3D Graphics for Dummies

CHRIS RYAN

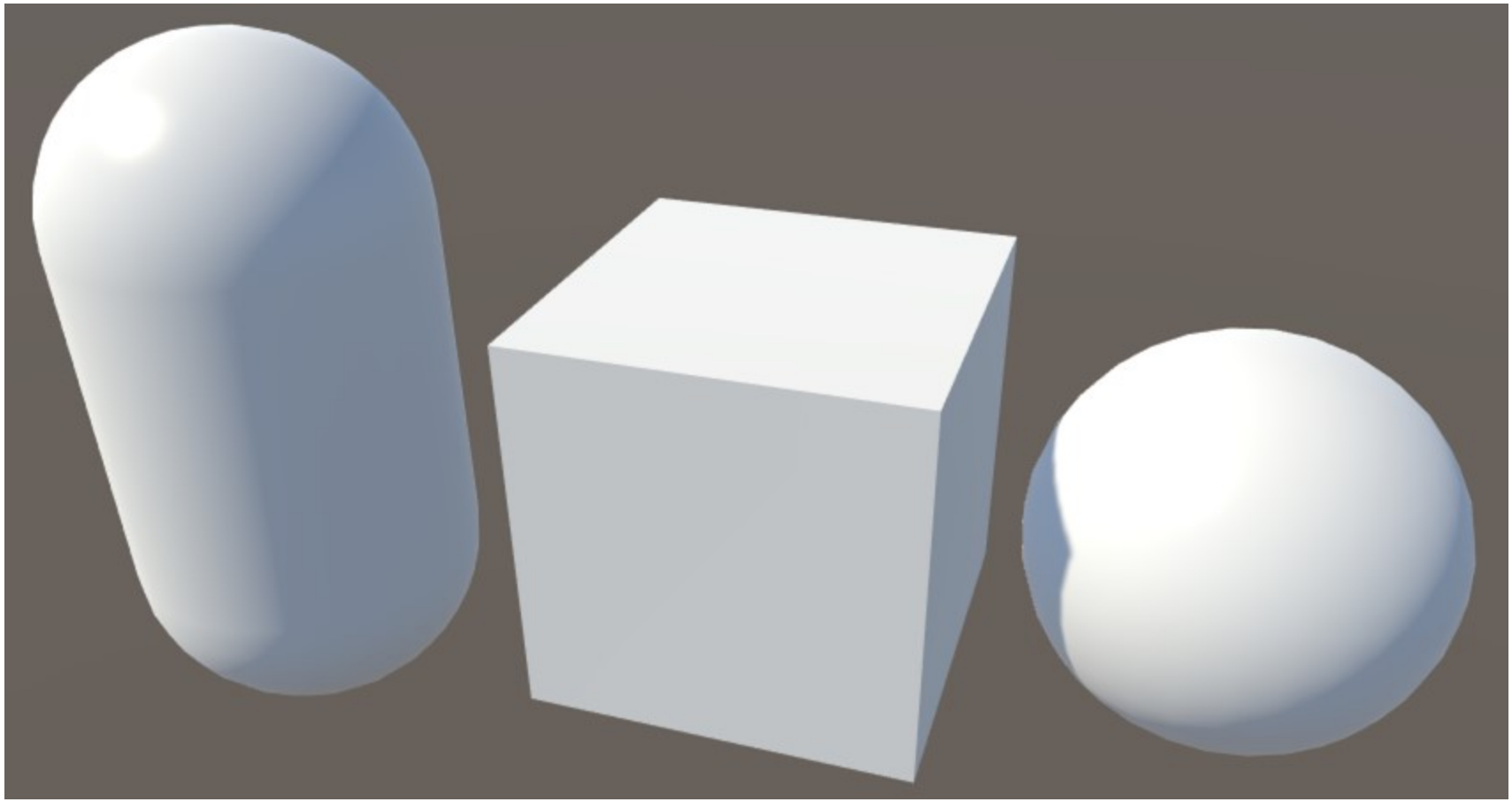
3D Graphics for Dummies

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

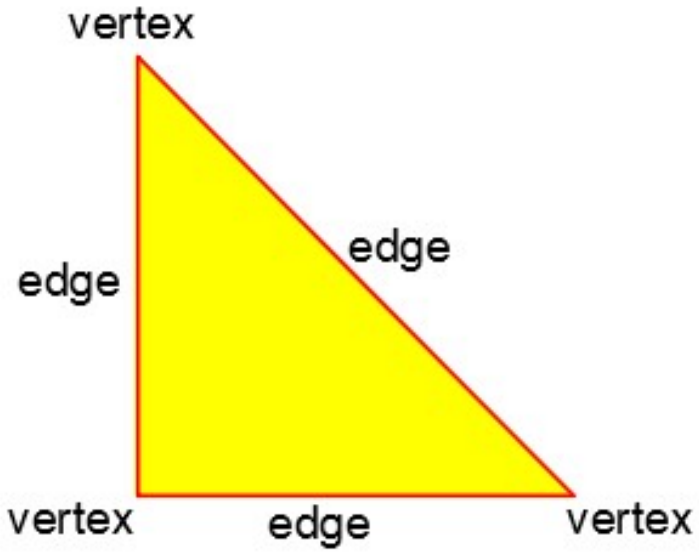
Chris Ryan CppCon 2021

github.com/ChrisR98008/CppCon2021

3D Graphics for Dummies

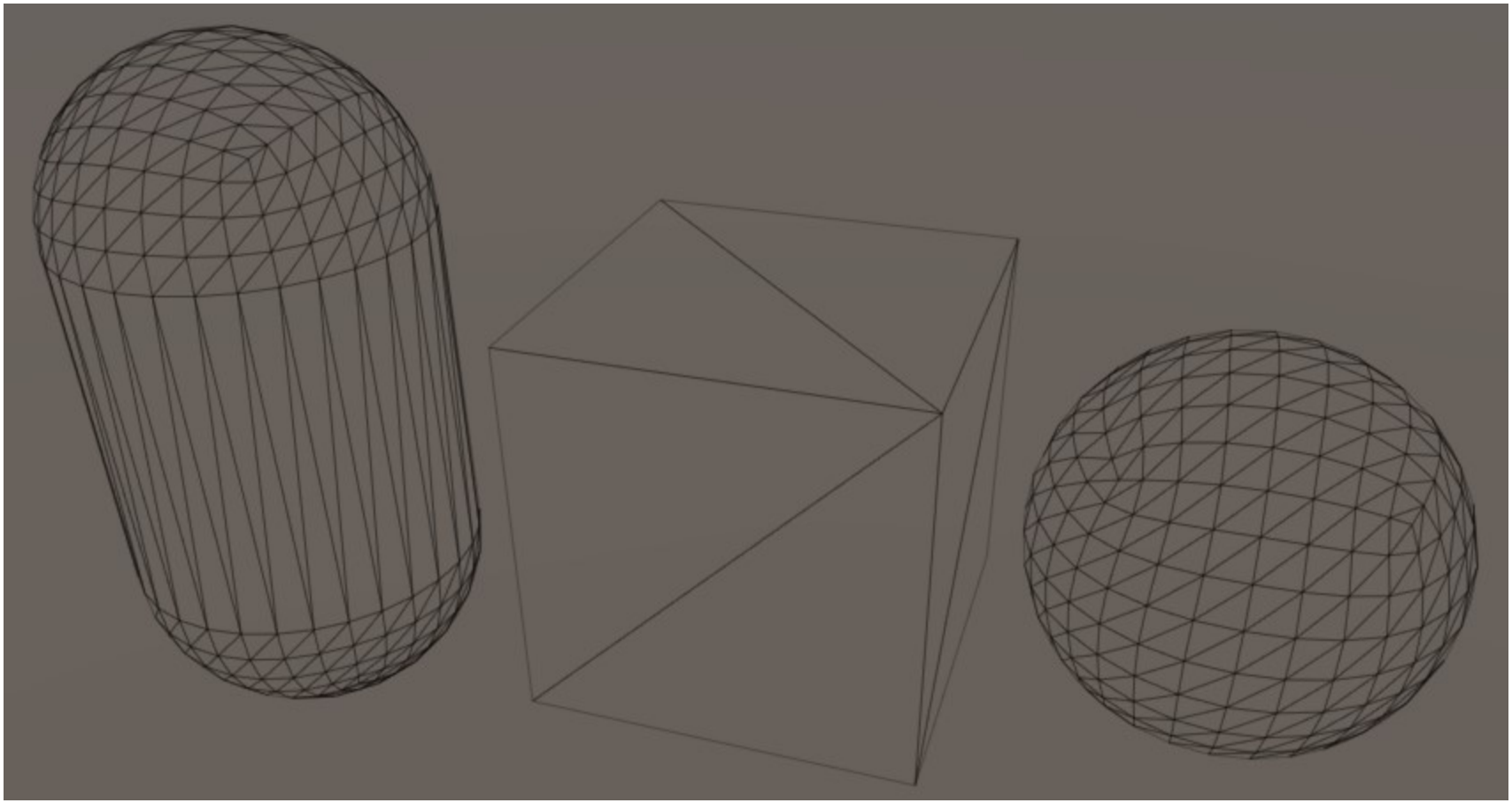


Draw Lots of Triangles

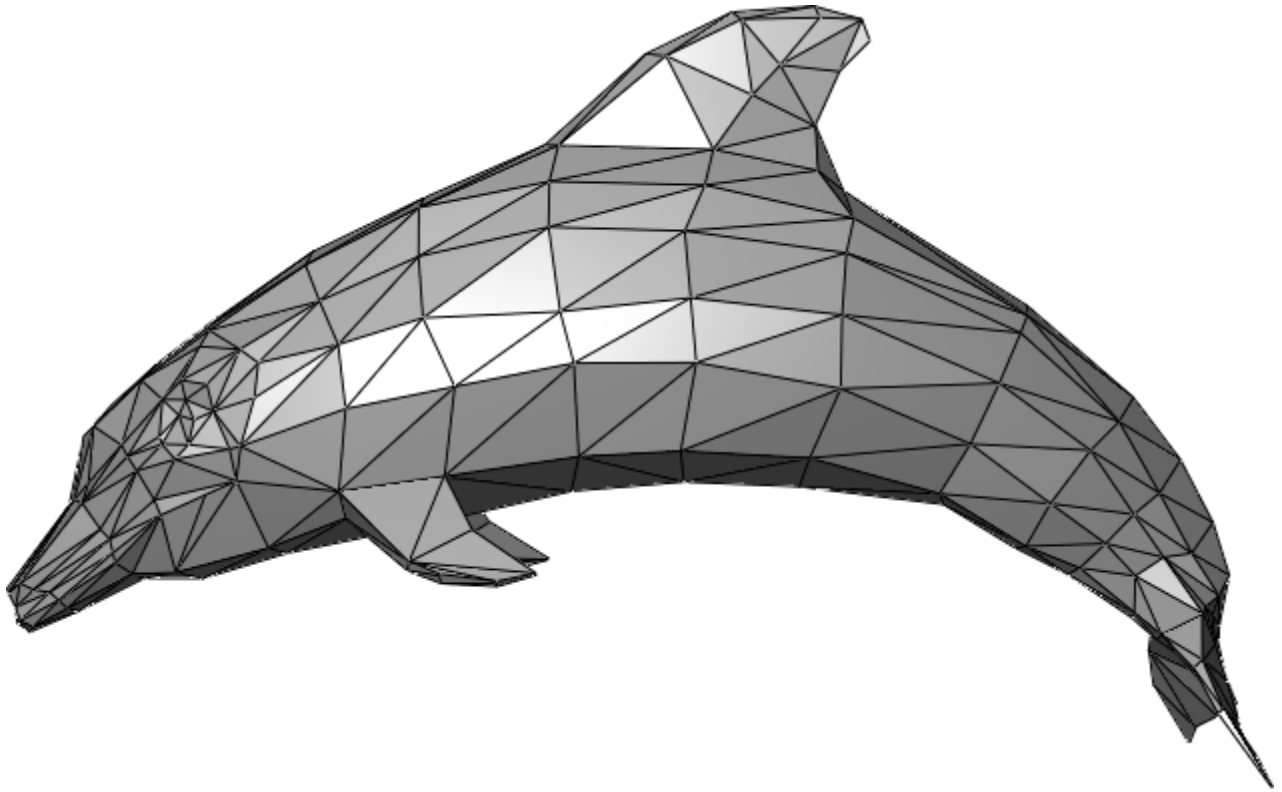


sometimes also called **polygons** or **polys**

