Documentation

Krisztián Szabó

September 6, 2024

In this PDF, I aim to describe concepts related to computer graphics, with a focus on the mathematical aspects. While this information may be useful to others, please note that it is primarily for my own purposes, and I do not assume responsibility for any errors.

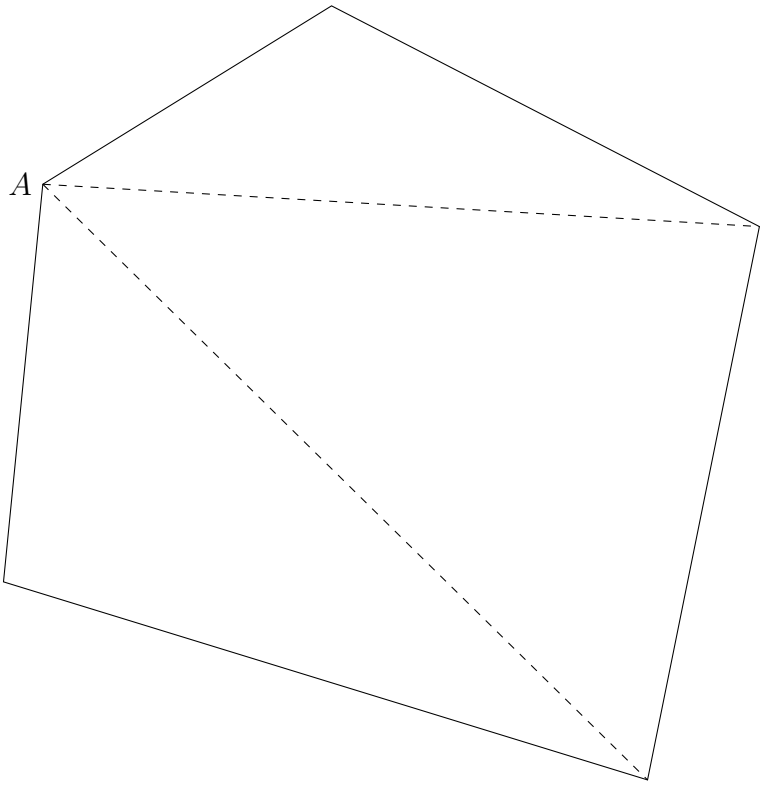
Contents

1 Convex Mesh Division

In this section, a simple yet effective method to divide any convex polyhedron into arbitrarily small tetrahedrons is explained. This approach is useful for creating destructible objects. It is assumed that a convex polyhedron is provided, where only the vertices of each face are known, and each face can be any polygon.

1.1 Polyhedron to Tetrahedrons

First, we aim to divide our polyhedron into smaller tetrahedrons and subsequently divide those tetrahedrons further. As you might have guessed, a tetrahedron is also a polyhedron. We will use a slightly different approach for dividing tetrahedrons. For simplicity, convert all faces into triangles, which is straightforward:

