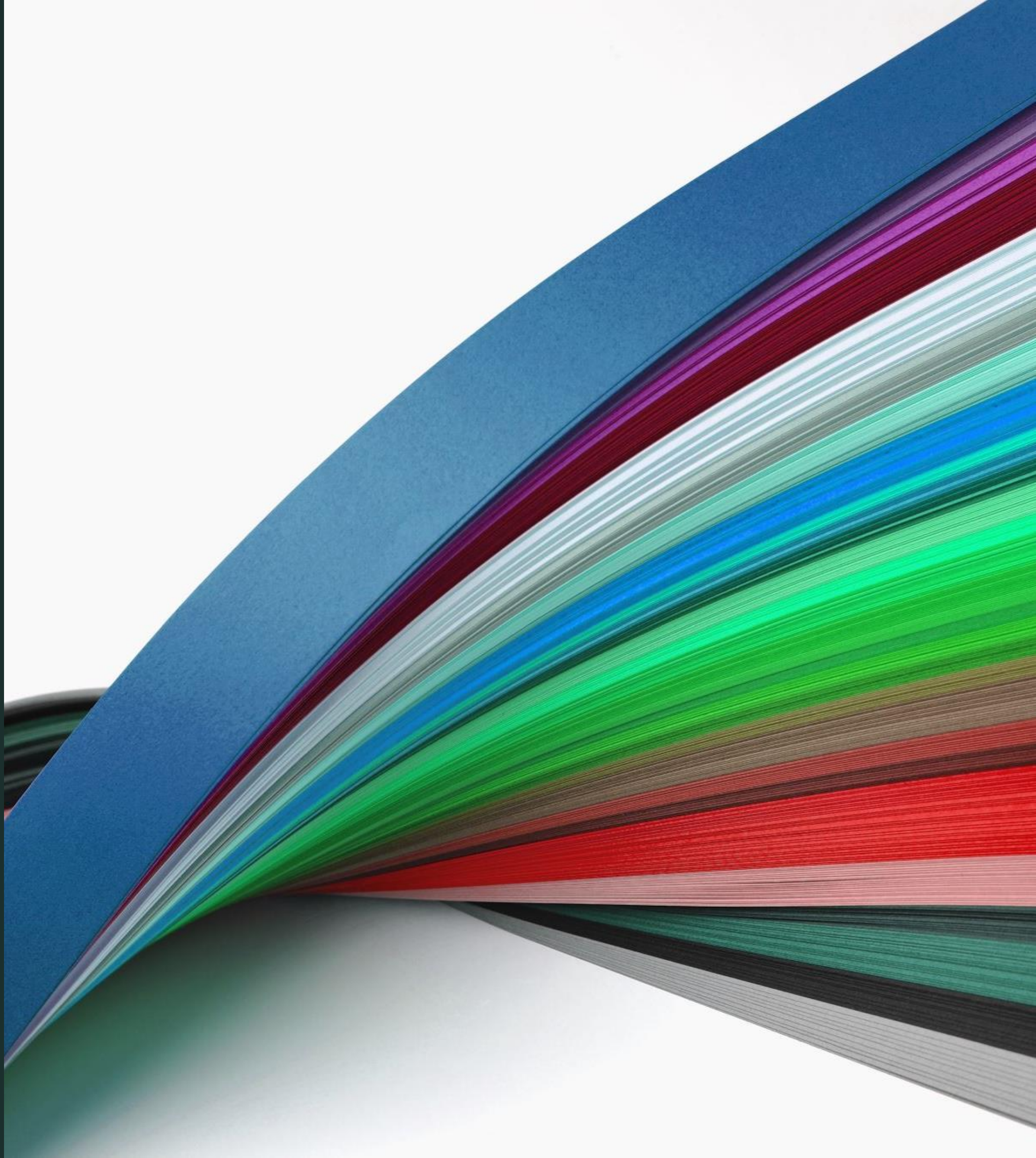
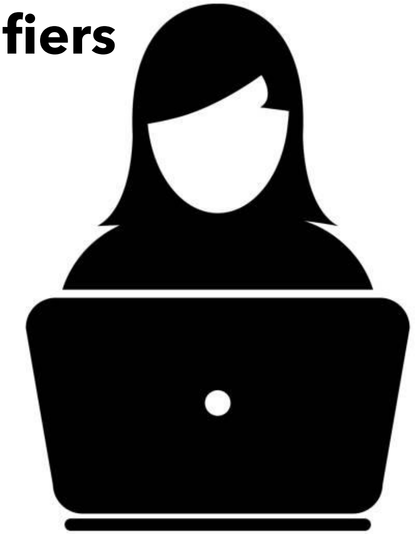
XR-WWW1

Webudvikling



Plan

1. HTML
2. CSS
3. Responsive Design
4. Individuel aflevering
5. JavaScript: Basics
6. JavaScript: JSON & Fetch
7. JavaScript: Sessions & Cookies
8. Gruppe aflevering
9. WebGL Simpel Trekant
10. WebGL Geometri & Rotation
11. WebGL Tekstur & Belysning
12. ➡ **WebGL Perspektiv & Modifiers**

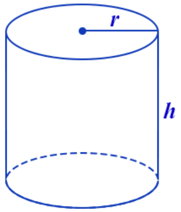


Opsummering

- Hvad er UV Koordinater
- Tilføj UV koordinater til shaders
- Display Options
- Indlæsning af tekstur fil
- Lav en midlertidig tekstur
- Tekstur filtre
- Opdater kode med UV koordinater
- Fortælle grafikkort om nye attributter
- Belysning af 3D overflader
- Lambertian (Diffuse) Shading
- Tilføjelse af normaler til kode

<u>Praksis:</u>

1. Implementering: indlæsning af tekstur og tilføjelse af UV koordinater
2. Implementering: belysning ved tilføjelse af normaler og Lambertian shading i fragment shader
3. Implementering af cylinder + UI

