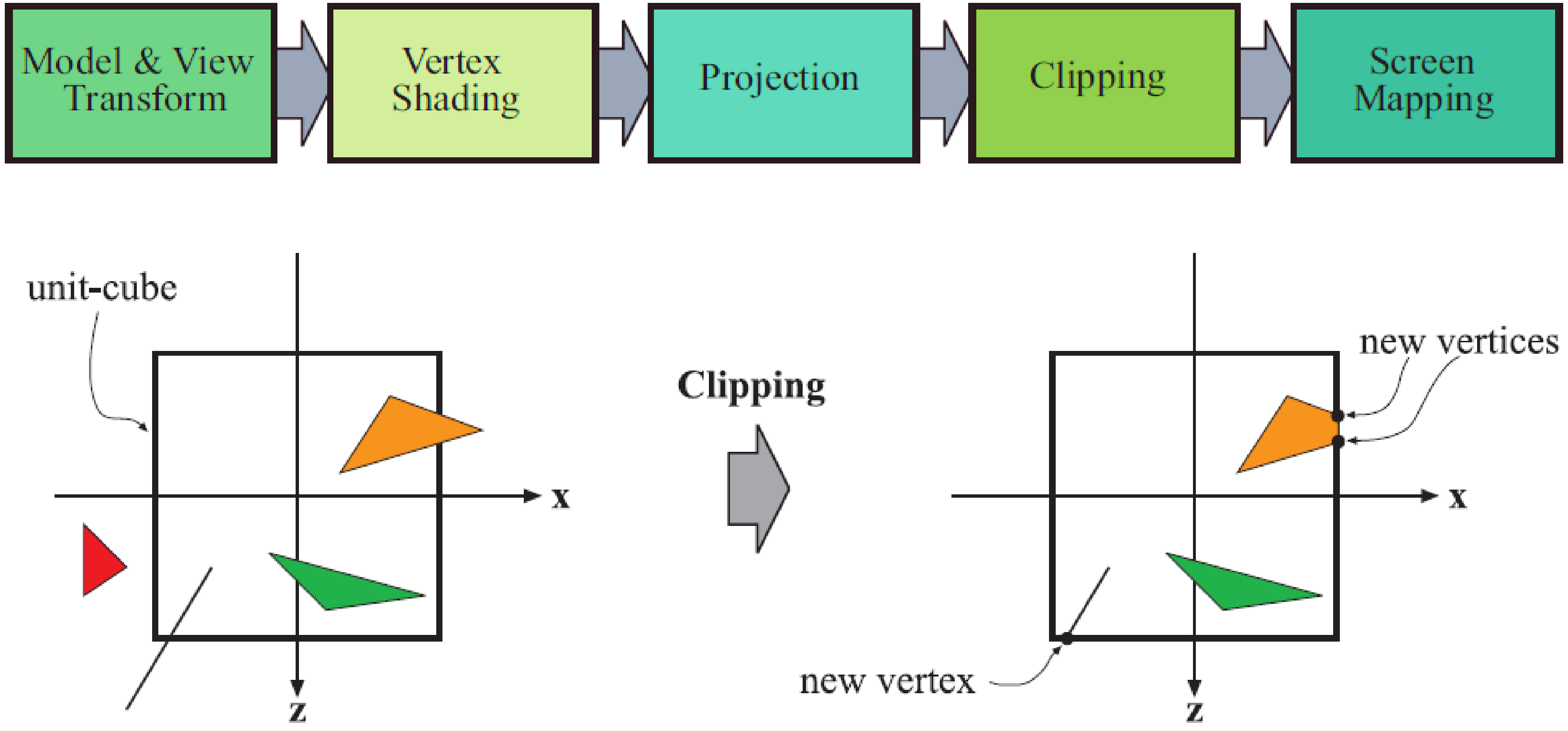



2D-
Objekte
Transformation
(Objekt- nach
Weltkoordinaten)
Clipping
Transformation
(Welt- nach
Bildkoordinaten)
Rasteri-
sierung

