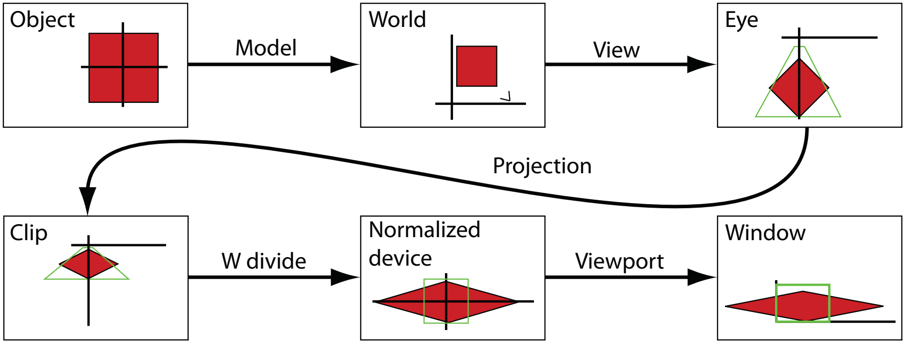
02561 Computer Graphics

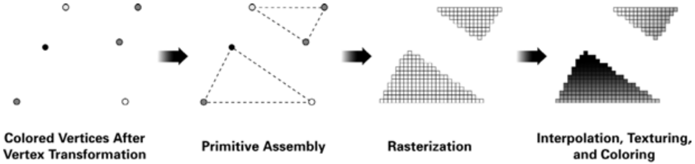
Model, view, projection

Jeppe Revall Frisvad

September 2020

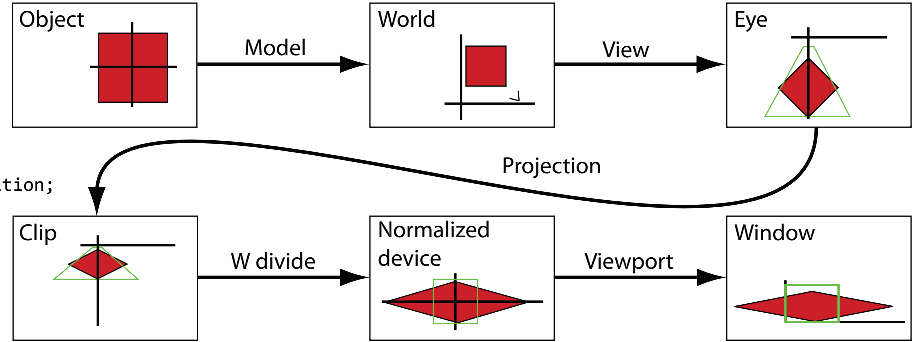
Rasterization pipeline





Rasterization pipeline

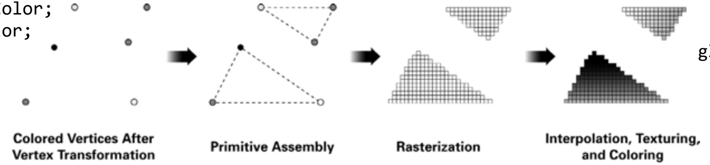
```
attribute vec4 vPosition;
```




```
gl_Position
= P*V*M*vPosition;
```

```
gl_FragCoord
```

```
attribute vec4 vColor;
varying vec4 fColor;
fColor = vColor;
```




```
varying vec4 fColor;
gl_FragColor = fColor;
```

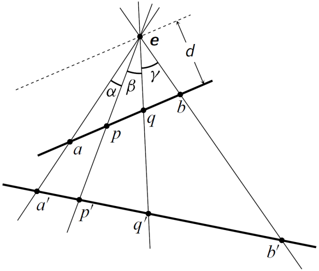
vertex shader

fragment shader

Why 4-vectors and what is w?

