INFOGR – Computer Graphics

J. Bikker - April-July 2016 - Lecture 9: "Transformations"

Welcome!

Today's Agenda:

- Affine Transforms
- Projection
- Pipeline Recap
- Rasterization

Transforms

Spaces – Translation

Translation is not a linear transform.

With linear transforms, we get:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{pmatrix}$$

But we need something like:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + x_t \\ y + y_t \end{pmatrix}$$

We can do this with a combination of linear transformations and translations called *affine transformations*.

Transforms

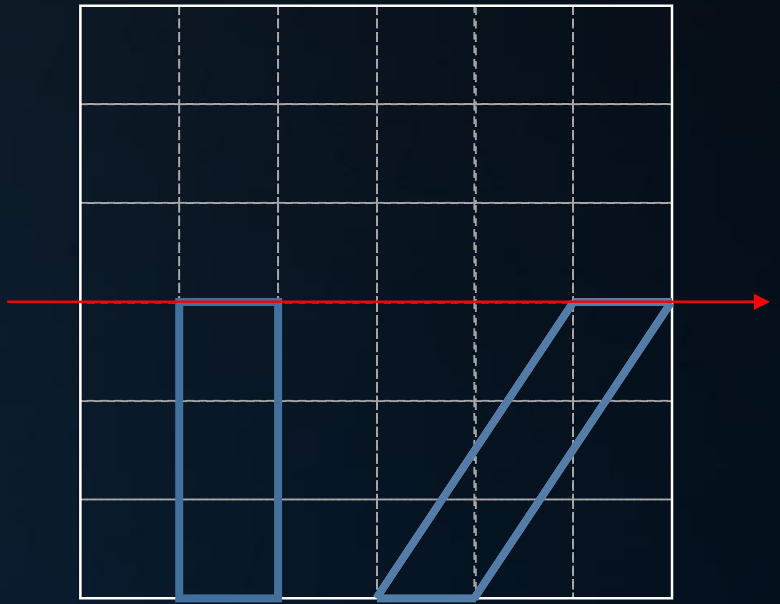
Spaces – Translation

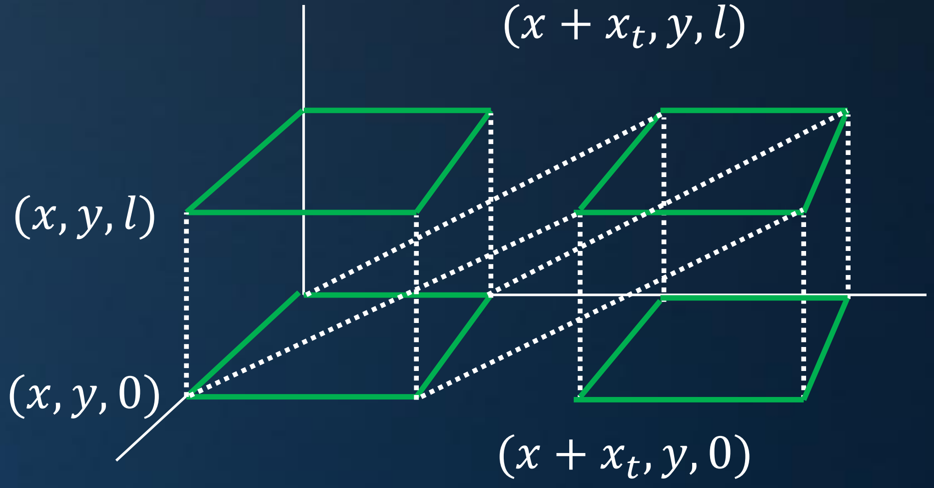
Observe: in 2D, shearing “pushes things sideways” (e.g., in the x direction), in a “fixed level” (the y value).

We are thus performing a translation in a 1D subspace (a line), using matrix multiplication in 2D.

In 3D, shearing leads to translation in a 2D subspace, i.e. a plane.

$$A = \begin{pmatrix} 1 & \frac{2}{3} \\ 0 & 1 \end{pmatrix};\ A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + \frac{2}{3}y \\ y \end{pmatrix}$$





Transforms

Spaces – Translation

By adding a 3rd dimension to 2D space, we can use matrix multiplication to do translation.

$$M\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x + x_t \\ y + y_t \\ z \end{pmatrix}$$

But: what does matrix M look like? What about x_t and y_t?
And how do we deal with the third coordinate z?

Transforms

Spaces – Translation

Shearing in 3D based on the z coordinate is a simple generalization of 2D shearing:

$$\begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1x + 0y + x_t z \\ 0x + 1y + y_t z \\ 0x + 0y + 1z \end{pmatrix} = \begin{pmatrix} x + x_t z \\ y + y_t z \\ z \end{pmatrix}$$

The final step is to set z to 1.

$$\begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + x_t \\ y + y_t \\ 1 \end{pmatrix}$$

Transforms

Spaces – Translation

Translations in 2D can be represented as shearing in 3D by looking at the plane $z = 1$.

By representing our 2D points (x, y) by 3D vectors $(x, y, 1)$, we can translate them about (x_t, y_t) by applying the following matrix:

$$\begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + x_t \\ y + y_t \\ 1 \end{pmatrix}$$

That works for points. What about vectors? We use the following transform:

$$\begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$$

Transforms

Spaces – Translation

Affine transformations (i.e., linear transformations and translations) can be done with matrix multiplication if we add *homogeneous coordinates*, i.e.

- A third coordinate $z = 1$ to each *point*: $p = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$
- A third coordinate $\mathrm{z} = 0$ to each vector: $\vec{v} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$
- A third row (0 0 1) to each matrix.

Transforms

Spaces – Translation

These concepts apply naturally to 3D, in which case we again add a homogeneous coordinate, i.e.

- A fourth coordinate $w = 1$ to each *point;*
- A fourth coordinate $w = 0$ to each vector;
- A fourth row (0 0 0 1) to each matrix.

Today's Agenda:

- Affine Transforms
- Projection
- Pipeline Recap
- Rasterization

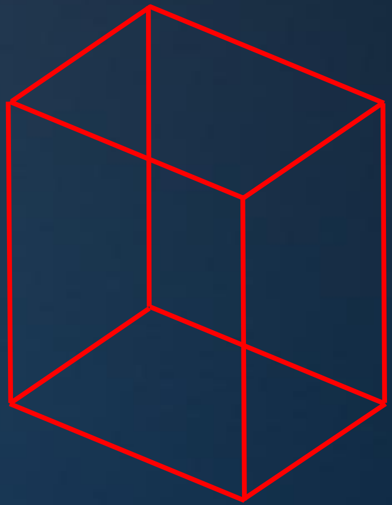
Perspective

Projection – Applying matrices, working our way backwards

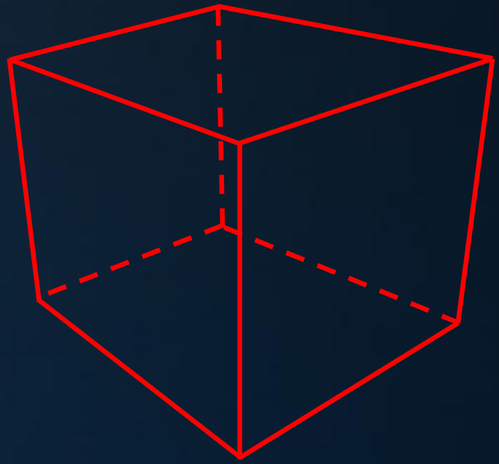
Goal: create 2D images of 3D scenes

Standard approach: linear perspective (in contrast to e.g. fisheye views)

Parallel projection:



Perspective projection:



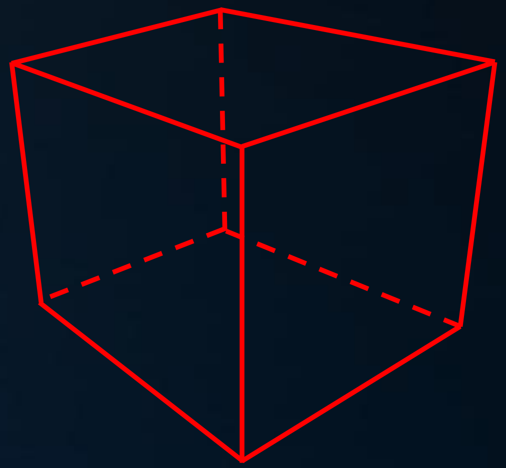
Perspective

Parallel projection:

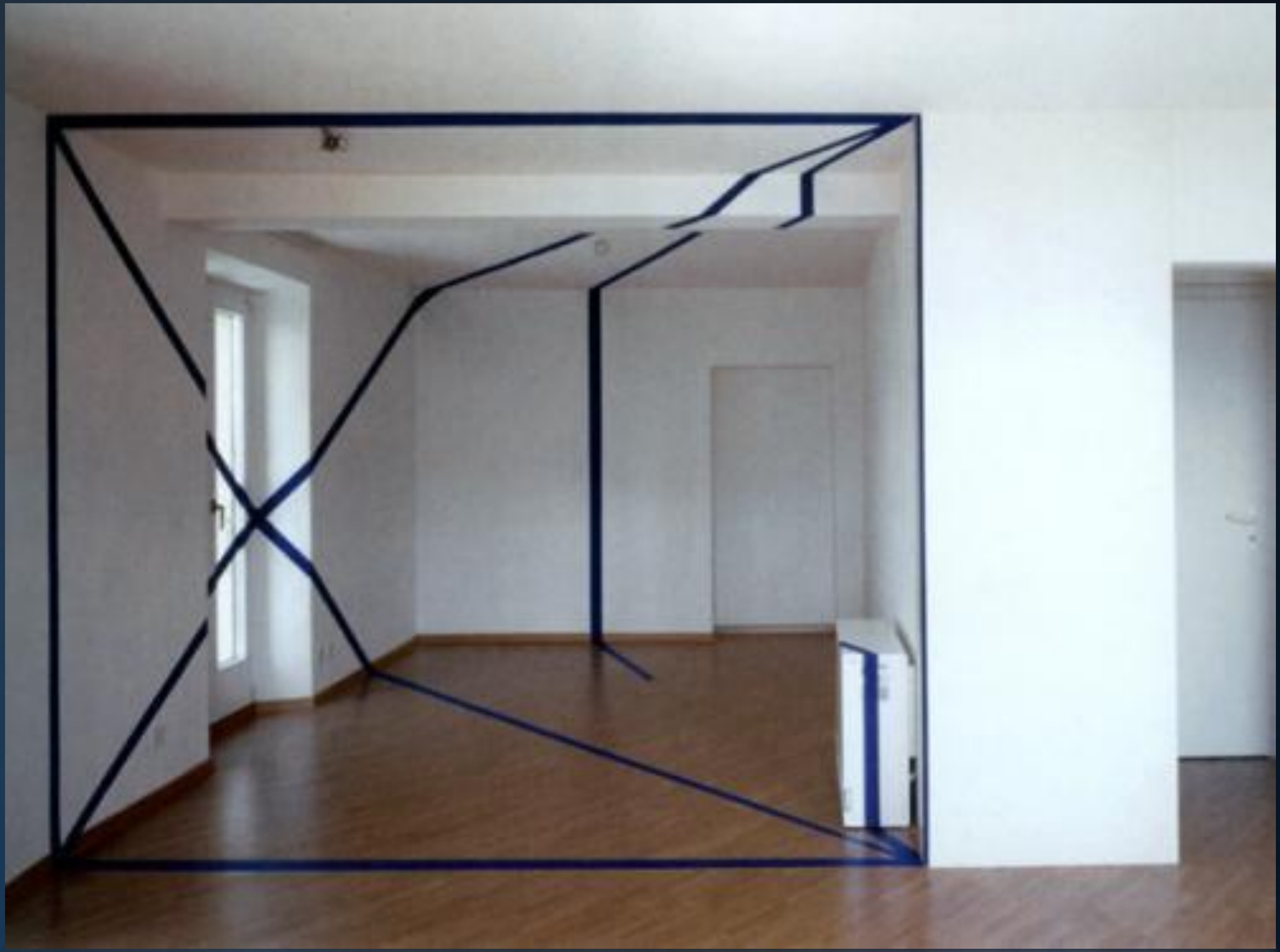
Maps 3D points to 2D by moving them along a *projection direction* until they hit an *image plane*.

