GPU-accelerated real-time rendering of n-dimensional objects

Pre-scientific thesis written by

Oliver Kovacs

Class 8B

BRG STEYR MICHAELERPLATZ

Supervisor: Mag.^{a} Dr.^{in} Sara Hinterplattner B.Sc.

BRG Steyr Michaelerplatz
Michaelerplatz 6
4400 Steyr

Steyr, February 2023

Acknowledgements

I would like to thank Univ.-Prof. Dr. Ing. habil. Oliver Bimber (Head of the Institute of Computer Graphics at Johannes Kepler University Linz) for providing valuable insights and giving helpful feedback on this thesis.

INSTITUTE OF COMPUTER GRAPHICS

JKU JOHANNES KEPLER UNIVERSITY LINZ

Abstract

N-dimensional rendering is a fascinating but computationally expensive process. This makes it ill-suited for real-time applications. GPUs have been conventionally used to accelerate rendering workloads, however they are heavily optimized for 3D rendering. This thesis explains the mathematics underpinning n-dimensional rendering and takes a look at how it can be optimally realized on modern graphics hardware. It will give some workarounds to overcome the inherent limitations of components designed not with this use case in mind.

A reference implementation written in C++ and utilizing Vulkan is provided in the form of vulkan-xd, exemplifying some of the techniques discussed. Some basic performance analysis was conducted on the project giving a benchmark for the achievable efficiency.

Further research is required to come up with additional optimizations and to accurately assess the exact performance to be expected.

Contents

1. Overview

Rendering is the process of creating a 2D image of some 3D data using a computer. Real-time implies that a frame has to be rendered within a given short time period.[1] Use cases of real-time rendering include but are not limited to video games, modeling software, CAD, and data visualization.

1.1. Approaches

There exist multiple rendering algorithms with different benefits and drawbacks. Some examples include wire-frame and polygon based rendering, ray tracing and ray marching. This thesis will focus on wire-frames due to their high performance and because they generalize well to arbitrary dimensions.[2]

1.2. Contents of the thesis

Chapter 2 gives an overview of 3D rendering, chapter 3 covers the mathematics of rendering n-dimensional objects, while chapter 4 discusses considerations concerning computer hardware. Chapter 5 explains the reference implementation.

[1]Marschner and Shirley, 2016, p. 438.
[2]Marschner and Shirley, 2016, pp. 139–140.

1.3. Notation

- Vectors are written as lowercase letters in bold typeface, for example $\boldsymbol{v}$.
- A subscript after a vector like $\boldsymbol{v}_i$ represents the ith component of the vector $\boldsymbol{v}$. Thus a vector $\boldsymbol{v} \in \mathbb{R}^n$ is
$$\boldsymbol{v} = \begin{bmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_n \end{bmatrix} .$$
For $n \leq 4$ sometimes x, y, z and w are also used, for example
$$\boldsymbol{v} = \begin{bmatrix} \boldsymbol{v}_x \\ \boldsymbol{v}_y \\ \boldsymbol{v}_z \end{bmatrix} .$$
- Matrices are written as uppercase letters in bold typeface, for example $\boldsymbol{M}$.
- $(a_i)_{i \in A}$ where $A = [u, v] = \{x \in \mathbb{Z} \mid u \leq x \leq v\}$ denotes a sequence. It can be thought of as some function $f : A \to B$ that maps i to a_i
- $\boldsymbol{a} \cdot \boldsymbol{b}$ denotes the dot product of the two vectors $\boldsymbol{a} \in \mathbb{R}^n$ and $\boldsymbol{b} \in \mathbb{R}^n$. It is defined as
$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=1}^{n} \boldsymbol{a}_i \boldsymbol{b}_i = \boldsymbol{a}_1 \boldsymbol{b}_1 + \boldsymbol{a}_2 \boldsymbol{b}_2 + \ldots + \boldsymbol{a}_n \boldsymbol{b}_n .$$
- The floor function $\lfloor x \rfloor$ maps $x \in \mathbb{R}$ to the greatest integer less than x. Similarly the ceil function $\lceil x \rceil$ maps $x \in \mathbb{R}$ to the smallest integer greater than x.
- The modulo operator $a \bmod b$ where $a, b \in \mathbb{R}$ returns the signed remainder of the division $\frac{a}{b}$.
- $\begin{pmatrix} n \\ r \end{pmatrix}$ denotes the binomial coefficient of n and r. It is defined as
$$\begin{pmatrix} n \\ r \end{pmatrix} = \frac{n!}{k!(n-k)!} .$$
- $n!$ denotes the factorial of n. It is defined as
$$n! = \prod_{i=1}^{n} i = n \times (n-1) \times \cdots \times 1 .$$

2. 3D Rendering

This chapter will give a brief explanation of the mathematics behind 3-dimensional wire-frame rendering.

2.1. Meshing

Given a 2-dimensional manifold embedded in $\mathbb{R}^3$ it is possible to approximate it by a polygon mesh. The process of creating this polygon mesh is called *manifold meshing*. There are multiple approaches to it but they will not be discussed in this thesis.[1]
In this case the polygon mesh is defined to be a collection of vertices, edges and faces. For the sake of simplicity, this thesis will only use 3-polytopes as examples, which can trivially be converted to a polygon mesh.[2]

2.1.1. Vertices

The *vertices* of the polygon are the points where two edges meet. They can be represented as a euclidean vector $\boldsymbol{v} \in \mathbb{R}^3$.[3]

[1] De Laat, 2011, p. 1.
[2] Kelly, 2016, p. 5.
[3] Kelly, 2016, p. 5.

2.1.2. Edges

The *edges* of the polygon mesh are the line segment where the faces meet. Each edge can be defined in terms of two vertices.[4]

2.1.3. Faces

The *faces* of the polygon mesh are the polygons bounding it. Each face can be defined as a set of edges.[5]
In computer graphics polygon meshes usually only use triangles as faces because they are easier to work with. This is not a problem as all polygons can be broken up into triangles.[6] However faces are not relevant when only rendering the wire-frame, so this limitation does not have to be imposed upon the mesh used. Faces will be ignored from this point on.

2.2. Representation

As established, meshes can be thought of as a collection of vertices and edges for the puroposes of this thesis.

Thus a mesh could be defined as something like a list of m vertices $(v_i)_{i\in\{0,\ldots,m-1\}}$ and a list of k edges $(w_i)_{i\in\{0,\ldots,k-1\}}$ where each $v_i \in \mathbb{R}^3$ and each $w_i \in \{(a,b) \mid a,b \in \{v_0,\ldots,v_{m-1}\}\}$. Notice how v and w are indexed from 0. This is a matter of taste, however, indexing from 0 will simplify implementing this later on.

[4] Kelly, 2016, p. 5.
[5] Kelly, 2016, p. 6.
[6] Marschner and Shirley, 2016, p. 438.

2.2.1. Example

A unit 3-cube with $m = 8$ vertices $(v_i)_{i\in\{0,\dots,7\}}$ could be defined like

$$\begin{aligned}
v_0 &= (0\ 0\ 0)\\
v_1 &= (1\ 0\ 0)\\
v_2 &= (0\ 1\ 0)\\
v_3 &= (1\ 1\ 0)\\
v_4 &= (0\ 0\ 1)\\
v_5 &= (1\ 0\ 1)\\
v_6 &= (0\ 1\ 1)\\
v_7 &= (1\ 1\ 1)\ .
\end{aligned}$$

This could also be written as

$$v_i = \begin{bmatrix} i \bmod 2 \\ \left\lfloor \frac{i}{2} \right\rfloor \bmod 2 \\ \left\lfloor \frac{i}{4} \right\rfloor \bmod 2 \end{bmatrix}.$$

The $k = 12$ edges of the cube could be defined as

$$\begin{aligned}
w_0 &= (v_0, v_1) \\
w_1 &= (v_2, v_3) \\
w_2 &= (v_4, v_5) \\
w_3 &= (v_6, v_7) \\
w_4 &= (v_0, v_2) \\
w_5 &= (v_1, v_3) \\
w_6 &= (v_4, v_6) \\
w_7 &= (v_5, v_7) \\
w_8 &= (v_0, v_4) \\
w_9 &= (v_1, v_5) \\
w_{10} &= (v_2, v_6) \\
w_{11} &= (v_3, v_7)\ .
\end{aligned}$$

2.3. Transformations

A transformation is a function $T\colon X \to X$ that maps a set to itself. There are some *geometric transformations* of the form $T\colon \mathbb{R}^3 \to \mathbb{R}^3$ that are especially useful for manipulating the vertices of a mesh. These are usually *linear transformations*.[7]

A transformation T is linear if and only if[8]

1. $T(\boldsymbol{u} + \boldsymbol{v}) = T(\boldsymbol{u}) + T(\boldsymbol{v})$ and
2. $T(c\boldsymbol{u}) = cT(\boldsymbol{u})$

for all vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ and all scalars c.

[7] Marschner and Shirley, 2016, p. 109.
[8] Strang, 2011.

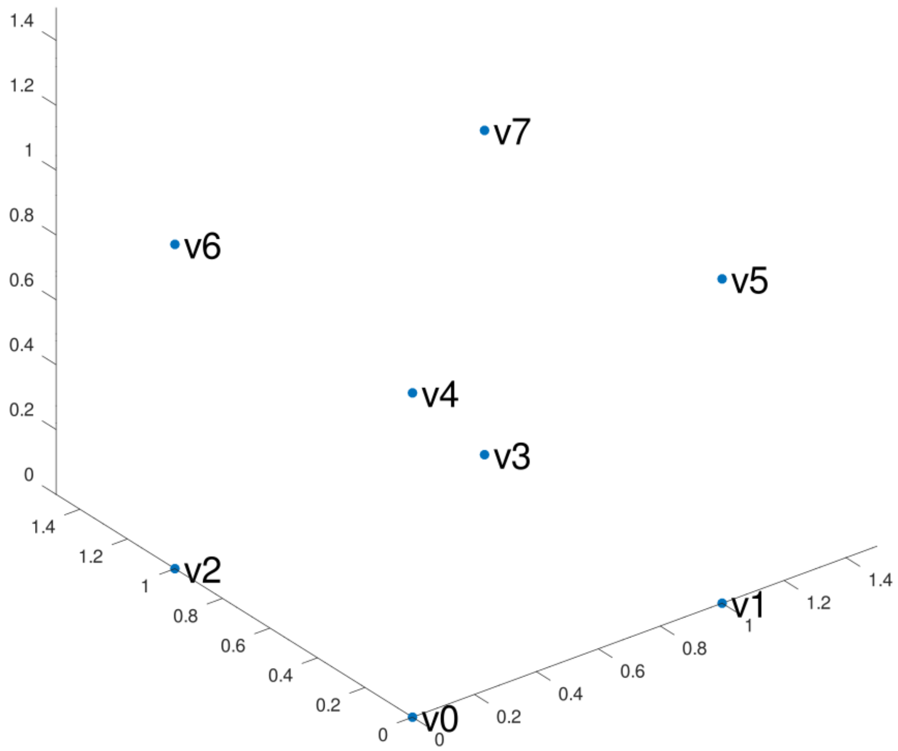

Figure 2.1.: Visualization of the vertices of a 3-cube.

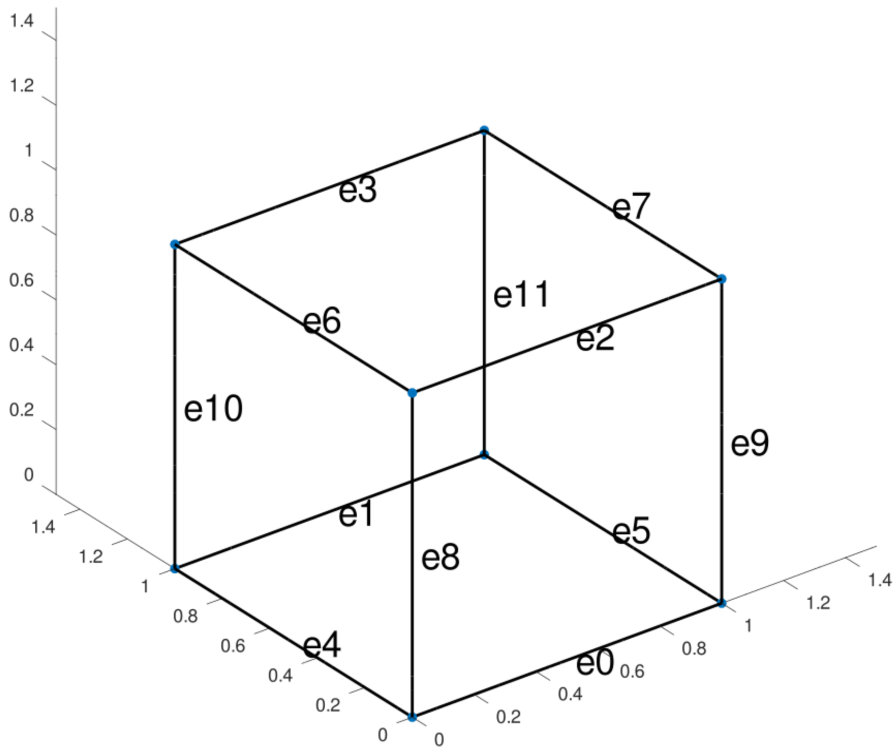

Figure 2.2.: Visualization of the edges of a 3-cube.

Note how 2. implies that $T(\mathbf{0}) = \mathbf{0}$ because otherwise $T(c\mathbf{0}) \neq cT(\mathbf{0})$ for $c \neq 0$.[9]

All linear transformations T in a vector space can be represented by some matrix $\boldsymbol{A}$ like[10]

$$T(\boldsymbol{v}) = \boldsymbol{A}\boldsymbol{v}\,.$$

Conversely, multiplying a vector by a matrix is always a linear transformation.
Proof A.4

It is common to apply multiple transformations to a vertex, the order usually does matter. *Rendering* a mesh involves applying the transformations to each vertex of it.[11] This results in 2-dimensional points on a plane. The line segments that are formed by connecting these points as defined by the edges are the *wire-frame*.

2.3.1. Scaling

To scale some point along the axes by $\boldsymbol{s}$ the components are multiplied one by one[12]

$$\boldsymbol{p}' = \begin{bmatrix} \boldsymbol{s}_x \boldsymbol{p}_x \\ \boldsymbol{s}_y \boldsymbol{p}_y \\ \boldsymbol{s}_z \boldsymbol{p}_z \end{bmatrix}.$$

This can be represented using a matrix as

$$\boldsymbol{p}' = \begin{bmatrix} \boldsymbol{s}_x & 0 & 0 \\ 0 & \boldsymbol{s}_y & 0 \\ 0 & 0 & \boldsymbol{s}_z \end{bmatrix} \boldsymbol{p}\,.$$

[9] Strang, 2011.
[10] Rudin, 1976, p. 210.
[11] Marschner and Shirley, 2016, p. 115.
[12] Boreskov and Shikin, 2014, p. 123.

2.3.2. Rotation

Rotation is the most complicated of the transformations.
This thesis will use Euler angles to represent rotation. These are three numbers α, β and γ that specify the rotation around the X-, Y-, and Z-axis respectively.[13] Euler angles are intuitive but also have drawbacks. Two axes aligning results in *gimbal lock* which can make interpolating between two rotations difficult. A possible solution would be representing rotation in another way, for example using quaternions. However, Euler angles are chosen nonetheless because they are easier to understand and generalize simpler in higher dimensions.[14]

Rotation can be broken down to matrices representing rotation around individual axes.[15]

$$\boldsymbol{R_x}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$
$$\boldsymbol{R_y}(\beta) = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix}$$
$$\boldsymbol{R_z}(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then these are multiplied together yielding a single rotation matrix.

$$\boldsymbol{R}(\alpha, \beta, \gamma) = \boldsymbol{R_x}(\alpha)\boldsymbol{R_y}(\beta)\boldsymbol{R_z}(\gamma)$$

To rotate some point $\boldsymbol{p}$ it is multiplied by this rotation matrix.

$$\boldsymbol{p}' = \boldsymbol{R}(\alpha, \beta, \gamma)\boldsymbol{p}$$

[13] Boreskov and Shikin, 2014, p. 66.
[14] Boreskov and Shikin, 2014, p. 270.
[15] Marschner and Shirley, 2016, p. 124.

2.3.3. Translation

To move, or *translate*, a point by a specific amount in a specific direction $\boldsymbol{t} \in \mathbb{R}^3$ the following formula can be used:[16]

$$\boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{t}\,.$$

Translation is not a linear transformation as $T(\mathbf{0}) = \mathbf{0}$ might not hold true. Therefore it cannot be represented by a 3×3 matrix. However translation in 3-dimensional space can be thought of as shearing in 4-dimensional space. Shearing is a linear transformation[17], thus a 4×4 matrix can be used to represent translation. The points have to be extended with a 4th non-zero w component.

$$\begin{bmatrix} \boldsymbol{p}'_x \\ \boldsymbol{p}'_y \\ \boldsymbol{p}'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \boldsymbol{t}_x \\ 0 & 1 & 0 & \boldsymbol{t}_y \\ 0 & 0 & 1 & \boldsymbol{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{p}_x \\ \boldsymbol{p}_y \\ \boldsymbol{p}_z \\ 1 \end{bmatrix}$$

2.3.4. Projection

Projection maps points in a 3-dimensional space to a 2-dimensional plane. There are multiple approaches to projection with different benefits and drawbacks. This thesis will discuss two of them: orthographic projection and perspective projection.

Orthographic Projection

Orthographic projection is arguably the simplest form of projection. Given a projection plane π and a direction $\boldsymbol{l}$ to project along, the projection $\boldsymbol{p}'$ of a point $\boldsymbol{p}$ is the intersection of the line through $\boldsymbol{p}$ parallel to $\boldsymbol{l}$ with the plane π.[18] We can define π as the set of all points $\boldsymbol{r}$ for which the equation $\boldsymbol{r} \cdot \boldsymbol{n} + d = 0$ holds true. $\boldsymbol{n}$ is a normal (unit-)vector to π and $d = \boldsymbol{r}_0 \cdot \boldsymbol{n}$ is the distance

[16] Marschner and Shirley, 2016, p. 128.
[17] Marschner and Shirley, 2016, p. 111.
[18] Boreskov and Shikin, 2014, p. 70.

between π and the origin $\mathbf{0}$, where $\boldsymbol{r}_0$ is any point of π. The projection direction $\boldsymbol{l}$ is usually chosen so that it is normal to the plane π.[19]
The line through $\boldsymbol{p}$ parallel to $\boldsymbol{l}$ can be defined as $g\colon \boldsymbol{p} + t\boldsymbol{l}$ where $t \in \mathbb{R}$.

Solving for the projection $\boldsymbol{p'}$ yields the following formula:

$$\boldsymbol{p'} = \boldsymbol{p} - \frac{d + \boldsymbol{p} \cdot \boldsymbol{n}}{\boldsymbol{l} \cdot \boldsymbol{n}} \boldsymbol{l} \, .$$

Proof A.1.1

However choosing a specific plane to project onto can substantially reduce the complexity of projection. If we choose π to be the XY-plane and the direction of projection the Z-axis the formula simplifies to

$$\boldsymbol{p'} = \begin{bmatrix} \boldsymbol{p}_x \\ \boldsymbol{p}_y \\ 0 \end{bmatrix} .$$

Proof A.1.2

This can also be represented as the following matrix multiplication

$$\boldsymbol{p'} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \boldsymbol{p} \, .$$

The points are parallelly "squashed" into the plane. This makes orthographic projection simple to implement but the results hard to understand as the depth information is lost.[20]

Perspective Projection

Perspective projection is similar to orthographic projection, there is still a projection plane π, however the points aren't projected along a fixed direction but instead towards a projection center $\boldsymbol{c}$. Thus the projection $\boldsymbol{p'}$ of the point

[19] Boreskov and Shikin, 2014, p. 29.
[20] Boreskov and Shikin, 2014, p. 71.

