panikzettel.htwr-aachen.de

Basic Techniques in Computer Graphics Panikzettel™

Luca Oeljeklaus, Julian Schakib, Christoph von Oy

Version 1 — 04.08.2024

Contents

1 Representing Objects 3

2 Coordinate Systems 4
- 2.1 Extended Coordinates . . . 4
- 2.2 Homogeneous Coordinates . . . 4

3 Mappings 4
- 3.1 Linear and Affine Mappings . . . 5
- 3.2 Projective Mappings . . . 6
- 3.3 Vanishing Points . . . 6
- 3.4 Camera Model . . . 7

4 Clipping 9
- 4.1 Line Clipping . . . 9
- 4.2 Polygon Clipping . . . 10

5 Rasterisation 10
- 5.1 Line Rasterisation . . . 10
- 5.2 Polygon Rasterisation . . . 11
- 5.3 Triangulation . . . 12
- 5.4 Voronoi Diagram . . . 13

6 Lighting 14
- 6.1 Local Lighting . . . 14
- 6.2 Shading . . . 16
- 6.3 Shadows . . . 16

7 Texturing 18
- 7.1 Environment maps . . . 18
- 7.2 Anti-Aliasing . . . 18

8 Polygonal Meshes 20
- 8.1 Properties . . . 20
- 8.2 Euler's Formula . . . 21

8.3 Platonic Solids . . . 21
8.4 Datastructures . . . 22

9 Constructive Solid Geometry **23**
9.1 Quadrics . . . 24
9.2 Operations . . . 25

10 Scene Representation **26**
10.1 Culling . . . 26
10.2 Optimisation structures . . . 27

11 Volumetric Rendering **28**
11.1 Signed Distance Function . . . 28
11.2 Direct Volume Rendering . . . 28
11.3 Indirect Volume Rendering . . . 29

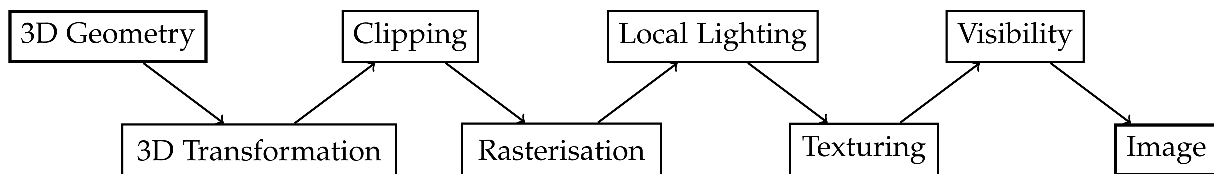
12 Curves **29**
12.1 Bézier Curves . . . 30
12.2 De Casteljau Algorithm . . . 30
12.3 Bézier Spline . . . 30

Introduction

This Panikzettel covers the lecture Basic Techniques in Computer Graphics, held in the winter semester of 17/18 by Prof. Dr. Leif Kobbelt and is mainly based on the content of the lecture, its slides, and previous lecture notes.

This Panikzettel is Open Source. We appreciate comments and suggestions at `https://git.rwth-aachen.de/philipp.schroer/panikzettel`.

The following diagram roughly outlines the rendering pipeline, after which this Panikzettel is loosely structured.



Rendering Pipeline

1 Representing Objects

In general, we use two different forms to represent objects, the parametric and the implicit form.

Simply speaking, we can say that a *parametric function* takes in some variables and gives out a point of the object it defines.

Contrary to this, an *implicit function* takes a point in space as input and returns 0 (hence *kernel*) if it lies within the object and something else if it doesn't.

Definition: Parametric Form

An object is defined through the range of a function:

$$f : D \to \mathbb{R}^3$$

Definition: Implicit Form

An object is defined through the kernel of a function:

$$f : \mathbb{R}^3 \to \mathbb{R}$$

For the parametric form, it is easy to find all points of the object by simply plugging in all values $x \in D$. However it is difficult to determine if a certain point lies within an object.

The opposite holds for the implicit form, where checking if a point lies in an object is a simple test, but finding all points is much more difficult.

Consider a circle with centre $c \in \mathbb{R}^2$ and radius $r \in \mathbb{R}$. A parametric form $f : \mathbb{R} \to \mathbb{R}^2$, that returns points on the edge of the circle, could be

$$f(\alpha) = \begin{pmatrix} c + cos(\alpha) \cdot r \\ c + sin(\alpha) \cdot r \end{pmatrix}$$

To get a point lying on the edge with this formula, one just needs to plug in an angle.

But what if one wants to check whether a *given* point lies on the edge? One would have to check the point against all possible results of the above formula to be sure. For that case an implicit form works a lot better:

$$||p - c|| - r = 0$$

Now to check if a point $p \in \mathbb{R}^2$ lies on the edge of the circle it can be simply plugged into the above formula.

It is clear that the choice of a representation depends on the application.

2 Coordinate Systems

2.1 Extended Coordinates

To distinguish points from vectors, we can *extend* the coordinates to include a fourth one. Points will have 1 as their fourth coordinate while vectors will have 0.

These coordinates are mathematically reasonable as the addition of two vectors and subtraction of two points yield vectors while the addition of a point and a vector yields another point. However, the addition of two points is ambiguous.

Definition: Extended Coordinates

Using extended coordinates, the point $p = (p_1, p_2, p_3)^T \in \mathbb{R}^3$ and vector $v = (v_1, v_2, v_3)^T \in \mathbb{R}^3$ are noted as:

$$p = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix}, \quad v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix} \qquad p, v \in \mathbb{R}^4$$

2.2 Homogeneous Coordinates

Homogeneous coordinates allow *storing* dividends in the fourth component of a point by multiplying them.

Thus, divisions accumulate and in a last step called *de-homogenisation* the point is divided by that fourth component. As such, divisions can be treated as *multiplications*, which allows concatenating projective matrices with other matrices as explained in the following.

Definition: Homogeneous Coordinates

With homogeneous coordinates, we represent a point $p = (p_1, p_2, p_3)^T \in \mathbb{R}^3$, for any $w \neq 0$ as follows:

$$p = \begin{pmatrix} wp_1 \\ wp_2 \\ wp_3 \\ w \end{pmatrix} \in \mathbb{R}^4$$

3 Mappings

In the following section, we will be introducing mappings which we will formalise through matrices. Applying such a mapping to a point or a vector can then be written as a matrix multiplication with the matrix on the left-hand side.

Multiple mappings can be applied consecutively by multiplying them in the correct order. For example, let $M_1, \ldots, M_n \in \mathbb{R}^4$, be $n \in \mathbb{N}$ mappings and $p \in \mathbb{R}^4$ a point. Then $p' = M_n \cdot \ldots \cdot M_1 \cdot p$ is the result when applying these mappings to p, with M_1 being applied first and M_n applied last. The order obviously matters, as matrix multiplication is in general not commutative.

3.1 Linear and Affine Mappings

Scaling

A scaling operation multiplies each axis by a given factor. Choosing a factor greater or smaller than 1 results in stretching or shrinking respectively. A factor of exactly 1 results in no change at all.

It is obvious that lengths are (generally) not preserved, and that angles are only preserved if every axis is scaled by the same factor.

Definition: Scaling

This scaling matrix $S(\alpha, \beta, \gamma) \in R^{4\times4}$ scales the x, y and z axes by factors of α, β and γ respectively:

$$S = \begin{pmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation

The rotation operation is used to rotate an object around an axis. In general, any axis can be used for rotation. Though, since this is rather complex, we mostly rotate around the 3 main axes.

Definition: Rotation

In $\mathbb{R}^3$ space we have three intuitive axes around which we can rotate, with rotation matrices $R_x(\alpha), R_y(\alpha), R_z(\alpha) \in \mathbb{R}^4$ for an angle of α given as follows:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To rotate around an arbitrary axis $n = (n_x, n_y, n_z, 1)^T, ||n|| = 1$ we use the following formula:

$$R(n, \alpha) = \cos\alpha \cdot I + (1 - \cos\alpha) \cdot nn^T - \sin\alpha \begin{pmatrix} 0 & n_z & -n_y & 0 \\ -n_z & 0 & n_x & 0 \\ n_y & -n_x & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation preserves lengths and angles but changes orientation and position.

Translation

Translating an object comes down to shifting its position by a given additive offset. This of course preserves lengths, angles and orientation, but naturally not position.

Definition: Translation

To translate by an offset of $t = (t_1, t_2, t_3)^T \in \mathbb{R}^3$, we write our translation matrix $T(t) \in R^{4\times4}$ as follows:

$$T(t) = \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3.2 Projective Mappings

For the subject of projective mappings, we will be using homogeneous coordinates.