Perspective projection:

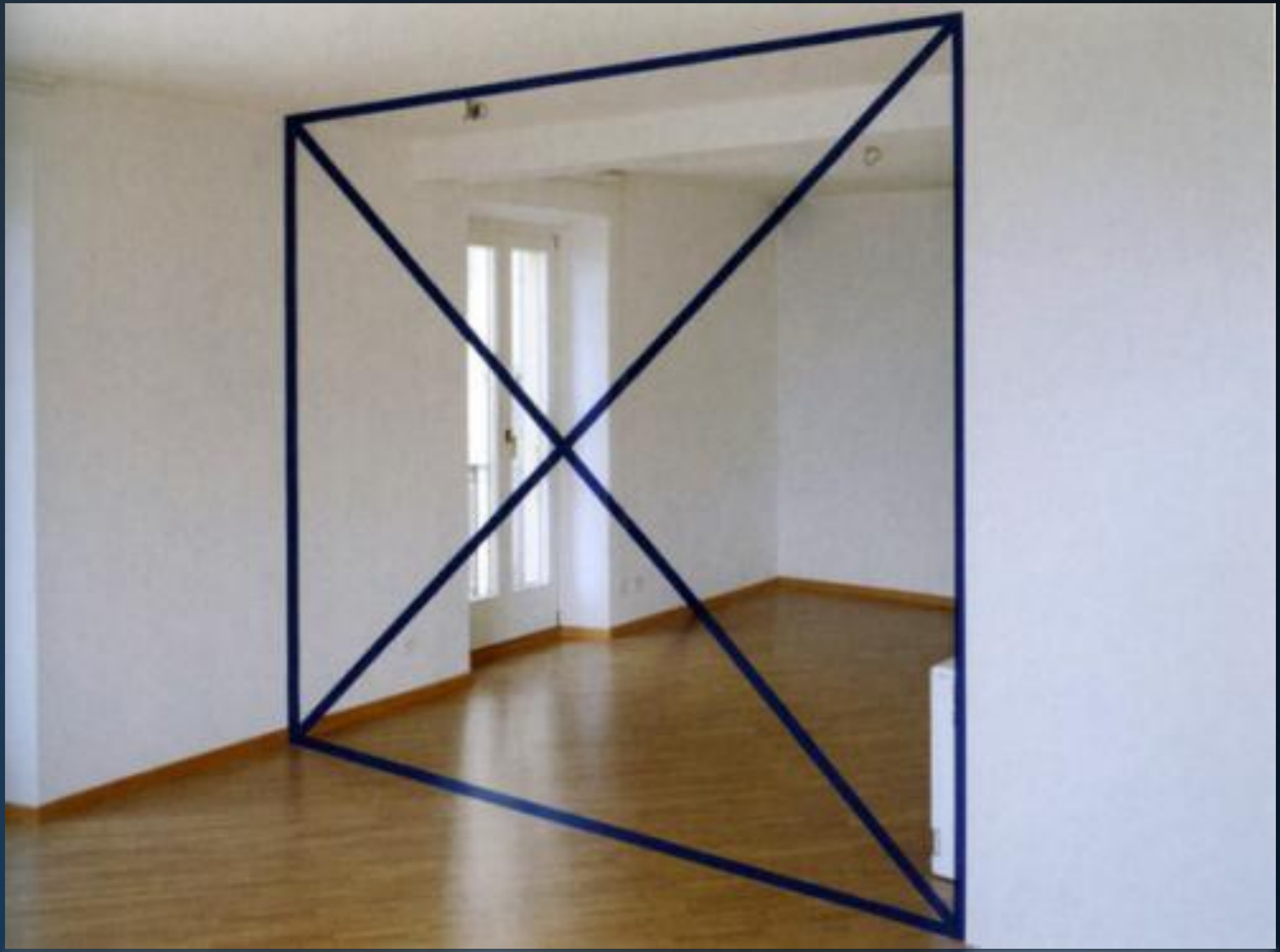
Maps 3D points to 2D by projecting them along lines that pass through a single *viewpoint* until they hit an image plane.



Perspective



Perspective



Perspective

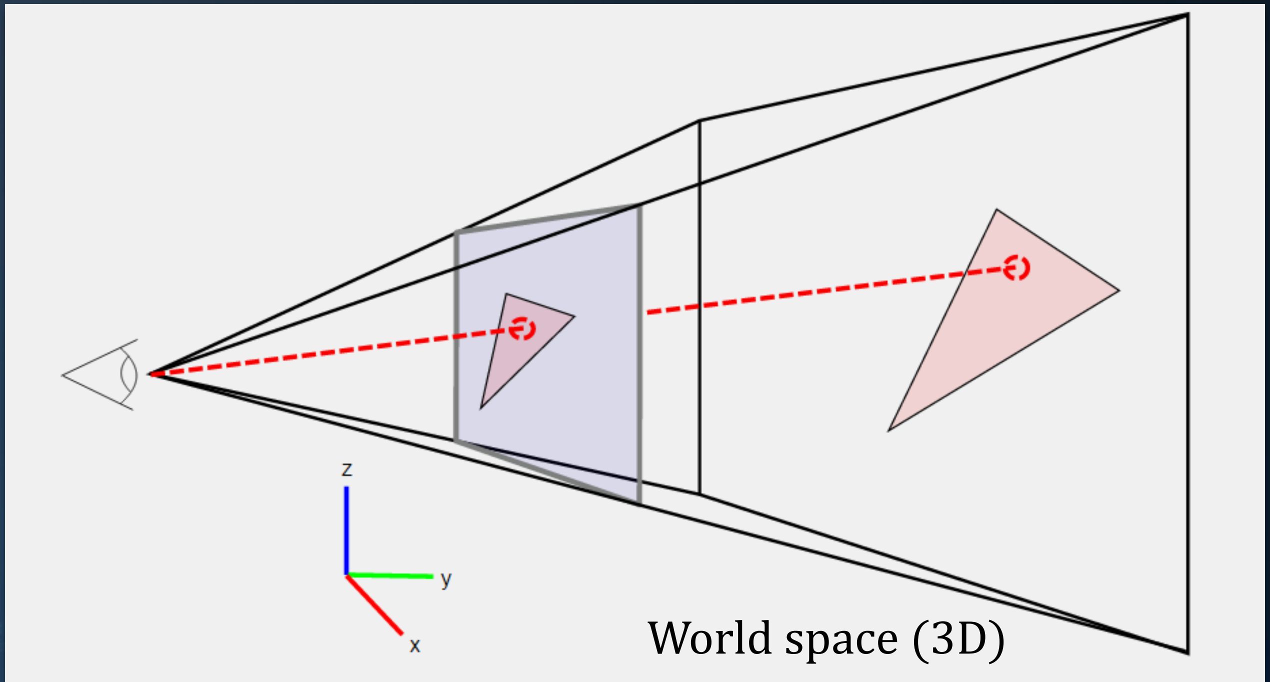
Perspective

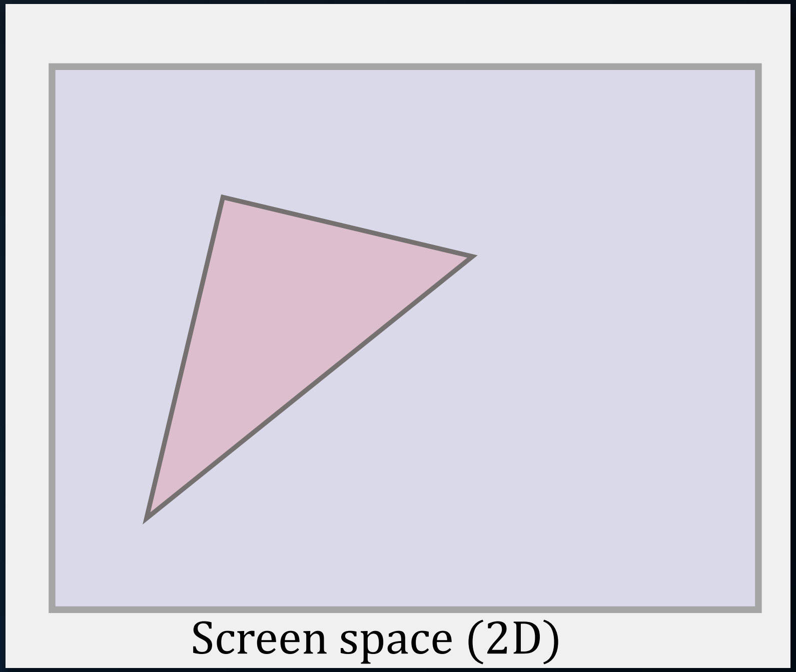
Perspective

Perspective

Perspective

Perspective projection





We get our 3D objects perspective correct on the 2D screen by applying a sequence of matrix operations.

