3D Graphics for Dummies

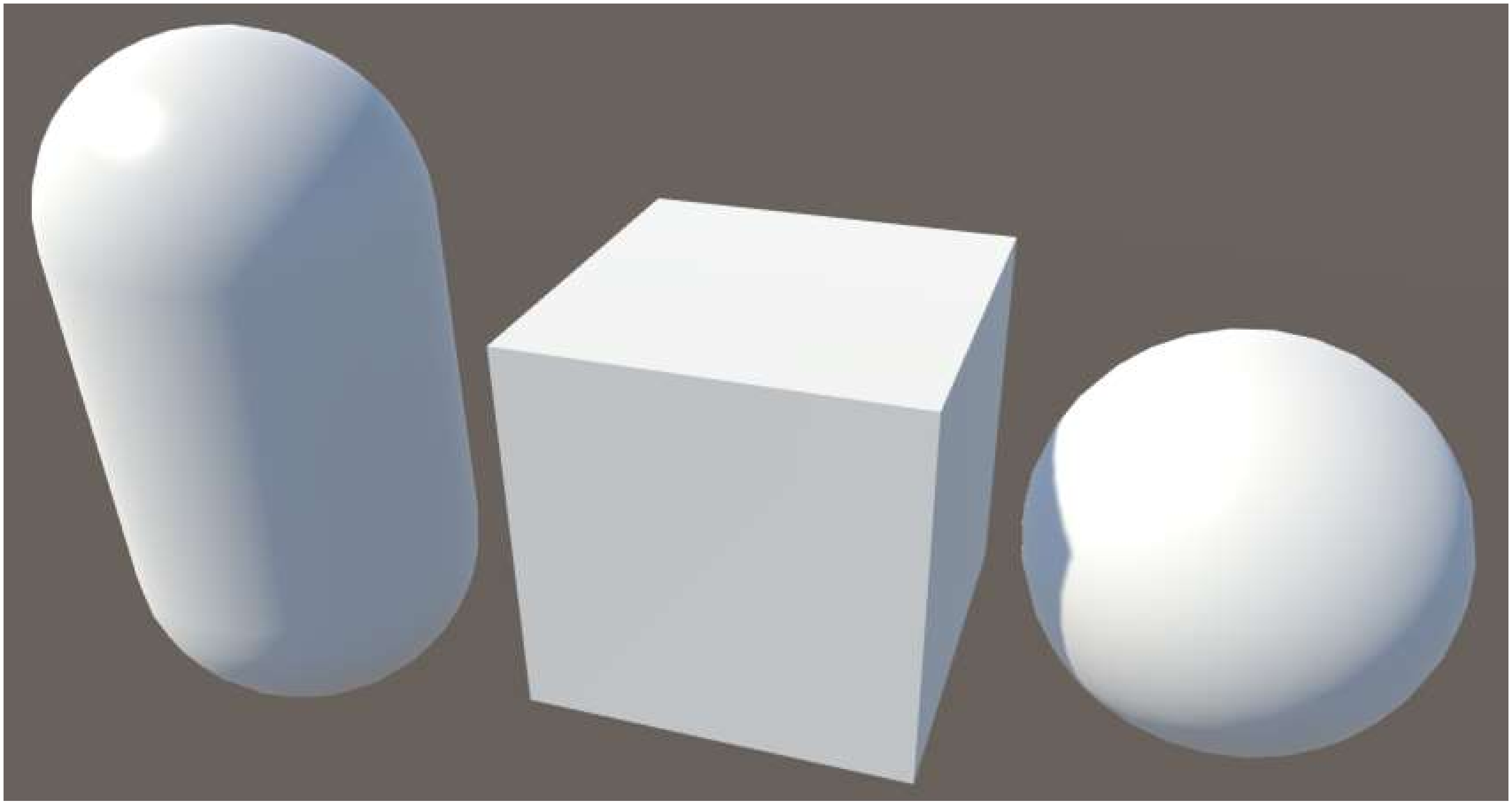
Significant content “borrowed” from Dan Chang @ Nintendo NTD “with permission”