Standard Projection

The standard projection is defined by placing the camera at the origin, with the viewing vector being $(0, 0, -1)^T$ and the image plane being $z = -1$.

Projecting the point $(x, y, z, 1)^T$ using standard projection and then dehomogenizing will yield

$$(\frac{x}{-z}, \frac{y}{-z})^T.$$

Definition: Standard Projection

The matrix for the standard projection looks as follows:

$$P_{std} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Arbitrary Camera or Image Plane

All of this being a bit boring, we can also select an arbitrary image plane, defined through $n = (n_x, n_y, n_z)^T$ and a focal distance δ, while keeping our camera at the origin but pointed at the image plane.

Definition: Arbitrary Image Plane

For an arbitrary image plane, we have the following projection matrix:

$$P_{aip} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{n_x}{\delta} & \frac{n_y}{\delta} & \frac{n_z}{\delta} & 0 \end{pmatrix}$$

We can also do the opposite by picking an arbitrary camera position, keeping the image plane passing through the origin and perpendicular to the viewing vector.

All of these projections can be deduced mathematically, but we're not going to do this here.

Definition: Arbitrary Camera Position

The corresponding projection matrix for an arbitrary camera position with a focal distance of δ and a position n is given by:

$$p_{acp} = \begin{pmatrix} 1 - n_x^2 & -n_x n_y & -n_x n_z & 0 \\ -n_x n_y & 1 - n_y^2 & -n_y n_z & 0 \\ -n_x n_z & -n_y n_z & 1 - n_z^2 & 0 \\ \frac{n_x}{\delta} & \frac{n_y}{\delta} & \frac{n_z}{\delta} & 1 \end{pmatrix}$$

3.3 Vanishing Points

Due to perspective foreshortening, parallel lines can meet at a certain distant point, the *vanishing point*. You can observe this in nature: Take for example train rails that run parallel. When you look down the tracks you will notice that the two rails, although actually apart, meet at the horizon.

To find a line's vanishing point on the image plane, we can simply intersect the line with the image plane, starting from the viewer's position. All other lines that are parallel to this line will, at infinity, appear at the same position. Also, lines that run parallel to the image plane do not have a vanishing point.

Why is that? Let's have a look at a line in explicit representation: $L(\lambda) = (x, y, z)^T + \lambda \cdot (d_x, d_y, d_z)^T$. When applying the standard projection from earlier we get

$$P_{std}(L(\lambda)) = \begin{pmatrix} \frac{x+\lambda d_x}{-z-\lambda d_z} & \frac{y+\lambda d_y}{-z-\lambda d_z} \end{pmatrix}^T.$$

In case $d_z = 0$, which means that the line runs *parallel to the image plane*, $\lim_{\lambda\to\infty} P_{std}(L(\lambda)) = \infty$, which we interpret as the lack of a vanishing point.

However, when $d_z \neq 0$, then $\lim_{\lambda\to\infty} P_{std}(L(\lambda)) = \begin{pmatrix} \frac{d_x}{-d_z} & \frac{d_y}{-d_z} \end{pmatrix}$ is the vanishing point of that line. So apparently the position of the vanishing point of a line depends on its *direction only*, not its position, thus all parallel lines share the same vanishing point.

3.4 Camera Model

Look-At Transformation

The first of our three transformations is the *look-at transformation* which performs a basis change such that the camera is located at the origin and set up as the standard projection.

Definition: Look-At Transformation

Given a camera position of c and orthonormalised direction, up and right vectors d, u and r, the look-at transformation comes down to applying the following matrix to the scene:

$$M_{LookAt} = \begin{pmatrix} r_x & r_y & r_z & -r^T c \\ u_x & u_y & u_z & -u^T c \\ -d_x & -d_y & -d_z & d^T c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Algorithm: Look-At Transformation

Input: Camera position c, direction and up vectors d and u.
Output: A frustum-able scene with the camera in standard projection setup.

1. Generate *right* vector $r = d \times u$ as the cross product of d and u.
2. Calculate new *up* vector $u = d \times r$.
3. Normalise d, u and r.
4. Translate the whole scene by $-c$. The camera is now at $(0, 0, 0, 1)^T$.
5. Rotate the scene such that $d = (0, 0, -1, 0)^T$.
6. Rotate the scene such that $u = (0, 1, 0, 0)^T$ and $r = (1, 0, 0, 0)^T$.

Frustum Transformation

The *frustum transformation* is the second one. After having defined a viewing frustum, the transformation compresses it into the $[-1,1]^3$ cube.

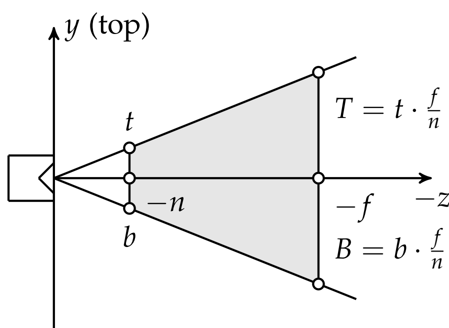
Such a viewing frustum can be thought of as a pyramid with its top at the camera, truncated by a near plane at the top and a far plane at the bottom, both orthogonal to the z-axis.

It is defined by the coordinates of its top, bottom, left and right edges on the near plane, or t, b, l and r respectively.

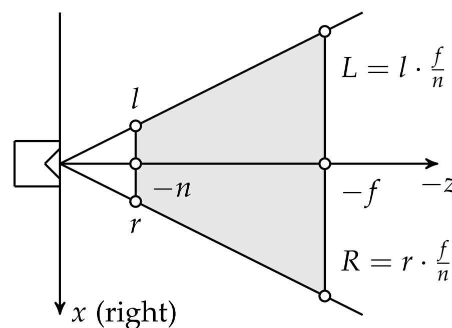
Definition: Frustum Transformation

Given a viewing frustum with far plane $z = -f$, near plane $z = -n \geq -f$ and left, right, top and bottom coordinates l, r, t and b, the frustum transformation is applied to a scene with the following matrix:

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



Frustum sideview



Frustum topview

Viewport Transformation

To get the scene from the cube to our screen we apply a parallel projection in which we simply neglect the z-coordinate. Our image now has coordinates $[-1,1]^2$. But a screens aspect ratio is usually different from $1:1$, for example $16:9$, so to display the image we have to *scale* it and *add an offset* by applying the *window-to-viewport* map.

After de-homogenisation, this gives us homogeneous coordinates which represent screen coordinates.

Definition: Viewport Transformation

Given a screen width w and height h, the viewport transformation can be applied as the following matrix:

$$M_{viewport} = \begin{pmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} + l \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} + b \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4 Clipping

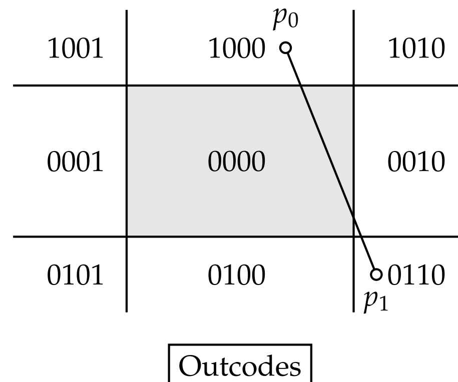
4.1 Line Clipping

Cohen-Sutherland Algorithm

The *Cohen-Sutherland algorithm* uses *outcodes*. An outcode, defined for a point, consists of 4 bits, one for each border of the bounding box, ordered like this: top, bottom, right, left (TBRL).

If a point lies beyond a certain border line the corresponding bit is activated, as shown in the diagram.

For a line segment there are two points through which we can test if it is fully contained or fully outside in the box.



Outcodes

If two outcodes fulfil $(o_1 \wedge o_2) \neq 0$, then their line lies completely beyond at least one border, thus it is rejected. If $(o_1 \vee o_2) = 0$, then their line is completely contained in the bounding box.

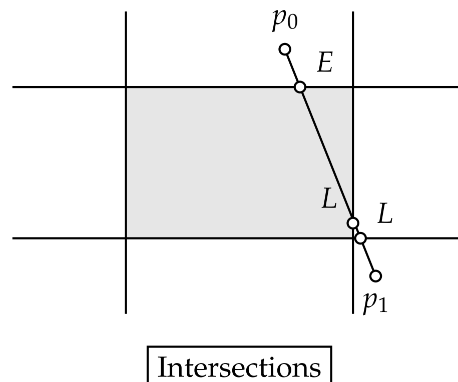
If neither is fulfilled, the line may or may not be partially contained in the bounding box and while it would be possible to determine this with this algorithm, it is computationally expensive.