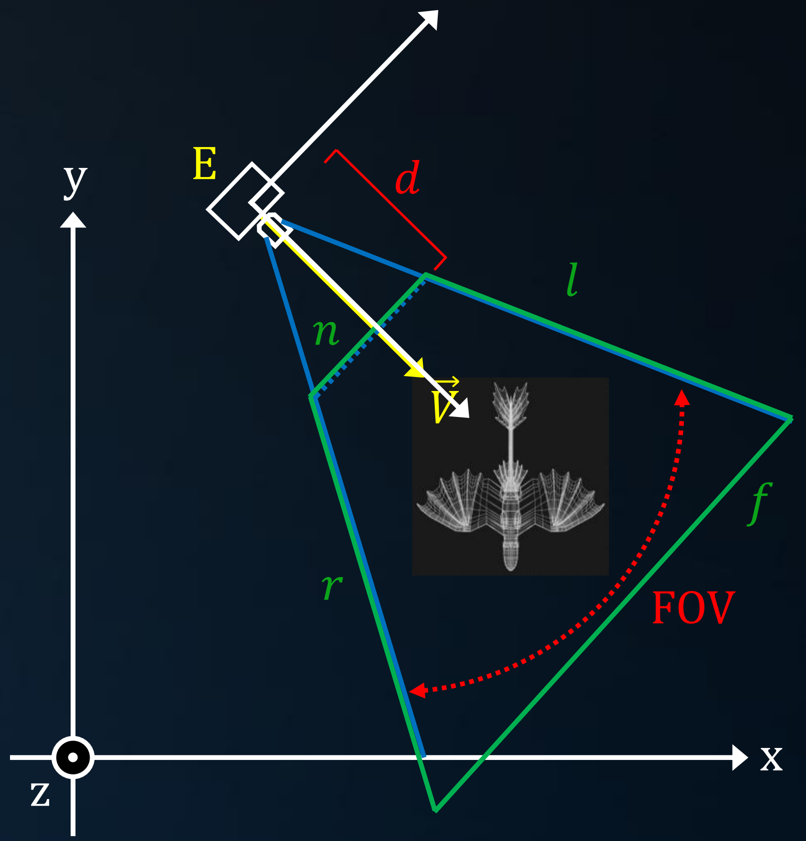
Perspective

The camera is defined by:

- Its position E
- The view direction $\vec{V}$
- The image plane (defined by its distance d, aspect ratio and the field of view)

The *view frustum* is the volume visible from the camera. It is defined by:

- A near and a far plane n and f;
- A left and a right plane l and r;
- A top and a bottom plane t and b (in 3D).



The world according to the camera:

Camera space

Perspective

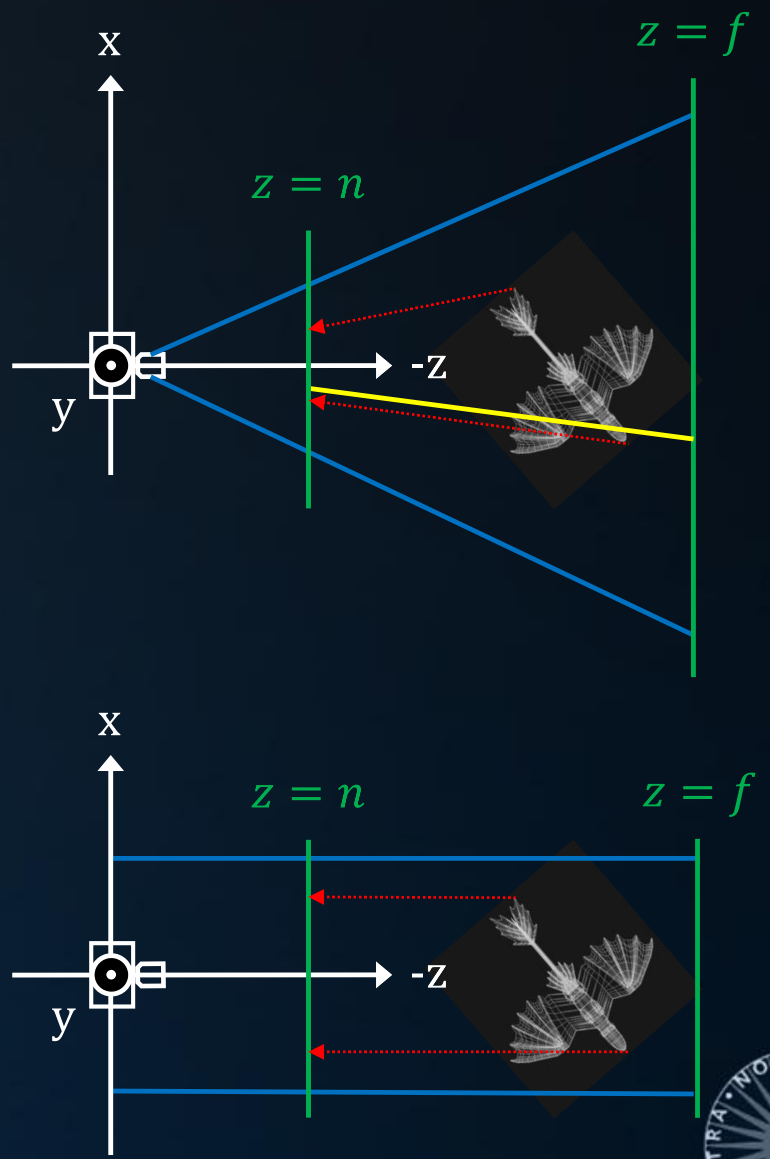
Camera space: looking down negative z.

We can now map from (x, y, z) to (x_s, y_s)
(but this mapping is not trivial)

Projection (and later: clipping) becomes easier when we switch to an *orthographic* view volume.

This time the mapping is:
$(x, y, z) \rightarrow (x, y) \rightarrow (x_s, y_s)$.

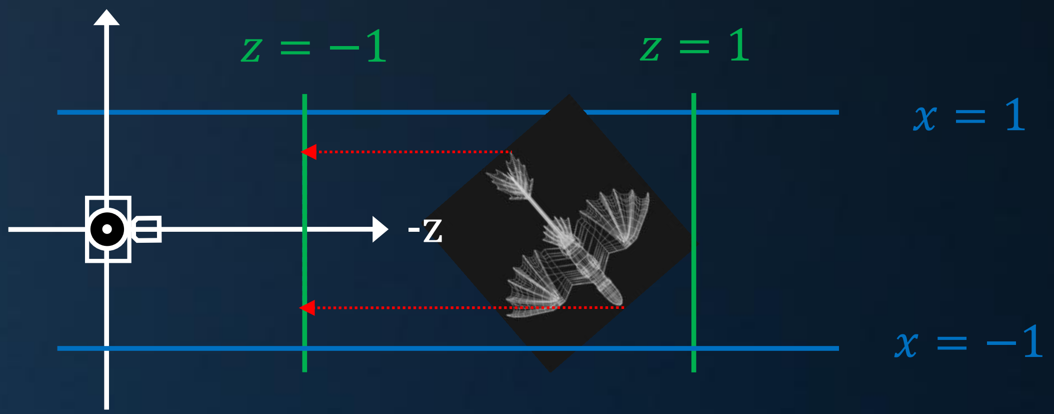
Going from camera space to the orthographic view volume can be achieved using a matrix multiplication.



Perspective

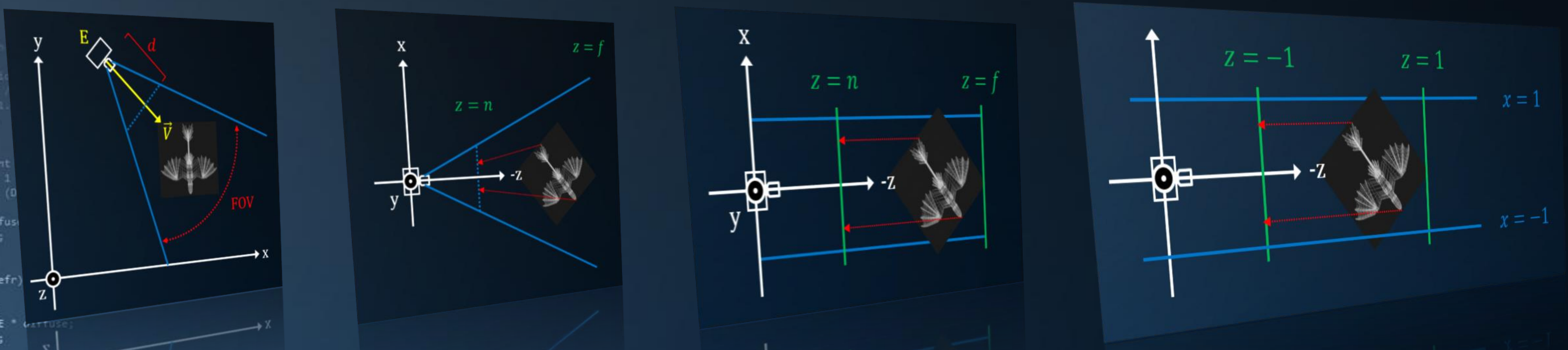
The final transform is the one that takes us from the orthographic view volume to the canonical view volume.

Again, this is done using a matrix.



Perspective

Transformation Pipeline



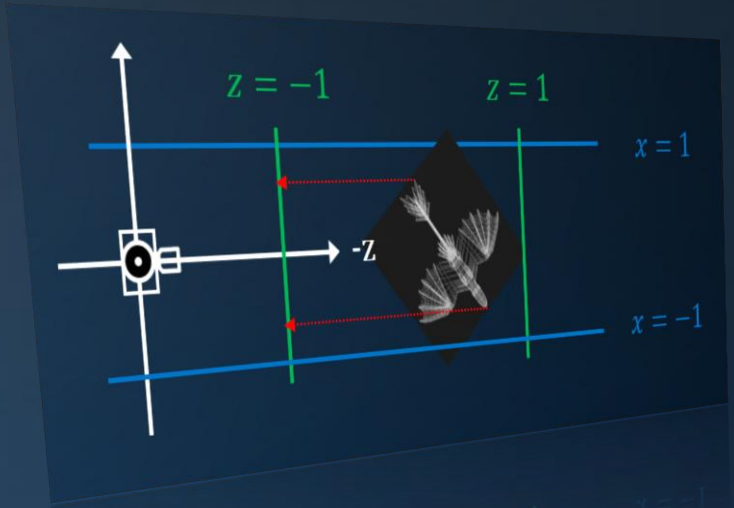
World space ➔ camera space ➔ orthographic view ➔ canonical view

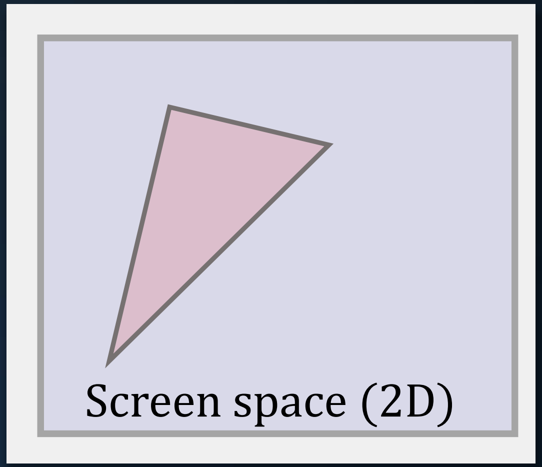
$I \times M_{camera} \times M_{ortho} \times M_{canonical}$

These can be collapsed into a single 4×4 matrix.