Clipping

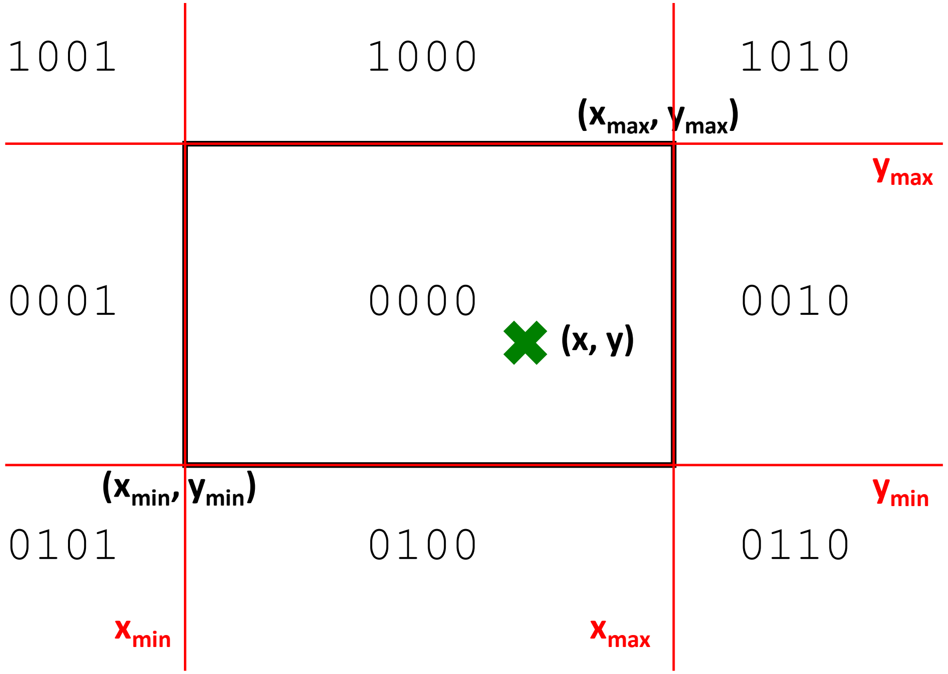

1001
1000
1010
B
D
E
0001
0000
0010
A
C
0101
0100
0110

Primitive Clipping



Ausblick: Einordnung in Verarbeitungspipeline bei 3D-Rendering

Bounding Box und Bereichscode

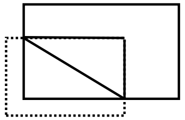


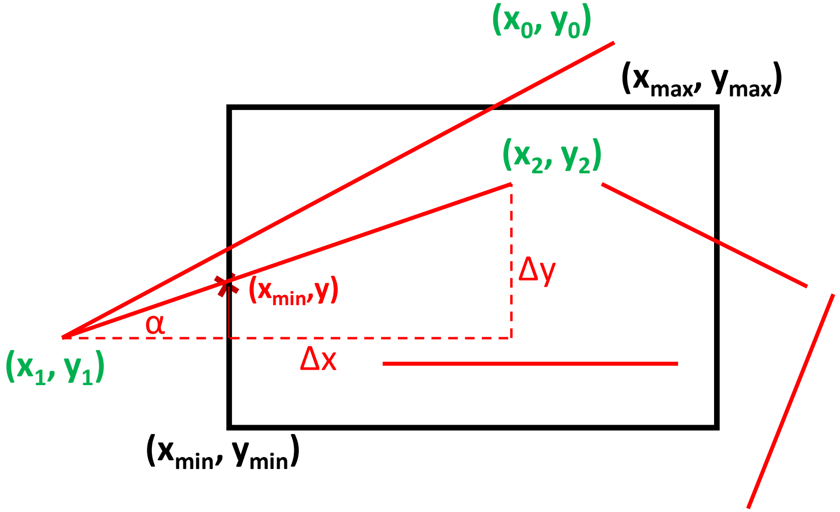

```
int LEFT = 1, RIGHT = 2,
    BOTTOM = 4, TOP = 8,
    MIDDLE = 0;
```

```
int code(float x, float y)
{
    int c = MIDDLE;

    if (x < xmin) c += LEFT;
    if (x > xmax) c += RIGHT;
    if (y < ymin) c += BOTTOM;
    if (y > ymax) c += TOP;

    return c;
}
```

Line Clipping

- Clipping algorithms ensure that elements outside given area are cut off
- A computational efficient method based on coarse preselection was developed by *Cohen & Sutherland*
  - Only works for lines, not with faces!