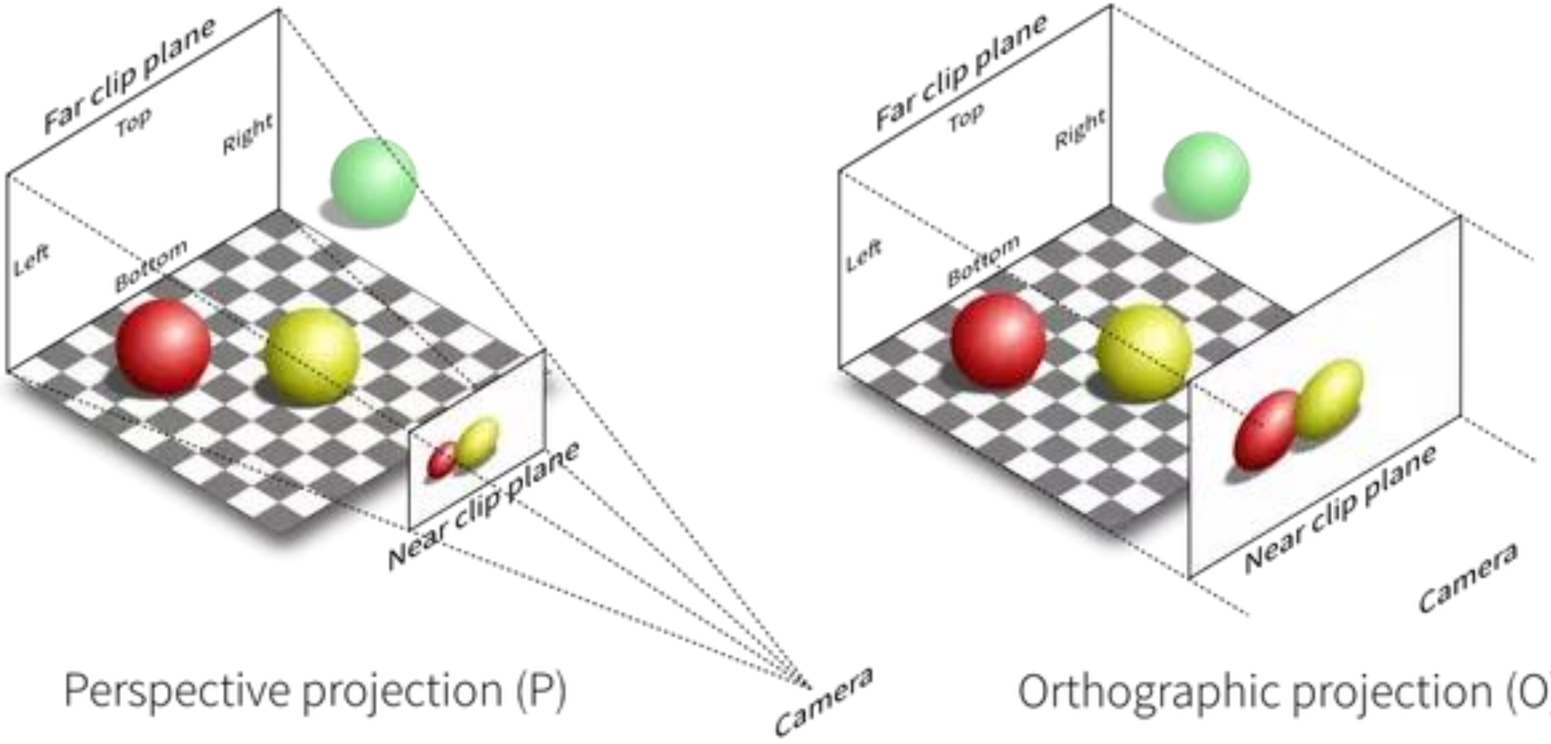


WebGL

Kan I huske dette fra DAP?

Indtil nu har vi kun brugt Ortografisk projektering, da vi får dette foræret af grafikkortet.

Men hvordan implementerer vi perspektiv projektering?



WebGL

$$\tilde{\mathbf{P}} = \begin{pmatrix} \tilde{p}_x \\ \tilde{p}_y \\ \tilde{p}_z \end{pmatrix} = \begin{pmatrix} f\,\frac{p_x}{p_z} \\ f\,\frac{p_y}{p_z} \\ f \end{pmatrix} \in \mathbb{R}^3 \longmapsto \underline{\tilde{\mathbf{P}}} = \begin{pmatrix} f\,p_x \\ f\,p_y \\ f\,p_z \\ p_z \end{pmatrix} \in \mathbb{H}^3$$

$$\underline{\tilde{\mathbf{P}}} = \begin{pmatrix} f\,p_x \\ f\,p_y \\ f\,p_z \\ p_z \end{pmatrix} = \underbrace{\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & f & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathtt{A}} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

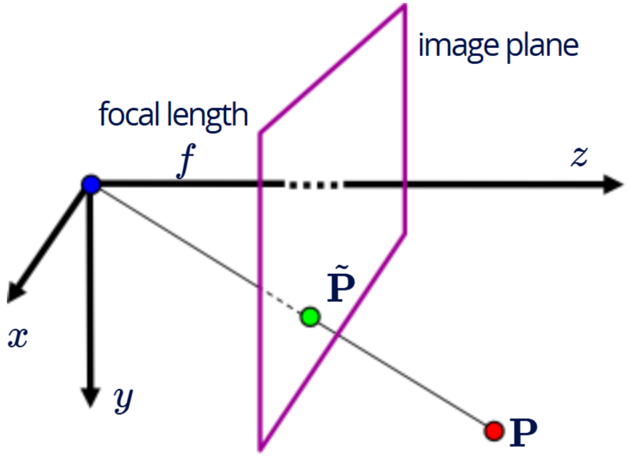
$$\underline{\tilde{\mathbf{P}}} = \mathtt{A}\,\underline{\mathbf{P}}$$

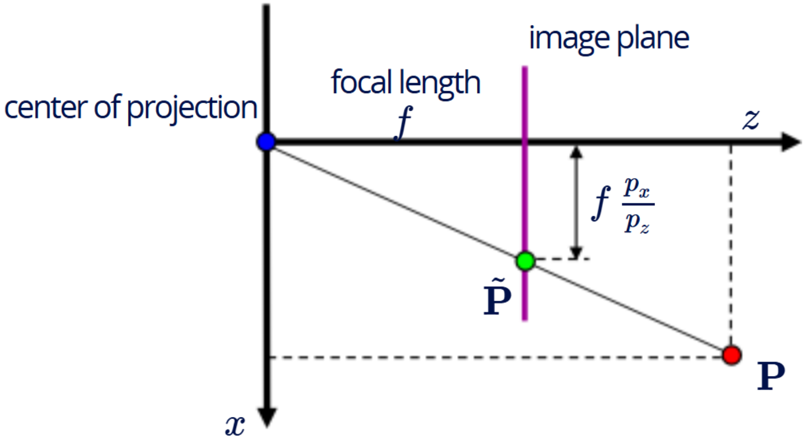
The formula for mapping a 3D point $\mathbf{P} = (p_x, p_y, p_z)^\top$ to a point $\tilde{\mathbf{P}} = (\tilde{p}_x, \tilde{p}_y, \tilde{p}_z)^\top$ located at the image plane of the camera is given by:

$$\tilde{\mathbf{P}} = \left(f\,\frac{p_x}{p_z}\,, f\,\frac{p_y}{p_z}\,, f\right)^\top$$