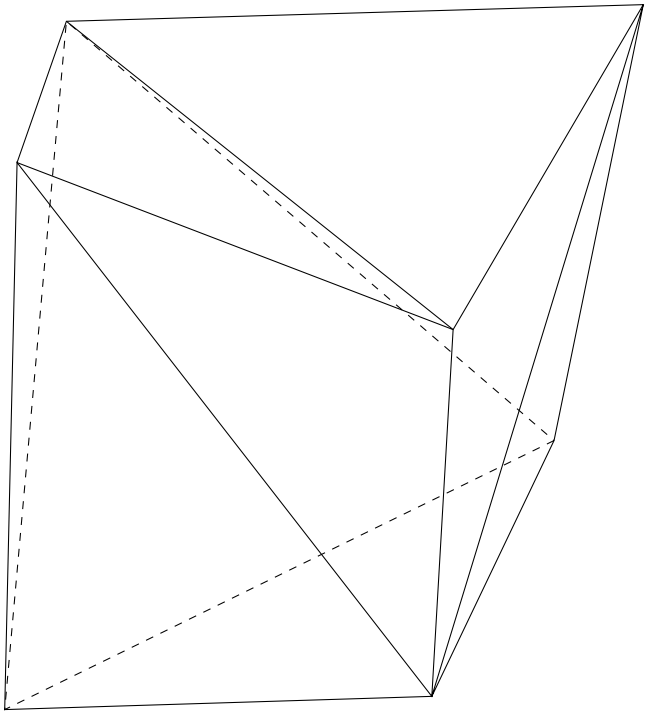


All you need to do is select any vertex A and connect it to every vertex in the polygon except its neighbors.

```
struct Triangle {
    glm::vec3 v1;
    glm::vec3 v2;
    glm::vec3 v3;
};

std::vector<Triangle> GetTrianglesFromPolygon(const std::vector<glm::vec3
    >& polygon) {
```

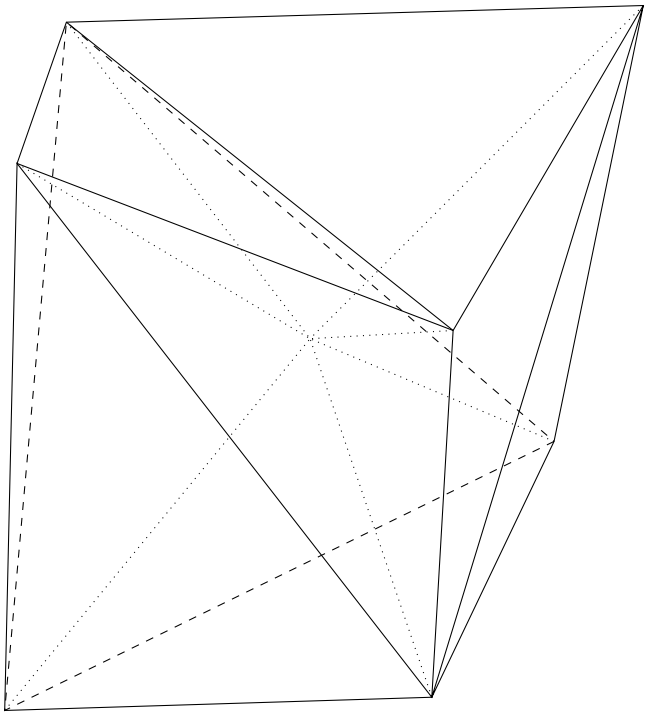
```
        std::vector<Triangle> result;

        for (int i = 2; i < polygon.size(); ++i) {
            result.push_back(Triangle{polygon[0], polygon[i-1], polygon[i]});
        }

        return result;
    }
```

Now we have any array of triangles that build up our convex polyhedron.



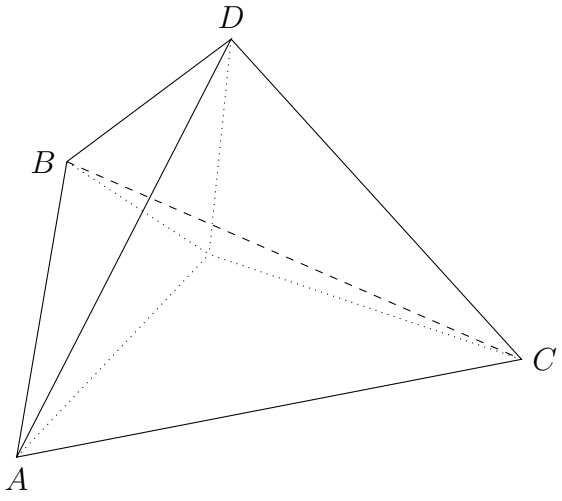
The simplest method for dividing a convex polyhedron into smaller tetrahedrons is to find the center of the polyhedron and then connect this center to each of the polyhedron's triangular faces.

```
    std::vector<Tetra> GetTetras(const std::vector<Triangle>& triangles) {
        glm::vec3 center{0.0f, 0.0f, 0.0f};
        for (const auto& t : triangles) {
            center += t.v1; center += t.v2; center += t.v3;
        }
        center /= (triangles.size() * 3.0f);
        std::vector<Tetra> tetras;
        for (const auto& t : triangles)
        {
            tetras.push_back({t.v1, t.v2, t.v3, center});
        }
        return tetras;
    }
```

