

INFOGR – Computer Graphics

J. Bikker - April-July 2015 - Lecture 6: "Transformations"

Welcome!



Today's Agenda:

- Projection
- Pipeline Recap
- Rasterization



Perspective

Projection – Applying matrices, working our way backwards

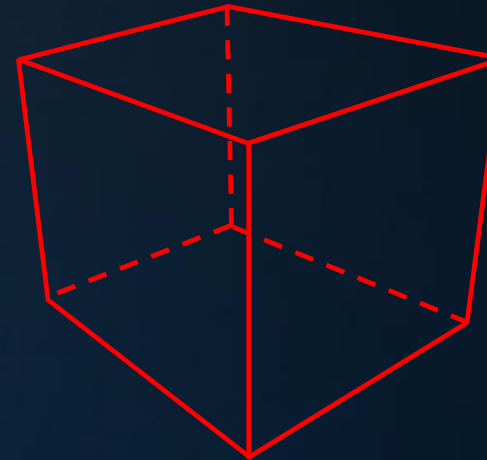
Goal: create 2D images of 3D scenes

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

Parallel
projection:



Perspective
projection:



Perspective

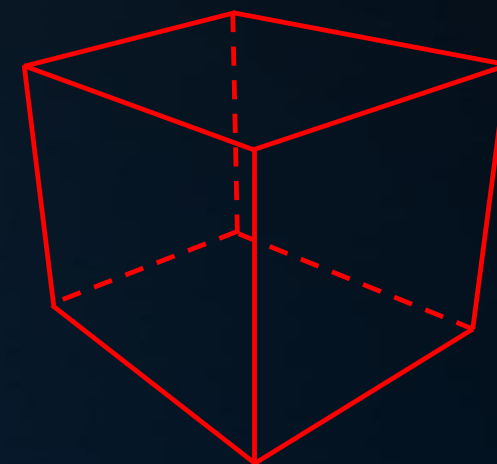
Parallel projection:

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.

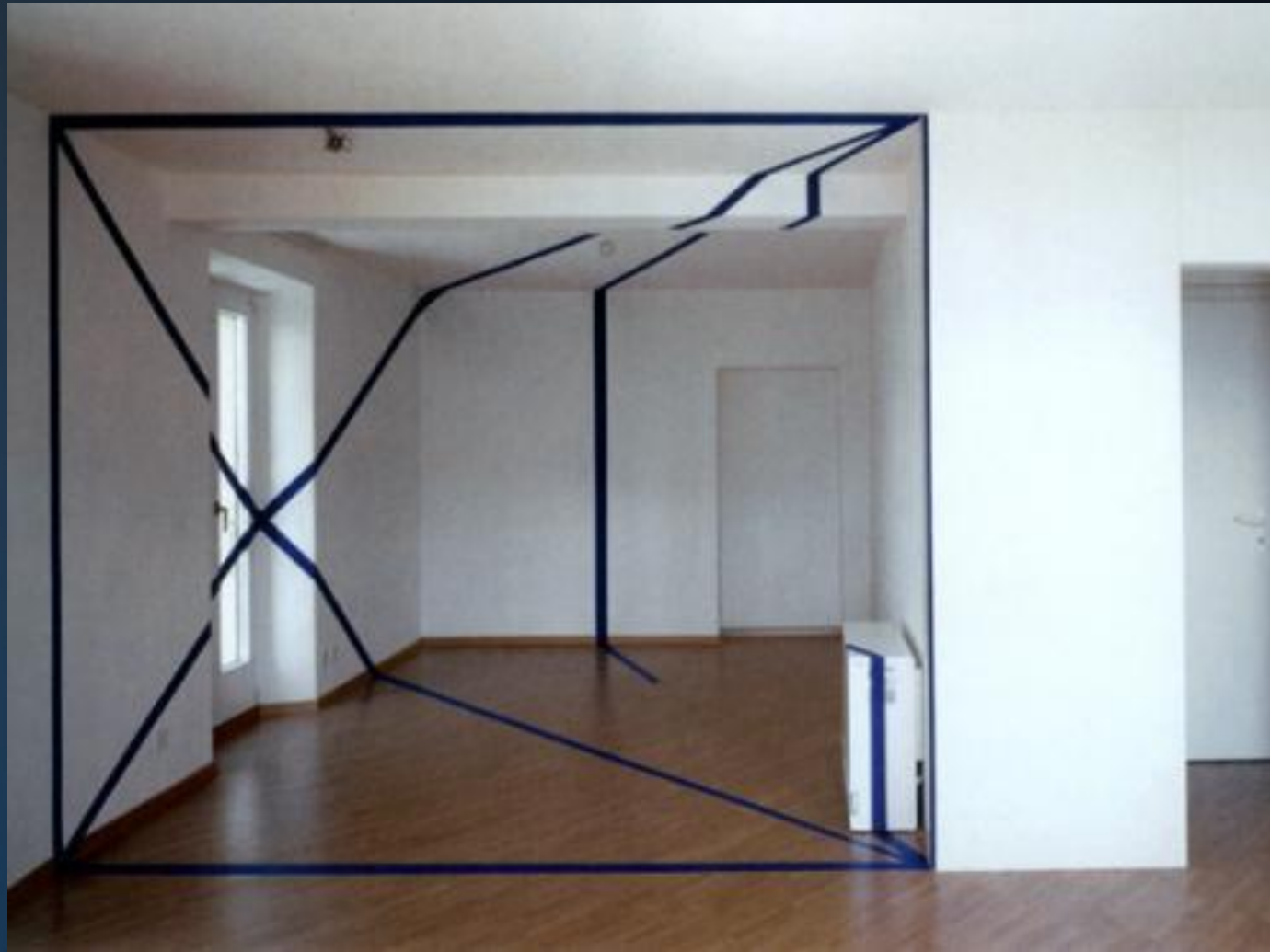


Perspective

```

    if (depth < MAXDEPTH)
    {
        Vec inside = L;
        Vec nt = nc / nc;
        Vec nnt = 1.0f - nnt * nnt;
        Vec D, N;
        Vec a = nt - nc;
        Vec b = nt + nc;
        Vec Tr = 1 - (RB + (1 - RB) * nnt);
        Vec R = (D * nnt - N * (1 - Tr));
        Vec E * diffuse;
        Vec refl;
        Vec refl + refr;
        Vec D, N;
        Vec refl * E * diffuse;
        Vec refl;
        Vec MAXDEPTH);
        Vec survive = SurvivalProbability( diffuse );
        Vec estimation - doing it properly, closely following walls;
        Vec if;
        Vec radiance = SampleLight( &rand, I, &L, &align );
        Vec e.x + radiance.y + radiance.z > 0) && (e.x + radiance.y + radiance.z > 0);
        Vec v = true;
        Vec brdfPdf = EvaluateDiffuse( L, N );
        Vec factor = diffuse * INVPI;
        Vec weight = Mis2( directPdf, brdfPdf );
        Vec cosThetaOut = dot( N, L );
        Vec E * ((weight * cosThetaOut) / directPdf) * (radiance);
        Vec random walk - done properly, closely following walls;
        Vec survive);
        Vec brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
        Vec survive;
        Vec pdf;
        Vec n = E * brdf * (dot( N, R ) / pdf);
        Vec n = true;
    }

```



Perspective

```

    if (depth < MAXDEPTH)
    {
        // Inside the glass
        nt = inside / (1 + 0.05 * depth);
        nc = nt / nc;
        nnt = nt * nt;
        cos2t = 1.0f - nnt;
        D, N );
    }

    // Refracted ray
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (RB + (1 - RB) * nnt);
    Tr) R = (D * nnt - N * (cos2t > 0.5 ? 1 : -1));

    // Diffuse reflection
    E * diffuse;
    = true;

    // Refracted ray
    refl + refr) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;

        MAXDEPTH)

    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following walk
    if;
    radiance = SampleLight( &rand, I, &L, &light );
    e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
    {
        v = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        at3 factor = diffuse * INVPI;
        at weight = Mis2( directPdf, brdfPdf );
        at cosThetaOut = dot( N, L );
        E * ((weight * cosThetaOut) / directPdf) * (radiance
        random walk - done properly, closely following walk
        survive)

    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```