Perspective

Transformation Pipeline





Canonical view ➔ screen

We need one last transform:

From canonical view (-1..1) to 2D screen space ($N_x \times N_y$).

Perspective

Transformation Pipeline

STEP ONE: canonical view to screen space

Vertices in the canonical view are orthographically projected on an $n_x \times n_y$ image.

We need to map the square $[-1,1]^2$ onto a rectangle $[0, n_x] \times [0, n_y]$. Matrix:

$$\begin{pmatrix} \frac{n_x}{2} & 0 & \frac{n_x}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

This is assuming we already threw away z to get an orthographic projection. We will however combine all matrices in the end, so we actually need a 4×4 matrix:

$$M_{vp} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective

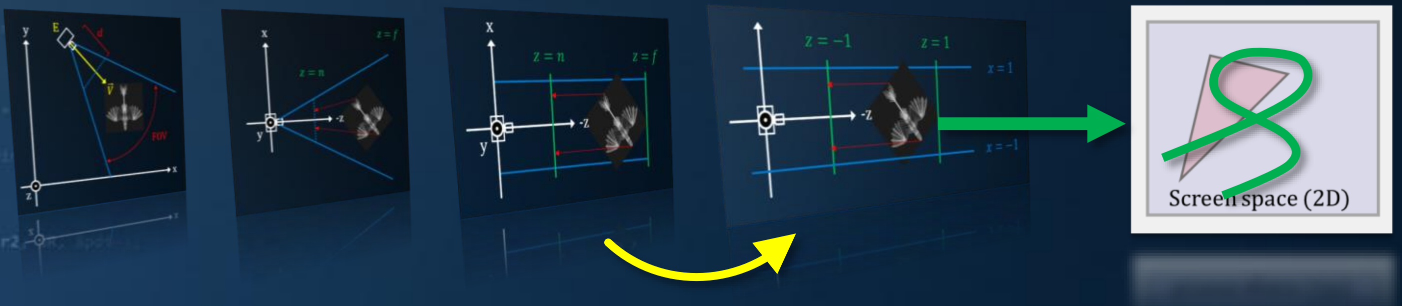
Transformation Pipeline

STEP ONE: canonical view to screen space

We now know the final transform for the vertices:

$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} \begin{pmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{pmatrix}$$

Next step: getting from the orthographic view volume to the canonical view volume.



Perspective

Transformation Pipeline

STEP TWO: orthographic view volume to canonical view volume

The orthographic view volume is an axis aligned box $[l, r] \times [b, t] \times [n, f]$. We want to scale this to a $2 \times 2 \times 2$ box centered around the origin.

Scaling to [-1,1]: Moving the center to the origin: Combined:

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

Perspective

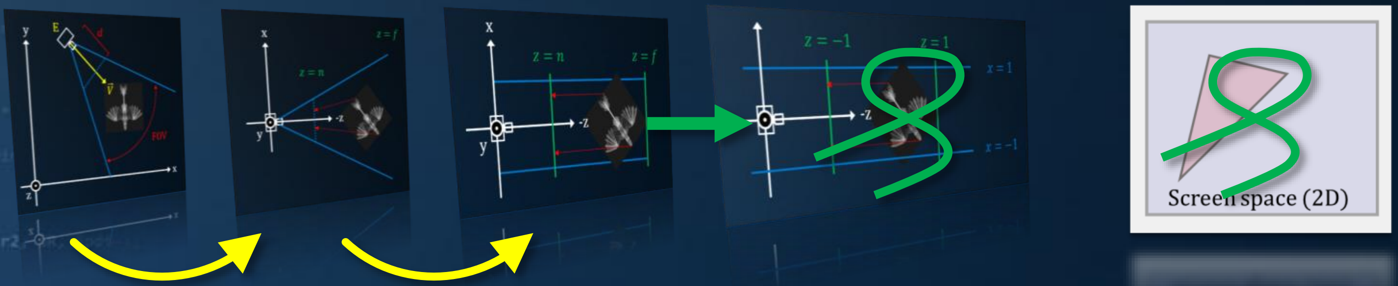
Transformation Pipeline

STEP TWO: orthographic view volume to canonical view volume

The final transforms for the vertices are thus:

$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{canonical} \begin{pmatrix} x_{ortho} \\ y_{ortho} \\ z_{ortho} \\ 1 \end{pmatrix}$$

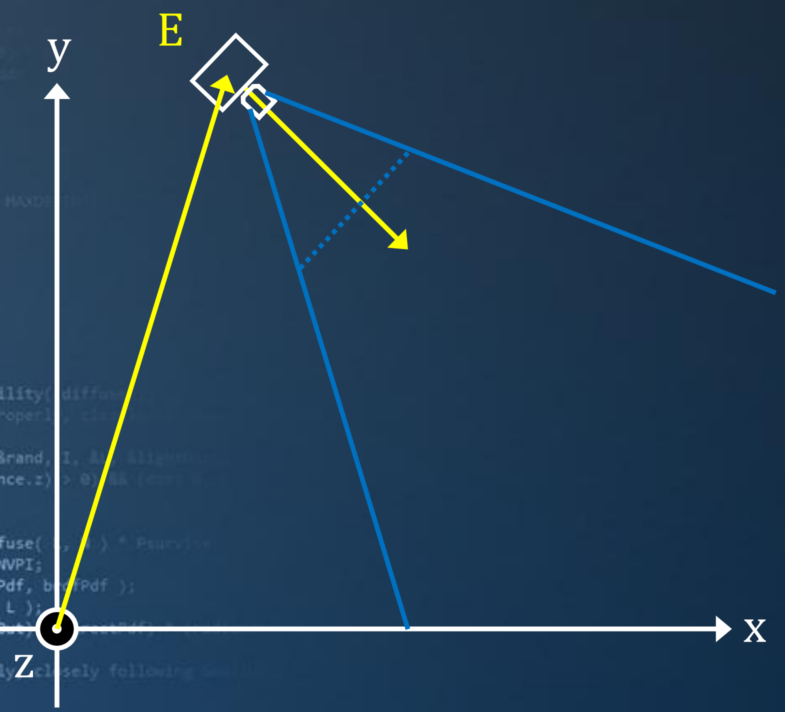
Next step: getting from camera space to the orthographic view volume.

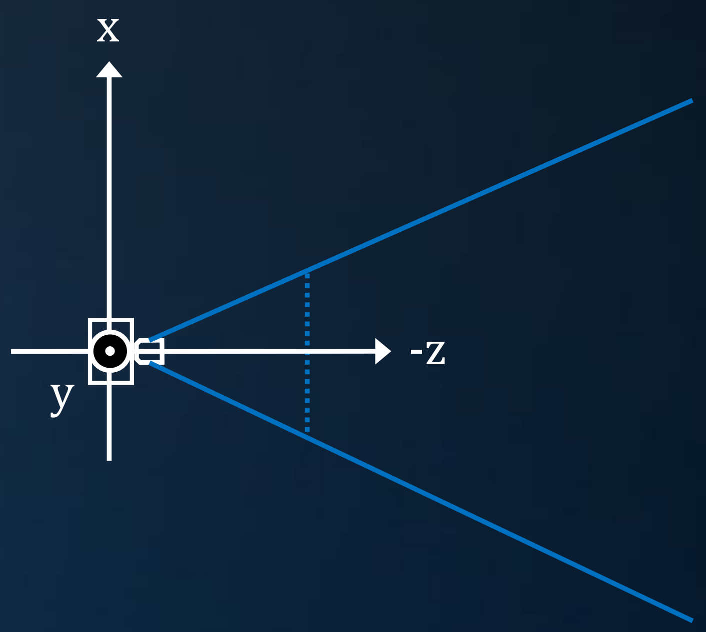


Perspective

Transformation Pipeline

STEP THREE: camera space to orthographic view volume





Translate:

$$\begin{pmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

i.e., the inverse of the camera translation.

Rotate:

We will use the inverse of the basis defined by the camera orientation.

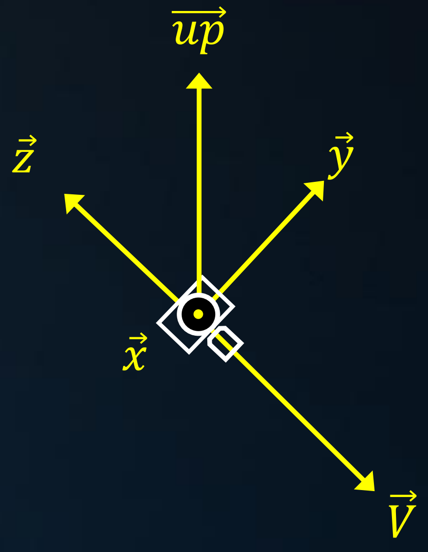
Perspective

Transformation Pipeline

STEP THREE: camera space to orthographic view volume

Basis defined by the camera orientation:

z-axis: $-\vec{V}$ (convention says we look down –z)
x-axis: $-\vec{V} \times \vec{up}$
y-axis: $\vec{V} \times \vec{x}$



Matrix:

$$\begin{pmatrix} X_x & Y_x & -V_x & 0 \\ X_y & Y_y & -V_y & 0 \\ X_z & Y_z & -V_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Inverse:

$$\begin{pmatrix} X_x & X_y & X_z & 0 \\ Y_x & Y_y & Y_z & 0 \\ -V_x & -V_y & -V_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = M_{camera}$$