This follows immediately from the figure by application of the intercept theorem, since

$\frac{\tilde{p}_x}{f} = \frac{p_x}{p_z}$ and $\frac{\tilde{p}_y}{f} = \frac{p_y}{p_z}$





WebGL

$$\tilde{\mathbf{P}} = \begin{pmatrix}\tilde{p}_x\\ \tilde{p}_y\\ \tilde{p}_z\end{pmatrix} = \begin{pmatrix} f\,\frac{p_x}{p_z}\\ f\,\frac{p_y}{p_z}\\ f\end{pmatrix} \in \mathbb{R}^3 \longmapsto \underline{\tilde{\mathbf{P}}} = \begin{pmatrix} f\,p_x\\ f\,p_y\\ f\,p_z\\ p_z\end{pmatrix} \in \mathbb{H}^3$$

$$\underline{\tilde{\mathbf{P}}} = \begin{pmatrix} f\,p_x\\ f\,p_y\\ f\,p_z\\ \boxed{-p_z}\end{pmatrix} = \underbrace{\begin{bmatrix} f & 0 & 0 & 0\\ 0 & f & 0 & 0\\ 0 & 0 & f & 0\\ 0 & 0 & \boxed{-1} & 0\end{bmatrix}}_{\mathtt{A}} \begin{pmatrix} p_x\\ p_y\\ p_z\\ 1\end{pmatrix}$$

$$\underline{\tilde{\mathbf{P}}} = \mathtt{A}\,\underline{\mathbf{P}}$$

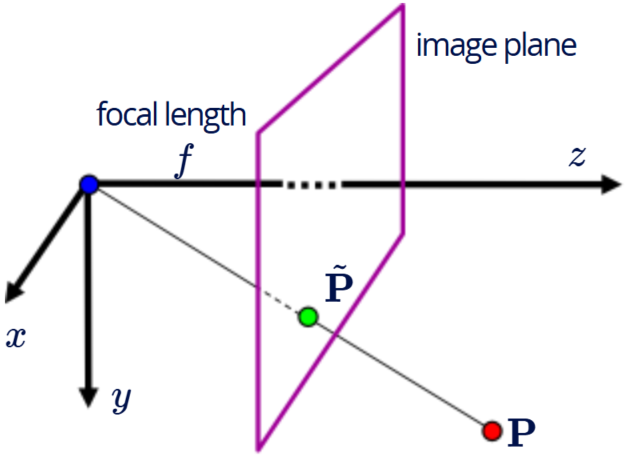
I OpenGL kigger vi langs den negative z akse, så disse to værdier skulle være negative.

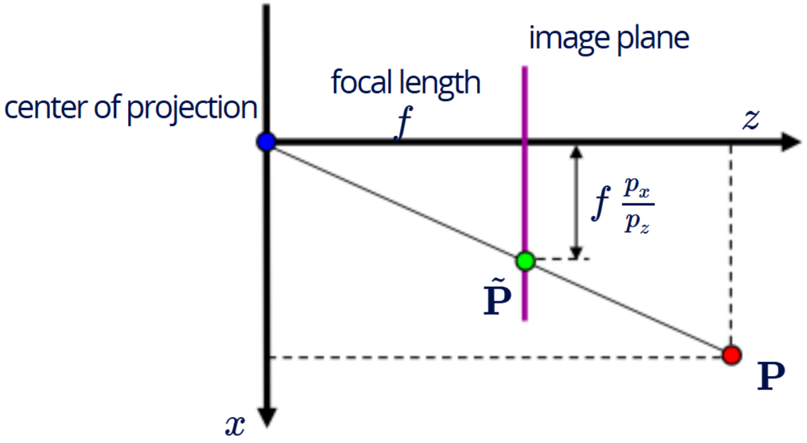
The formula for mapping a 3D point $\mathbf{P} = (p_x, p_y, p_z)^\top$ to a point $\tilde{\mathbf{P}} = (\tilde{p}_x, \tilde{p}_y, \tilde{p}_z)^\top$ located at the image plane of the camera is given by:

$$\tilde{\mathbf{P}} = \left(f\,\frac{p_x}{p_z}\,,\, f\,\frac{p_y}{p_z}\,,\, f\right)^\top$$

This follows immediately from the figure by application of the intercept theorem, since

$\frac{\tilde{p}_x}{f} = \frac{p_x}{p_z}$ and $\frac{\tilde{p}_y}{f} = \frac{p_y}{p_z}$





WebGL

In OpenGL, there is a so-called near- and a far-clipping plane

The near-plane and far-plane are located parallel to the image plane

Points are only displayed if their z-coordinate lies within the range defined by the near- and far-plane

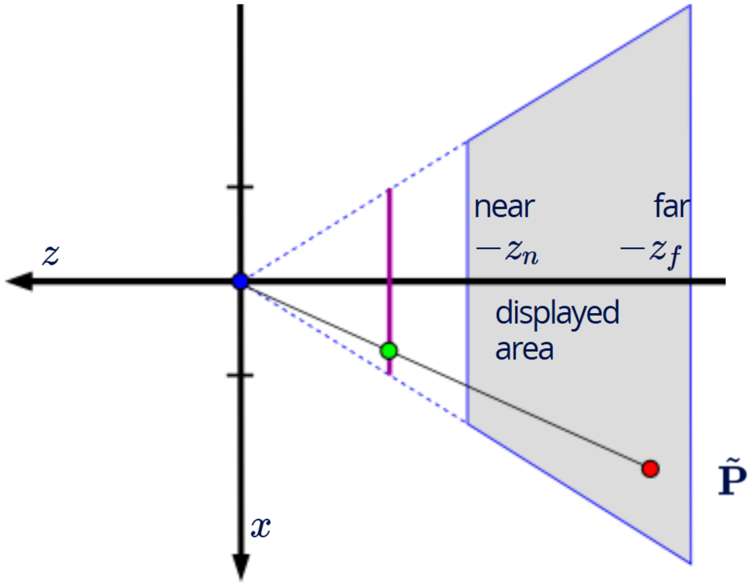
To this end, a new linear mapping is defined, such that for points with a z-coordinate on the near-plane it holds:

$$p_z = -z_n \quad \mapsto \quad \tilde{p}_z = -1$$

and for points on the far-plane:

$$p_z = -z_f \quad \mapsto \quad \tilde{p}_z = 1$$

In order to accomplish this, two new parameters α and β are added to the linear transformation matrix



$$\underline{\tilde{\mathbf{P}}} = \begin{pmatrix} f\,p_x \\ f\,p_y \\ \alpha\,p_z + \beta \\ -p_z \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \in \mathbb{H}^3$$

$$\tilde{\mathbf{P}} = (\tilde{p}_x, \tilde{p}_y, \tilde{p}_z)^\top = \left(f\,\frac{p_x}{-p_z}\,, f\,\frac{p_y}{-p_z}\,, -\alpha + \frac{\beta}{-p_z} \right)^\top \in \mathbb{R}^3$$

$$p_z = -z_n \mapsto \tilde{p}_z = -1 \quad \Rightarrow -\alpha + \frac{\beta}{z_n} = -1$$

$$p_z = -z_f \mapsto \tilde{p}_z = 1 \quad \Rightarrow -\alpha + \frac{\beta}{z_f} = 1$$

WebGL

Solving the equation system for α and β provides:

$$\alpha = \frac{z_f + z_n}{z_n - z_f}$$

$$\beta = \frac{2 z_f \, z_n}{z_n - z_f}$$

Thus, for the new projection matrix we have:

$$\underline{\tilde{\mathbf{P}}} = \underbrace{\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_f + z_n}{z_n - z_f} & \frac{2 z_f \, z_n}{z_n - z_f} \\ 0 & 0 & -1 & 0 \end{bmatrix}}_{\mathtt{A}} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

$$\underline{\tilde{\mathbf{P}}} = \mathtt{A} \, \underline{\mathbf{P}}$$

$$\underline{\tilde{\mathbf{P}}} = \begin{pmatrix} f \, p_x \\ f \, p_y \\ \alpha \, p_z + \beta \\ -p_z \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \in \mathbb{H}^3$$

$$\tilde{\mathbf{P}} = (\tilde{p}_x, \tilde{p}_y, \tilde{p}_z)^\top = \left(f \, \frac{p_x}{-p_z} \, , f \, \frac{p_y}{-p_z} \, , -\alpha + \frac{\beta}{-p_z} \right)^\top \in \mathbb{R}^3$$

$$p_z = -z_n \mapsto \tilde{p}_z = -1 \qquad \Rightarrow -\alpha + \frac{\beta}{z_n} = -1$$

$$p_z = -z_f \mapsto \tilde{p}_z = 1 \qquad \Rightarrow -\alpha + \frac{\beta}{z_f} = 1$$

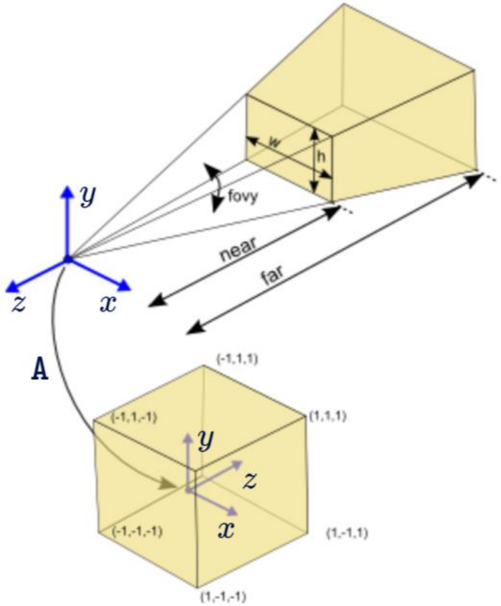
WebGL

Perspective Projection Matrix

$$\mathbf{A} = \begin{bmatrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{\text{far}+\text{near}}{\text{near}-\text{far}} & \frac{2*\text{far}*\text{near}}{\text{near}-\text{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

with $f = \text{cotan}(0.5 * \text{fovy})$

and $\text{aspect} = \text{w}/\text{h}$



```
function Perspective(fovy, aspect, near, far, matrix)
{
    // Fill array with zeros
    matrix.fill(0);
    // Focal length
    const f = Math.tan(fovy * Math.PI / 360.0);
    // Setup matrix
    matrix[0] = f / aspect;
    matrix[5] = f;
    matrix[10] = (far + near)     / (near - far);
    matrix[11] = (2 * far * near) / (near - far);
    matrix[14] = -1;
}
```

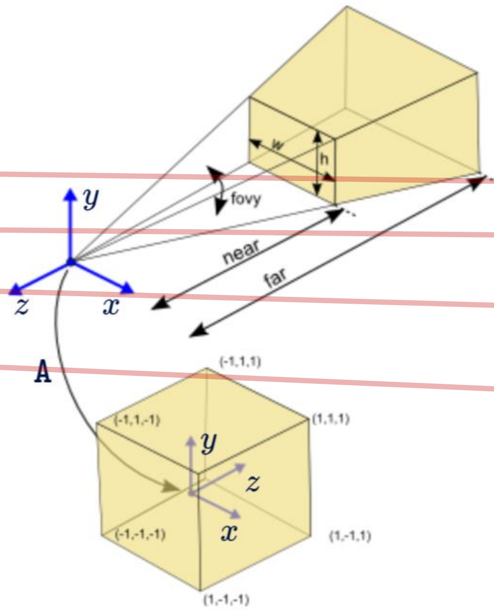
WebGL

Perspective Projection Matrix

$$\mathbf{A} = \begin{bmatrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{\text{far}+\text{near}}{\text{near}-\text{far}} & \frac{2*\text{far}*\text{near}}{\text{near}-\text{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

with $f = \text{cotan}(0.5 * \text{fovy})$

and $\text{aspect} = \text{w}/\text{h}$



```
function Perspective(fovy, aspect, near, far, matrix)
{
    // Fill array with zeros
    matrix.fill(0);
    // Focal length
    const f = Math.tan(fovy * Math.PI / 360.0);
    // Setup matrix
    matrix[0] = f / aspect;
    matrix[5] = f;
    matrix[10] = (far + near)     / (near - far);
    matrix[11] = (2 * far * near) / (near - far);
    matrix[14] = -1;
}
```