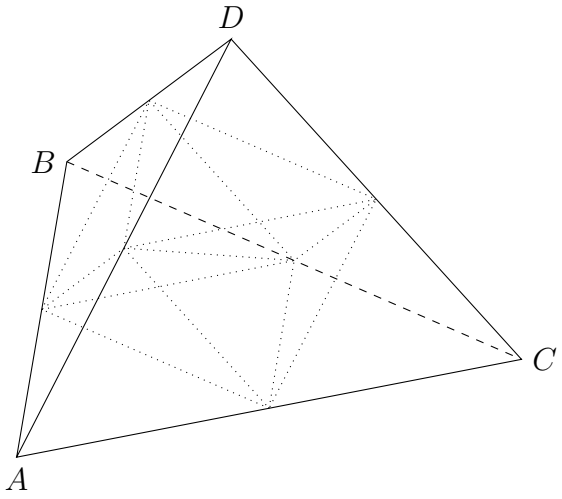


1.2 Tetrahedrons to tetrahedrons

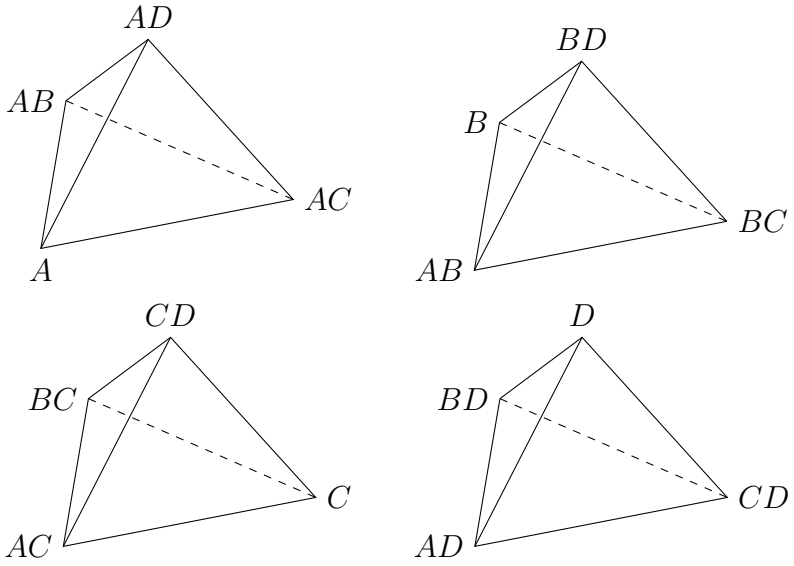
We have successfully divided our mesh into tetrahedrons. While further subdivision might be desirable, this method may not be the most effective for dividing the tetrahedrons themselves. Let's explore why this particular tetrahedron could be an example of our polyhedron:



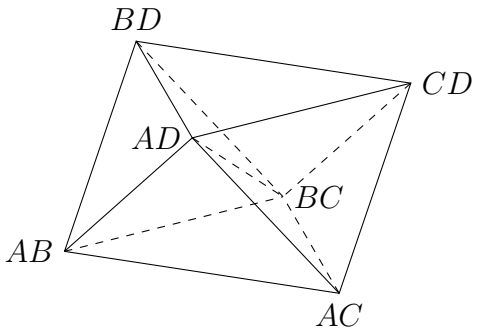
As illustrated, continuing this pattern would result in all the sub-tetrahedrons having identical faces. This would produce very thin tetrahedrons with wide faces, which might not be ideal for our purposes. Therefore, let's explore a different algorithm. Our overall goal is to calculate the midpoints of all the edges in each tetrahedron and use these midpoints to generate smaller tetrahedrons.



The illustration may be tricky, but there are actually eight smaller tetrahedrons within the larger one. Identifying the vertices is straightforward:



Within the tetrahedron, there is a hexahedron comprised of four tetrahedrons.