Chris Ryan

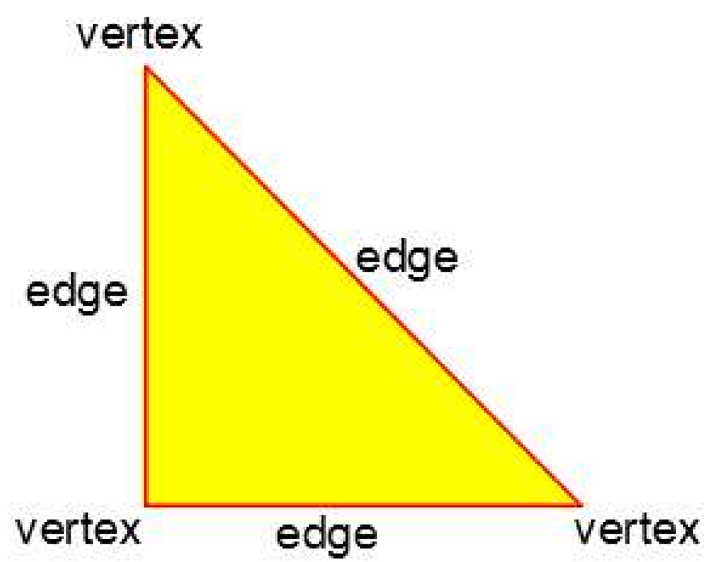
Northwest C++ User Group Nov 18th 2020

github.com/ChrisRyan98008/NwCpp-Nov2020

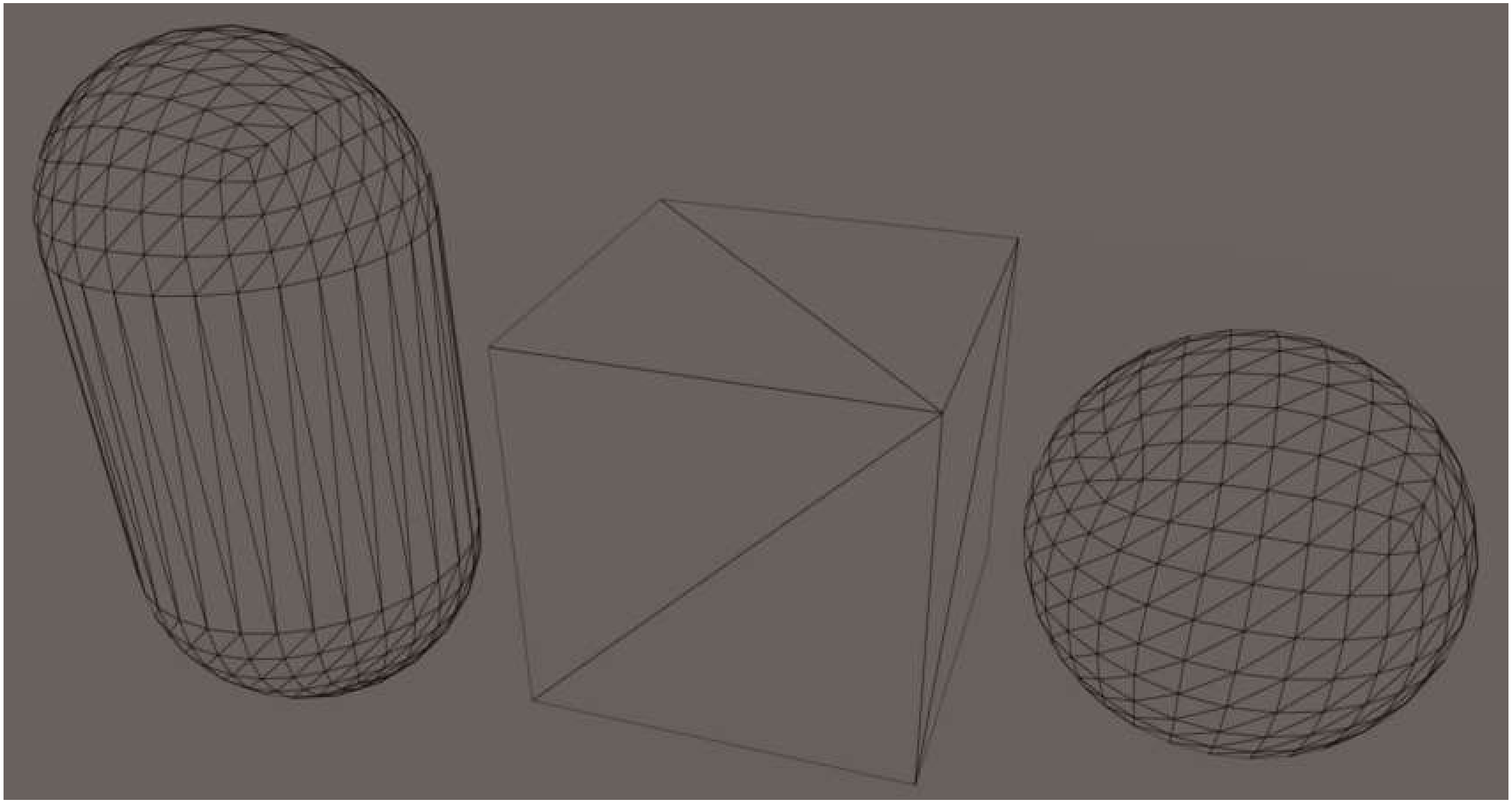
3D Graphics for Dummies

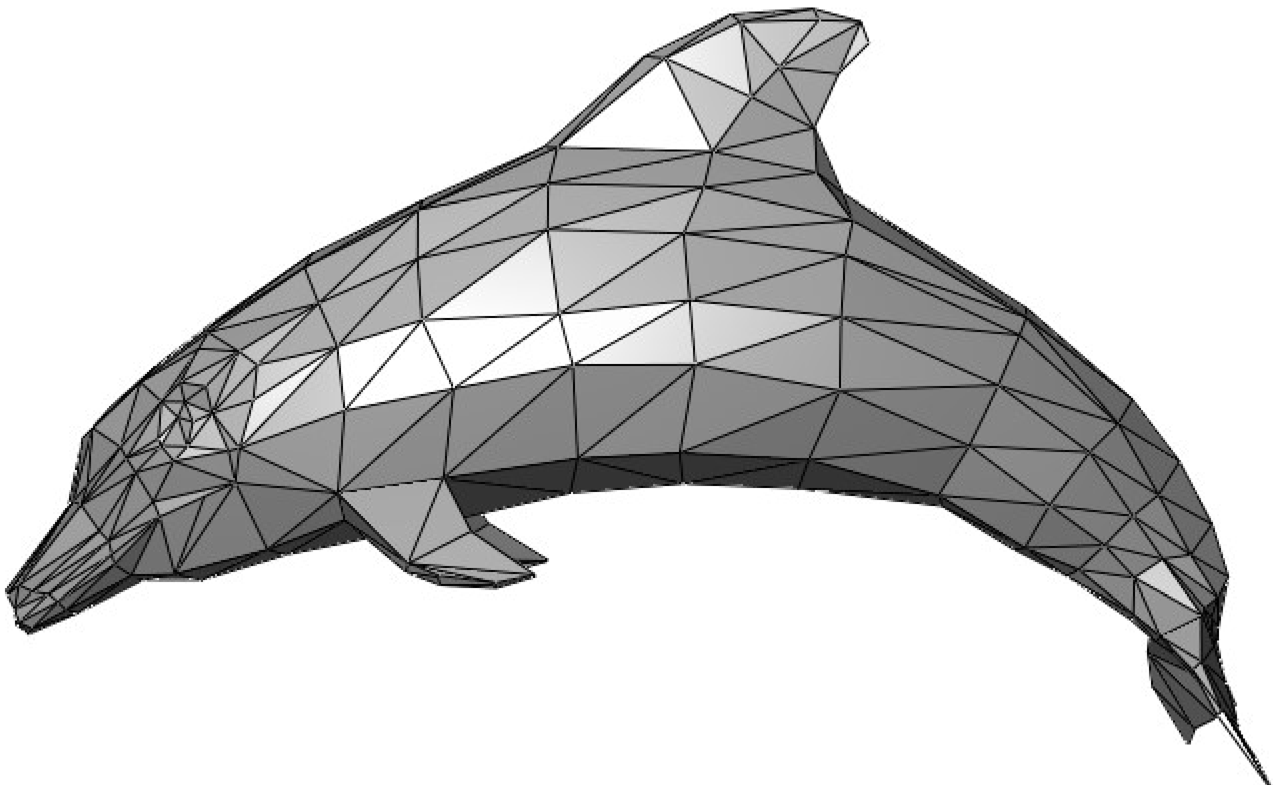


Draw Lots of Triangles

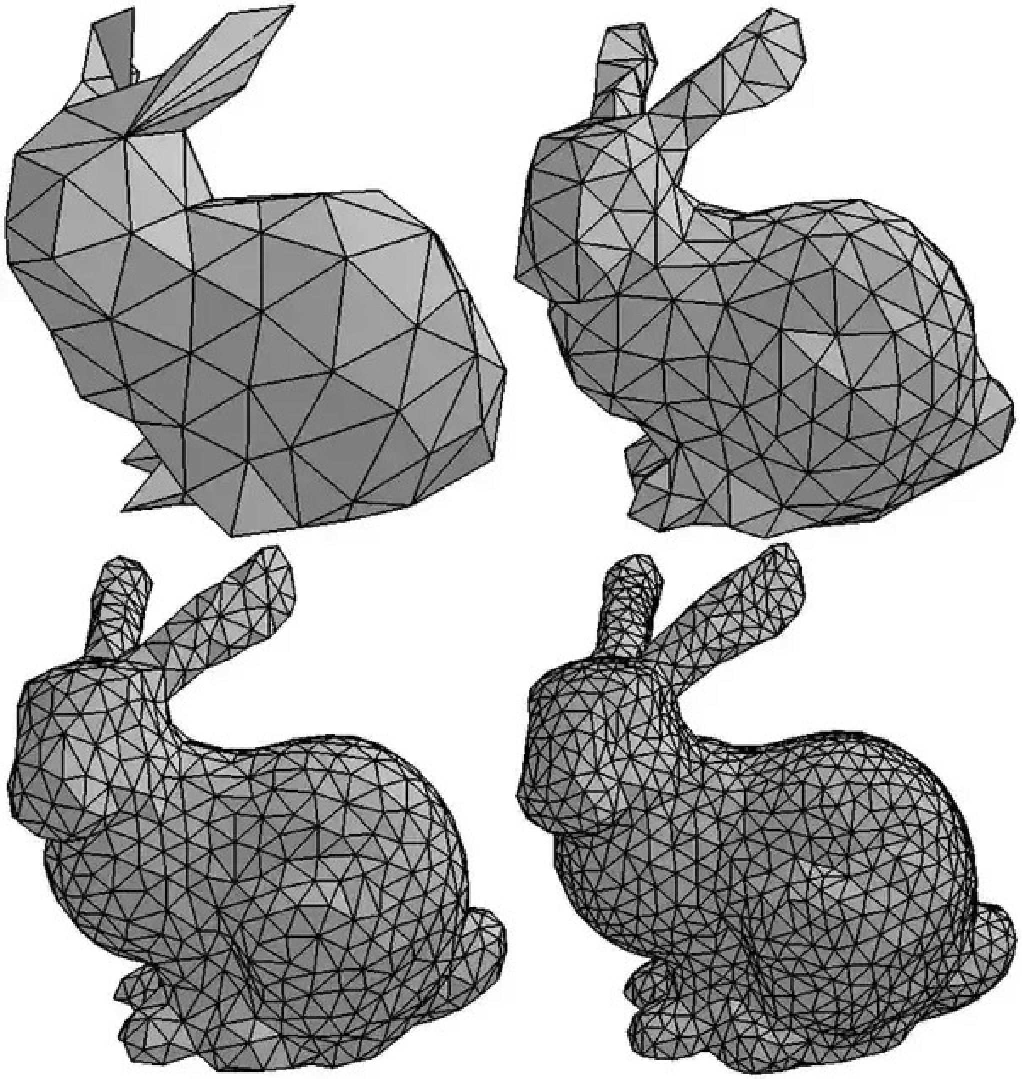


sometimes also called **polygons** or **polys**

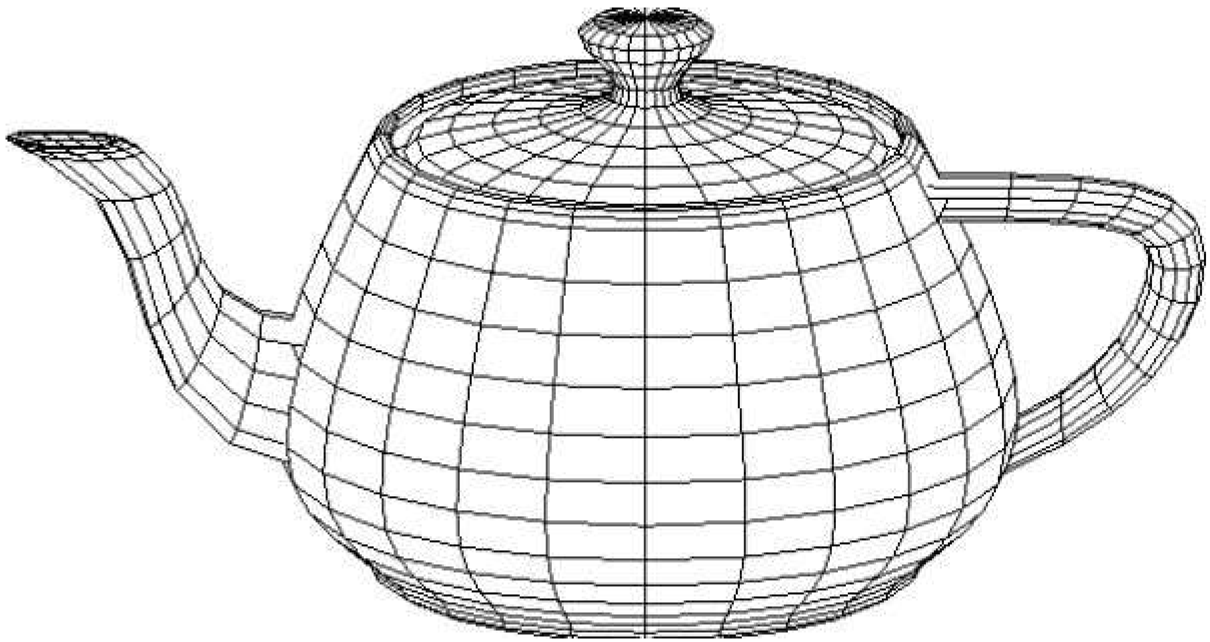


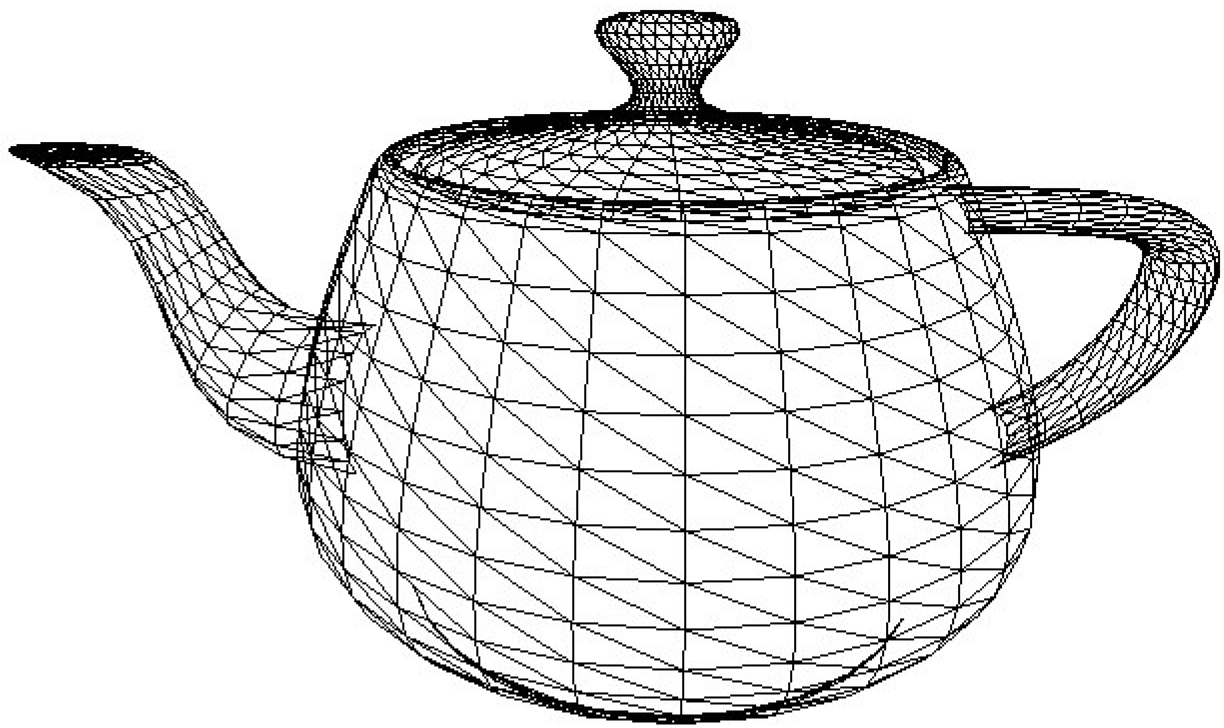


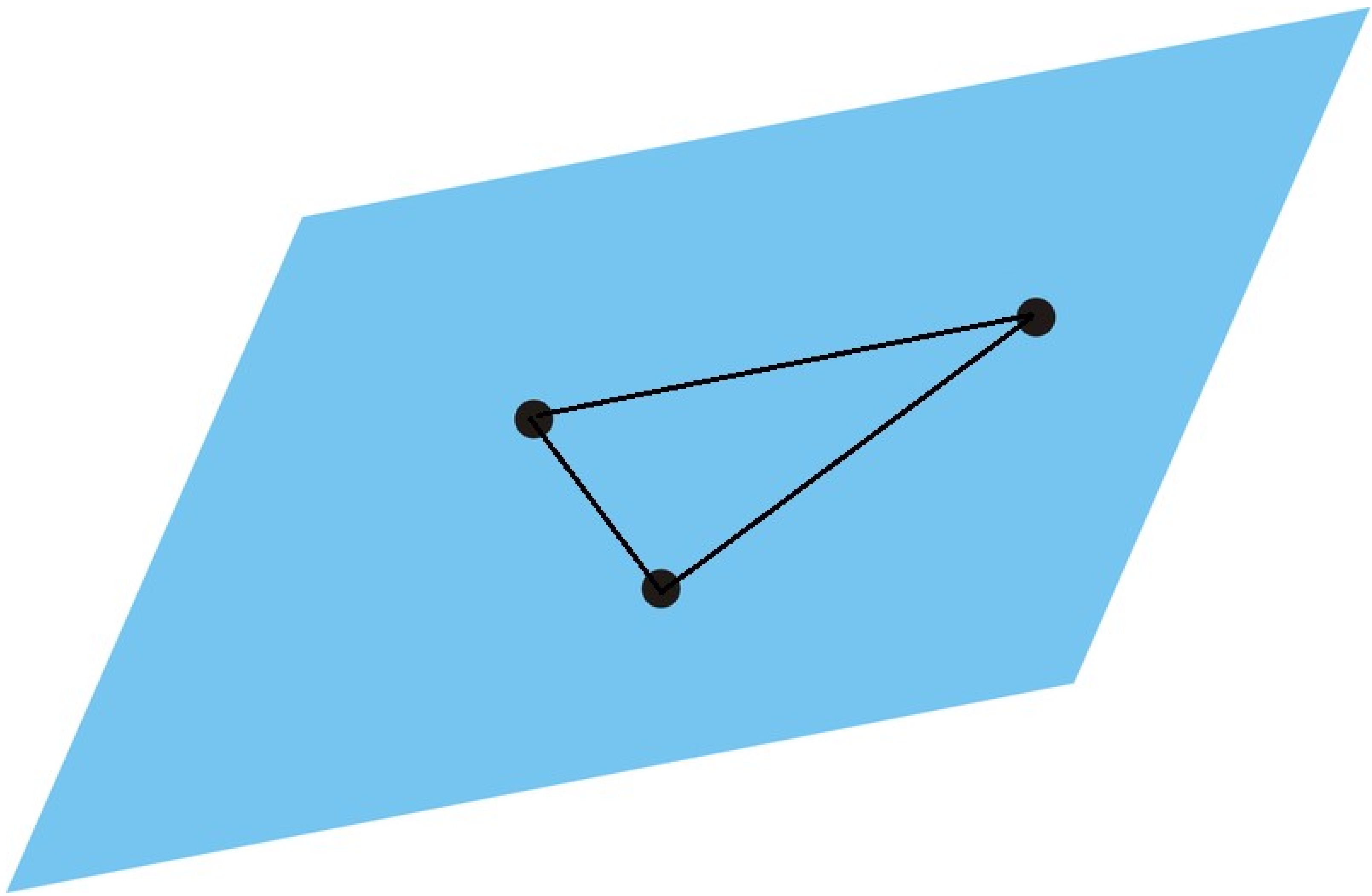
3D Graphics for Dummies

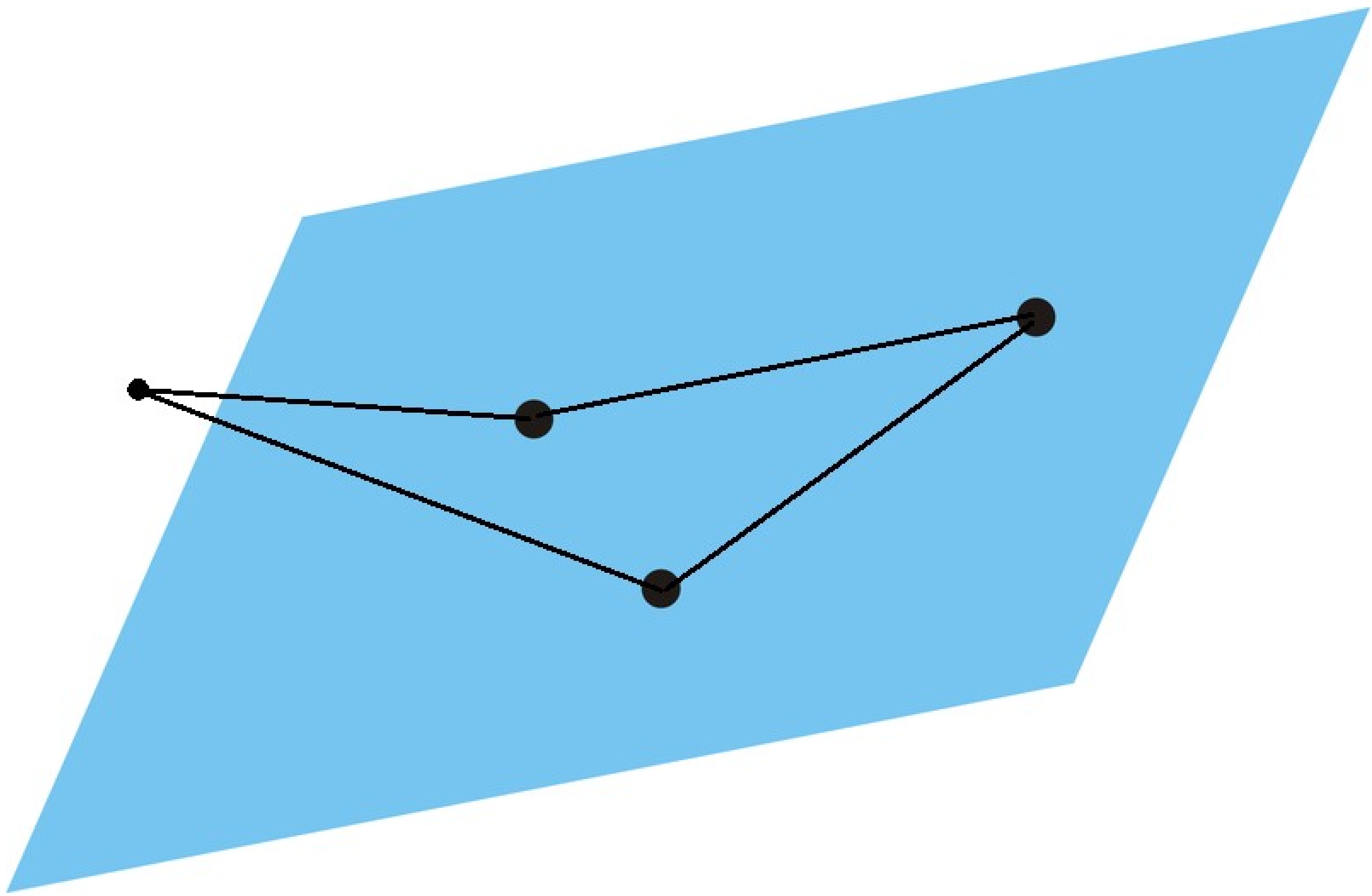


3D Graphics for Dummies



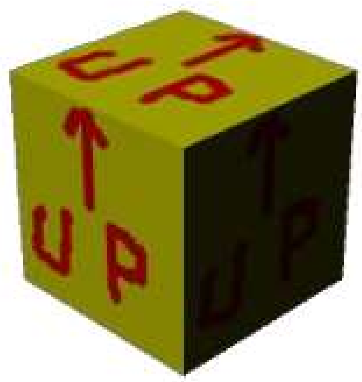






A Model

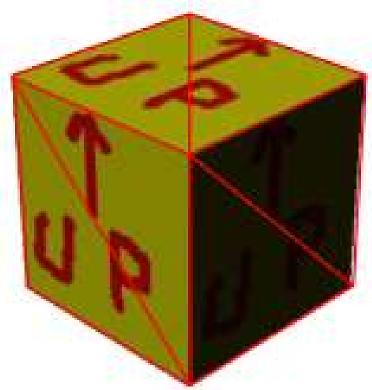
- Only care about visible surfaces (no data about interior)



- Sometimes also called a **3D object** or an **object**

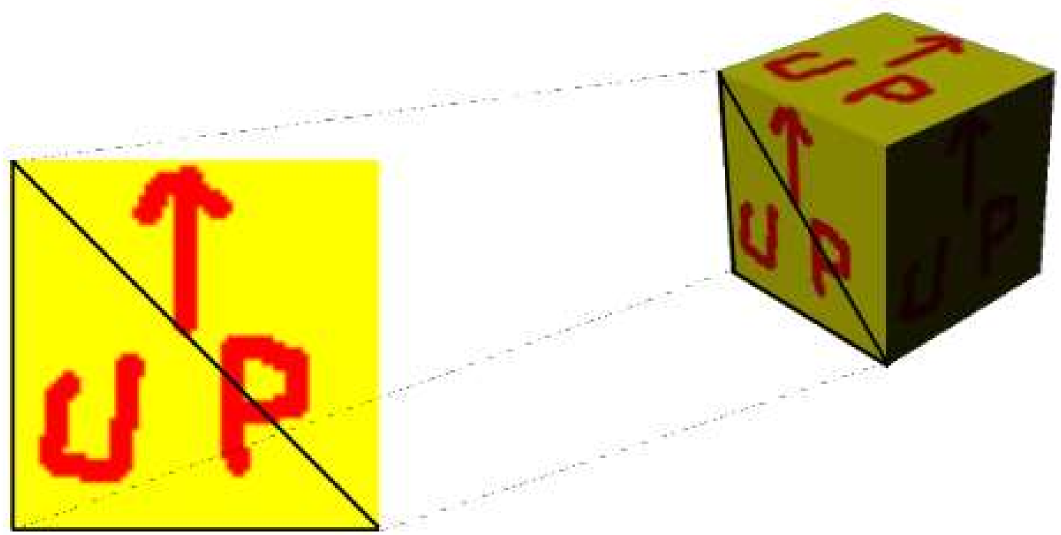
A Model

- Surfaces represented with a collection of triangles
- **Mesh** - a collection of triangles
- **Model** - a collection of meshes



Textures

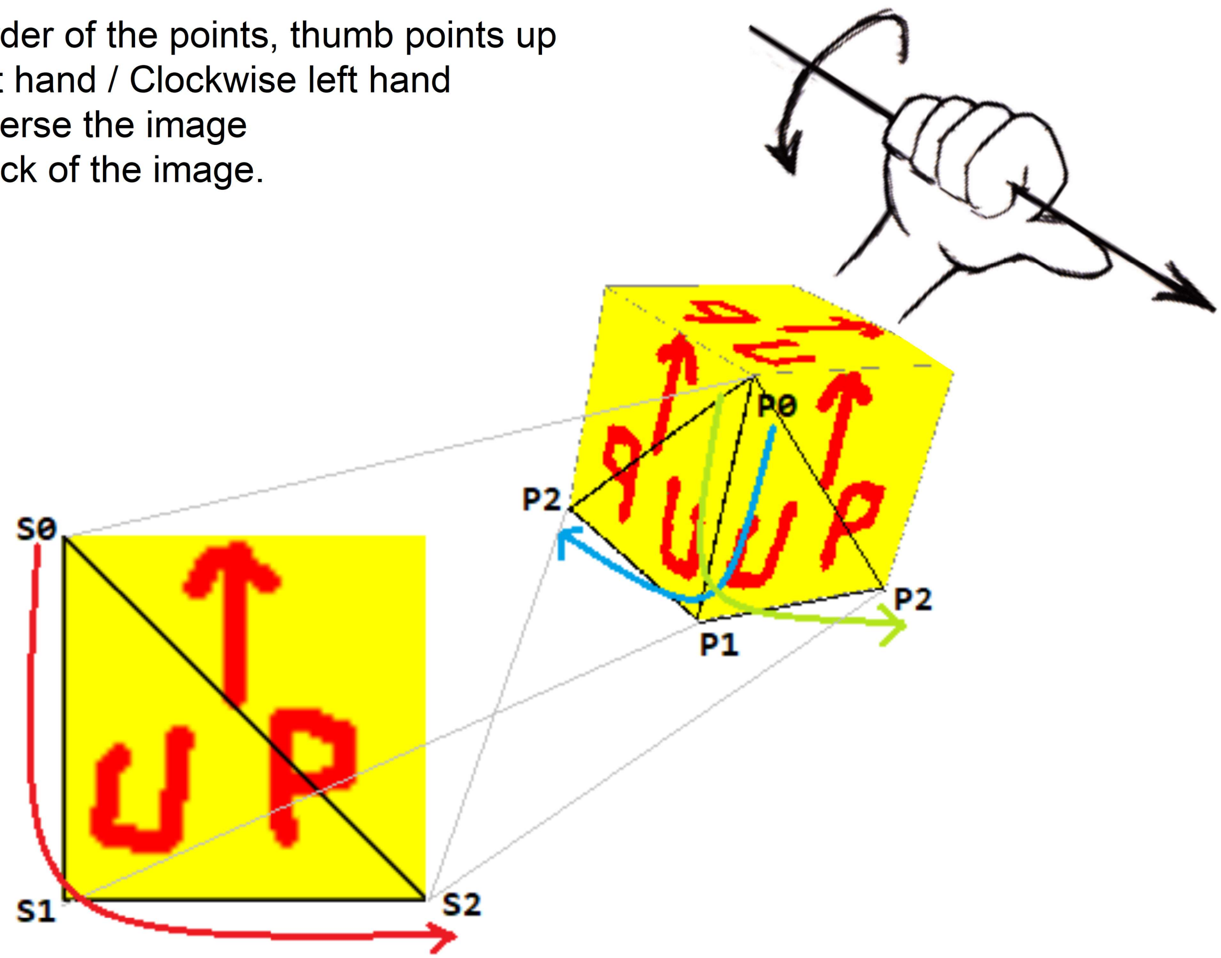
- Are 2D images which are applied to triangles
- Rough analogy: stickers or decals
  - Except textures can be stretched
  - Each triangle specifies which part of texture gets applied to it



3D Graphics for Dummies

Right Hand / Left Hand Rule / Winding

Fingers curled in the order of the points, thumb points up
Counter clockwise right hand / Clockwise left hand
Opposite directions reverse the image
Looking through the back of the image.



