1 Grafikpipeline

1.1 Computer Vision vs. Computer Grafik

Das Kernproblem kann als Inversion betrachtet werden:

- **Computer Vision (CV)**: Prozess von *Rechts nach Links*. Aus Bildern (Pixelrastern) werden Informationen (Objekte, Tiefe) extrahiert. Ziel: Verstehen, was zu sehen ist ("Inverse Grafik").
- **Computer Grafik (CG)**: Prozess von *Links nach Rechts*. Aus abstrakten Daten (Zahlen, Modellen) werden Bilder erzeugt. Ziel: Realistische oder stilisierte Darstellung von Informationen.

1.2 Computing Paradigmen

Die Entwicklung der Hardware beeinflusst die Interaktion und Darstellung:

1. **Mainframe / Large Scale:** Zentralisierte Rechenleistung, Zugriff über Terminals.
2. **Personal Computing (Desktop):** Eigene Rechenleistung, GUI-basiert (WIMP: Windows, Icons, Menus, Pointer).
3. **Networked Computing:** Vernetzung von Systemen (WAN, LAN, Internet).
4. **Mobile Computing:** Smartphones, Tablets. Einschränkungen bei Leistung und Batterie, aber hohe Portabilität.
5. **Collaborative Computing:** Gemeinsames Arbeiten (z.B. Multi-Touch Tables).
6. **Virtual Reality (VR) & Augmented Reality (AR):**

   Virtual Reality (VR): Immersion in eine vollständig virtuelle Welt (z.B. CAVE, Head Mounted Displays).

   Augmented Reality (AR): Nahtlose Integration virtueller Objekte in die reale Welt zur Erweiterung der Wahrnehmung (z.B. HUDs, Smart Glasses, Smartphone-Kamera). Wichtig: Synchronisation mit der echten Welt.
7. **Ubiquitous / Invisible Computing:** Computer verschwinden in der Umgebung (Ambient Intelligence, Wearables, IoT).

1.3 Virtuelle Charaktere und Wahrnehmung

Bei der Darstellung menschenähnlicher Avatare tritt ein psychologisches Phänomen auf:

Uncanny Valley

Das **Uncanny Valley** (Akzeptanzlücke) beschreibt den Effekt, dass die Akzeptanz eines künstlichen Charakters schlagartig abfällt, wenn dieser sehr menschenähnlich ist, aber nicht perfekt wirkt (z.B. Zombies, Leichen). Stilisierte Charaktere (Comic) werden oft besser akzeptiert als fast-realistische.

1.4 Die 3D-Grafikpipeline

Die Grafikpipeline beschreibt den Weg von der geometrischen Beschreibung einer Szene bis zum fertigen Pixelbild auf dem Monitor. Sie wird typischerweise in vier Hauptstufen unterteilt.

1. **Anwendung (CPU):** Modellierung, Eingabe, Szenegraph.
2. **Geometrieverarbeitung (GPU):** Transformationen, Beleuchtung, Clipping.
3. **Rasterisierung (GPU):** Umwandlung von Primitiven in Fragmente (Pixelkandidaten).

4. **Ausgabe:** Darstellung, Speicherung.

1.4.1 Stufe 1: Anwendung (Application)

Hier finden Berechnungen statt, die nicht fest in der Hardware verdrahtet sind (Software-Seite).

- **Eingabe:** Verarbeitung von Nutzereingaben (Maus, Tastatur, Tracking).
- **Modellierung:** Erstellung von 3D-Modellen (aus CT-Daten, Laserscans oder manuellem Design).
- **Primitive:** Die Grundbausteine der Grafik sind **Punkte**, **Linien** und **Dreiecke** (Polygone). Dreiecke sind bevorzugt, da sie immer planar (eben) sind.

Räumliche Datenstrukturen Um Berechnungen zu beschleunigen (z.B. Kollisionserkennung oder Sichtbarkeitstest), werden Hüllkörper und Raumunterteilungen genutzt.