```

100
101 (depth < MAXDEPTH)
102
103     nc = inside / l * 1.0f;
104     nt = nt / nc; dde = 1.0f / nc;
105     pos2t = 1.0f - nnt * wnt;
106     D, N );
107 )
108
109     at a = nt - nc, b = nt + nc;
110     at Tr = 1 - (RB + (1 - RB) * a);
111     Tr) R = (D * nnt - N * (dde
112
113     E * diffuse;
114     = true;
115
116
117     refl + refr)) && (depth < MAXDEPTH)
118
119     D, N );
120     refl * E * diffuse;
121     = true;
122
123
124 MAXDEPTH)
125
126 survive = SurvivalProbability( diffuse );
127 estimation - doing it properly, classically
128 if;
129 radiance = SampleLight( &rand, l, &l, &light,
130     e.x + radiance.y + radiance.z) > 0) && (max
131
132 w = true;
133 at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
134 at3 factor = diffuse * INVPI;
135 at weight = Mis2( directPdf, brdfPdf );
136 at cosThetaOut = dot( N, L );
137 E * ((weight * cosThetaOut) / directPdf) * (radiance
138
139 random walk - done properly, closely following
140 (survive)
141
142
143 at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
144 survive;
145 pdf;
146 n = E * brdf * (dot( N, R ) / pdf);
147 sion = true;

```



```

    (depth < MAXDEPTH)
    {
        int inside = 1;
        int nt = nt / nc, ddo = 0;
        double cos2t = 1.0f - nnt * nnt;
        Vec D, N );
    }

    Vec a = nt - nc, b = nt * nc;
    Vec Tr = 1 - (RR + (1 - RR) * a);
    Vec R = (D * nnt - N * (ddo +
    E * diffuse;
    = true;

    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;

    MAXDEPTH)

    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, clearly
    if;

    radiance = SampleLight( &rand, I, &L, &light,
    e.x + radiance.y + radiance.z) > 0) && (max <
    w = true;
    Vec brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    Vec t3 factor = diffuse * INVPI;
    Vec weight = Mix2( directPdf, brdfPdf );
    Vec cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following survival
    ve)

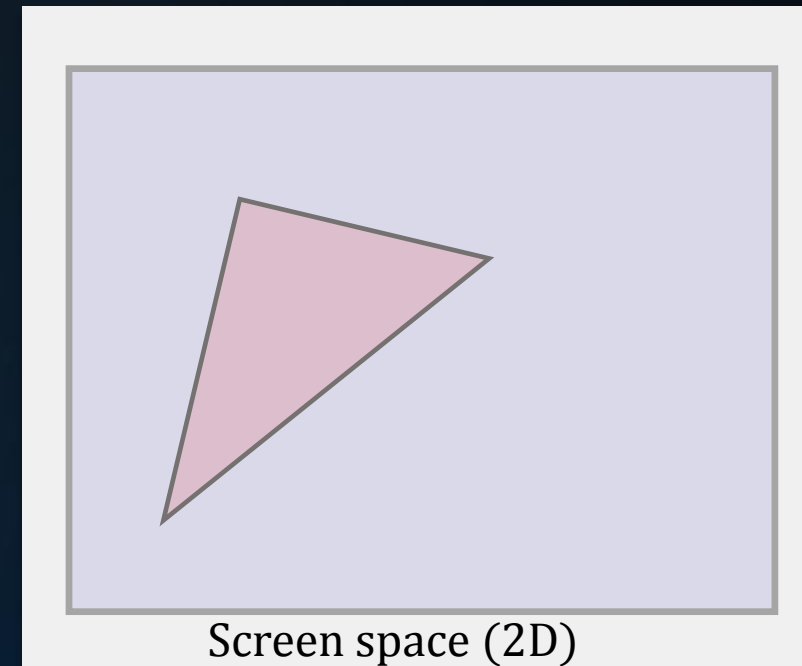
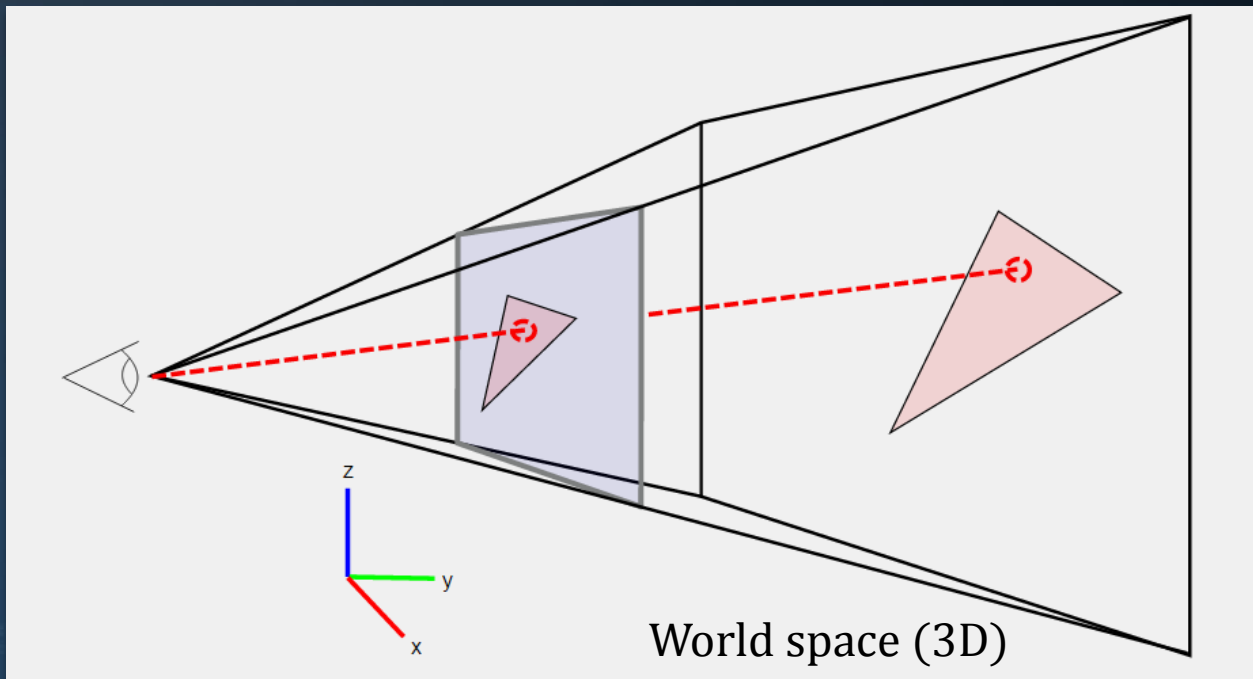
    ;
    Vec t3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



Perspective

Perspective projection



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



Perspective

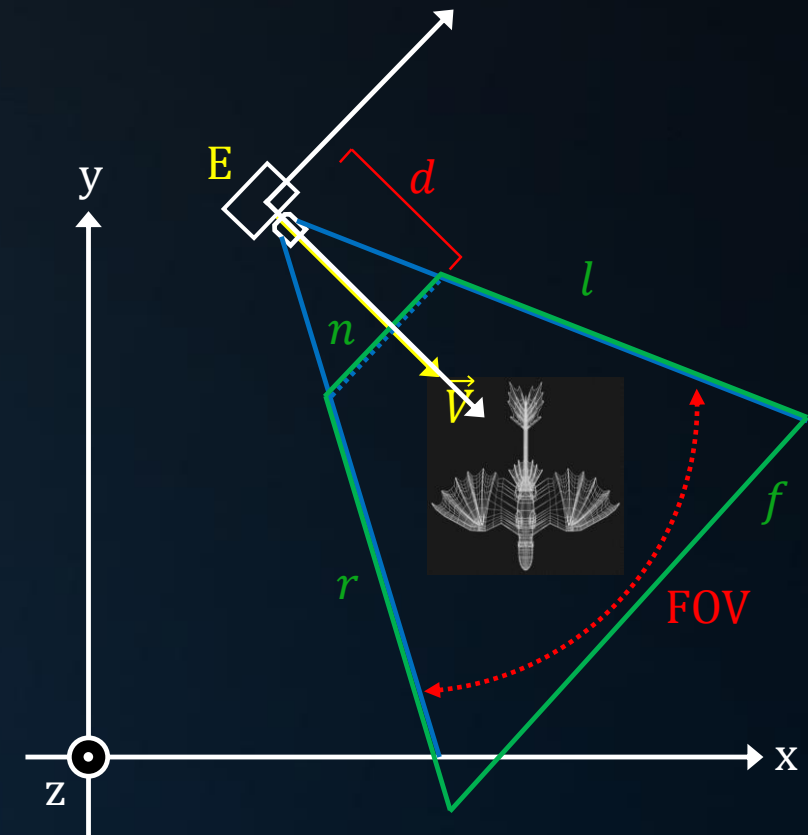
Perspective projection

The camera is defined by:

- Its position E
- The view direction \vec{V}
- The image plane (defined by its distance d 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

Perspective projection

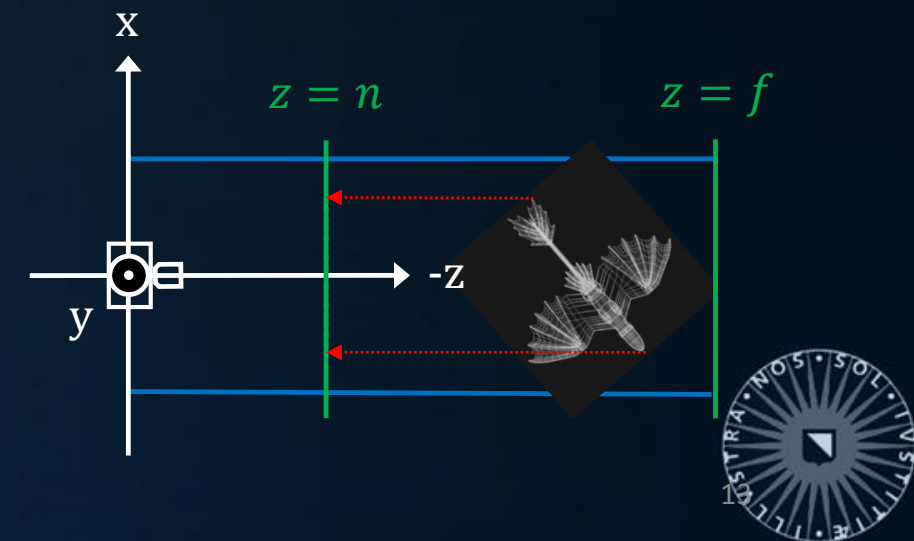
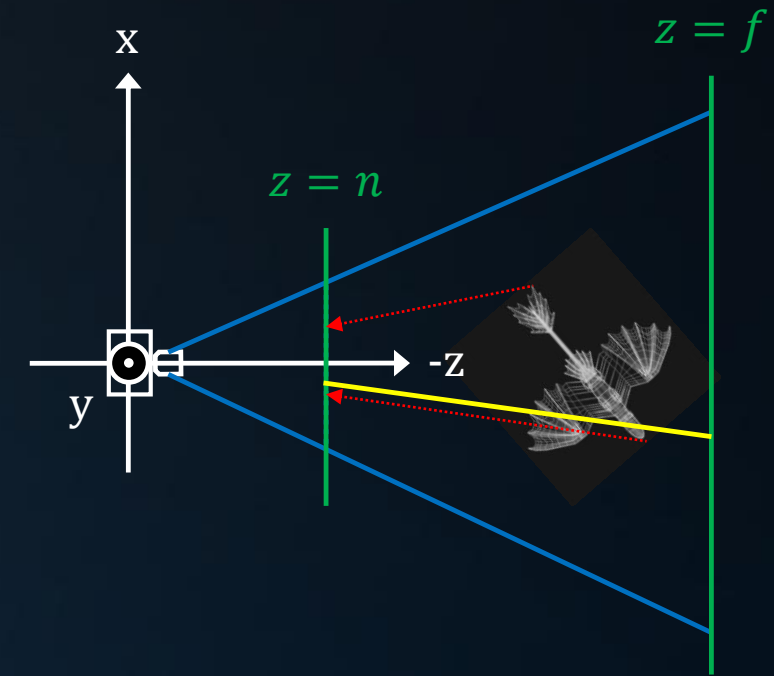
Camera space: looking down negative z.

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.

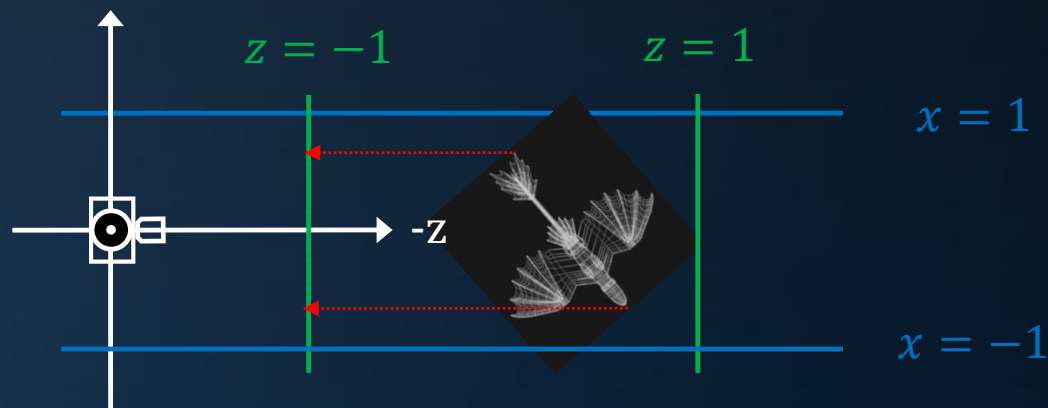


Perspective

Perspective projection

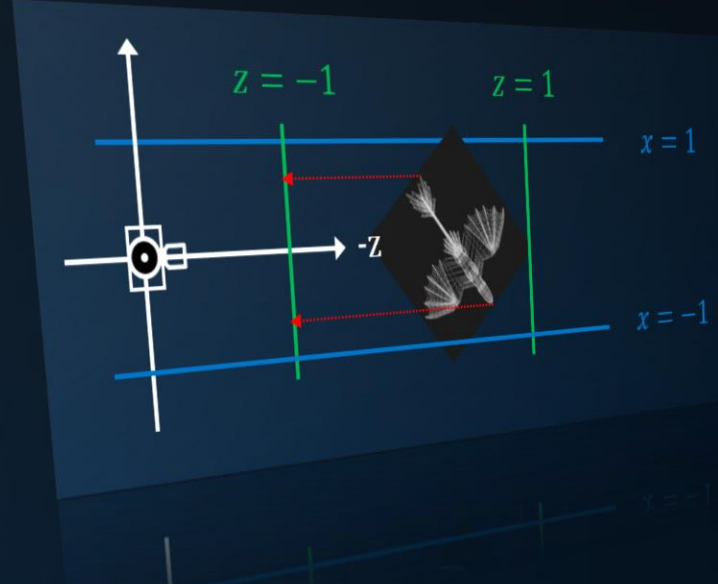
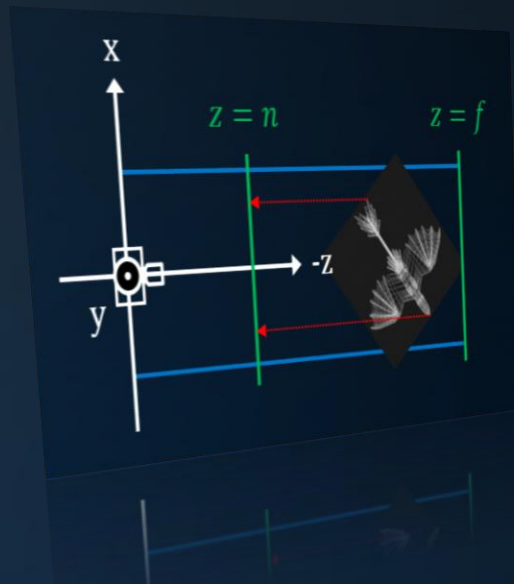
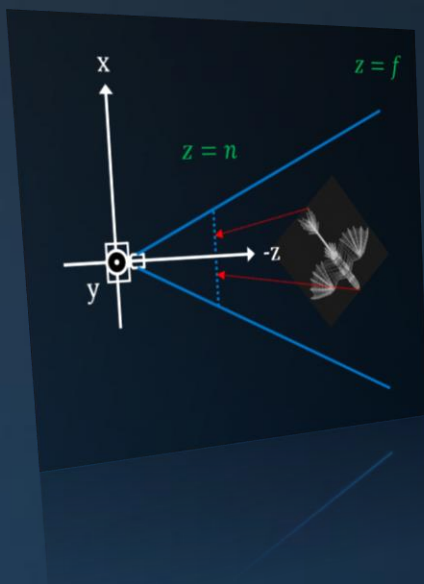
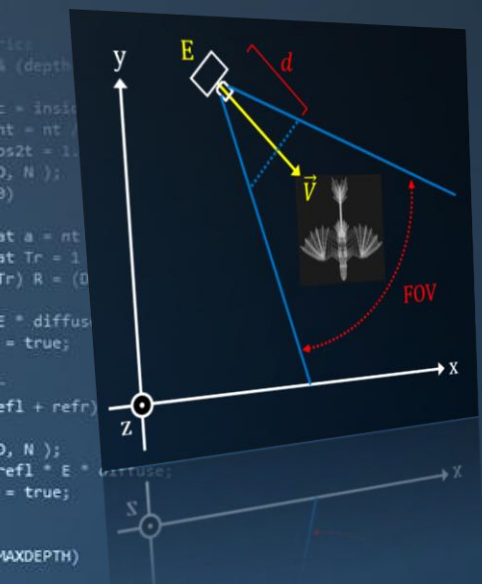
The final transform is the one that takes us from the orthographic view volume to the canonical view volume.