Model has Matrix

- Every model has a `Matrix`
  - Specifies Position in the world
  - Specifies Orientation in the world
  - Specifies Scale (size) in the world
- A `Matrix` used in this way is also called an "**model to world space** matrix"
  - or "**model to world** matrix"
  - or sometimes just "**world** matrix"

Coordinate Spaces Overview

- Model Space

↓

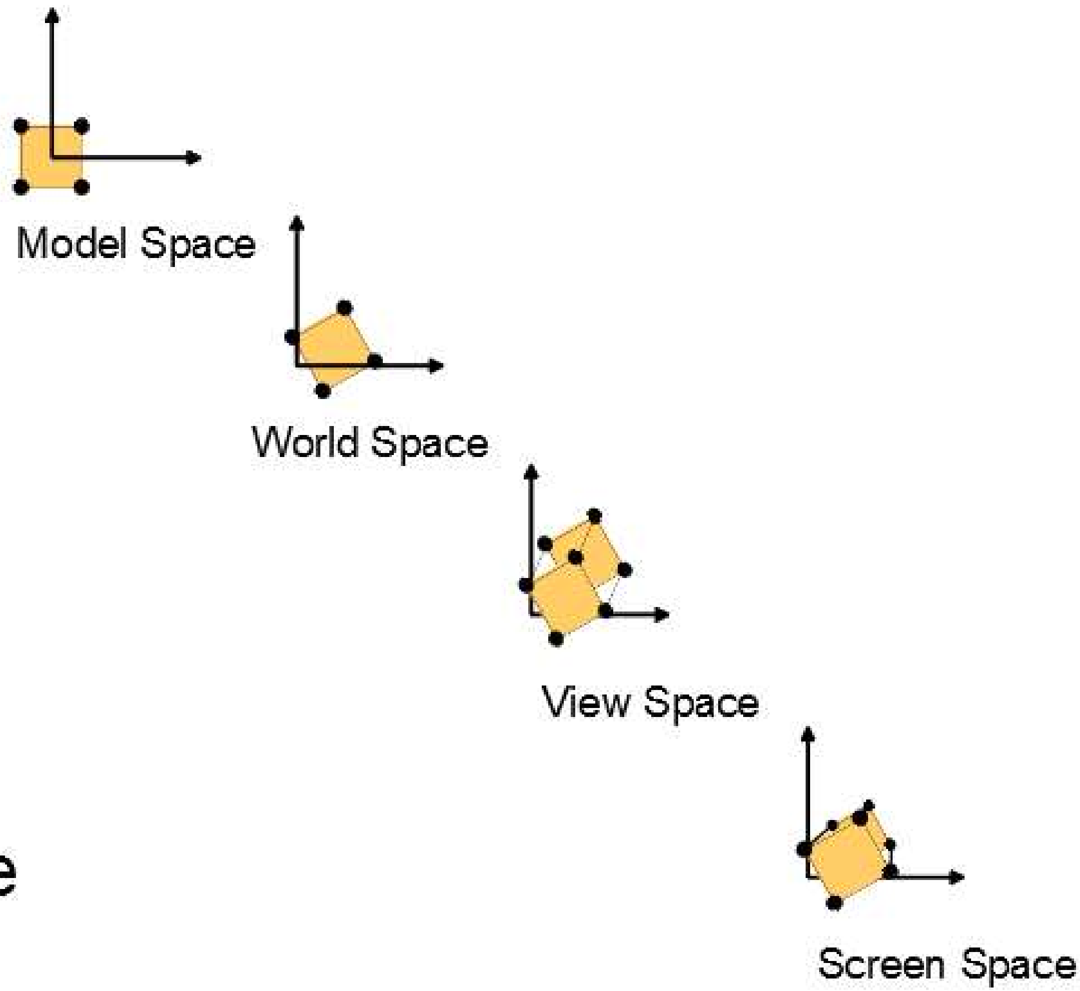
- World Space

↓

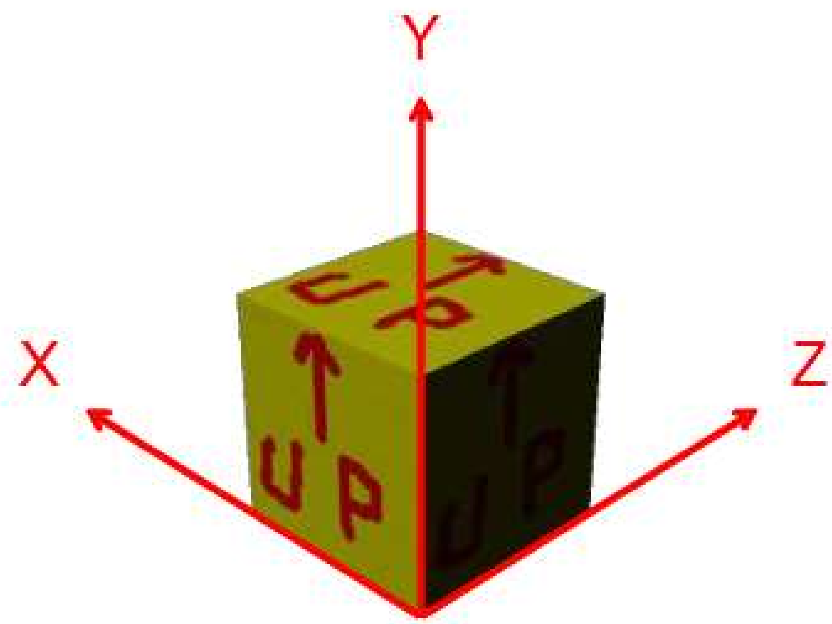
- View Space

↓

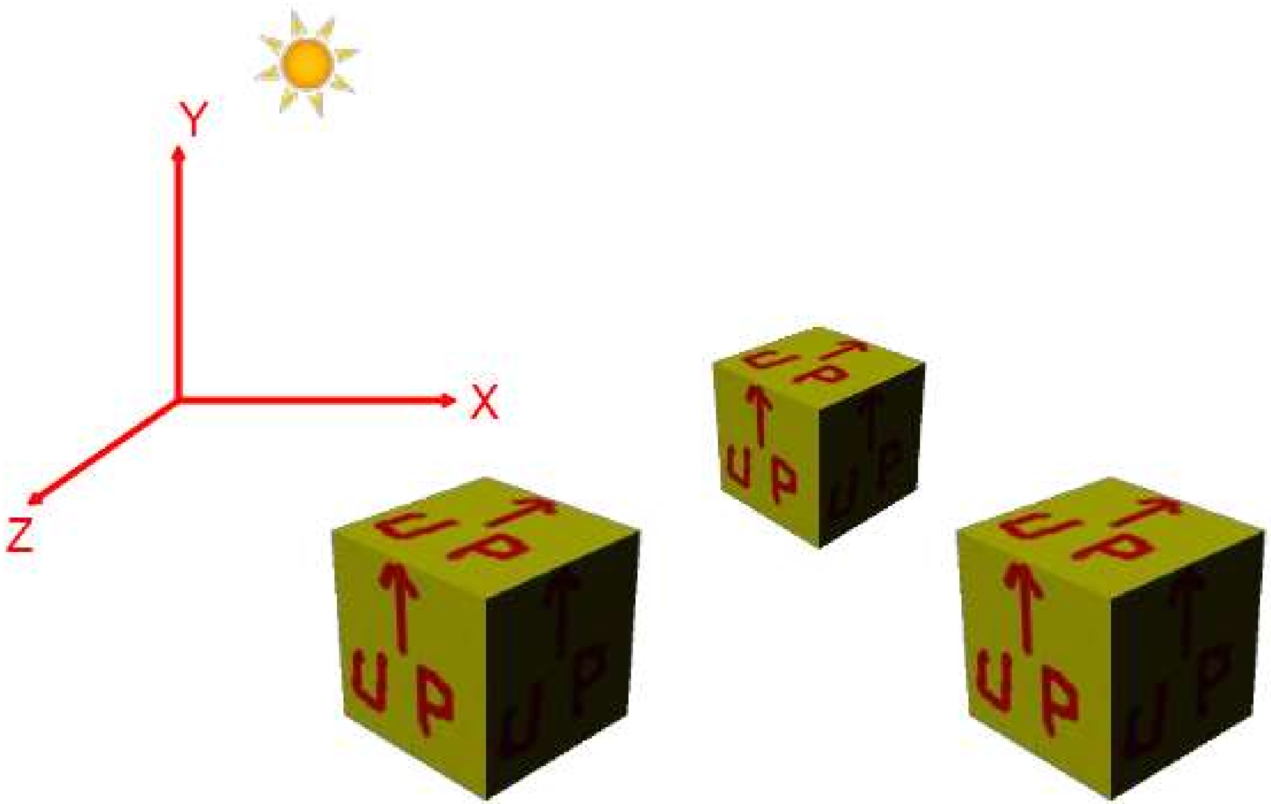
- Screen Space



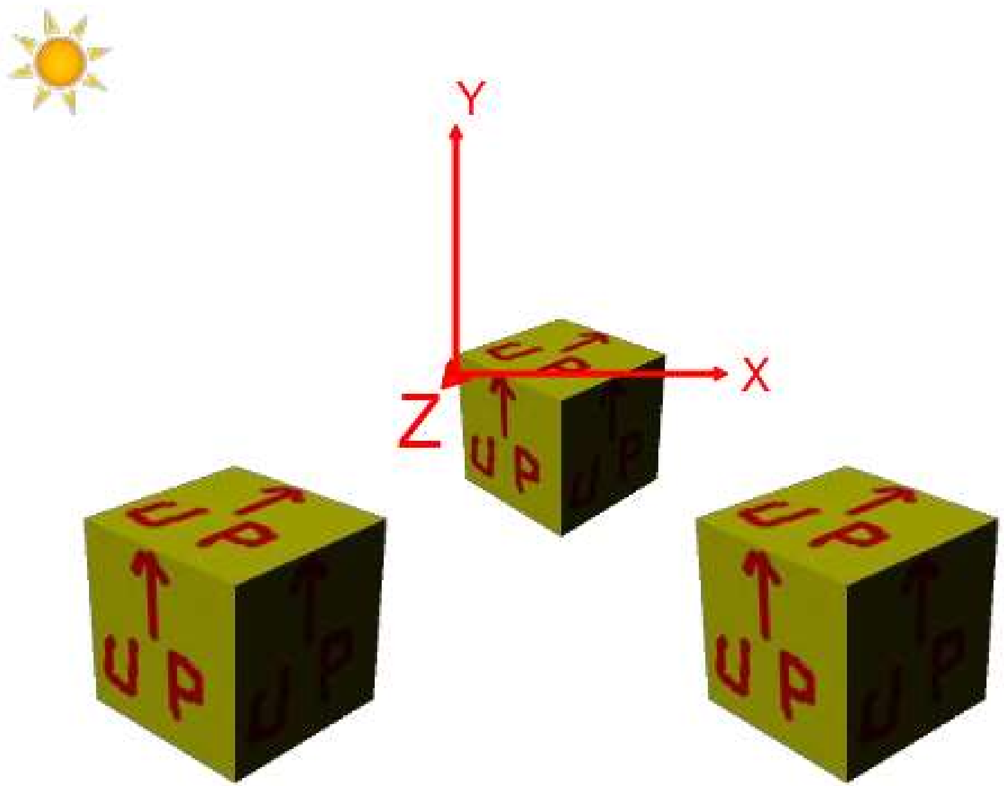
Model Space



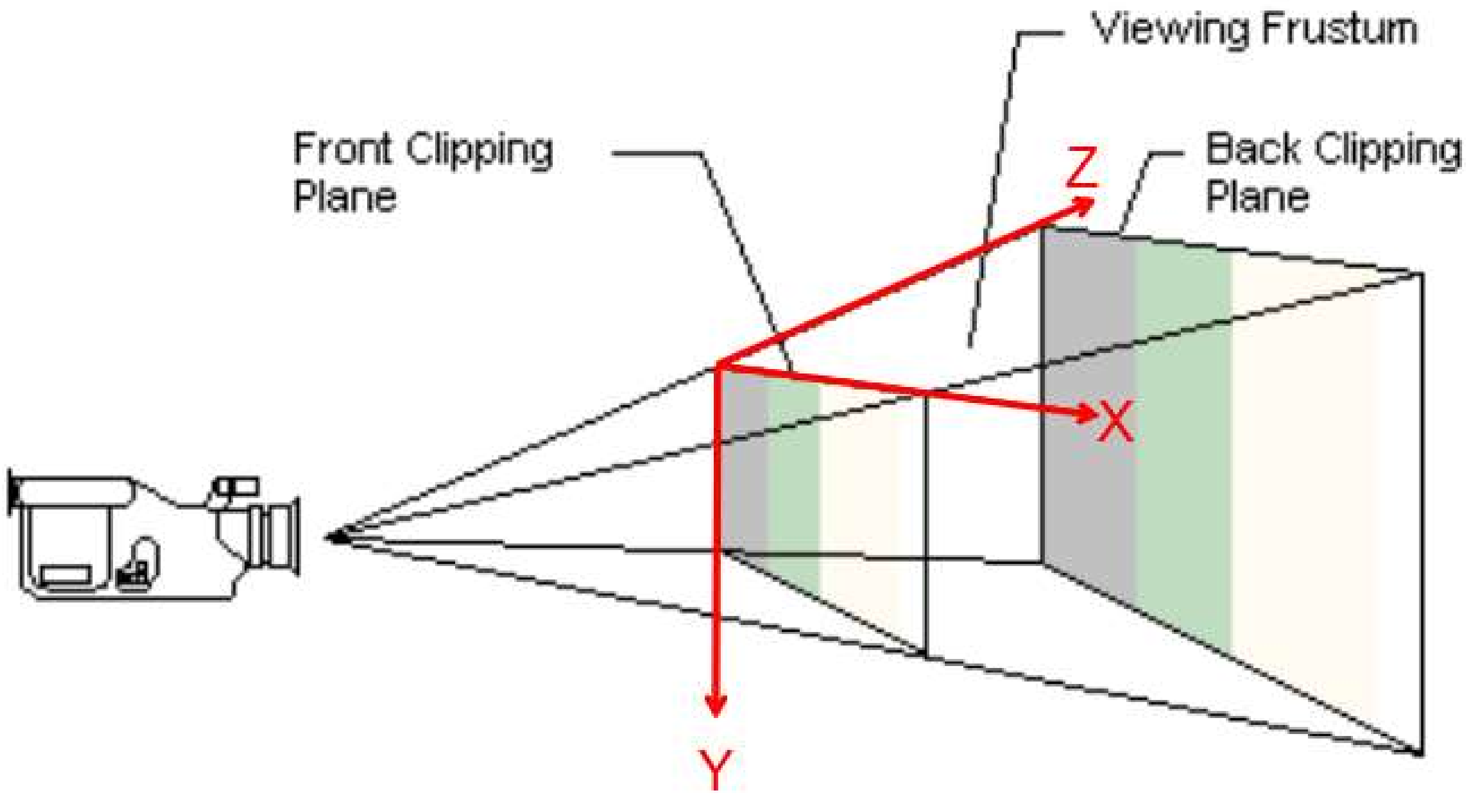
World Space



View Space

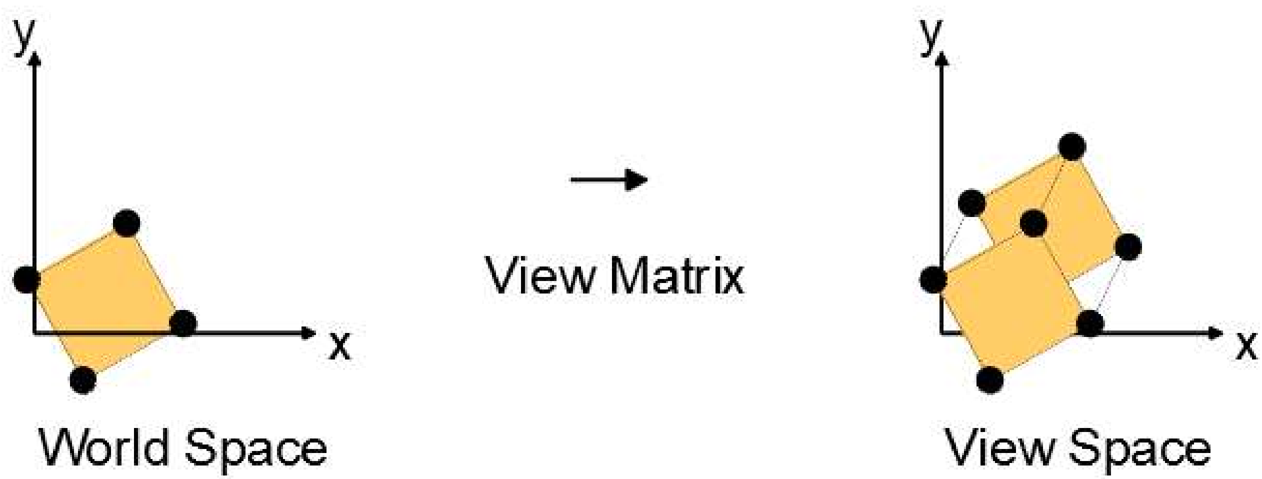


Screen Space



World Space to View Space

- View Matrix ("oriented from camera view")
- World Space $\times$ View Matrix = View Space
- View Space sometimes called Camera Space



Creating View Matrices

```
PointOfView(Point cameraPosition,
            Point cameraTarget,
            Vector cameraUpVector);
```

- **`cameraPosition`** - **position** of the camera in the world
- **`cameraTarget`** - the **position** in the world we're looking at
- **`cameraUpVector`** – the camera's up **vector**

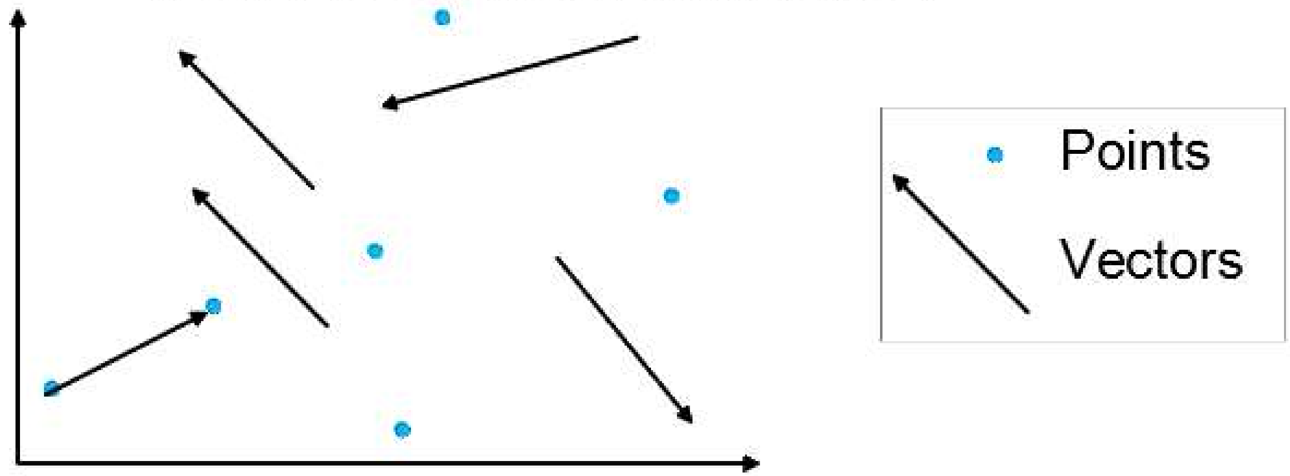
Creating Projection Matrices

```
FieldOfView(float fieldOfView,
            float aspectRatio,
            float nearPlaneDistance,
            float farPlaneDistance);
```

- `fieldOfView` - Field of view in the y direction
- `aspectRatio` - Width divided by height
- `nearPlaneDistance` - Distance to the near plane
- `farPlaneDistance` - Distance to the far plane

Points and Vectors

- Compare points and vectors:
  - Point has only one property: location
  - Vector has two properties: length & direction
    - One interpretation: vectors get you from one point ("tail") to another point ("head")

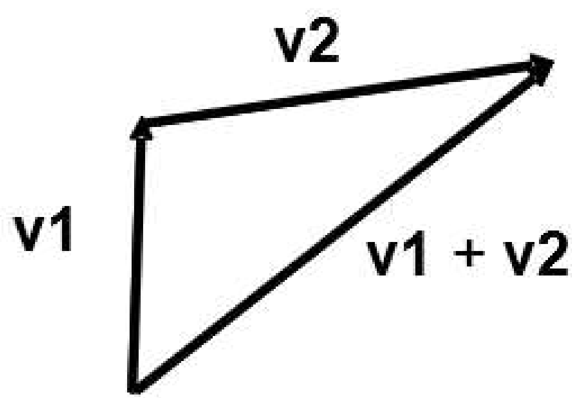


Points and Vectors

- Usually write either this way: (3, 4, 5)
  - For point in 3D, write as (3, 4, 5, 1)
  - For vector in 3D, write as (3, 4, 5, 0)
- When you subtract two points, you get a vector!
  - Adding two points is undefined
- When you add or subtract two vectors, you get a vector

Vector addition

- Takes two vectors as input (**v1** and **v2**)
  - (*v1x*, *v1y*, *v1z*)
  - (*v2x*, *v2y*, *v2z*)
- Output is a vector
- Place vectors "head to tail"
- Output = (*v1x* + *v2x*, *v1y* + *v2y*, *v1z* + *v2z*)



Skipping:

- Vector * scalar
- Vector Length
- Normalize Vector
- Vector Dot Product
- Vector Cross Product
- Reflection Vector
- Triangle and Plane Normals
- Plane Equation
- “Square up” vectors