WebGL

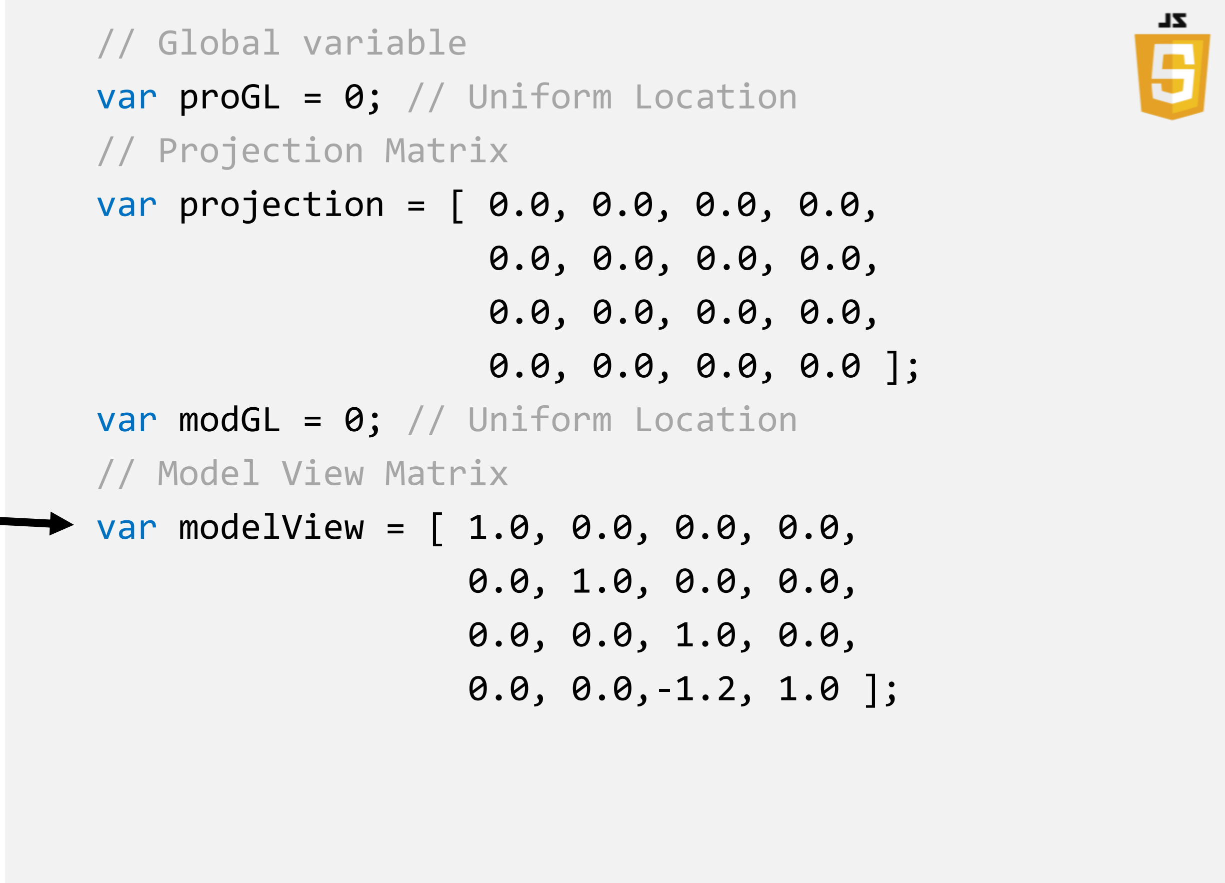
Tilføj nye variabler som globale variabler.

```
// Global variable
var proGL = 0; // Uniform Location
// Projection Matrix
var projection = [ 0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0 ];
var modGL = 0; // Uniform Location
// Model View Matrix
var modelView = [ 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  0.0, 0.0,-1.2, 1.0 ];
```

WebGL

Model View er en transformerings matrix som kan flytte og rotere viewporten (kameraet)

Transformering af ”kamera”



```
// Global variable
var proGL = 0; // Uniform Location
// Projection Matrix
var projection = [ 0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0 ];
var modGL = 0; // Uniform Location
// Model View Matrix
var modelView = [ 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  0.0, 0.0,-1.2, 1.0 ];
```

WebGL

Hvis vi flytter kamera langs z aksen vil vi få en zoom effekt.

Ulempe: Risiko for "near plane clipping" af geometri.

Dolly Zoom:
Flyt kamera langs z
aksen

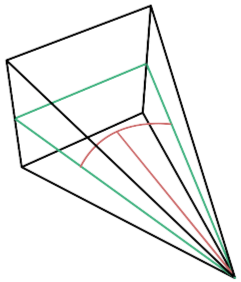
```
// Global variable
var proGL = 0; // Uniform Location
// Projection Matrix
var projection = [ 0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0 ];
var modGL = 0; // Uniform Location
// Model View Matrix
var modelView = [ 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  0.0, 0.0,-1.2, 1.0 ];
```

WebGL

Ændring af Field of View vil ændre projektering af vores perspektiv?

Kan vi ikke bare ændre FOV for at zoom?

fov=60 degrees

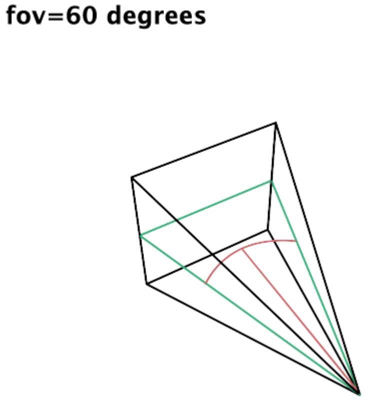


```
// Global variable
var proGL = 0; // Uniform Location
// Projection Matrix
var projection = [ 0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0 ];
var modGL = 0; // Uniform Location
// Model View Matrix
var modelView = [ 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  0.0, 0.0,-1.2, 1.0 ];
```

WebGL

Kombination af Dolly Zoom og ændring af FOV vil skabe en Vertigo effekt.

+



=

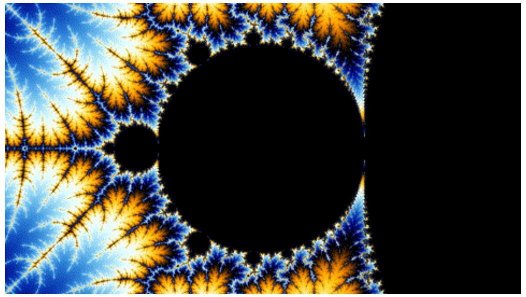
Vertigo Effekt

WebGL

Kan skalering af en transformations matrix bruges som zoom effekt?