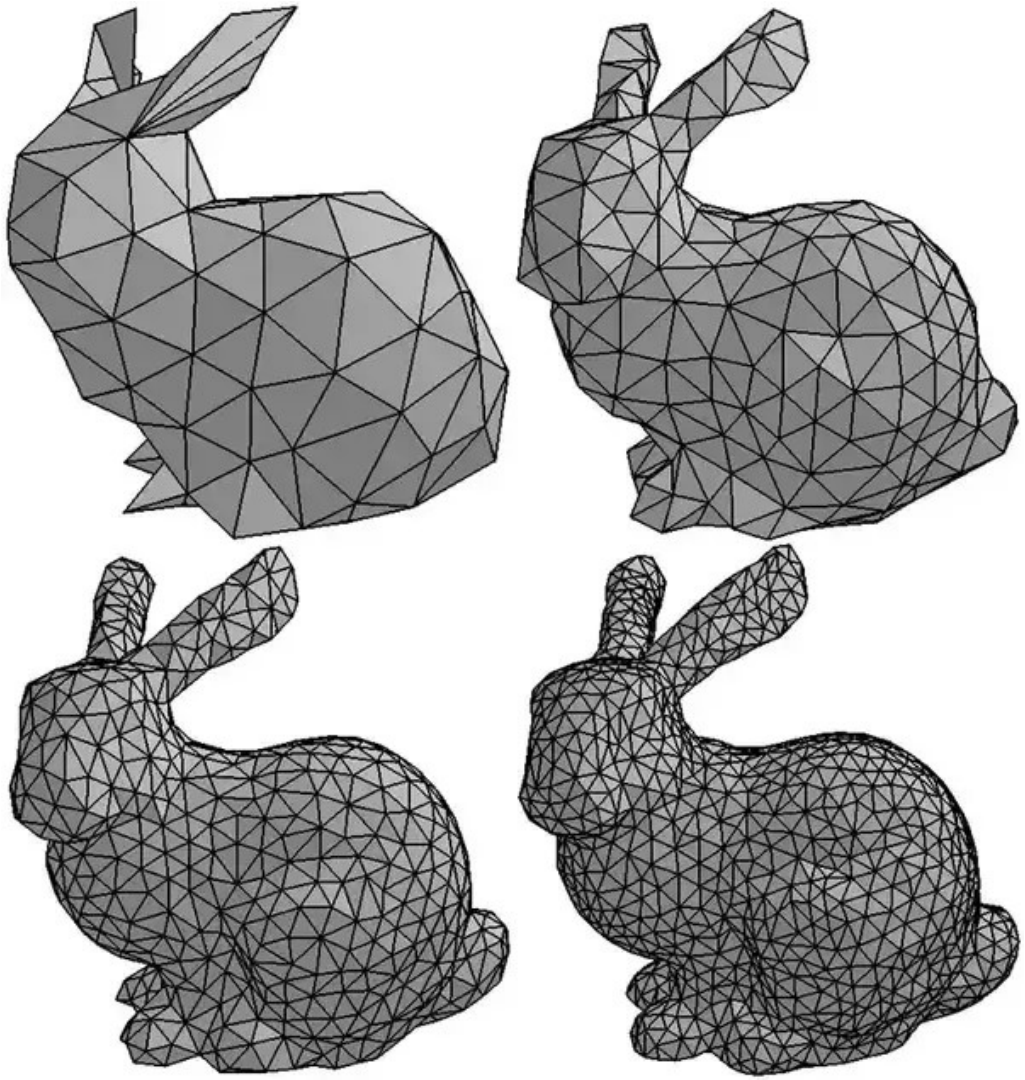
3D Graphics for Dummies



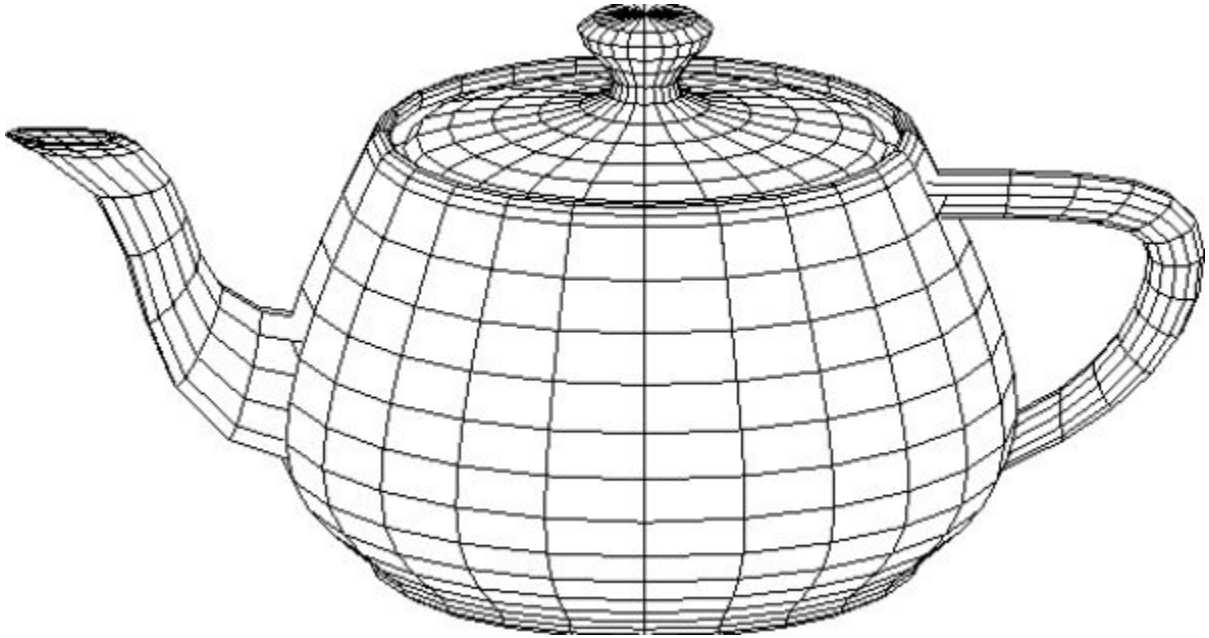
3D Graphics for Dummies



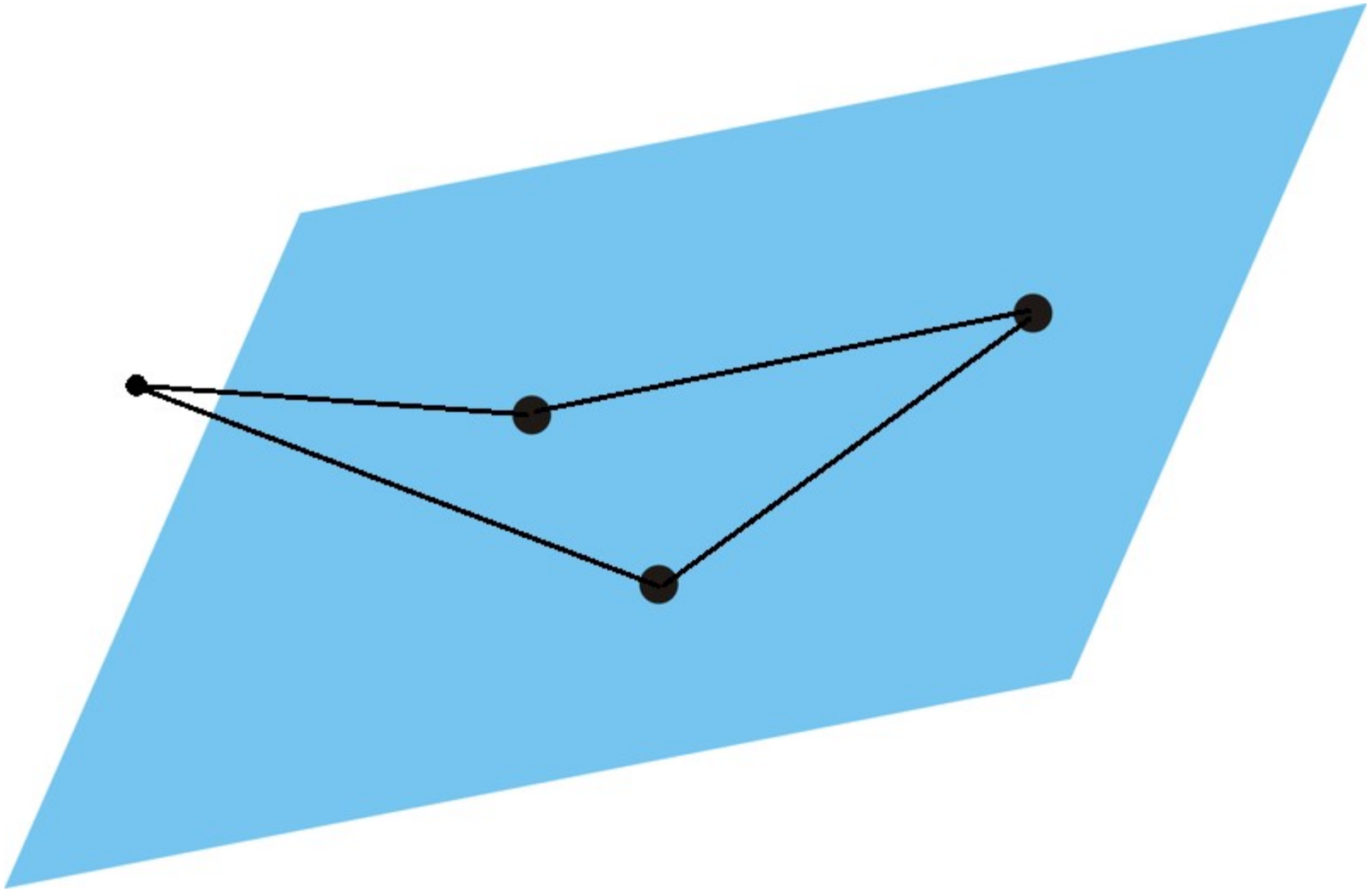
3D Graphics for Dummies



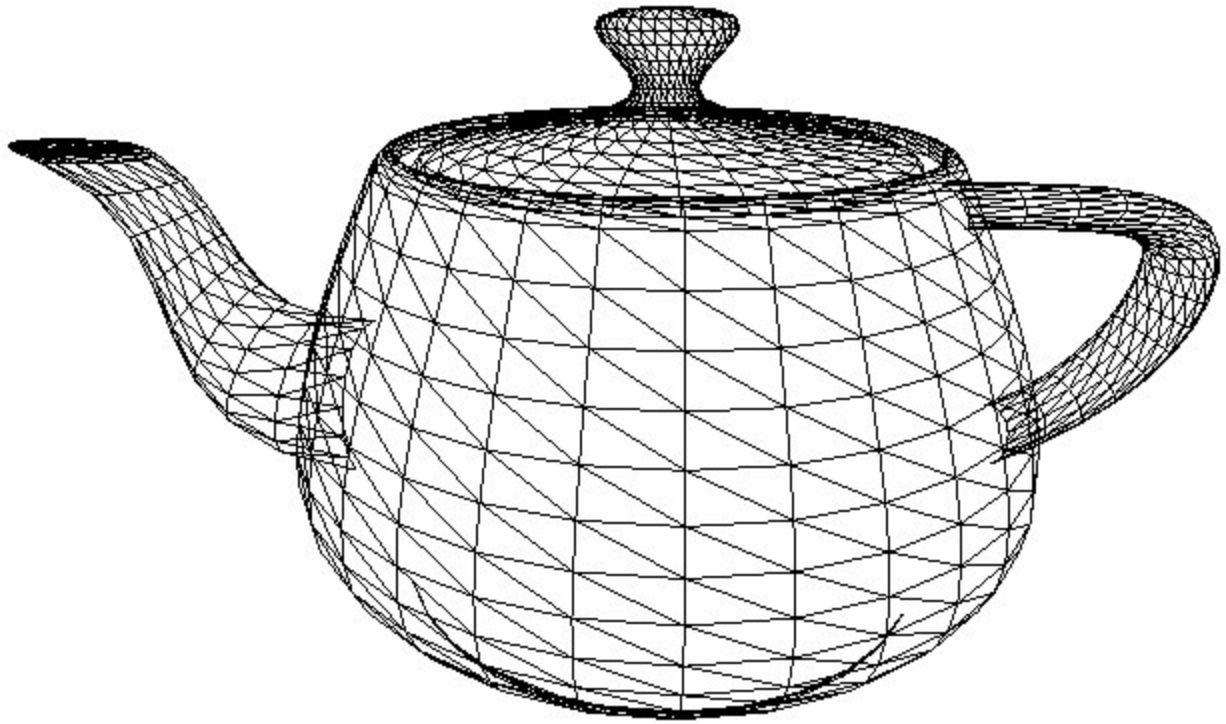
3D Graphics for Dummies



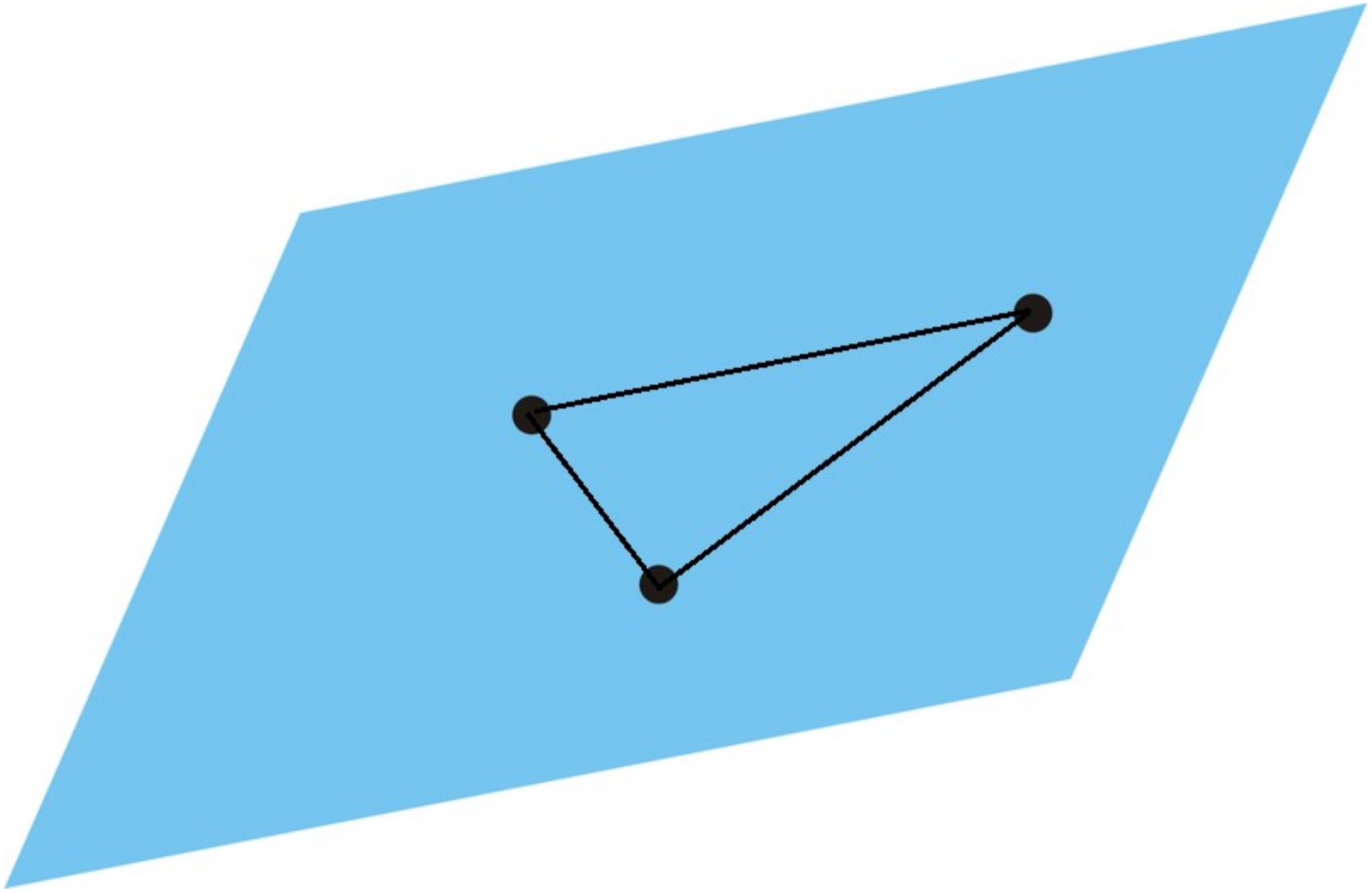
3D Graphics for Dummies



3D Graphics for Dummies



3D Graphics for Dummies



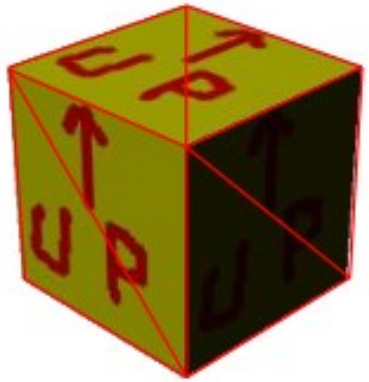
A Model