The Liang-Barsky Algorithm

The *Liang-Barsky algorithm* uses the parametric representation of a line segment between p_0 and p_1:

$$L(\lambda) = (1-\lambda)p_0 + \lambda p_1, \quad \lambda \in [0,1].$$

For each boundary line it calculates the parameter λ' at which the line crosses it and updates the boundaries of the line. Then, if the point enters a boundary, we use λ' to update λ_0, else we update λ_1.



Intersections

The entry test is done by calculating the direction $d = p_1 - p_0$, then calculating $n^T d$, n being the boundary's normal vector. If it's negative, it's entering the boundary, else it's exiting it.

In the end, we obtain a parameter boundary $\lambda \in [\lambda_0, \lambda_1]$. If $\lambda_0 \leq \lambda_1$, the line intersects the box and we render the segment, else we don't.

4.2 Polygon Clipping

Sutherland-Hodgman Algorithm

The *Sutherland-Hodgman algorithm* is an extension of the Cohen-Sutherland algorithm. What it does is reducing clipping against the whole rectangle to clipping against boundary after boundary. For each boundary it clips every edge of the polygon, with four possible outputs.

This is done through a pipeline: the output of the first boundary is fed into the second etc.

- *Inside (the current boundary)*: we return the edge between both vertices.
- *Outside (the current boundary)*: we return nothing.
- *Leaving*: we return the edge between the first vertex and the intersection.
- *Entering*: we return the edge between the intersection and the second vertex.

Liang-Barsky Algorithm

The Liang-Barsky algorithm for polygons is just an extension of the line version. It first clips each edge of the polygon and then, if necessary, inserts further edges between the boundary intersections and, further, if clipping takes place around the corners, also new vertices in the corners.

5 Rasterisation

5.1 Line Rasterisation

We recall the parametric and implicit representations of a line given by its endpoints $(x_0, y_0)^T$ and $(x_1, y_1)^T$.

Further, we limit our lines to slopes in $[0, 1]$, as all other slopes can easily be mirrored.

$$y = mx + t, \quad m = \frac{\Delta y}{\Delta x}, \quad t = y_0 - mx_0$$

$$ax + by + c = 0$$

$$a = \Delta y, \quad b = -\Delta x, \quad c = \Delta xt$$

Digital Differential Analysis

Digital Differential Analysis (DDA) calculates pixel coordinates by incremental addition and subsequent rounding of y.

This is better than calculating y by multiplication of x, but is still computationally costly as we still have to round y.

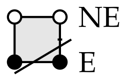
Algorithm: DDA

Input: A line given by its endpoints (x_0, y_0) and (x_1, y_1).
Output: None. (though it activates pixels)

1. While $x \leq x_1$:
2. a) $x := x + 1, \quad y := \lfloor y + m \rceil$.
   b) setPixel(x, y).

The Bresenham Midpoint Algorithm

Since our slopes are restricted to $[0, 1]$, the next pixel of the slope can be either *East* or *North East*.



Thus, we choose to activate either E $(x_i + 1, y_i)$ or NE $(x_i + 1, y_i + 1)$ by checking if the line passes above or below the midpoint $M = \frac{\text{E}+\text{NE}}{2}$.

We do this by simply evaluating for the implicit representation M $F(M) = F(x_i + 1, y_i + \frac{1}{2})$. This is positive if we are below, and negative if we are above the line.

Algorithm: Bresenham Midpoint Algorithm

Input: A line given by its endpoints (x_0, y_0) and (x_1, y_1).
Output: None. (though it activates pixels)

1. $a := y_1 - y_0,\ \ b := x_0 - x_1,\ \ d := 2a + b,$
2. $\Delta\text{E} := 2a,\ \ \Delta\text{NE} := 2(a + b)$
3. While $x \leq x_1$:
4. a) setPixel(x, y), $x := x + 1$
   b) If $d \leq 0:\ \ d += \Delta\text{E}$
   c) Else: $d = d + \Delta\text{NE},\ \ y := y + 1$
5. setPixel(x, y)

But calculating $F(M)$ for each step is costly. So, we save the variable $d = F(M) = a + \frac{b}{2}$ and increment it with each step.

Then, if E is chosen we update as follows:

$$
\begin{aligned}
d &= F(x_i + 2, y_i + \frac{1}{2}) \\
&= F(x_i + 1, y_i + \frac{1}{2}) + a \\
&= d + a
\end{aligned}
$$

If NE is chosen, we update like this:

$$
\begin{aligned}
d &= F(x_i + 2, y_i + \frac{3}{2}) \\
&= F(x_i + 1, y_i + \frac{1}{2}) + a + b \\
&= d + a + b
\end{aligned}
$$

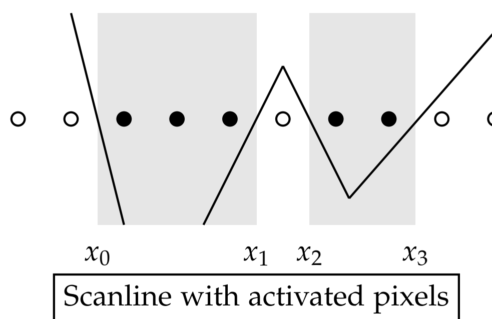
In real life, we multiply everything by 2 so that we can do our calculations using only integers.

5.2 Polygon Rasterisation

Scanline Conversion

Scanline conversion is pretty straightforward. After sorting our polygons by their highest value y-points, we traverse all scanlines (horizontal pixel lines) and check for intersection with these polygons.

While doing this, we compute the fill inside the polygons by sorting the intersection points and filling the spans between odd and even intersections. But we have to cover some edge cases.

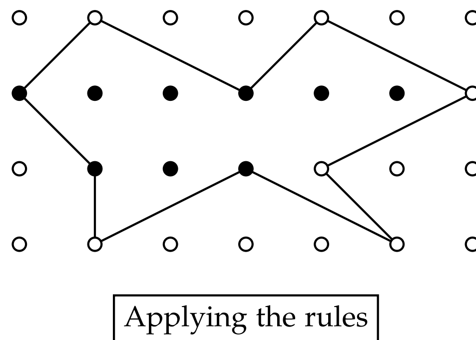


Scanline with activated pixels

We have to set horizontal spans to be half-open on the right, so that polygons with a common edge do not overlap or form gaps.

Similarly, we have to define vertices to be half-open at the top, which leads to two intersections if it is a bottom-end, none if it is a top-end and one if it is a side end.

Horizontal edges, although possibly crossing scanlines in an infinity of points, are covered the the half-openness of edges and thus generate no points.



Applying the rules

For this algorithm to be efficient, we need to apply a few further tricks. So, for each edge, we determine if they cross the current scanline by comparing their y-span to the the scanline. For this, we keep two ordered, appropriately named lists, the *passive* and the *active list*, which are sorted by y_{min} and y_{max} respectively.

The active list is empty at the beginning, and once we enter the scanline $y + 1$, we shift all edges with $y_{min} = y + 1$ from the passive to the active list. They are then discarded once $y_{max} \leq y + 1$.

Pineda Method

To rasterise **convex** polygons, we use the *Pineda method*. We take a list of its counterclockwise vertices $v_0, \ldots, v_n$ and define its edges E_i, $1 \leq i \leq n$ correspondingly. $F_i(x, y)$ being the implicit representation of such an edge, we compute, for any pixel candidate (x, y), the tuple $(F_1(x, y), \ldots, F_n(x, y))$. This pixel will be inside this polygon iff, $\forall\, i \in \{ 1, \ldots, n \} : \ F_i(x, y) \leq 0$.

5.3 Triangulation

Marching (Corner Cutting)

Conceptually, Corner Cutting is about taking a polygon, selecting a vertex, creating a triangle with its neighbours and removing it until only one triangle is left. The difficulty lies in is choosing a triangle that is actually *completely inside* the polygon.

First, we need to select a convex corner. Given the triangle $\Delta(p_{i-1}, p_i, pi + 1)$, this can be determined by the sign of the cross product $(p_{i-1} - p_i) \times (p_{i+1} - p_i)$. Which sign it needs is determined by the point with the lowest x-coordinate, since its triangle **must** be convex.

We must also check that no vertex lies within the triangle we want to cut of, so we check that, too, for each vertex, or better, only for the concave vertices.

Delaunay Triangulation

A very nice form of triangulation is the *Delaunay Triangulation*. Its main characteristic is that, for any triangle, the respective circumcircle does not contain any other point. This leads to a very regular triangulation.