Perspective

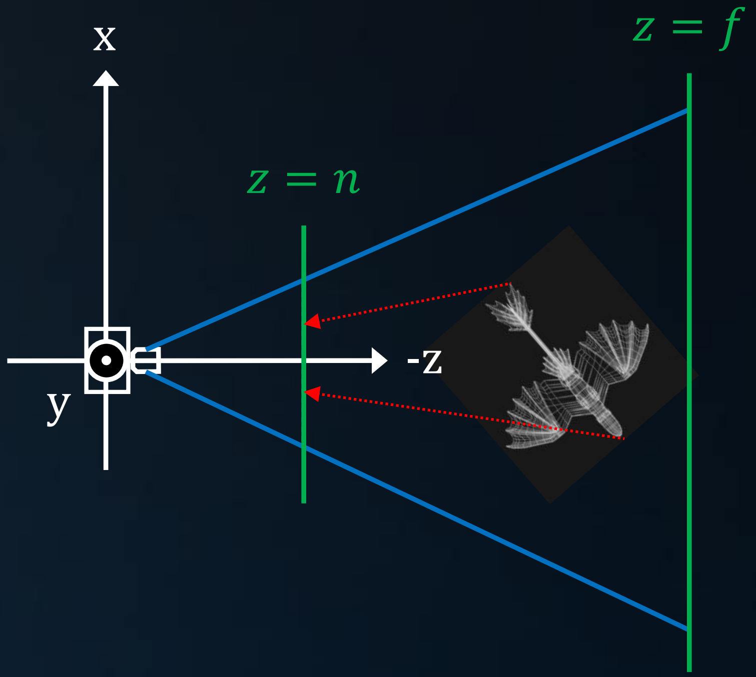
Transformation Pipeline

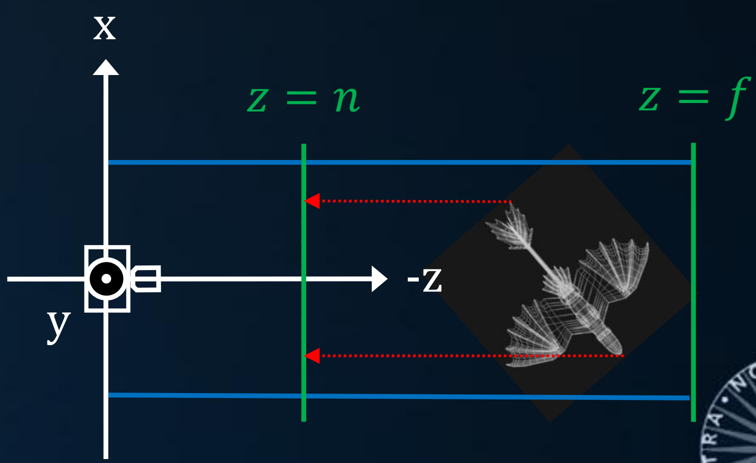
STEP THREE: camera space to orthographic view volume

The combined transform so far:

$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp}\, M_{canonical}\, M_{camera} \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{pmatrix}$$

One thing is still missing: perspective.





Perspective

Q: What is perspective?
A: The size of an object on the screen is proportional to $1/z$.

More precisely:

$$y_s = \frac{d}{z} y \quad (\text{and } x_s = \frac{d}{z} x\ )$$

where d is the distance of the view plane to the camera.

Q: How do we capture scaling based on distance in a matrix?
A: ...

Dividing by z can't be done using linear nor affine transforms.

Perspective

Let's have a look at homogeneous coordinates again.

Recall:

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_1x + b_1y + c_1z \\ a_2x + b_2y + c_2z \\ a_3x + b_3y + c_3z \end{pmatrix}$$

With homogeneous coordinates, we get:

$$\begin{pmatrix} a_1 & b_1 & c_1 & T_x \\ a_2 & b_2 & c_2 & T_y \\ a_3 & b_3 & c_3 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a_1x + b_1y + c_1z + T_x \\ a_2x + b_2y + c_2z + T_y \\ a_3x + b_3y + c_3z + T_z \\ 1 \end{pmatrix} = \begin{pmatrix} (a_1x + b_1y + c_1z + T_x)/1 \\ (a_2x + b_2y + c_2z + T_y)/1 \\ (a_3x + b_3y + c_3z + T_z)/1 \\ 1 \end{pmatrix}$$

Perspective

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{pmatrix} = \begin{pmatrix} a_1 & b_1 & c_1 & T_x \\ a_2 & b_2 & c_2 & T_y \\ a_3 & b_3 & c_3 & T_z \\ a_4 & b_4 & c_4 & w \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a_1x + b_1y + c_1z + T_x \\ a_2x + b_2y + c_2z + T_y \\ a_3x + b_3y + c_3z + T_z \\ a_4x + b_4y + c_4z + w \end{pmatrix}$$

Recall that using homogeneous coordinates $(x, y, z, 1)$ represents (x, y, z).
The homogeneous vector (x, y, z, w) represents $\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$.
The division by w is called *homogenization*.

Notice that this doesn't change any part of our framework when $w = 1$.

Perspective

So, multiplying by this matrix

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \times \begin{pmatrix} a_1 & b_1 & c_1 & T_x \\ a_2 & b_2 & c_2 & T_y \\ a_3 & b_3 & c_3 & T_z \\ a_4 & b_4 & c_4 & w \end{pmatrix}$$

and homogenization, creates this vector:

$$\begin{pmatrix} (a_1x + b_1y + c_1z + Tx) \;/\; (a_4x + b_4y + c_4z + w) \\ (a_2x + b_2y + c_2z + Ty) \;/\; (a_4x + b_4y + c_4z + w) \\ (a_3x + b_3y + c_3z + Tz) \;/\; (a_4x + b_4y + c_4z + w) \\ 1 \end{pmatrix}$$

How do we chose the coefficients of the matrix so that we get correct perspective correction?

I.e., something like this:

$$\begin{pmatrix} nx/z \\ ny/z \\ z \\ 1 \end{pmatrix}$$

Perspective

The matrix we are looking for is:

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{pmatrix} \xrightarrow{\text{homogenize}} \begin{pmatrix} nx/z \\ ny/z \\ n+f-fn/z \\ 1 \end{pmatrix}$$

Let's verify.

What happened to z? ➔ $z' = n + f - \frac{fn}{z}$

- $z = n$: $z' = n$
- $z = f$: $z' = f$
- All other z yield values between n and f (but: proportional to $\frac{1}{z}$).

Perspective

Combining with the orthographic projection matrix gives us:

$$M_{ortho} \times \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Perspective

Transformation Pipeline

To transform a single world vertex we thus apply:

$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{perspective} M_{camera} \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{pmatrix}$$

1. M_{camera}: takes us from world space to camera space;
2. $M_{perspective}$: from camera space to canonical;
3. M_{vp}: takes us from canonical to screen space.

$$M_{vp} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_{perspective} \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

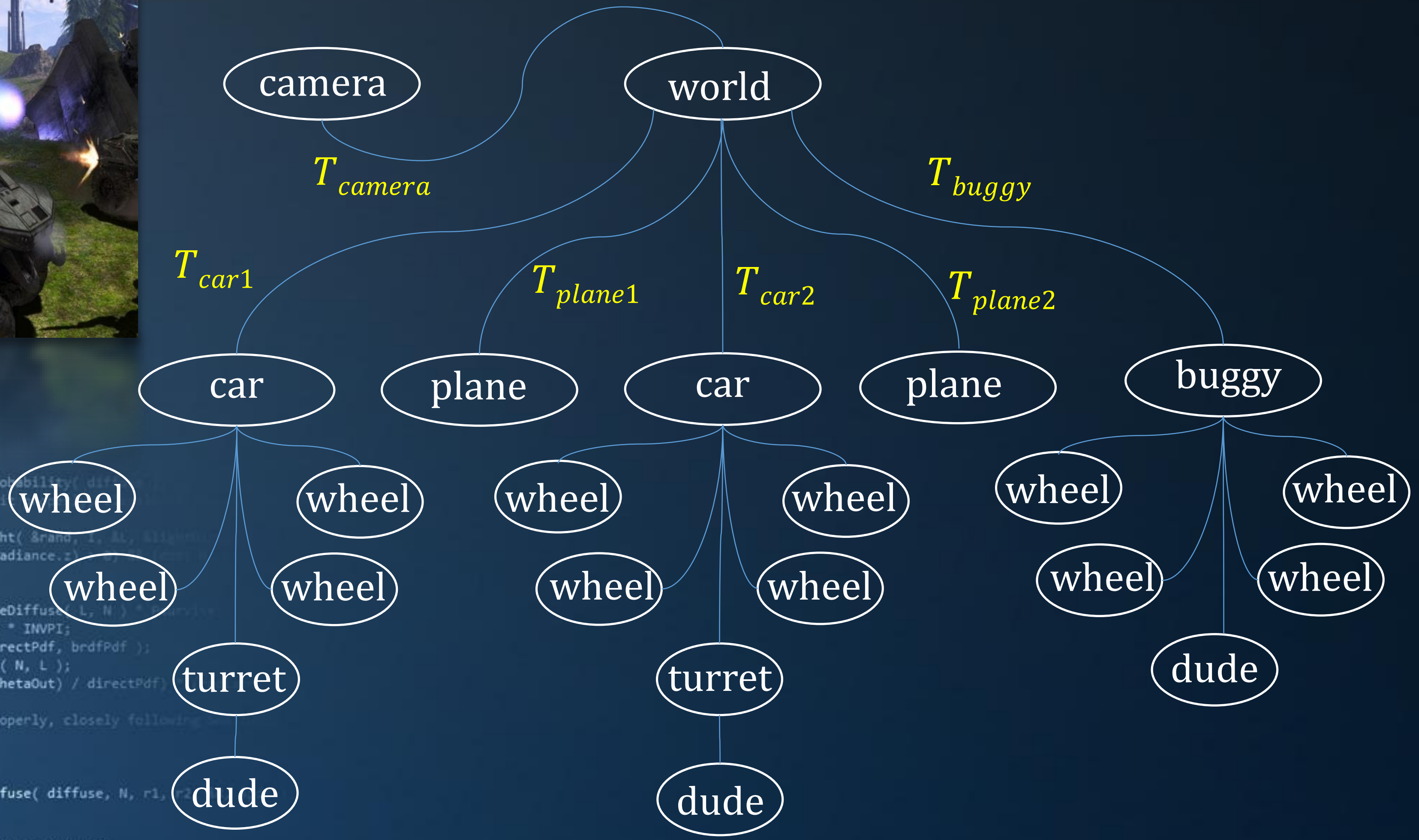
$$M_{camera} = \begin{pmatrix} X_x & X_y & X_z & -E_x \\ Y_x & Y_y & Y_z & -E_y \\ -V_x & -V_y & -V_z & -E_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Today’s Agenda:

- Affine Transforms
- Projection
- Pipeline Recap
- Rasterization

Pipeline Recap

Scenegraph



Animation, culling, tessellation, ...

meshes

Transform

vertices

Project

vertices

Rasterize

fragment positions

Shade

pixels

Postprocessing

Pipeline Recap

Transformations

World space to screen space:

$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{perspective} M_{camera} \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{pmatrix}$$

Object space to world space:

$$\begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{pmatrix} = M_{local} M_{parent} \begin{pmatrix} x_{local} \\ y_{local} \\ z_{local} \\ 1 \end{pmatrix}$$

In all cases, we construct a *single* 4×4 matrix, which we then apply to all vertices of a mesh.