$\boldsymbol{p}$ is the intersection of π and the line through $\boldsymbol{p}$ and $\boldsymbol{c}$ where $\boldsymbol{p} \neq \boldsymbol{c}$.[21]

Assuming the simple case that π is the plane $z = 0$ and $\boldsymbol{c} = (0\ 0\ d)$ we can define $\boldsymbol{q} = (0\ 0\ \boldsymbol{p}_z)$ and $\boldsymbol{q}' = (0\ 0\ 0)$. Then $\triangle_{\boldsymbol{cpq}} \sim \triangle_{\boldsymbol{cp'q'}}$ therefore $\|\overline{\boldsymbol{cp}}\| : \left\|\overline{\boldsymbol{cp}'}\right\| = \|\overline{\boldsymbol{cq}}\| : \left\|\overline{\boldsymbol{cq}'}\right\|$. Solving for $\boldsymbol{p}'$ we get

$$\boldsymbol{p}' = -\frac{d}{\boldsymbol{p}_z - d}\boldsymbol{p}\,.$$

Proof A.2

In matrix form this could be written as

$$\boldsymbol{p}' = \begin{bmatrix} -\frac{d}{\boldsymbol{p}_z - d} & 0 & 0 \\ 0 & -\frac{d}{\boldsymbol{p}_z - d} & 0 \\ 0 & 0 & -\frac{d}{\boldsymbol{p}_z - d} \end{bmatrix} \boldsymbol{p}\,.$$

This makes objects that are further away from the plane appear smaller, resulting in a more natural looking output that is often easier to understand.

[21] Boreskov and Shikin, 2014, p. 72.

3. N-D Rendering

This chapter will generalize the concepts explained in chapter 2 to arbitrarily high dimensions.

3.1. Representation

Suppose that the vertices of a mesh are points in an n-dimensional euclidean space with the axes $x_1, x_2, \ldots, x_n$. These axes are given by the vectors $\boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{e_n}$. They are defined as

$$\boldsymbol{e_{a}}_i = 0 \text{ except } \boldsymbol{e_{a}}_a = 1$$

which results in

$$\boldsymbol{e_1} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \boldsymbol{e_2} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \ldots, \quad \boldsymbol{e_n} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

3.2. Transformations

The transformations need to be generalized to the form $T \colon \mathbb{R}^n \to \mathbb{R}^n$.

3.2.1. Scaling

To scale a point by $\boldsymbol{s}$ the components are multiplied one by one.

$$\boldsymbol{p}' = \begin{bmatrix} \boldsymbol{s}_1\boldsymbol{p}_1 \\ \boldsymbol{s}_2\boldsymbol{p}_2 \\ \vdots \\ \boldsymbol{s}_n\boldsymbol{p}_n \end{bmatrix}$$

This can be represented using a matrix as

$$\boldsymbol{p}' = \begin{bmatrix} \boldsymbol{s}_1 & 0 & \cdots & 0 \\ 0 & \boldsymbol{s}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \boldsymbol{s}_n \end{bmatrix} \boldsymbol{p} \, .$$

3.2.2. Rotation

Rotation is the most complicated transformation to generalize. Even though rotation is often intuitively understood as rotating *around an axis* it is more helpful to think of it as rotating *in a 2-dimensional plane*. For example, in 3 dimensions, rotating around the X-axis is equivalent to rotating in the YZ-plane.[1] All points of a plane parallel to the YZ-plane will be mapped to points of that same plane.
We can define a plane in terms of two axes, therefore the amount of planes to rotate around in n dimensions can be calculated as

$$\frac{n(n-1)}{2} \, .$$

Proof A.3

Consequently this is also the amount of angles needed to represent the rotation of a point. The angle specifying rotation around the $x_a x_b$-plane will be denoted

[1]Noll, 1967, pp. 469–470.

as α_{ab}. A rotation matrix $\boldsymbol{R_{ab}}$ can then be created with the elements[2]

$$\begin{aligned} \boldsymbol{R_{ab}}_{ii} &= 1 \text{ except } \boldsymbol{R_{ab}}_{aa} = \boldsymbol{R_{ab}}_{bb} = \cos\alpha_{ab} \\ \boldsymbol{R_{ab}}_{ij} &= 0 \text{ except } \boldsymbol{R_{ab}}_{ab} = -\boldsymbol{R_{ab}}_{ba} = -\sin\alpha_{ab}\,. \end{aligned}$$

The rotation matrix yielded will resemble something like this:

$$\boldsymbol{R_{ab}}(\alpha_{ab}) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cos\alpha_{ab} & \cdots & 0 & \cdots & -\sin\alpha_{ab} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & \sin\alpha_{ab} & \cdots & 0 & \cdots & \cos\alpha_{ab} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}.$$

These individual matrices can again be combined to a single rotation matrix by multiplying them together.[3]

$$\boldsymbol{R}(\alpha_{12}, \ldots, \alpha_{(n-1)n}) = \prod_{i=1}^{n-1} \prod_{j=i+1}^{n} \boldsymbol{R_{ij}}(\alpha_{ij})$$

To rotate some point $\boldsymbol{p}$ it is multiplied by this rotation matrix.

$$\boldsymbol{p}' = \boldsymbol{R}(\alpha_{12}, \ldots, \alpha_{(n-1)n})\boldsymbol{p}$$

3.2.3. Translation

Translation is just adding two vectors, therefore in n dimensions it can still be written as

$$\boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{t}\,.$$

[2] Noll, 1967, p. 469.
[3] Noll, 1967, p. 470.

As explained in subsection 2.3.3, translation in n dimensions is not linear. It has to be regarded as an $n+1$ dimensional shear. Thus a $(n+1) \times (n+1)$ matrix is used.

$$\begin{bmatrix} \boldsymbol{p}'_1 \\ \boldsymbol{p}'_2 \\ \vdots \\ \boldsymbol{p}'_n \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 & \boldsymbol{t}_1 \\ 0 & 1 & \cdots & 0 & \boldsymbol{t}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \boldsymbol{t}_n \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \vdots \\ \boldsymbol{p}_n \\ 1 \end{bmatrix}$$

3.2.4. Projection

Projection can be understood of as mapping a point $\boldsymbol{p_n}$ in an n-dimensional space to an $(n-1)$-dimensional hyperplane π. Thus a point can be repeatedly projected down to lower-dimensional hyperplanes until a 2-hyperplane is reached. Note that the bold subscript denotes the dimensionality of the point, the ith component of this point is $\boldsymbol{p_n}_i$.

Orthographic Projection

Orthographic projection projects points parallelly along a given direction of projection $\boldsymbol{l}$ onto the "projection hyperplane" π. For the sake of simplicity, the direction x_n and the hyperplane $x_n = 0$ is chosen for every iteration. Then the projection $\boldsymbol{p_{n-1}}$ can be calculated with the formula

$$\boldsymbol{p_{n-1}} = \begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ 0 \end{bmatrix}.$$

Proof A.1.2

This can be represented as the matrix

$$\boldsymbol{p_{n-1}} = \begin{bmatrix} 1 & \cdots & 0 & 0 \\ \vdots & \ddots & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \boldsymbol{p_n} \,.$$

It is trivial to see that repeatedly projecting the point until reaching a 2-dimensional plane just results in

$$\boldsymbol{p_2} = \begin{bmatrix} \boldsymbol{p_n}_1 \\ \boldsymbol{p_n}_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} .$$

In matrix form this would look like

$$\boldsymbol{p_2} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \boldsymbol{p_n} .$$

Perspective Projection

Perspective projection involves a center of projection $\boldsymbol{c} \in \mathbb{R}^n$ and the "projection hyperplane" π. $\boldsymbol{p_{n-1}}$ is the intersection of $\overline{\boldsymbol{c}\boldsymbol{p_n}}$ and π.
For the sake of simplicity, $\boldsymbol{c}$ is chosen to be a point on the x_n axis with the nth component $d \in \mathbb{R}^*$ and π is chosen to be the hyperplane $x_n = 0$.

Solving the above statement for $\boldsymbol{p_{n-1}}$ yields the formula

$$\boldsymbol{p_{n-1}} = -\frac{d}{\boldsymbol{p_n}_n - d} \begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ 0 \end{bmatrix} .$$

Proof A.2

4. Computer Hardware

This chapter will discuss the workings of modern computer hardware and give suggestions on how to optimally utilize it for rendering as discussed in the previous chapters.

Computer *hardware* is generally understood to be set of physical components that a computer is made up of. On the flip-side, *software* is the set of programs to be executed by a computer. A *program* is a collection of instructions.

It should be noted that this chapter is written primarily with consumer computers in mind, however most of the concepts explained should apply to other types as well.

4.1. Components

Note. *Only the rendering-relevant components will be discussed.*

4.1.1. Central processing unit

The central processing unit (CPU) performs calculations by sequentially executing instructions given to it. These can be arithmetic or logical computations or input/output (I/O) like loading or storing data. The exact form of these instructions depends on instruction set architecture (ISA), however luckily this is of little importance when programming.[1]

[1] Bryant and O'Hallaron, 2016, p. 93.

4.1.2. Random-access memory

Random-access memory (RAM) is a form of volatile storage. It is used to store programs and the data they manipulate.[2]

4.1.3. Graphics processing unit

The graphics processing unit (GPU) is a specialized piece of hardware used to *accelerate* the rendering of images. Thus rendering using a GPU is sometimes referred to as *GPU-accelerated* or simply *hardware-accelerated* rendering.

Rendering a frame of a scene can involve millions of computations, processing them sequentially with the CPU would take too long. The GPU alleviates this problem by allowing to process huge amounts of data in a parallel fashion.

4.1.4. Display

An *electronic visual display*, sometimes referred to just as a *screen*, is a peripheral device used to produce a temporary image of some visual data.

Displays typically output a raster of *pixels*. Each pixel is assigned some color.

Technically, a display is not strictly required for rendering in and of itself, as the results could also be simply saved for later usage, for example in the form of a video file.

4.2. The Graphics Pipeline

Understanding the graphics pipeline is of vital importance to comprehend how a GPU works. Rendering can be seen as manipulating data in a sequence of steps (stages). The collection of these steps is called the graphics pipeline.[3]

[2] Bryant and O'Hallaron, 2016, pp. 94–96.
[3] Boreskov and Shikin, 2014, p. 4.

4.2.1. Graphics APIs

A graphics application programming interface (API) is an interface through which programs communicate with the GPU. It provides a useful abstraction over the underlying hardware and enables greater portability. Some examples for graphics APIs include, but are not limited to:

- OpenGL
- Vulkan
- Direct3D.

Different graphics APIs provide various amount of control over the hardware. This results in some minor differences between them. Efforts have been made to keep the following sections as API-agnostic as possible.

4.2.2. Stages

Roughly speaking the graphics pipeline can be split up into 3 major parts:

- application stage
- geometry stage
- raster stage.[4]

```
[Application]  ---->  [Geometry]  ---->  [Raster]
```

Figure 4.1.: Pipeline stages

Some parts of the pipeline are hard-coded while others can compute custom logic in the form of a *shader*. Shaders can be thought of as programs that run on the GPU.

[4] Boreskov and Shikin, 2014, p. 5.

Application stage

The application stage is the part of the program running on the CPU. It creates the data that is subsequently fed to the GPU by making calls to the graphics API. Non-rendering-relevant logic, like for example collision detection, would be implemented here.

Geometry stage

The geometry stage runs on the GPU. It is responsible for processing the vertices by applying the computations described in chapter 2 like transformations and projections. The exact logic for how this is ought to be done is specified in the *vertex shader*. Vertex shaders can be thought of as small programs that are paralelly invoked for each vertex. Clipping (the removal of non-visible parts of objects) is also done in the geometry stage, as well as window-viewport transformations (the mapping of objects to an area of the screen).

Raster stage

Rasterization is arguably one of the most important tasks of the GPU. Rasterization is the process of creating a raster image of some abstract vector image data. In practice this means taking data that describes a primitive shape and calculating which pixels are affected by its presence. These pixels are then colored by a *fragment shader* (sometimes also referred to as *pixel shader*). Fragment shaders can be thought of as small programs that are paralelly invoked for each pixel. The primitives usually supported are dots, lines and triangles.

Further post-processing can be done on the resulting image, however this thesis should not go into those details.

4.3. N-D rendering using the GPU

Unsurprisingly, GPUs have become very efficient at 3D rendering. Similarly, with a few hacks n-D rendering can also be accelerated using GPUs. Some differences between 3D and n-D rendering will be explained in the following.

4.3.1. Matrix limitations

Representing transformations as matrices is a common pattern in 3D computer graphics. The position, rotation and scale (from here on called the *transform*) of an object are stored as vectors. Every frame the CPU computes a 4×4 *model matrix* in accordance with the rules explained in chapter 2. It is then multiplied with the *view matrix* and *projection matrix* yielding the *model view projection matrix*. This matrix is subsequently passed to the GPU using a *uniform* variable. The vertex shader reads this uniform and uses it to transform the vertex by performing a single matrix multiplication on it.

However, using this approach for n-D rendering has two drawbacks:

1. memory usage
2. computational complexity.

Memory usage

The transformation matrix for an n-dimensional object takes up

$$S_{\text{mat}} = S_{\text{num}} \cdot (n+1)^2$$

bytes, where S_{num} is the size of a single number in bytes. Assuming 32-bit floating point numbers yields $S_{\text{num}} = 4$.

In contrast, an n-dimensional transform takes up only

$$S_{\text{transform}} = S_{\text{num}} \cdot \frac{n(n+3)}{2}$$

bytes.[Proof A.5]

This means that for large n the transformation matrix is about twice the size of the transform itself because

$$\lim_{n \to \infty} \frac{S_{\text{mat}}}{S_{\text{transform}}} = 2\,.$$

Proof A.6

Computational complexity

A naive way of creating an n-dimensional model matrix would require

$$\frac{n(n-1)}{2} + 1$$

matrix multiplications, creating the projection matrix requires $n - 3$.

Matrix multiplication itself has a computational complexity of $O(n^3)$ meaning that creating the model matrix would be $O(n^5)$.

Remark. *Technically there are matrix multiplication algorithms with a better computational complexity than $O(n^3)$, however the improvement they offer is negligible in this case.*[5]

Thus one should seriously consider to abstain from matrices altogether. An alternative would be to pass the raw transform of an object through a uniform to the shader as one would do with the matrix. The shader can then manipulate the vertex accordingly. This can actually be achieved with a computational complexity of $O(n^2)$ as demonstrated in subsection 5.3.1.

4.3.2. Attribute limitations

A further problem is getting the vertex data to the vertex shader. Usually this is done with a vertex buffer object (VBO). The coordinates of each object are sequentially stored in a buffer object called the VBO. The VBO is then used to source an *attribute* variable in the vertex shader. Each instance of the vertex shader (responsible for a unique vertex) thus receives a slice of the VBO corresponding to the coordinates of the vertex in question.

Because in 3D rendering each vertex consists of at maximum 3 coordinates and its transformation requires a 4×4 matrix the size of such a "slice" (correctly *attribute*) is oftentimes limited to a length of 4 numbers.[6][7] Yet for n-D rendering n numbers are required.

[5] Bläser, 2013, pp. 2–3.
[6] Segal and Akeley, 2022, pp. 352–354, 358.
[7] The Khronos® Vulkan Working Group, 2023, pp. 1973–1974, 3123–3152.

Multiple attributes

One possible workaround would be to use multiple attributes. Although the size of a single attribute is limited, one can usually use more than one of them. Then the attributes would be read from the VBO in an interleaved way so that the vertex shader is able to piece together the coordinates of the vertex. A drawback of this approach is that it does not lend itself well to a general implementation that works for every n because the amount of attributes would also vary. Thus some sort of preprocessing of the vertex shader would be required.

Vertex pulling

A likely better solution is *vertex pulling*. Vertex pulling is a technique where all the vertex data is stored in some global buffer on the GPU. Then each instance of the vertex shader extracts the data relevant to it according to some custom logic.[8] It is easy to see how this alleviates the problem by allowing each shader to access as much data as is required.

Optimally the graphics API and hardware provide the ability to create and use shader storage buffer objects (SSBOs). An SSBO is large general-purpose buffer.[9]

If there is no access to SSBOs one could also abuse *textures* to achieve a similar result.[10] Textures are data containers intended to store images which are then read in a fragment shader to be displayed. With some access logic they can however be turned into a quasi-SSBO. The performance hit seems to be tolerable, but one should still be cautious of the performance implications of such unorthodox use.[11]

In extreme cases even a uniform buffer object (UBO) could be considered. UBOs are like SSBOs but faster and smaller.[12] Of course this should only be done if the amount of required vertices is for some reason severely limited.

[8] Cozzi and Riccio, 2012, pp. 293–294.
[9] OpenGL Wiki, 2020.
[10] Cozzi and Riccio, 2012, pp. 294–296.
[11] Cozzi and Riccio, 2012, pp. 296–298.
[12] OpenGL Wiki, 2020.

5. Reference Implementation

This chapter will give an overview of the reference implementation `vulkan-xd`. The source code can be found on GitHub under https://github.com/OliverKovacs/vulkan-xd and in Appendix B.

5.1. Requirements

The following requirements were established for a reference implementation:

- It should provide a good example of the techniques discussed.
- The code should be straight forward and easy to understand.
- The code should be portable to a reasonable degree.
- It should be serve as a good starting point for further projects.

5.2. Decisions

Vulkan was chosen as the graphics API because it is cross-platform (in contrast to for example or Direct3D or Metal) and it provides lower level control over the hardware than OpenGL.

C++ was a natural pick as the programming language as official language bindings are provided for it and there is an abundance of documentation and online resources on how to use it with Vulkan. Make was chosen as the build system and Clang as the compiler, although other compilers should work just fine. The shaders are written in GLSL and compiled with shaderc to SPIR-V.

Alongside the Vulkan SDK the reference implementation also depends on GLFW for cross-platform windowing and GLM for some math utilities.

5.3. Source code

Note. *No attempts will be made to explain parts of the source code as any such would inevitable devolve into a Vulkan tutorial. The best way to understand the source code is by reading it, however this requires a considerable amount of knowledge of Vulkan and C++.*

The source code is structured the following way:

```
vulkan-xd/
├── build
│   ├── assets
│   │   └── texture.png
│   ├── main
│   └── shaders
│       ├── shader.frag.spv
│       └── shader.vert.spv
├── LICENSE.md
├── Makefile
├── README.md
└── src
    ├── assets
    │   └── texture.png
    ├── include
    ├── main
    │   ├── config.hpp
    │   ├── keybinds.cpp
    │   ├── main.cpp
    │   ├── profiler.sh
    │   ├── scene.cpp
    │   ├── vertex.cpp
    │   ├── vertex.hpp
    │   ├── vulkan.cpp
    │   ├── xdvk.cpp
    │   ├── xdvk.hpp
    │   └── xdvk.t.hpp
    └── shaders
        ├── shader.frag
        └── shader.vert
```

The C++ source files are located under `src/main/`. Of special interest are `vulkan.cpp`, which contains most of the Vulkan related code and the `xdvk.*`

files containing the logic for creating and working with higher dimensional objects.

Shaders are located under `src/shaders` and are probably the most interesting part of the program. Therefore the most important parts of the vertex shader exemplifying the techniques discussed in section 4.3 will be provided below.