- As with curves (Week 3), we can include more advanced (rational) transformations in a matrix representation if we use homogeneous coordinates.
- Homogeneous coordinates: add a w-coordinate that we divide by in the end. In this projective space, we have vectors $(x, y, z, w) \mapsto (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$.
  - Position vectors: $(x, y, z, 1)$.
  - Direction vectors: $(x, y, z, 0)$ (at infinity and thus invariant under translation).
- Points along a straight line passing through the origin (the origin excluded) are equivalent in projective space (equivalence relation):

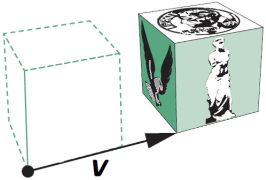
$$(x, y, z, 1) \sim (wx, wy, wz, w)\,, \quad \text{for } w \neq 0\,.$$



- Perspective is mapping of 3D shapes to a surface. This is what a camera does.
- If we move our virtual camera to the origin ($\boldsymbol{e}$) and rotate the coordinate system to the basis of the image plane, we can use w to do perspective projection (to d).

Transformation matrices

- Translation of a point $\boldsymbol{x}$ along a vector $\boldsymbol{v}$ to a point $\boldsymbol{x}'$:



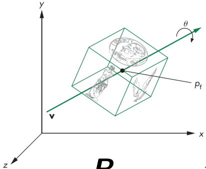
$$\begin{bmatrix} \boldsymbol{x}' \\ 1 \end{bmatrix} = \boldsymbol{T} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}, \quad \boldsymbol{T} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{v} \\ 0 & 1 \end{bmatrix},$$

$\boldsymbol{T} \in \mathbb{R}^{4\times4}$ is a translation matrix and
$\boldsymbol{I} \in \mathbb{R}^{3\times3}$ is an identity matrix.

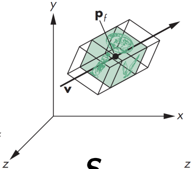
- Other transformations for which $\boldsymbol{x}' = \boldsymbol{A}\boldsymbol{x}$:

$$\begin{bmatrix} \boldsymbol{x}' \\ 1 \end{bmatrix} = \boldsymbol{B} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} \boldsymbol{A} & 0 \\ 0 & 1 \end{bmatrix},$$

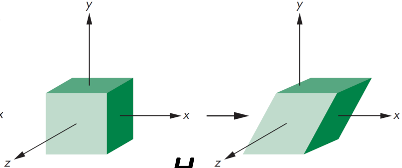
- $\boldsymbol{A} \in \mathbb{R}^{3\times3}$ is a rotation matrix ($\boldsymbol{R}$), scaling matrix ($\boldsymbol{S}$), or shearing matrix ($\boldsymbol{H}$).



$\boldsymbol{R}$



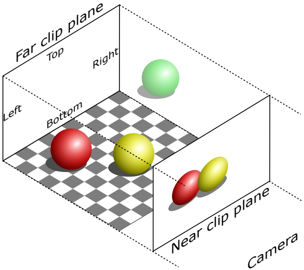
$\boldsymbol{S}$



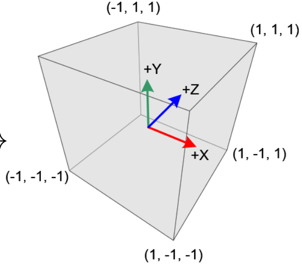
$\boldsymbol{H}$

Normalized device coordinates (NDC)

- If we use identity matrices for all vertex shader transformations ($\boldsymbol{M} = \boldsymbol{V} = \boldsymbol{P} = \boldsymbol{I}$) and $w = 1$, we are effectively working in NDC space.



$\leftrightarrow$



default
view volume
(NDC cube)

- We have so far used $\boldsymbol{M} = \boldsymbol{V} = \boldsymbol{P} = \boldsymbol{I}$ and $w = 1$ (no transformation).
- The image plane is then $z = -1$ and no geometry outside the NDC cube is drawn.
- Any standard projection matrix $\boldsymbol{P}$ (perspective or orthographic) flips the sign of the z-coordinate to have a right-handed coordinate system. The view direction in eye space is thus normally the negative z-axis.