The algorithm to create such a triangulation uses three operations to achieve this:

- *1-3 Split*: A vertex that lies strictly within a triangle is connected to the three vertices.
- *2-4 Split*: A vertex that lies on the edge between two triangles destroys that edge and is connected to all four nodes.
- *Edge Flip*: Given two triangles sharing an edge, if the circumcircle of one of the triangles contains the remaining point of the other triangle, that shared edge is flipped to connect the two vertices that weren't connected.

Algorithm: Delaunay Triangulation

Input: A set of points $\{ p_1, ..., p_n \}$.
Output: A Delaunay Triangulation of the point set.

1. Add three further points around the point cloud of $p_1, ..., p_n$ and create edges between them.
2. For each point from the set:
   a) If it lies within a triangle, perform a 1-3 Split.
   b) If it lies within a tron the edge of two triangles, perform a 2-4 Split.
   c) Restore the Delaunay property with Edge Flips.
3. Finally, delete the three points and corresponding edges and faces that were added in the first step.

5.4 Voronoi Diagram

A *Voronoi Diagram* is the dual mesh to a Delaunay triangulation. What it does is splitting the space into Voronoi regions. Given a set of points, the Voronoi region of a point is the *convex* area around it to which it is the closest vertex.

Sweepline

The sweepline algorithm generates a Voronoi diagram for a given set of points. Its main aspect is the detection of *circle* and *point* events.

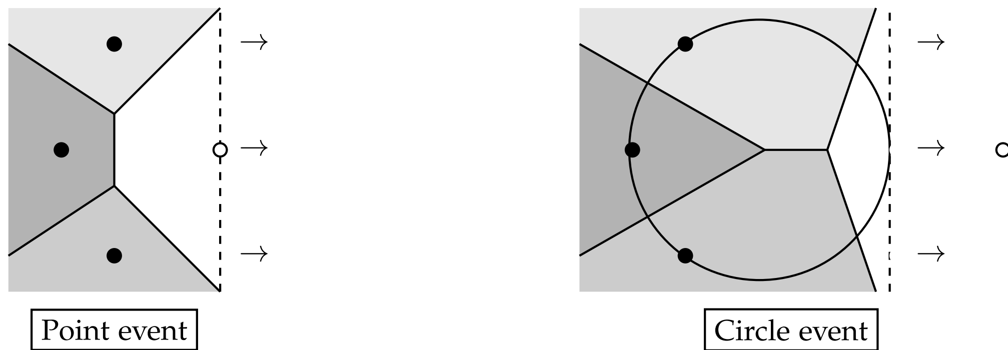
A circle event occurs when the sweepline reaches a point where three points that have already been discovered form a circle without another point being detected.

A point event occurs when the sweepline detects a new point before such a circle can be formed.

Algorithm: Sweepline Algorithm

Input: A set of points $\{ p_1, ..., p_n \}$.
Output: A Voronoi Diagram of the point set.

1. Sort points by x-coordinate.
2. *Sweep* along the x-axis.
3. Detect and process point and circle events.



The two diagrams above illustrate point and circle events. Note that the dashed line and the arrows symbolise the sweepline and its movement.

6 Lighting

6.1 Local Lighting

With the Phong Model, we can compute the lighting of any point p of colour α and normal n, given a viewing position v and light source position l.

The material property α is given by a diagonal matrix $\alpha \in \mathbb{R}^{3\times 3}$ with scalar entries $\alpha_r, \alpha_g, \alpha_b \in [0,1]$ for the red, green and blue component respectively.

The light intensities C are given by three dimensional colour vectors.

On the right, we preemptively introduce the full formula used by the Phong model. In the following sections we will analyse its components.

Definition: Phong Model

The local lighting of the Phong model is given by:

$$C_{orig} = C_a \cdot \alpha_a + \sum_l \Big(\text{spot}(p,l) \cdot \text{att}(p,l) \cdot \left[C_d(p,n,l) + C_{sp}(p,n,v,l) \right] \Big)$$

It can be decomposed into *ambient lighting*, *diffuse lighting* and *specular lighting*, as well as *spotlight* and *attenuation effects*. We will first introduce the three different forms of lighting and then the secondary effects.

Ambient Lighting

Since usually no scene is completely dark, *ambient lighting* independent of light source or the viewers position is determined only by surface materials and ambient light intensity.

Definition: Ambient Lighting

Given C_A, the ambient light intensity, the ambient lighting is given by:

$$C_a = \alpha_a \cdot C_A$$

Diffuse Lighting

Diffuse Lighting models the reflection on matte or rough surfaces. It can also be formulated as the amount of light hitting a certain point from a certain light source.

While dependent on the material, it is also dependent on the relative position of the light source to the surface.

Definition: Diffuse Lighting

Given a light source emitting the light C_l, the diffuse lighting is given by:

$$C_d(p, n, l) = \alpha_d \cdot C_l \cdot \frac{n^T \cdot (l - p)}{||l - p||}$$

Specular Lighting

Specular lighting models the behaviour of light on a shiny surface, off which it is reflected with the same angle with which it hit it. The outgoing vector is given by r. Thus, it is of course dependent on the position of the light source, but also on the position of the viewer.

We also need to take into account the *shininess exponent* s, since there are varying degrees of shininess. The higher s is, the shinier the surface, and thus the preciser the reflection is.

As the classic formula is pretty expensive to compute, it is easier to approximate it using the *Phong-Blinn approximation*.

Definition: Specular Lighting

$$C_{sp}(p, n, v, l) = \alpha_{sp} \cdot C_l \cdot \left(\frac{r^T(v - p)}{||r|| \cdot ||v - p||} \right)^s$$

with

$$r = (2nn^T - I)(l - p)$$

Definition: Phong-Blinn Approximation

$$C_{sp}(p, n, v, l) \approx \alpha_{sp} \cdot C_l \cdot \left(n^T h \right)^s$$

with

$$h = \frac{v + l}{||v + l||}$$

Attenuation

The attenuating factor for a point, given a light source, is about the distance of the point from the light source; the greater the distance between them, the weaker the reflection should be.

For a more accurate representation, we do not only consider a linear, but also a quadratic attenuation.

Definition: Attenuation

Given linear and quadratic attenuation factors att_{lin} and att_{quad}, the attenuation for a point and a light source is given by:

$$\text{att}(p, l) = \frac{1}{\text{att}_{\text{lin}} \cdot ||p - l|| + \text{att}_{\text{quad}} \cdot ||p - l||^2}$$

Spotlight

If, instead of our usual light sources which emit in all directions equally, the light source at l is a spotlight emitting light focused in a direction d then the further we are away from this vector, the weaker the reflection will be.

The impact of that distance is given by the exponent f.

Definition: Spotlight

The spotlight factor of a light source, given a direction d and an exponent f is defined as:

$$\text{spot}(p, l) = \left(\frac{d^T(p - l)}{||p - l||} \right)^f$$

Depth Cueing

In outdoor scenes we want to be able to model the blue-gray of far away objects. This is done through depth cueing. Factoring in the distance to the viewer b, it overlays the colour of an object with such a filter.

Definition: Depth Cueing

Given a blue-gray colour filter C_{dc}, we can model an atmosphere like this:

$$C_{final} = b \cdot C_{orig} + (1 - b) \cdot C_{DC}$$

6.2 Shading

Shading algorithms are applied to give the impression of depth by varying levels of darkness on objects. We take a look at three possible approaches in the following.

Gouraud shading

Gouraud shading computes lighting values *per vertex* and interpolates them over a polygon, i.e. for each fragment on the polygon, the colour values from the vertices are interpolated.

While Gouraud shading works great for shading surfaces that reflect light diffusely it may not always work great with specular light. Its shape depends on the underlying polygons and the quality of it thus depends on the number of vertices of your model. It is computationally less expensive than Phong shading though.

Flat shading

Flat shading computes lighting values for *one vertex* per polygon and uses the resulting colour for the entire polygon, resulting in a single flat colour for every polygon. The individual polygons can be seen.

Phong shading

Not to be confused with the Phong lighting model, Phong *shading* interpolates the lighting parameters across the polygon and computes the lighting *per fragment*, not per vertex.

Although more computationally expensive than Gouraud shading, you get good looking, round, smooth specular highlights that move smoothly along the surface as the camera, model or light moves. No visible artefacts from the polygon edges.