1. Check, which lines are completely inside or outside
   - → Inside test for both endpoints
   - Inside, if both endpoints inside rectangle
   - Line definitely outside, if bitwise AND (&) of both endpoints' code is *not* 0
2. Else, without loss of generality, calculate intersection for point (x_{min}, y) to the left of clipping rectangle
   - Equation of line: $y = m \cdot x + b$
   - With $m = \dfrac{\Delta y}{\Delta x} = \dfrac{y_2 - y_1}{x_2 - x_1} = \tan(\alpha)$

(intercept theorem)

$$\frac{y - y_1}{x_{\min} - x_1} = \frac{y_2 - y_1}{x_2 - x_1}$$

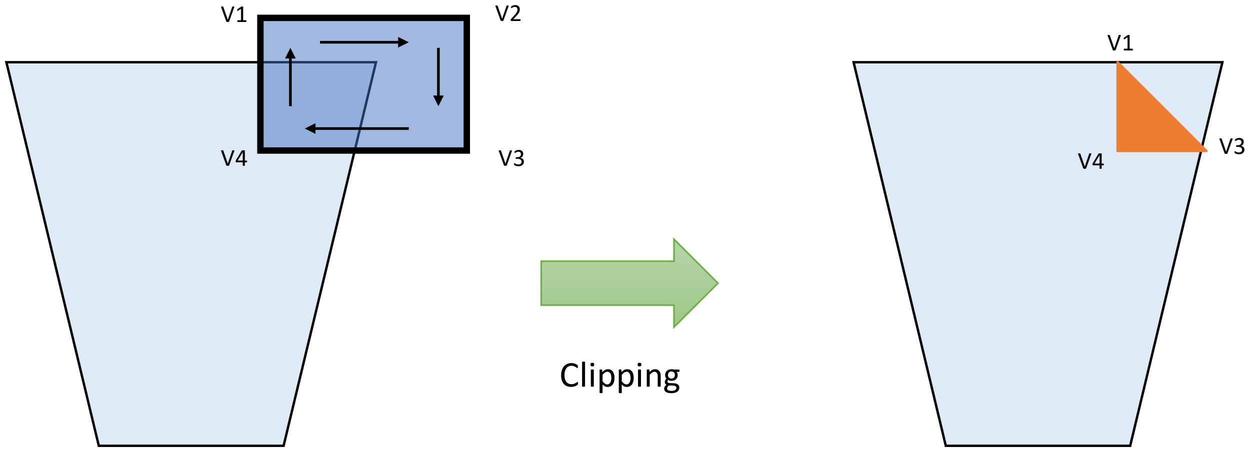
$$\Rightarrow y = y_1 + m \cdot (x_{\min} - x_1)$$

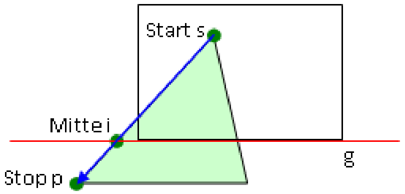
Cohen-Sutherland Line Clipping

```
bool clip(float &x1, float &y1, float &x2, float &y2)
{
  float x, y;
  int c1 = code(x1, y1);
  int c2 = code(x2, y2);

  while ((c1 != MIDDLE) || (c2 != MIDDLE))
  {
   // line completely outside clipping rect
   if ((c1 & c2) != 0)
      return false;

   int c = (c1 == MIDDLE) ? c2 : c1;

   if (c & LEFT) {
      x = xmin;
      y = y1 + (y2-y1) * (xmin-x1)/(x2-x1);
   }
   else if (c & RIGHT) {
      x = xmax;
      y = y1 + (y2-y1) * (xmax-x1)/(x2-x1);
   }
   else if (c & BOTTOM) {
      y = ymin;
      x = x1 + (x2-x1) * (ymin-y1)/(y2-y1);
   }
   else if (c & TOP) {
      y = ymax;
      x = x1 + (x2-x1) * (ymax-y1)/(y2-y1);
   }

   if (c == c1) {
      x1 = x; y1 = y;
      c1 = code(x1, y1);
   }
   else {
      x2 = x; y2 = y;
      c2 = code(x2, y2);
   }
  }

  // line (x1, y1, x2, y2) inside rect -> can draw
  return true;
}
```