Animation, culling, tessellation, ...

meshes

Transform

vertices

Project

vertices

Rasterize

fragment positions

Shade

pixels

Postprocessing

Pipeline Recap

Transformations

Rendering a scene graph is done using a recursive function:

```
void SGNode::Render( mat4& M )
{
   mat4 M’ = Mlocal * M;
   mesh->Rasterize( M’ );
   for( int i = 0; i < childCount; i++ )
      child[i]->Render( M’ );
};
```

Here, matrix concatenation is part of the recursive flow.

Pipeline Recap

Transformations

To transform meshes to world space, we call SGNode::Render with an identity matrix.

To transform meshes to camera space, we call it with the *inverse* transform of the camera.

Remember: the world revolves around the viewer; instead of turning the viewer, we turn the world in the opposite direction.

```
void SGNode::Render( mat4& M )
{
   mat4 M’ = Mlocal * M;
   mesh->Rasterize( M’ );
   for( int i = 0; i < childCount; i++ )
      child[i]->Render( M’ );
};
```

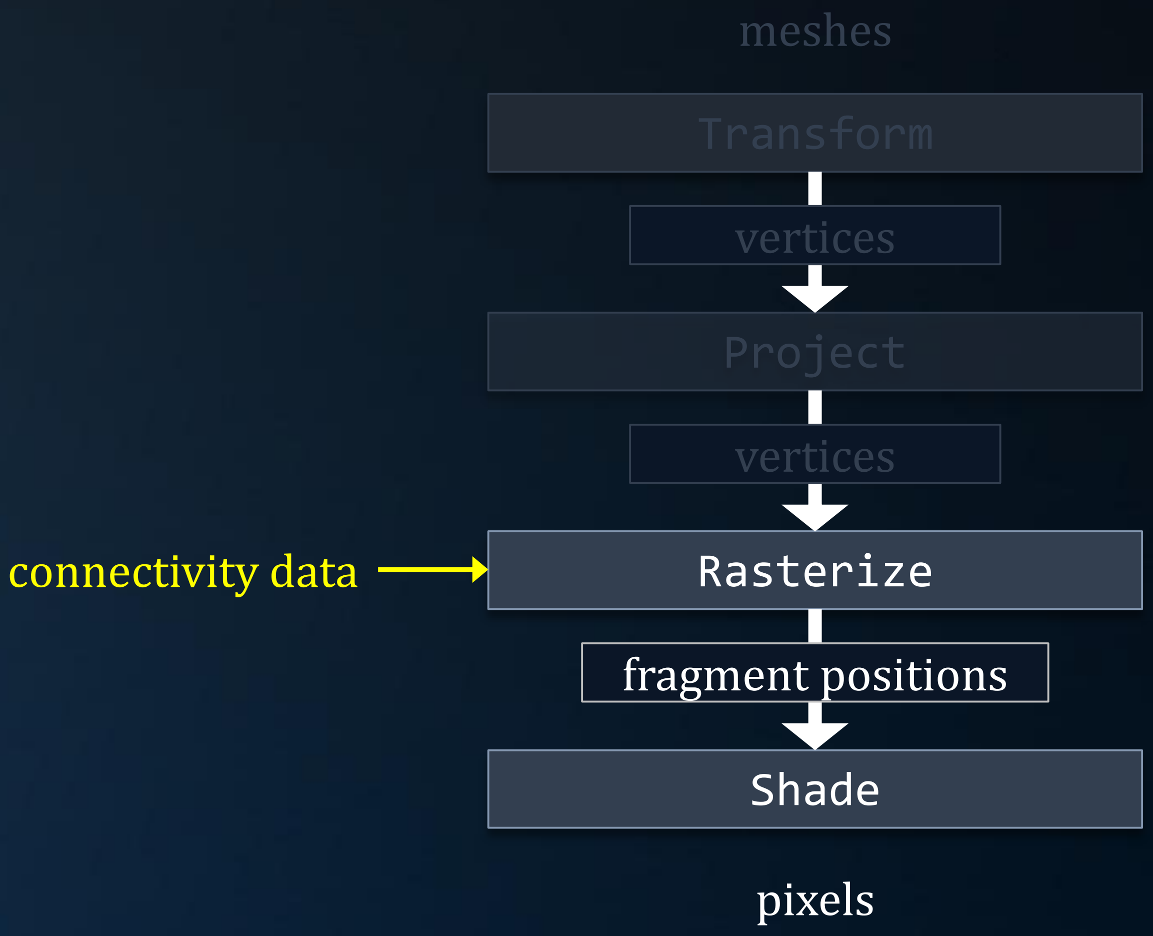
Pipeline Recap

After projection

The output of the projection stage is a stream of vertices for which we know 2D screen positions.

The vertex stream must be combined with connectivity data to form triangles.

'Triangles' on a raster consist of a collection of pixels, called *fragments.*



Today's Agenda:

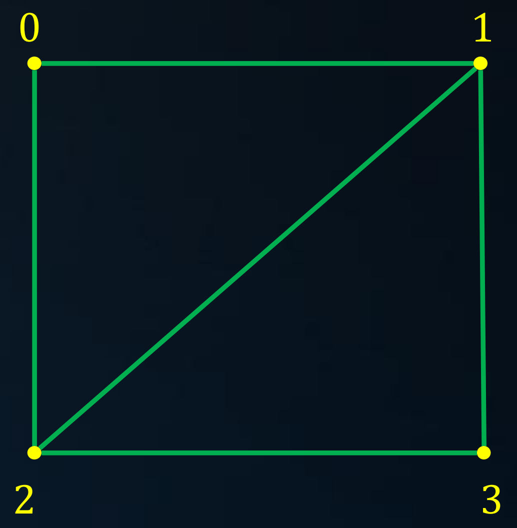
- Affine Transforms
- Projection
- Pipeline Recap
- Rasterization

Rasterization

Connectivity data

Two triangles forming a quad, using four vertices:

0	1	2	1	3	2



Note:

- Connectivity data has no relation to actual vertex positions.
- Triangles are typically defined in clockwise order around the triangle normal.

These two notes can be contradictory, but in practice, they rarely are.

Rasterization

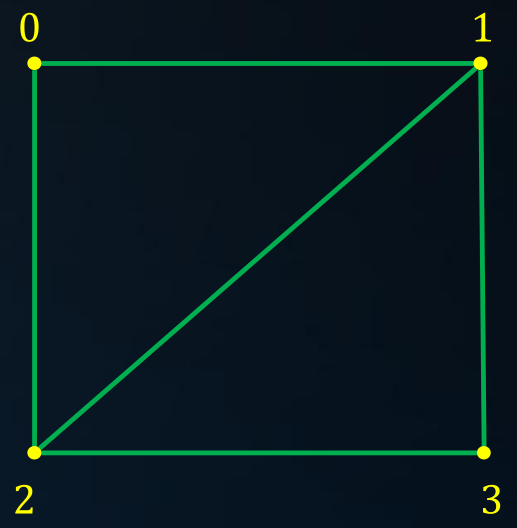
Connectivity data

We can store triangles more efficiently using triangle strips.

0	1	2	3

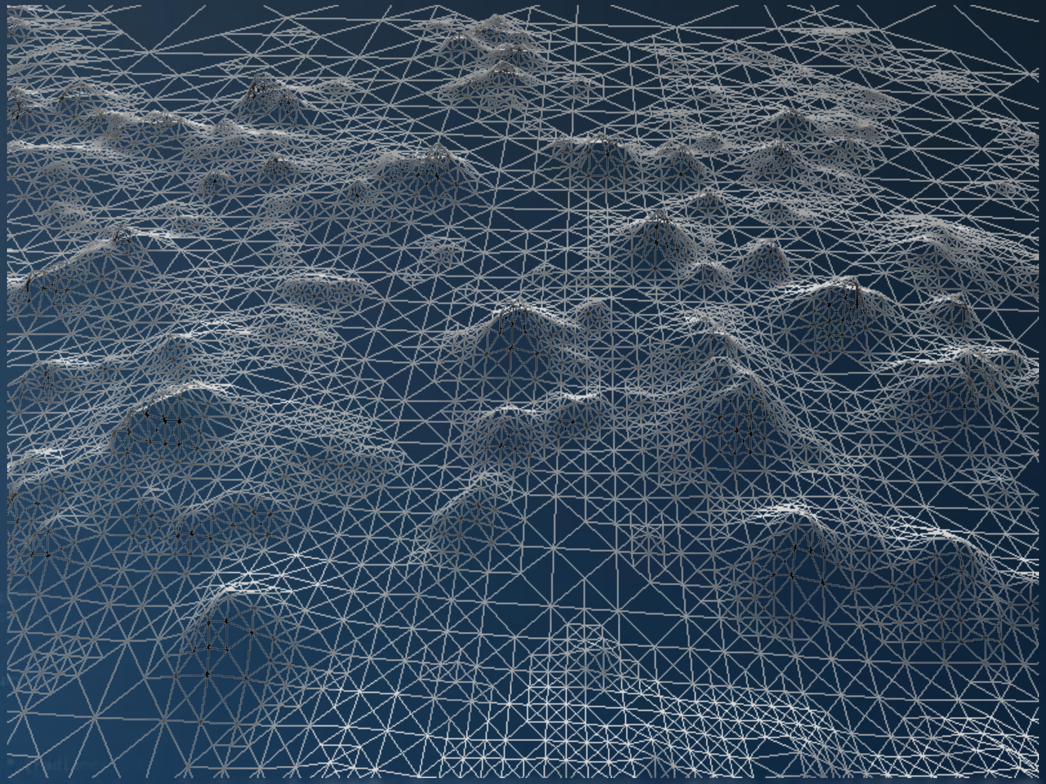
Here, the first three vertex indices specify the first triangle. After that, subsequent triangles use the previous two indices, plus one extra vertex.

It is rarely possible to define a complete mesh using a single triangle strip. However, we can generally reduce a mesh to a small set of strips.