Row Vectors

- MonoGame uses row vectors
  - Written (3, 4, 5, 0), or [3 4 5 0]
- Unity uses column vectors
  - Written $(3, 4, 5, 0)^T$, or

$$\begin{bmatrix} 3 \\ 4 \\ 5 \\ 0 \end{bmatrix}$$

Matrix Math

$$\mathbf{v} * \mathbf{M} = [vx\ vy\ vz\ vw] * \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix}$$

$$= [(vx*m11 + vy*m21 + vz*m31 + vw*m41)\ (vx*m12 + vy*m22 + vz*m32 + vw*m42)\ (vx*m13 + vy*m23 + vz*m33 + vw*m43)\ (vx*m14 + vy*m24 + vz*m34 + vw*m44)]$$

3D Graphics for Dummies

```
Supported operations
type correct and type coherent: vector, point, matrix, mesh

types:
    matrix  (4x4)
    vector  (1x4) (xyz,w=0)
    point   (1x4) (xyz,w=1)
    mesh    (poly collection)

matrix operators
    matrix *= matrix;
    matrix  = matrix * matrix;

vector operators
    vector  = vector * matrix;
    vector *= matrix;
    vector  = vector * vector;      // vector cross product
    vector *= vector;               // vector cross product
    vector  = Normalize(vector);

point operators
    point  = point * matrix;
    point  *= matrix;
    point  = point + vector;
    point += vector;
    vector = point - point;

mesh operators (poly collection) used as a model, world, or screen
    mesh  = mesh + mesh;
    mesh += mesh;
    mesh  = mesh * matrix;
    mesh *= matrix;
    mesh.PerspectiveDivide();
```

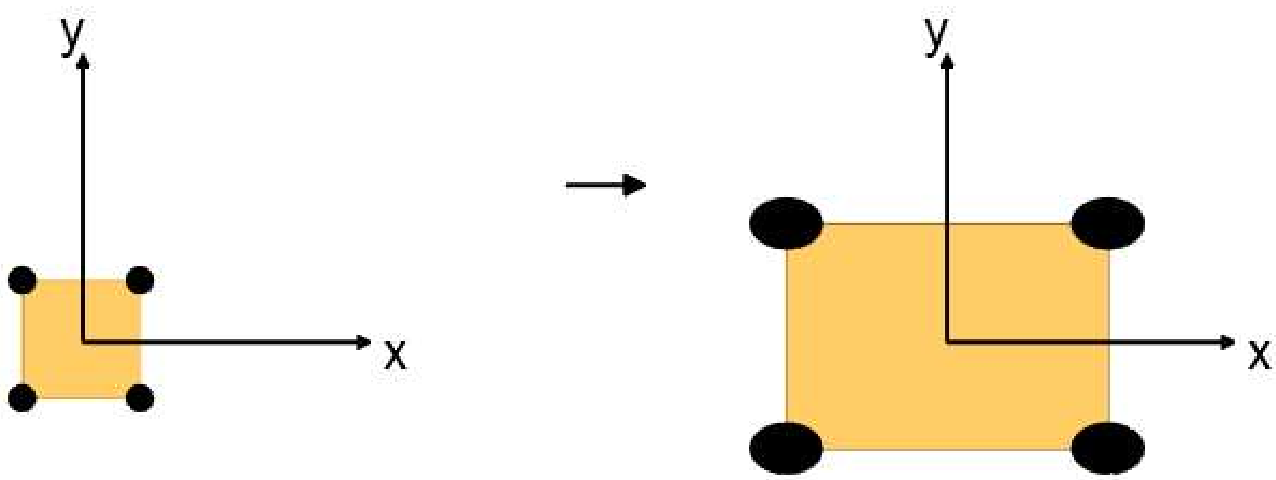
3D Graphics for Dummies

```
Manipulator Matrices:

  Matrix Identity();
  Matrix RotateX(float degrees);
  Matrix RotateY(float degrees);
  Matrix RotateZ(float degrees);
  Matrix Scale(float x, float y, float z);
  Matrix Translate(float x, float y, float z);


Transforms Matrices:

  Matrix PointOfView(const Point& eye, const Point& target, const Vector& up);
  Matrix FieldOfView(float fovAngle, float aspectRatio, float nearPlaneDistance, float
     farPlaneDistance);
  Matrix Viewport(Rect& view, float minZ, float maxZ);
```

Scaling Matrix

$$\left[\begin{array}{ccc|c} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

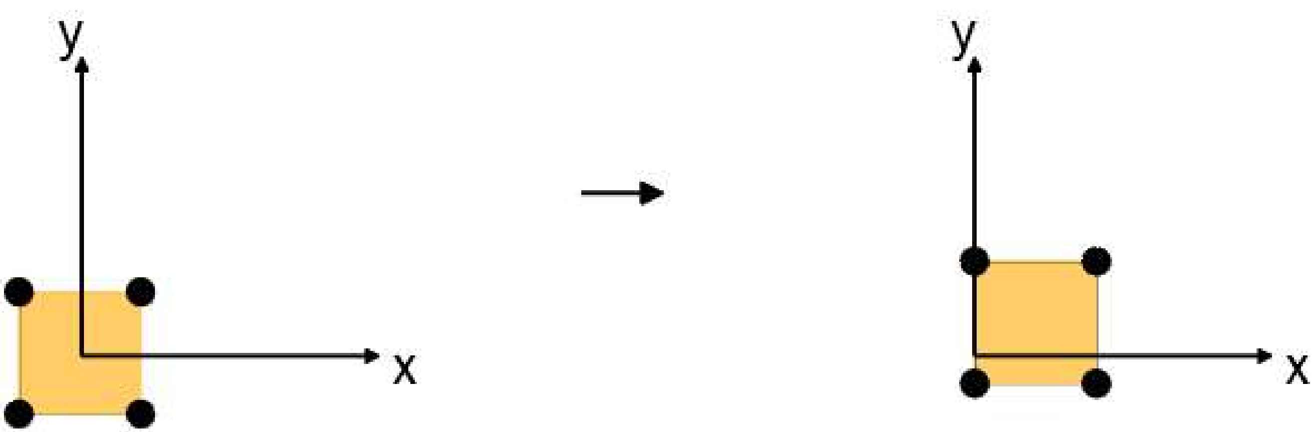
```
float x = 3, y = 2, z = 1;
Matrix mat = Scale(x, y, z);
```



Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

```
float x = 5, y = 2, z = 0;
Matrix mat = Translate(x, y, z);
```



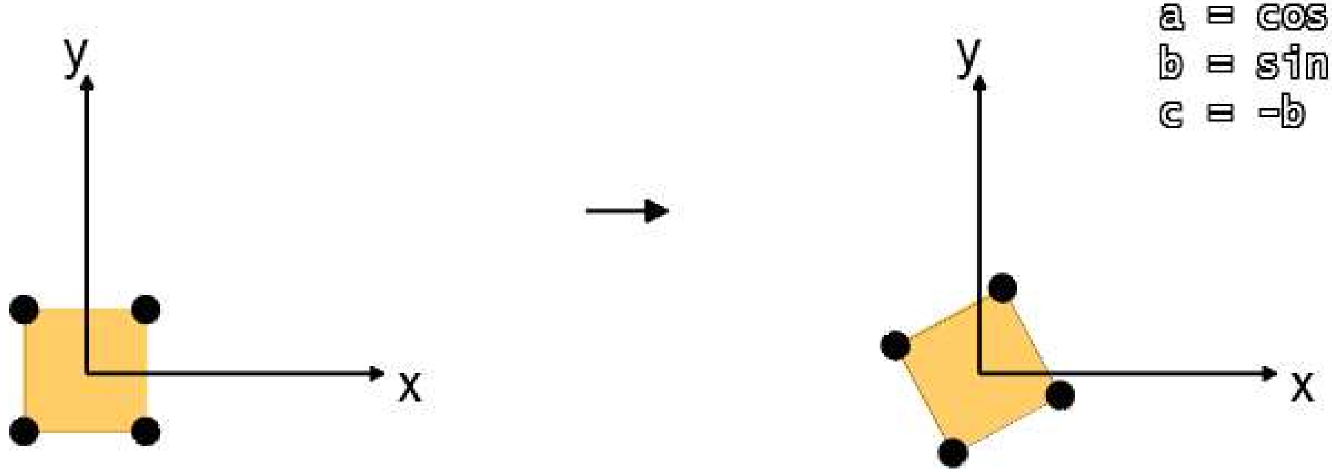
Z Rotation Matrix

```
Matrix mat = RotationZ(angle);
```

$$\begin{bmatrix} a & b & 0 & 0 \\ c & a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z axis

a = cos(Φ)
b = sin(Φ)
c = -b



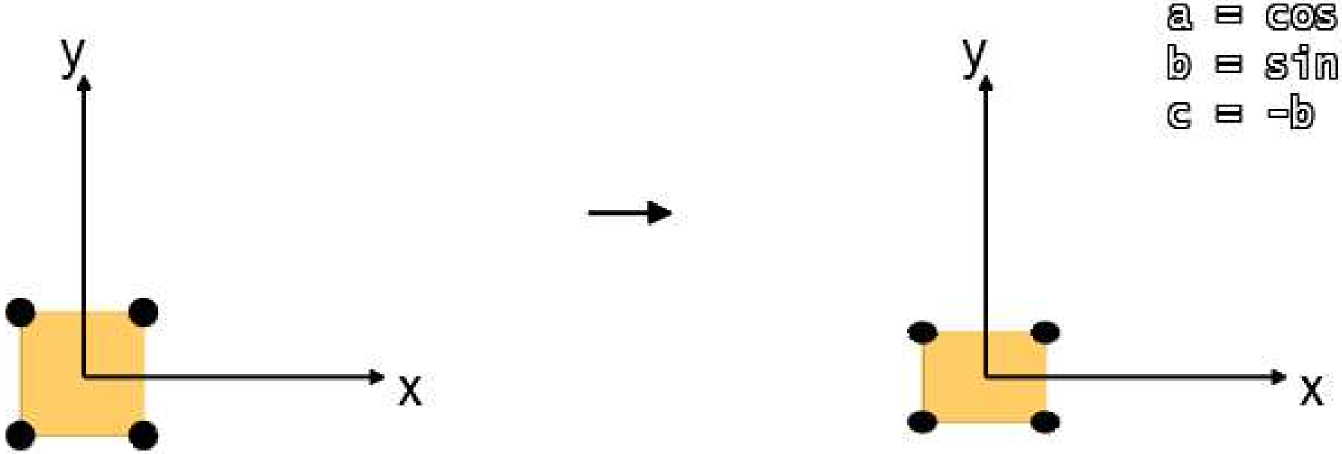
X Rotation Matrix

```
Matrix mat = RotationX(angle);
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & a & b & 0 \\ 0 & c & a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X axis

$a = \cos(\Phi)$
$b = \sin(\Phi)$
$c = -b$



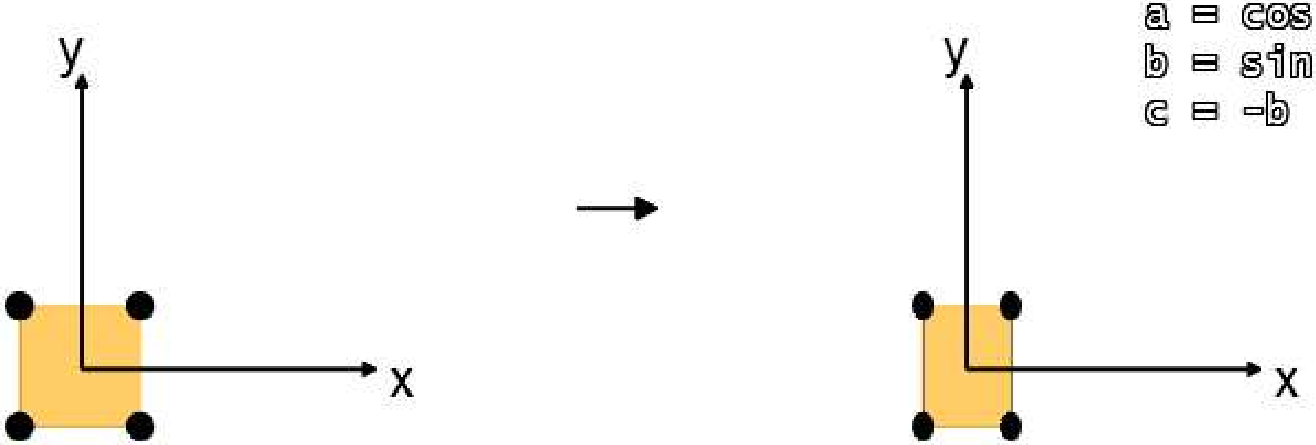
Y Rotation Matrix

```
Matrix mat = RotationY(angle);
```

$$
\left[\begin{array}{ccc|c}
a & 0 & c & 0 \\
0 & 1 & 0 & 0 \\
b & 0 & a & 0 \\
\hline
0 & 0 & 0 & 1
\end{array}\right]
$$

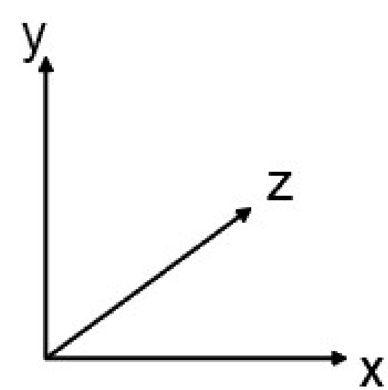
Y axis

$a = \cos(\Phi)$
$b = \sin(\Phi)$
$c = -b$



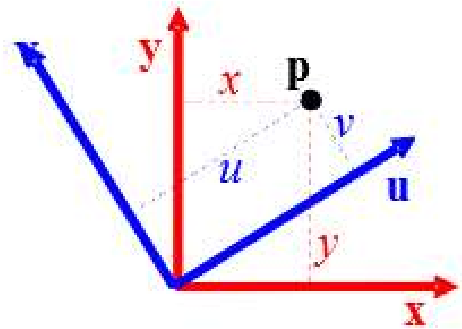
Orthonormal Basis

- Basis: a space is totally defined by a set of vectors – any point is a linear combination of the basis
- Orthogonal: dot product of any two vector is zero
- Normal: magnitude is one
- Orthonormal: orthogonal + normal
- Most common example:

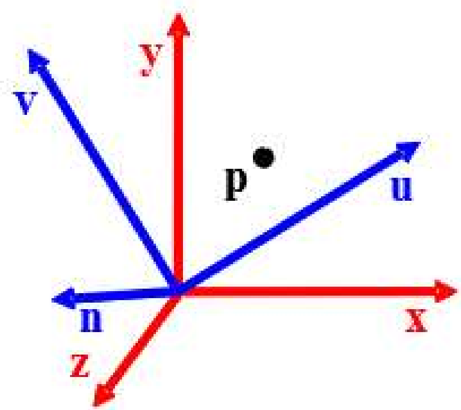


Change of Orthonormal Basis

- Given:
  coordinate frames
  xyz and **uvn**
  point **p** = (p_x, p_y, p_z)



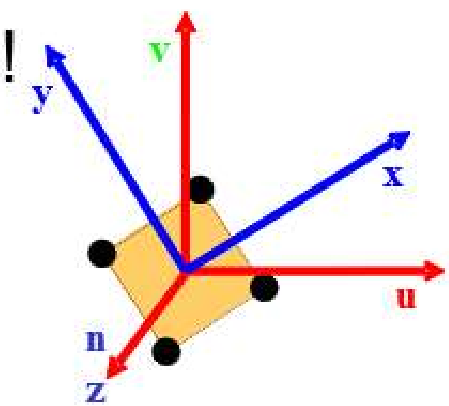
- Find:
  p = (p_u, p_v, p_n)



Change of Orthonormal Basis

- Simply fill in columns with vectors **u**, **v**, **n** (coordinates of new basis vector relative to old)
- Dotting with vectors **u**, **v**, **n**
- Can think of X, Y or Z rotations as changes in orthonormal basis
- Basis deals only w/ upper-left 3x3!

$$\begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Matrix Math Multiply Order

- v * M1 * M2 * M3

$$= [[(v * M1) * M2] * M3]$$

$$= v * (M1 * M2 * M3)$$

$$M = M1 * M2 * M3$$

$$= v * M$$

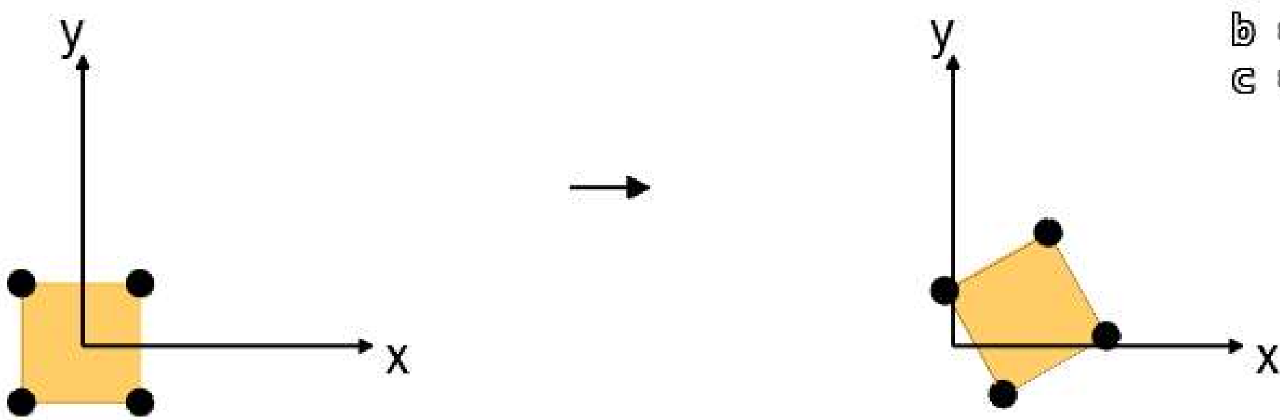
Combining Matrices

```
Matrix matRot = RotationZ(angle);

float x = 5, y = 2, z = 0;
Matrix matTrans = Translation(x, y, z);