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

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

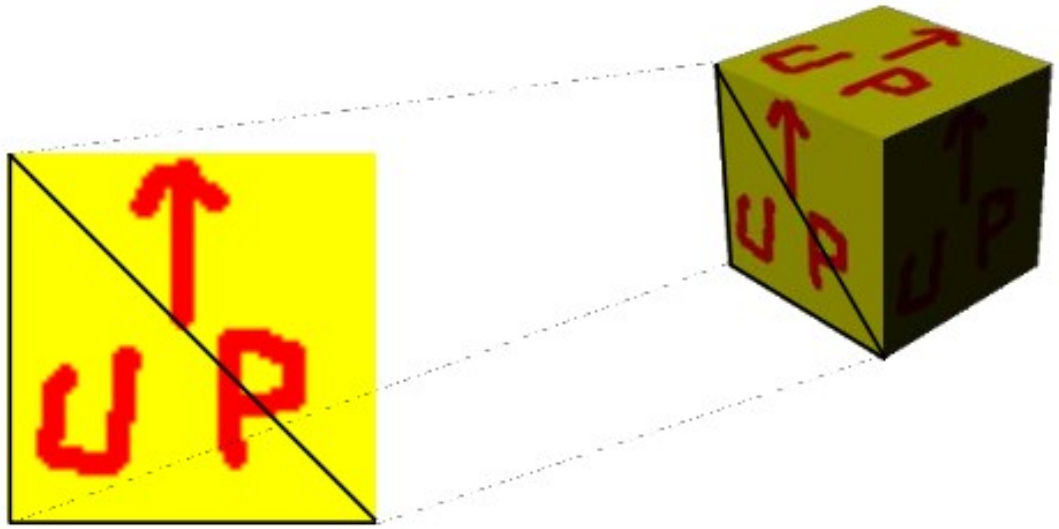
A Model

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



Textures

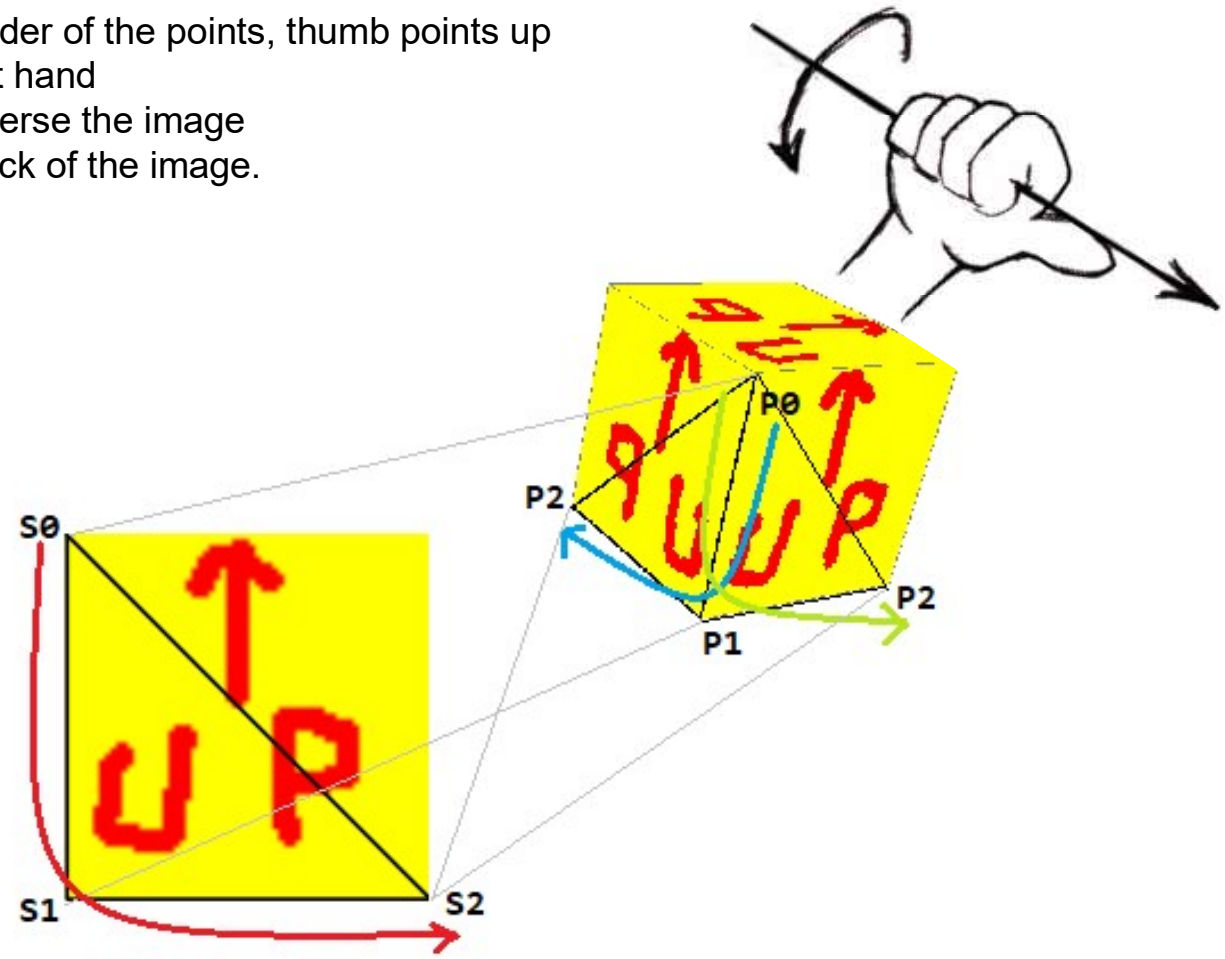
- Are 2D images which are applied to triangles
- Rough analogy: stickers or decals
  - Except textures can be stretched
  - Each triangle specifies which part of texture gets applied to it