The algorithm is quite straightforward. You start with six vertices and need to select two that do not share a common vertex. Following this labeling approach, if you examine the labels of the chosen midpoints, they will encompass all the vertices. In this case, the appropriate pairs of vertices might be:

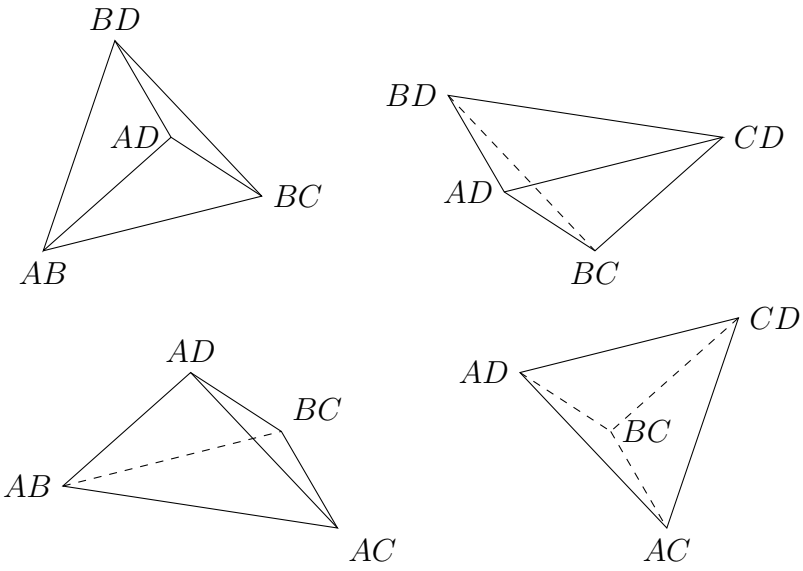
$$AD - BC,\ AB - CD,\ BD - AC$$

All four tetrahedrons will contain these two vertices. Next, we need to determine the order in which to assemble the remaining four vertices. For instance, if we choose AD and BC as the two vertices, the tetrahedrons should be assembled as follows:

1. AD, BC, $\cdot$, $\cdot$
2. AD, BC, $\cdot$, $\cdot$
3. AD, BC, $\cdot$, $\cdot$
4. AD, BC, $\cdot$, $\cdot$

where the $\cdot$ symbols represent the remaining vertices. The algorithm is also quite simple: you need to choose two vertices that share exactly one vertex.

1. AD, BC, $B\mathbf{D}$, $C\mathbf{D}$
2. AD, BC, $\mathbf{B}D$, $A\mathbf{B}$
3. AD, BC, $\mathbf{A}B$, $\mathbf{A}C$
4. AD, BC, $A\mathbf{C}$, $\mathbf{C}D$



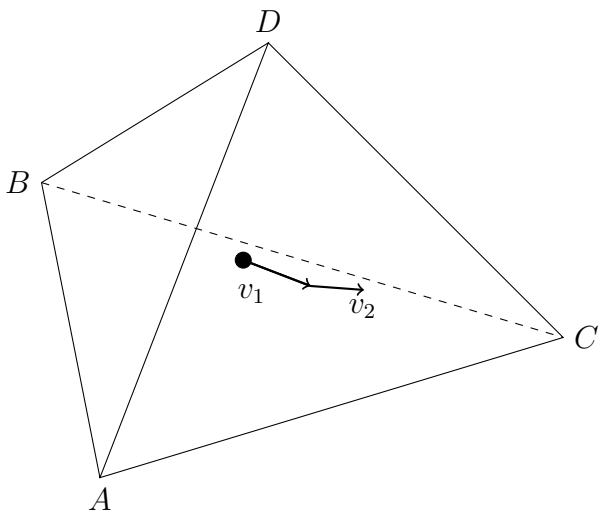
The good news is that the midpoints calculated initially do not need to be exact; they can be any points along the edges. This flexibility can lead to a noisy division.