6.3 Shadows

To introduce the effects of shadows, a term $S(p, l)$ is added to the lighting equation that tests if the point p is in the shadow of the light source at l:

$$S(p, l) = \begin{cases} 0, & \text{if the light } l \text{ is blocked at point } p, \\ 1, & \text{else.} \end{cases}$$

Then the colour can be computed as:

$$C_{orig} = C_a \cdot \alpha_a + \sum_l \left(\text{spot}(p, l) \cdot \text{att}(p, l) \cdot \text{S}(p, l) \cdot \left[C_d(p, n, l) + C_{sp}(p, n, v, l) \right] \right)$$

Shadows are caused by objects blocking light. What we effectively perform are visibility tests from the perspective of the light source. We could (pre-)compute shadows based on the light and the geometry of occluding objects (occluders) or shadow textures, but this approach is usually too slow for real-time applications. Instead of using two-dimensional shapes, projected on occluded objects (occludees), we compute the *three-dimensional region* that lies in the shadow of an object and light source, for example as *Shadow Volumes* which are described in the following.

Shadow Volumes

A Shadow Volume is a cone-like volume to infinity, starting at the silhouette of an object, cast by a light source. A silhouette of an object can be easily found as it is built by the edges that separate front- and back-facing polygons w.r.t. the light source. The number of shadow volumes we need to compute is $\#occluder \cdot \#lights$.

During visibility tests we raycast through each pixel of the final image into the scene onto an intersection point to find the closest polygon. We can extend this and check whether this point lies within a volume by simply counting all intersections of the ray with shadow volumes. For every volume entered or left by the ray, we increment or decrement the counter respectively. If the counter is non-zero as we reach the intersection point, it lies inside of (at least) one shadow volume.

This method fails if the camera is positioned inside a shadow volume and is referred to as *z-fail*. To avoid this issue, we add a back-cap to each volume and instead of tracking the ray intersections with volumes *in front of* p, we count the intersections *behind* p, with incrementation and decrementation switched.

Shadow Maps

By rendering the scene as seen from the position of the light source and storing the contents of the resulting Z-Buffer, we get so called Shadow Maps that store the distances between the light source and occluders.

To check whether a point p lies inside of a shadow, we simply project it onto our shadow map with the model-view and projection matrix used to create the shadow map. We can now compare the depth values: If the depth of p is greater than the stored value on the shadow map, the point lies behind another object and the light has no effect on its colour.

Perspective Shadow Maps

The resolution of the shadow map is the resolution of the depth buffer. This means that many fragments get projected onto the same pixel of the shadow map. This effect is expressed by the formula on the right.

Definition: Perspective Shadow Maps

$$h_s \cdot \frac{r_s}{r_i} \cdot \frac{cos(\beta)}{\cos(\alpha)}$$

The first fraction is perspective aliasing and the second projective aliasing. Calculating the shadow map using normalised device coordinates reduces the perspective aliasing to 1. The projective aliasing can not be reduced in general.

Properties of Shadow Volumes

- Precise shadows, little aliasing
- Works with omnidirectional lights
- Requires closed mesh
- Need to be rendered twice for each frame, no hardware support, complexity and amount of occluders can make this *slow*

Properties of Shadow Maps

- Very general method
- Resource hungry
- Incompatible with omnidirectional lights
- Discrete, thus imprecise; artefacts
- Can be done on hardware, *fast*

7 Texturing

The art of texturing is to map a 2-dimensional texture onto the surface of a mesh. This *U-V-Mapping*, called this way because the texture coordinates are labelled u and v, is a big topic. The original topic of textures was to give a model more colour detail.

In more complex materials, U-V-Mapping can project a *map* onto the geometry, assigning a material property value, like the specular value, to each point on the geometry.

7.1 Environment maps

Using texture maps we can also precompute the approximate reflection of the environment of e.g. a sphere. All we need is a photograph of such a proxy reflecting the environment as the texture map. Now, given a *normalised* direction vector d of the surface, we can easily map the proper texture colour onto it:

1. Add 1 to the z-coordinate of d.
2. Normalise d.
3. Use the x- and y-coordinates of d to look up colour in the texture.

Magnification & Minification

When projecting the scene onto the image plane using nearest-neighbour, we may face two issues:

1. Magnification: many screen pixels display only few texels, so texels will appear jagged.
2. Minification: many texels are mapped to one pixel (which can only take one colour), which results in Moiré patterns.

A solution to this is to use *bilinear interpolation* that looks up the colour interpolating multiple colours.

7.2 Anti-Aliasing

Aliasing are a set of effects that occur during different steps of the rendering pipeline:

- **Texture alias:** For the computation of the colour of a fragment based on a texture, only one point is sampled from the texture even if an area is projected onto the fragment. This leads to moire-like patterns if the area is large and to hard edges between individual texels if the area is small.
- **Geometry alias:** To determine if a triangle covers a pixel, one point is sampled and a binary decision is made. This leads to jagged edges when two triangles with stark colour differences meet.
- **Shader alias:** If a texture creates an edge within a polygon, for example a transparent texture, the edge can have alias effects that are not corrected by anti-aliasing algorithms used for geometry alias.
- **Shadow alias**

MIP-Mapping

The optimal solution to texture aliasing is to integrate over the area of the texture that is projected onto the fragment. This solution is very complex and slow. For ray-tracing supersampling, taking the average colour of multiple points is optimal.

For rasterisation, a good way is to use *MIP-Maps* and *trilinear filtering*. The texture is stored in multiple resolutions, increasing the memory by 33%. Each resolution is called a *level*. The GPU calculates the optimal level for a given fragment, the farther the texture is away, the lower is the resolution.

In the general case, the calculated level is between to existing levels and the projected position of the texture is between to texels. Therefore the colours of the two levels, and in each level between the tow texels in u and v direction, is interpolated. Resulting in trilinear interpolation.

Anisotropic Filtering

Most of the time, a pixel is not projected into a perfect square on the texture. The pixel can span more texels in the u than in the v direction. This is alleviated by *anisotropic filtering*. Multiple points in different positions on the project area are sampled using trilinear interpolation. The number and position of sampling points can be based on the angle of the sampled texture.

Full-Scene Anti-Aliasing

The problem with geometry aliasing is that in the end only one triangle is contributing to the colour of a pixel, even if multiple triangles cover the pixel partially.

The optimal solution is to compute the size of the area that is covered by a triangle for each pixel, sample the colour and then blend the colours with a weighted manner. This is slow. Seriously, don't implement this, except if you own Nvidia stocks.

A trivial solution is to render the scene with twice the resolution and scale the image down as the last step. This approach called *Full-Scene Anti-Aliasing* or FSAA uses 4 coverage samples for each pixel, and creates one fragment if at least one sample point is covered. It then samples the colour at the same positions, that are covered and therefore creates 4 colours that are blended after the visibility test.

Multisample Anti-Aliasing

A downside to FSAA is that the sampling positions are on a uniform gird. This creates artefacts for edges that run parallel to the image grid. Second, for a pixel where two triangles cover halve of it, two (similar) colours a sampled for each triangle, using more computations power.

Multisample Anti-Aliasing or MSAA uses a changeable number of coverage samples at variable positions. If the triangle covers the pixel, one fragment is created and the colour is sampled once and copied to all cover sample points, creating multiple colour values that are blended in the end of the rendering pipeline.

Coverage Sampling Anti-Aliasing

It is rare for a pixel to be covered by more than 4 triangles. *Coversampling Anti-Aliasing* or CSAA saves memory by storing up to four colours and a mapping of the colours to the cover sampling positions, instead of copying the sampled colour the corresponding sample positions. The final colours of the pixel are blended weighted by the number of positions each colour is mapped to.

Postprocessing Anti-Aliasing

For shader aliasing, a simple solution is to use *Postprocessing Anti-Aliasing* or FXAA. It detects edges after the rendering pipeline is finished and blends them. This gets rid of shader aliasing but can smooth colour edges that are meant to be sharp, for example the edge between the UI and the background in a game.

The edge detection can use information form the rendering pipeline like depth values, colour values or even the previous frame.

8 Polygonal Meshes

We can approximate objects using polygonal meshes. Such a mesh consists of a geometry, which defines the shape of an object through the vertices, and a topology, which specifies the edges and faces of an object. In the following, we will mainly deal with triangular meshes in $\mathbb{R}^2$ and $\mathbb{R}^3$ space and will refer to polygonal meshes simply as meshes. We will let V, E and F stand for the number of vertices, edges and faces respectively.

8.1 Properties

For any mesh (the *primal mesh*), we define its *dual mesh* as a mesh in which any k-dimensional entity is replaced by a $(2-k)$-dimensional one. Or, for our purposes, we replace faces by points and vice versa, and keep the vertices.

Our meshes are 2-*manifolds*, meaning that they are surfaces with no extra entities attached. Formally, they fulfil

- the *local disk property* (for any point, there is an ϵ-ball whose intersection with the surface is homeomorphic to a disc),
- the *edge ordering property* (edges can be uniquely ordered in a (counter-)clockwise fashion) and
- the *face count property* (interior edges have exactly two neighbouring faces, boundary edges exactly one).