3D Graphics for Dummies

Winding / Right Hand Rule

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

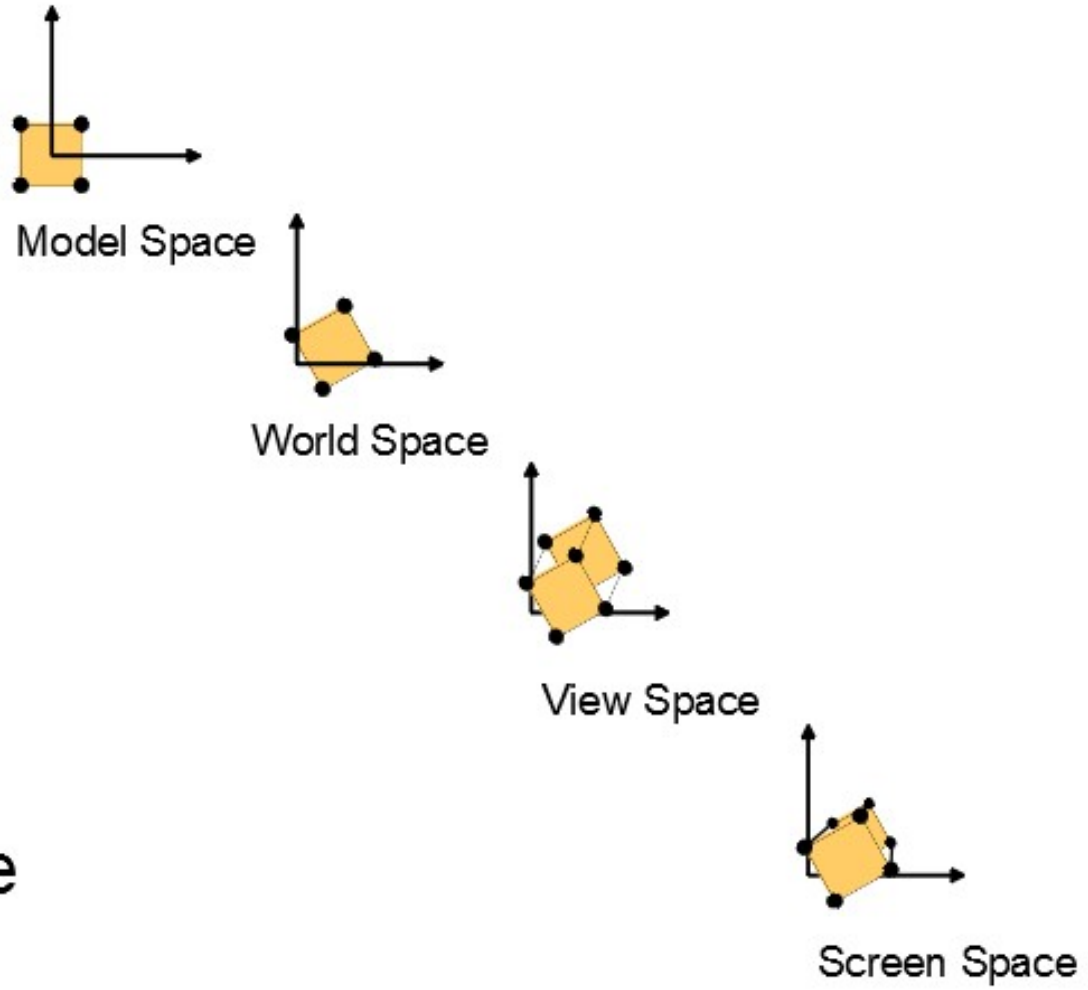


Model has Matrix

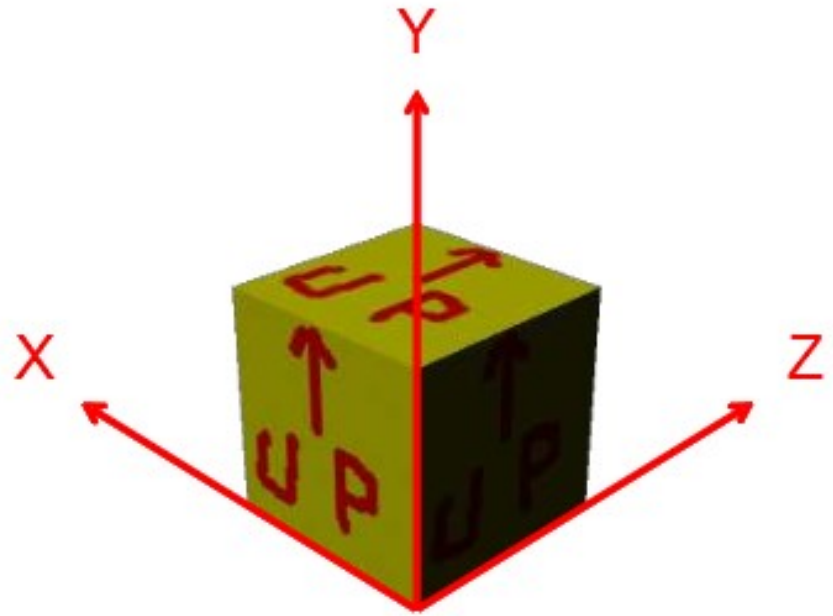
- Every model has a **`Matrix`**
  - Specifies Position in the world
  - Specifies Orientation in the world
  - Specifies Scale (size) in the world
- A **`Matrix`** used in this way is also called an "**model to world space** matrix"
  - or "**model to world** matrix"
  - or sometimes just "**world**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

Coordinate Spaces Overview

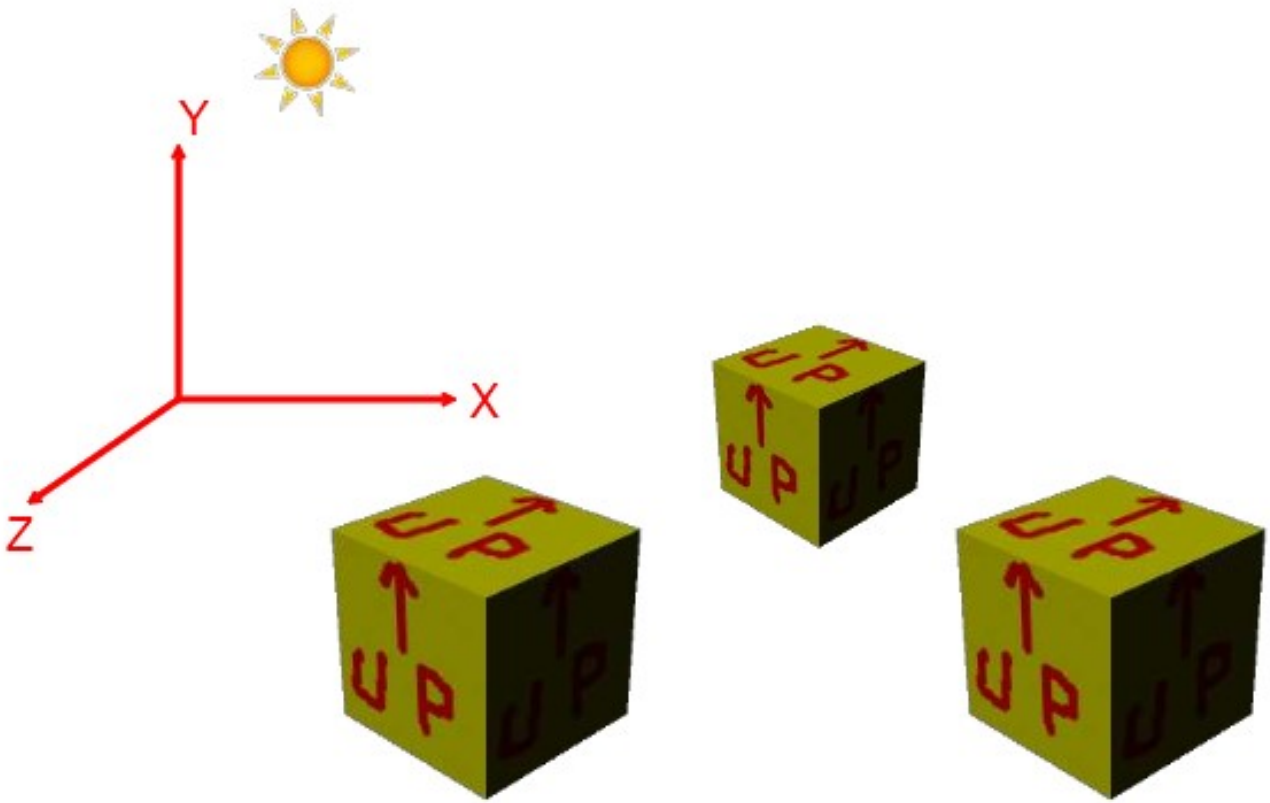
- Model Space
- World Space
- View Space
- Screen Space



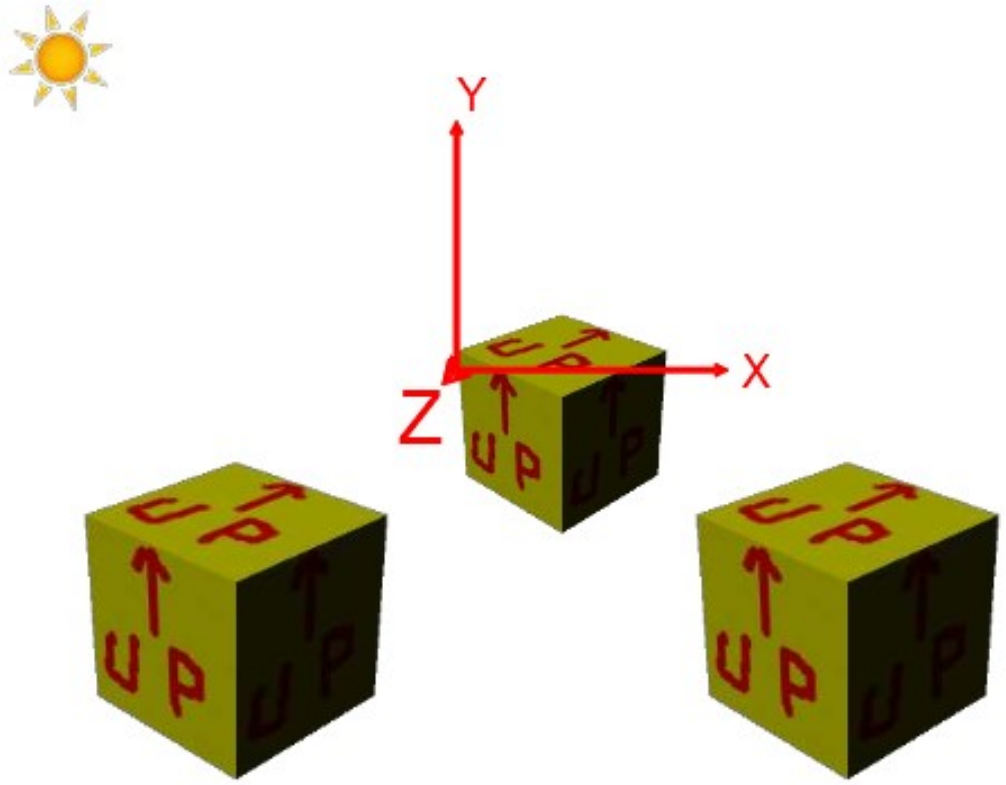
Model Space



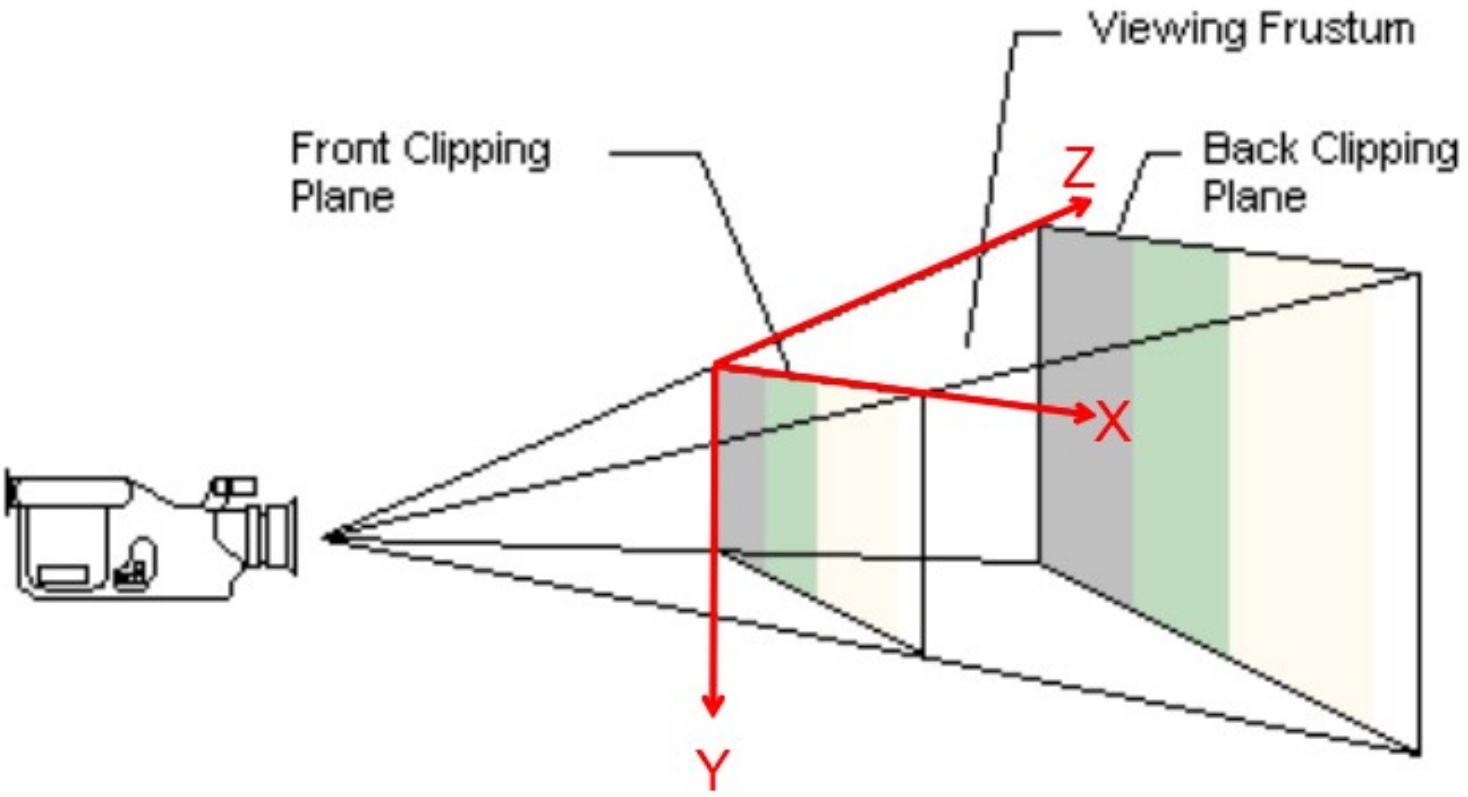
World Space



View Space



Screen Space

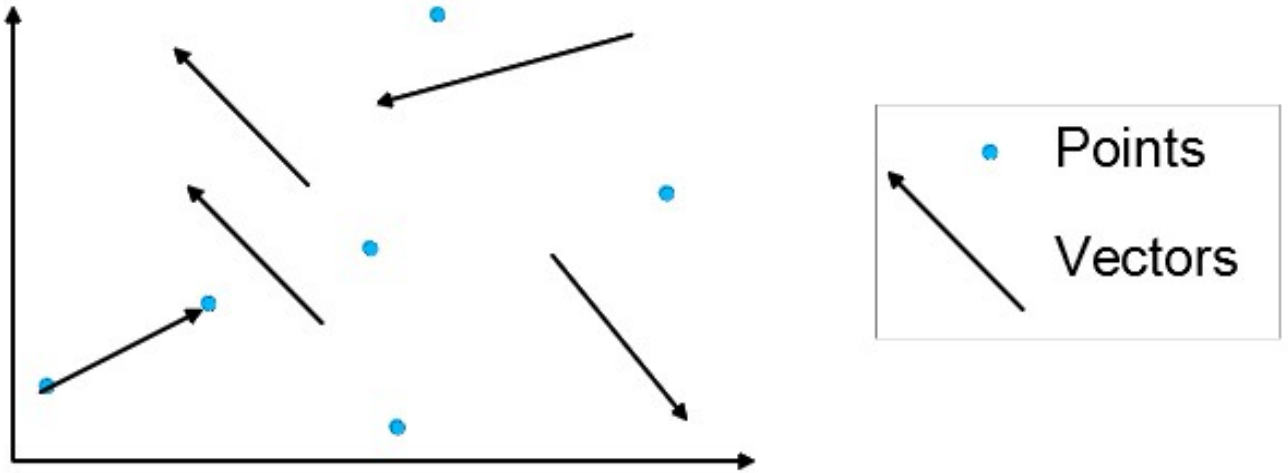


World Space to View Space

- View Matrix (“oriented from camera view”)
- World Space × View Matrix = View Space
- View Space sometimes called Camera Space