1.3 Triangle Vertex Ordering

In computer graphics, the ordering of triangle vertices is crucial. When viewing a mesh from any position in the world, the triangles you see must be consistently ordered, either clockwise or counterclockwise. In this subsection, I provide a simple algorithm to order the triangles in your mesh. Suppose we have a list of triangles that make up our mesh. Using the glm library, the following code illustrates this:

```
bool IsClockwise(const Triangle& triangle, const glm::vec3& polyCenter)
{
    glm::vec3 triangleCenter = (triangle.v1 + triangle.v2 + triangle.v3) / 3.0f;
    glm::vec3 outVector = triangleCenter - polyCenter;

    glm::vec3 v1 = triangle.v2 - triangle.v1;
    glm::vec3 v2 = triangle.v3 - triangle.v1;
    glm::vec3 normal = glm::cross(v1, v2);

    return glm::dot(normal, outVector) < 0;
}
```

Consider a convex polyhedron. We can easily calculate its center point by adding up all the vertices and then dividing by their count. For simplicity, I have chosen a tetrahedron:



First of all, we know that given three triangle vertices, they are either ordered clockwise or counterclockwise. To ensure the desired ordering, you can examine the current order and, if necessary, swap two vertices in the triangle. The key to determining the order is the cross product of the triangle's edge vectors. The cross product of two vectors always results in a vector that is perpendicular to the plane defined by the vectors, following the right-hand rule.

We can calculate the normal vector of a triangle by computing the cross product of two vectors along its edges. However, the order of the vectors matters, as it can change the

sign of the resulting normal vector. We need to determine whether the normal is pointing outwards or inwards relative to our polyhedron.
As shown in the image above, the vector is pointing outwards. There are different methods to check if the normal is pointing outwards, but I chose to check the dot product with the vector from the polyhedron center to the triangle center. By examining the sign of the dot product, we can determine the direction of the normal vector. If the dot product is negative, it means that the angle between the two vectors is greater than 90 degrees, indicating that the normal vector is pointing outwards.

2 Cube builder

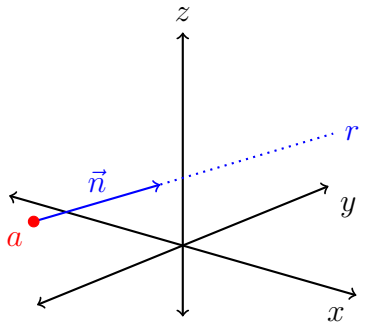
In this section, the methods and algorithms used in the development of the Cube Builder project are outlined. This includes a simplified 3D Digital Differential Analyzer (DDA) algorithm, ray-cube intersection techniques, and other relevant concepts.

Before diving into the implementation details, the primary goal of this project is first discussed. From the player's perspective, the concept is straightforward: a space is divided into cubes where blocks can be placed and destroyed. The implementation works as follows: when the player wishes to build or break a block, the first step is to identify the *target* block, which is the cube the player is looking at. To achieve this, a DDA algorithm is initially used to determine all the cubes intersected by the ray defined by the player's position and facing direction. Next, the closest block to the player that is already placed is identified. If the player wishes to break a block, the target block is destroyed. If the player wishes to place a block, the side of the target block currently being observed is determined, and a new block is placed adjacent to it.

2.1 Intersection of cubes

First, a straightforward method for examining ray intersections with cubes is explored. In this project, rays of finite length (more accurately referred to as segments) are used. A ray can be defined by its starting point $a \in \mathbb{R}^3$ and direction vector $\vec{n} \in \mathbb{R}^3$.

$$r(t) := \vec{n} \cdot t + a \quad (t \in \mathbb{R}_0^+)$$



Segments can be defined with two points a, $b \in \mathbb{R}^3$ as follows:

$$s(t) := (b - a) \cdot t + a \quad (t \in [0,\, 1])$$

where a is the starting point and b is the endpoint. Our goal now is to determine whether our segment intersects a cube or not. Let us define a cube in space using two points c_1, $c_2 \in \mathbb{R}^3$. We impose the following restriction when defining a cube with these two points: the point $c_1 \in \mathbb{R}^3$ must satisfy the condition that

$$c_1 + \begin{pmatrix} t \\ t \\ t \end{pmatrix} = c_2 \quad (t > 0).$$

This will easen our task, with avoiding ambiguity. With this restriction we can say the following: our segment $s(t)$ ($t \in [0,\, 1]$) intersects our cube (that is represented as above) if and only if there exits a parameter $t \in [0,\, 1]$ such that