Sometimes, our meshes are *closed*, that is, there are no boundary edges.

8.2 Euler's Formula

Euler's Formula gives the relation between the number of vertices, faces, edges and the genus of an object has and is given by:

$$V - E + F = 2(1 - g)$$

This can easily be proven by induction over the size of the mesh for planar (vertex insertion, vertex addition) and 3D meshes (face split, edge split).

Definition: Genus

The genus of an object is given by the amount of handles attached to it. That is, a sphere has genus 0, a donut has genus 1 etc. It can be derived from the Euler formula:

$$g = 1 - \frac{V - E + F}{2}$$

For triangle meshes, we can define a constant c such that $c = 2(1 - g) = V - E + F$. By then splitting the edges into halfedges, we have 3 halfedges per face, thus we write, $3F = HE \Leftrightarrow 3F = 2E$.

Inserting this into the Euler Formula, we have $2V - F = c$. And since normally, V and F are considerably larger than c, we have $F \approx 2V$. We can also derive that $HE \approx 6V$.

8.3 Platonic Solids

Platonic solids are convex polyhedra with the properties that they are closed, that they have genus 0, that all faces are regular p-gons and and that all vertices have valence q.

We can derive all existing platonic solids by using the Euler formula and the above properties. This is done by assigning half edges to vertices $HE = 2E = qV$ and to faces $HE = 2E = pF$ respectively. We can transform these equations, insert the results into the Euler Formula and obtain E, V and F in function of p and q.

Definition: Schläfli-Symbol

For a give platonic solid made of regular p-gons and vertex valence q, the Schläfli-Symbol is given by:

$$\{p, q\}$$

Then, as $p \geq 3$, $2p - pq + 2q > 0$ and $V, E, F > 0$ must hold, we can derive all possible platonic solids. As an example, the cube, which has vertex valence 3 and is made of squares, is given by the Schläfli-Symbol $\{4, 3\}$. Additionally, a solid with Schläfli-Symbol $\{m, n\}$ is dual to the solid with symbol $\{n, m\}$.

8.4 Datastructures

Triangle Lists

A possibility would be to simply store any triangle by specifying its three vertices.

Shared Vertex

We can reduce the redundancy by saving a list of vertices and their coordinates on one hand and a list of triangles made of said vertices on the other. This is the principle of the *shared vertex* technique.

Triangle Strips

This redundance can be further reduced if instead of saving the topology separately, we implicitly saved it through ordering of the vertices. This construct is called a *triangle strip*. In practice, we require more than one of these strips to describe the topology.

Triangle Fans

The idea of triangle fans is to save faces sharing a central vertex. For N faces, we save $N+2$ edges.

Edge Based Structure

The *Winged Edge* datastructure saves, for vertices and faces, pointers to an incident edge each, and for each edge pointers to its two vertices.

Further to that, it stores four edge pointers, two for each incident vertex, of which one is oriented clockwise, the other counterclockwise. Finally, it also stores one pointer for each incident face.

Definition: Winged Edge Structure

The Winged Edge structure saves, for each edge, vertex and face, the following data:

$$e = [^*v_0, ^*v_1, ^*e_{0c}, ^*e_{0cc}, ^*e_{1c}, ^*e_{1cc}, ^*f_0, ^*f_1]$$

$$v = [^*e], \quad f = [^*e]$$

Face Based Structures

This structure is impractical since in trying to jump along the faces around one vertex, we need to check which face we came from to continue in the right direction. Further, we do not actually save the edges, thus they are not represented.

Definition: Face Based Structure

In a face based structure, for each face, we save pointers to its vertices and its neighbouring faces, and a pointer to one face for each vertex:

$$f = [^*v_0, ^*v_1, ^*v_2, ^*f_0, ^*f_1, ^*f_2] \quad v = [^*f]$$

Halfedge Based Structure

Here, we replace and edge v_0v_1 by two halfedges $\overrightarrow{v_0v_1}$ and $\overrightarrow{v_1v_0}$.For each such halfedge, we store four pointers, one to its opposite halfedge, one to its succeeding halfedge, one to the vertex it points to and one to its incident face.

Definition: Halfedge Based Structure

For a halfedge h, we save

$$h = [^*h_{opp}, ^*v, ^*f, ^*h_{next}].$$

For a vertex v and a face f we store

$$v = [^*h], \quad f = [^*h]$$

Algorithm: Valence Coding

Input: A triangle mesh.
Output: An ordered list of integers.

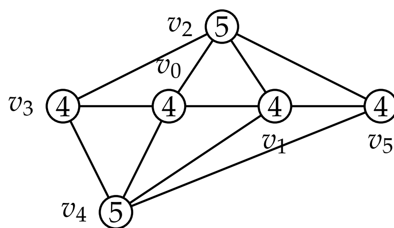
1. Select a starting triangle v_0, v_1, v_2. This decides if you store c(cw)ly.
2. Create a list starting with the valences $[p_0, p_1, p_2]$.
3. While not all vertices have been saved, starting at p_0:
4. a) In a (c)cw fashion, add the valences of all non-saved neighbours of p_i. *remember the ghost vertex*
   b) Move to p_{i+1}.

We will now give an example to make these algorithms clearer.

Given the following list

$$\{\, 4, 4, 5, 4, 5, 4, \mathbf{4} \,\}$$

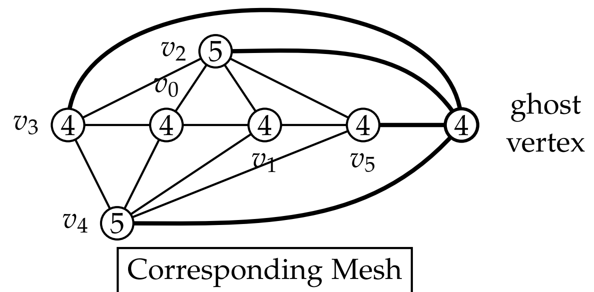
and working counterclockwisely, we obtain the mesh on the right. Conversely, it is also the list we would obtain if we wanted to store the following mesh:



Algorithm: Valence Decoding

Input: An ordered list of vertex valences $[p_0, \dots, p_n] \in \mathbb{N}$.
Output: A triangle mesh.

1. Create vertex v_0.
2. Create p_0 vertices $v_1, \dots, v_{p_0+1}$ (counter)clockwisely around v_0.
3. Connect v_0 to all of these and each to its two neighbours.
4. As long as the list isn't empty:
5. a) From v_i, move to v_{i+1}.
   b) Create $p_{i+1} - v(v_{i+1})$ further vertices (c)cw-ly.
   c) Connect them just as described above.
6. Should there be a *ghost vertex*, create a final vertex and connect it to all outer vertices.



Corresponding Mesh

This is due to the ghost vertex being used to store how many vertices are located on the edges of the mesh. Thus, for example, if we stored v_2, we would store $p_2 = 4 + 1 = 5$, as we need to remember the ghost vertex.

9 Constructive Solid Geometry

In Constructive Solid Geometry (CSG), we will have the possibility of combining different objects through boolean operations. Thus, we will be describing these objects using implicit representations. In $\mathbb{R}^3$, if a point p is contained by an object defined by $F : \mathbb{R}^3 \to \mathbb{R}$ if $F(p) \leq 0$. Conversely, if it isn't, $F(p) > 0$ will hold.

We will consider primitives that can be given as matrix-vector products of the following form, that is, by quadratic polynomials.

$$F(x,y,z) = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \cdot \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

9.1 Quadrics

For quadratic polynomials, using an upper right corner matrix suffices. Preferably, symmetric matrices as described on the right may be used, as symmetric matrices have convenient properties.

Note that the matrices on the right are not equal, but their uses yield the same results for quadrics.

In the following, we will give matrices for a few basic primitives in $\mathbb{R}^3$. These can easily be transformed into polynomials by inserting them into the formula above, for easier understanding.

Definition: Quadrics

Quadrics are primitives in $\mathbb{R}^3$ that can be defined by polynomials of degree 2.