5.3.1. Vertex shader

```
void fetchVertex(
    inout float[n] vertex,
    int stride,
    int offset
) {
    int block = n + stride;
    int index = offset + gl_VertexIndex * block;
    for (int i = 0; i < n; i++) {
        vertex[i] = ssbo.vertices[index + i];
    }
}
```

Listing 5.1: Vertex pulling

```
void scaleVertex(inout float vertex[n], inout float scale[n]) {
    for (int i = 0; i < n; i++) {
        vertex[i] *= scale[i];
    }
}

void rotateVertex(
    inout float vertex[n],
    inout float rotation[a_n],
) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n; j++) {
            if (j <= i) continue;
            const float a = rotation[
                a_n
                - int(float((n - i - 1) * (n - i)) / 2.0)
                + j
                - i
                - 1
            ];
```

```
            const float cos_a = cos(a);
            const float sin_a = sin(a);
            const float vi = vertex[i];
            const float vj = vertex[j];
            vertex[i] = vi *  cos_a + vj * sin_a;
            vertex[j] = vi * -sin_a + vj * cos_a;
        }
    }
}

void translateVertex(
    inout float vertex[n],
    inout float position[n]
) {
    for (int i = 0; i < n; i++) {
        vertex[i] += position[i];
    }
}

void transformVertex
    inout float vertex[n],
    Transform transform
) {
    scaleVertex(vertex, transform.scale);
    rotateVertex(vertex, transform.rotation);
    translateVertex(vertex, transform.position);
}
```

Listing 5.2: Transformations

```
void projectVertex(inout float vertex[n]) {
    float z_diff = canvas_z - camera_z;
    for (int i = n - 1; i >= 2; i--) {
        float w = z_diff / (canvas_z - vertex[i]);
        for (int j = 0; j < n; j++) {
            if (j >= i) break;
            vertex[j] *= w;
        }
    }
}
```

Listing 5.3: Projection

```
// ...

float[n] vertex;
```

```
Transform transform;
fetchVertex(vertex, 0, int(constants.vertexIndex));
fetchTransform(transform, int(constants.transformIndex), 0, 0);
transformVertex(vertex, transform);
projectVertex(vertex);

// ...
```

Listing 5.4: All

5.4. Performance

Some rudimentary performance analysis was conducted on the reference implementation.

5.4.1. Disclaimer

The given results should not be taken at face value. As already discussed, the focus of this project was not performance but rather simplicity and extensibility. Thus comparing these numbers to other benchmarks makes little sense. They should much more be seen as a demonstration that higher-dimensional rendering can be achieved, and done so with any reasonable performance at all.

5.4.2. Setup

The performance data was collected using the `VK_LAYER_MESA_overlay` Vulkan layer. All tests where conducted on a Dell Inspiron 7580 Laptop with the following specifications:

```
OS:    Pop!_OS 22.04 LTS
CPU:   Intel i7-8565U (4 Cores, 8 Threads @ 4.6 MHz)
RAM:   2x8 GB (DDR4, 2667 MT/s)
GPU0:  Intel WhiskeyLake-U GT2 UHD Graphics 620
GPU1:  NVIDIA GeForce MX150 (Pascal, 2 GB GDDR5 VRAM)
```

Three groups of scenes were benchmarked on both the integrated and dedicated GPU. These scenes are:

1. Default: A 4-cube (tesseract) and 24-cell (icositetrachoron).
2. Hypercube: A single n-cube.
3. Hypercubes: 250 randomly distributed n-cubes.

The latter two scenes were benchmarked with three dimensionalities each: 4D, 8D and 12D.

The average frames per second (FPS) over a time period of 10 second as well as minimum and maximum frame time were captured. VSync was disabled while benchmarking for obvious reasons.

The source code for each benchmarked scene is stored on a separate git branch. The naming scheme for these branches is

benchmark-<scene>-<dimensions> .

5.4.3. Results

scene	dimensions	vertices[1]	GPU0			GPU1		
			FPS	min[2]	max[3]	FPS	min[2]	max[3]
default	4	256	281	3.0	5.0	912	0.8	1.9
hypercube	4	64	283	2.8	4.9	943	0.8	1.6
	8	2048	235	3.6	5.5	844	0.9	2.1
	12	19 K	112	6.4	10.9	323	2.4	4.2
hypercubes	4	16 K	238	3.6	5.6	692	1.1	2.3
	8	512 K	110	6.4	11.5	413	2.1	3.7
	12	12 M	3	326.0	344.5	33	28.3	33.4

1: Amount of vertices per frame ($K = 10^3$, $M = 10^6$)

2: Minimum frame time in milliseconds

3: Maximum frame time in milliseconds

5.5. Gallery

This section contains some images rendered by vulkan-xd. The line width was adjusted for better visibility. All images were directly extracted from the swapchain framebuffer or depth attachment using RenderDoc.

A video demo of vulkan-xd can be found online under:
https://www.youtube.com/watch?v=yCssM-TOu4w.

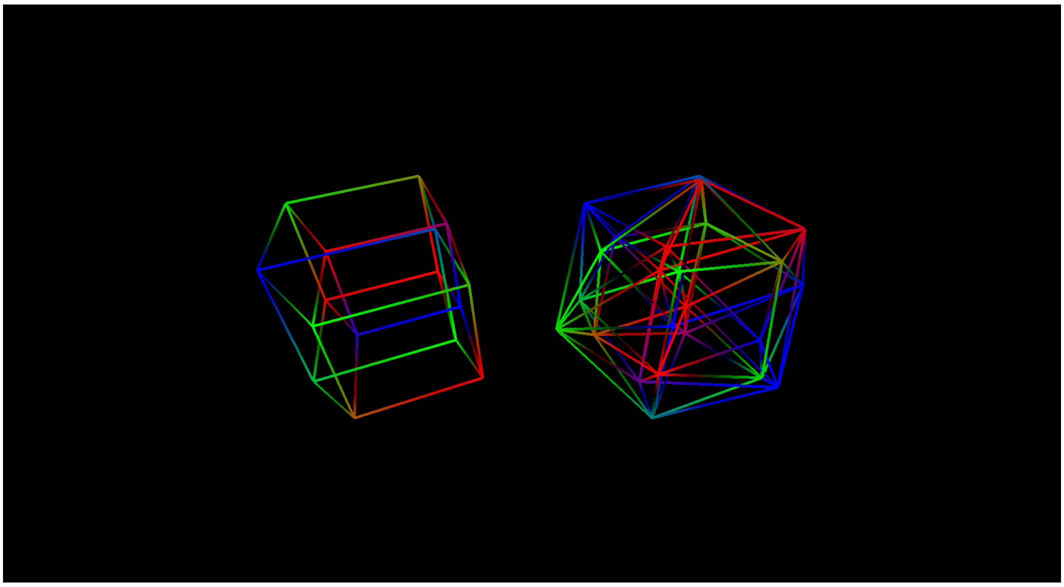

Figure 5.1.: Default scene showing a 4-cube (tesseract) and 24-cell (icositetrachoron).

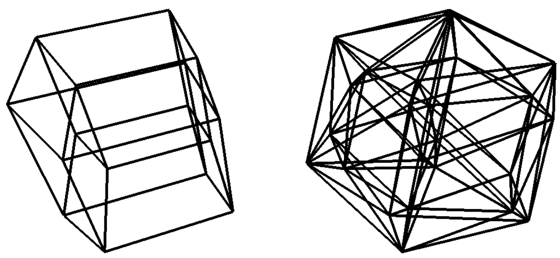

Figure 5.2.: 2D depth attachment of Figure 5.1.

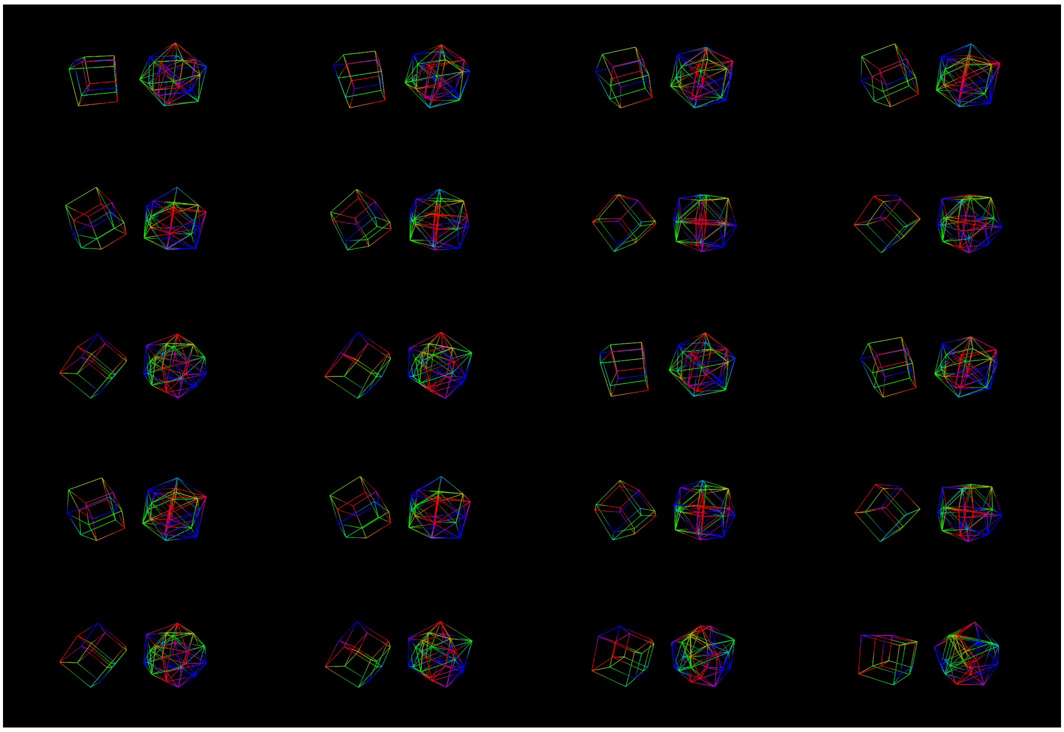

Figure 5.3.: First 20 frames of the default scene.

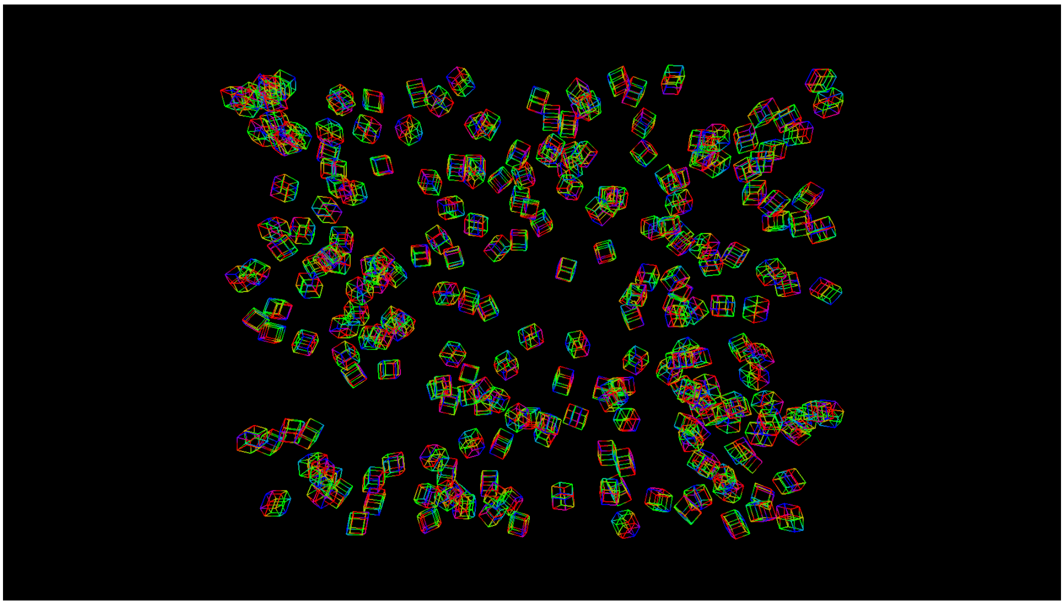

Figure 5.4.: 250 randomly distributed 4-cubes (tesseracts).

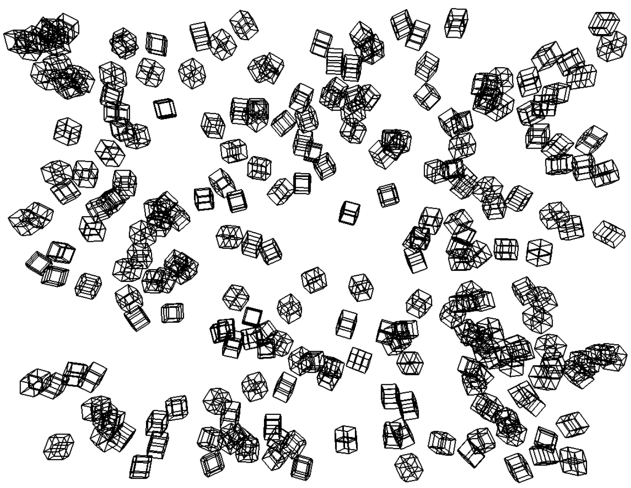

Figure 5.5.: 2D depth attachment of Figure 5.4.

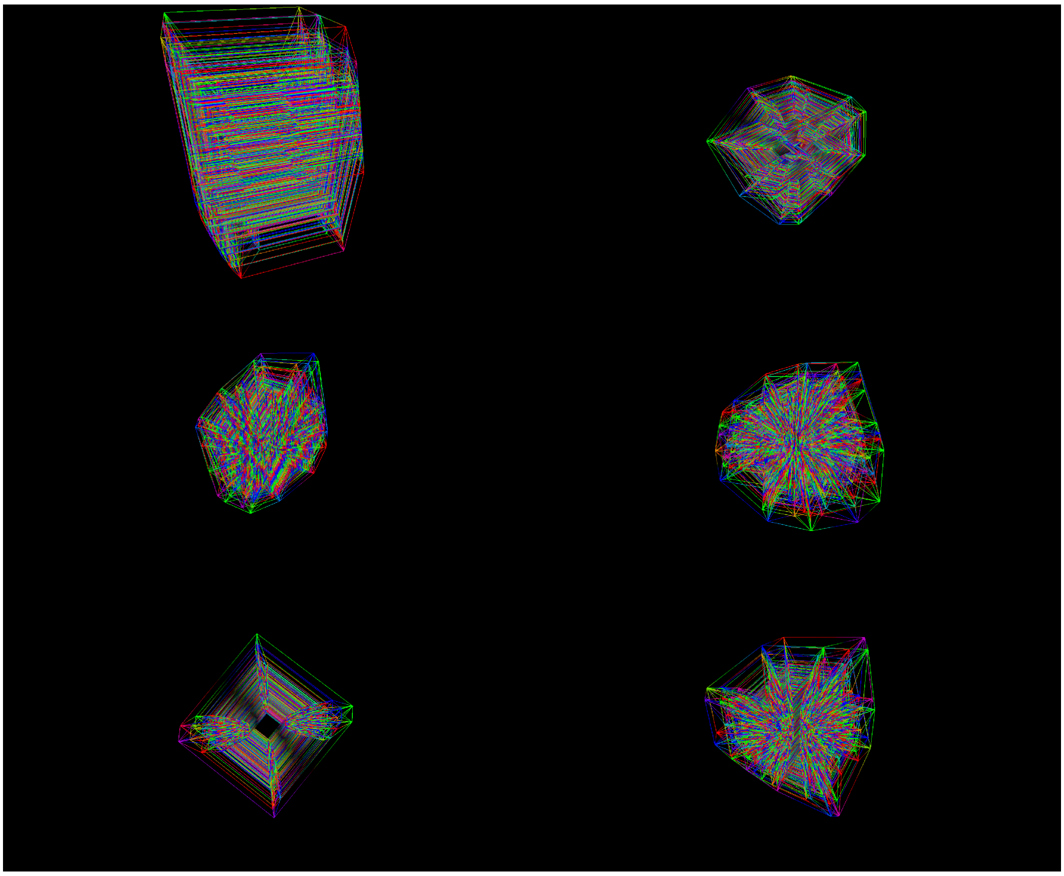

Figure 5.6.: 12-cube in different rotations.

Appendix

Appendix A.

Proofs

A.1. Orthographic projection

A.1.1. General case

Proposition A.1.1. *Let $\boldsymbol{p_n}$ be some point in an n-dimensional euclidean space with the axes $x_1, x_2, \ldots, x_n$. Let π be the hyperplane given by $\boldsymbol{r} \cdot \boldsymbol{n} + d = 0$ where*

1. $\boldsymbol{n} \perp \pi$,
2. $\|\boldsymbol{n}\| = 1$,
3. $d = \boldsymbol{r_0} \cdot \boldsymbol{n}$ *and*
4. $\boldsymbol{r_0} \in \pi$.

Let $\boldsymbol{l} \not\perp \boldsymbol{n}$ be the projection direction. Then the orthographic projection $\boldsymbol{p_{n-1}} = \boldsymbol{p_n} + t\boldsymbol{l} \cap \pi$ where $t \in \mathbb{R}$ of $\boldsymbol{p_n}$ is given by the formula

$$\boldsymbol{p_{n-1}} = \boldsymbol{p_n} - \frac{d + \boldsymbol{p_n} \cdot \boldsymbol{n}}{\boldsymbol{l} \cdot \boldsymbol{n}} \boldsymbol{l} \, .$$

Note. *Geometrically d represents the distance between the hyperplane π and the origin $\mathbf{0}$.*

Proof. Solve for t.

$$
\begin{aligned}
&\begin{cases}
\boldsymbol{p_{n-1}} = \boldsymbol{p_n} + t\boldsymbol{l} \\
\boldsymbol{p_{n-1}} \cdot \boldsymbol{n} + d = 0
\end{cases} \\
(\boldsymbol{p_n} + t\boldsymbol{l}) \cdot \boldsymbol{n} + d &= 0 \\
(\boldsymbol{p_n} + t\boldsymbol{l}) \cdot \boldsymbol{n} &= -d \\
\boldsymbol{p_n} \cdot \boldsymbol{n} + t\boldsymbol{l} \cdot \boldsymbol{n} &= -d \\
t\boldsymbol{l} \cdot \boldsymbol{n} &= -d - \boldsymbol{p_n} \cdot \boldsymbol{n} \\
t &= -\frac{d + \boldsymbol{p_n} \cdot \boldsymbol{n}}{\boldsymbol{l} \cdot \boldsymbol{n}}
\end{aligned}
$$

Solve for $\boldsymbol{p_{n-1}}$.

$$
\begin{aligned}
\boldsymbol{p_{n-1}} &= \boldsymbol{p_n} - t\boldsymbol{l} \\
\boldsymbol{p_{n-1}} &= \boldsymbol{p_n} - \frac{d + \boldsymbol{p_n} \cdot \boldsymbol{n}}{\boldsymbol{l} \cdot \boldsymbol{n}}\boldsymbol{l}
\end{aligned}
$$

[1] □

A.1.2. Specific case

Proposition A.1.2. *Assume that the projection plane is* $x_n = 0$ *and the projection direction* x_n*. Then the projection becomes*

$$
\boldsymbol{p_{n-1}} = \begin{bmatrix} \boldsymbol{p}_{\boldsymbol{n}1} \\ \vdots \\ \boldsymbol{p}_{\boldsymbol{n}n-1} \\ 0 \end{bmatrix}.
$$

[1]Boreskov and Shikin, 2014, p. 71.

Proof. The projection plane is $x_n = 0$ and the projection direction x_n.

$$
\begin{aligned}
&\implies \boldsymbol{n} = \boldsymbol{l} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \\
&\implies \boldsymbol{r_0} = \boldsymbol{0} \\
&\implies d = 0
\end{aligned}
$$

Substitute and simplify.

$$
\begin{aligned}
\boldsymbol{p_{n-1}} &= \boldsymbol{p_n} - \frac{d + \boldsymbol{p_n} \cdot \boldsymbol{n}}{\boldsymbol{l} \cdot \boldsymbol{n}} \boldsymbol{l} \\
\boldsymbol{p_{n-1}} &= \boldsymbol{p_n} - \frac{0 + \boldsymbol{p_n}_n}{1} \boldsymbol{l} \\
\boldsymbol{p_{n-1}} &= \boldsymbol{p_n} - \boldsymbol{p_n}_n \boldsymbol{l} \\
\boldsymbol{p_{n-1}} &= \boldsymbol{p_n} - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \boldsymbol{p_n}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ 0 \end{bmatrix}
\end{aligned}
$$

□

Alternative proof using perspective projection

Proof. Orthographic projection can be interpreted as a special case of perspective projection where the center of projection is infinitely far behind the projection plane.[2]

$$
\implies d = \infty
$$

Calculate the limit.

$$
\boldsymbol{p_{n-1}} = \lim_{d \to \infty} \left(-\frac{d}{\boldsymbol{p_n}_n - d} \begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ 0 \end{bmatrix} \right) = \begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ 0 \end{bmatrix}
$$

[2] Noll, 1967, p. 470.