Exercise: depth in drawing program (W02 extra)

- With little effort, we can make the drawing program always draw newer points, triangles, and circles on top of previous drawings.
- Do this by adding a z-coordinate to every position in the vertex buffer.
- Let this z-coordinate decrease with the index

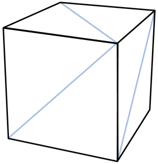
$$z = 1 - 2\frac{i+1}{n},$$

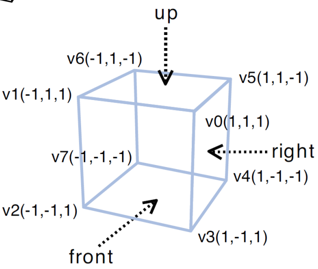
  where i is index and n is the maximum number of vertices.
- Remember that buffer data allocation and attribute pointer specification need to know that an extra coordinate was added.
- Now enable a depth test to have the graphics pipeline draw the geometry with smallest z-coordinate regardless of the order of the draw calls.
  - In the init function:

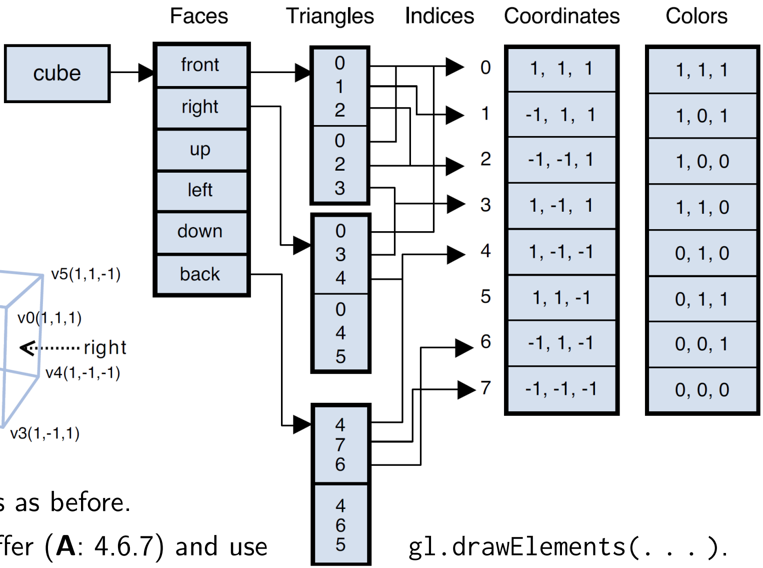
    ```
    gl.enable(gl.DEPTH_TEST);
    ```
  - In the render function:

    ```
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    ```

Drawing a cube using an indexed face set

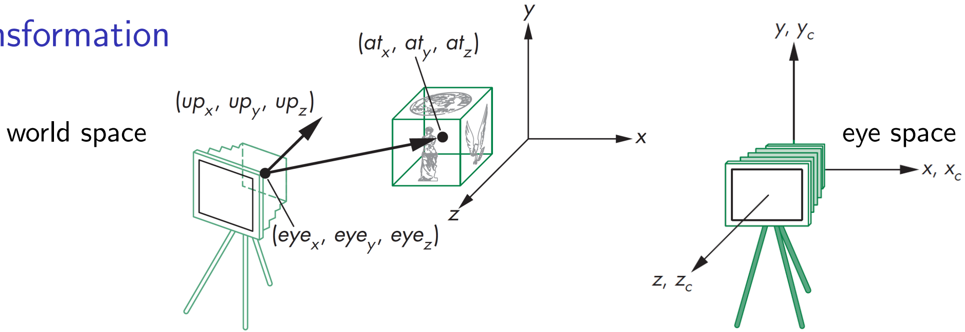






- Positions and colors as before.
- Create an index buffer (**A**: 4.6.7) and use `gl.drawElements(. . . )`.

View transformation



- Eye space has the camera at the origin looking down the negative z-axis.
- The view transformation performs a change of coordinates.
- Given eye point $\boldsymbol{e}$, look-at point $\boldsymbol{a}$, and up vector $\boldsymbol{u}$ in world space, the basis vectors of eye space in world space coordinates are

$$\vec{b}_1 = \frac{\boldsymbol{u} \times \vec{b}_3}{\|\boldsymbol{u} \times \vec{b}_3\|}, \qquad \vec{b}_2 = \vec{b}_3 \times \vec{b}_1\,, \qquad \vec{b}_3 = \frac{\boldsymbol{e} - \boldsymbol{a}}{\|\boldsymbol{e} - \boldsymbol{a}\|}.$$