Fordel: ingen "near plane clipping" af geometri.

Skalering (Zoom?)



```
// Global variable
var proGL = 0; // Uniform Location
// Projection Matrix
var projection = [ 0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0 ];
var modGL = 0; // Uniform Location
// Model View Matrix
var modelView = [ 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  0.0, 0.0,-1.2, 1.0 ];
```

WebGL

Model View matrixen er en transformations matrix.

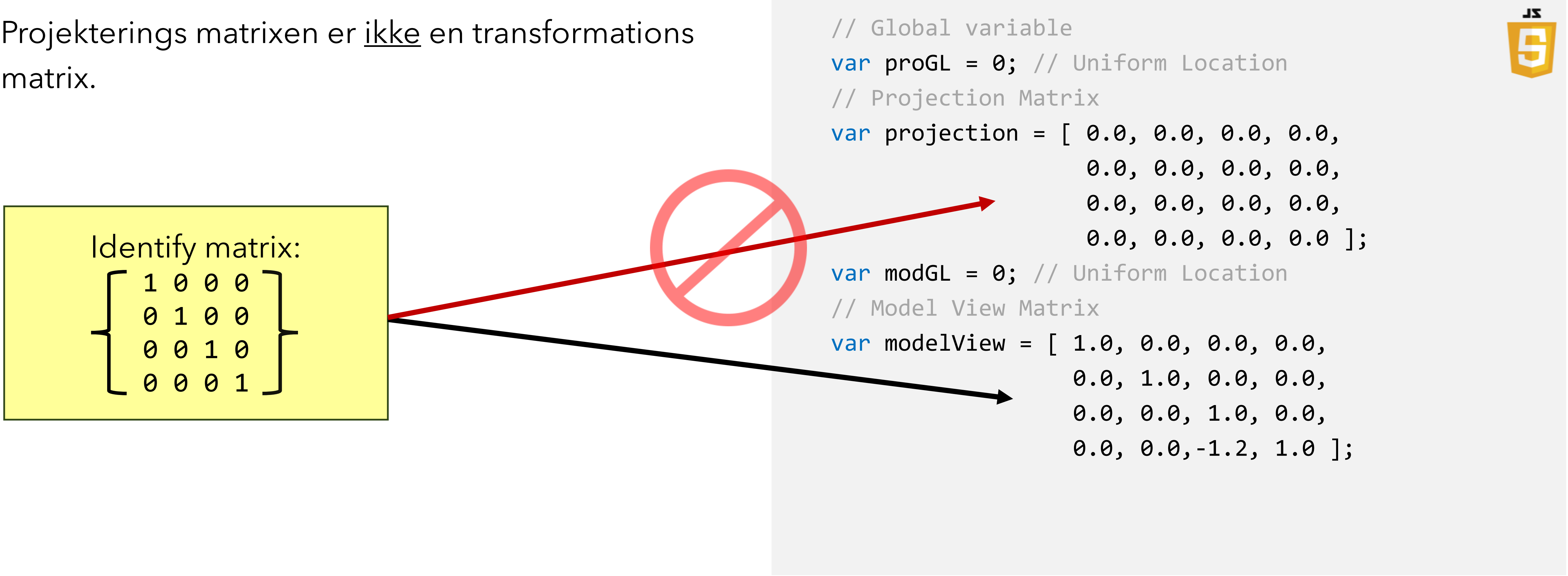
Identify matrix:

$$\left\{\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}\right\}$$

```
// Global variable
var proGL = 0; // Uniform Location
// Projection Matrix
var projection = [ 0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0 ];
var modGL = 0; // Uniform Location
// Model View Matrix
var modelView = [ 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  0.0, 0.0,-1.2, 1.0 ];
```

WebGL

Projekterings matrixen er ikke en transformations matrix.



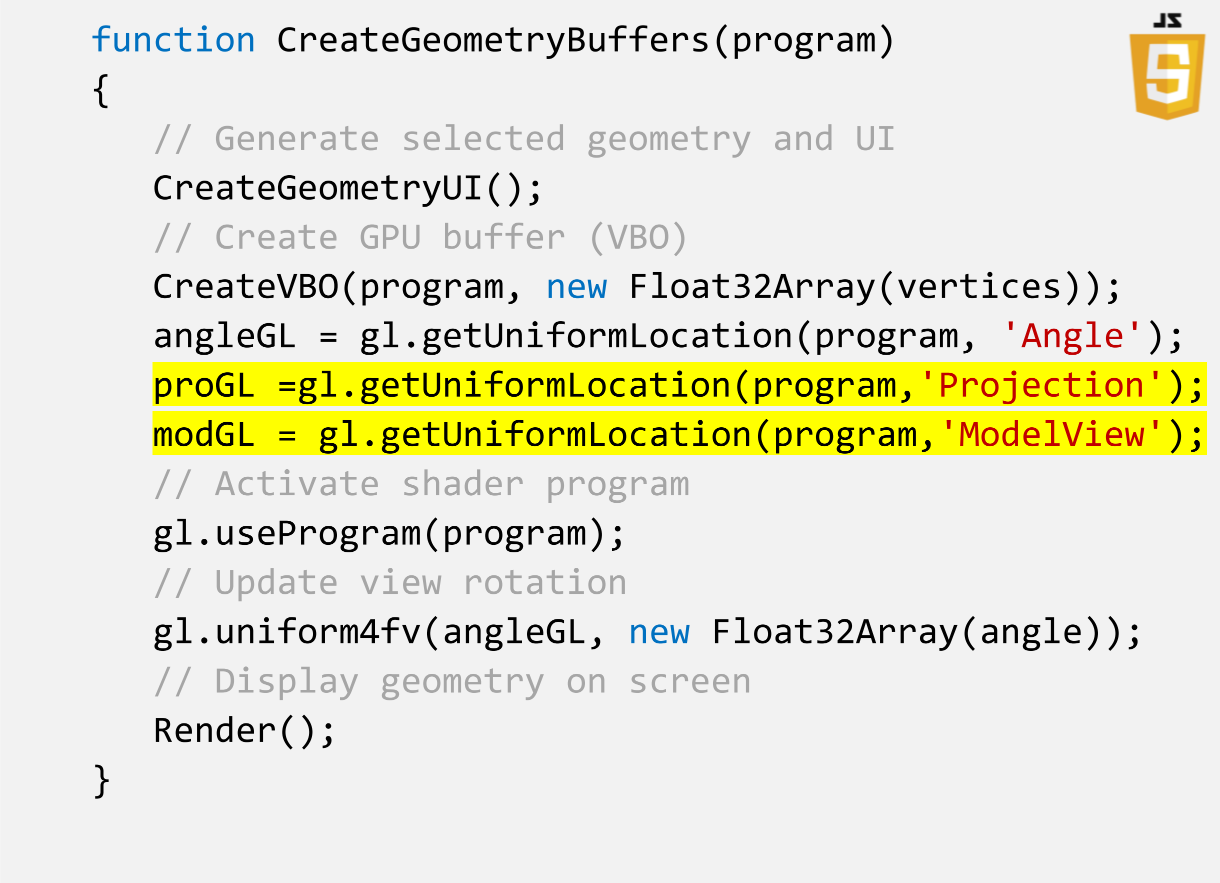
Identify matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
// Global variable
var proGL = 0; // Uniform Location
// Projection Matrix
var projection = [ 0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 0.0 ];
var modGL = 0; // Uniform Location
// Model View Matrix
var modelView = [ 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  0.0, 0.0,-1.2, 1.0 ];
```

WebGL

Fortæl grafikkortet at vi vil havde to uniform lokationer i funktionen CreateGeometryBuffers(), som kan identificeres i vertex shaderen.