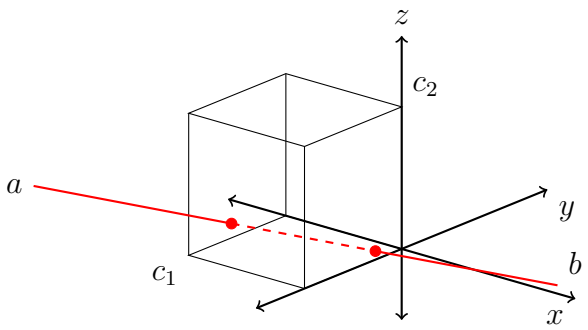
$$(c_1)_x \leq s(t)_x \leq (c_2)_x$$

$$(c_1)_y \leq s(t)_y \leq (c_2)_y$$

$$(c_1)_z \leq s(t)_z \leq (c_2)_z$$

Let us see an example:

$$c_1 := \begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix},\ c_2 := \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},\ a := \begin{pmatrix} -1.5 \\ -2 \\ 0.65 \end{pmatrix},\ b := \begin{pmatrix} 0.8 \\ 0.65 \\ -0.25 \end{pmatrix}$$



$$s(t) = (b-a)\cdot t + a = \begin{pmatrix} 2.3 \\ 2.65 \\ -0.9 \end{pmatrix} \cdot t + \begin{pmatrix} -1.5 \\ -2 \\ 0.65 \end{pmatrix} \quad (t \in [0,\, 1])$$

If we have two points $u,\, v \in \mathbb{R}^3$ let $\overset{\star}{\leq}$ denote the comparison which is considered true if

$$x_x \leq y_x \text{ and } x_y \leq y_y \text{ and } x_z \leq y_z.$$

Now the segment we have defined earlier intersects the cube if and only if there exists $t \in [0,\, 1]$ such that

$$c_1 \overset{\star}{\leq} s(t) \overset{\star}{\leq} c_2$$

$$\Longleftrightarrow$$

$$\begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix} \overset{\star}{\leq} \begin{pmatrix} 2.3 \\ 2.65 \\ -0.9 \end{pmatrix} \cdot t + \begin{pmatrix} -1.5 \\ -2 \\ 0.65 \end{pmatrix} \overset{\star}{\leq} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$\Longleftrightarrow$$

$$\begin{pmatrix} 0.5 \\ 1 \\ -0.65 \end{pmatrix} \overset{\star}{\leq} \begin{pmatrix} 2.3 \\ 2.65 \\ -0.9 \end{pmatrix} \cdot t \overset{\star}{\leq} \begin{pmatrix} 1.5 \\ 2 \\ 0.35 \end{pmatrix}.$$