□

A.2. Perspective projection

Proposition A.2.1. *Let $\boldsymbol{p_n}$ be a point in an n-dimensional euclidean space with the axes $x_1, x_2, \ldots, x_n$. Let π be the hyperplane $x_n = 0$ and $\boldsymbol{c}$ a point on the x_n axis with the nth component $d \in \mathbb{R}$. Then the perspective projection $\boldsymbol{p_{n-1}} = \overline{\boldsymbol{cp_n}} \cap \pi$ of $\boldsymbol{p_n}$ for $\boldsymbol{p_n} \neq \boldsymbol{c}$ is given by*

$$\boldsymbol{p_{n-1}} = -\frac{d}{\boldsymbol{p_n}_n - d}\begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ 0 \end{bmatrix}.$$

Proof. $\boldsymbol{c}$ is a point on the x_n axis with the nth component $d \in \mathbb{R}$.

$$\implies \boldsymbol{c} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ d \end{bmatrix}$$

Getting the parametric representation of $\overline{\boldsymbol{cp_n}}$.

$$\overline{\boldsymbol{cp_n}} = \boldsymbol{c} + t\overrightarrow{\boldsymbol{cp_n}} \text{ where } t \in \mathbb{R}$$

Calculating the direction vector.

$$\overrightarrow{\boldsymbol{cp_n}} = \boldsymbol{p_n} - \boldsymbol{c} = \begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ \boldsymbol{p_n}_n - d \end{bmatrix}$$

Substituting and solving for t:

$$\begin{cases} \boldsymbol{c} + t(\boldsymbol{p_n} - \boldsymbol{c}) \\ x_n = 0 \end{cases}$$

$$\begin{aligned} \boldsymbol{c}_n + t(\boldsymbol{p_n}_n - \boldsymbol{c}_n) &= 0 \\ d + t(\boldsymbol{p_n}_n - d) &= 0 \\ t(\boldsymbol{p_n}_n - d) &= -d \\ t &= -\frac{d}{\boldsymbol{p_n}_n - d} \end{aligned}$$

Solving for $\boldsymbol{p_{n-1}}$ with t:

$$\begin{aligned} \boldsymbol{p_{n-1}} &= \boldsymbol{c} + t\overrightarrow{\boldsymbol{cp_n}} \\ \boldsymbol{p_{n-1}} &= \boldsymbol{c} + t\overrightarrow{\boldsymbol{cp_n}} \\ \boldsymbol{p_{n-1}} &= \boldsymbol{c} - \frac{d}{\boldsymbol{p_n}_n - d}\overrightarrow{\boldsymbol{cp_n}} \\ \boldsymbol{p_{n-1}} &= \boldsymbol{c} - \frac{d}{\boldsymbol{p_n}_n - d}\begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ \boldsymbol{p_n}_n - d \end{bmatrix} \end{aligned}$$

Simplifying:

$$\boldsymbol{p_{n-1}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ d \end{bmatrix} - \frac{d}{\boldsymbol{p_n}_n - d} \begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ \boldsymbol{p_n}_n - d \end{bmatrix}$$

$$\boldsymbol{p_{n-1}} = -\frac{d}{\boldsymbol{p_n}_n - d} \left(\begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ \boldsymbol{p_n}_n - d \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ 0 \\ -\frac{\boldsymbol{p_n}_n - d}{d} d \end{bmatrix} \right)$$

$$\boldsymbol{p_{n-1}} = -\frac{d}{\boldsymbol{p_n}_n - d} \begin{bmatrix} \boldsymbol{p_n}_1 \\ \vdots \\ \boldsymbol{p_n}_{n-1} \\ 0 \end{bmatrix}$$

□

A.3. Amount of rotation planes in n dimensions

Proposition A.3.1. *The amount of rotation planes in an n-dimensional space is given by* $\frac{n(n-1)}{2}$.

Proof. A rotation plane is given by two different unordered axes. There are n axes in n dimensions. The binomial coefficent can be used to calculate the amount of combinations to choose 2 out of n axes.

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$$

□

A.4. Multiplying by a square matrix is a linear transformation

Proposition A.4.1. *Let* $\boldsymbol{A} \in \mathcal{M}_{n \times n}\mathbb{R}$ *and* $\boldsymbol{x} \in \mathbb{R}^n$*.* $T(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x}$ *is always a linear transformation.*

Proof. T is transformation because for all $\boldsymbol{v} \in \mathbb{R}^n$

$$\boldsymbol{A}\boldsymbol{v} \in \mathbb{R}^n \ .$$

T is linear because for all $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ and $c \in \mathbb{R}$

1. $\boldsymbol{A}(\boldsymbol{v} + \boldsymbol{w}) = \boldsymbol{A}\boldsymbol{v} + \boldsymbol{A}\boldsymbol{w}$ and
2. $\boldsymbol{A}(c\boldsymbol{v}) = c\boldsymbol{A}\boldsymbol{v}$.

□

A.5. Size of an n-dimensional transform

Proposition A.5.1. *The amount of bytes* $S_{transform}$ *an* n*-dimensional transform takes up is given by the formula* $S_{transform} = S_{num} \cdot \frac{n(n+3)}{2}$ *where* S_{num} *is the size of a single number.*

Proof. A transform is made up of a position, size and rotation.
The position and size are each an element of $\mathbb{R}^n$, thus they are $2n$ numbers in total.
The rotation consists of $\frac{n(n-1)}{2}$ numbers, see section A.3.
Thus the the total amount of numbers is

$$2n + \frac{n(n-1)}{2} = \frac{n(n+3)}{2} \ .$$

Each of these numbers takes up S_{num} bytes, so

$$S_{\text{transform}} = S_{\text{num}} \cdot \frac{n(n+3)}{2} \ .$$

□

A.6. The size of a transformation matrix is twice the size of the transform for large n

Proposition A.6.1. *The limit* $\lim_{n\to\infty} \frac{S_{mat}}{S_{transform}}$ *evaluates to 2.*

Proof. Solve the limit:

$$\begin{aligned}
\lim_{n\to\infty} \frac{S_{\text{mat}}}{S_{\text{transform}}} &= \lim_{n\to\infty} \frac{S_{\text{num}} \cdot (n+1)^2}{S_{\text{num}} \cdot \frac{n(n+3)}{2}} \\
&= \lim_{n\to\infty} 2 \cdot \frac{(n+1)^2}{n(n+3)} \\
&= \lim_{n\to\infty} 2 \cdot \frac{n^2+2n+1}{n^2+3n} \\
&= \lim_{n\to\infty} 2 \cdot \frac{1+\frac{2}{n}+\frac{1}{n^2}}{1+\frac{3}{n}} \\
&= 2 \cdot \frac{1+0+0}{1+0} \\
&= 2
\end{aligned}$$

□

Appendix B.

Source code

A more practical, clonable and runnable version of the source code can be found on GitHub under https://github.com/OliverKovacs/vulkan-xd. This is only provided for the sake of completeness.

B.1. src/main/

B.1.1. config.hpp

```cpp
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#include <array>
#include <cstdint>
#include <iostream>

const char* vblank_mode_str = std::getenv("vblank_mode");
size_t vblank_mode = vblank_mode_str
    ? std::stoi(vblank_mode_str)
    : 1;

struct Config {
    const size_t default_width = 1920;
    const size_t default_height = 1080;
    bool log_validation_layer = false;
    bool vsync = vblank_mode != 0;
    std::vector<VkPresentModeKHR> preferredPresentModes = {
        this->vsync
            ? VK_PRESENT_MODE_MAILBOX_KHR       // vsync
            : VK_PRESENT_MODE_IMMEDIATE_KHR,    // no vsync, tearing possible
    };
};

const Config config;
```

Listing B.1: src/main/config.hpp

B.1.2. keybinds.cpp

```cpp
#include <GLFW/glfw3.h>

#include <array>
#include <cstdint>

struct Keybinds {
    std::array<uint32_t, 8> directions = {
        GLFW_KEY_A,
        GLFW_KEY_D,
        GLFW_KEY_W,
        GLFW_KEY_S,
        GLFW_KEY_Q,
        GLFW_KEY_E,
        GLFW_KEY_R,
        GLFW_KEY_F
    };
    std::array<uint32_t, 10> rotations = {
        GLFW_KEY_1,
        GLFW_KEY_2,
        GLFW_KEY_3,
        GLFW_KEY_4,
        GLFW_KEY_5,
        GLFW_KEY_6,
        GLFW_KEY_7,
        GLFW_KEY_8,
        GLFW_KEY_9,
        GLFW_KEY_0
    };
};

const Keybinds keybinds {};
```

Listing B.2: src/main/keybinds.cpp

B.1.3. main.cpp

```cpp
#include <algorithm>
#include "vulkan.cpp"
#include "keybinds.cpp"
#include "xdvk.hpp"
#include "scene.cpp"

void Vulkan::callback() {
    for (size_t i = 0; i < scene.entities.size(); i++) {
        auto &entity = scene.entities[i];
        for (size_t j = 0; j < xdvk::rotationSize(DIMENSION); j++) {
            entity.transform.rotation[j] += deltaTime * 0.6 * (j + 0.2) / xdvk::rotationSize(DIMENSION);
        }
        bool shift = glfwGetKey(window, GLFW_KEY_LEFT_SHIFT) == GLFW_PRESS;
        // bool alt = glfwGetKey(window, GLFW_KEY_LEFT_ALT) == GLFW_PRESS;
        float sign = (1 - 2 * shift);
        for (size_t j = 0; j < std::min(xdvk::rotationSize(DIMENSION), keybinds.rotations.size()); j++)
    {
            entity.transform.rotation[j] += deltaTime * 2 * sign * (glfwGetKey(window, keybinds.
    rotations[j]) == GLFW_PRESS);
        }
        for (size_t j = 0; j < std::min((size_t)DIMENSION * 2, keybinds.directions.size()); j++) {
            entity.transform.position[j >> 1] += deltaTime * (1.0 - (!(j & 1) << 1)) * (glfwGetKey(
    window, keybinds.directions[j]) == GLFW_PRESS);
        }
        std::copy_n(scene.entities[i].transform.buffer, entity.geometry.vertices.size(), &storageVectors
    [1][i * xdvk::transformSize(DIMENSION)]);
        entity.geometry.transformBufferIndex = i * xdvk::transformSize(DIMENSION);
    }
}

```

```cpp
auto main() -> int {
    Vulkan app;

    std::cout << "vsync: " << config.vsync << std::endl;

    try {
        app.run();
    }
    catch (const std::exception& error) {
        std::cerr << "ERROR: " << error.what() << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Listing B.3: src/main/main.cpp

B.1.4. scene.cpp

```cpp
#define SCENE_DEFAULT
// #define SCENE_HYPERCUBE
// #define SCENE_TESSERACTS

#ifdef SCENE_DEFAULT

void Vulkan::createModel() {

    const auto hypercube_id = scene.add();
    auto &hypercube = scene.get(hypercube_id);

    const auto icositetrachoron_id = scene.add();
    auto &icositetrachoron = scene.get(icositetrachoron_id);

    std::vector<Vertex> vertices;

    std::vector<std::vector<uint32_t>> indices;
    indices.resize(scene.entities.size());
    xdvk::hypercubeIndices(indices[0], DIMENSION);
    xdvk::icositetrachoronIndices(indices[1]);

    const float size = 0.45;
    xdvk::hypercubeVertices(hypercube.geometry.vertices, DIMENSION, size);
    xdvk::icositetrachoronVertices(icositetrachoron.geometry.vertices, 2 * size);

    for (size_t i = 0; i < scene.entities.size(); i++) {
        Vertex vertex{ static_cast<glm::float32>(i) };
        vertices.resize(scene.entities[i].geometry.vertices.size());
        std::fill_n(vertices.begin(), vertices.size(), vertex);
        createVertexBuffer(vertices, scene.entities[i].geometry.vertexBuffer, scene.entities[i].geometry
  .vertexBufferMemory);
        createIndexBuffer(indices[i], scene.entities[i].geometry.indexBuffer, scene.entities[i].geometry
  .indexBufferMemory);
        scene.entities[i].geometry.indexBufferSize = indices[i].size();
    }

    storageVectors[1].resize(scene.entities.size() * xdvk::transformSize(DIMENSION));

    hypercube.transform.position[0] = -1.0;
    icositetrachoron.transform.position[0] =  1.0;

    size_t index = 0;
    for (size_t i = 0; i < scene.entities.size(); i++) {
        auto &entity = scene.entities[i];
        entity.geometry.vertexBufferIndex = index;
        size_t size = entity.geometry.vertices.size();
        storageVectors[0].resize(index + size);
        std::copy_n(entity.geometry.vertices.data(), size, &storageVectors[0][index]);
        index += size;
```

```
        std::copy_n(scene.entities[i].transform.buffer, size, &storageVectors[1][i * xdvk::transformSize
    (DIMENSION)]);
        entity.geometry.transformBufferIndex = i * xdvk::transformSize(DIMENSION);
    }

    storageVectors[2].resize(1);
}

#endif /* SCENE_DEFAULT */

#ifdef SCENE_HYPERCUBE

void Vulkan::createModel() {

    const float size = 0.45;
    std::vector<Vertex> vertices;
    std::vector<uint32_t> indices;

    // hypercube entity
    const auto entity_id = scene.add();
    auto &entity = scene.get(entity_id);

    xdvk::hypercubeIndices(indices, DIMENSION);
    xdvk::hypercubeVertices(entity.geometry.vertices, DIMENSION, size);

    Vertex vertex{ static_cast<glm::float32>(0) };
    vertices.resize(entity.geometry.vertices.size());
    std::fill_n(vertices.begin(), vertices.size(), vertex);

    createVertexBuffer(vertices, entity.geometry.vertexBuffer, entity.geometry.vertexBufferMemory);
    createIndexBuffer(indices, entity.geometry.indexBuffer, entity.geometry.indexBufferMemory);

    entity.geometry.indexBufferSize = indices.size();

    // vertex ssbo data
    storageVectors[0].resize(entity.geometry.vertices.size());
    std::copy_n(entity.geometry.vertices.data(), entity.geometry.vertices.size(), &storageVectors[0][0])
    ;
    entity.geometry.vertexBufferIndex = 0;

    // transform ssbo data
    storageVectors[1].resize(xdvk::transformSize(DIMENSION));
    std::copy_n(entity.transform.buffer, xdvk::transformSize(DIMENSION), &storageVectors[1][0]);
    entity.geometry.transformBufferIndex = 0;

    // ??
    storageVectors[2].resize(1);
}

#endif /* SCENE_HYPERCUBE */

#ifdef SCENE_TESSERACTS

void Vulkan::createModel() {

    const size_t n = 250;
    const float size = 0.05;

    std::vector<Vertex> vertices;
    std::vector<uint32_t> indices;
    std::vector<float> vertex_data;

    xdvk::hypercubeIndices(indices, DIMENSION);
    xdvk::hypercubeVertices(vertex_data, DIMENSION, size);

    VkBuffer indexBuffer;
    VkDeviceMemory indexBufferMemory;

    createIndexBuffer(indices, indexBuffer, indexBufferMemory);

    vertices.resize(vertex_data.size());

    // vertex ssbo data
    storageVectors[0].resize(vertex_data.size());
```

```cpp
    std::copy_n(vertex_data.data(), vertex_data.size(), &storageVectors[0][0]);

    // transform ssbo data
    storageVectors[1].resize(n * xdvk::transformSize(DIMENSION));

    for (size_t i = 0; i < n; i++) {
        const auto entity_id = scene.add();
        auto &entity = scene.get(entity_id);

        entity.geometry.vertices.resize(vertices.size());

        float x_range = 4.0;
        float y_range = 3.0;

        entity.transform.position[0] = x_range * static_cast<float>(rand()) / static_cast<float>(RAND_MAX) - (x_range / 2.0);
        entity.transform.position[1] = y_range * static_cast<float>(rand()) / static_cast<float>(RAND_MAX) - (y_range / 2.0);

        for (size_t j = 0; j < xdvk::rotationSize(DIMENSION); j++) {
            entity.transform.rotation[j] = 2.0 * M_PI * static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        }

        Vertex vertex{ static_cast<glm::float32>(i) };
        std::fill_n(vertices.begin(), vertices.size(), vertex);

        createVertexBuffer(vertices, entity.geometry.vertexBuffer, entity.geometry.vertexBufferMemory);

        entity.geometry.indexBuffer = indexBuffer;
        entity.geometry.indexBufferMemory = indexBufferMemory;
        entity.geometry.indexBufferSize = indices.size();

        entity.geometry.vertexBufferIndex = 0;
        entity.geometry.transformBufferIndex = i * xdvk::transformSize(DIMENSION);

        std::copy_n(entity.transform.buffer, xdvk::transformSize(DIMENSION), &storageVectors[1][i * xdvk::transformSize(DIMENSION)]);
    }

    storageVectors[2].resize(1);
}

#endif /* SCENE_TESSERACTS */
```

Listing B.4: `src/main/scene.cpp`

B.1.5. `vertex.cpp`

```cpp
#include "vertex.hpp"

auto Vertex::getBindingDescription() -> VkVertexInputBindingDescription {
    VkVertexInputBindingDescription bindingDescription{};
    bindingDescription.binding = 0;
    bindingDescription.stride = sizeof(Vertex);
    bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

    return bindingDescription;
}

auto Vertex::getAttributeDescriptions() -> std::array<VkVertexInputAttributeDescription, 1> {
    std::array<VkVertexInputAttributeDescription, 1> attributeDescriptions{};

    // attributeDescriptions[0].binding = 0;
    // attributeDescriptions[0].location = 0;
    // attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
    // attributeDescriptions[0].offset = offsetof(Vertex, pos);

    // attributeDescriptions[1].binding = 0;
```

```
    // attributeDescriptions[1].location = 1;
    // attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
    // attributeDescriptions[1].offset = offsetof(Vertex, color);

    // attributeDescriptions[2].binding = 0;
    // attributeDescriptions[2].location = 2;
    // attributeDescriptions[2].format = VK_FORMAT_R32G32_SFLOAT;
    // attributeDescriptions[2].offset = offsetof(Vertex, texCoord);

    attributeDescriptions[0].binding = 0;
    attributeDescriptions[0].location = 0;
    attributeDescriptions[0].format = VK_FORMAT_R32_SFLOAT;
    attributeDescriptions[0].offset = offsetof(Vertex, entity);

    return attributeDescriptions;
}

auto Vertex::operator==(const Vertex &other) const -> bool {
    //TODO fix
    // return pos == other.pos && color == other.color && texCoord == other.texCoord;
    return entity == other.entity;
}
```

Listing B.5: `src/main/vertex.cpp`

B.1.6. `vertex.hpp`

```
#ifndef VERTEX_H
#define VERTEX_H

#include <array>

#include <vulkan/vulkan.h>

#define GLM_FORCE_RADIANS
#define GL_FORCE_DEPTH_ZERO_TO_ONE
#define GLM_ENABLE_EXPERIMENTAL
#include <glm/glm.hpp>

struct Vertex {
    glm::float32 entity = 0.0F;

    static auto getBindingDescription() -> VkVertexInputBindingDescription;
    static auto getAttributeDescriptions() -> std::array<VkVertexInputAttributeDescription, 1>;

    auto operator==(const Vertex &other) const -> bool;
};

#endif
```

Listing B.6: `src/main/vertex.hpp`

B.1.7. `vulkan.cpp`

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#define GLM_FORCE_RADIANS
#define GL_FORCE_DEPTH_ZERO_TO_ONE
#define GLM_ENABLE_EXPERIMENTAL
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