- **Hüllkörper (Bounding Volumes):** Einfache geometrische Formen, die ein komplexes Objekt umschließen.
  - *Kugel (Sphere):* Einfache Distanzberechnung, oft nicht passgenau.
  - *AABB (Axis Aligned Bounding Box):* Achsenparallel, einfach zu berechnen, rotiert nicht mit dem Objekt.
  - *OBB (Oriented Bounding Box):* Passt sich der Rotation an, teurer in der Berechnung.
- **Raumunterteilung (Space Partitioning):**
  - **Gitter (Grids):** Regelmäßige Unterteilung. Problem: Speicheraufwendig bei leeren Räumen.
  - **Quadtree (2D) / Octree (3D):** Hierarchische Unterteilung. Ein Quadrat/Würfel wird bei Bedarf in 4 (bzw. 8) kleinere Einheiten geteilt. Effizient für ungleichmäßig verteilte Szenen.
  - **BSP-Tree (Binary Space Partitioning):** Der Raum wird durch Ebenen rekursiv in zwei Hälften geteilt. Wichtig für den *Painters Algorithm*, da die Zeichenreihenfolge (Vorn/Hinten) vorberechnet werden kann.

1.4.2 Stufe 2: Geometrieverarbeitung (Geometry Processing)

In dieser Stufe werden die Vertizes (Eckpunkte) der Primitive transformiert und beleuchtet.

Transformationen Objekte müssen vom lokalen Koordinatensystem in das Weltkoordinatensystem und schließlich in das Sichtsystem der Kamera überführt werden. Dies geschieht durch affine Transformationen (Translation, Rotation, Skalierung, Scherung) mittels Matrizenmultiplikation.

Beleuchtung (Illumination) Ziel ist die Simulation der Interaktion von Licht und Material. Das **Phong-Beleuchtungsmodell** ist ein lokales Modell (berücksichtigt nur direkte Lichtquellen, keine Reflexionen zwischen Objekten).

Die Lichtintensität I_{total} an einem Punkt setzt sich zusammen aus:

$$I_{total} = I_{amb} + I_{diff} + I_{spec}$$

1. **Ambientes Licht (I_{amb}):** Grundhelligkeit der Szene (indirekte Beleuchtung simuliert durch Konstante).

$$I_{amb} = k_{amb} \cdot C_{amb}$$

2. **Diffuse Reflexion (I_{diff}):** Matte Oberflächen (Lambert-Reflexion). Helligkeit hängt vom Winkel zwischen Lichtvektor L und Oberflächennormale N ab.

$$I_{diff} = k_{diff} \cdot C_{light} \cdot (\vec{N} \cdot \vec{L})$$

3. **Spiegelnde Reflexion (I_{spec}):** Glanzpunkte (Highlights). Hängt vom Winkel zwischen Reflexionsvektor R und Blickvektor V ab. Der Exponent m bestimmt die Rauhigkeit (je höher m, desto kleiner und schärfer der Glanzpunkt).

$$I_{spec} = k_{spec} \cdot C_{light} \cdot (\vec{R} \cdot \vec{V})^m$$

Shading-Verfahren (Interpolation) Wie wird die Beleuchtung auf das gesamte Polygon angewendet?

- **Flat Shading**: Ein Normalenvektor für das ganze Polygon. Eine Farbe pro Polygon. Sieht "kantig" aus.
- **Gouraud Shading**: Beleuchtung wird an den Eckpunkten (Vertizes) berechnet. Die resultierenden *Farbwerte* werden über das Polygon interpoliert. Glatter Verlauf, aber Glanzpunkte können verloren gehen, wenn sie in der Mitte des Polygons liegen.
- **Phong Shading**: Die *Normalenvektoren* werden über das Polygon interpoliert. Die Beleuchtungsgleichung wird für jedes Pixel berechnet. Bestes Ergebnis, aber rechenaufwendigsten.

Clipping & Culling Optimierungsschritte, um nicht sichtbare Geometrie frühzeitig zu verwerfen.