```
function CreateGeometryBuffers(program)
{
   // Generate selected geometry and UI
   CreateGeometryUI();
   // Create GPU buffer (VBO)
   CreateVBO(program, new Float32Array(vertices));
   angleGL = gl.getUniformLocation(program, 'Angle');
   proGL =gl.getUniformLocation(program,'Projection');
   modGL = gl.getUniformLocation(program,'ModelView');
   // Activate shader program
   gl.useProgram(program);
   // Update view rotation
   gl.uniform4fv(angleGL, new Float32Array(angle));
   // Display geometry on screen
   Render();
}
```

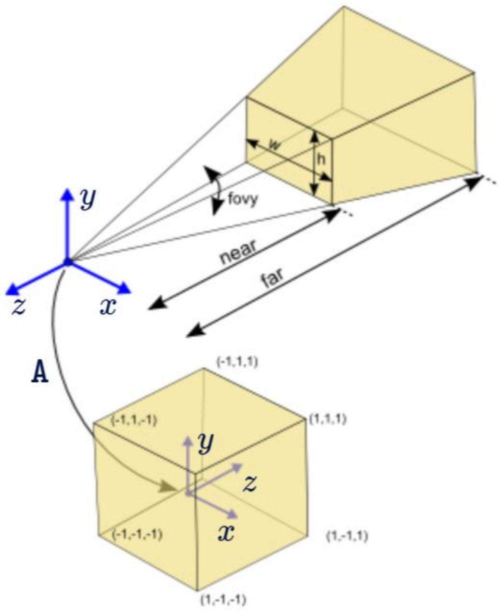
WebGL

Perspective Projection Matrix

$$\mathbf{A} = \begin{bmatrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{\text{far}+\text{near}}{\text{near}-\text{far}} & \frac{2*\text{far}*\text{near}}{\text{near}-\text{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

with $f = \text{cotan}(0.5 * \text{fovy})$

and aspect $= \text{w}/\text{h}$



```
function Perspective(fovy, aspect, near, far) {
    // Fill array with zeros
    projection.fill(0);
    // Focal length
    const f = Math.tan(fovy * Math.PI / 360.0);
    // Setup matrix
    projection[0] = f / aspect;
    projection[5] = f;
    projection[10] = (far + near)     / (near - far);
    projection[11] = (2 * far * near) / (near - far);
    projection[14] = -1;
    gl.uniformMatrix4fv(proGL, false, new Float32Array(
projection));
    gl.uniformMatrix4fv(modGL, false, new Float32Array(
modelView));
}
```

WebGL

Opdater Render() funktionen.

```
function Render()
{
   gl.clearColor(0.0, 0.4, 0.6, 1.0);
   gl.clear(gl.COLOR_BUFFER_BIT |
            gl.DEPTH_BUFFER_BIT );
   // Dolly Zoom
   const zoom = document.getElementById('zoom').value;
   modelView[14] = -zoom;
   // Perspective Projection
   const fov = document.getElementById('fov').value;
   const aspect = gl.canvas.width / gl.canvas.height;
   Perspective(fov, aspect, 1.0, 2000.0);
  // Draw Geometry
  gl.drawArrays(gl.TRIANGLES, 0, vertices.length /11);
}
```

WebGL

Tilføj to HTML **number input** elementer:

Field of View:

```
<input type="number" id="fov" value="90"
onchange="Render();">
```

Zoom:

```
<input type="number" id="zoom" value="1.2"
onchange="Render();">
```

```
body onload="InitWebGL();">
    <canvas id="gl" width="800px" height="600px">
         WebGL is not supported!
    </canvas>
    <textarea id="vs" spellcheck="false"></textarea>
    <textarea id="fs" spellcheck="false"></textarea>
    <select id="shape" onchange="InitShaders();">
         <option selected>Triangle</option>
         <option>Quad</option>
    </select><br>
    <input type="number" id="fov" value="90"
onchange="Render();"> FOV<br><input type="number"
id="zoom" value="1.2" onchange="Render();"> Zoom<br>
    <div id="ui">Generate UI for geometry here!</div>
    <script src="js/webgl.js" defer></script>
</body>
```

WebGL

Tilføje uniform matrixer: **Projection** og **ModelView** til vertex shaderen.

Jeg har omdøbt matX og matY til mX og mY, for at få plads til koden på sliden.