```cpp
#include <glm/gtx/hash.hpp>

#define STB_IMAGE_IMPLEMENTATION
#include "../include/stb/stb_image.h"

#ifdef PROFILER
#include <coz.h>
#else
#define COZ_PROGRESS
#endif

#include "vertex.hpp"
#include "xdvk.hpp"
#include "config.hpp"

#include <iostream>
#include <fstream>
#include <stdexcept>
#include <algorithm>
#include <chrono>
#include <vector>
#include <cstring>
#include <cstdlib>
#include <cstdint>
#include <array>
#include <optional>
#include <set>
#include <unordered_map>
#include <source_location>

#define UNUSED(expr) do { (void)(expr); } while (0)

const std::string TEXTURE_PATH = "./assets/texture.png";
const std::string SHADER_DIRECTORY = "./shaders/";
const std::string VERT_SHADER = "shader.vert.spv";
const std::string FRAG_SHADER = "shader.frag.spv";

const size_t MAX_FRAMES_IN_FLIGHT = 2;

const uint32_t DIMENSION = 4;
const std::vector<size_t> SSBO_RESERVE_SIZE = {
    (1L << 32),
    (1L << 32),
    255,
};

const std::vector<const char*> VALIDATION_LAYERS = {
    "VK_LAYER_KHRONOS_validation",
    // "VK_LAYER_MESA_overlay",
};

const std::vector<const char*> DEVICE_EXTENSIONS = {
    VK_KHR_SWAPCHAIN_EXTENSION_NAME
};

#ifdef NDEBUG
const bool enableValidationLayers = false;
#else
const bool enableValidationLayers = true;
#endif

auto CreateDebugUtilsMessengerEXT(
    VkInstance instance,
    const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDebugUtilsMessengerEXT* pDebugMessenger
) -> VkResult {
    auto func = reinterpret_cast<PFN_vkCreateDebugUtilsMessengerEXT>(vkGetInstanceProcAddr(instance, "
     vkCreateDebugUtilsMessengerEXT"));
    if (func != nullptr) {
        return func(instance, pCreateInfo, pAllocator, pDebugMessenger);
    }
    return VK_ERROR_EXTENSION_NOT_PRESENT;
}

```

```cpp
void DestroyDebugUtilsMessengerEXT(VkInstance instance, VkDebugUtilsMessengerEXT debugMessenger, const
    VkAllocationCallbacks* pAllocator) {
    auto func = reinterpret_cast<PFN_vkDestroyDebugUtilsMessengerEXT>(vkGetInstanceProcAddr(instance, "
    vkDestroyDebugUtilsMessengerEXT"));
    if (func != nullptr) {
        func(instance, debugMessenger, pAllocator);
    }
}

struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
    std::optional<uint32_t> presentFamily;

    auto isComplete() -> bool {
        return graphicsFamily.has_value() && presentFamily.has_value();
    }
};

struct SwapChainSupportDetails {
    VkSurfaceCapabilitiesKHR capabilities;
    std::vector<VkSurfaceFormatKHR> formats;
    std::vector<VkPresentModeKHR> presentModes;
};

/*
namespace std {
    template<> struct hash<Vertex> {
        size_t operator()(Vertex const &vertex) const {
            return ((hash<glm::vec3>()(vertex.pos) ^
                    (hash<glm::vec3>()(vertex.color) << 1)) >> 1) ^
                    (hash<glm::vec2>()(vertex.texCoord) << 1);
        }
    };
}
*/

#define VK_CHECK_RESULT(r)                                      \
{                                                               \
    VkResult result = (r);                                      \
    if (result != VK_SUCCESS) {                                 \
        std::source_location location =                         \
            std::source_location::current();                    \
        std::stringstream ss;                                   \
        ss << "ERROR: VkResult is " << result                   \
            << " at " <<  location.function_name()              \
            << "() at " << location.file_name()                 \
            << ":" << location.line()                           \
            << ":" << location.column();                        \
        throw std::runtime_error(ss.str());                     \
    }                                                           \
}

struct UniformBufferObject {
    alignas(4) glm::vec2 res;
    alignas(4) glm::float32 time;
};

struct PushConstants {
    glm::float32 entity;
    glm::float32 vertexIndex;
    glm::float32 indexIndex;
    glm::float32 transformIndex;
};

class Vulkan {
public:
    void run() {
        initWindow();
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    size_t width = config.default_width;
```

```
    size_t height = config.default_height;

    GLFWwindow* window;

    VkInstance instance;
    VkDebugUtilsMessengerEXT debugMessenger;
    VkSurfaceKHR surface;

    VkDevice device;
    VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
    VkPhysicalDeviceProperties properties = {};

    VkQueue graphicsQueue;
    VkQueue presentQueue;

    VkSwapchainKHR swapChain;
    std::vector<VkImage> swapChainImages;
    VkFormat swapChainImageFormat;
    VkExtent2D swapChainExtent;
    std::vector<VkImageView> swapChainImageViews;
    std::vector<VkFramebuffer> swapChainFramebuffers;

    VkRenderPass renderPass;
    VkDescriptorSetLayout descriptorSetLayout;
    VkPipelineLayout pipelineLayout;
    VkPipeline graphicsPipeline;

    VkCommandPool commandPool;

    VkImage depthImage;
    VkDeviceMemory depthImageMemory;
    VkImageView depthImageView;

    VkImage textureImage;
    VkDeviceMemory textureImageMemory;
    VkImageView textureImageView;
    VkSampler textureSampler;

    std::vector<Vertex> vertexVector;
    // std::vector<uint32_t> indexVector;
    // VkBuffer vertexBuffer;
    // VkDeviceMemory vertexBufferMemory;
    // VkBuffer indexBuffer;
    // VkDeviceMemory indexBufferMemory;

    std::vector<std::vector<VkBuffer>> uniformBuffers;
    std::vector<std::vector<VkDeviceMemory>> uniformBuffersMemory;
    size_t uniformBufferCount = 1;

    std::vector<std::vector<float>> storageVectors;
    std::vector<std::vector<VkBuffer>> storageBuffers;
    std::vector<std::vector<VkDeviceMemory>> storageBuffersMemory;
    size_t storageBufferCount = 3;

    VkDescriptorPool descriptorPool;
    std::vector<VkDescriptorSet> descriptorSets;

    std::vector<VkCommandBuffer> commandBuffers;

    std::vector<VkSemaphore> imageAvailableSemaphores;
    std::vector<VkSemaphore> renderFinishedSemaphores;
    std::vector<VkFence> inFlightFences;
    std::vector<VkFence> imagesInFlight;
    size_t currentFrame = 0;

    xdvk::Scene<DIMENSION> scene = xdvk::Scene<DIMENSION>(256);

    bool framebufferResized = false;

    float time;
    float deltaTime = 0;
    std::chrono::time_point<std::chrono::high_resolution_clock> now = std::chrono::high_resolution_clock
      ::now();
    std::chrono::time_point<std::chrono::high_resolution_clock> lastTime = std::chrono::
      high_resolution_clock::now();
```

```

    void initWindow() {
        glfwInit();

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

        window = glfwCreateWindow(static_cast<int>(width), static_cast<int>(height), "vulkan-xd",
  nullptr, nullptr);
        glfwSetWindowUserPointer(window, this);
        glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
    }

    static void framebufferResizeCallback(GLFWwindow *window, int width, int height) {
        auto *app = reinterpret_cast<Vulkan*>(glfwGetWindowUserPointer(window));
        app->width = width;
        app->height = height;
        app->framebufferResized = true;
    }

    void initVulkan() {
        createInstance();
        setupDebugMessenger();
        createSurface();
        pickPhysicalDevice();
        createLogicalDevice();
        createSwapChain();
        createImageViews();
        createRenderPass();
        createDescriptorSetLayout();
        createGraphicsPipeline();
        createCommandPool();
        createDepthResources();
        createFramebuffers();
        createTextureImage();
        createTextureImageView();
        createTextureSampler();
        createStorageVectors();
        createModel();
        // loadModel();
        // createVertexBuffer();
        // createIndexBuffer();
        createUniformBuffers();
        createStorageBuffers();
        createDescriptorPool();
        createDescriptorSets();
        createCommandBuffers();
        createSyncObjects();
    }

    void mainLoop() {
        while (!static_cast<bool>(glfwWindowShouldClose(window))) {
            glfwPollEvents();
            drawFrame();
        }

        vkDeviceWaitIdle(device);
    }

    void cleanupSwapChain() {
        vkDestroyImageView(device, depthImageView, nullptr);
        vkDestroyImage(device, depthImage, nullptr);
        vkFreeMemory(device, depthImageMemory, nullptr);

        for (auto framebuffer : swapChainFramebuffers) {
            vkDestroyFramebuffer(device, framebuffer, nullptr);
        }

        vkFreeCommandBuffers(device, commandPool, static_cast<uint32_t>(commandBuffers.size()),
  commandBuffers.data());

        vkDestroyPipeline(device, graphicsPipeline, nullptr);
        vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
        vkDestroyRenderPass(device, renderPass, nullptr);

        for (auto imageView : swapChainImageViews) {
```

```
            vkDestroyImageView(device, imageView, nullptr);
        }

        vkDestroySwapchainKHR(device, swapChain, nullptr);

        for (size_t i = 0; i < swapChainImages.size(); i++) {
            for (size_t j = 0; j < uniformBuffers.size(); j++) {
                vkDestroyBuffer(device, uniformBuffers[j][i], nullptr);
                vkFreeMemory(device, uniformBuffersMemory[j][i], nullptr);
            }
            for (size_t j = 0; j < storageBuffers.size(); j++) {
                vkDestroyBuffer(device, storageBuffers[j][i], nullptr);
                vkFreeMemory(device, storageBuffersMemory[j][i], nullptr);
            }
        }

        vkDestroyDescriptorPool(device, descriptorPool, nullptr);
    }

    void cleanup() {
        cleanupSwapChain();

        vkDestroySampler(device, textureSampler, nullptr);
        vkDestroyImageView(device, textureImageView, nullptr);

        vkDestroyImage(device, textureImage, nullptr);
        vkFreeMemory(device, textureImageMemory, nullptr);

        vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);

        /*
        vkDestroyBuffer(device, indexBuffer, nullptr);
        vkFreeMemory(device, indexBufferMemory, nullptr);

        vkDestroyBuffer(device, vertexBuffer, nullptr);
        vkFreeMemory(device, vertexBufferMemory, nullptr);
        */

        //TODO fix scene cleanup
        for (auto entity : scene.entities) {
            vkDestroyBuffer(device, entity.geometry.vertexBuffer, nullptr);
            vkFreeMemory(device, entity.geometry.vertexBufferMemory, nullptr);
            vkDestroyBuffer(device, entity.geometry.indexBuffer, nullptr);
            vkFreeMemory(device, entity.geometry.indexBufferMemory, nullptr);
        }

        for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
            vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
            vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
            vkDestroyFence(device, inFlightFences[i], nullptr);
        }

        vkDestroyCommandPool(device, commandPool, nullptr);

        vkDestroyDevice(device, nullptr);

        if (enableValidationLayers) {
            DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
        }

        vkDestroySurfaceKHR(instance, surface, nullptr);
        vkDestroyInstance(instance, nullptr);

        glfwDestroyWindow(window);

        glfwTerminate();
    }

    void recreateSwapChain() {
        int width = 0;
        int height = 0;
        glfwGetFramebufferSize(window, &width, &height);
        while (width == 0 || height == 0) {
            glfwGetFramebufferSize(window, &width, &height);
            glfwWaitEvents();
```

```
        }

        vkDeviceWaitIdle(device);

        cleanupSwapChain();

        createSwapChain();
        createImageViews();
        createRenderPass();
        createGraphicsPipeline();
        createDepthResources();
        createFramebuffers();
        createUniformBuffers();
        createStorageBuffers();
        createDescriptorPool();
        createDescriptorSets();
        createCommandBuffers();

        imagesInFlight.resize(swapChainImages.size(), VK_NULL_HANDLE);
    }

    void createInstance() {
        if (enableValidationLayers && !checkValidationLayerSupport(VALIDATION_LAYERS)) {
            throw std::runtime_error("validation layers requested, but not available!");
        }

        VkApplicationInfo appInfo{};
        appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
        appInfo.pApplicationName = "Hello Triangle";
        appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
        appInfo.pEngineName = "No Engine";
        appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
        appInfo.apiVersion = VK_API_VERSION_1_0;

        VkInstanceCreateInfo createInfo{};
        createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
        createInfo.pApplicationInfo = &appInfo;

        auto extensions = getRequiredExtensions();
        createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
        createInfo.ppEnabledExtensionNames = extensions.data();

        VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo{};
        if (enableValidationLayers) {
            createInfo.enabledLayerCount = static_cast<uint32_t>(VALIDATION_LAYERS.size());
            createInfo.ppEnabledLayerNames = VALIDATION_LAYERS.data();

            populateDebugMessengerCreateInfo(debugCreateInfo);
            createInfo.pNext = &debugCreateInfo;
        } else {
            createInfo.enabledLayerCount = 0;

            createInfo.pNext = nullptr;
        }

        VK_CHECK_RESULT(vkCreateInstance(&createInfo, nullptr, &instance))
    }

    void populateDebugMessengerCreateInfo(VkDebugUtilsMessengerCreateInfoEXT& createInfo) {
        createInfo = {};
        createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
        createInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
        createInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
        createInfo.pfnUserCallback = debugCallback;
    }

    void setupDebugMessenger() {
        if (!enableValidationLayers) return;

        VkDebugUtilsMessengerCreateInfoEXT createInfo;
        populateDebugMessengerCreateInfo(createInfo);

        VK_CHECK_RESULT(CreateDebugUtilsMessengerEXT(instance, &createInfo, nullptr, &debugMessenger))
```

```
    }

    void createSurface() {
        VK_CHECK_RESULT(glfwCreateWindowSurface(instance, window, nullptr, &surface))
    }

    void pickPhysicalDevice() {
        uint32_t deviceCount = 0;
        vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);

        if (deviceCount == 0) {
            throw std::runtime_error("failed to find GPUs with Vulkan support!");
        }

        std::vector<VkPhysicalDevice> devices(deviceCount);
        vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());

        for (const auto& device : devices) {
            if (isDeviceSuitable(device)) {
                physicalDevice = device;
                break;
            }
        }

        std::cout << "devices:" << std::endl;
        VkPhysicalDevice dev;
        VkPhysicalDeviceProperties props{};
        for (const auto &device : devices) {
            dev = device;
            vkGetPhysicalDeviceProperties(dev, &props);
            std::cout << props.deviceName << std::endl;
        }

        if (physicalDevice == VK_NULL_HANDLE) {
            throw std::runtime_error("failed to find a suitable GPU!");
        }

        vkGetPhysicalDeviceProperties(physicalDevice, &properties);
        std::cout << "using gpu: " << properties.deviceName << std::endl;
    }

    void createLogicalDevice() {
        QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

        std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
        std::set<uint32_t> uniqueQueueFamilies = {indices.graphicsFamily.value(), indices.presentFamily.value()};

        float queuePriority = 1.0F;
        for (uint32_t queueFamily : uniqueQueueFamilies) {
            VkDeviceQueueCreateInfo queueCreateInfo{};
            queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
            queueCreateInfo.queueFamilyIndex = queueFamily;
            queueCreateInfo.queueCount = 1;
            queueCreateInfo.pQueuePriorities = &queuePriority;
            queueCreateInfos.push_back(queueCreateInfo);
        }

        VkPhysicalDeviceFeatures deviceFeatures{};
        deviceFeatures.samplerAnisotropy = VK_TRUE;

        VkDeviceCreateInfo createInfo{};
        createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;

        createInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
        createInfo.pQueueCreateInfos = queueCreateInfos.data();

        createInfo.pEnabledFeatures = &deviceFeatures;

        createInfo.enabledExtensionCount = static_cast<uint32_t>(DEVICE_EXTENSIONS.size());
        createInfo.ppEnabledExtensionNames = DEVICE_EXTENSIONS.data();

        if (enableValidationLayers) {
            createInfo.enabledLayerCount = static_cast<uint32_t>(VALIDATION_LAYERS.size());
            createInfo.ppEnabledLayerNames = VALIDATION_LAYERS.data();
```

```
        } else {
            createInfo.enabledLayerCount = 0;
        }


        VK_CHECK_RESULT(vkCreateDevice(physicalDevice, &createInfo, nullptr, &device));

        vkGetDeviceQueue(device, indices.graphicsFamily.value(), 0, &graphicsQueue);
        vkGetDeviceQueue(device, indices.presentFamily.value(), 0, &presentQueue);
    }

    void createSwapChain() {
        SwapChainSupportDetails swapChainSupport = querySwapChainSupport(physicalDevice);

        VkSurfaceFormatKHR surfaceFormat = chooseSwapSurfaceFormat(swapChainSupport.formats);
        VkPresentModeKHR presentMode = chooseSwapPresentMode(swapChainSupport.presentModes);
        VkExtent2D extent = chooseSwapExtent(swapChainSupport.capabilities);

        uint32_t imageCount = swapChainSupport.capabilities.minImageCount + 1;
        if (swapChainSupport.capabilities.maxImageCount > 0 && imageCount > swapChainSupport.capabilities.maxImageCount) {
            imageCount = swapChainSupport.capabilities.maxImageCount;
        }

        VkSwapchainCreateInfoKHR createInfo{};
        createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
        createInfo.surface = surface;

        createInfo.minImageCount = imageCount;
        createInfo.imageFormat = surfaceFormat.format;
        createInfo.imageColorSpace = surfaceFormat.colorSpace;
        createInfo.imageExtent = extent;
        createInfo.imageArrayLayers = 1;
        createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;

        QueueFamilyIndices indices = findQueueFamilies(physicalDevice);
        std::array<uint32_t, 2> queueFamilyIndices = {indices.graphicsFamily.value(), indices.presentFamily.value()};

        if (indices.graphicsFamily != indices.presentFamily) {
            createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
            createInfo.queueFamilyIndexCount = 2;
            createInfo.pQueueFamilyIndices = queueFamilyIndices.data();
        } else {
            createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
        }

        createInfo.preTransform = swapChainSupport.capabilities.currentTransform;
        createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
        createInfo.presentMode = presentMode;
        createInfo.clipped = VK_TRUE;

        VK_CHECK_RESULT(vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapChain))

        vkGetSwapchainImagesKHR(device, swapChain, &imageCount, nullptr);
        swapChainImages.resize(imageCount);
        vkGetSwapchainImagesKHR(device, swapChain, &imageCount, swapChainImages.data());

        swapChainImageFormat = surfaceFormat.format;
        swapChainExtent = extent;
    }

    void createImageViews() {
        swapChainImageViews.resize(swapChainImages.size());

        for (size_t i = 0; i < swapChainImages.size(); i++) {
            swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat, VK_IMAGE_ASPECT_COLOR_BIT);
        }
    }

    void createRenderPass() {
        VkAttachmentDescription colorAttachment{};
        colorAttachment.format = swapChainImageFormat;
        colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
```

```
        colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
        colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
        colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

        VkAttachmentDescription depthAttachment{};
        depthAttachment.format = findDepthFormat();
        depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
        depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
        depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

        VkAttachmentReference colorAttachmentRef{};
        colorAttachmentRef.attachment = 0;
        colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

        VkAttachmentReference depthAttachmentRef{};
        depthAttachmentRef.attachment = 1;
        depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

        VkSubpassDescription subpass{};
        subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
        subpass.colorAttachmentCount = 1;
        subpass.pColorAttachments = &colorAttachmentRef;
        subpass.pDepthStencilAttachment = &depthAttachmentRef;