Change of coordinates

- Given basis vectors $\vec{b}_1, \vec{b}_2, \vec{b}_3 \in \mathbb{R}^{3\times 1}$ of space a in coordinates of space b, the change of basis matrix ${}_b\boldsymbol{M}_a \in \mathbb{R}^{3\times 3}$ from a to b is

$$ {}_b\boldsymbol{M}_a = \begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \end{bmatrix}. $$

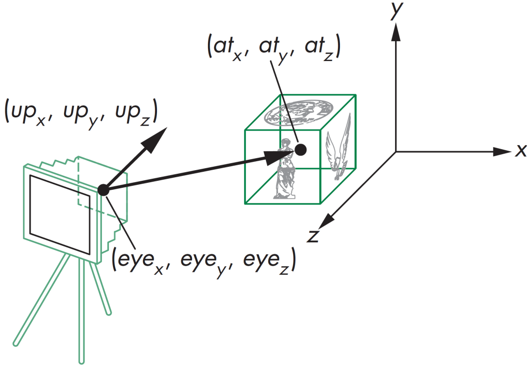
- If the basis is orthonormal: ${}_a\boldsymbol{M}_b = {}_b\boldsymbol{M}_a^{-1} = {}_b\boldsymbol{M}_a^T$.
- Change of coordinates is translation and rotation: $\boldsymbol{V} = \boldsymbol{R}\boldsymbol{T} = \begin{bmatrix} \vec{b}_1^T & -\boldsymbol{e}\cdot\vec{b}_1 \\ \vec{b}_2^T & -\boldsymbol{e}\cdot\vec{b}_2 \\ \vec{b}_3^T & -\boldsymbol{e}\cdot\vec{b}_3 \\ 0 & 1 \end{bmatrix}.$
  - Translation to displace geometry so that it is positioned relative to the origin as it was previously positioned relative to the camera:

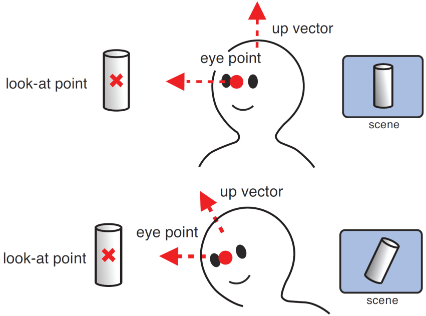
$$ \boldsymbol{T} = \begin{bmatrix} \boldsymbol{I} & -\boldsymbol{e} \\ 0 & 1 \end{bmatrix}. $$

  - Rotation to perform change of basis from world space to eye space:

$$ \boldsymbol{R} = \begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \end{bmatrix}^T. $$

The look-at function





- To configure camera extrinsics (position and orientation), we use an eye point $\boldsymbol{e}$, a look-at point $\boldsymbol{a}$, and an up vector $\boldsymbol{u}$:

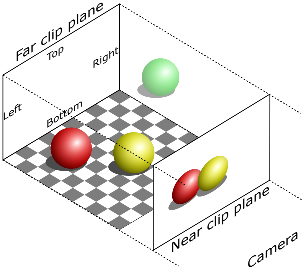
```
var V = lookAt(eye, at, up);
```

- Send to shader using

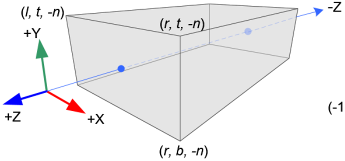
```
gl.uniformMatrix4fv(VLoc, false, flatten(V));
```

Orthographic projection

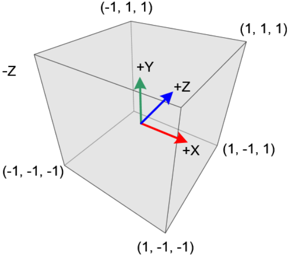
- Selecting an arbitrary view volume



orthographic projection



view volume



NDC cube

- The ortho function creates a projection matrix transforming from an orthographic view volume to the NDC cube.

  var P = ortho(left, right, bottom, top, near, far);