```
attribute vec3 Pos;
attribute vec3 Color;
uniform vec4 Angle;
uniform mat4 Projection;
uniform mat4 ModelView;
varying vec3 vertexColor;
void main() {
    float coX = cos(Angle.x);
    float siX = sin(Angle.x);
    mat4 mX = mat4(vec4(1.0, 0.0, 0.0, 0.0),
                   vec4(0.0, coX, siX, 0.0),
                   vec4(0.0,-siX, coX, 0.0),
                   vec4(0.0, 0.0, 0.0, 1.0));
    float coY = cos(Angle.y);
    float siY = sin(Angle.y);
    mat4 mY = mat4(vec4(coY, 0.0,-siY, 0.0),
                   vec4(0.0, 1.0, 0.0, 0.0),
                   vec4(siY, 0.0, coY, 0.0),
                   vec4(0.0, 0.0, 0.0, 1.0));
    gl_Position = Projection * ModelView * mY * mX *vec4(Pos,1.0);
    vertexColor = Color;
}
```

WebGL

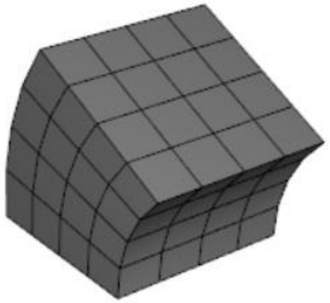
Vi kan lave vertex shaderen mere performance venligt ved at omdanne mX og mY til uniform mat4.

Det betyder at vi skal lave rotations matrixerne i vores JavaScript kode og opdater for dem i vores mousemove event listener.

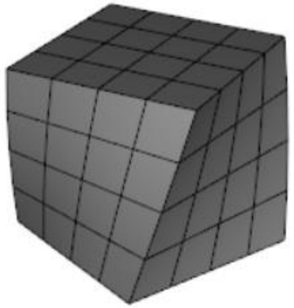
Er der nogen der kan fortælle mig, hvorfor denne måde er mere performance venligt?

```
attribute vec3 Pos;
attribute vec3 Color;
uniform mat4 Projection;
uniform mat4 ModelView;
uniform mat4 mY;
uniform mat4 mY;
varying vec3 vertexColor;
void main()
{
  gl_Position = Projection * ModelView * mY * mX * vec4(Pos, 1.0);
  vertexColor = Color;
}
```

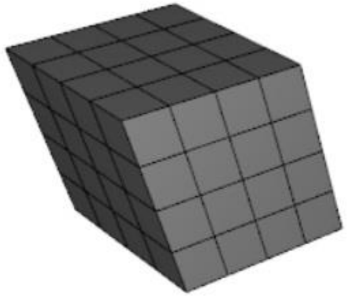
Modifiers



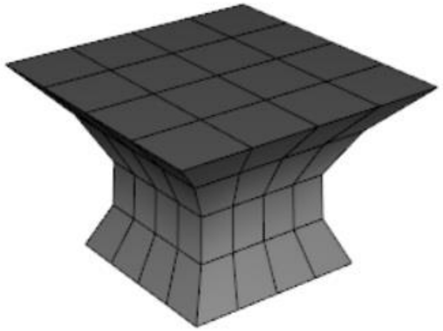
Bend



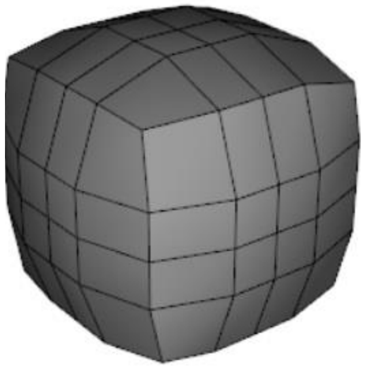
Twist



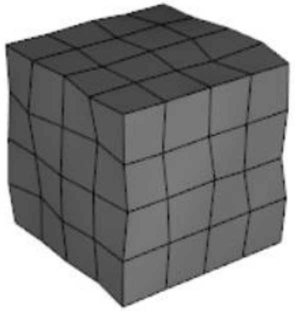
Skew



Taper



Bulge



Noise

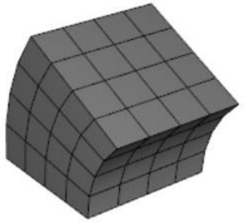
WebGL

Bend modifieren er en meget simpel modifier, der roterer vertexer omkring et given akse.

Vi kan implementer den som vist i vertex shaderen.

Hvem kan fortælle hvad koden gør, og hvorfor det kun virker optimalt for geometri der er præcist 1 unit høj?

Hvordan kan det implementeres på en bedre og mere dynamisk måde?



Bend

```
attribute vec3 Pos;
attribute vec3 Color;
uniform mat4 Projection;
uniform mat4 ModelView;
uniform mat4 mY;
uniform mat4 mY;
varying vec3 vertexColor;
void main()
{
    float coZ = cos(Position.y + 0.5);
    float siZ = sin(Position.y + 0.5);
    mat4 bend = mat4(vec4( coZ, siZ, 0.0, 0.0),
                     vec4(-siZ, coX, 0.0, 0.0),
                     vec4( 0.0, 0.0, 1.0, 0.0),
                     vec4( 0.0, 0.0, 0.0, 1.0));
    vec4 v = bend * vec4(Pos, 1.0);
    gl_Position = Projection * ModelView * mY * mX * v;
    vertexColor = Color;
}
```

WebGL

Øvelse: **Perspektiv & Modifiers**

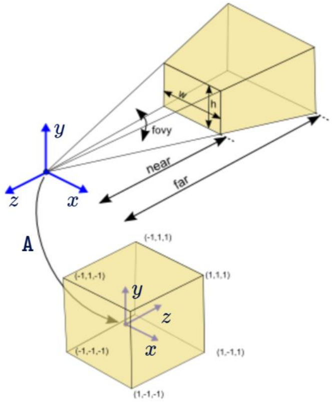
1) Implementer perspektiv projekterings matrix som anvist i denne præsentation.

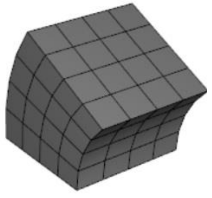
2) Implementer Bend modifieren og lav dernæst <u>dine egne modifers,</u> som kan bruges til MSU1 afleveringen. Generer tilhørende UI ved brug af HTML elementer, der gør det muligt at ændrer på modifieren.

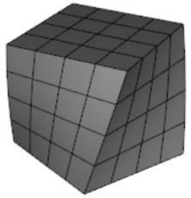
$$\mathtt{A} = \begin{bmatrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{\text{far}+\text{near}}{\text{near}-\text{far}} & \frac{2*\text{far}*\text{near}}{\text{near}-\text{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

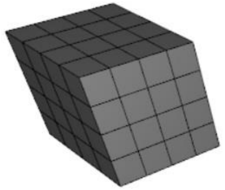
with $f = \text{cotan}(0.5 * \text{fovy})$

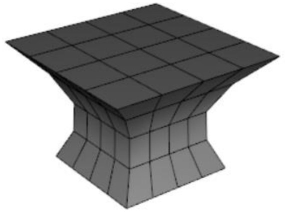
and $\text{aspect} = \text{w}/\text{h}$




Bend


Twist


Skew


Taper