        VkSubpassDependency dependency{};
        dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
        dependency.dstSubpass = 0;
        dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
        dependency.srcAccessMask = 0;
        dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
        dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;

        std::array<VkAttachmentDescription, 2> attachments = {colorAttachment, depthAttachment};
        VkRenderPassCreateInfo renderPassInfo{};
        renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
        renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
        renderPassInfo.pAttachments = attachments.data();
        renderPassInfo.subpassCount = 1;
        renderPassInfo.pSubpasses = &subpass;
        renderPassInfo.dependencyCount = 1;
        renderPassInfo.pDependencies = &dependency;

        VK_CHECK_RESULT(vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass))
    }

    void createDescriptorSetLayout() {
        //TODO make dynamic

        VkDescriptorSetLayoutBinding uboLayoutBinding{};
        uboLayoutBinding.binding = 0;
        uboLayoutBinding.descriptorCount = 1;
        uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        uboLayoutBinding.pImmutableSamplers = nullptr;
        uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

        VkDescriptorSetLayoutBinding samplerLayoutBinding;
        samplerLayoutBinding.binding = 1;
        samplerLayoutBinding.descriptorCount = 1;
        samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
        samplerLayoutBinding.pImmutableSamplers = nullptr;
        samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

        // shader storage buffer object bindings

        std::array<VkDescriptorSetLayoutBinding, 3> ssboLayoutBindings;
```

```
        ssboLayoutBindings[0].binding = 2;
        ssboLayoutBindings[0].descriptorCount = 1;
        ssboLayoutBindings[0].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        ssboLayoutBindings[0].pImmutableSamplers = nullptr;
        ssboLayoutBindings[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

        ssboLayoutBindings[1].binding = 3;
        ssboLayoutBindings[1].descriptorCount = 1;
        ssboLayoutBindings[1].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        ssboLayoutBindings[1].pImmutableSamplers = nullptr;
        ssboLayoutBindings[1].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

        ssboLayoutBindings[2].binding = 4;
        ssboLayoutBindings[2].descriptorCount = 1;
        ssboLayoutBindings[2].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        ssboLayoutBindings[2].pImmutableSamplers = nullptr;
        ssboLayoutBindings[2].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

        const size_t bindingCount = 5;
        std::array<VkDescriptorSetLayoutBinding, bindingCount> bindings = {uboLayoutBinding,
     samplerLayoutBinding, ssboLayoutBindings[0], ssboLayoutBindings[1], ssboLayoutBindings[2]};
        VkDescriptorSetLayoutCreateInfo layoutInfo{};
        layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
        layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
        layoutInfo.pBindings = bindings.data();

        VK_CHECK_RESULT(vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout))
    }

    void createGraphicsPipeline() {
        VkShaderModule vertShaderModule = createShaderModuleFromPath(VERT_SHADER);
        VkShaderModule fragShaderModule = createShaderModuleFromPath(FRAG_SHADER);

        //TODO dynamic specialisation maps
        std::array<VkSpecializationMapEntry, 1> specializationMapEntries{};
        specializationMapEntries[0].constantID = 0;
        specializationMapEntries[0].size = sizeof(uint32_t);
        specializationMapEntries[0].offset = 0;

        VkSpecializationInfo specializationInfo{};
        specializationInfo.dataSize = sizeof(uint32_t);
        specializationInfo.mapEntryCount = static_cast<uint32_t>(specializationMapEntries.size());
        specializationInfo.pMapEntries = specializationMapEntries.data();
        specializationInfo.pData = &DIMENSION;

        VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
        vertShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
        vertShaderStageInfo.module = vertShaderModule;
        vertShaderStageInfo.pName = "main";
        vertShaderStageInfo.pSpecializationInfo = &specializationInfo;

        VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
        fragShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
        fragShaderStageInfo.module = fragShaderModule;
        fragShaderStageInfo.pName = "main";

        std::array<VkPipelineShaderStageCreateInfo, 2> shaderStages = {vertShaderStageInfo,
     fragShaderStageInfo};

        VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
        vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

        auto bindingDescription = Vertex::getBindingDescription();
        auto attributeDescriptions = Vertex::getAttributeDescriptions();

        vertexInputInfo.vertexBindingDescriptionCount = 1;
        vertexInputInfo.vertexAttributeDescriptionCount = static_cast<uint32_t>(attributeDescriptions.
     size());
        vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
        vertexInputInfo.pVertexAttributeDescriptions = attributeDescriptions.data();

        VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
        inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
```

```
        // inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
        inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_LINE_LIST;
        inputAssembly.primitiveRestartEnable = VK_FALSE;

        VkViewport viewport{};
        viewport.x = 0.0F;
        viewport.y = 0.0F;
        viewport.width = static_cast<float>(swapChainExtent.width);
        viewport.height = static_cast<float>(swapChainExtent.height);
        viewport.minDepth = 0.0F;
        viewport.maxDepth = 1.0F;

        VkRect2D scissor{};
        scissor.offset = {0, 0};
        scissor.extent = swapChainExtent;

        VkPipelineViewportStateCreateInfo viewportState{};
        viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
        viewportState.viewportCount = 1;
        viewportState.pViewports = &viewport;
        viewportState.scissorCount = 1;
        viewportState.pScissors = &scissor;

        VkPipelineRasterizationStateCreateInfo rasterizer{};
        rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
        rasterizer.depthClampEnable = VK_FALSE;
        rasterizer.rasterizerDiscardEnable = VK_FALSE;
        rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
        rasterizer.lineWidth = 1.0F;
        rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
        rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
        rasterizer.depthBiasEnable = VK_FALSE;

        VkPipelineMultisampleStateCreateInfo multisampling{};
        multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
        multisampling.sampleShadingEnable = VK_FALSE;
        multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;

        VkPipelineDepthStencilStateCreateInfo depthStencil{};
        depthStencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
        depthStencil.depthTestEnable = VK_TRUE;
        depthStencil.depthWriteEnable = VK_TRUE;
        depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
        depthStencil.depthBoundsTestEnable = VK_FALSE;
        depthStencil.stencilTestEnable = VK_FALSE;

        VkPipelineColorBlendAttachmentState colorBlendAttachment{};
        colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
    VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
        colorBlendAttachment.blendEnable = VK_FALSE;

        VkPipelineColorBlendStateCreateInfo colorBlending{};
        colorBlending.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
        colorBlending.logicOpEnable = VK_FALSE;
        colorBlending.logicOp = VK_LOGIC_OP_COPY;
        colorBlending.attachmentCount = 1;
        colorBlending.pAttachments = &colorBlendAttachment;
        colorBlending.blendConstants[0] = 0.0F;
        colorBlending.blendConstants[1] = 0.0F;
        colorBlending.blendConstants[2] = 0.0F;
        colorBlending.blendConstants[3] = 0.0F;

        VkPushConstantRange pushConstant{};
        pushConstant.offset = 0;
        pushConstant.size = sizeof(PushConstants);
        pushConstant.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

        VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
        pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
        pipelineLayoutInfo.setLayoutCount = 1;
        pipelineLayoutInfo.pSetLayouts = &descriptorSetLayout;
        pipelineLayoutInfo.pushConstantRangeCount = 1;
        pipelineLayoutInfo.pPushConstantRanges = &pushConstant;

```

```
        VK_CHECK_RESULT(vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &pipelineLayout))

        VkGraphicsPipelineCreateInfo pipelineInfo{};
        pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
        pipelineInfo.stageCount = 2;
        pipelineInfo.pStages = shaderStages.data();
        pipelineInfo.pVertexInputState = &vertexInputInfo;
        pipelineInfo.pInputAssemblyState = &inputAssembly;
        pipelineInfo.pViewportState = &viewportState;
        pipelineInfo.pRasterizationState = &rasterizer;
        pipelineInfo.pMultisampleState = &multisampling;
        pipelineInfo.pDepthStencilState = &depthStencil;
        pipelineInfo.pColorBlendState = &colorBlending;
        pipelineInfo.layout = pipelineLayout;
        pipelineInfo.renderPass = renderPass;
        pipelineInfo.subpass = 0;
        pipelineInfo.basePipelineHandle = VK_NULL_HANDLE;

        VK_CHECK_RESULT(vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &graphicsPipeline))

        vkDestroyShaderModule(device, fragShaderModule, nullptr);
        vkDestroyShaderModule(device, vertShaderModule, nullptr);
    }

    void createFramebuffers() {
        swapChainFramebuffers.resize(swapChainImageViews.size());

        for (size_t i = 0; i < swapChainImageViews.size(); i++) {
            std::array<VkImageView, 2> attachments = {
                swapChainImageViews[i],
                depthImageView
            };

            VkFramebufferCreateInfo framebufferInfo{};
            framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
            framebufferInfo.renderPass = renderPass;
            framebufferInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
            framebufferInfo.pAttachments = attachments.data();
            framebufferInfo.width = swapChainExtent.width;
            framebufferInfo.height = swapChainExtent.height;
            framebufferInfo.layers = 1;

            VK_CHECK_RESULT(vkCreateFramebuffer(device, &framebufferInfo, nullptr, &swapChainFramebuffers[i]))
        }
    }

    void createCommandPool() {
        QueueFamilyIndices queueFamilyIndices = findQueueFamilies(physicalDevice);

        VkCommandPoolCreateInfo poolInfo{};
        poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
        poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily.value();

        VK_CHECK_RESULT(vkCreateCommandPool(device, &poolInfo, nullptr, &commandPool))
    }

    void createDepthResources() {
        VkFormat depthFormat = findDepthFormat();
        createImage(swapChainExtent.width, swapChainExtent.height, depthFormat, VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, depthImage, depthImageMemory);
        depthImageView = createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT);
    }

    auto findSupportedFormat(const std::vector<VkFormat> &candidates, VkImageTiling tiling, VkFormatFeatureFlags features) -> VkFormat {
        for (VkFormat format : candidates) {
            VkFormatProperties props;
            vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);

            if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == features) {
                return format;
```

```
            }
            else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) ==
    features) {
                return format;
            }
        }

        throw std::runtime_error("failed to find supported format!");
    }

    auto findDepthFormat() -> VkFormat {
        return findSupportedFormat(
            {VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D24_UNORM_S8_UINT},
            VK_IMAGE_TILING_OPTIMAL,
            VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
        );
    }

    static auto hasStencilComponent(VkFormat format) -> bool {
        return format == VK_FORMAT_D32_SFLOAT || format == VK_FORMAT_D32_SFLOAT_S8_UINT;
    }

    void createTextureImage() {
        int texWidth;
        int texHeight;
        int texChannels;
        stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texHeight, &texWidth, &texChannels,
    STBI_rgb_alpha);
        VkDeviceSize imageSize = static_cast<VkDeviceSize>(texWidth) * texHeight * 4;

        if (!pixels) {
            throw std::runtime_error("failed to load texture image!");
        }

        VkBuffer stagingBuffer;
        VkDeviceMemory stagingBufferMemory;

        createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
    VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);

        void *data;
        vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
            memcpy(data, pixels, static_cast<size_t>(imageSize));
        vkUnmapMemory(device, stagingBufferMemory);

        stbi_image_free(pixels);

        createImage(texWidth, texHeight, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_TILING_OPTIMAL,
    VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, textureImage, textureImageMemory);

        transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);
            copyBufferToImage(stagingBuffer, textureImage, static_cast<uint32_t>(texWidth), static_cast<
    uint32_t>(texHeight));
        transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);

        vkDestroyBuffer(device, stagingBuffer, nullptr);
        vkFreeMemory(device, stagingBufferMemory, nullptr);
    }

    void createTextureImageView() {
        textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_SRGB,
    VK_IMAGE_ASPECT_COLOR_BIT);
    }

    void createTextureSampler() {
        VkSamplerCreateInfo samplerInfo{};
        samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
        samplerInfo.magFilter = VK_FILTER_LINEAR;
        samplerInfo.minFilter = VK_FILTER_LINEAR;
        samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

```
        samplerInfo.anisotropyEnable = VK_TRUE;
        samplerInfo.maxAnisotropy = properties.limits.maxSamplerAnisotropy;
        samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
        samplerInfo.unnormalizedCoordinates = VK_FALSE;
        samplerInfo.compareEnable = VK_FALSE;
        samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
        samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
        samplerInfo.mipLodBias = 0.0F;
        samplerInfo.minLod = 0.0F;
        samplerInfo.maxLod = 0.0F;

        VK_CHECK_RESULT(vkCreateSampler(device, &samplerInfo, nullptr, &textureSampler))
    }

    auto createImageView(VkImage image, VkFormat format, VkImageAspectFlags aspectFlags) -> VkImageView {
        VkImageViewCreateInfo viewInfo{};
        viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        viewInfo.image = image;
        viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
        viewInfo.format = format;
        viewInfo.subresourceRange.aspectMask = aspectFlags;
        viewInfo.subresourceRange.baseMipLevel = 0;
        viewInfo.subresourceRange.levelCount = 1;
        viewInfo.subresourceRange.baseArrayLayer = 0;
        viewInfo.subresourceRange.layerCount = 1;

        VkImageView imageView;
        VK_CHECK_RESULT(vkCreateImageView(device, &viewInfo, nullptr, &imageView))

        return imageView;
    }

    void createImage(uint32_t width, uint32_t height, VkFormat format, VkImageTiling tiling, VkImageUsageFlags usage, VkMemoryPropertyFlags properties,
    VkImage &image, VkDeviceMemory &imageMemory) {
        VkImageCreateInfo imageInfo{};
        imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
        imageInfo.imageType = VK_IMAGE_TYPE_2D;
        imageInfo.extent.width = width;
        imageInfo.extent.height = height;
        imageInfo.extent.depth = 1;
        imageInfo.mipLevels = 1;
        imageInfo.arrayLayers = 1;
        imageInfo.format = format;
        imageInfo.tiling = tiling;
        imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        imageInfo.usage = usage;
        imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
        imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

        VK_CHECK_RESULT(vkCreateImage(device, &imageInfo, nullptr, &image))

        VkMemoryRequirements memRequireMents;
        vkGetImageMemoryRequirements(device, image, &memRequireMents);

        VkMemoryAllocateInfo allocInfo{};
        allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        allocInfo.allocationSize = memRequireMents.size;
        allocInfo.memoryTypeIndex = findMemoryType(memRequireMents.memoryTypeBits, properties);

        VK_CHECK_RESULT(vkAllocateMemory(device, &allocInfo, nullptr, &imageMemory))

        vkBindImageMemory(device, image, imageMemory, 0);
    }

    void transitionImageLayout(VkImage image, VkFormat format, VkImageLayout oldLayout, VkImageLayout newLayout) {
        VkCommandBuffer commandBuffer = beginSingleTimeCommands();

        VkImageMemoryBarrier barrier{};
        barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
        barrier.oldLayout = oldLayout;
        barrier.newLayout = newLayout;
        barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
```

```
        barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        barrier.image = image;
        barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        barrier.subresourceRange.baseMipLevel = 0;
        barrier.subresourceRange.levelCount = 1;
        barrier.subresourceRange.baseArrayLayer = 0;
        barrier.subresourceRange.layerCount = 1;
        barrier.srcAccessMask = 0;
        barrier.dstAccessMask = 0;

        VkPipelineStageFlags sourceStage;
        VkPipelineStageFlags destinationStage;

        if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL)
     {
            barrier.srcAccessMask = 0;
            barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

            sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
            destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
        }
        else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout ==
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
            barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
            barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

            sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
            destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
        }
        else {
            throw std::invalid_argument("unsupported layout transition!");
        }

        vkCmdPipelineBarrier(
            commandBuffer,
            sourceStage, destinationStage,
            0,
            0, nullptr,
            0, nullptr,
            1, &barrier
        );

        endSingleTimeCommands(commandBuffer);
    }

    void copyBufferToImage(VkBuffer buffer, VkImage image, uint32_t width, uint32_t height) {
        VkCommandBuffer commandBuffer = beginSingleTimeCommands();

        VkBufferImageCopy region{};
        region.bufferOffset = 0;
        region.bufferRowLength = 0;
        region.bufferImageHeight = 0;

        region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        region.imageSubresource.mipLevel = 0;
        region.imageSubresource.baseArrayLayer = 0;
        region.imageSubresource.layerCount = 1;

        region.imageOffset = {0, 0, 0};
        region.imageExtent = {
            width,
            height,
            1
        };

        vkCmdCopyBufferToImage(
            commandBuffer,
            buffer,
            image,
            VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
            1,
            &region
        );

        endSingleTimeCommands(commandBuffer);
```

```
    }

    void createStorageVectors() {
        storageVectors.resize(storageBufferCount);
        for (size_t i = 0; i < storageVectors.size(); i++) {
            storageVectors[i].reserve(SSBO_RESERVE_SIZE[i]);
        }
    }

    virtual void createModel();

    template<typename T>
    void createVertexBuffer(std::vector<T> &vector, VkBuffer &buffer, VkDeviceMemory &bufferMemory) {
        createVectorBuffer(vector, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
      buffer, bufferMemory);
    }

    template<typename T>
    void createIndexBuffer(std::vector<T> &vector, VkBuffer &buffer, VkDeviceMemory &bufferMemory) {
        createVectorBuffer(vector, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT,
     buffer, bufferMemory);
    }

    void createUniformBuffers() {

        uniformBuffers.resize(uniformBufferCount);
        uniformBuffersMemory.resize(uniformBufferCount);

        for (size_t i = 0; i < uniformBuffers.size(); i++) {
            VkDeviceSize bufferSize = sizeof(UniformBufferObject);

            uniformBuffers[i].resize(swapChainImages.size());
            uniformBuffersMemory[i].resize(swapChainImages.size());

            for (size_t j = 0; j < swapChainImages.size(); j++) {
                createBuffer(
                    bufferSize,
                    VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
                    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                    uniformBuffers[i][j],
                    uniformBuffersMemory[i][j]
                );
            }
        }
    }

    void createStorageBuffers() {

        storageBuffers.resize(storageBufferCount);
        storageBuffersMemory.resize(storageBufferCount);

        for (size_t i = 0; i < storageBuffers.size(); i++) {
            VkDeviceSize bufferSize = storageVectors[i].size() * sizeof(storageVectors[i][0]);

            storageBuffers[i].resize(swapChainImages.size());
            storageBuffersMemory[i].resize(swapChainImages.size());

            for (size_t j = 0; j < swapChainImages.size(); j++) {
                createBuffer(
                    bufferSize,
                    VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
                    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                    storageBuffers[i][j],
                    storageBuffersMemory[i][j]
                );
            }
        }
    }

    template<typename T>
    void createVectorBuffer(std::vector<T> &vector, VkBufferUsageFlags usage, VkBuffer &buffer,
     VkDeviceMemory &bufferMemory) {
        VkDeviceSize bufferSize = vector.size() * sizeof(vector[0]);

        VkBuffer stagingBuffer;
```

```
        VkDeviceMemory stagingBufferMemory;
        createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
      VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);

        void* data;
        vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
            memcpy(data, vector.data(), (size_t)bufferSize);
        vkUnmapMemory(device, stagingBufferMemory);

        createBuffer(bufferSize, usage, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, buffer, bufferMemory);

        copyBuffer(stagingBuffer, buffer, bufferSize);

        vkDestroyBuffer(device, stagingBuffer, nullptr);
        vkFreeMemory(device, stagingBufferMemory, nullptr);
    }

    void createDescriptorPool() {

        // std::array<VkDescriptorPoolSize, 5> poolSizes{};
        std::vector<VkDescriptorPoolSize> poolSizes;
        poolSizes.resize(5);
        poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        poolSizes[0].descriptorCount = static_cast<uint32_t>(swapChainImages.size());
        poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
        poolSizes[1].descriptorCount = static_cast<uint32_t>(swapChainImages.size());
        poolSizes[2].type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        poolSizes[2].descriptorCount = static_cast<uint32_t>(swapChainImages.size());
        poolSizes[3].type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        poolSizes[3].descriptorCount = static_cast<uint32_t>(swapChainImages.size());
        poolSizes[4].type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        poolSizes[4].descriptorCount = static_cast<uint32_t>(swapChainImages.size());

        VkDescriptorPoolCreateInfo poolInfo{};
        poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
        poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
        poolInfo.pPoolSizes = poolSizes.data();
        poolInfo.maxSets = static_cast<uint32_t>(swapChainImages.size());

        //TODO free poolSize
        VK_CHECK_RESULT(vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool))
    }

    void createDescriptorSets() {
        std::vector<VkDescriptorSetLayout> layouts(swapChainImages.size(), descriptorSetLayout);
        VkDescriptorSetAllocateInfo allocInfo{};
        allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
        allocInfo.descriptorPool = descriptorPool;
        allocInfo.descriptorSetCount = static_cast<uint32_t>(swapChainImages.size());
        allocInfo.pSetLayouts = layouts.data();

        descriptorSets.resize(swapChainImages.size());
        VK_CHECK_RESULT(vkAllocateDescriptorSets(device, &allocInfo, descriptorSets.data()))

        for (size_t i = 0; i < swapChainImages.size(); i++) {

            //TODO make dynamic

            // uniform
            VkDescriptorBufferInfo uniformBufferInfo{};
            uniformBufferInfo.buffer = uniformBuffers[0][i];
            uniformBufferInfo.offset = 0;
            uniformBufferInfo.range = sizeof(UniformBufferObject);

            // texture
            VkDescriptorImageInfo imageInfo{};
            imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
            imageInfo.imageView = textureImageView;
            imageInfo.sampler = textureSampler;

            // shader storage
            std::array<VkDescriptorBufferInfo, 3> storageBufferInfo{};
            storageBufferInfo[0].buffer = storageBuffers[0][i];
            storageBufferInfo[0].offset = 0;
            storageBufferInfo[0].range = storageVectors[0].size() * sizeof(storageVectors[0][0]);
```

```

            storageBufferInfo[1].buffer = storageBuffers[1][i];
            storageBufferInfo[1].offset = 0;
            storageBufferInfo[1].range = storageVectors[1].size() * sizeof(storageVectors[1][0]);

            storageBufferInfo[2].buffer = storageBuffers[2][i];
            storageBufferInfo[2].offset = 0;
            storageBufferInfo[2].range = storageVectors[2].size() * sizeof(storageVectors[2][0]);

            std::array<VkWriteDescriptorSet, 5> descriptorWrites{};
            descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
            descriptorWrites[0].dstSet = descriptorSets[i];
            descriptorWrites[0].dstBinding = 0;
            descriptorWrites[0].dstArrayElement = 0;
            descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
            descriptorWrites[0].descriptorCount = 1;
            descriptorWrites[0].pBufferInfo = &uniformBufferInfo;

            descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
            descriptorWrites[1].dstSet = descriptorSets[i];
            descriptorWrites[1].dstBinding = 1;
            descriptorWrites[1].dstArrayElement = 0;
            descriptorWrites[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
            descriptorWrites[1].descriptorCount = 1;
            descriptorWrites[1].pImageInfo = &imageInfo;

            descriptorWrites[2].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
            descriptorWrites[2].dstSet = descriptorSets[i];
            descriptorWrites[2].dstBinding = 2;
            descriptorWrites[2].dstArrayElement = 0;
            descriptorWrites[2].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
            descriptorWrites[2].descriptorCount = 1;
            descriptorWrites[2].pBufferInfo = &storageBufferInfo[0];

            descriptorWrites[3].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
            descriptorWrites[3].dstSet = descriptorSets[i];
            descriptorWrites[3].dstBinding = 3;
            descriptorWrites[3].dstArrayElement = 0;
            descriptorWrites[3].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
            descriptorWrites[3].descriptorCount = 1;
            descriptorWrites[3].pBufferInfo = &storageBufferInfo[1];

            descriptorWrites[4].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
            descriptorWrites[4].dstSet = descriptorSets[i];
            descriptorWrites[4].dstBinding = 4;
            descriptorWrites[4].dstArrayElement = 0;
            descriptorWrites[4].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
            descriptorWrites[4].descriptorCount = 1;
            descriptorWrites[4].pBufferInfo = &storageBufferInfo[2];

            vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()),
      descriptorWrites.data(), 0, nullptr);
        }
    }

    void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags properties,
     VkBuffer &buffer, VkDeviceMemory &bufferMemory) {
        VkBufferCreateInfo bufferInfo{};
        bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
        bufferInfo.size = size;
        bufferInfo.usage = usage;
        bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

        VK_CHECK_RESULT(vkCreateBuffer(device, &bufferInfo, nullptr, &buffer))

        VkMemoryRequirements memRequirements;
        vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

        VkMemoryAllocateInfo allocInfo{};
        allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        allocInfo.allocationSize = memRequirements.size;
        allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

        VK_CHECK_RESULT(vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory))