They can be given as defined above using either of the following notations:

$$\begin{pmatrix} a & b & c & d \\ 0 & e & f & g \\ 0 & 0 & h & i \\ 0 & 0 & 0 & j \end{pmatrix} \equiv \begin{pmatrix} a & \frac{b}{2} & \frac{c}{2} & \frac{d}{2} \\ \frac{b}{2} & e & \frac{f}{2} & \frac{g}{2} \\ \frac{c}{2} & \frac{f}{2} & h & \frac{i}{2} \\ \frac{d}{2} & \frac{g}{2} & \frac{i}{2} & j \end{pmatrix}$$

Definition: Implicit Sphere

A sphere of radius r and midpoint m can be written as a quadric using the matrix

$$Q_{sphere} = \begin{pmatrix} 1 & 0 & 0 & -m_x \\ 0 & 1 & 0 & -m_y \\ 0 & 0 & 1 & -m_z \\ -m_x & -m_y & -m_z & (m_x^2 + m_y^2 + m_z^2 - r^2) \end{pmatrix}$$
$$\equiv (x - m_x)^2 + (y - m_y)^2 + (z - m_z)^2 - r^2$$

Definition: Implicit Cylinder

A cylinder of radius r, axis $(0,0,1)^T$ and going through $(m_x, m_y, 0)^T$ can be written as a quadric using the matrix

$$Q_{cylinder} = \begin{pmatrix} 1 & 0 & 0 & -m_x \\ 0 & 1 & 0 & -m_y \\ 0 & 0 & 1 & 0 \\ -m_x & -m_y & 0 & (m_x^2 + m_y^2 + -r^2) \end{pmatrix}$$
$$\equiv (x - m_x)^2 + (y - m_y)^2 - r^2$$

Definition: Implicit Cone

A cone of apex a, and opening angle α along the z-axis is given by:

$$Q_{cone} = \begin{pmatrix} 1 & 0 & 0 & -a_x \\ 0 & 1 & 0 & -a_y \\ 0 & 0 & -\tan^2\alpha & a_z\tan^2\alpha \\ -a_x & -a_y & a_z\tan^2\alpha & (a_x^2 + a_y^2 - a_z^2\tan^2\alpha) \end{pmatrix}$$
$$\equiv (x-a_x)^2 + (y-a_y)^2 - \tan^2\alpha(z-a_z)^2$$

9.2 Operations

Transformation

We have previously introduced transformations in the form of matrices. These also hold for CSG objects and can be applied very easily by applying the maths described on the right.

Definition: Transformation

Given the object defined by a quadric $v^T \cdot Q \cdot v = 0$, we transform it by $M \in \mathbb{R}^{4\times4}$ and obtain the quadric defined by $v^T \cdot Q' \cdot v = 0$ with:

$$Q' = (M^{-1})^T \cdot Q \cdot M^{-1}$$

Union, Intersection and Subtraction

Given two objects F_1 and F_2 given by their implicit functions, all the points lying inside them are given by

$$S_i = \{ p \in \mathbb{R}^3 : F_i(p) \leq 0 \}, i \in \{1, 2\}.$$

With this we can intuitively and formally define the Boolean union, intersection and subtraction operations.

The union requires the point to only be contained by one of the objects. So if the minimum of the two functions is negative (implying that at least one is), then the point lies within the union.

The intersection on the other hand requires both objects to contain a point. Thus, only if the maximum of both functions is negative (implying that both are), then the point lies in the intersection.

Definition: Union

The union of F_1 and F_2 is given by:

$$S = F(p) = \min \{ F_1(p), F_2(p) \}$$

Definition: Intersection

The intersection of F_1 and F_2 is given by:

$$S = F(p) = \max \{ F_1(p), F_2(p) \}$$

Definition: Subtraction

The subtraction of F_1 and F_2, that is, the points contained in F_1 but not F_2, is given by:

$$S = F(p) = \max \{ F_1(p), -F_2(p) \}$$

Finally, the subtraction requires that a point be contained by F_1 but not F_2. Since the points not in F_2 are contained by the object $-F_2$, we can compute this as the intersection of F_1 and $-F_2$.

10 Scene Representation

10.1 Culling

When rendering a scene, some objects might not be fully or even at all visible for the camera. We want to render only the objects that are actually visible. We call this problem *visibility determination*, and there are different approaches to solving it.

Culling is the process that allows us to avoid rendering certain faces that are not visible for the camera. We will be summarising a few of them, ordered by increasing order of computational cost.

Backface Culling

The simplest of our techniques is backface culling. It is pretty intuitive as it simply allows removing the polygons not facing the camera, and thus assuming that all meshes are closed. Further, there are situations in which som ploygons *do* face the camera but still aren't visible.

Hierarchical Frustum Culling

Frustum culling includes backface culling, but also tests objects if they lie inside the viewing frustum. If they are inside or intersect with it, they are rendered, else they aren't.

This testing of objects can be excessively expensive for complex objects, thus it is often faster, though less accurate, to construct bounding boxes around objects and testing the boxes.

We now introduce three types of boxes, sorted by decreasing included empty space and increasing intersection test costs.

The *bounding sphere* simply constructs a sphere around the object. The *axis-aligned bounding box* constructs a box around the object while keeping it aligned to the axes (duh!). Similarly obvious is the *oriented bounding box* which creates the box of smallest volume around an object.

The hierarchical part comes in by creating bounding boxes around bounding boxes. If such a bounding box is fully in- or outside, so are all the children. If it's an intersection, we test the children individually.

Portal Culling

Portal culling applies frustum culling, then works with high level knowledge of a scene and divides architectural scenes in rooms connected by portals. Then we use the frustum to detect the visible portals and recursively generate further, smaller frustums and cull further. This can become difficult if reflective surfaces are present.

Occlusion Culling

Occlusion culling renders objects front to back, while keeping track of rendered objects as occluders. If, for an object, less than a given amount of fragments n (Level Of Detail, LOD) are visible because the rest are covered by occluders, we do not render said object (because if we did, the tests would take forever and be pretty much useless).

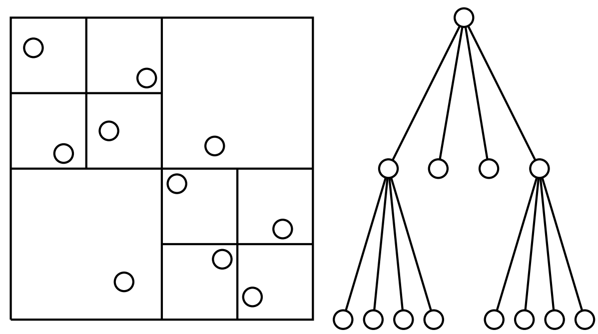
10.2 Optimisation structures

Quad/Octrees

Remember *Binary Trees*, also called Bintrees? Well, *Quadtrees* and *Octrees* are pretty similar in principle, except that you use them to split 2 and 3 dimensional space respectively.

Take a square. You can divide it into four smaller, equally sized squares. Similarly, a cube can be divided into 8 equally sized cubes.

On the right you see a quadtree, but it should by now be pretty obvious what an octree would look like.



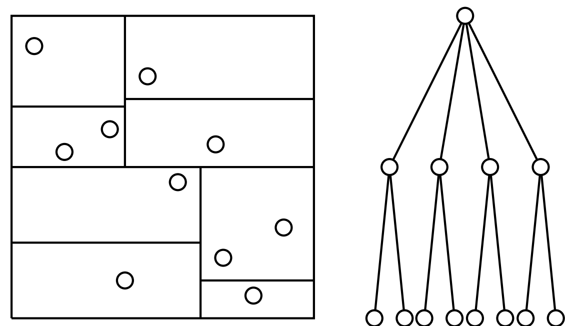
Quadtrees exemplified

KD Trees

Now, we saw that quadtrees aren't very apt at splitting scenes into balanced subdivisions. That's why we are going to introduce *KD trees*.

Contrary to quadtrees these are not independent of the scene, but they are capable of dividing it into more or less equal parts by splitting horizontally and vertically not at the centres, but rather where it actually makes sense.

However, we still split horizontally, then vertically, then horizontally and so on.



KD Trees exemplified

Binary Space Partitioning

Binary Space Partitioning (BSP) is just a further generalisation of KD Trees where we don't even split horizontally or vertically, but only how it best suits us to further balance our tree.

We will not illustrate this concept at this point because we're too lazy to create the corresponding diagram. However, as a reader, picture BSP as a diagram of a KD tree in which all lines, instead of being all nice and perpendicular to each other, are skewed and diagonal and do not satisfy your inner OCD at all.

11 Volumetric Rendering

11.1 Signed Distance Function

A *Signed Distance Function* is a special case of an implicit function in that it also returns the *sign* between a point and the object. It of course keeps the *sign*, positive if outside, negative if inside. The name should now seem well chosen.

Definition: Signed Distance Function

$$F(x) = \min_{p \in S} ||p - x|| \cdot \begin{cases} +1 & \text{if } x \text{ outside} \\ -1 & \text{else} \end{cases}$$

Another concept are *Value Functions* which attribute, to any point in space, a value corresponding to a property, e.g. temperature, density etc. From such functions, we can generate sets of points with the same property, creating surfaces which we call *iso-surfaces*.