Rasterization

Connectivity data

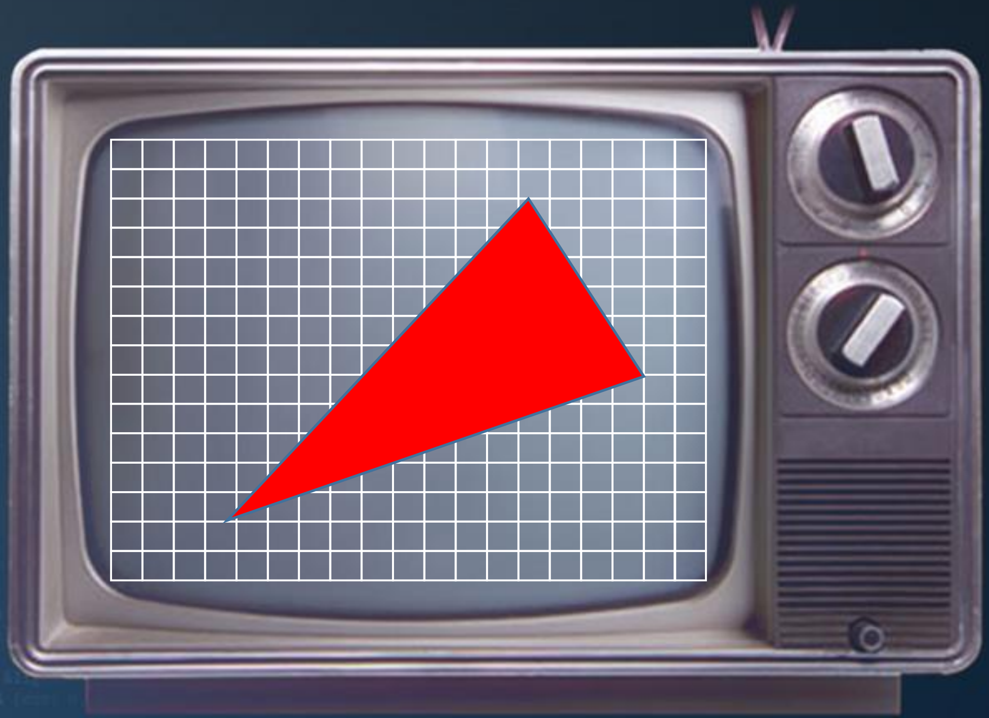


On modern hardware, triangle strips are rarely used:

- The memory reduction affects only the connectivity data, which is small compared to vertex data;
- Multiple strips for a single mesh may incur significant overhead in the driver.

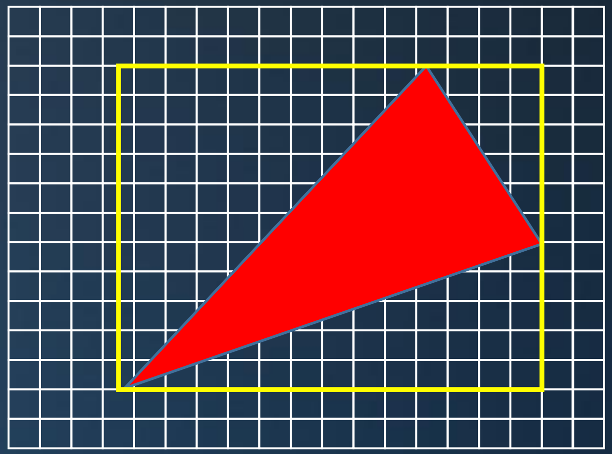
Rasterization

Triangle rasterization



Rasterization

Triangle rasterization



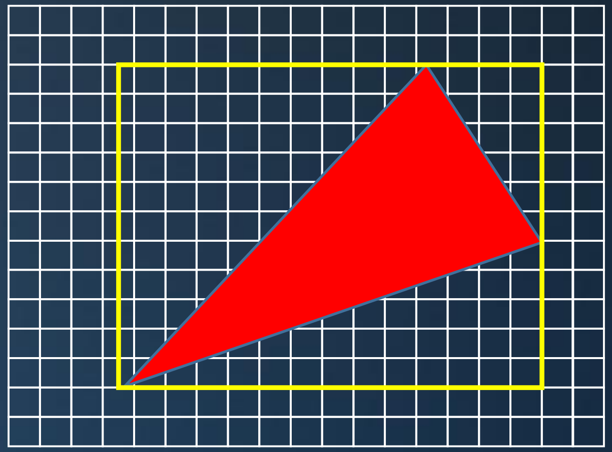
Rasterizing a triangle, method 1:
(from the book, 8.1.2)

1. Determine the axis-aligned bounding box of the triangle;
2. For each pixel within this box, determine whether it is inside the triangle.

Drawback: at least 50% of the pixels will be rejected.

Rasterization

Triangle rasterization



Rasterizing a triangle, method 2:
(see e.g. fatmap.txt, fatmap2.txt)

1. Per scanline (within the bounding box), determine the left and right side of the triangle;
2. Per scanline, draw a horizontal line from the left to the right.

Drawback: not as easy to execute in parallel on GPUs.

Rasterization

Triangle rasterization

So far, we have seen how to fill a triangle, or more accurately: how to determine which pixels it overlaps.

To shade the triangle, we need more information.

Per pixel:

- Color (e.g. from a texture);
- Normal;
- Interpolated per-vertex shading information.

Rasterization

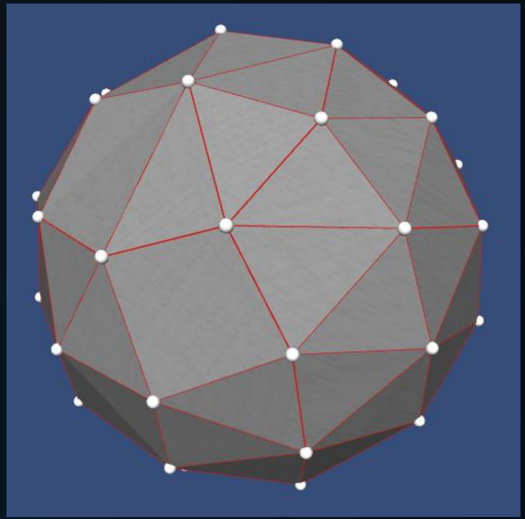
Sanity check

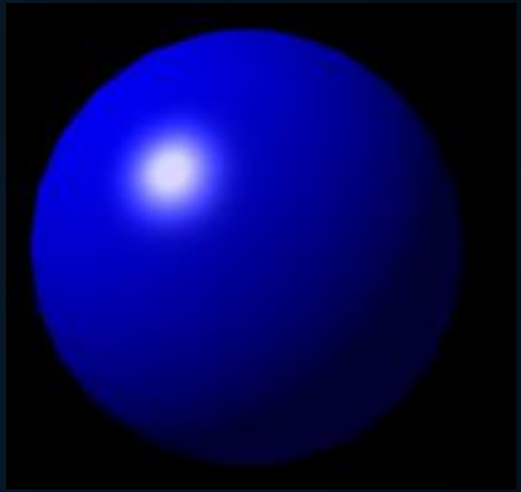
Let’s take a brief moment to meditate on the madness on the previous slide.

Per pixel:

- Normal

A triangle is defined by three vertices. All points on the triangle lie in the same plane. Therefore, the normal for each point on the triangle is the same.

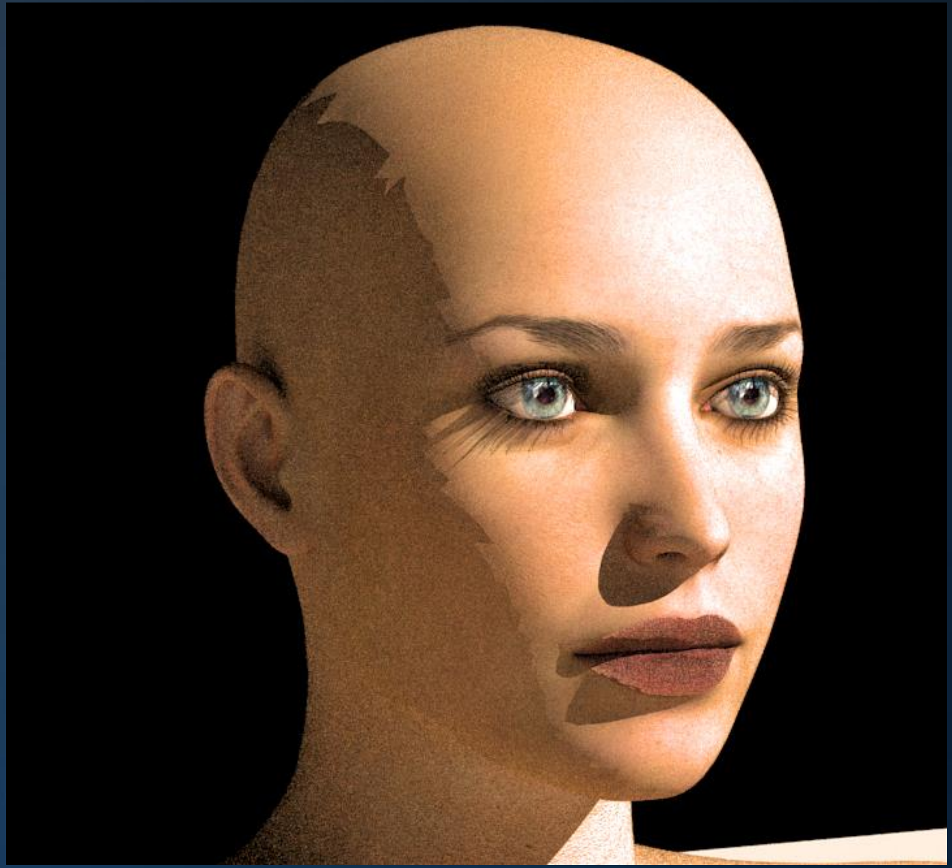


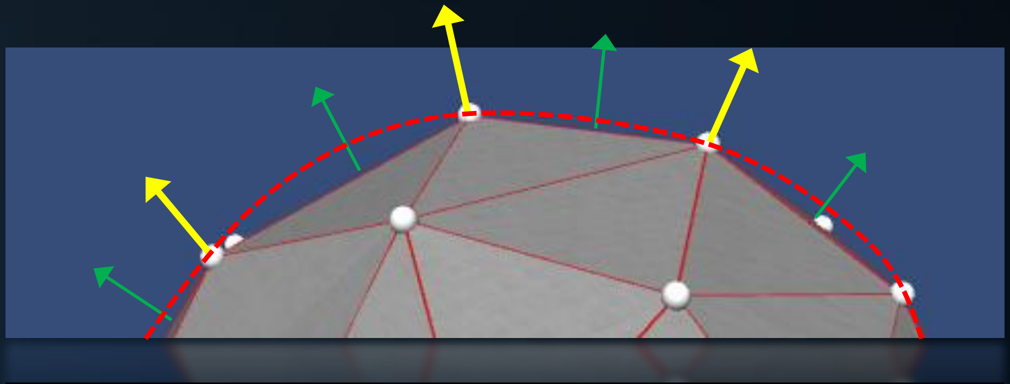


Rasterization

Sanity check

Normal interpolation can cause some bad behavior:



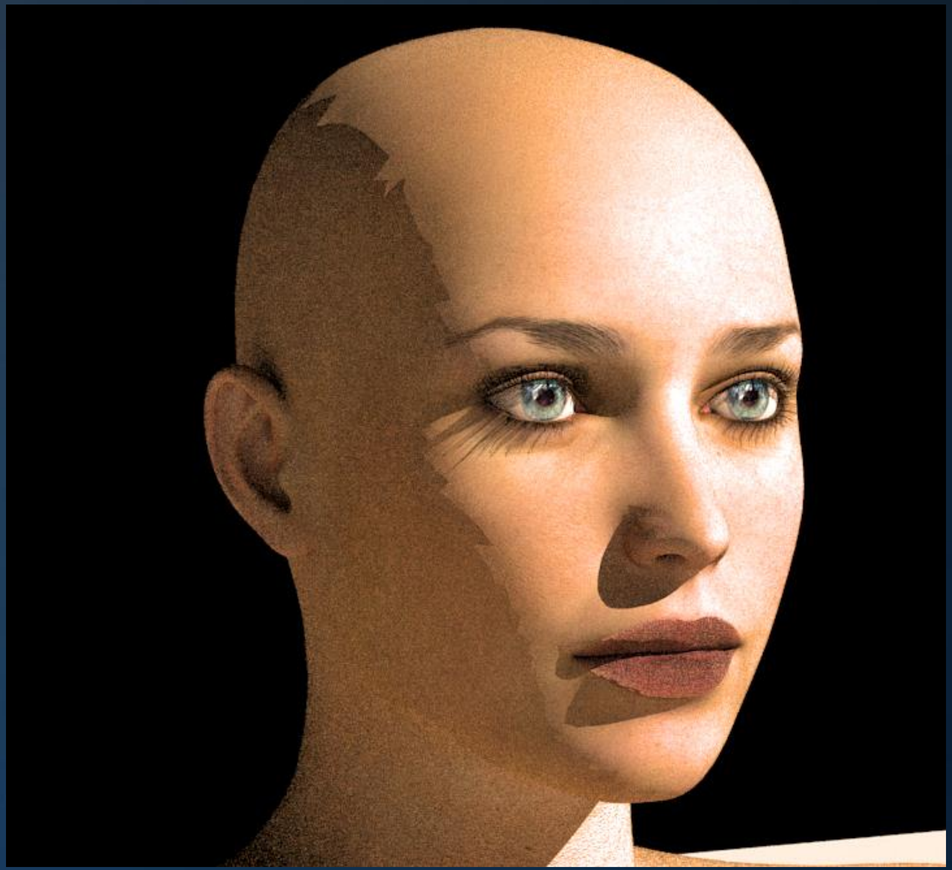


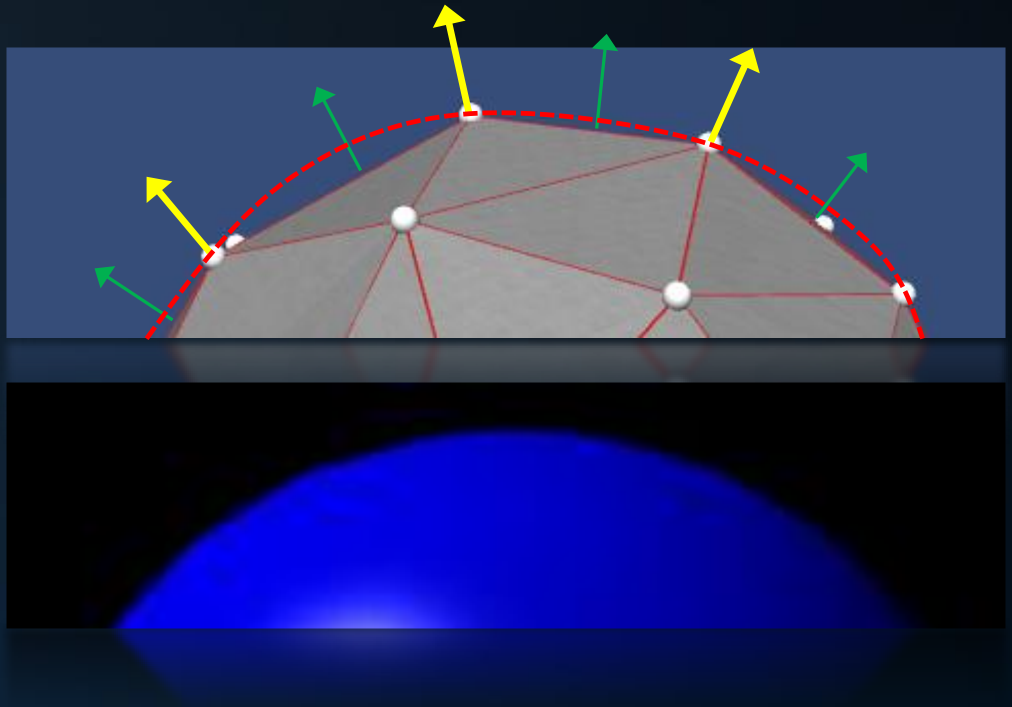
Shadows are still cast by the not-so-smooth geometry.

Rasterization

Sanity check

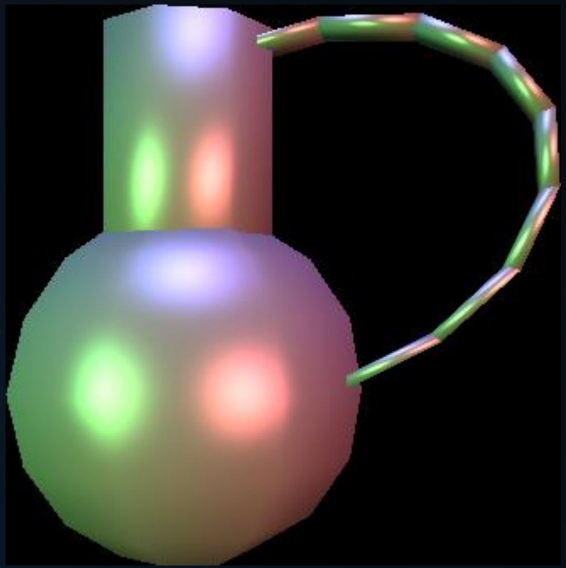
Normal interpolation can cause some bad behavior:

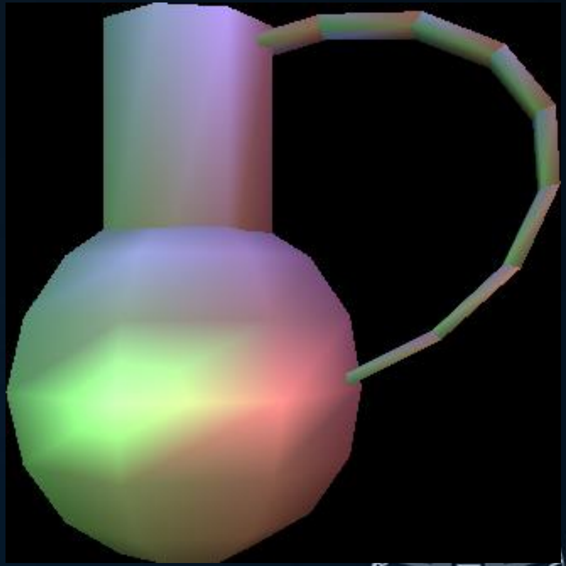




Shadows are still cast by the not-so-smooth geometry.

Rasterization





Sanity check

Shading interpolation:

Normal interpolation is costly: a linearly interpolated normal needs normalization, which involves a square root.

Solution: calculate shading per vertex, and interpolate.

Rasterization

Sanity check

Shading:

In nature, the color of a surface is the sum of all the light reflected by the surface towards the camera.

Incoming light:

- Direct light (arriving from light sources);
- Indirect light (arriving via other surfaces).

Incoming light is partially absorbed, partially reflected.
Light is generally not reflected uniformly in all directions.

Rasterization

Triangle rasterization

Interpolating per-vertex values over a triangle:

Barycentric coordinates.

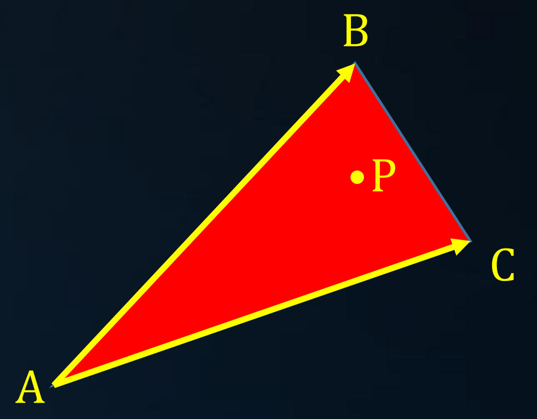
Any point on the triangle can be parameterized by two values:

$$P(\lambda_1, \lambda_2) = A + \lambda_1(B - A) + \lambda_2(C - A)$$

where $0 \leq \lambda_1, \lambda_2 \leq 1$, and $\lambda_1 + \lambda_2 \leq 1$.

Or, reversed:

$$\lambda_1 = P \cdot (B - A) - P \cdot A$$
$$\lambda_2 = P \cdot (C - A) - P \cdot A$$



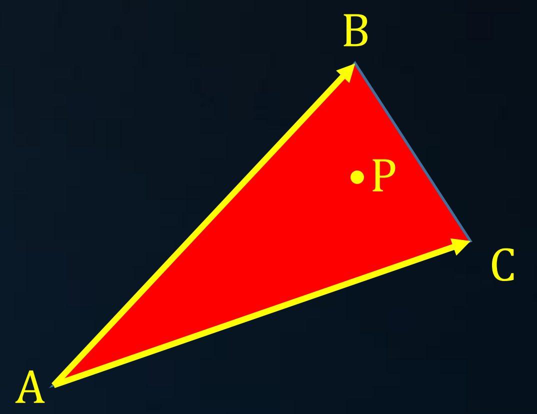
Rasterization

Triangle rasterization

$$P(\lambda_1, \lambda_2) = A + \lambda_1(B - A) + \lambda_2(C - A)$$

Given the vertex normals N_A, N_B and N_C, we can now calculate the interpolated per-pixel normal N_P:

$$N_P = N_A + \lambda_1(N_B - N_A) + \lambda_2(N_C - N_A)$$



Remember that an interpolated normal is typically not normalized.

Today's Agenda:

- Affine Transforms
- Projection
- Pipeline Recap
- Rasterization

INFOGR – Computer Graphics

J. Bikker - April-July 2016 - Lecture 9: "Transformations"

END of "Transformations"

next lecture: “Shading Models”