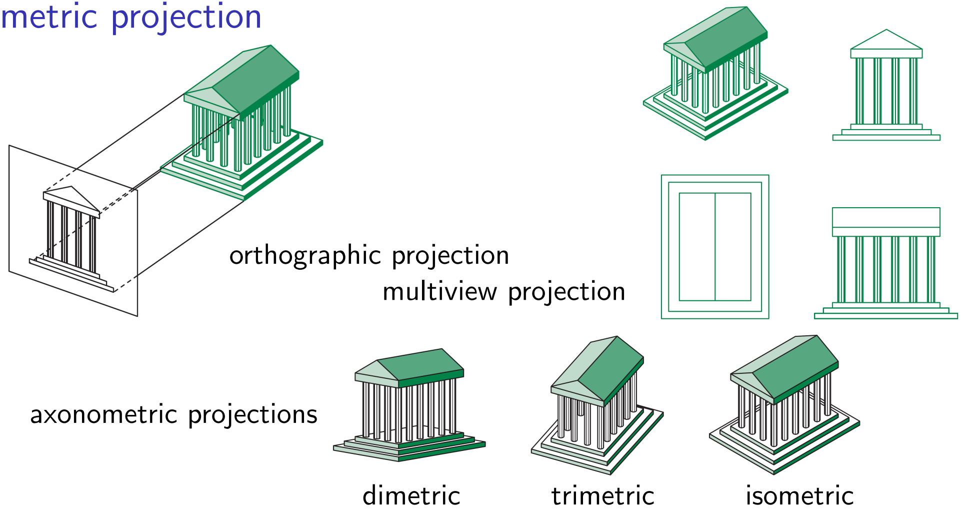
  $\boldsymbol{P} \quad l \quad r \quad b \quad t \quad n \quad f$

$$\boldsymbol{P} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Send to shader using

```
gl.uniformMatrix4fv(PLoc, false, flatten(P));
```

Axonometric projection



- Axonometric views are about symmetry and foreshortening of distances in the three principal directions around a corner.
  A cube in isometric view is seen with three edges of equal length meeting at a corner.

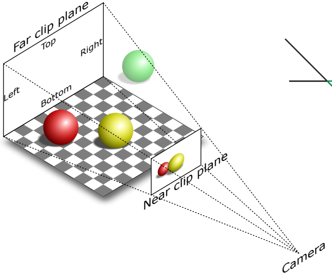
Exercise: isometric view of a wireframe cube (W03P1)

- Draw a cube using an indexed face set (**A**: 4.6.1 and 4.6.7).
- Functions are available for building transformation matrices (**A**: 4.11.2–4.11.3):

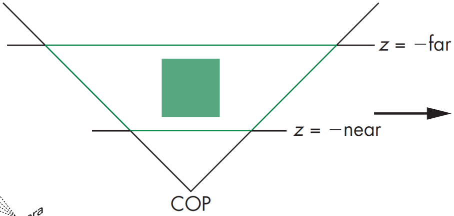
```
var I = mat4();      // identity matrix
var R = rotate(angle, direction);
var Rx = rotateX(angle);
var Ry = rotateY(angle);
var Rz = rotateZ(angle);
var S = scalem(s_x, s_y, s_z);
var T = translate(t_x, t_y, t_z);
var c = mult(a, b);  // c = a*b
```

- Use a model matrix to scale and translate the cube (**A**: 4.9) so that its diagonal is from $(0, 0, 0)$ to $(1, 1, 1)$, or change the coordinates of the vertex positions.
- Draw a wireframe cube by modifying the indices in the indexed face set and using the draw mode `gl.LINES` instead of `gl.TRIANGLES`.
- Use the `lookAt` function to construct a view matrix (**A**: 5.3.3) so that the wireframe cube is rendered in isometric view.