- **Clipping**: Abschneiden von Geometrie, die aus dem Sichtvolumen (Frustum) herausragt.
- **Backface Culling**: Entfernen von Polygonen, die von der Kamera wegzeigen (Rückseiten). *Test:* Skalarprodukt aus Blickrichtung s und Normale n. Wenn $n \cdot s > 0$, ist es eine Rückseite (bei entsprechender Vektor-Definition).

1.4.3 Stufe 3: Rasterisierung

Umwandlung der kontinuierlichen geometrischen Primitive in diskrete Pixel (Fragmente).

Linien-Algorithmen Der **Bresenham-Algorithmus** (1965) ermöglicht das Zeichnen von Linien unter ausschließlicher Verwendung von Integer-Arithmetik (Ganzzahlen).

- *Idee:* Entscheidung, ob das nächste Pixel "rechts" oder "rechts oben" gesetzt wird, basierend auf einem Fehlerterm, der akkumuliert wird.
- Vermeidet langsame Gleitkommaoperationen.

Bresenham-Algorithmus (für 1. Oktant)

Annahme: Linie im ersten Oktanten, d.h. $0 < \Delta y \leq \Delta x$ (Steigung zwischen 0 und 1).
Gegeben: Startpunkt (x_{start}, y_{start}) und Endpunkt (x_{end}, y_{end}).
Vorbereitung:

- Berechne Differenzen: $dx = x_{end} - x_{start}$ und $dy = y_{end} - y_{start}$
- Initialisiere Startpixel: $x = x_{start}$, $y = y_{start}$
- Initialisiere Fehlerterm: $fehler = dx/2$ (Integer-Division)
- Setze erstes Pixel (x, y)

Iteration (Schleife solange $x < x_{end}$):

1. Schritt in die *schnelle Richtung* (hier x):
$$x = x + 1$$
2. Fehlerterm aktualisieren (Idealgerade weicht ab):
$$fehler = fehler - dy$$
3. **Entscheidung:** Ist der Fehler zu groß geworden ($fehler < 0$)?
   - **JA:** Wir müssen auch in die *langsame Richtung* (y) gehen, um der Linie zu folgen.
   $$y = y + 1$$
   $$fehler = fehler + dx$$
   (Fehlerterm korrigieren)
   - **NEIN:** Wir bleiben auf der gleichen y-Höhe.
4. Setze Pixel (x, y)

Polygon-Rasterisierung **Scanline-Algorithmus**: Eine horizontale Linie wandert zeilenweise über das Bild. Schnittpunkte mit Polygonkanten werden berechnet und sortiert. Pixel zwischen Paaren von Schnittpunkten werden gefüllt (Paritätsregel: Umschalten zwischen Malen/Nicht-Malen bei Kantenübertritt).

Verdeckungsrechnung (Visibility) Wie wird bestimmt, welches Objekt vorne liegt?

Z-Buffer Algorithmus

Für jedes Pixel wird neben der Farbe auch ein Tiefenwert (z-Wert) gespeichert.

- Initialisierung: Bildspeicher = Hintergrundfarbe, Z-Buffer = unendlich (max Tiefe).
- Für jedes Pixel eines neuen Polygons: Berechne Tiefe z_{neu}.
- *Test:* Ist $z_{neu} < z_{gespeichert}$?
- *Ja:* Schreibe Farbe in Bildspeicher, aktualisiere $z_{gespeichert} = z_{neu}$.
- *Nein:* Verwerfe Pixel (verdeckt).

Vorteile: Reihenfolge der Objekte egal, einfache Hardware-Implementierung.
Nachteile: Speicherbedarf, Transparenz schwierig, Z-Fighting (Genauigkeitsprobleme).

Alternativ: **Painters Algorithm**. Sortiere Polygone nach Tiefe (weit weg → nah) und zeichne sie übereinander. Problem: Zyklische Überlappungen und komplexe Sortierung ($O(n^2)$).

1.4.4 Stufe 4: Ausgabe

Finales Schreiben in den Framebuffer, Darstellung auf CRT, LCD, Beamer oder Speicherung in Dateien.