```

```
        vkBindBufferMemory(device, buffer, bufferMemory, 0);
    }

    auto beginSingleTimeCommands() -> VkCommandBuffer {
        VkCommandBufferAllocateInfo allocInfo{};
        allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        allocInfo.commandPool = commandPool;
        allocInfo.commandBufferCount = 1;

        VkCommandBuffer commandBuffer;
        vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);

        VkCommandBufferBeginInfo beginInfo{};
        beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
        beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

        vkBeginCommandBuffer(commandBuffer, &beginInfo);

        return commandBuffer;
    }

    void endSingleTimeCommands(VkCommandBuffer commandBuffer) {
        vkEndCommandBuffer(commandBuffer);

        VkSubmitInfo submitInfo{};
        submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
        submitInfo.commandBufferCount = 1;
        submitInfo.pCommandBuffers = &commandBuffer;

        vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
        vkQueueWaitIdle(graphicsQueue);

        vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
    }

    void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
        VkCommandBuffer commandBuffer = beginSingleTimeCommands();

        VkBufferCopy copyRegion{};
        copyRegion.size = size;
        vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);

        endSingleTimeCommands(commandBuffer);
    }

    auto findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags properties) -> uint32_t {
        VkPhysicalDeviceMemoryProperties memProperties;
        vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);

        for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
            if (
                static_cast<bool>(typeFilter & (1 << i)) &&
                (memProperties.memoryTypes[i].propertyFlags & properties) == properties
            ) {
                return i;
            }
        }

        throw std::runtime_error("failed to find suitable memory type!");
    }

    void createCommandBuffers() {
        commandBuffers.resize(swapChainFramebuffers.size());

        VkCommandBufferAllocateInfo allocInfo{};
        allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        allocInfo.commandPool = commandPool;
        allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        allocInfo.commandBufferCount = (uint32_t) commandBuffers.size();

        VK_CHECK_RESULT(vkAllocateCommandBuffers(device, &allocInfo, commandBuffers.data()))

        for (size_t i = 0; i < commandBuffers.size(); i++) {
            VkCommandBufferBeginInfo beginInfo{};
```

```cpp
            beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

            VK_CHECK_RESULT(vkBeginCommandBuffer(commandBuffers[i], &beginInfo))

            VkRenderPassBeginInfo renderPassInfo{};
            renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
            renderPassInfo.renderPass = renderPass;
            renderPassInfo.framebuffer = swapChainFramebuffers[i];
            renderPassInfo.renderArea.offset = {0, 0};
            renderPassInfo.renderArea.extent = swapChainExtent;

            std::array<VkClearValue, 2> clearValues;
            clearValues[0].color = {{0.0f, 0.0f, 0.0f, 1.0f}};
            clearValues[1].depthStencil = {1.0f, 0};

            renderPassInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());
            renderPassInfo.pClearValues = clearValues.data();

            vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

                vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

                // vkCmdSetLineWidth(commandBuffers[i], 5.0);

                VkDeviceSize offsets[] = {0};

                for (size_t j = 0; j < scene.entities.size(); j++) {
                    xdvk::Entity entity = scene.entities[j];

                    PushConstants constants{};
                    constants.entity = static_cast<glm::float32>(j);
                    constants.vertexIndex = static_cast<glm::float32>(entity.geometry.vertexBufferIndex);
                    constants.indexIndex = static_cast<glm::float32>(entity.geometry.indexBufferIndex);
                    constants.transformIndex = static_cast<glm::float32>(entity.geometry.transformBufferIndex);

                    vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, &entity.geometry.vertexBuffer, offsets);
                    vkCmdBindDescriptorSets(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineLayout, 0, 1, &descriptorSets[i], 0, nullptr);
                    vkCmdPushConstants(commandBuffers[i], pipelineLayout, VK_SHADER_STAGE_VERTEX_BIT, 0, sizeof(PushConstants), &constants);

                    if (entity.geometry.drawIndexed) {
                        vkCmdBindIndexBuffer(commandBuffers[i], entity.geometry.indexBuffer, 0, VK_INDEX_TYPE_UINT32);
                        vkCmdDrawIndexed(commandBuffers[i], entity.geometry.indexBufferSize, 1, 0, 0, 0);
                    }
                    else {
                        vkCmdDraw(commandBuffers[i], entity.geometry.vertices.size(), 1, 0, 0);
                    }
                }

            vkCmdEndRenderPass(commandBuffers[i]);

            VK_CHECK_RESULT(vkEndCommandBuffer(commandBuffers[i]))
        }
    }

    void createSyncObjects() {
        imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
        renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
        inFlightFences.resize(MAX_FRAMES_IN_FLIGHT);
        imagesInFlight.resize(swapChainImages.size(), VK_NULL_HANDLE);

        VkSemaphoreCreateInfo semaphoreInfo{};
        semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

        VkFenceCreateInfo fenceInfo{};
        fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
        fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

        for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
```

```
            VK_CHECK_RESULT(vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphores[i]))
            VK_CHECK_RESULT(vkCreateSemaphore(device, &semaphoreInfo, nullptr, &renderFinishedSemaphores[i]))
            VK_CHECK_RESULT(vkCreateFence(device, &fenceInfo, nullptr, &inFlightFences[i]))
        }
    }

    virtual void callback();

    void updateUniformBuffer(uint32_t currentImage) {
        static auto startTime = std::chrono::high_resolution_clock::now();

        auto currentTime = std::chrono::high_resolution_clock::now();
        time = std::chrono::duration<float, std::chrono::seconds::period>(currentTime - startTime).count();

        UniformBufferObject ubo{};

        ubo.res = glm::vec2(static_cast<float>(width), static_cast<float>(height));
        ubo.time = glm::float32(time);

        for (size_t i = 0; i < uniformBuffers.size(); i++) {
            void *data;
            vkMapMemory(device, uniformBuffersMemory[i][currentImage], 0, sizeof(ubo), 0, &data);
                memcpy(data, &ubo, sizeof(ubo));
            vkUnmapMemory(device, uniformBuffersMemory[i][currentImage]);
        }
    }

    void updateStorageBuffer(uint32_t currentImage) {
        for (size_t i = 0; i < storageBuffersMemory.size(); i++) {
            void* data;
            vkMapMemory(device, storageBuffersMemory[i][currentImage], 0, storageVectors[i].size() * sizeof(storageVectors[i][0]), 0, &data);
                memcpy(data, storageVectors[i].data(), storageVectors[i].size() * sizeof(storageVectors[i][0]));
            vkUnmapMemory(device, storageBuffersMemory[i][currentImage]);
        }
    }

    void drawFrame() {

        COZ_PROGRESS

        lastTime = now;
        now = std::chrono::high_resolution_clock::now();
        deltaTime = std::chrono::duration<float, std::chrono::seconds::period>(now - lastTime).count();

        vkWaitForFences(device, 1, &inFlightFences[currentFrame], VK_TRUE, UINT64_MAX);

        uint32_t imageIndex;
        VkResult result = vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);

        if (result == VK_ERROR_OUT_OF_DATE_KHR) {
            recreateSwapChain();
            return;
        }
        if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
            throw std::runtime_error("failed to acquire swap chain image!");
        }

        callback();

        updateUniformBuffer(imageIndex);
        updateStorageBuffer(imageIndex);

        if (imagesInFlight[imageIndex] != VK_NULL_HANDLE) {
            vkWaitForFences(device, 1, &imagesInFlight[imageIndex], VK_TRUE, UINT64_MAX);
        }
        imagesInFlight[imageIndex] = inFlightFences[currentFrame];

        VkSubmitInfo submitInfo{};
        submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
```

```cpp

        std::array<VkSemaphore, 1> waitSemaphores = {imageAvailableSemaphores[currentFrame]};
        std::array<VkPipelineStageFlags, 1> waitStages = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
    };
        submitInfo.waitSemaphoreCount = 1;
        submitInfo.pWaitSemaphores = waitSemaphores.data();
        submitInfo.pWaitDstStageMask = waitStages.data();

        submitInfo.commandBufferCount = 1;
        submitInfo.pCommandBuffers = &commandBuffers[imageIndex];

        std::array<VkSemaphore, 1> signalSemaphores = {renderFinishedSemaphores[currentFrame]};
        submitInfo.signalSemaphoreCount = 1;
        submitInfo.pSignalSemaphores = signalSemaphores.data();

        vkResetFences(device, 1, &inFlightFences[currentFrame]);

        VK_CHECK_RESULT(vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]))

        VkPresentInfoKHR presentInfo{};
        presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

        presentInfo.waitSemaphoreCount = 1;
        presentInfo.pWaitSemaphores = signalSemaphores.data();

        std::array<VkSwapchainKHR, 1> swapChains = {swapChain};
        presentInfo.swapchainCount = 1;
        presentInfo.pSwapchains = swapChains.data();

        presentInfo.pImageIndices = &imageIndex;

        result = vkQueuePresentKHR(presentQueue, &presentInfo);

        if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR || framebufferResized) {
            framebufferResized = false;
            recreateSwapChain();
        } else if (result != VK_SUCCESS) {
            throw std::runtime_error("failed to present swap chain image!");
        }

        currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
    }

    auto createShaderModuleFromPath(std::string path) -> VkShaderModule {
        auto shaderCode = readShaderCode(path);
        return createShaderModule(shaderCode);
    }

    auto readShaderCode(std::string path) -> std::vector<char> {
        // return readFile(fmt::format("{}{}", SHADER_DIRECTORY, path));
        return readFile(SHADER_DIRECTORY + path);
    }

    auto createShaderModule(const std::vector<char>& code) -> VkShaderModule {
        VkShaderModuleCreateInfo createInfo{};
        createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
        createInfo.codeSize = code.size();
        createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());

        VkShaderModule shaderModule;
        VK_CHECK_RESULT(vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule))

        return shaderModule;
    }

    auto chooseSwapSurfaceFormat(const std::vector<VkSurfaceFormatKHR>& availableFormats) ->
     VkSurfaceFormatKHR {
        for (const auto& availableFormat : availableFormats) {
            if (availableFormat.format == VK_FORMAT_B8G8R8A8_SRGB && availableFormat.colorSpace ==
     VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
                return availableFormat;
            }
        }

        return availableFormats[0];
```

```cpp
    }

    static auto chooseSwapPresentMode(const std::vector<VkPresentModeKHR>& availablePresentModes) ->
    VkPresentModeKHR {
        for (const auto& preferredPresentMode : config.preferredPresentModes) {
            for (const auto& availablePresentMode : availablePresentModes) {
                if (availablePresentMode == preferredPresentMode) {
                    return availablePresentMode;
                }
            }
        }

        return VK_PRESENT_MODE_FIFO_KHR;
    }

    static auto getSwapChainPresentModeName(VkPresentModeKHR presentMode) -> std::string {
        switch(presentMode) {
            case VK_PRESENT_MODE_IMMEDIATE_KHR:
                return "VK_PRESENT_MODE_IMMEDIATE_KHR";
            case VK_PRESENT_MODE_MAILBOX_KHR:
                return "VK_PRESENT_MODE_MAILBOX_KHR";
            case VK_PRESENT_MODE_FIFO_KHR:
                return "VK_PRESENT_MODE_FIFO_KHR";
            case VK_PRESENT_MODE_FIFO_RELAXED_KHR:
                return "VK_PRESENT_MODE_FIFO_RELAXED_KHR";
            case VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR:
                return "VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR";
            case VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR:
                return "VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR";
            default:
                return "unknown (" + std::to_string(presentMode) + ")";
        }
    }

    auto chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) -> VkExtent2D {
        if (capabilities.currentExtent.width != UINT32_MAX) {
            return capabilities.currentExtent;
        }
        int width = static_cast<int>(config.default_width);
        int height = static_cast<int>(config.default_height);
        glfwGetFramebufferSize(window, &width, &height);

        VkExtent2D actualExtent = {
            static_cast<uint32_t>(width),
            static_cast<uint32_t>(height)
        };

        actualExtent.width = std::clamp(actualExtent.width, capabilities.minImageExtent.width,
    capabilities.maxImageExtent.width);
        actualExtent.height = std::clamp(actualExtent.height, capabilities.minImageExtent.height,
    capabilities.maxImageExtent.height);

        return actualExtent;
    }

    auto querySwapChainSupport(VkPhysicalDevice device) -> SwapChainSupportDetails {
        SwapChainSupportDetails details;

        vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device, surface, &details.capabilities);

        uint32_t formatCount;
        vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, nullptr);

        if (formatCount != 0) {
            details.formats.resize(formatCount);
            vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, details.formats.data());
        }

        uint32_t presentModeCount;
        vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, nullptr);

        if (presentModeCount != 0) {
            details.presentModes.resize(presentModeCount);
            vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, details.
    presentModes.data());
```

```
        }

        return details;
    }

    auto isDeviceSuitable(VkPhysicalDevice device) -> bool {
        QueueFamilyIndices indices = findQueueFamilies(device);

        bool extensionsSupported = checkDeviceExtensionSupport(device, DEVICE_EXTENSIONS);

        bool swapChainAdequate = false;
        if (extensionsSupported) {
            SwapChainSupportDetails swapChainSupport = querySwapChainSupport(device);
            swapChainAdequate = !swapChainSupport.formats.empty() && !swapChainSupport.presentModes.
 empty();
        }

        VkPhysicalDeviceFeatures supportedFeatures;
        vkGetPhysicalDeviceFeatures(device, &supportedFeatures);

        return indices.isComplete() && extensionsSupported && swapChainAdequate && static_cast<bool>(
 supportedFeatures.samplerAnisotropy);
    }

    static auto checkDeviceExtensionSupport(VkPhysicalDevice device, std::vector<const char*>
 deviceExtensions) -> bool {
        uint32_t extensionCount;
        vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, nullptr);

        std::vector<VkExtensionProperties> availableExtensions(extensionCount);
        vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, availableExtensions.data
 ());

        std::set<std::string> requiredExtensions(deviceExtensions.begin(), deviceExtensions.end());

        for (const auto& extension : availableExtensions) {
            requiredExtensions.erase(extension.extensionName);
        }

        return requiredExtensions.empty();
    }

    auto findQueueFamilies(VkPhysicalDevice device) -> QueueFamilyIndices {
        QueueFamilyIndices indices;

        uint32_t queueFamilyCount = 0;
        vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);

        std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
        vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());

        uint32_t i = 0;
        for (const auto& queueFamily : queueFamilies) {
            if (queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
                indices.graphicsFamily = i;
            }

            VkBool32 presentSupport = false;
            vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &presentSupport);

            if (presentSupport) {
                indices.presentFamily = i;
            }

            if (indices.isComplete()) {
                break;
            }

            i++;
        }

        return indices;
    }

    auto getRequiredExtensions() -> std::vector<const char*> {
```

```
        uint32_t glfwExtensionCount = 0;
        const char** glfwExtensions;
        glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

        std::vector<const char*> extensions(glfwExtensions, glfwExtensions + glfwExtensionCount);

        if (enableValidationLayers) {
            extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
        }

        return extensions;
    }

    static auto checkValidationLayerSupport(std::vector<const char*> validationLayers) -> bool {
        uint32_t layerCount;
        vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

        std::vector<VkLayerProperties> availableLayers(layerCount);
        vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());

        for (const char* layerName : validationLayers) {
            bool layerFound = false;

            for (const auto& layerProperties : availableLayers) {
                if (strcmp(layerName, layerProperties.layerName) == 0) {
                    layerFound = true;
                    break;
                }
            }

            if (!layerFound) return false;
        }

        return true;
    }

    static auto readFile(const std::string& filename) -> std::vector<char> {
        std::ifstream file(filename, std::ios::ate | std::ios::binary);

        if (!file.is_open()) {
            throw std::runtime_error("failed to open file: " + filename);
        }

        auto fileSize = static_cast<size_t>(file.tellg());
        std::vector<char> buffer(fileSize);

        file.seekg(0);
        file.read(buffer.data(), fileSize);
        file.close();

        return buffer;
    }

    static VKAPI_ATTR auto VKAPI_CALL debugCallback(
        VkDebugUtilsMessageSeverityFlagBitsEXT /* messageSeverity */,
        VkDebugUtilsMessageTypeFlagsEXT /* messageType */,
        const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
        void* /* pUserData */
    ) -> VkBool32 {
        if (config.log_validation_layer) {
            std::cerr << "validation layer: " << pCallbackData->pMessage << std::endl;
        }
        return VK_FALSE;
    }
};
```

