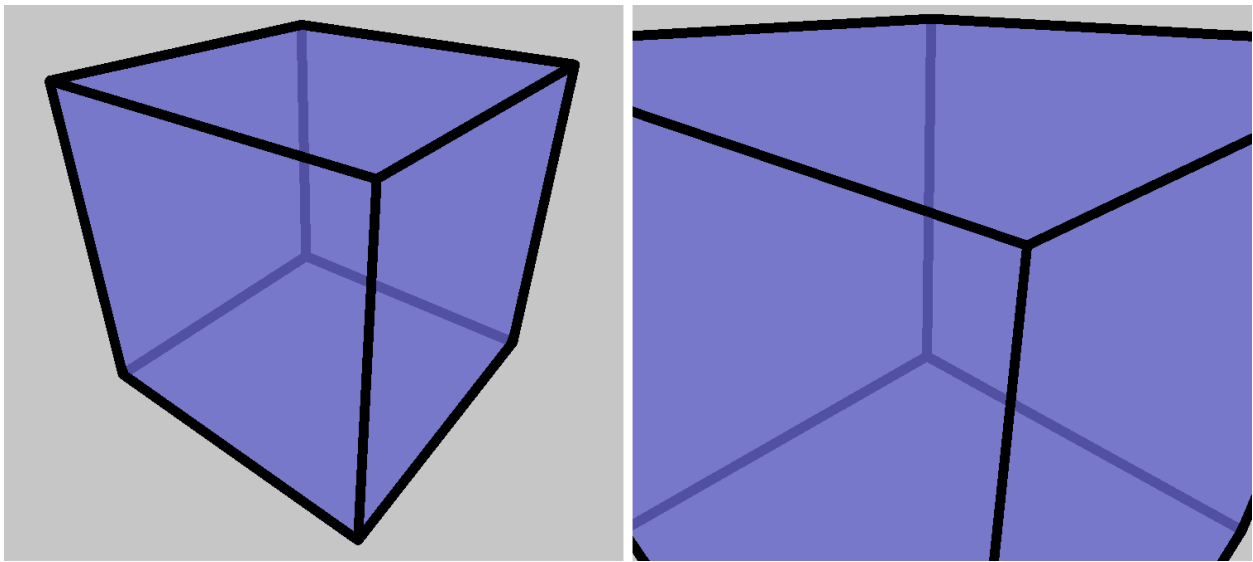
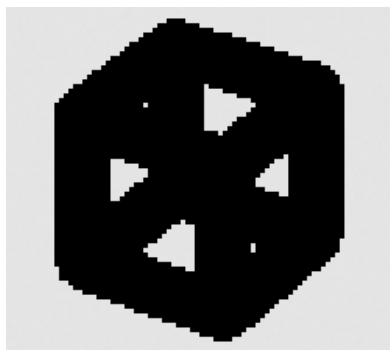


Depth Based Edge Thickness

In the original technique, all edges and caps were drawn with widths defined by the normalized perpendicular vector, and optionally, a scaling factor. Since the perpendicular is a screen-space value, the edges' drawn thickness will remain screen-space dependent. As the object is zoomed in and out, the width of the edge for a given object will remain the same width.



This effect may be desirable. However, it can lead to distant objects having edges that overpower the entire object.