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")

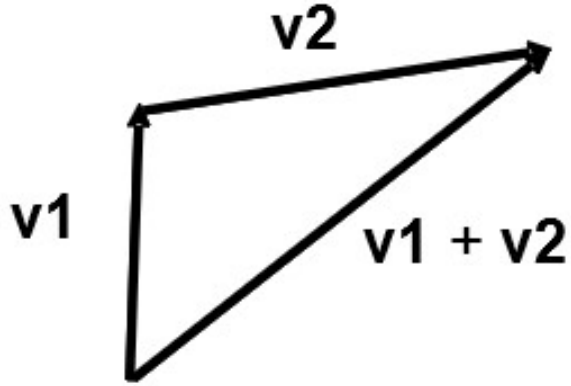


Points and Vectors

- 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

3D Graphics for Dummies

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

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

vector operators:
    vector = vector * matrix;       vector *= matrix;
    vector = vector * vector;       vector *= vector;
    vector = Normalize(vector);

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

matrix operators:
    atrix = matrix * matrix;       matrix *= matrix;

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

```
class Matrix
{
    float _d[4][4];
    Matrix(...);

    Matrix& operator = (const Matrix& rhs)
    Matrix Multiply(const Matrix& rhs) const
};

class Data4 // base of point & vector
{
    float _x, _y, _z, _w;
    Data4(...) {}

    Data4& operator = (const Data4& rhs);
    void Multiply(const Matrix& rhs);
};
```

3D Graphics for Dummies

```
class Vector : public Data4
{
    Vector(...) : Data4(x, y, z, 0) {}

//  void Multiply(const Matrix& rhs);  //in the base
    Vector CrossProduct(const Vector& rhs) const;
    Vector Normalize() const;
};

class Point : public Data4
{
    Point(...) : Data4(x, y, z, 1) {}

//  void Multiply(const Matrix& rhs);  //in the base
    Point   Add(const Vector& rhs) const;
    Vector  Subtract(const Point& rhs) const;
    void    PerspectiveDivide();
};
```

3D Graphics for Dummies

```
struct Polygon
{
//  set of 3D points
//  serface ID (aka texture)
//  serface coordinates
};

class Mesh
{
    std::map<Point, uint>   _mapPoints;
    std::vector<Point>      _points;
    std::vector<Polygon>    _polygons;

    Mesh(...) {}
    Mesh operator = (const Mesh& rhs);
    Mesh&   AddTo(const Mesh& rhs);
    Mesh&   Multiply(const Matrix& matrix);
    void    PerspectiveDivide();
};
```

```
Vector  Normalize   (Vector& rhs);              // Vector::Normalize()
Vector  operator *  (Vector& lhs, Matrix& rhs); // Vector::Multiply(rhs)
Vector& operator *= (Vector& lhs, Matrix& rhs); // Vector::Multiply(rhs)
Vector  operator *  (Vector& lhs, Vector& rhs); // Vector::CrossProduct(rhs)
Vector& operator *= (Vector& lhs, Vector& rhs); // Vector::CrossProduct(rhs)

Point   operator *  (Point& lhs,  Matrix& rhs); // Point::Multiply(rhs)
Point&  operator *= (Point& lhs,  Matrix& rhs); // Point::Multiply(rhs)
Point   operator +  (Point& lhs,  Vector& rhs); // Point::Add(rhs)
Point&  operator += (Point& lhs,  Vector& rhs); // Point::Add(rhs)
Vector  operator -  (Point& lhs,  Point&  rhs); // Point::Subtract(rhs)

Matrix  operator *  (Matrix& lhs, Matrix& rhs); // Matrix::Multiply(rhs)
Matrix& operator *= (Matrix& lhs, Matrix& rhs); // Matrix::Multiply(rhs)
```

3D Graphics for Dummies

```
using Model = Mesh;
using World = Mesh;
using Screen = Mesh;

Mesh    operator +  (Mesh&  lhs,  Mesh&   rhs); // Mesh::AddTo(rhs)
Mesh&   operator += (Mesh&  lhs,  Mesh&   rhs); // Mesh::AddTo(rhs)
Mesh    operator *  (Mesh&  lhs,  Matrix& rhs); // Mesh::Multiply(rhs)
Mesh&   operator *= (Mesh&  lhs,  Matrix& rhs); // Mesh::Multiply(rhs)
```

3D Graphics for Dummies

```
Matrix Identity();

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

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);
Screen ScreenTrasnform(World& world, Rect& rect,
                       Point eye, Point target, Vector up,
                       float fovAngle, float nearPlane, float farPlane);
```

Matrix Math

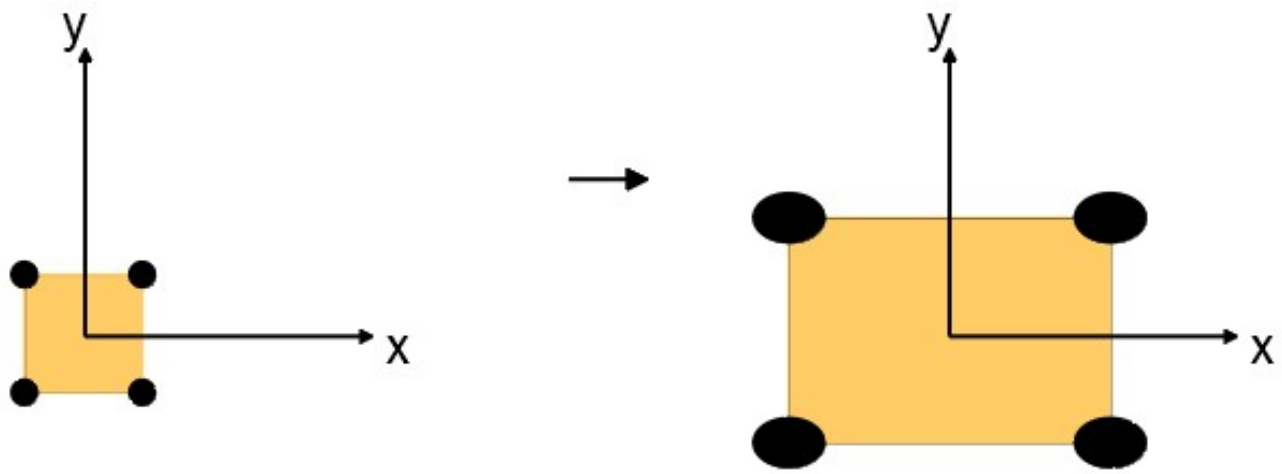
$$\mathbf{v} * \mathbf{M} = [vx\ vy\ vz\ vw] * \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix}$$

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

Scaling Matrix

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

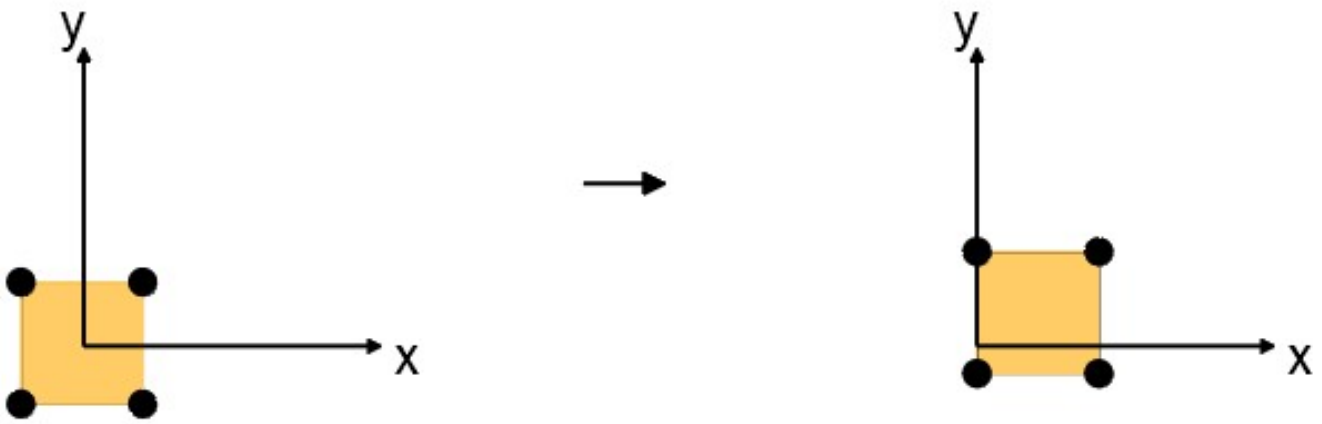
```
float x = 3, y = 2, z = 1;
Matrix mat = Scale(x, y, z);
```



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);
```



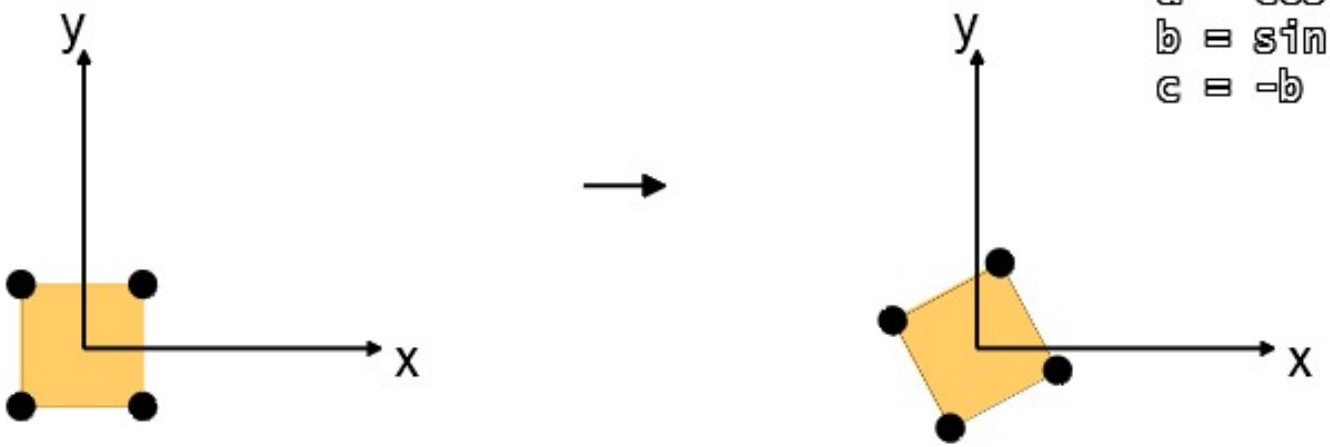
Z Rotation Matrix

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

Z axis

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

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



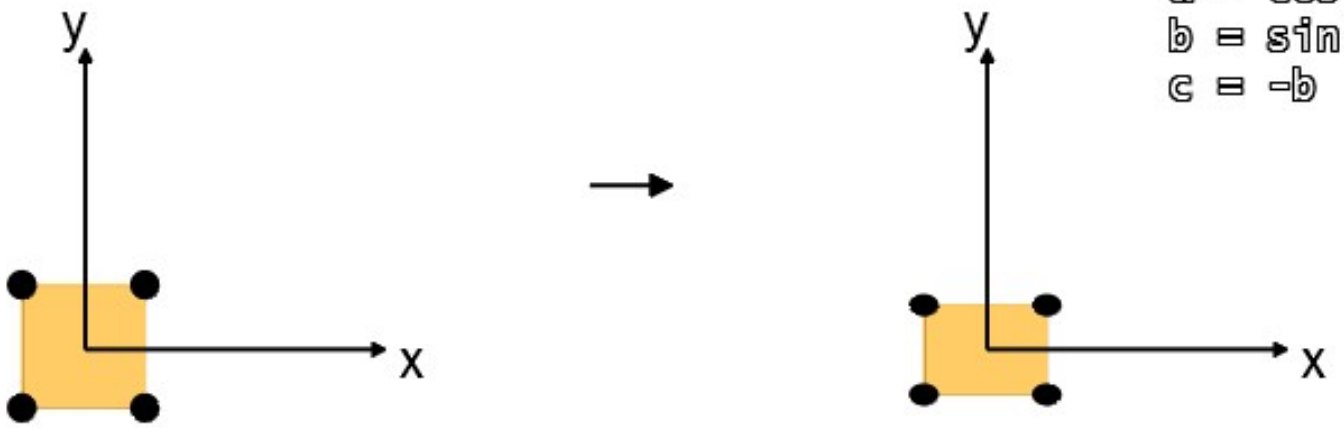
X Rotation Matrix