An example for these are CT or MRT Scans, which use tissue density as a function to specify differing surfaces. This means that in practice, we won't have continuous functions, but discrete samples which, we will assume w.l.o.g., are given for a regular 3D grid with distances of 1.

The technique we will be using is similar to bilinear interpolation and called *trilinear interpolation*. Given how our grid is defined, we can consider it as a bunch of cubes, each specified by eight measurement points.

$$\begin{aligned} F(x,y,z) = \;& F_{0,0,0} \cdot (1-x)(1-y)(1-z) \\ &+ F_{1,0,0} \cdot x(1-y)(1-z) \\ &+ F_{0,1,0} \cdot (1-x)y(1-z) \\ &+ F_{1,1,0} \cdot xy(1-z) \\ &+ F_{0,0,1} \cdot (1-x)(1-y)z \\ &+ F_{1,0,1} \cdot x(1-y)z \\ &+ F_{0,1,1} \cdot (1-x)yz \\ &+ F_{1,1,1} \cdot xyz \end{aligned}$$

Thus, for a point located within such a cube, we can approximate its value function by interpolating it using these eight points $F_{a,b,c}, \;\; a,b,c \in \{0,1\}$.

11.2 Direct Volume Rendering

Given a volumetric scene, we can use *ray casting* to determine the colour of a pixel by casting a ray for each pixel and adding the colours and opacities of the different layers.

For this, there are two approaches: Front-to-back ray casting and back-to-front ray casting. In the following c_i represents the colour of the ith layer, α_i its opacity and $\hat{c}_i = \alpha_i c_i$ its opacity-weighted colour with $i \in \{0, ..., k\}$. We assume that c_i and α_i are values in $[0,1]$.

Definition: Front-to-Back

With $\hat{C}_0 = \hat{c}_0$ and $A_0 = \alpha_0$, we apply

$$\begin{aligned} \hat{C}_i &= \hat{C}_{i-1} + (1 - A_{i-1})\hat{c}_i \\ A_i &= A_{i-1} + (1 - A_{i-1})\alpha_i \end{aligned}$$

recursively until $i = k$ or until $A_i = 1$, that is, when we have reached total opacity.

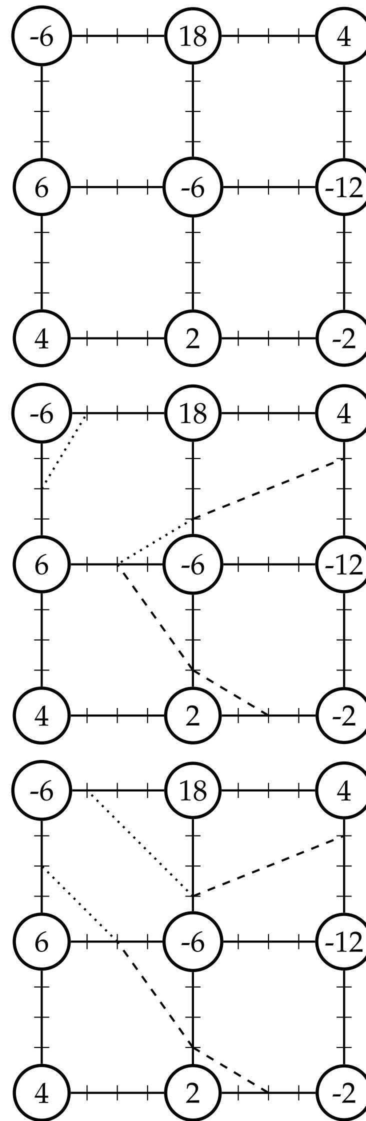
Definition: Back-to-Front

Let $\hat{C}_k = \hat{c}_k$. Then, with

$$\hat{C}_i = \hat{c}_i + (1 - \hat{\alpha})\hat{C}_{i+1}$$

the final opacity-weighted colour is $\hat{C}_0$.

11.3 Indirect Volume Rendering



For indirect volume rendering, we use the *Marching Cubes* algorithm. To make it simple, we discretely sample our scene with a three-dimensional grid, measuring a certain metric. This allows us to create a signed distance function, from which we can compute a mesh.

We will illustrate it using a simplified algorithm, that is, the marching *squares* algorithm.

As the function is a signed distance function, points with negative value lie within the polygon, while points with positive values lie outside. And the distance part allows us to compare the ratios between the points to approximate more precisely how the polygon is shaped.

However, the dotted lines in our example are problematic, as both the displayed polygon shapes are valid interpretations. There are two common solutions to this issue. Either, we assume the sign of the centre of any square to be positive or negative, which we can always do, or we do a further measurement whenever there is more than one possibility, which depends on the use case.

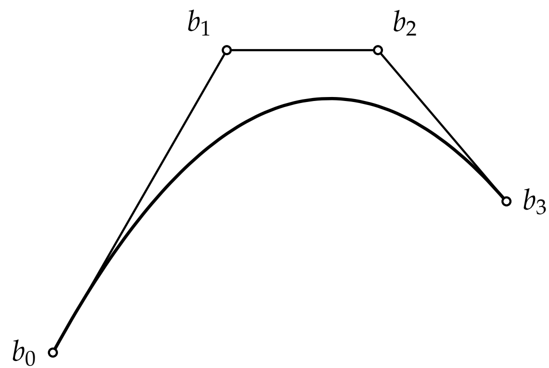
Note that this method can be imprecise and prone to aliasing, as it may cut off, for example, entire triangles contained in a single square. There are many sampling methods to make this more precise.

12 Curves

In this chapter, unlike in the lecture, we will not go into the formal aspects of the subject as this would exceed the scope of what we are trying to achieve. However, we highly recommend you check it out in the lecture, as it is important for a good understanding of the substance.

12.1 Bézier Curves

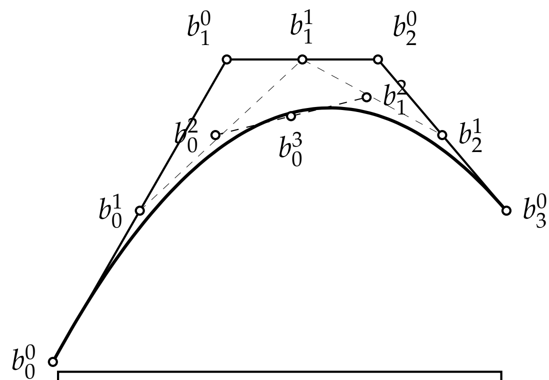
A *Bézier curve* given by $n+1$ control points is the curve given by a polynomial of degree n. As such, the curve on the right, which is specified by 4 control points, is of degree 3.



Cubic Bézier Curve

12.2 De Casteljau Algorithm

The *De Casteljau algorithm* allows us to discretise a given Bézier curve with an arbitrarily high precision by evaluating the polynomial for any amount of parameters $t \in [0,1]$. Its application is, somewhat poorly, visualised on the right.



De Casteljau visualised for $t = \frac{1}{2}$

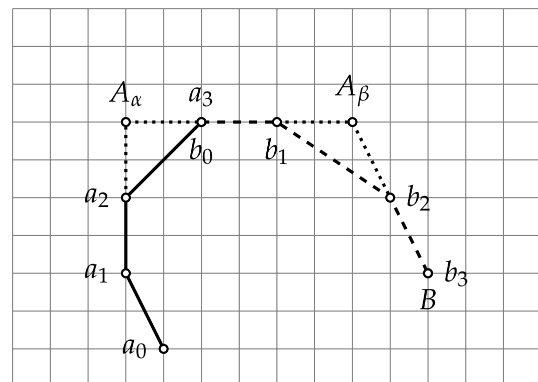
12.3 Bézier Spline

A *Bézier Spline*, or composite Bézier curve is a curve made up of multiple Bézier curves which are, in applications, usually of degree 3. This guarantees a high stability, as opposed to a single, high degree curve.

Given a set of points 4 control points and a further set of points we want to interpolate, we can use the *A-Frame construction* to create further control points that create a Bézier spline through these points.

On the right, we are given $a_i,\ i \in \{0,1,2,3\}$ and B. Then, the dotted lines and the As are our A-frame elements while the dashed segments along with $b_i,\ i \in \{0,1,2,3\}$, with $a_3 = b_0$ and $b_3 = B$ the next interpolated Bézier curve.

The A-Frame construction tells us that $[a_1, a_2] = [a_2, A_\alpha]$, that $[A_\alpha, a_3] = [b_0, b_1] = [b_1, A_\beta]$ and that $[A_\beta, b_2] = [b_2, b_3]$.



A-Frames visualised

As an addition to the lecture, we recommend you create a few examples of these concepts and complete them graphically on your own as to get an intuitive feel for them.