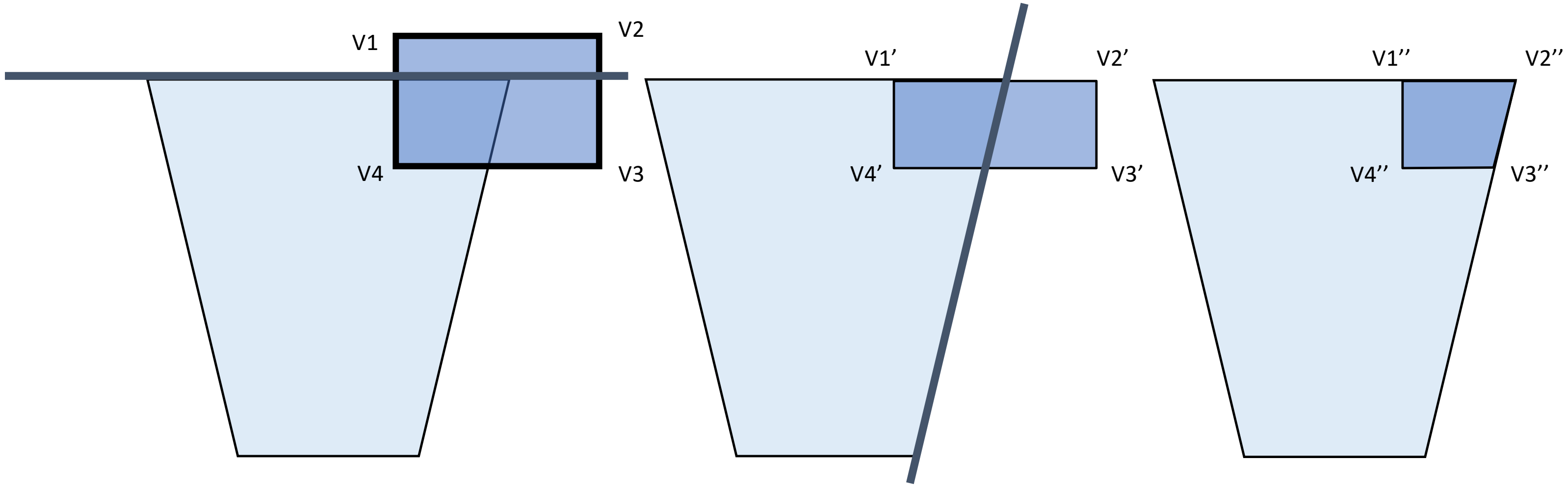
Exkurs: Clippen von Polygonen – Lösung nach Sutherland-Hodgman