```
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

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



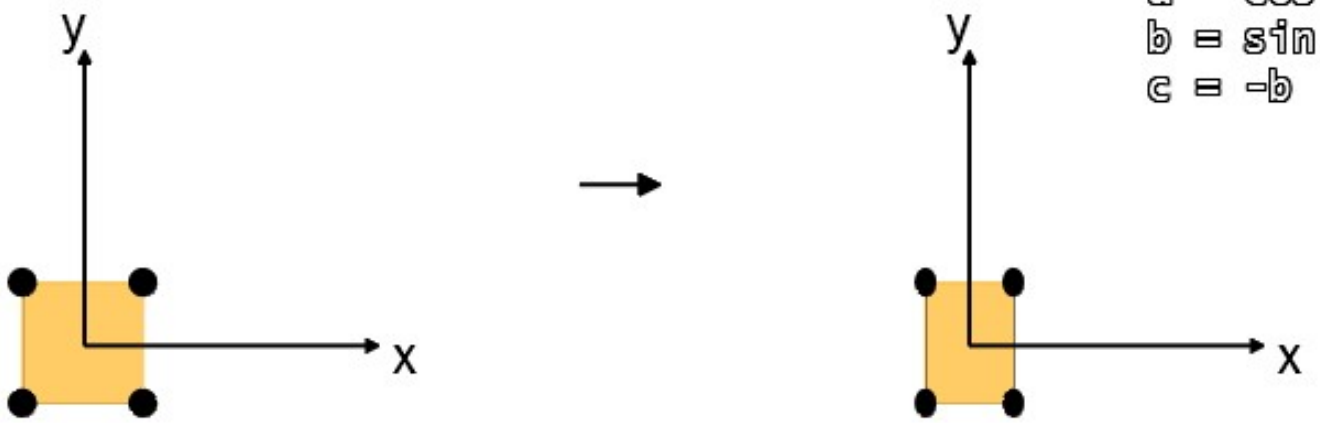
Y Rotation Matrix

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

Y axis

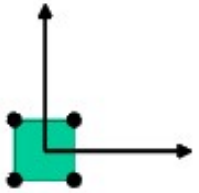
$a = \cos(\Phi)$
$b = \sin(\Phi)$
$c = -b$

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

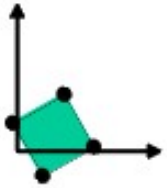


Coordinate Spaces Overview

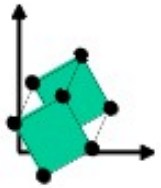
- Model Space
- World Space
- View Space
- Screen Space



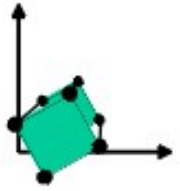
Model Space



World Space

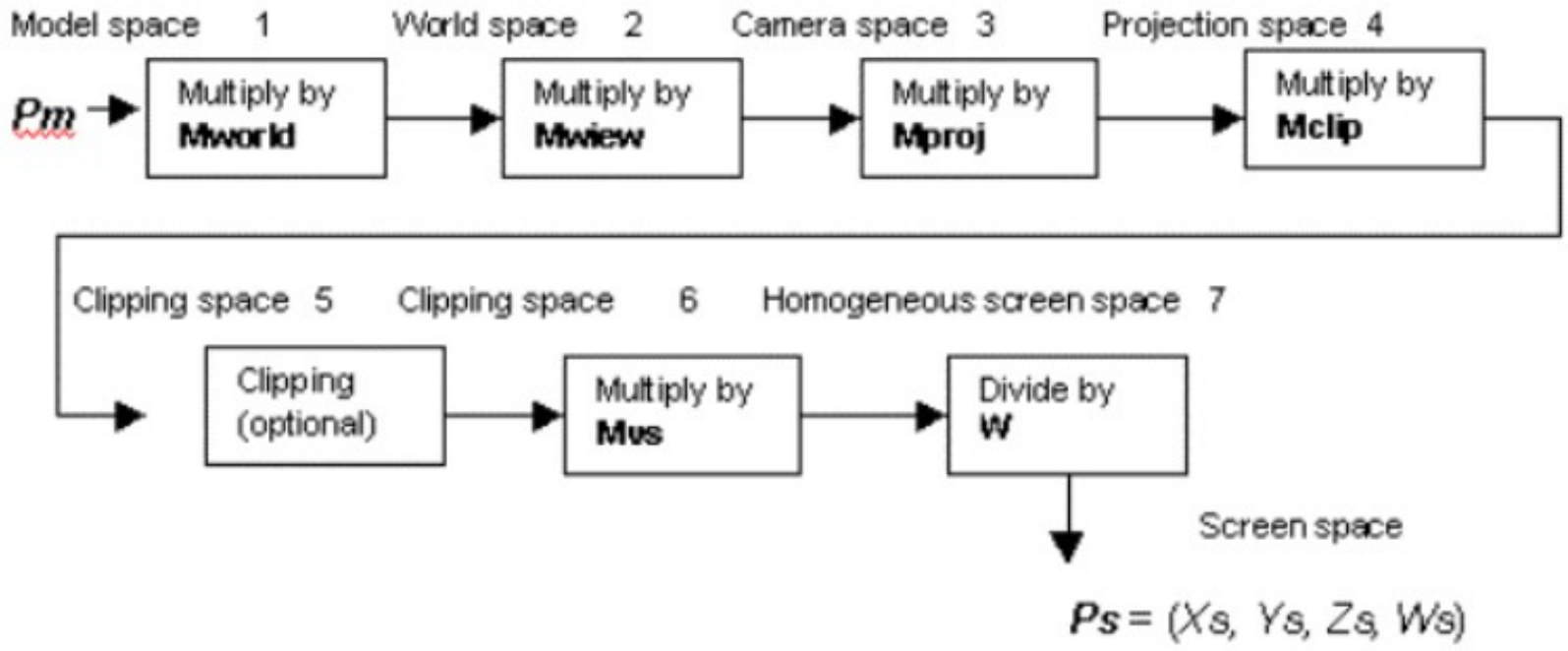


View Space



Screen Space

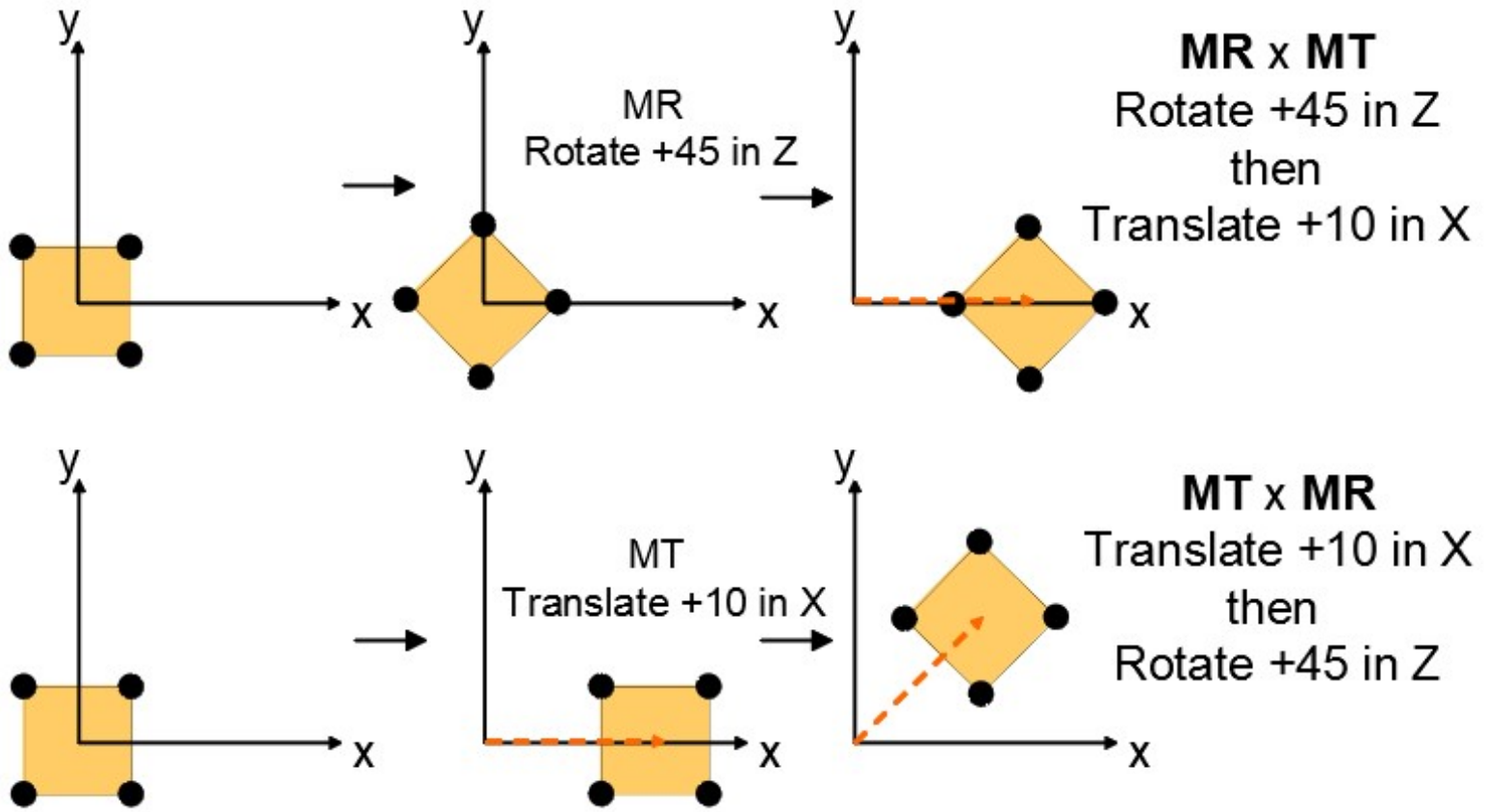
Transformation Pipeline



World Matrix

```
Matrix matRot = RotationZ(30);

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

Matrix world = matRot * matTrans;
```

Geometric Interpretation: Matrix Multiplication is Not Commutative



Matrix Math Multiply Order

- $v * M1 * M2 * M3$

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

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

$M = M1 * M2 * M3$

$= v * M$

World Matrix

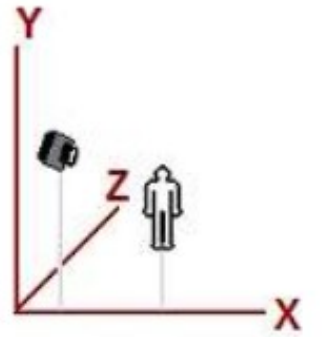
```
Matrix matRot = RotationZ(30);

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

Matrix world = matRot * matTrans;
```

View Matrix

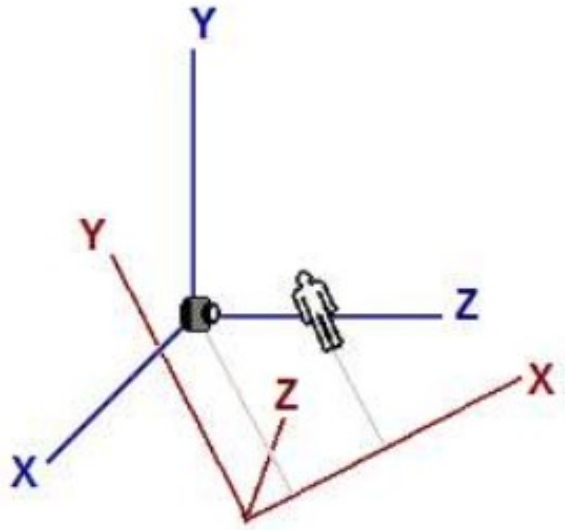
```
Point cameraPosition(0, 92, 5);
Point cameraTarget(0, 100, 0);
Vector cameraUpVector(0, 1, 0);
Matrix view = PointOfView( cameraPosition,
                           cameraTarget,
                           cameraUpVector);
```

$$\begin{vmatrix} 1.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.530 & -0.848 & 0.000 \\ 0.000 & 0.848 & 0.530 & 0.000 \\ 0.000 & -53.00 & 75.37 & 1.000 \end{vmatrix}$$