Again, this is done using a matrix.



Perspective

Perspective projection



World space → camera space → orthographic view → canonical view

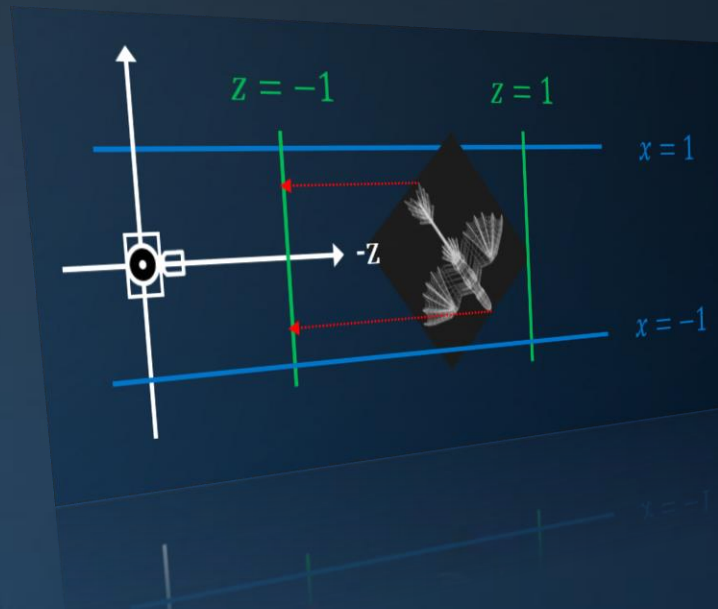
$I \times M_{\text{camera}} \times M_{\text{ortho}} \times M_{\text{canonical}}$

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

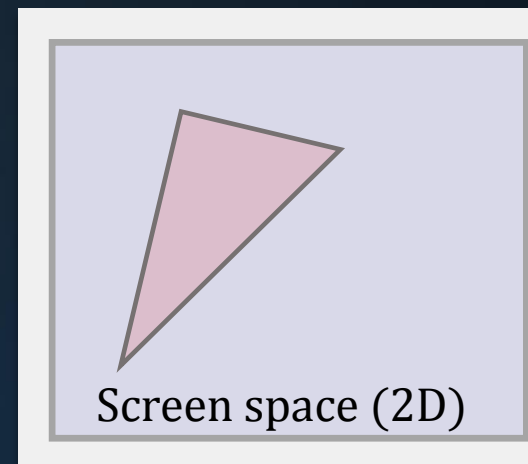


Perspective

Perspective projection



Canonical view



screen

We need one last transform:

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



Perspective

Perspective projection

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:

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

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



Perspective

Perspective projection

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

Perspective projection

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

Perspective projection

STEP TWO: orthographic view volume to canonical view volume

The final transforms for the vertices are thus:

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

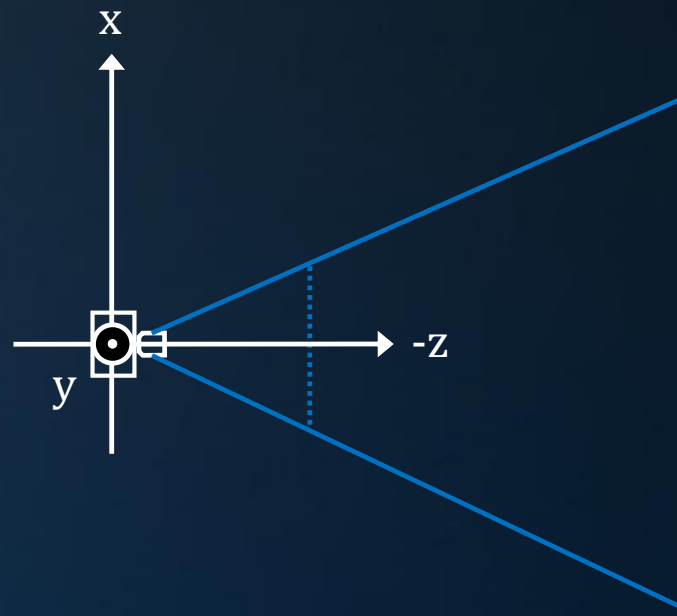
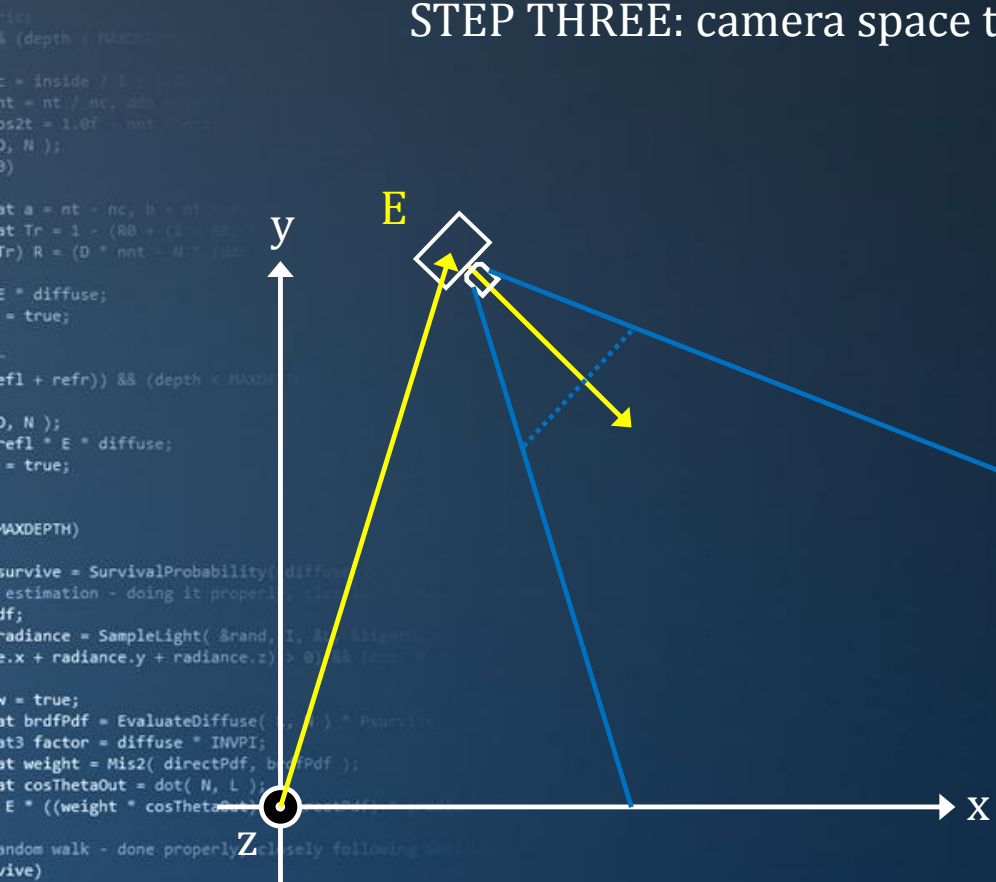
Next step: getting from camera space to the orthographic view volume.