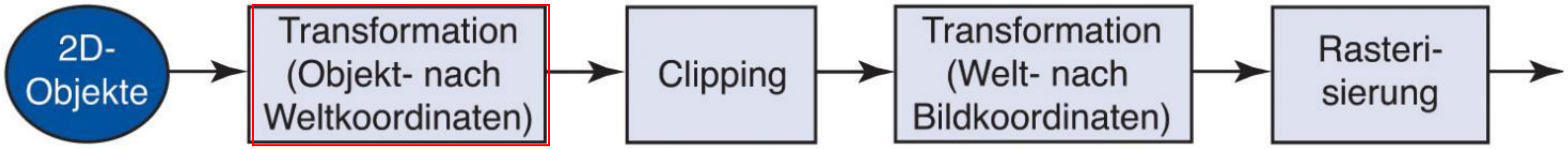
Problem bei
Cohen-Sutherland





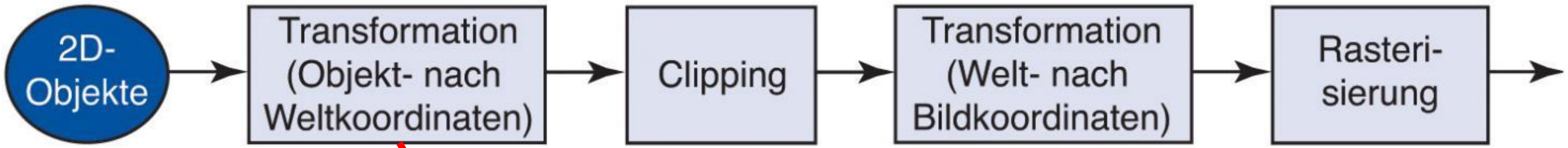
Teste Polygon nacheinander gegen alle Kanten





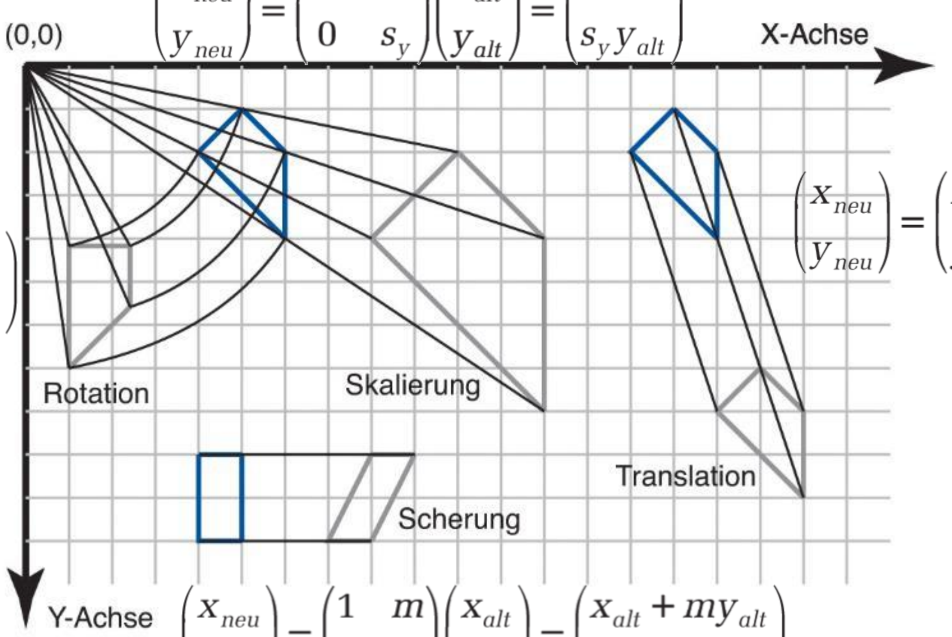
2D-Transformationen

Ausblick: Objekte transformieren



$$\begin{pmatrix} x_{neu} \\ y_{neu} \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x_{alt} \\ y_{alt} \end{pmatrix} = \begin{pmatrix} s_x x_{alt} \\ s_y y_{alt} \end{pmatrix}$$

$$\begin{pmatrix} x_{neu} \\ y_{neu} \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x_{alt} \\ y_{alt} \end{pmatrix} = \begin{pmatrix} \cos\alpha x_{alt} - \sin\alpha y_{alt} \\ \sin\alpha x_{alt} + \cos\alpha y_{alt} \end{pmatrix}$$



$$\begin{pmatrix} x_{neu} \\ y_{neu} \end{pmatrix} = \begin{pmatrix} x_{alt} \\ y_{alt} \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = \begin{pmatrix} x_{alt} + t_x \\ y_{alt} + t_y \end{pmatrix}$$

$$\begin{pmatrix} x_{neu} \\ y_{neu} \end{pmatrix} = \begin{pmatrix} 1 & m \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{alt} \\ y_{alt} \end{pmatrix} = \begin{pmatrix} x_{alt} + m y_{alt} \\ y_{alt} \end{pmatrix}$$

Vielen Dank!

Noch Fragen?