Matrix mat = matRot * matTrans;
```

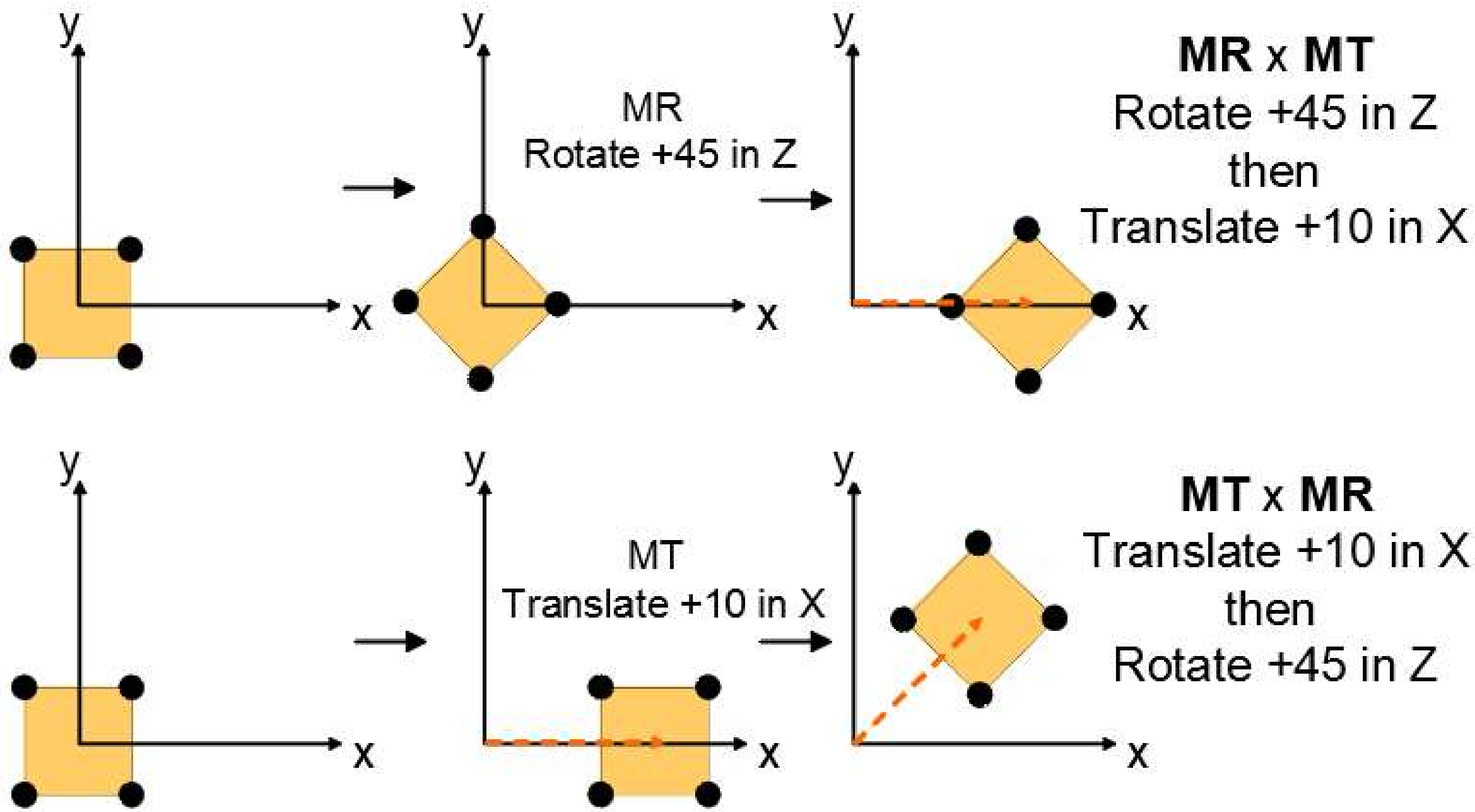
$$\begin{bmatrix} a & b & 0 & 0 \\ c & a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

Z axis

$a = \cos(\phi)$
$b = \sin(\phi)$
$c = -b$



Geometric Interpretation: Matrix Multiplication is Not Commutative



Coordinate Spaces Overview

- Model Space

↓

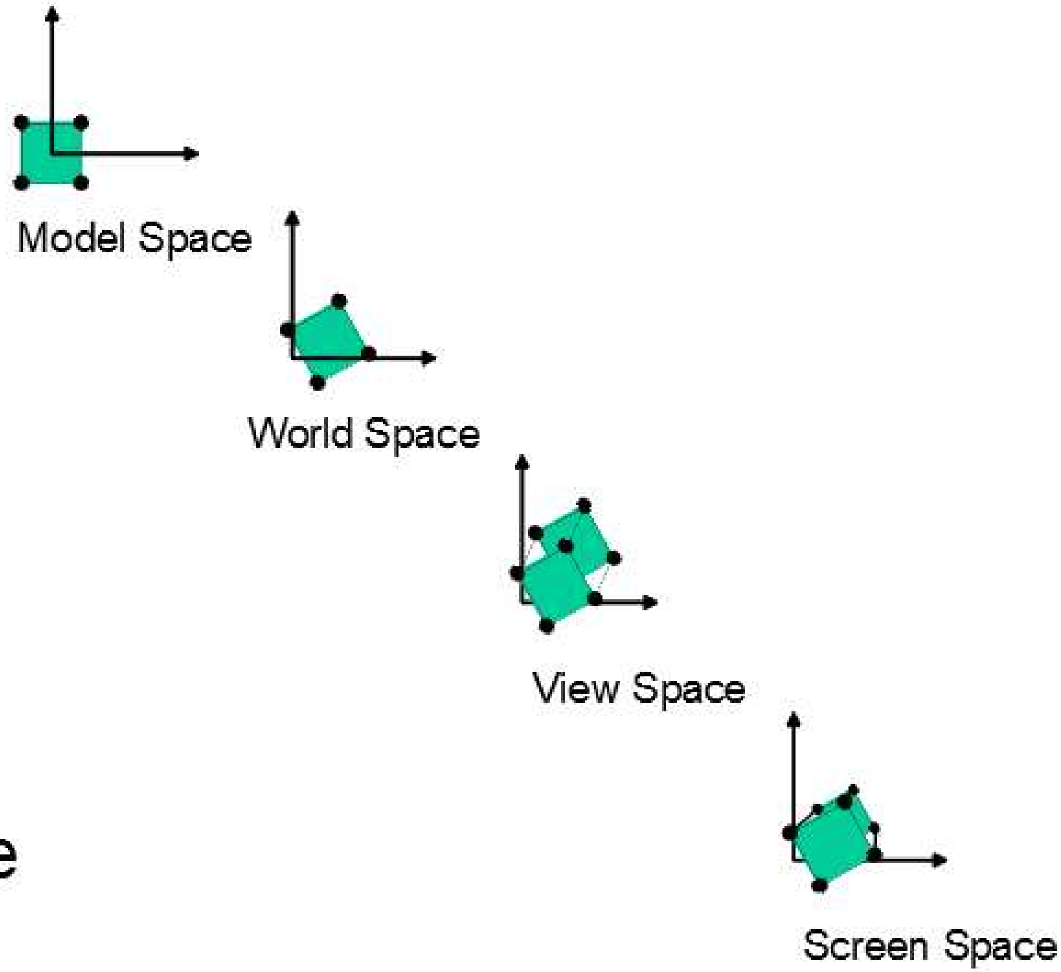
- World Space

↓

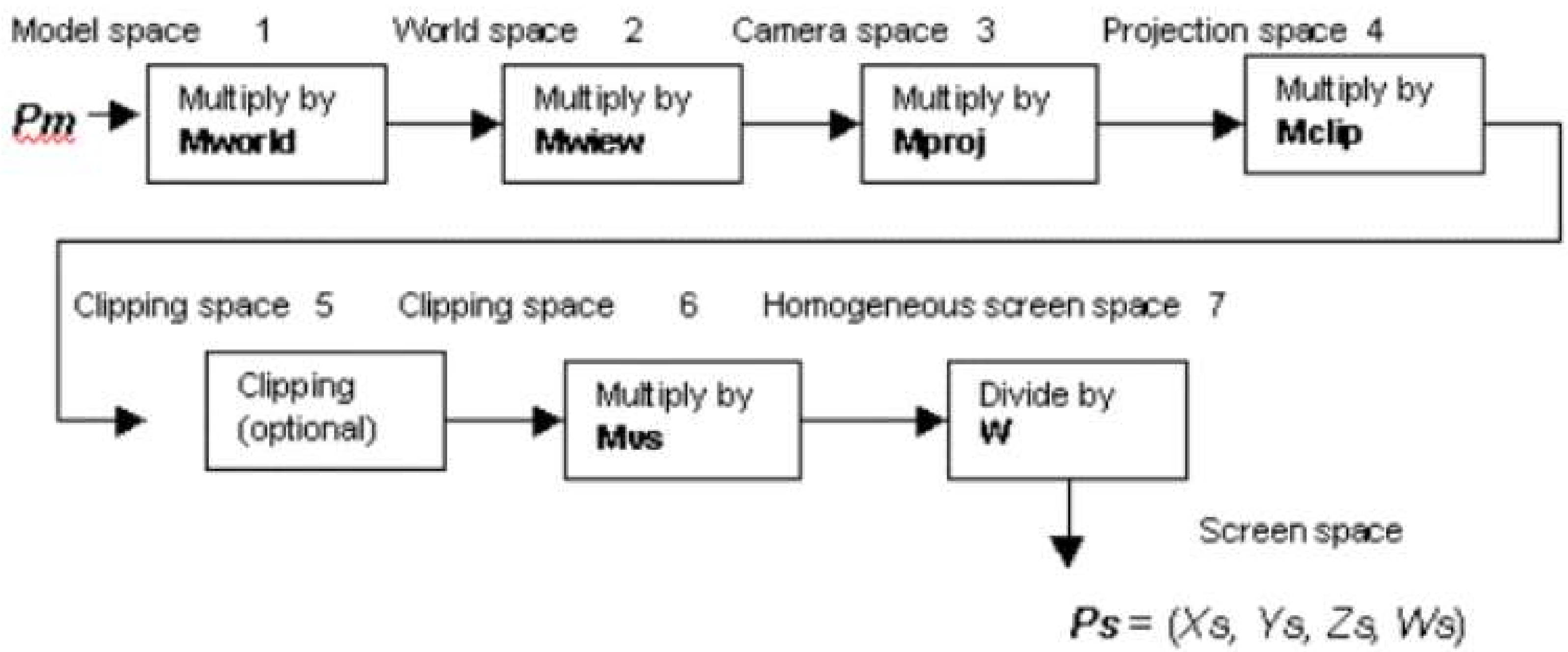
- View Space

↓

- Screen Space



Transformation Pipeline



World Matrix

```
Matrix matRot = RotationZ(30);

Matrix matTrans = Translation(0, 100, 0);

Matrix world = matRot * matTrans;
```

```
| 0.866  0.500  0.000  0.000 |
|-0.500  0.866  0.000  0.000 |
| 0.000  0.000  1.000  0.000 |
| 0.000  100.0  0.000  1.000 |
```

Model Space to World Space

$$\begin{matrix} (-1, 1, 0) \\ (1, 1, 0) \\ (-1, -1, 0) \\ (1, -1, 0) \end{matrix} \times \begin{vmatrix} .87 & .50 & 0.0 & 0.0 \\ -.50 & .87 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 100 & .0 & 1.0 \end{vmatrix} = \begin{matrix} (-1.37, 100.4, 0) \\ (0.37, 101.4, 0) \\ (-0.37, 98.6, 0) \\ (1.37, 99.6, 0) \end{matrix}$$

Model Space to World Space

$$
\begin{matrix}
(-1, 1, 0, 1) \\
(1, 1, 0, 1) \\
(-1,-1, 0, 1) \\
(1,-1, 0, 1)
\end{matrix}
\times
\begin{vmatrix}
.87 & .50 & 0.0 & 0.0 \\
-.50 & .87 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 \\
0.0 & 100 & .0 & 1.0
\end{vmatrix}
=
\begin{matrix}
(-1.37, 100.4, 0, 1) \\
(0.37, 101.4, 0, 1) \\
(-0.37, 98.6, 0, 1) \\
(1.37, 99.6, 0, 1)
\end{matrix}
$$

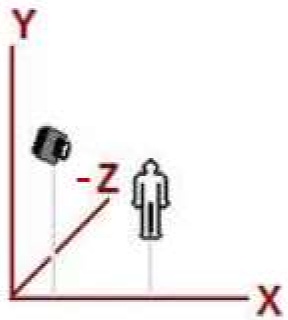
View Matrix

```
Point cameraPosition(0, 92, 5);
Point cameraTarget(0, 100, 0);
Vector cameraUpVector(0, 1, 0);
Matrix view = PointOfView( cameraPosition,
                           cameraTarget,
                           cameraUpVector);
```

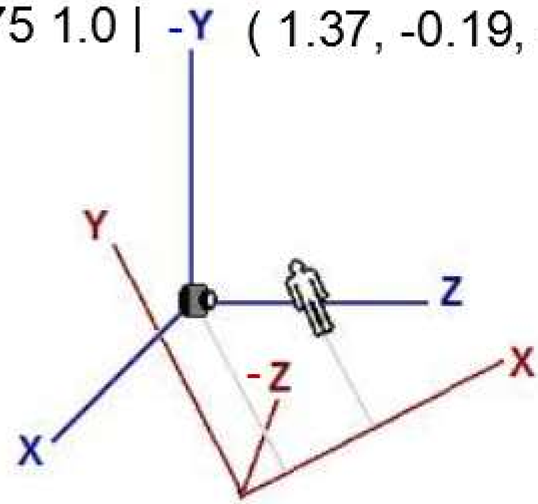
```
| 1.000  0.000  0.000  0.000 |
| 0.000  0.530 -0.848  0.000 |
| 0.000  0.848  0.530  0.000 |
| 0.000 -53.00  75.37  1.000 |
```

World Space to View Space

$$
\begin{pmatrix} -1.37, & 100.4, & 0, & 1 \\ 0.37, & 101.4, & 0, & 1 \\ -0.37, & 98.6, & 0, & 1 \\ 1.37, & 99.6, & 0, & 1 \end{pmatrix}
\times
\begin{vmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & .53 & .85 & 0.0 \\ 0.0 & -.85 & .53 & 0.0 \\ 0.0 & -53 & -75 & 1.0 \end{vmatrix}
=
\begin{pmatrix} -1.37, & 0.19, & -9.74, & 1 \\ 0.37, & 0.72, & -10.59, & 1 \\ -0.37, & -0.72, & -8.28, & 1 \\ 1.37, & -0.19, & -9.12, & 1 \end{pmatrix}
$$

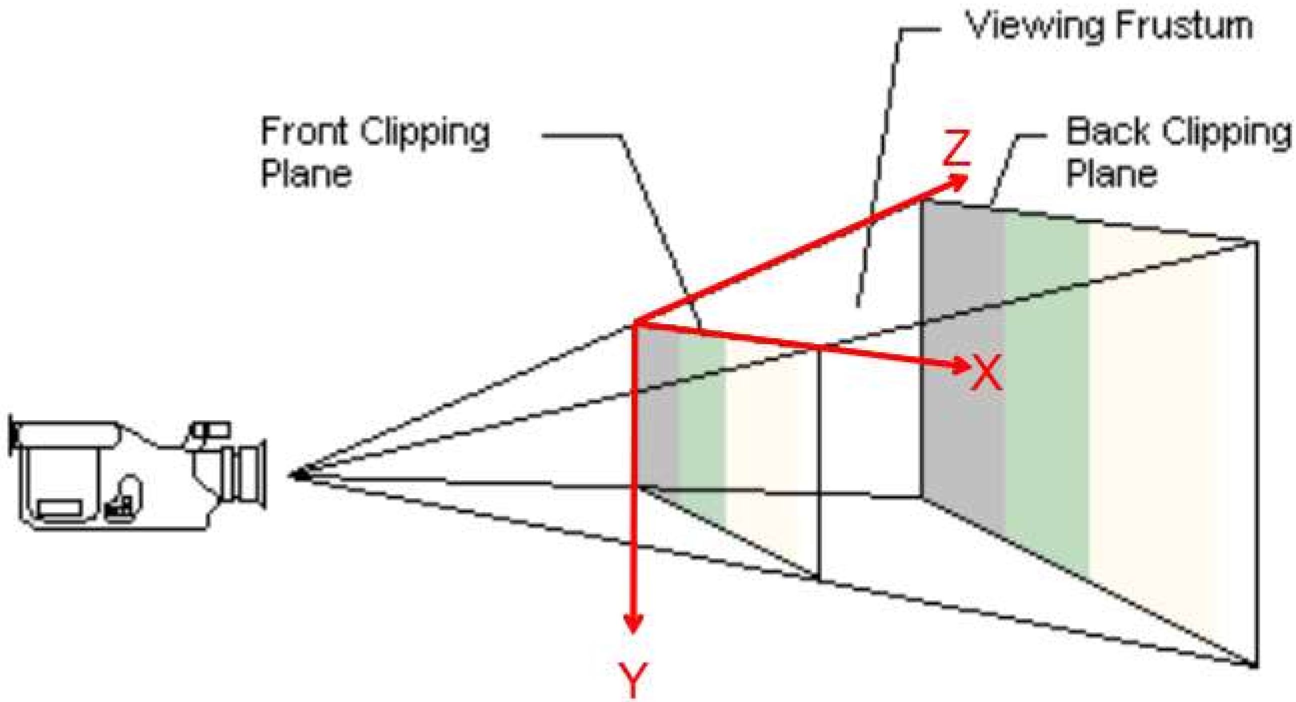


world space



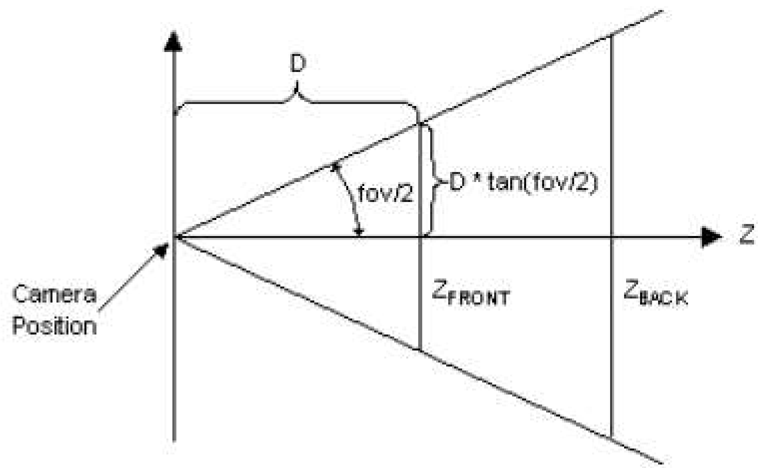
view (camera) space

Add Perspective: View Frustum



Projection Matrix

```
float fieldOfView = 45;
float aspectRatio = 1.333;
float nearPlaneDistance = 1;
float farPlaneDistance = 100;
Matrix projection = FieldOfView(fieldOfView,
                                aspectRatio,
                                nearPlaneDistance,
                                farPlaneDistance);
```




```
| 1.811  0.000  0.000  0.000 |
| 0.000  2.414  0.000  0.000 |
| 0.000  0.000 -1.010 -1.000 |
| 0.000  0.000 -1.010  0.000 |
```

View Space to Projection Space

$$
\begin{pmatrix} -1.37 & 0.19 & -9.74 & 1 \\ 0.37 & 0.72 & -10.59 & 1 \\ -0.37 & -0.72 & -8.28 & 1 \\ 1.37 & -0.19 & -9.12 & 1 \end{pmatrix} \times \begin{vmatrix} 1.8 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.4 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.0 & -1.0 \\ 0.0 & 0.0 & -1.0 & 0.0 \end{vmatrix} = \begin{pmatrix} -2.47 & 0.47 & 8.83 & 9.74 \\ 0.66 & 1.75 & 9.69 & 10.59 \\ -0.66 & -1.75 & 7.35 & 8.28 \\ 2.47 & -0.47 & 8.21 & 9.12 \end{pmatrix}
$$

Clipping

- Clipping volume for all points (px, py, pz, pw)

  -pw < px <= pw

  -pw < py <= pw

  0 < pz <= pw

Viewport Scale Matrix

- Derived from `Viewport` settings

$$\begin{bmatrix} dwWidth/2 & 0 & 0 & 0 \\ 0 & -dwHeight/2 & 0 & 0 \\ 0 & 0 & dvMaxZ - dvMinZ & 0 \\ dwX + dwWidth/2 & dwHeight/2 + dwY & dvMinz & 1 \end{bmatrix}$$