Perspective

Perspective projection

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

Perspective projection

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: $\vec{V} \times \vec{x}$

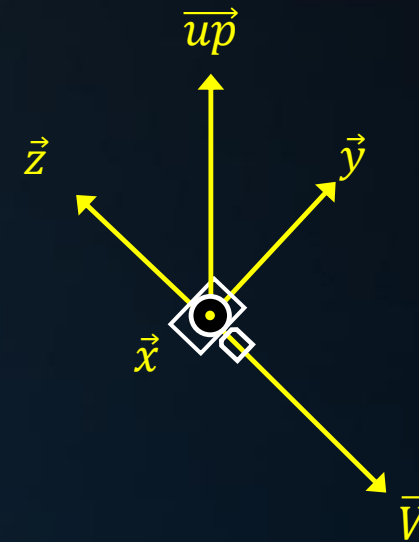
Matrix:

Inverse:

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

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

$$\times \begin{pmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = M_{camera}$$



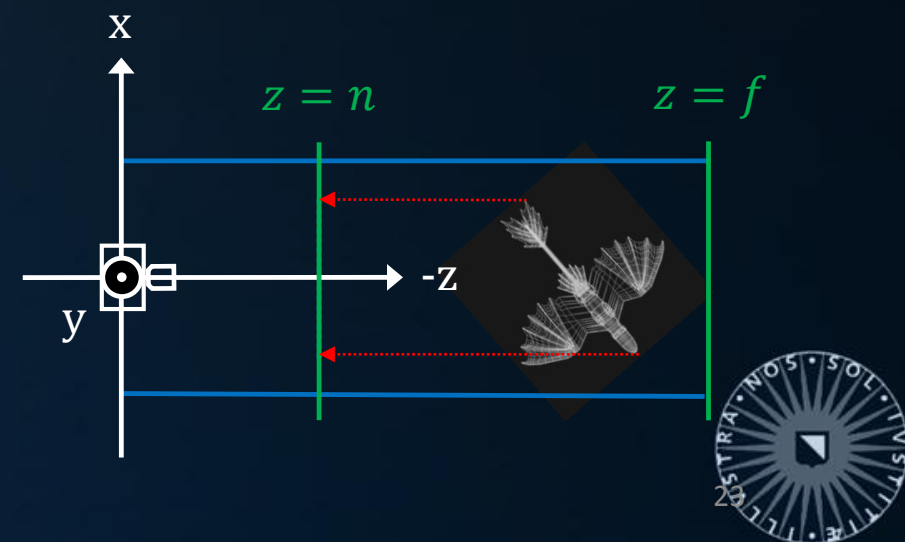
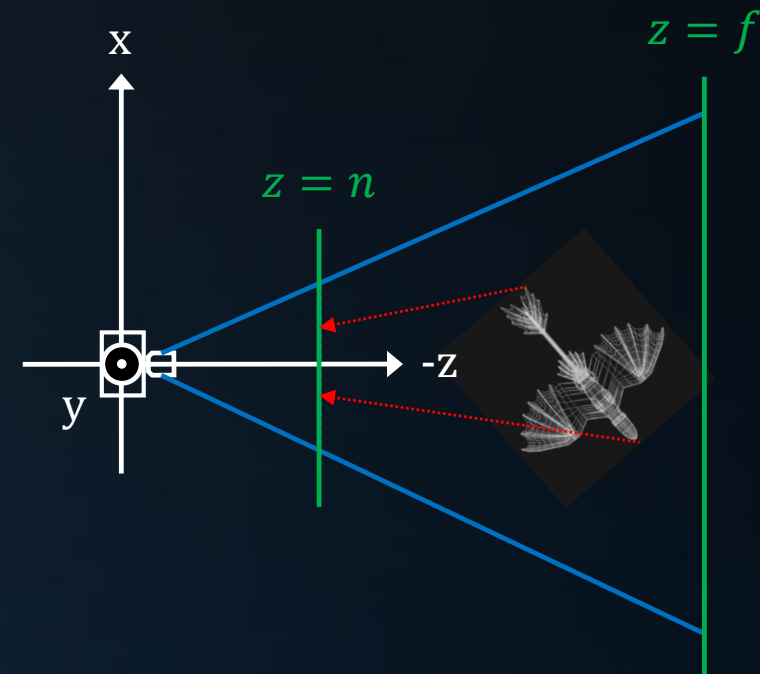
Perspective projection

STEP THREE: camera space to orthographic view volume

The combined transform so far:

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

One thing is still missing: perspective.



Perspective

Perspective projection

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

Perspective projection

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

Perspective projection

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

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, where $w = 1$.



Perspective

Perspective projection

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

Perspective projection

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 ? $\rightarrow z' = n + f - \frac{fn}{z}$

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



Perspective

Perspective projection

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

Perspective projection

To transform a single world vertex we thus apply:

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

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

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

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

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



Today's Agenda:

- Projection
- Pipeline Recap
- Rasterization



```

graph TD
    world((world)) --- T_camera[T_camera] camera((camera))
    world --- T_car1[T_car1] car1((car))
    world --- T_plane1[T_plane1] plane1((plane))
    world --- T_car2[T_car2] car2((car))
    world --- T_plane2[T_plane2] plane2((plane))
    world --- T_buggy[T_buggy] buggy((buggy))

    car1 --- wheel1_1((wheel))
    car1 --- wheel1_2((wheel))
    car1 --- wheel1_3((wheel))
    car1 --- turret1((turret))
    car1 --- dude1((dude))

    plane1 --- wheel2_1((wheel))
    plane1 --- wheel2_2((wheel))
    plane1 --- turret2((turret))

    car2 --- wheel3_1((wheel))
    car2 --- wheel3_2((wheel))
    car2 --- wheel3_3((wheel))
    car2 --- turret3((turret))
    car2 --- dude2((dude))

    plane2 --- wheel4_1((wheel))
    plane2 --- wheel4_2((wheel))
    plane2 --- turret4((turret))

    buggy --- wheel5_1((wheel))
    buggy --- wheel5_2((wheel))
    buggy --- wheel5_3((wheel))
    buggy --- dude3_1((dude))
    buggy --- dude3_2((dude))
  