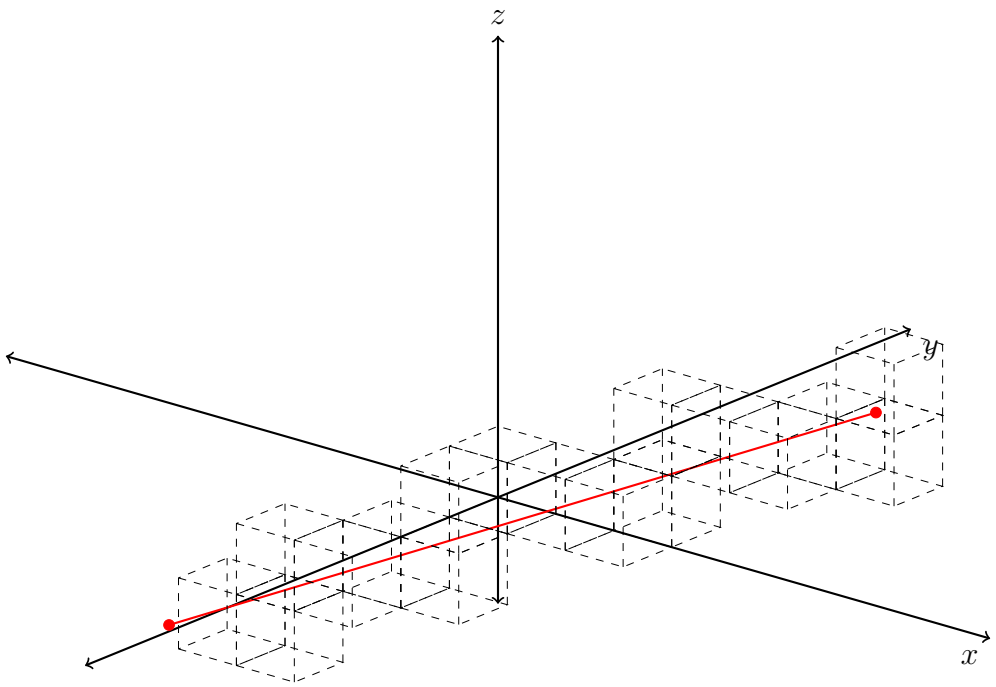
These inequalities yield the following result: two intersection points, $s(t_1) = (-0.63, -1, 0.31)$ and $s(t_2) = (0, -0.27, 0.06)$, with parameters $t_1 = 0.377358$ and $t_2 = 0.652174$. It is important to note that in this project, the objective was only to determine whether any intersection points exist. For this purpose, the following function can be used.

```
bool DoesRayIntersectCube(const glm::vec3& rayStart, const glm::vec3&
    rayDirection, const glm::vec3& cubeMin, const glm::vec3& cubeMax)
{
    glm::vec3 tMin = (cubeMin - rayStart) / rayDirection;
    glm::vec3 tMax = (cubeMax - rayStart) / rayDirection;

    glm::vec3 t1 = glm::min(tMin, tMax);
    glm::vec3 t2 = glm::max(tMin, tMax);

    float tEnter = std::max(std::max(t1.x, t1.y), t1.z);
    float tExit = std::min(std::min(t2.x, t2.y), t2.z);

    return tEnter <= tExit && tExit >= 0;
}
```

2.2 DDA algorithm

The Digital Differential Analyzer (DDA) algorithm is a simple and efficient method used for rasterizing lines, curves, and surfaces in computer graphics. It works by incrementally plotting points along the line between two given endpoints. The DDA algorithm calculates the intermediate values along the line by iteratively adding a fixed small step to the start point, ensuring that the plotted line closely approximates the true line.

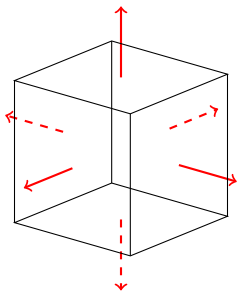
In 3D graphics, a variant of the DDA algorithm is often used for tasks such as ray tracing, where it helps in stepping through 3D space to find intersections with objects like cubes. The algorithm operates by advancing the ray in small, discrete steps, checking for intersections with the grid or objects at each step. The DDA algorithm is particularly valued for its simplicity and ease of implementation, though it may not be as accurate or efficient as other more complex algorithms in certain scenarios.



The implementation of the algorithm that was ultimately used can be found in the following **GitHub repository**.

2.3 Intersection of faces

Given the information from the previous two sections, breaking the target block in the game can be accomplished quite easily. However, when the player wants to build a block, additional calculations are required to determine the adjacent block to the target. It is necessary to identify the correct side where the player intends to place a new block. Note that if the target block is known, its side-neighbors can be associated with the cube's side normal vectors:



A straightforward technique can be employed to eliminate some sides before determining intersections. This involves calculating the dot product of the ray's direction vector with each side normal of the cube. Only the sides where the dot product is negative are of interest, as this indicates that the normal vector is oriented towards the observer. When the dot product is zero, the ray's direction vector is perpendicular to the normal vectors, meaning only the edge of the side is visible. Additionally, it is possible to check if there is a block in the neighboring position; if so, the corresponding neighbor can be disregarded.