```
| 320.0  0.000  0.000  0.000 |
| 0.000 -240.0  0.000  0.000 |
| 0.000  0.000  1.000  0.000 |
| 320.0  240.0  0.000  1.000 |
```

Projection Space to Homogeneous Screen Space

$$
\begin{array}{l}
(-2.47,\ 0.47,\ 8.83,\ 9.74) \\
(\ 0.66,\ 1.75,\ 9.69,10.59) \\
(-0.66,\ -1.75,\ 7.35,\ 8.28) \\
(\ 2.47,\ -0.47,\ 8.21,\ 9.12)
\end{array}
\times
\begin{vmatrix}
320 & 0.0 & 0.0 & 0.0 \\
0.0 & -240 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 \\
320 & 240 & 0.0 & 1.0
\end{vmatrix}
=
\begin{array}{l}
(2327,\ 2226,\ 8.83,\ 9.74) \\
(3602,\ 2123,\ 9.69,10.59) \\
(2436,\ 2406,\ 7.35,\ 8.28) \\
(3711,\ 2302,\ 8.21,\ 9.12)
\end{array}
$$

Divide by *w*

- Ready to show on screen

$$Xs = \frac{X}{W}, Ys = \frac{Y}{W}, Zs = \frac{Z}{W}, Ws = \frac{1}{W}$$

- Divide by *w* also called perspective divide

Homogeneous Screen Space to Screen Space

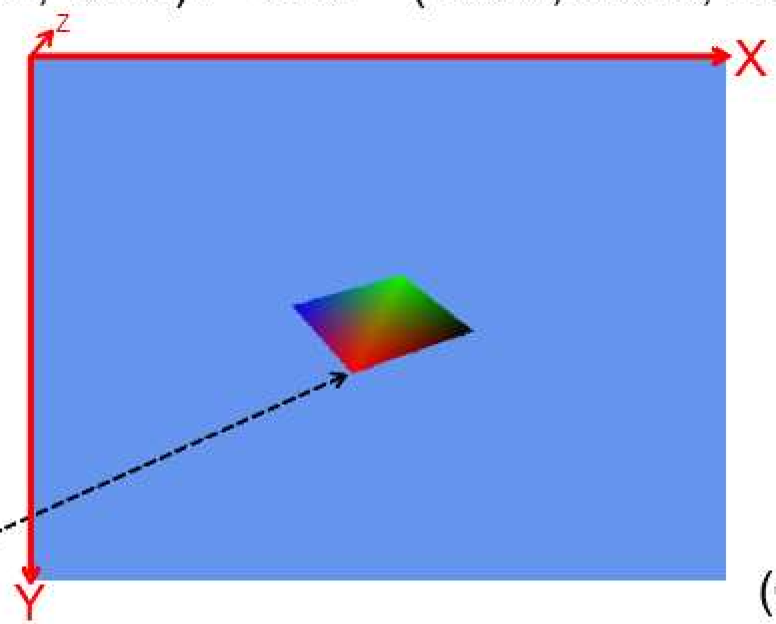
Homogeneous Screen / W = Screen Space

(2327, 2226, 8.83, 9.74) / 9.74 = (238.8, 228.5, 0.906, 1) (blue)

(3602, 2123, 9.69,10.59) / 10.59 = (340.0, 200.4, 0.915, 1) (green)

(2436, 2406, 7.35, 8.28) / 8.28 = (294.4, 290.7, 0.888, 1) (red)

(3711, 2302, 8.21, 9.12) / 9.12 = (406.8, 252.3, 0.899, 1) (black)



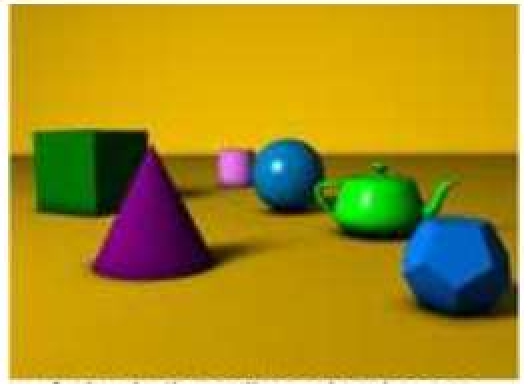
Red is closest to camera

Are screen space Z values used?

(640, 480)

Image Buffers and Z-Buffer

- Two **Image Buffers**:
  - One holds previously rendered frame which is being displayed on-screen