Listing B.7: `src/main/vulkan.cpp`

B.1.8. `xdvk.cpp`

```cpp
// Oliver Kovacs 2021 - xdvk - MIT

#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>
#include "vertex.hpp"
#include "xdvk.hpp"

#define N_EDGE 2

namespace xdvk {

    void hypercubeVertices(std::vector<float> &vertices, const uint32_t dimension, float size, uint32_t stride, uint32_t offset) {
        const uint32_t block = 1 + stride;
        const uint32_t n1 = pow(2, dimension);
        const uint32_t n2 = dimension;
        vertices.resize(offset + n1 * n2 * block);

        // { ±1, ..., ±1 } → dimension^2 vertices
        for (size_t i = 0; i < n1; i++) {
            for (size_t j = 0; j < n2; j++) {
                const uint32_t index = offset + (i * n2 + j) * block;
                vertices[index] = - size * (1.0 - (static_cast<uint32_t>(floor(i / pow(2.0, j))) % 2) * 2.0);
            }
        }
    }

    void hypercubeIndices(std::vector<uint32_t> &buffer, const uint32_t dimension, uint32_t stride, uint32_t offset) {
        const uint32_t block = N_EDGE + stride;
        const uint32_t n1 = dimension;
        const uint32_t n2 = pow(2, (dimension - 1));
        buffer.resize(offset + n1 * n2 * block);
        for (size_t i = 0; i < n1; i++) {
            for (size_t j = 0; j < n2; j++) {
                const uint32_t base = (j % static_cast<uint32_t>(pow(2, i))) + pow(2, (i + 1)) * floor(j / pow(2, i));
                const uint32_t index = offset + (i * n2 + j) * block;
                buffer[index]     = base;
                buffer[index + 1] = base + pow(2, i);
            }
        }
    }

    void hypercubeEdges(std::vector<float> &buffer, uint32_t dimension, uint32_t stride, uint32_t offset) {
        const uint32_t block = 8 + stride;
        const uint32_t n1 = dimension;
        const uint32_t n2 = pow(2, (dimension - 1));
        buffer.resize(offset + n1 * n2 * block);
        for (size_t i = 0; i < n1; i++) {
            for (size_t j = 0; j < n2; j++) {
                const uint32_t index = offset + (i * n2 + j) * block;
                buffer[index]     = index;
                buffer[index + 1] = i;
                buffer[index + 2] = j;
                buffer[index + 3] = 0;
                buffer[index + 4] = index;
                buffer[index + 5] = i;
                buffer[index + 6] = j;
                buffer[index + 7] = 1;
            }
        }
    }

    void icositetrachoronVertices(std::vector<float> &buffer, float size, uint32_t stride, uint32_t offset) {
        const uint32_t n_block = 1 + stride;
        const uint32_t n1 = 8;
        const uint32_t n2 = 4;
        buffer.resize(offset + 24 * 4 * n_block);

```

```
        // { ±1, 0, 0, 0 } → 8 vertices
        for (size_t i = 0; i < n1; i++) {
            for (size_t j = 0; j < n2; j++) {
                const uint32_t index = offset + (i * n2 + j) * n_block;
                buffer[index] = j == (i / 2) ? (static_cast<bool>(i % 2) ? size : -size) : 0.0F;
            }
        }

        const uint32_t n_filled = offset + n1 * n2 * n_block;

        // { ±0.5, ±0.5, ±0.5, ±0.5 } → 16 vertices
        hypercubeVertices(buffer, 4, 0.5 * size, stride, n_filled);
    }

    void icositetrachoronIndices(std::vector<uint32_t> &buffer, uint32_t stride, uint32_t offset) {
        const uint32_t N_EDGES = 96;
        const uint32_t n_block = N_EDGE + stride;
        const uint32_t n1 = 4;
        const uint32_t n2 = 2;
        const uint32_t n3 = 8;

        buffer.resize(offset + std::size_t{ N_EDGES * n_block });

        // 64 edges of first 8 vertices
        for (size_t i = 0; i < n1; i++) {
            for (size_t j = 0; j < n2; j++) {
                for (size_t k = 0; k < n3; k++) {
                    const uint32_t pow1 = (1 << i);
                    const uint32_t pow2 = (2 << i);
                    const uint32_t index = ((i * n2 + j) * n3 + k) * n_block;
                    buffer[index]     = i * n2 + j;
                    buffer[index + 1] = (k % pow1) + (k / pow1 * pow2) + 8 + j * pow1;
                }
            }
        }

        const uint32_t n_filled = n1 * n2 * n3 * n_block;

        // 32 edges of hypercube
        xdvk::hypercubeIndices(buffer, 4, stride, offset + n_filled);
        for (size_t i = 0; i < 32; i++) {
            for (size_t j = 0; j < 2; j++) {
                const uint32_t index = i * n_block + j + n_filled;
                buffer[index] += 8;
            }
        }
    }

    auto rotationSize(const uint32_t dimension) -> size_t {
        return dimension * (dimension - 1) / 2;
    }

    auto transformSize(const uint32_t dimension) -> uint32_t {
        return 2 * dimension + static_cast<uint32_t>(rotationSize(dimension));
    }
}
```

Listing B.8: `src/main/xdvk.cpp`

B.1.9. `xdvk.hpp`

```
// Oliver Kovacs 2021 - xdvk - MIT

#ifndef XDVK_HPP
#define XDVK_HPP

#include <cstdint>
#include <vector>
#include <vulkan/vulkan.hpp>
```

```
namespace xdvk {

    template<uint32_t D>
    struct Transform {
        float buffer[2 * D + D * (D - 1) / 2]; // NOLINT(*-avoid-c-arrays)
        float *position = buffer;
        float *scale = &buffer[D];
        float *rotation = &buffer[2 * D];

        Transform();
    };

    struct Geometry {

        // ssbo vertex data
        std::vector<float> vertices;

        // attribute vertex data
        VkBuffer vertexBuffer;
        VkDeviceMemory vertexBufferMemory;

        // indexed draw data
        VkBuffer indexBuffer;
        VkDeviceMemory indexBufferMemory;
        uint32_t indexBufferSize;
        bool drawIndexed = true;

        size_t vertexBufferIndex;
        size_t indexBufferIndex;
        size_t transformBufferIndex;
    };

    template<uint32_t D>
    struct Entity {
        uint64_t id;
        uint64_t components;
        Transform<D> transform;
        Geometry geometry;
    };

    struct Index {
        uint64_t id;
        uint32_t index;
        uint32_t next;
    };

    template<uint32_t D>
    class Scene {
        public:

        std::vector<Index> indices;
        std::vector<Entity<D>> entities;

        explicit Scene(size_t reserve);

        auto has(uint64_t id) -> bool;
        auto get(uint64_t id) -> Entity<D> &;
        auto add() -> uint64_t;
        void remove(uint64_t id);

        private:
        uint32_t entity_count = 0;
        uint32_t freelist;
    };

    void hypercubeVertices(std::vector<float> &vertices, uint32_t dimension, float size, uint32_t stride
      = 0, uint32_t offset = 0);
    void hypercubeIndices(std::vector<uint32_t> &buffer, uint32_t dimension, uint32_t stride = 0,
     uint32_t offset = 0);
    void hypercubeEdges(std::vector<float> &buffer, uint32_t dimension, uint32_t stride = 0, uint32_t
     offset = 0);
    void icositetrachoronVertices(std::vector<float> &buffer, float size, uint32_t stride = 0, uint32_t
     offset = 0);
    void icositetrachoronIndices(std::vector<uint32_t> &buffer, uint32_t stride = 0, uint32_t offset =
```

```
     0);

    template<uint32_t D>
    void hypercubeTransform(std::vector<float> &buffer, Transform<D> transform, uint32_t index, uint32_t
      stride, uint32_t offset);

    auto rotationSize(uint32_t dimension) -> size_t;
    auto transformSize(uint32_t dimension) -> uint32_t;

    template<typename T>
    void printVector(std::vector<T> vector, const std::string &name);

    template<typename T, size_t N>
    void printArray(std::array<T, N> array, const std::string &name);
}

#include "xdvk.t.hpp"

#endif /* XDVK_HPP */
```

Listing B.9: `src/main/xdvk.hpp`

B.1.10. `xdvk.t.hpp`

```
#ifndef XDVK_TPP
#define XDVK_TPP

#include <iostream>
#include "xdvk.hpp"

template<uint32_t D>
xdvk::Transform<D>::Transform() {
    std::fill_n(buffer, 2 * D + D * (D - 1) / 2, 0.0F);
    std::fill_n(scale, D, 1.0F);
};

namespace xdvk {

    #define INDEX_MASK 0xffffffff
    #define NEW_OBJECT_ID_ADD 0x100000000
    template<uint32_t D>
    Scene<D>::Scene(size_t reserve) {
        indices.resize(reserve);
        entities.reserve(reserve);
        for (size_t i = 0; i < reserve; i++) {
            indices[i].id = i;
            indices[i].next = i + 1;
        }
        freelist = 0;
    };

    template<uint32_t D>
    auto Scene<D>::has(uint64_t id) -> bool {
        Index &index = indices[id & INDEX_MASK];
        return index.id == id && index.index != UINT32_MAX;
    }

    template<uint32_t D>
    auto Scene<D>::get(uint64_t id) -> Entity<D> & {
        return entities[indices[id & INDEX_MASK].index];
    }

    template<uint32_t D>
    auto Scene<D>::add() -> uint64_t {
        Index &index = indices[freelist];
        freelist = index.next;
        index.id += NEW_OBJECT_ID_ADD;
        index.index = entity_count++;
        Entity<D> &entity = entities.emplace_back();
```

```
        entity.id = index.id;
        return entity.id;
    }

    template<uint32_t D>
    void Scene<D>::remove(uint64_t id) {
        Index &index = indices[id & INDEX_MASK];
        Entity<D> &entity = entities[index.index];
        entity = entities[--entity_count];
        entities.pop_back();
        indices[entity.id & INDEX_MASK].index = index.index;
        index.index = UINT32_MAX;
        index.next = freelist;
        freelist = id & INDEX_MASK;
    }

    template<uint32_t D>
    void hypercubeTransform(std::vector<float> &buffer, Transform<D> transform, uint32_t index, uint32_t
      stride, uint32_t offset) {
        uint32_t size = transformSize(D);
        const uint32_t block = size + stride;
        buffer.resize(offset + (index + 1) * block);
        std::copy_n(&transform.buffer[offset + index * stride], size, buffer.begin());
    }

    template<typename T>
    void printVector(std::vector<T> vector, const std::string &name) {
        std::cout << name << "[" << vector.size() << "] = [ ";
        for (auto elem : vector) {
            std::cout << elem << " ";
        }
        std::cout << "]" << std::endl;
    }

    template<typename T, size_t N>
    void printArray(std::array<T, N> array, const std::string &name) {
        std::cout << name << "[" << array.size() << "] = [ ";
        for (auto elem : array) {
            std::cout << elem << " ";
        }
        std::cout << "]" << std::endl;
    }
};

#endif /* XDVK_TPP */
```

Listing B.10: `src/main/xdvk.t.hpp`

B.2. src/shaders/

B.2.1. shader.frag

```
#version 450

layout(binding = 1) uniform sampler2D texSampler;

layout(location = 0) in vec4 fragColor;

layout(location = 0) out vec4 outColor;

void main() {
    outColor = fragColor;
}
```

Listing B.11: `src/shaders/shader.frag`

B.2.2. shader.vert

```
#version 450

#define pi 3.1415926535

layout(constant_id = 0) const int n = 3;
const int a_n = n * (n - 1) / 2;
const int t_n = 2 * n + a_n;

layout(push_constant) uniform Constants {
    float entity;
    float vertexIndex;
    float indexIndex;
    float transformIndex;
} constants;

layout(binding = 0) uniform UniformBufferObject {
    vec2 res;
    float time;
} ubo;

layout(binding = 2) readonly buffer StorageBuffer {
    float vertices[];
} ssbo;

layout(binding = 3) readonly buffer StorageBuffer2 {
    float transforms[];
} ssbo2;

layout(location = 0) in float entity;

layout(location = 0) out vec4 fragColor;

struct Transform {
    float position[n];
    float scale[n];
    float rotation[a_n];
};

vec3 xy_scale = vec3(1.0, 1.0, 1.0);

float canvas_z = 4.0;
float camera_z = 0.0;

void fetchVertex(inout float[n] vertex, int stride, int offset) {
    int block = n + stride;
    int index = offset + gl_VertexIndex * block;
    for (int i = 0; i < n; i++) {
        vertex[i] = ssbo.vertices[index + i];
    }
}

void fetchTransform(inout Transform transform, int transform_index, int stride, int offset) {
    int block = t_n + stride;
    int index = transform_index;
    for (int i = 0; i < n; i++) {
        transform.position[i] = ssbo2.transforms[index + i];
        transform.scale[i] = ssbo2.transforms[index + n + i];
    }
    for (int i = 0; i < a_n; i++) {
        transform.rotation[i] = ssbo2.transforms[index + 2 * n + i];
    }
}

void scaleVertex(inout float vertex[n], inout float scale[n]) {
    for (int i = 0; i < n; i++) {
        vertex[i] *= scale[i];
    }
}

void rotateVertex(inout float vertex[n], inout float rotation[a_n]) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n; j++) {
```

```
            if (j <= i) continue;
            const float a = rotation[a_n - int(float((n - i - 1) * (n - i)) / 2.0) + j - i - 1];
            const float cos_a = cos(a);
            const float sin_a = sin(a);
            const float vi = vertex[i];
            const float vj = vertex[j];
            vertex[i] = vi *  cos_a + vj * sin_a;
            vertex[j] = vi * -sin_a + vj * cos_a;
        }
    }
}

void translateVertex(inout float vertex[n], inout float position[n]) {
    for (int i = 0; i < n; i++) {
        vertex[i] += position[i];
    }
}

void transformVertex(inout float vertex[n], Transform transform) {
    scaleVertex(vertex, transform.scale);
    rotateVertex(vertex, transform.rotation);
    translateVertex(vertex, transform.position);
}

void projectVertex(inout float vertex[n]) {
    float z_diff = canvas_z - camera_z;
    for (int i = n - 1; i >= 2; i--) {
        float w = z_diff / (canvas_z - vertex[i]);
        for (int j = 0; j < n; j++) {
            if (j >= i) break;
            vertex[j] *= w;
        }
    }
}

vec3 arrayToVec3(float[n] array) {
    return vec3(array[0], array[1], array[2]);
}

void main() {
    if (ubo.res.x > ubo.res.y) xy_scale = vec3(ubo.res.y / ubo.res.x, 1.0, 1.0);
    else xy_scale = vec3(1.0, ubo.res.x / ubo.res.y, 1.0);

    float[n] vertex;
    Transform transform;
    fetchVertex(vertex, 0, int(constants.vertexIndex));
    fetchTransform(transform, int(constants.transformIndex), 0, 0);
    transformVertex(vertex, transform);
    projectVertex(vertex);

    vec3 pos = arrayToVec3(vertex) * vec3(0.5, 0.5, 0.5);

    pos *= xy_scale;
    gl_Position = vec4(pos.xy, 0.0, 1.0);

    float rgb_s = 0.5;
    vec4 color = vec4(
        sin(mod(gl_VertexIndex * rgb_s + ubo.time * 1 + 0 * pi / 3, 2 * pi)),
        sin(mod(gl_VertexIndex * rgb_s + ubo.time * 1 + 2 * pi / 3, 2 * pi)),
        sin(mod(gl_VertexIndex * rgb_s + ubo.time * 1 + 4 * pi / 3, 2 * pi)),
        1.0
    );

    fragColor = color;
}
```

Listing B.12: `src/shaders/shader.vert`

Acronyms

API application programming interface. 5, 27, 28, 31, 32
CAD computer-aided design. 7
CPU central processing unit. 5, 25, 26, 28, 29
FB framebuffer. 38
FPS frames per second. 37
GLFW Graphics Library Framework. 32
GLM OpenGL Mathematics. 32
GLSL OpenGL Shading Language. 32
GPU graphics processing unit. 5, 26–29, 31, 37
I/O input/output. 25
ISA instruction set architecture. 25
RAM random-access memory. 5, 26
SDK software development kit. 32
SPIR Standard Portable Intermediate Representation. 32
SSBO shader storage buffer object. 31
UBO uniform buffer object. 31
VBO vertex buffer object. 30, 31
VRAM video random-access memory. 36
VSync vertical synchronization. 37

Bibliography

Bläser, M. (2013). Fast matrix multiplication. *Theory of Computing*, 1–60 (cit. on p. 30).

Boreskov, A., & Shikin, E. (2014). *Computer graphics from pixels to programmable graphics hardware*. CRC Press. (Cit. on pp. 15–19, 26, 27, 45).

Bryant, R. E., & O'Hallaron, D. R. (2016). *Computer systems: A programmer's perspective*. (Cit. on pp. 25, 26).

Cozzi, P., & Riccio, C. (2012). *OpenGL Insights* (1st ed.). CRC Press. (Cit. on p. 31).

de Laat, D. (2011). *Approximating manifolds by meshes: Asymptotic bounds in higher codimension* (Master's thesis). University of Groningen. (Cit. on p. 9).

Kelly, S. (2016). *Computational methods for parametrization of polytopes*. Worcester Polytechnic Institute. (Cit. on pp. 9, 10).

Marschner, S., & Shirley, P. (2016). *Fundamentals of computer graphics*. CRC Press. (Cit. on pp. 7, 10, 12, 15–17).

Noll, A. M. (1967). A computer technique for displaying n-dimensional hyperobjects. *Communications of the ACM*, *10*(8), 469–473. https://doi.org/10.1145/363534.363544 (cit. on pp. 21, 22, 46)

OpenGL Wiki. (2020). *Shader storage buffer object*. Retrieved January 29, 2023, from https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object. (Cit. on p. 31)

Rudin, W. (1976). *Principles of mathematical analysis*. McGraw-Hill. (Cit. on p. 15).

Segal, M., & Akeley, K. (2022). The OpenGL® Graphics System: A Specification (cit. on p. 30).

Strang, G. (2011). Lecture 30: Linear transformations and their matrices. In *Linear Algebra - MIT Course 18.06SC*. Massachusetts Institute of Technology. https://ocw.mit.edu/courses/18-06sc-linear-algebra-fall-2011/pages/positive-definite-matrices-and-applications/linear-transformations-and-their-matrices/. (Cit. on pp. 12, 15)

The Khronos® Vulkan Working Group. (2023). Vulkan 1.3.240 - A Specification (with all registered Vulkan extensions) (cit. on p. 30).

List of Figures

Eidesstattliche Erklärung

Ich, Oliver Kovacs, erkläre hiermit eidesstattlich, dass ich diese vorwissenschaftliche Arbeit selbständig und ohne Hilfe Dritter verfasst habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als Zitate kenntlich gemacht und alle verwendeten Quellen angegeben habe.

Steyr, am ____________________ ________________________

Datum Unterschrift