```

meshes

vertices

Project

vertices

Rasterize

fragment positions

Shade

pixels

Postprocessing



Pipeline Recap



```

0, N );
refl * E * diffuse;
= true;

MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, close
if;
radiance = SampleLight( &rand, I, &t, &align
e.x + radiance.y + radiance.z) > @) && (survive)
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psum;
at3 factor = diffuse * INVPI;
at weight = Mix2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) *
random walk - done properly, closely following
survive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

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

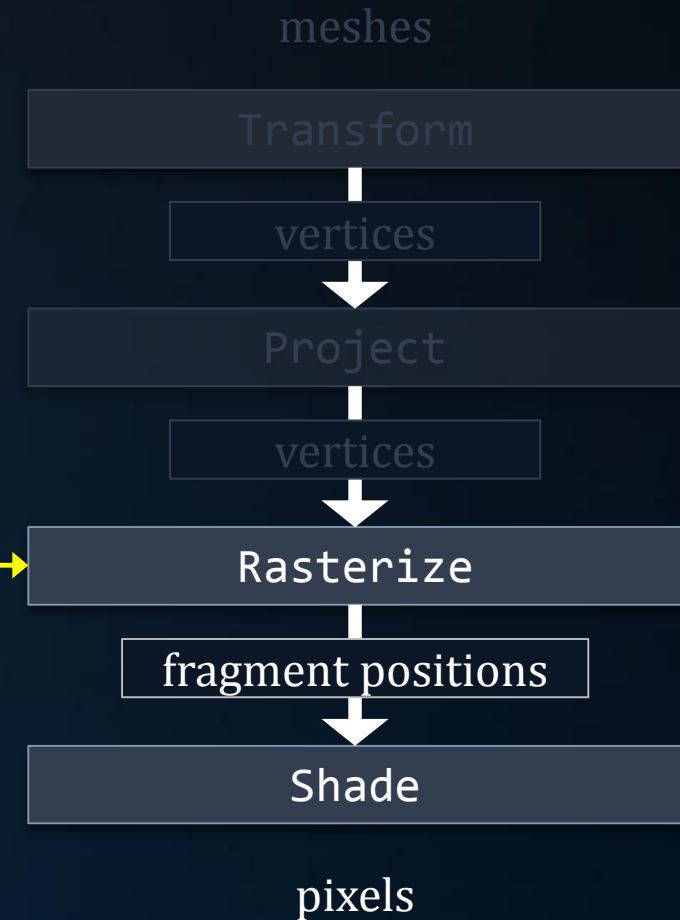
After projection

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*.

connectivity data



Today's Agenda:

- Projection
- Pipeline Recap
- Rasterization



Rasterization

Connectivity data

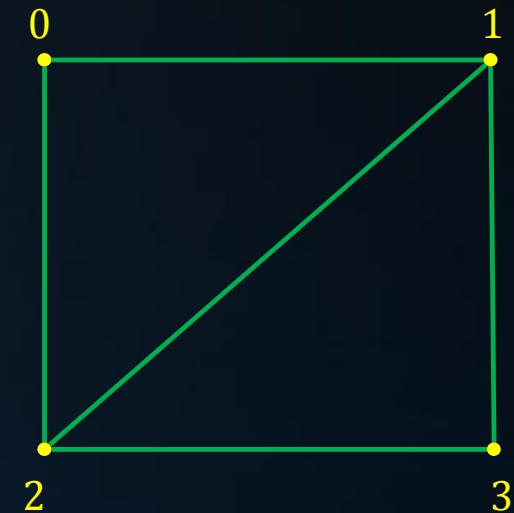
Two triangles forming a quad, using four vertices:

| | | | | | |
|---|---|---|---|---|---|
| 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.



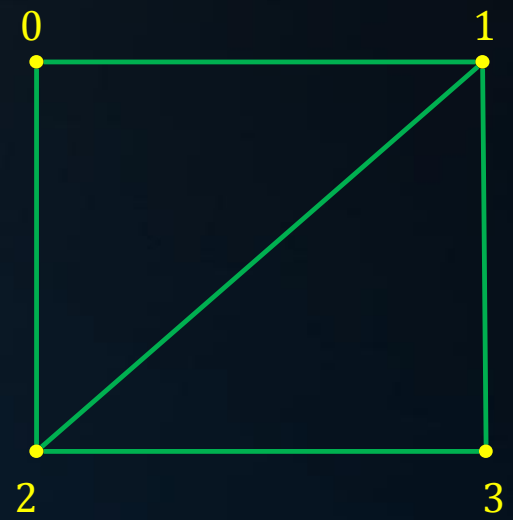
Rasterization

Connectivity data

We can store triangles more efficiently using triangle strips.

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
|---|---|---|---|

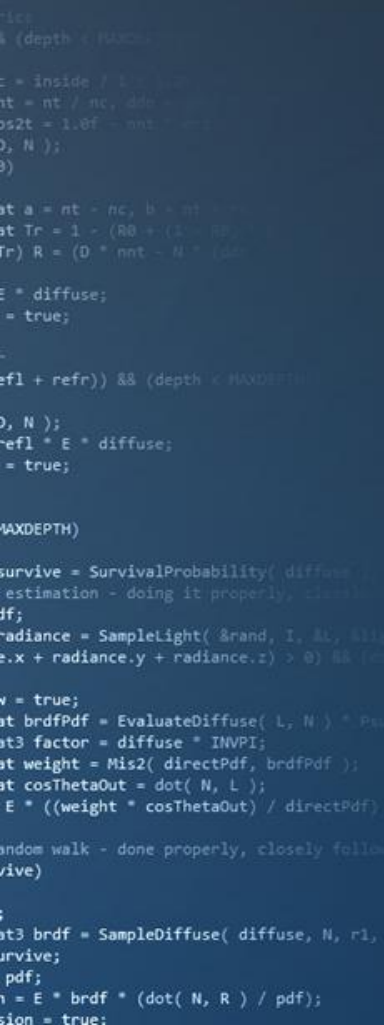
Here, the first three vertex indices specify the first triangle. After that, subsequent triangles use the previous two indices, plus one extra vertex.



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.



Connectivity data

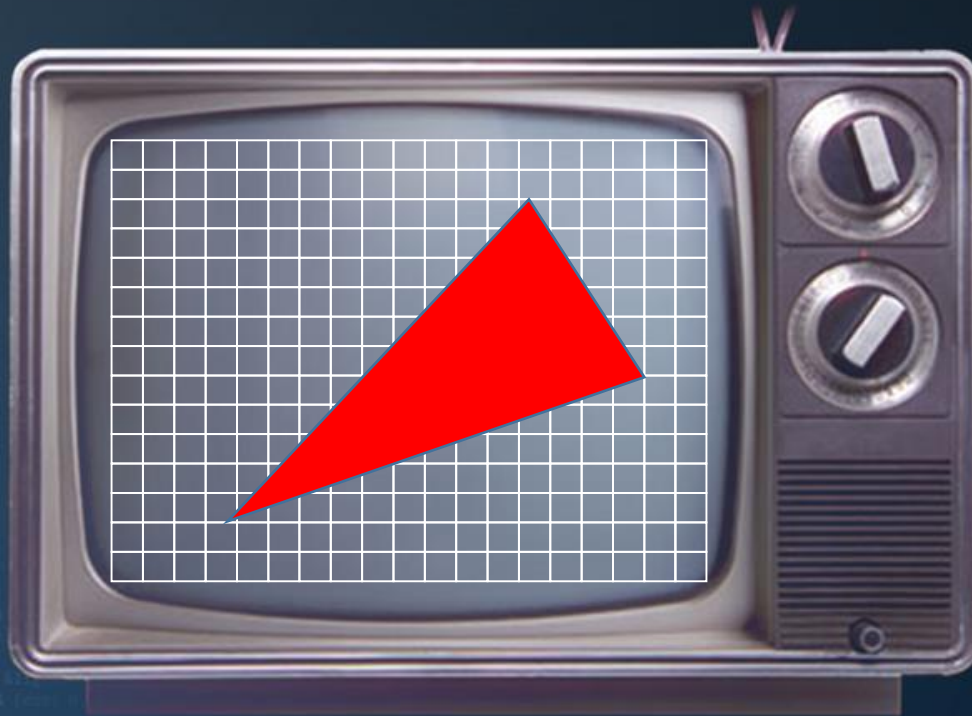


- The memory reduction affects only the connectivity data, which is small compared to vertex data;
- Multiple strips for a single mesh may incur significant overhead in the driver.

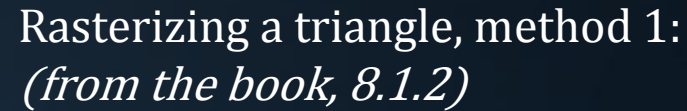


Rasterization

Triangle rasterization



Triangle rasterization

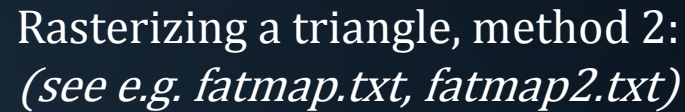


1. Determine the axis-aligned bounding box of the triangle;
2. For each pixel within this box, determine whether it is inside the triangle.

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



Triangle rasterization



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

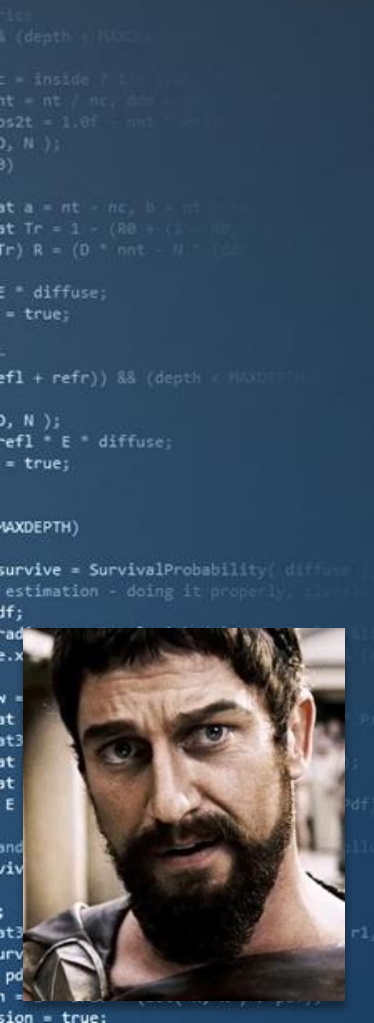
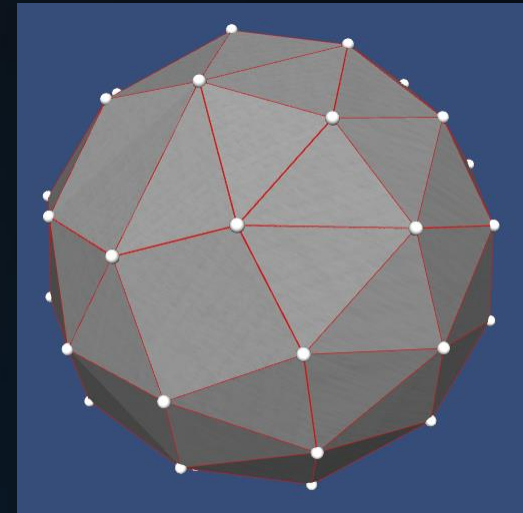
Sanity check

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.

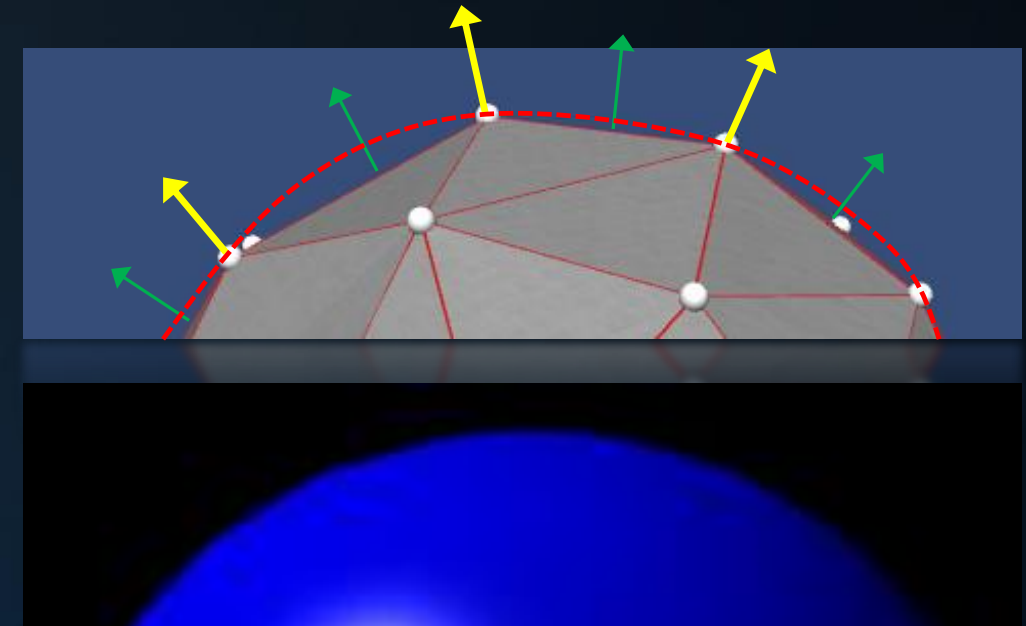
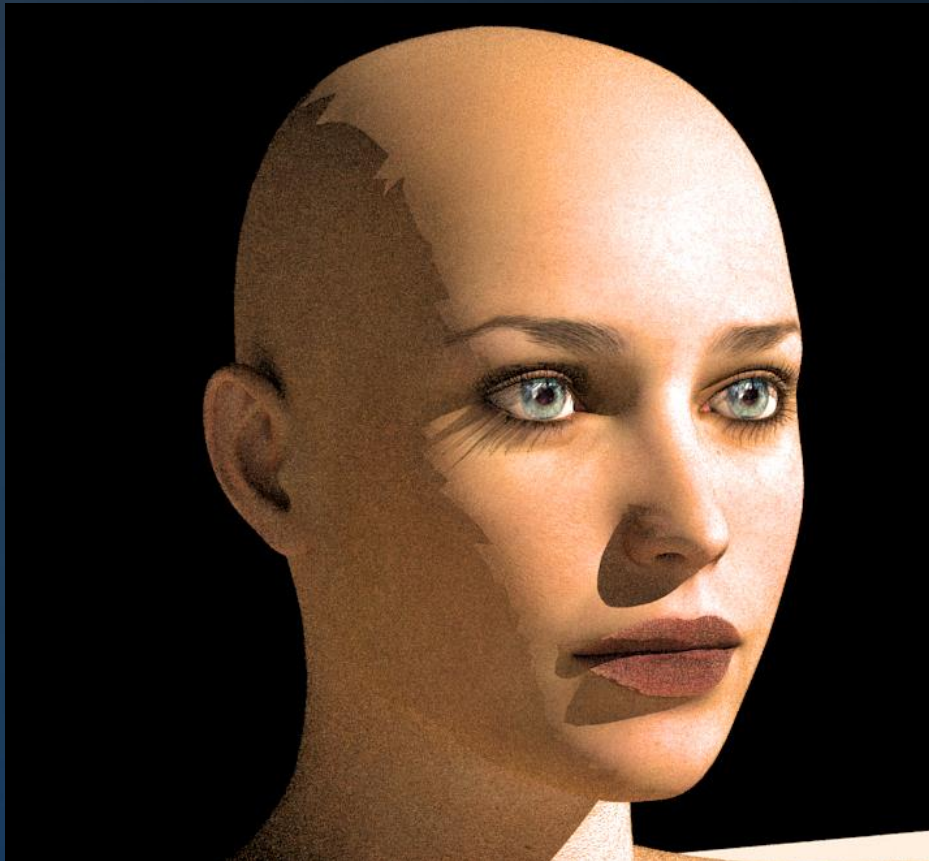


Rasterization

Sanity check

Normal interpolation can cause some bad behavior:

```
...ics
& (depth < MAXDEPTH)
{
    // Inside / Outside test
    int nt = nt / nc;
    float cos2t = 1.0f - nnt * nnt;
    if (cos2t < 0.0f)
        return 0;
    float a = nt - nc, b = nt + nc;
    float Tr = 1 - (R0 + (1 - R0) * cos2t);
    float R = (D * nnt - N * (cos2t));
    // Diffuse
    // Refractive
    if (refl + refr) && (depth < MAXDEPTH)
    {
        // Refractive
        float D, N;
        float refl * E * diffuse;
        // Refractive
        // Refractive
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, clearly
    if;
    rad
    e.x
    w =
    at
    at3
    at
    at
    at
    E
    and
    viv
    at3
    urv
    po
    n =
    ion = true;
```



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



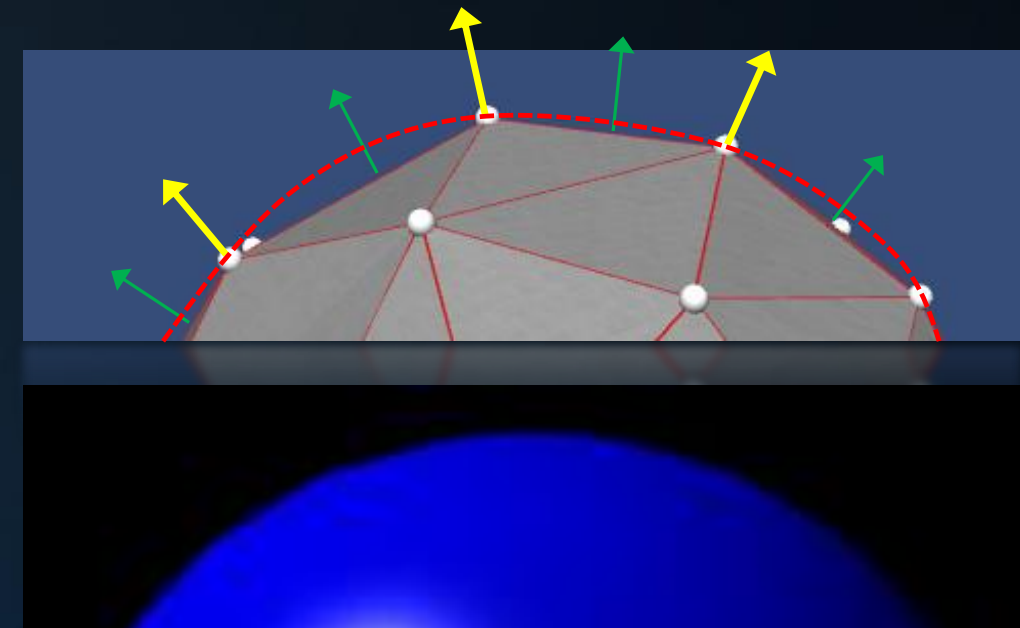
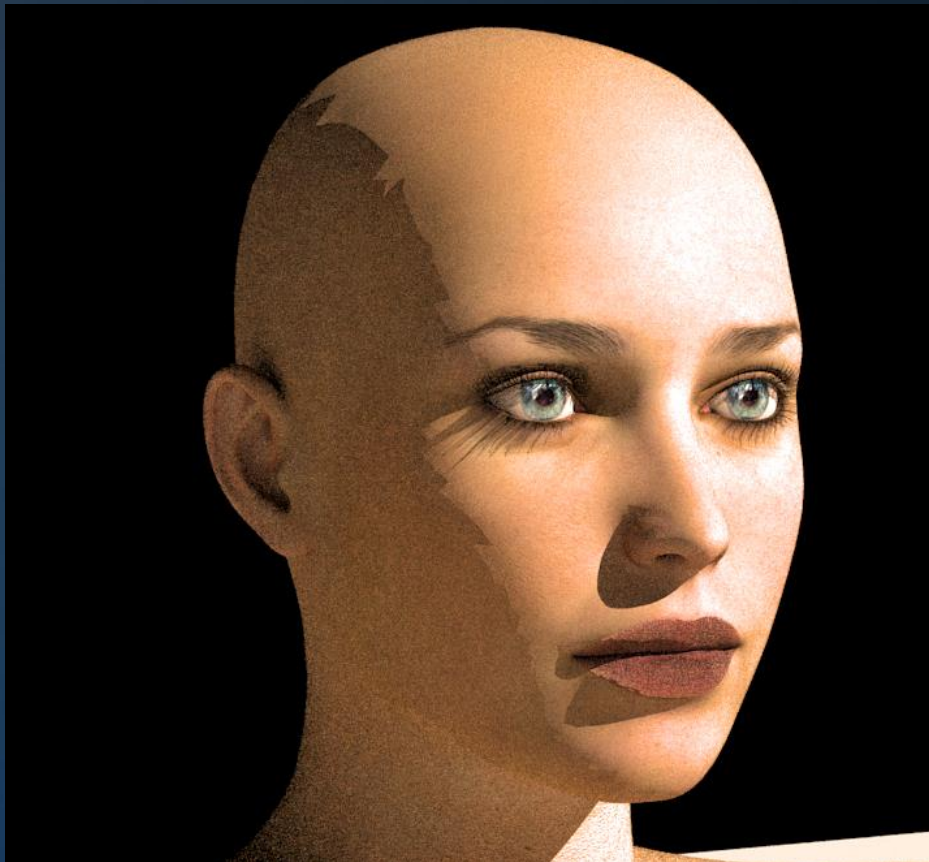
Rasterization

Sanity check

Normal interpolation can cause some bad behavior:

```
...ics
& (depth < MAXDEPTH)
{
    // Inside / Outside test
    int nt = nc / ncddo;
    float cos2t = 1.0f - nnt * nnt;
    float D, N;
    // ...
    float a = nt - nc, b = nt - nc;
    float Tr = 1 - (RB + (1 - RB) * cos2t);
    float R = (D * nnt - N * cos2t);
    // ...
    E * diffuse;
    // ...
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N;
        refl * E * diffuse;
        // ...
        MAXDEPTH)
    }
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, clearly
    if;
    rad
    e.x
    w =
    at
    at3
    at
    at
    E
    and
    viv
    at3
   urv
    po
    n =
    ion = true;

```



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



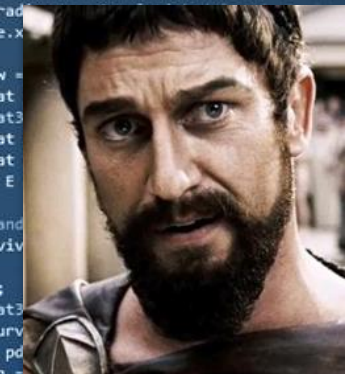
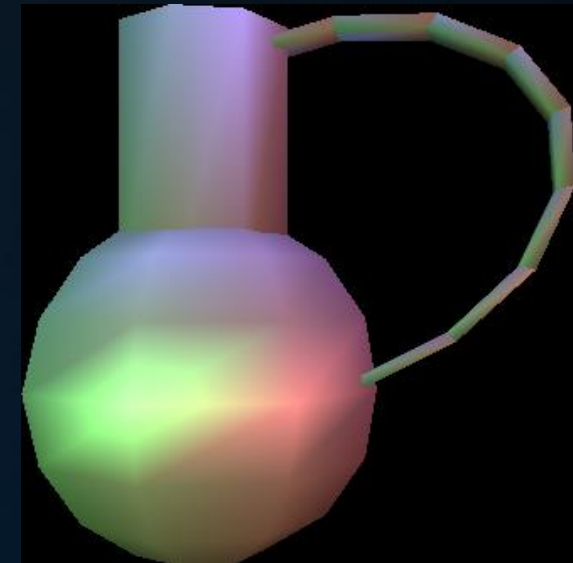
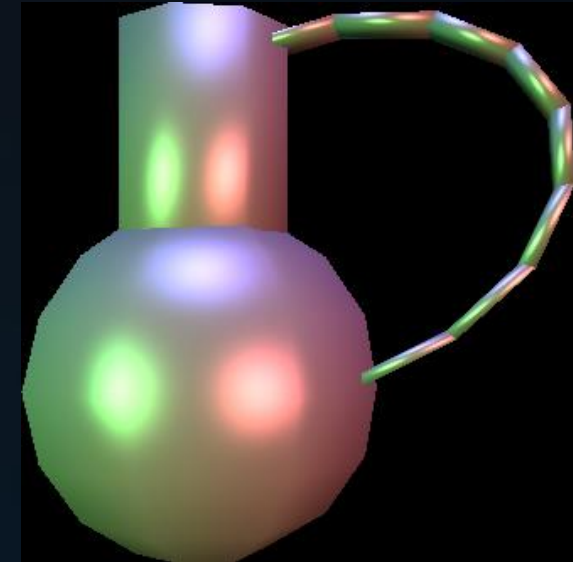
Rasterization

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

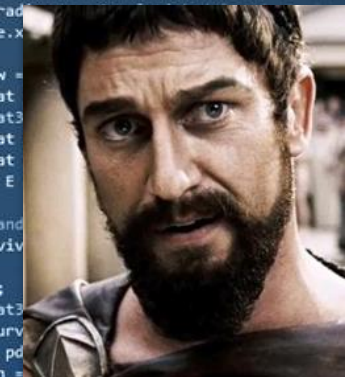
Shading:

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:

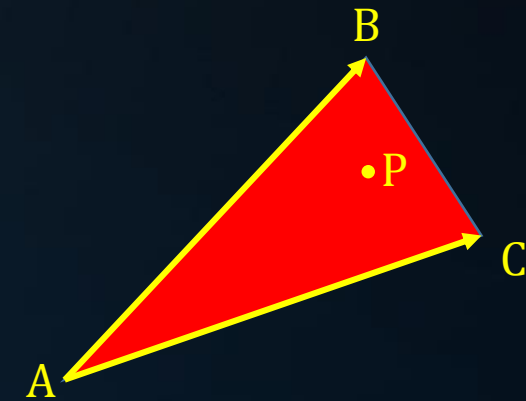
$$P(\lambda_1, \lambda_2) = A + \lambda_1(B - A) + \lambda_2(C - A)$$

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

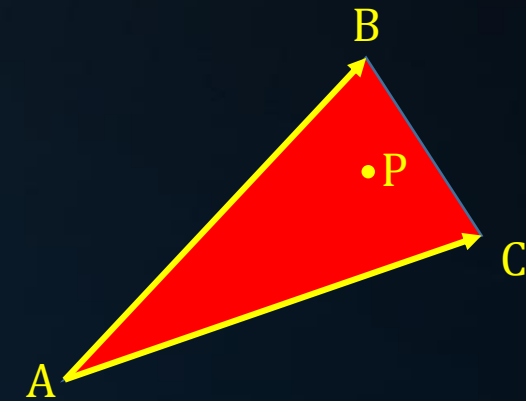
Triangle rasterization

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

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:

- Projection
- Pipeline Recap
- Rasterization



INFOGR – Computer Graphics

J. Bikker - April-July 2015 - Lecture 6: "Transformations"

END of "Transformations"

next lecture: "Visibility"