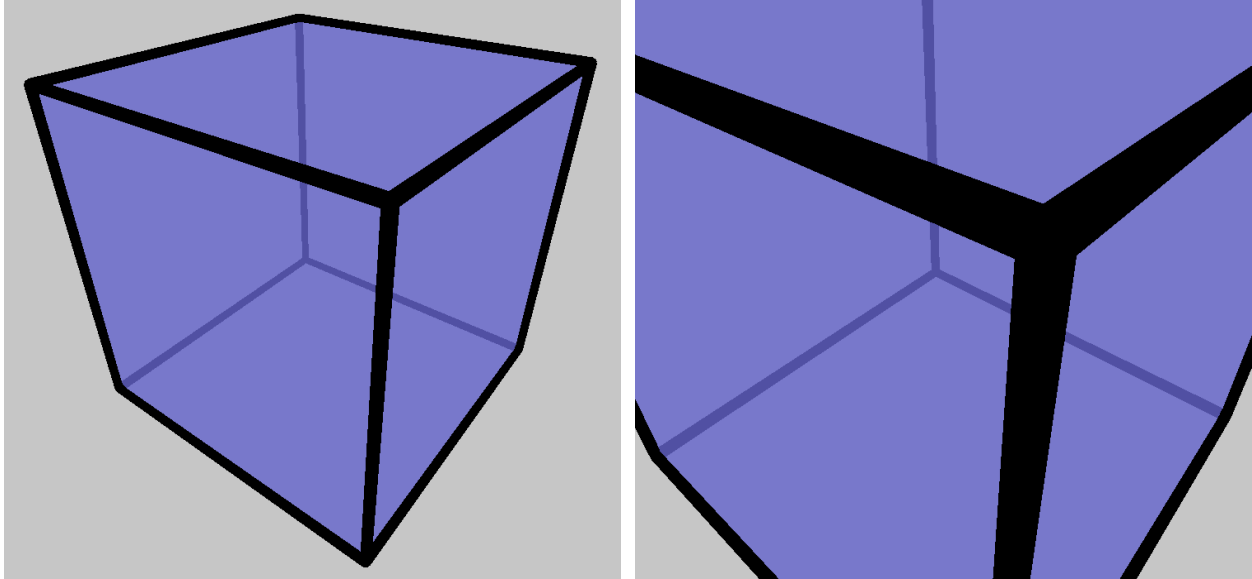


To allow for more realistically scaled edges and prevent the overpowering problem, the perpendicular vector of each edge point must be scaled by the distance between it and the camera. The reason both points' distance from the camera must be found is so that the edge thicknesses will be equal to similar points on other edges, allowing all edges to have the same thickness at their points of connectivity.

Naïve Approach

The naïve approach involves multiplying the world-space position of the points by the model and view matrices, shifting the points into view-space. Once there, their z depth coordinate is stored in terms of their distance along the negative z axis. To allow this value to be used as a scaling factor, the z value must be negated to make it positive, and inverted to allow the use of the scaling factor through multiplication. Then, a standard scaling factor can be multiplied in to modify the overall thickness at a given distance, and all other distances will receive proportional depth-based widths. This function is represented below:

```
float ThicknessScale(vec4 worldPoint)
{
    return standardScalingFactor / -(ModelViewMatrix * worldPoint).z;
}
```



Perspective Projection Optimization

There is an optimization if using perspective projection. The projection-space version of each edge point is already known because they are needed to calculate the screen-space vectors and other values used when making the edges. The inverse projection matrix is:

$$\begin{bmatrix} \frac{r}{n} & 0 & 0 & 0 \\ 0 & \frac{t}{n} & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{(n-f)}{(2fn)} & \frac{(n+f)}{(2fn)} \end{bmatrix}$$

inverse perspective matrix

Only the z component is needed, so sending the projected point from projection-space to view-space reduces to:

$$\text{depth} = 0 * (\text{projPoint.x} + \text{projPoint.y} + \text{projPoint.z}) + (-1 * \text{projPoint.w})$$

$$\text{depth} = -\text{projPoint.w}$$

Since "-projPoint.w" is a replacement for the "(ModelViewMatrix * worldSpacePoint).z;" plugging it into the above equation, the final optimized function becomes:

```
float ThicknessScale(vec4 projPoint)
{
    return standardScalingFactor / projPoint.w;
}
```

These thickness values are used as one would use screen-space scale factors: multiplying them into the perpendicular vector part of the output point calculations. Remember that for edges, both the s0 and s1 points must be used to create two separate thickness scale factors, whereas caps will only require one.

Orthographic Projection Considerations

Such a compact optimization is not available when using orthographic projection. After applying the same rules to the inverted orthographic projection matrix, the function becomes:

```
float ThicknessScale(vec4 projPoint)
{
    return (-2 * standardScalingFactor) / (projPoint.z * (n - f) -
        projPoint.w * (n + f));
}
```

Where n is the near plane distance and f is the far plane distance.

The amount of calculation is not significantly less than accessing the value through world-space vector multiplication with the ModelView matrix, and requires two additional pieces of data: the near and far plane distances. However, if the ModelView matrix is

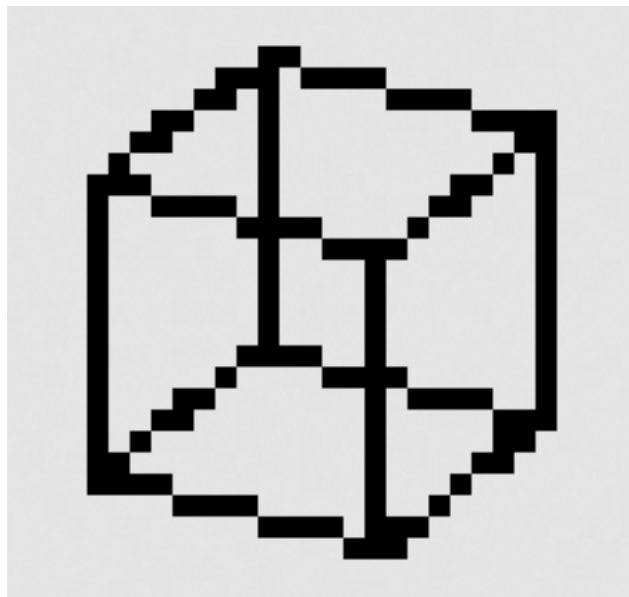
not available as a separate matrix from the ModelViewProjection matrix, this is an alternative method of retrieving the thickness.

Bottom Limit

One final step is to cap the bottom of the scale through the use of a max function. This will assure a minimum edge thickness no matter the distance of the object. Below, the function's minimum value is one, which will assure that the edge will always appear at least one pixel thick. The function becomes:

```
float ThicknessScale(vec4 projPoint)
{
    return MAX(1.0, standardScalingFactor / projPoint.w);
}
```

After zooming out, the object will look like this after magnification:



Edges with a minimum thickness of less than one should be avoided because they lead to flickering and disappearing edges.