World Space to View Space

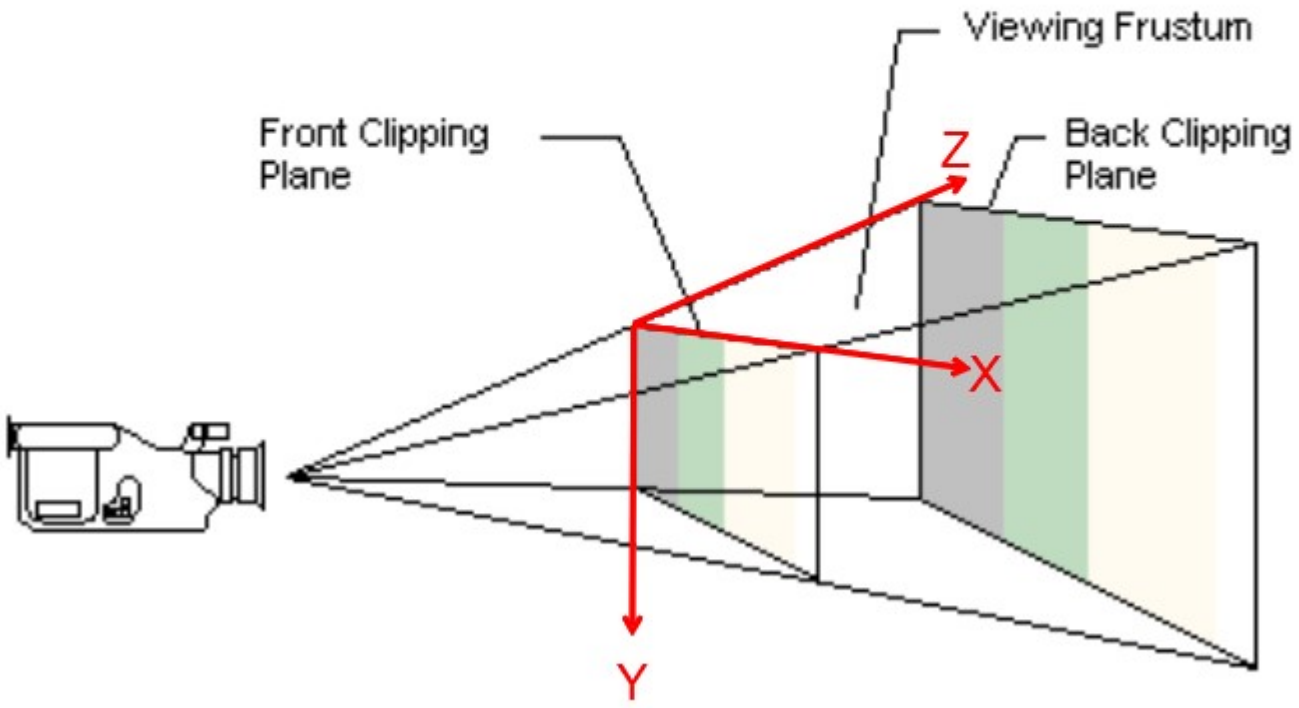


world space



view (camera) space

Add Perspective: View Frustum



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**

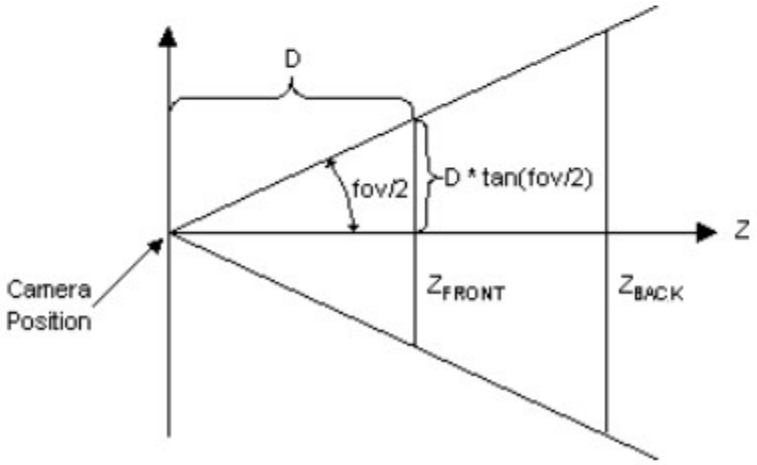
Creating Projection Matrices

```
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

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 |
```