  - One holds the **frame** currently being rendered
- Additional **Z-Buffer** (or **Depth Buffer**) stores depth information



A simple three dimensional scene

Z-buffer representation

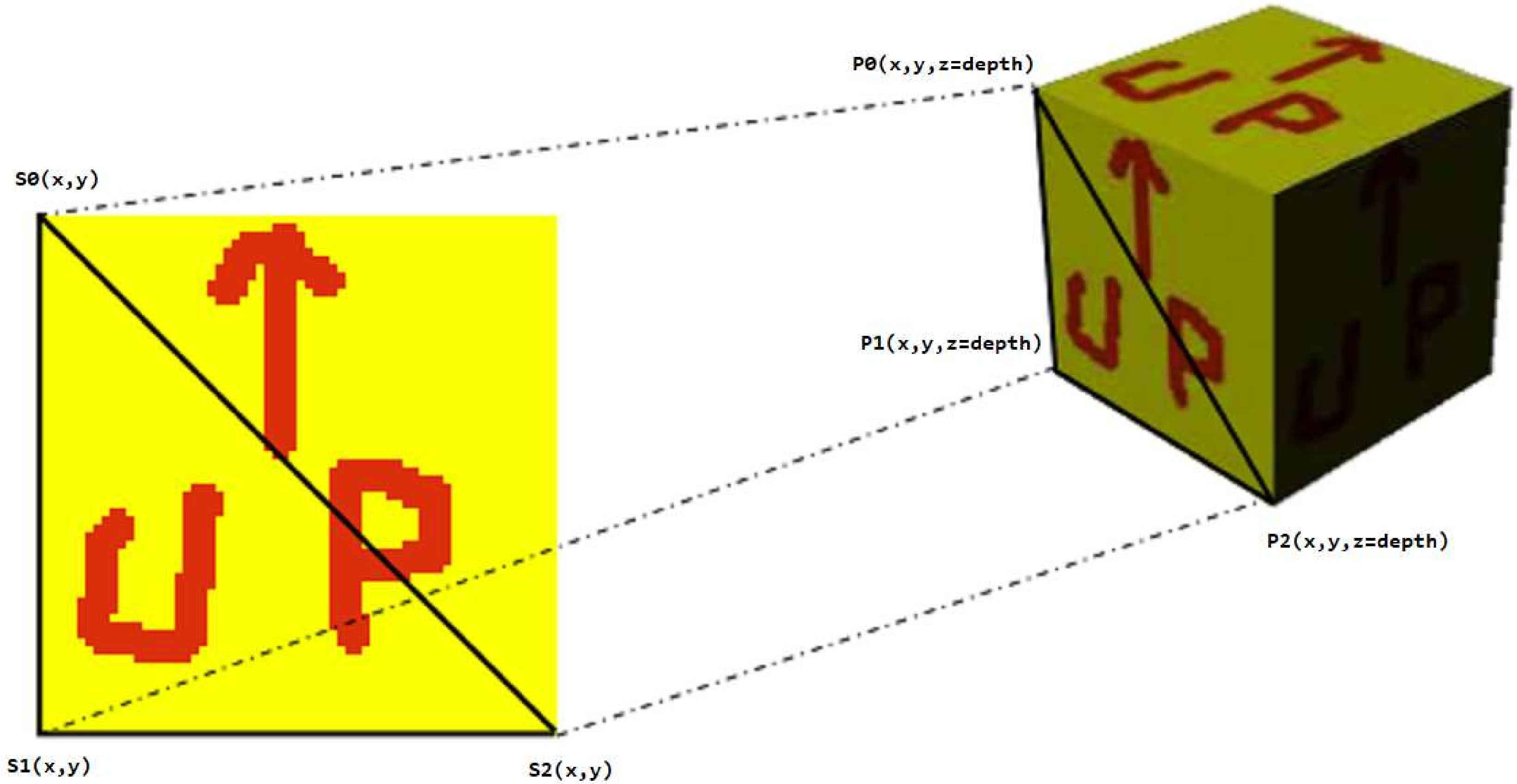
Z-Buffer Operation

- Before drawing each pixel on screen:
- **Z-Test**: compare Z value ([0 1] in view frustum) against value in Z-buffer
- If (Z value < value in Z-buffer), update image buffer & Z-buffer

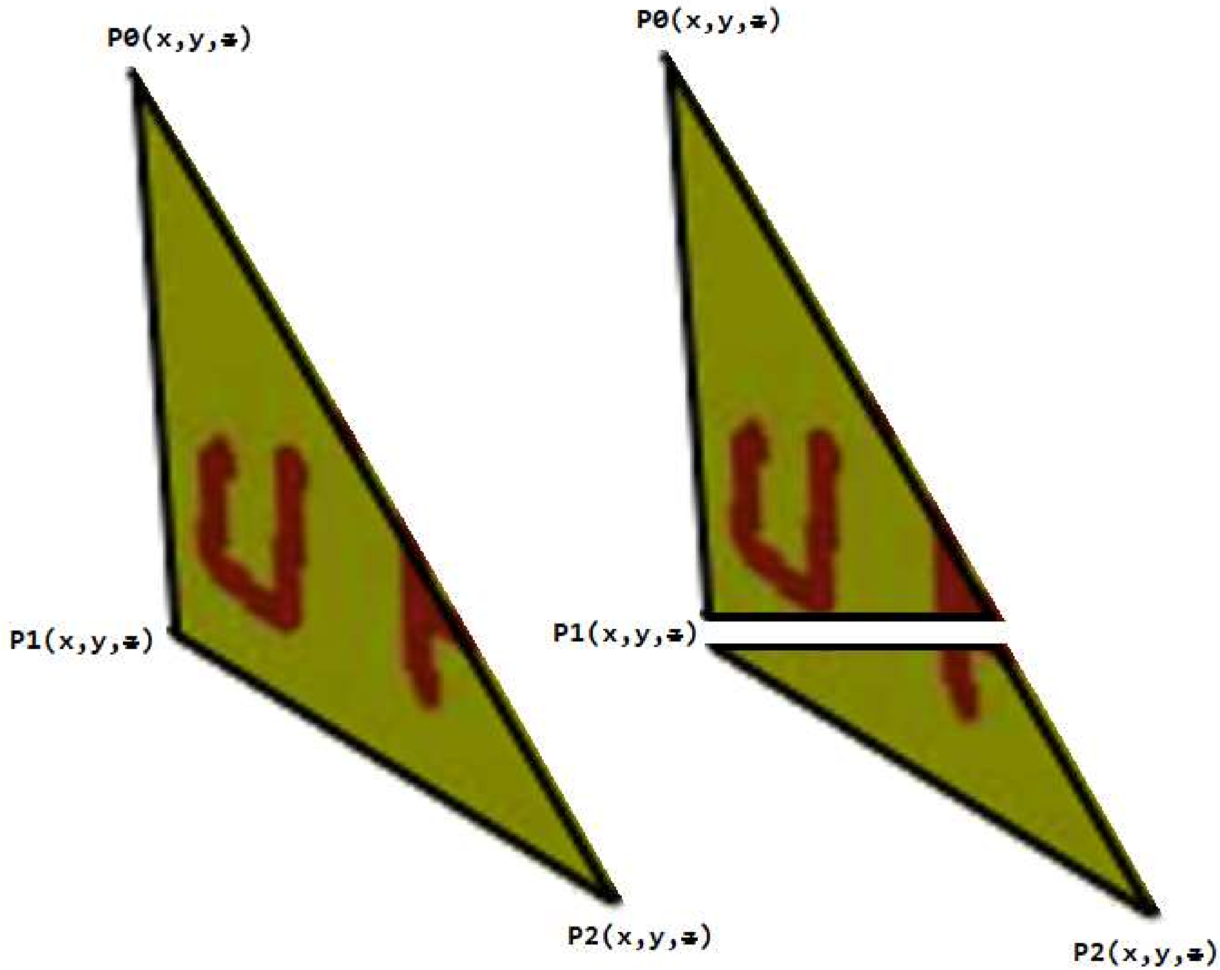
A simple three dimensional scene

Z-buffer representation

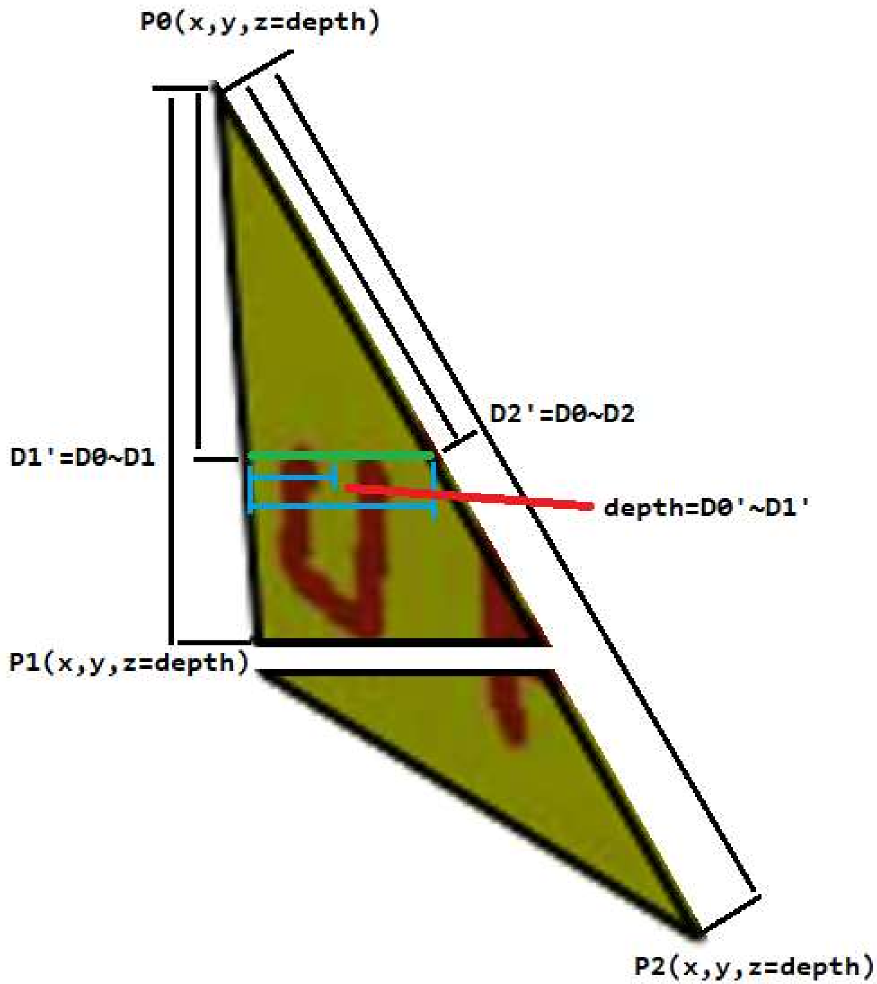
Depth / Texturing / Applying Surfaces



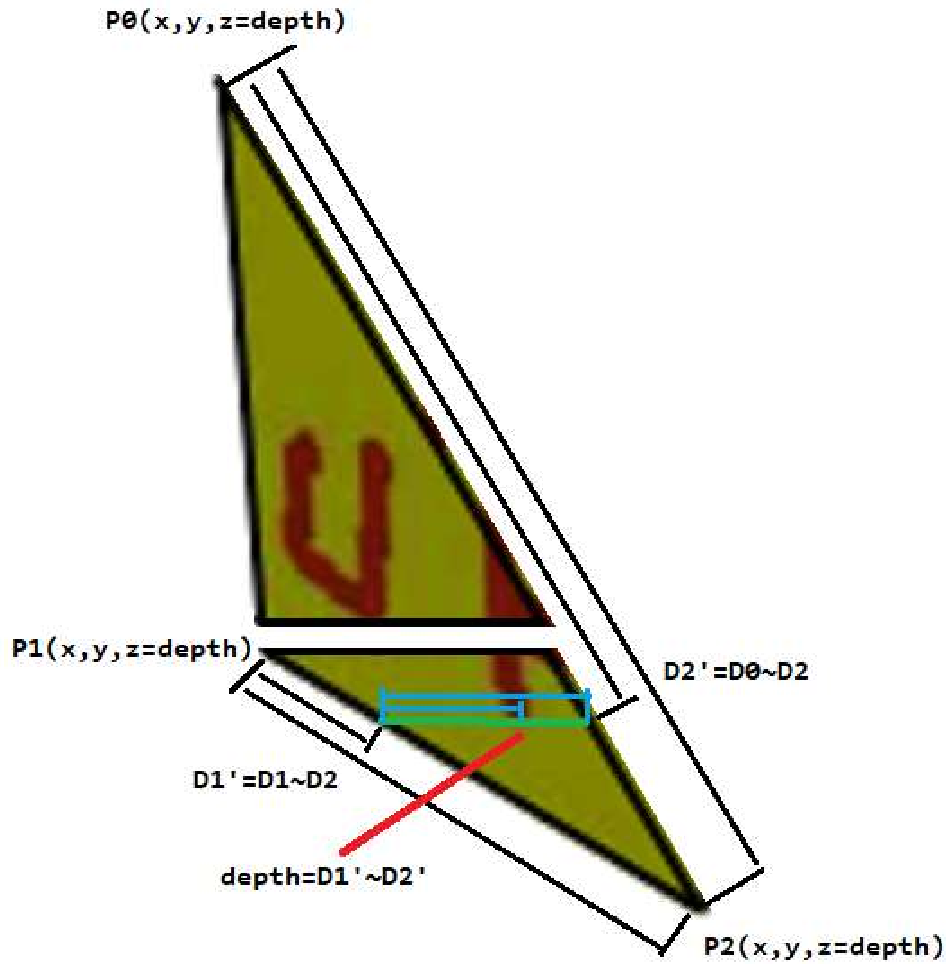
Rasterization: Depth



Rasterization: Depth

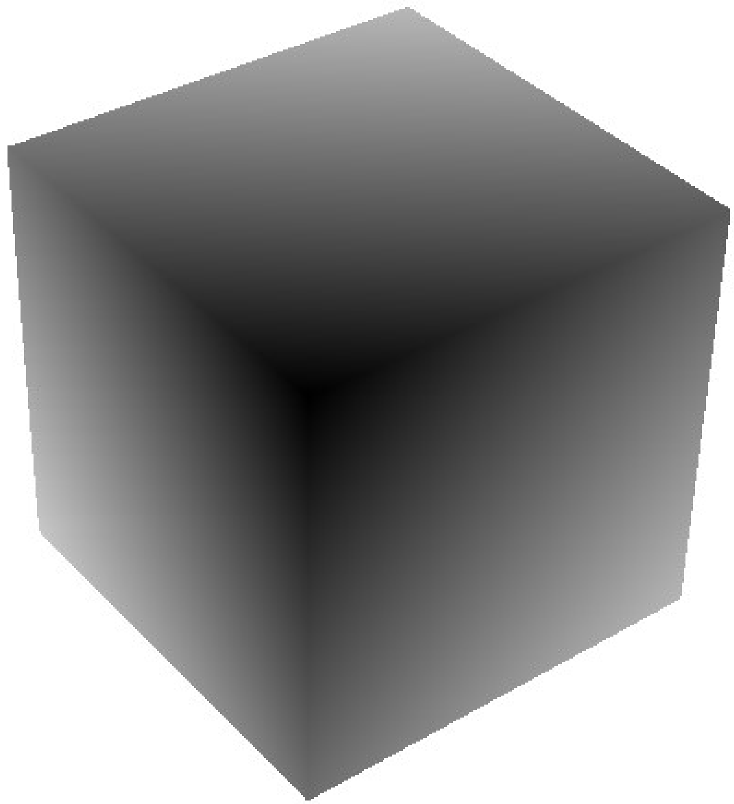


Rasterization: Depth

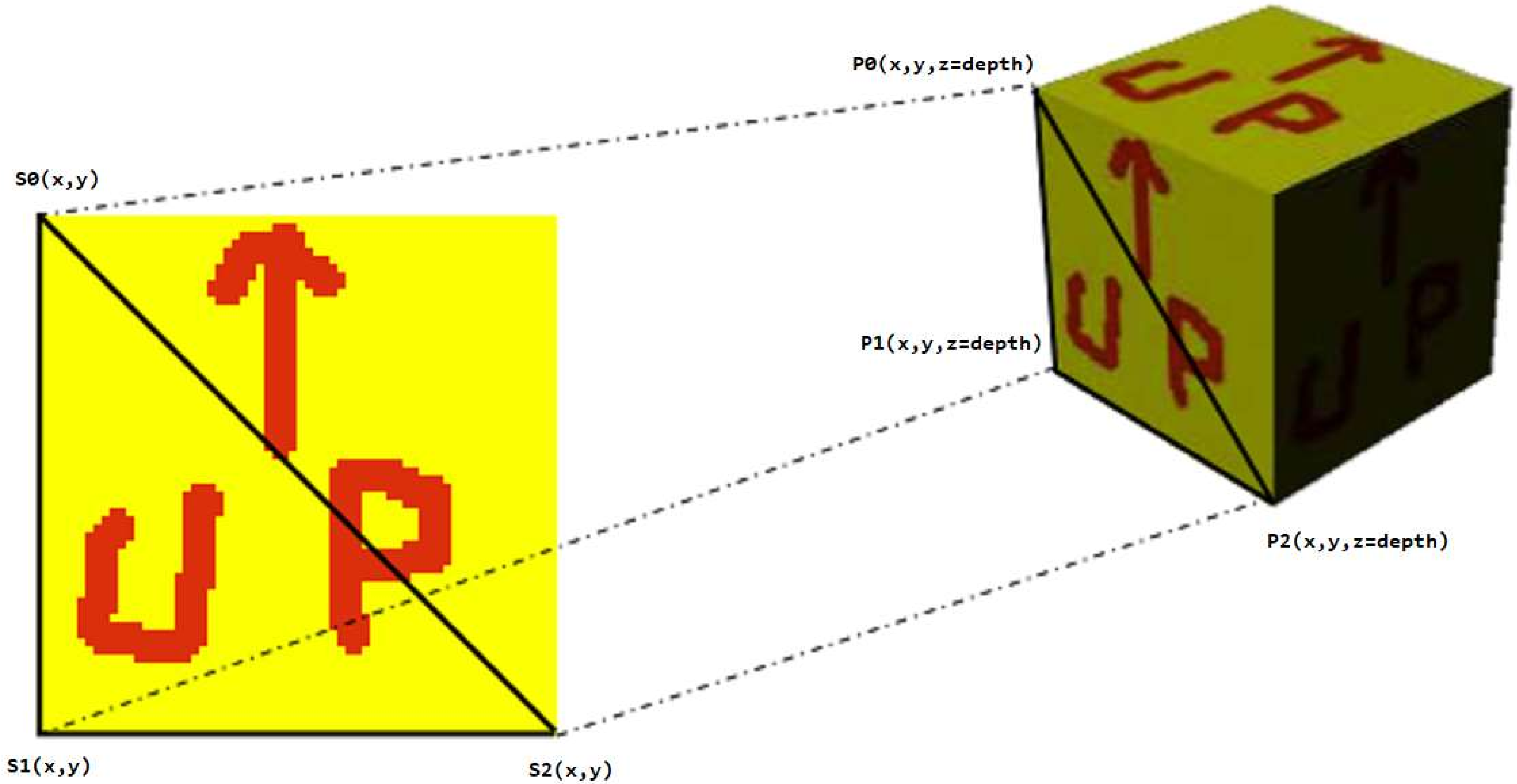


3D Graphics for Dummies

Rasterization: Depth

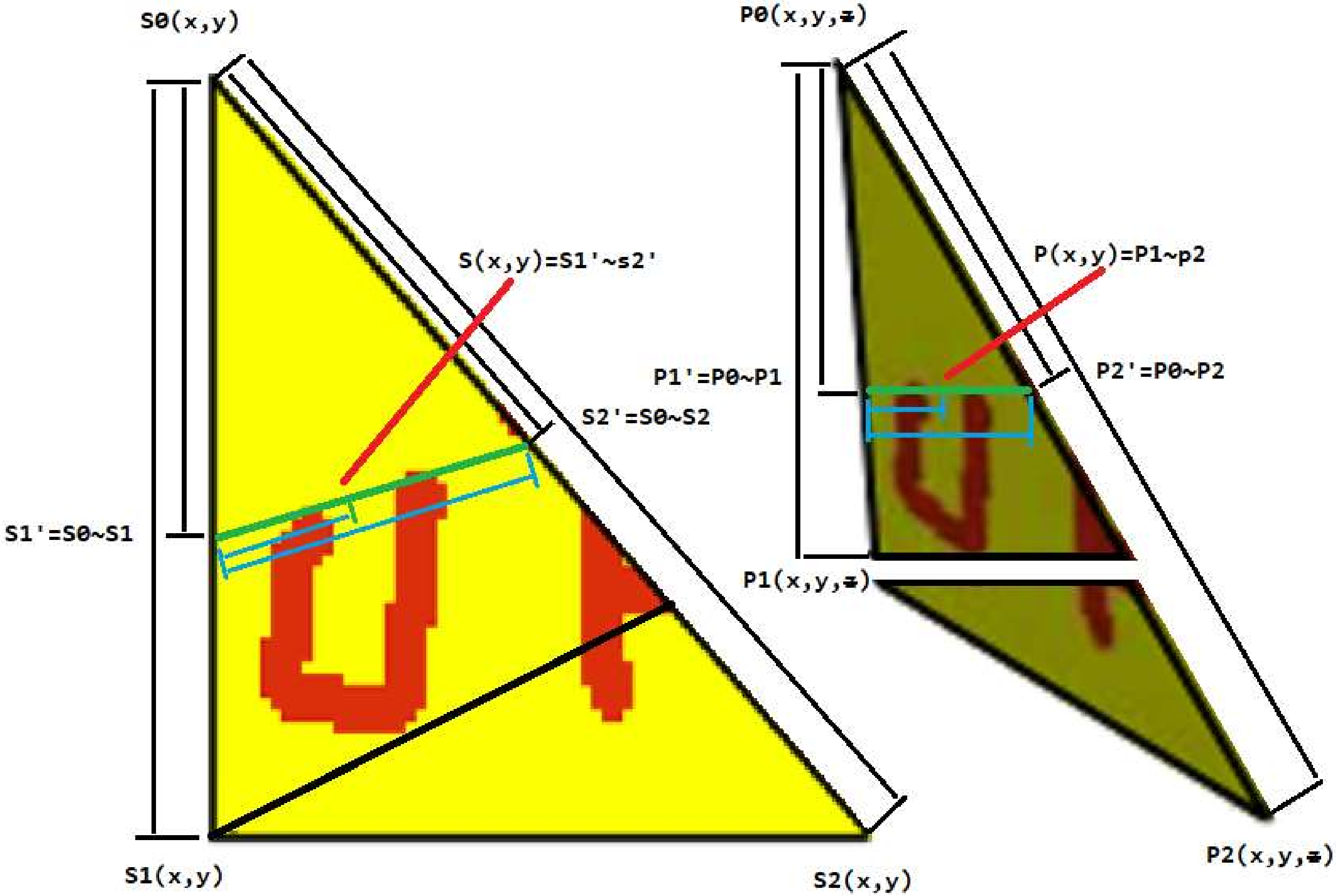


Depth / Texturing / Applying Surfaces

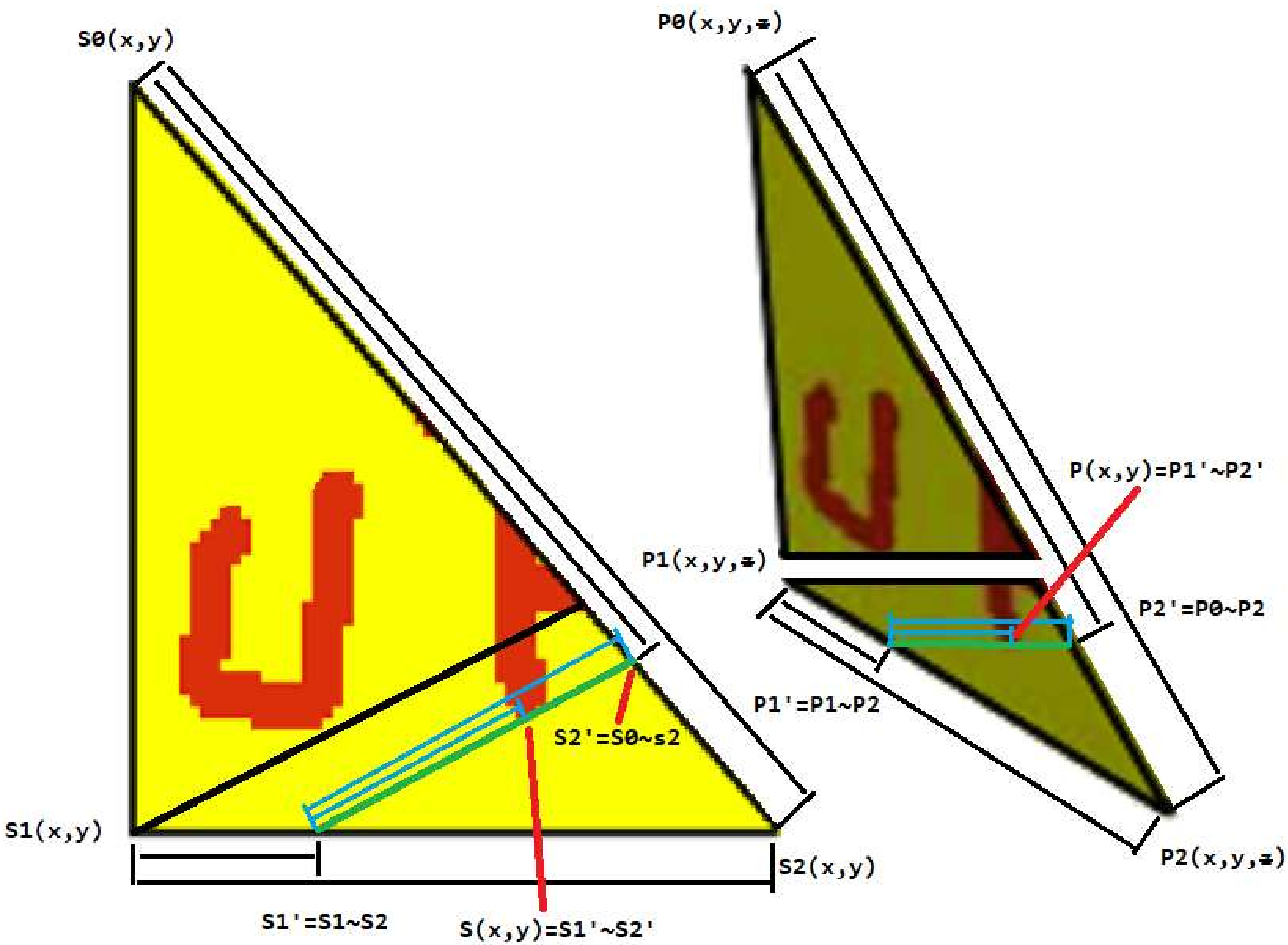


3D Graphics for Dummies

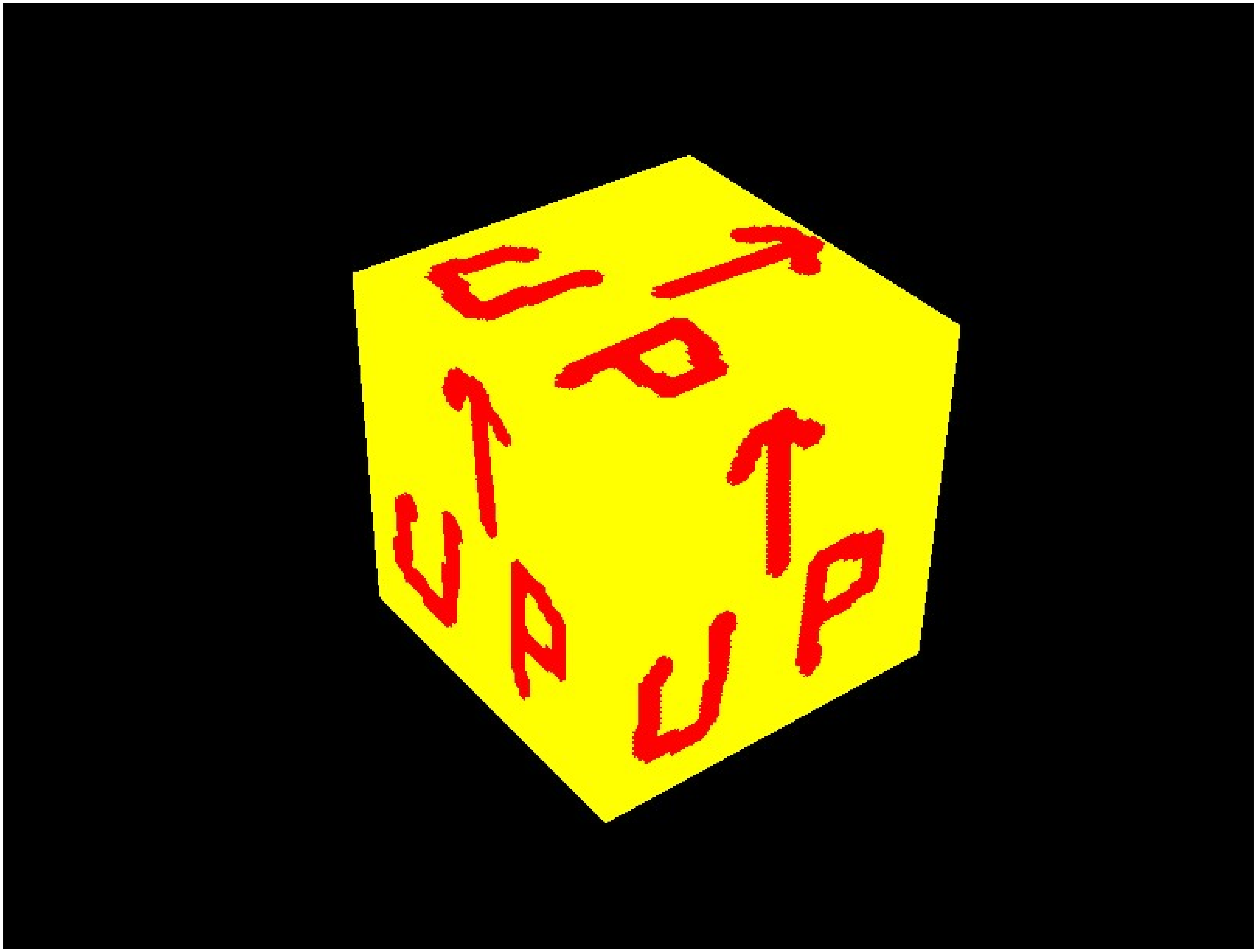
Rasterization: Texturing / Applying Surfaces



Rasterization: Texturing / Applying Surfaces

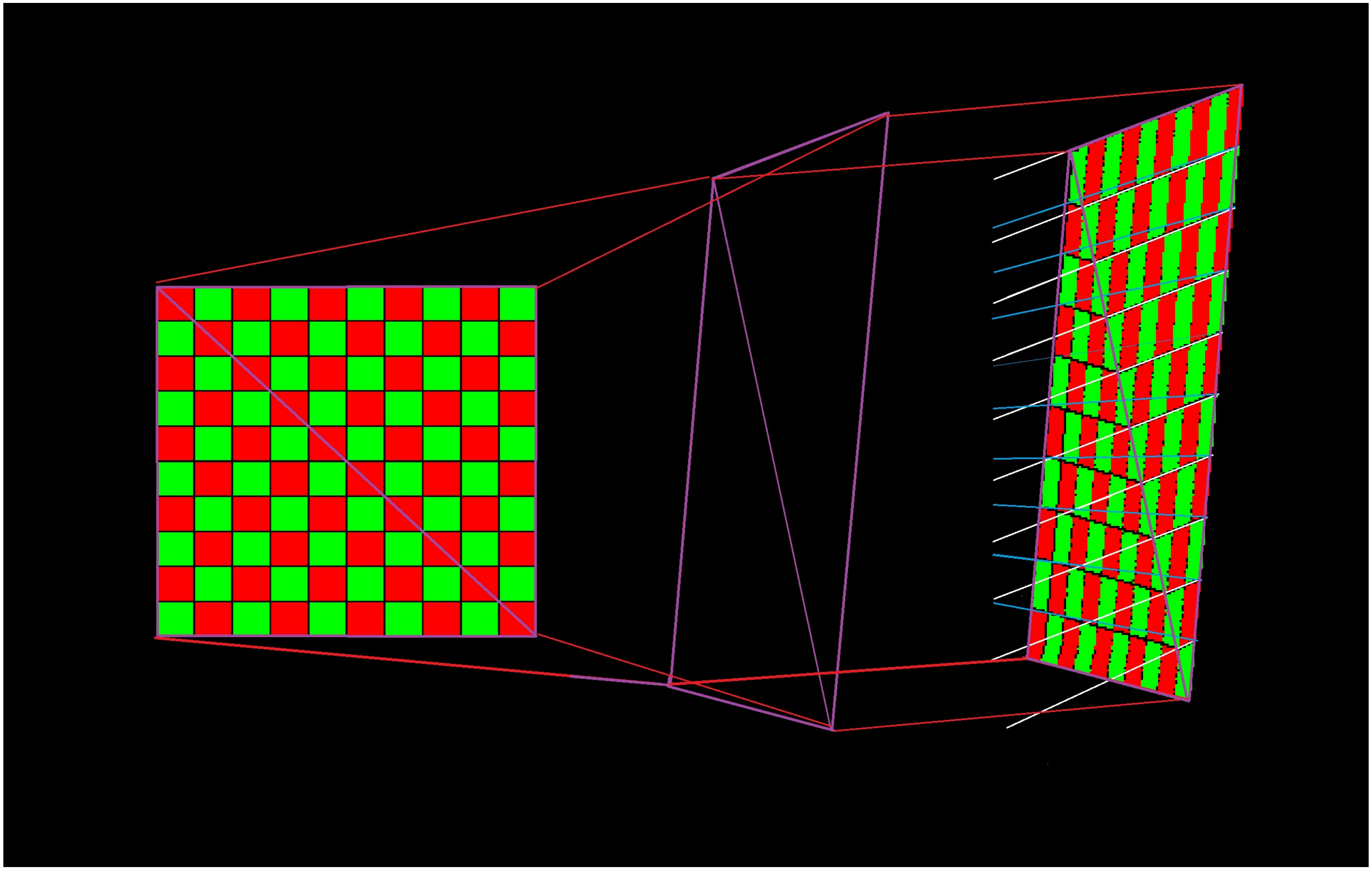


Rasterization: Texturing / Applying Surfaces

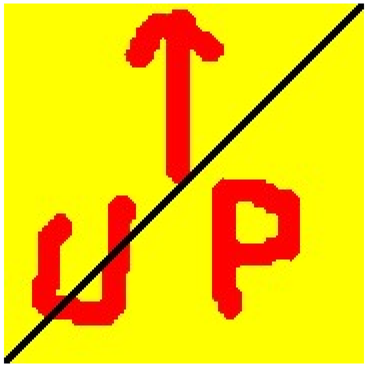


3D Graphics for Dummies

Rasterization: Affine Perspective Error with Triangle/Poly Interpolation
a.k.a.: Derivative Discontinuity in Interpolation

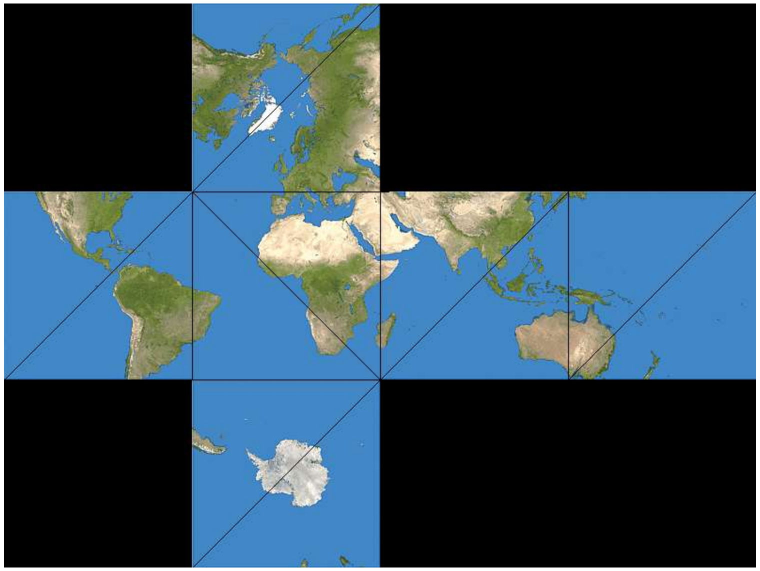