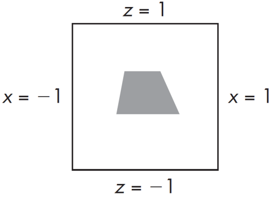
Perspective projection



perspective projection



clip space



NDC cube

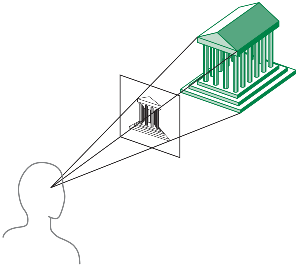
- The perspective function creates a projection matrix transforming the view frustum to clip space. The frustum becomes the NDC cube after w-divide.

var P = perspective(fovy, aspect, near, far);
$\boldsymbol{P}$ $\quad\alpha\quad A\quad n\quad f$

$$\boldsymbol{P} = \begin{bmatrix} \frac{1}{A}\cot\frac{\alpha}{2} & 0 & 0 & 0 \\ 0 & \cot\frac{\alpha}{2} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- α is the vertical field of view (angle in degrees)
- $A = \frac{w}{h}$ is the aspect ratio of the canvas.

Classical perspective views



perspective view:

three-point

two-point

one-point

- Classical perspective views are about the number of principal directions in the image with vanishing points.
  - 1-point: one vanishing point, two principal directions parallel to the image plane.
  - 2-point: two vanishing points, one principal direction parallel to the image plane.
  - 3-point: three vanishing points, no principal directions parallel to the image plane.
- When drawing a cube, look for edges that are parallel in the image.

Model, view, and projection matrices

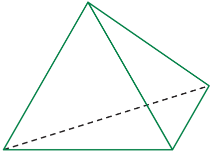
- Recommended mode of operation:
  - The projection matrix $\boldsymbol{P}$ depends on camera intrinsics and is set during initialization.
  - The view matrix $\boldsymbol{V}$ depends on camera extrinsics and is set during animation.
  - the model matrix $\boldsymbol{M}$ places objects in the scene and is set during rendering.
- We can use the same vertex buffer to draw multiple instances of an object.
- This simply requires a different model matrix for each instance.
- Before making a draw call, we set the model matrix of the instance to be rendered.
- The order of multiplication of matrices is important (matrix multiplication is not commutative):

$$
\begin{aligned}
\boldsymbol{x}_{\text{world}} &= \boldsymbol{M}\,\boldsymbol{x}_{\text{model}} \\
\boldsymbol{x}_{\text{view}} &= \boldsymbol{V}\,\boldsymbol{x}_{\text{world}} \\
\boldsymbol{x}_{\text{clip}} &= \boldsymbol{P}\,\boldsymbol{x}_{\text{view}} \\
\boldsymbol{x}_{\text{clip}} &= \boldsymbol{P}\boldsymbol{V}\boldsymbol{M}\,\boldsymbol{x}_{\text{model}}\,,
\end{aligned}
$$

Exercise: perspective views of three wireframe cubes (W03P2)

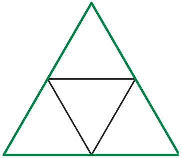
- Return to your program that draws a wireframe cube in isometric view.
- Use the `perspective` function to construct a projection matrix (**A**: 5.6.1) so that the cube is in a view frustum with a 45° vertical field of view.
- Use the `lookAt` function to construct a view matrix (**A**: 5.3.3) so that the cube is rendered in one-point perspective.
- Use rotation and translation matrices to construct model matrices for rendering three instances of the cube in one-, two-, and three-point perspectives.

Drawing a sphere using subdivision (**A**: 6.6)



- Take a tetrahedron (3D simplex) with vertex positions:

  $(0, 0, 1)$, $(0, \frac{2\sqrt{2}}{3}, -\frac{1}{3})$, $(-\frac{\sqrt{6}}{3}, -\frac{\sqrt{2}}{3}, -\frac{1}{3})$, $(\frac{\sqrt{6}}{3}, -\frac{\sqrt{2}}{3}, -\frac{1}{3})$.
- These are points on the unit sphere with center at the origin.



- Do Loop subdivision of the triangles:
  For two vertices $\boldsymbol{a}$ and $\boldsymbol{b}$ the edge midpoint is $\boldsymbol{c}' = \dfrac{\boldsymbol{a}+\boldsymbol{b}}{2}$.
- Normalize the new vertex positions
  to push them back onto the unit sphere: $\boldsymbol{c}' = \dfrac{\boldsymbol{a}+\boldsymbol{b}}{\|\boldsymbol{a}+\boldsymbol{b}\|}$.
- Subdivide each triangle while pushing all vertex positions into an array.
- Do n recursions.