```
modelUp =
{
    { {{ -1, -1,  1 }, { -1,  1,  1 }, {  1,  1,  1 }}, IDB_UP, {{   0,   0 }, {   0, 179 }, { 179, 179 }} },
    { {{  1,  1,  1 }, {  1, -1,  1 }, { -1, -1,  1 }}, IDB_UP, {{ 179, 179 }, { 179,   0 }, {   0,   0 }} },

    { {{ -1, -1, -1 }, { -1,  1, -1 }, {  1,  1, -1 }}, IDB_UP, {{   0,   0 }, {   0, 179 }, { 179, 179 }} },
    { {{  1,  1, -1 }, {  1, -1, -1 }, { -1, -1, -1 }}, IDB_UP, {{ 179, 179 }, { 179,   0 }, {   0,   0 }} },

    { {{ -1,  1, -1 }, { -1,  1,  1 }, {  1,  1,  1 }}, IDB_UP, {{   0,   0 }, {   0, 179 }, { 179, 179 }} },
    { {{  1,  1,  1 }, {  1,  1, -1 }, { -1,  1, -1 }}, IDB_UP, {{ 179, 179 }, { 179,   0 }, {   0,   0 }} },

    { {{ -1, -1, -1 }, { -1, -1,  1 }, {  1, -1,  1 }}, IDB_UP, {{   0,   0 }, {   0, 179 }, { 179, 179 }} },
    { {{  1, -1,  1 }, {  1, -1, -1 }, { -1, -1, -1 }}, IDB_UP, {{ 179, 179 }, { 179,   0 }, {   0,   0 }} },

    { {{  1, -1, -1 }, {  1, -1,  1 }, {  1,  1,  1 }}, IDB_UP, {{   0,   0 }, {   0, 179 }, { 179, 179 }} },
    { {{  1,  1,  1 }, {  1,  1, -1 }, {  1, -1, -1 }}, IDB_UP, {{ 179, 179 }, { 179,   0 }, {   0,   0 }} },

    { {{ -1, -1, -1 }, { -1, -1,  1 }, { -1,  1,  1 }}, IDB_UP, {{   0,   0 }, {   0, 179 }, { 179, 179 }} },
    { {{ -1,  1,  1 }, { -1,  1, -1 }, { -1, -1, -1 }}, IDB_UP, {{ 179, 179 }, { 179,   0 }, {   0,   0 }} },
};
```



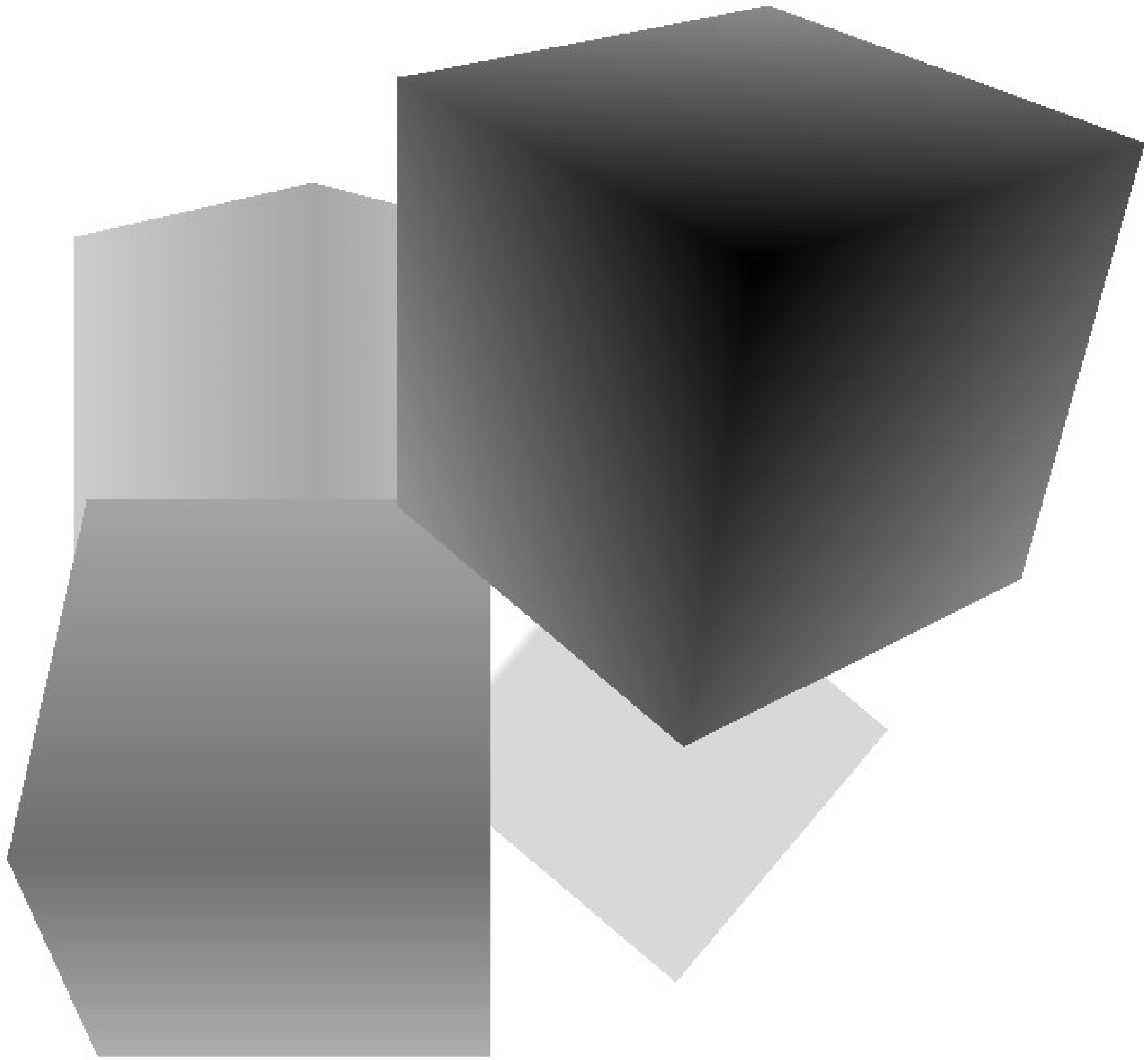
3D Graphics for Dummies

```
modelEarth =
{
    { {{  1,  1,  1 }, { -1,  1,  1 }, { -1,  1, -1 }}, IDB_EARTH, {{ 200, 400 }, { 200, 600 }, { 400, 600 }} },
    { {{ -1,  1, -1 }, {  1,  1, -1 }, {  1,  1,  1 }}, IDB_EARTH, {{ 400, 600 }, { 400, 400 }, { 200, 400 }} },

    { {{ -1, -1,  1 }, { -1,  1,  1 }, {  1,  1,  1 }}, IDB_EARTH, {{   0, 200 }, {   0, 400 }, { 200, 400 }} },
    { {{  1,  1,  1 }, {  1, -1,  1 }, { -1, -1,  1 }}, IDB_EARTH, {{ 200, 400 }, { 200, 200 }, {   0, 200 }} },

    { {{  1, -1, -1 }, {  1, -1,  1 }, {  1,  1,  1 }}, IDB_EARTH, {{ 400, 200 }, { 200, 200 }, { 200, 400 }} },
    { {{  1,  1,  1 }, {  1,  1, -1 }, {  1, -1, -1 }}, IDB_EARTH, {{ 200, 400 }, { 400, 400 }, { 400, 200 }} },

    { {{  1, -1, -1 }, { -1,  1, -1 }, {  1,  1, -1 }}, IDB_EARTH, {{ 400, 200 }, { 600, 400 }, { 400, 400 }} },
    { {{ -1,  1, -1 }, { -1, -1, -1 }, {  1, -1, -1 }}, IDB_EARTH, {{ 600, 400 }, { 600, 200 }, { 400, 200 }} },

    { {{ -1, -1,  1 }, {  1, -1,  1 }, {  1, -1, -1 }}, IDB_EARTH, {{ 200,   0 }, { 200, 200 }, { 400, 200 }} },
    { {{  1, -1, -1 }, { -1, -1, -1 }, { -1, -1,  1 }}, IDB_EARTH, {{ 400, 200 }, { 400,   0 }, { 200,   0 }} },

    { {{ -1, -1, -1 }, { -1, -1,  1 }, { -1,  1,  1 }}, IDB_EARTH, {{ 600, 200 }, { 800, 200 }, { 800, 400 }} },
    { {{ -1,  1,  1 }, { -1,  1, -1 }, { -1, -1, -1 }}, IDB_EARTH, {{ 800, 400 }, { 600, 400 }, { 600, 200 }} },
};
```

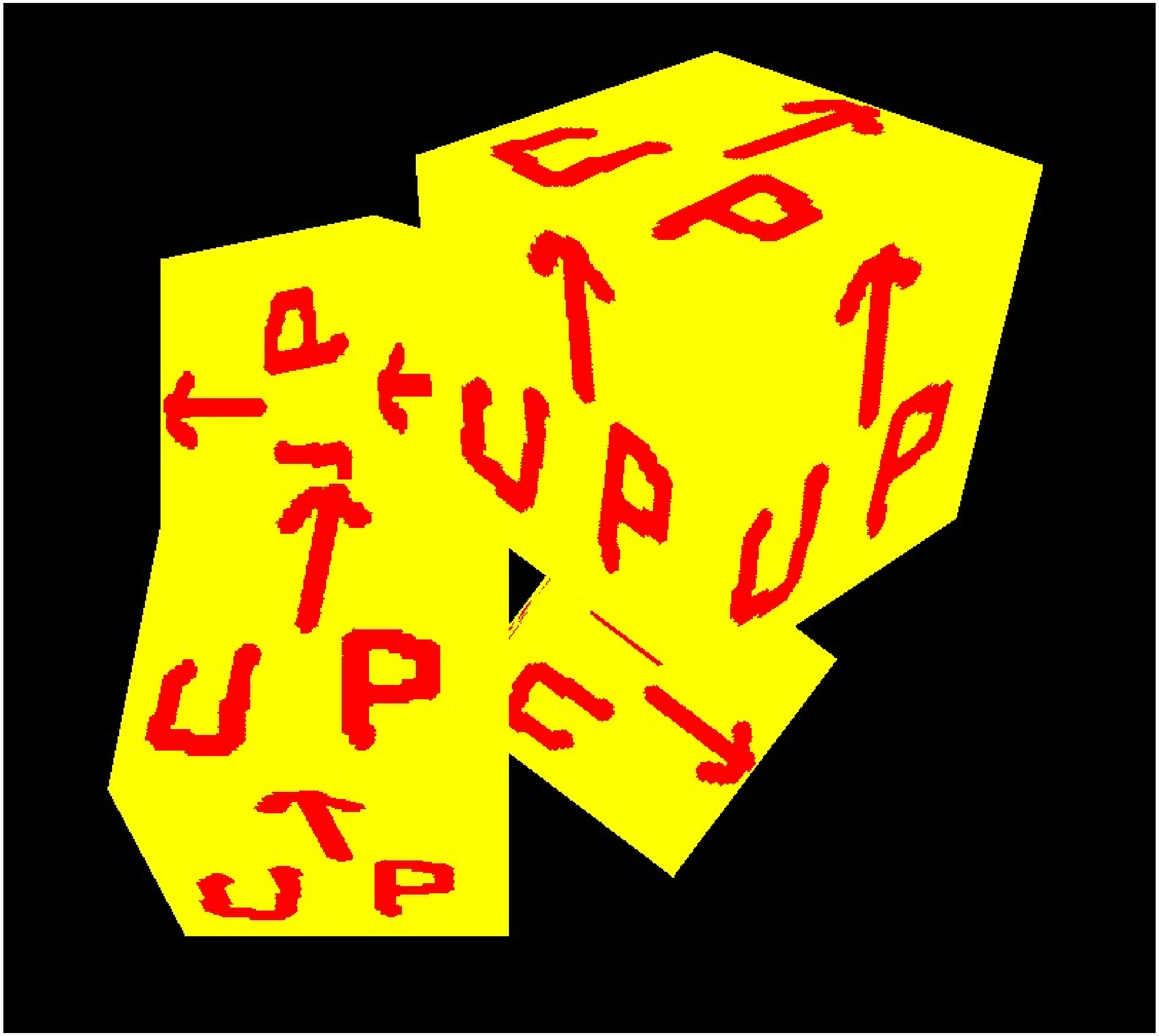


```
Screen CreateWorld(Rect& rect, Model& model, float angle, float fScale, float fOffset)
{
    Model modelX = model * Scale(fScale, fScale, fScale);
    Model modelY = modelX * RotateZ(90);
    Model modelZ = modelX * RotateY(90);

    World world;

    world += modelX * (RotateX(angle) *                                   Translate(-fOffset, -fOffset, -20));
    world += modelY * (RotateY(angle) *                                   Translate(-fOffset,  fOffset, -40));
    world += modelZ * (RotateZ(angle) *                                   Translate( fOffset, -fOffset, -60));
    world += modelX * (RotateX(angle) * RotateY(angle) * RotateZ(angle) * Translate( fOffset,  fOffset,   0));

    Matrix pov = PointOfView({ 0, 0, 100 }, { 0, 0, 0 }, { 0, 1, 0 });
    Matrix fov = FieldOfView(45, rect.AspectRatio(), 1, 100);
    Matrix view = Viewport(rect, 0, 100);

    Screen screen = world * (pov * fov * view);
    screen.PerspectiveDivide();

    return screen;
}
```

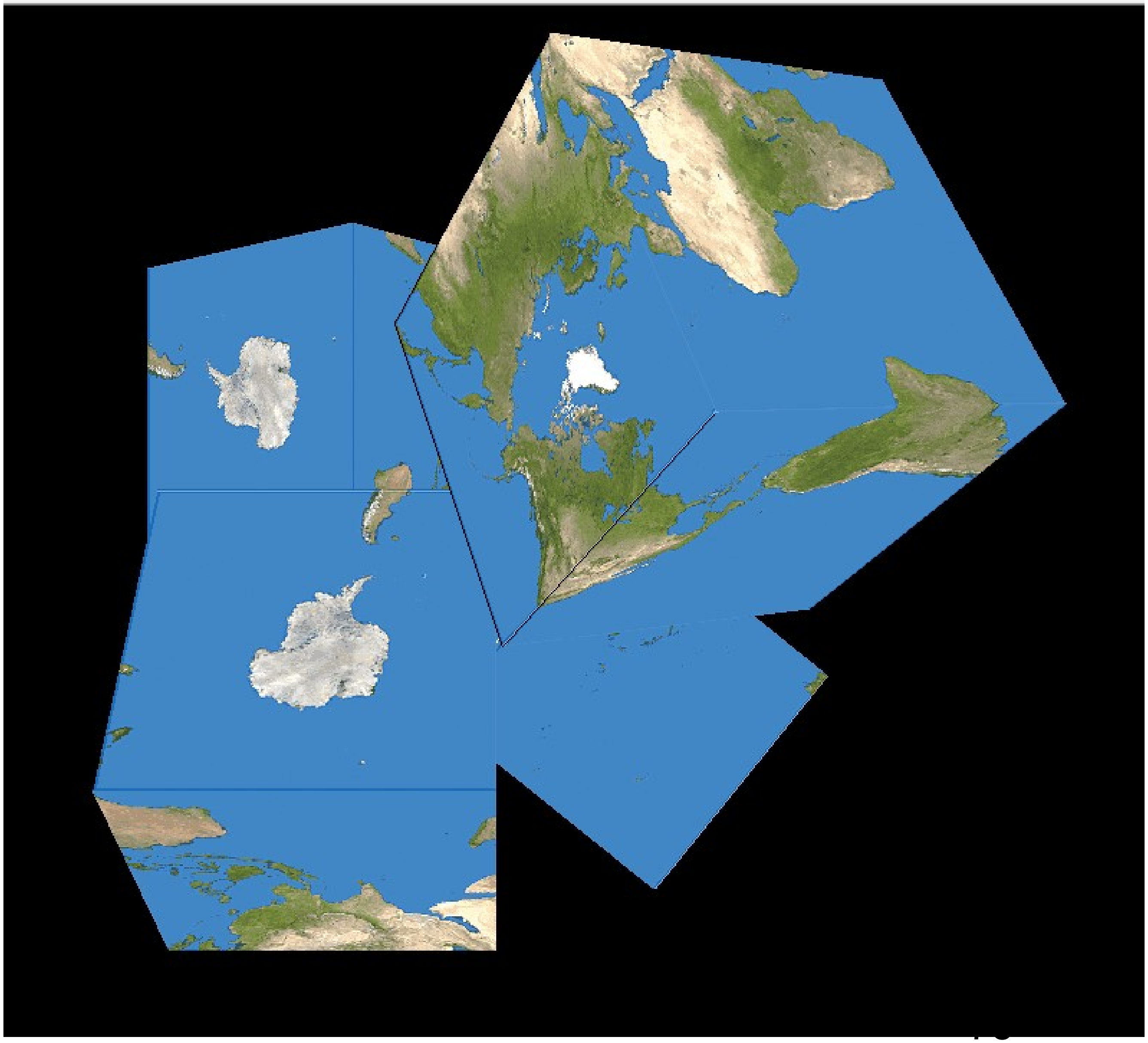
3D Graphics for Dummies



3D Graphics for Dummies



3D Graphics for Dummies



DEMO

Questions ?