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 |
```

Divide by *w*

- Ready to show on screen

$$X_S = \frac{X}{W}, Y_S = \frac{Y}{W}, Z_S = \frac{Z}{W}, W_S = \frac{1}{W}$$

- Divide by *w* also called perspective divide

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

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

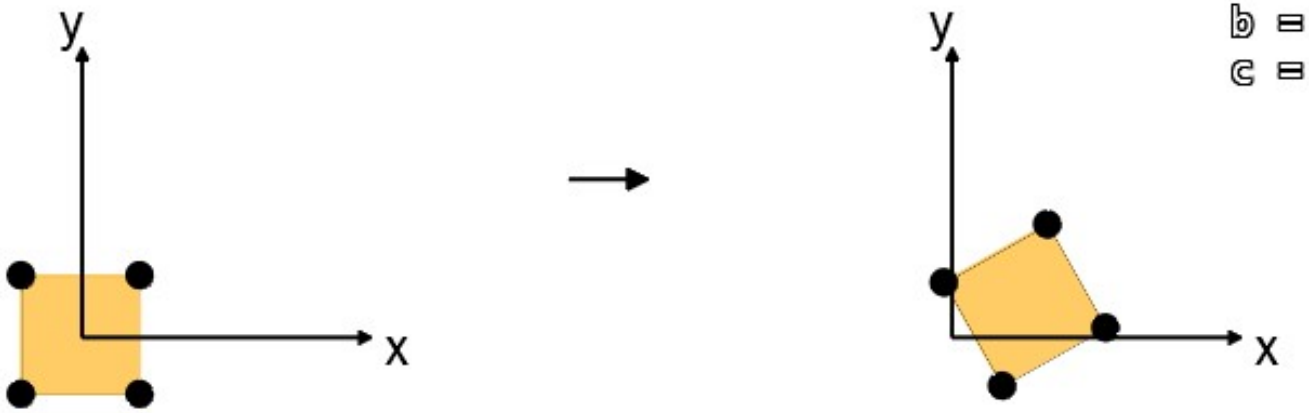


Image Buffers and Z-Buffer

- 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

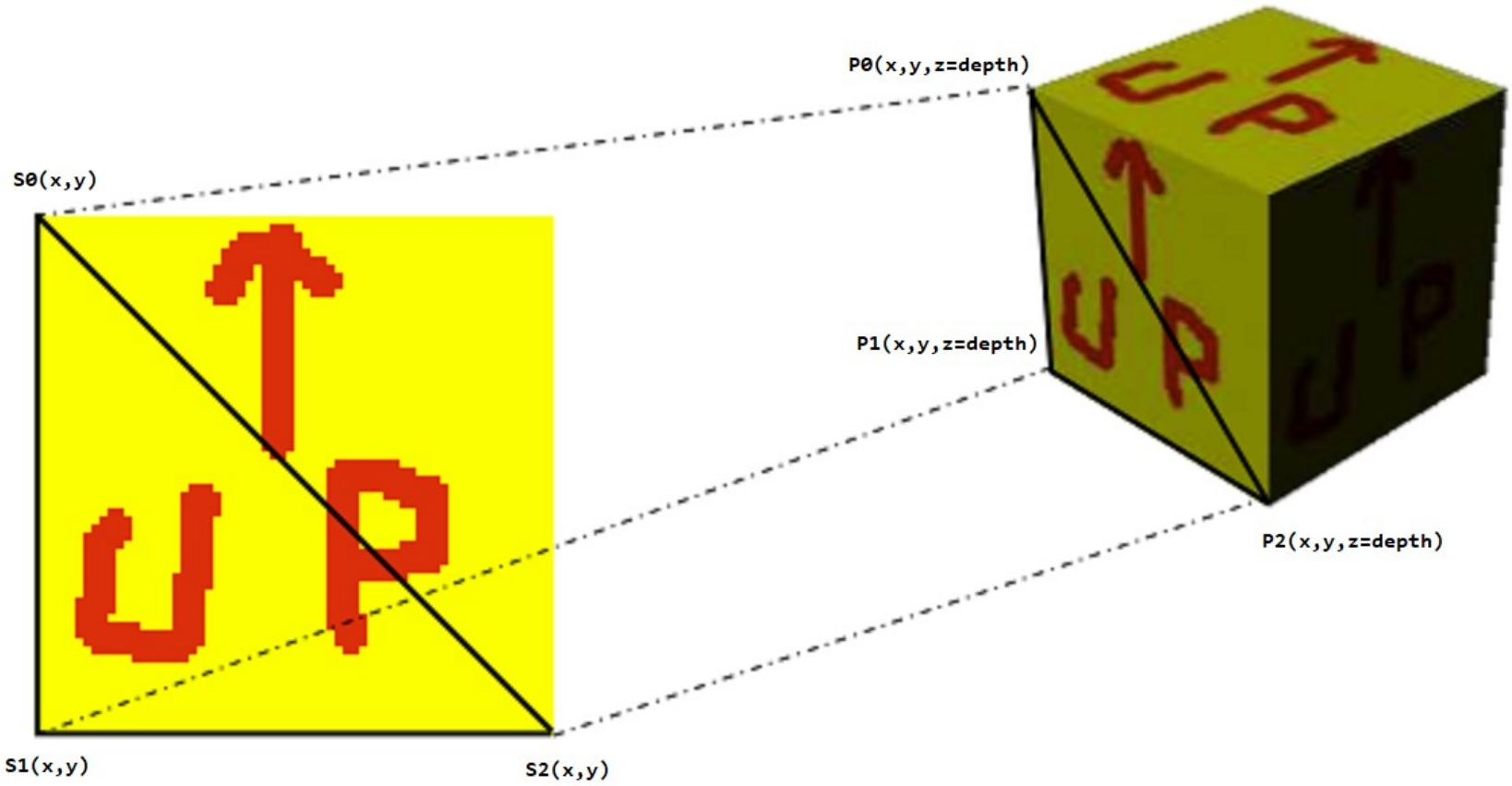
- Before drawing each pixel on screen:
- **Z-Test**: compare Z value ([0 1] in view frustum) against value in Z-buffer
- If (Z value < value in Z-buffer), update image buffer & Z-buffer

A simple three dimensional scene

Z-buffer representation

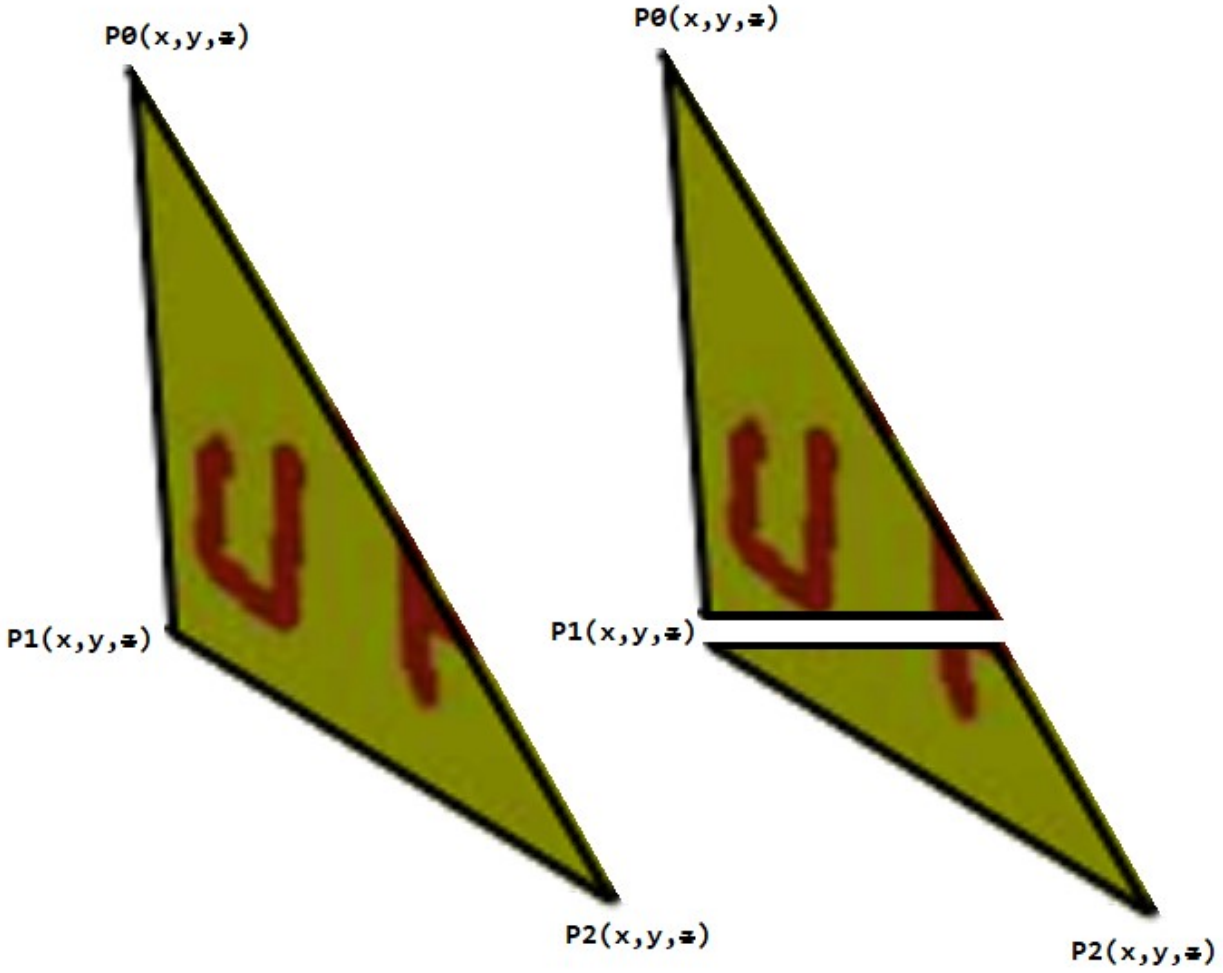
3D Graphics for Dummies

Depth / Texturing / Applying Surfaces



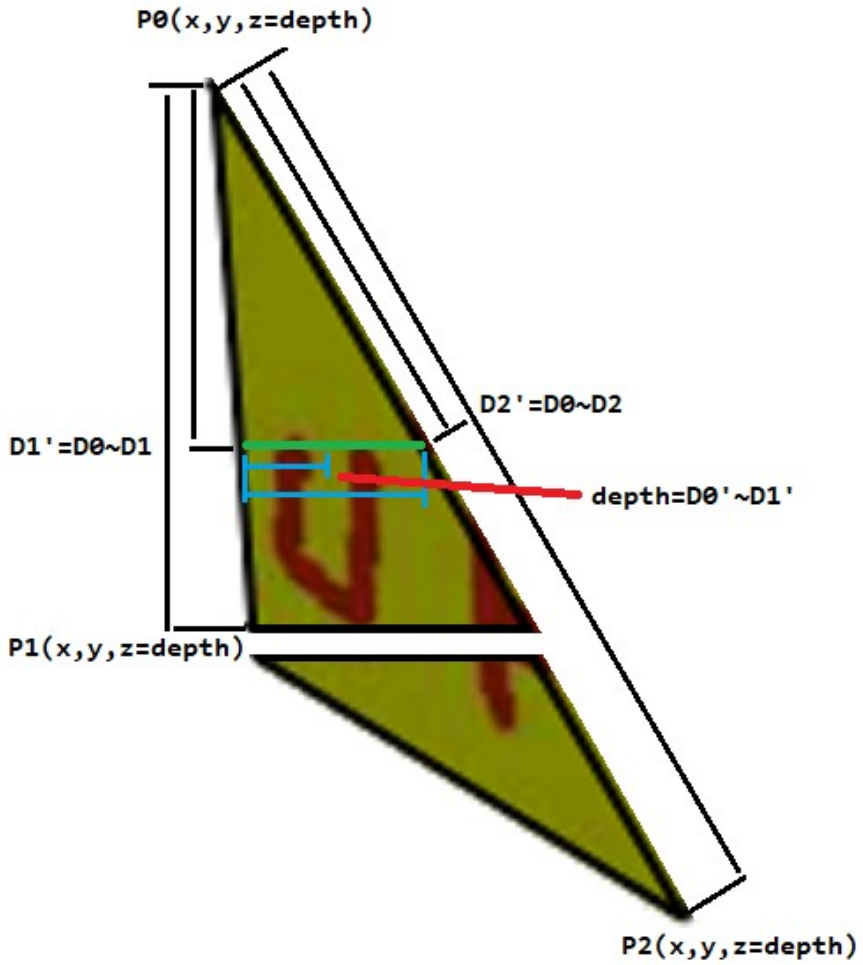
3D Graphics for Dummies

Rasterization: Depth



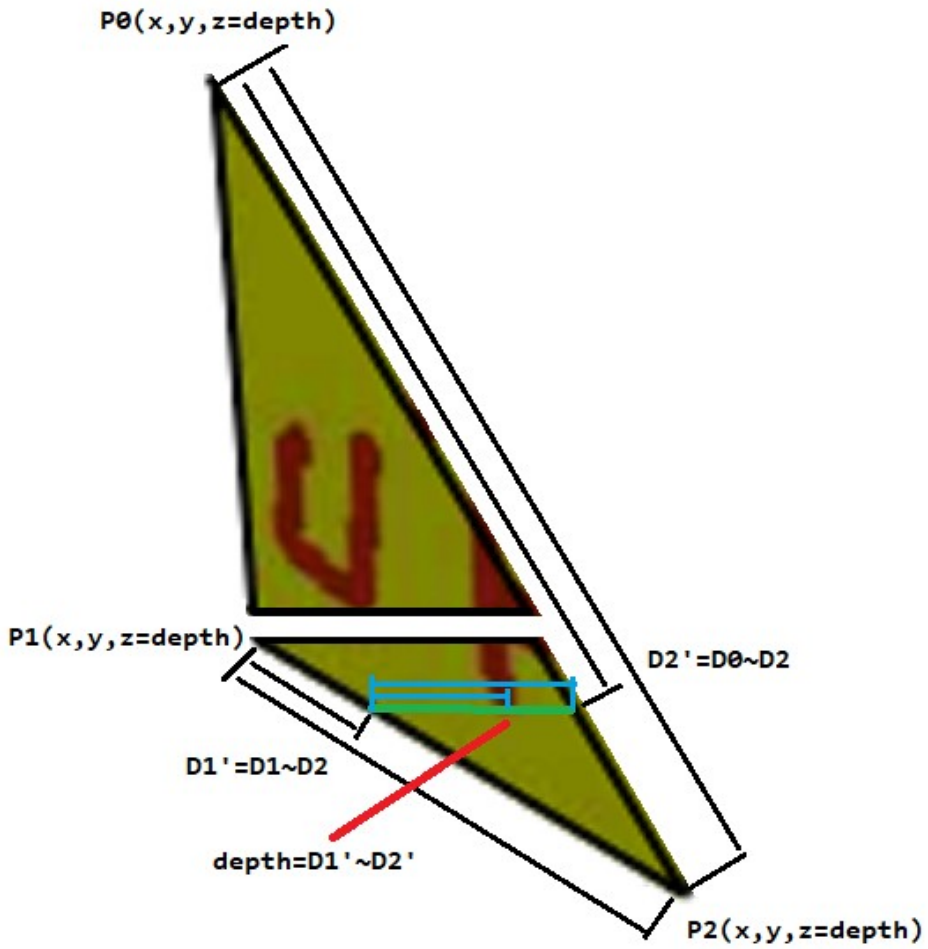
3D Graphics for Dummies

Rasterization: Depth



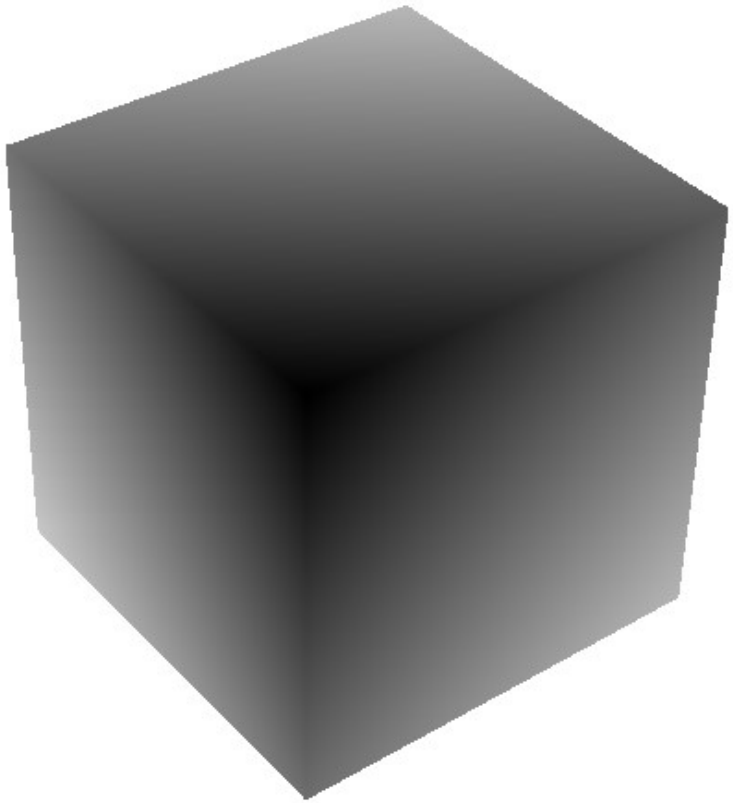
3D Graphics for Dummies

Rasterization: Depth



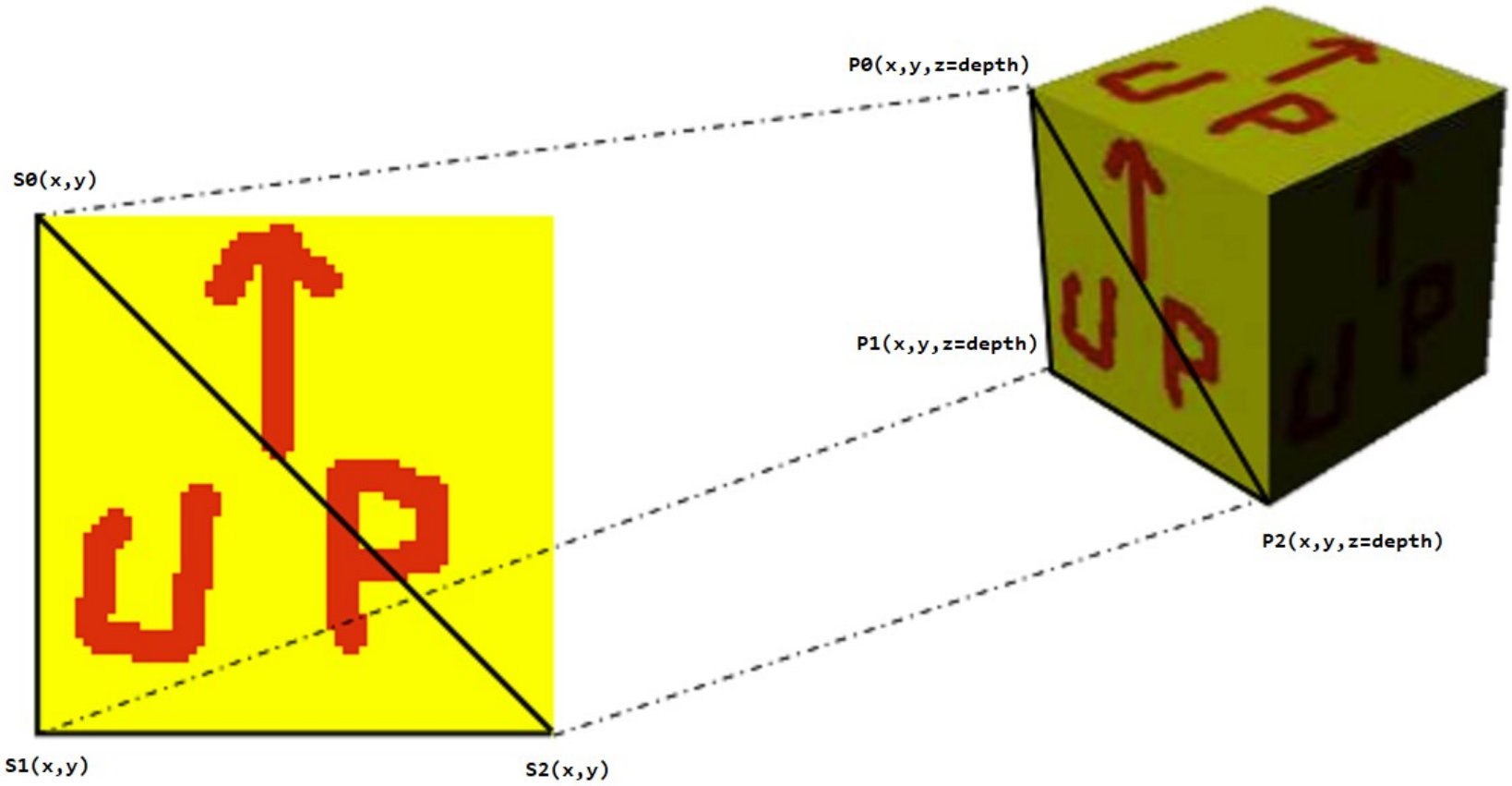
3D Graphics for Dummies

`Rasterization: Depth`



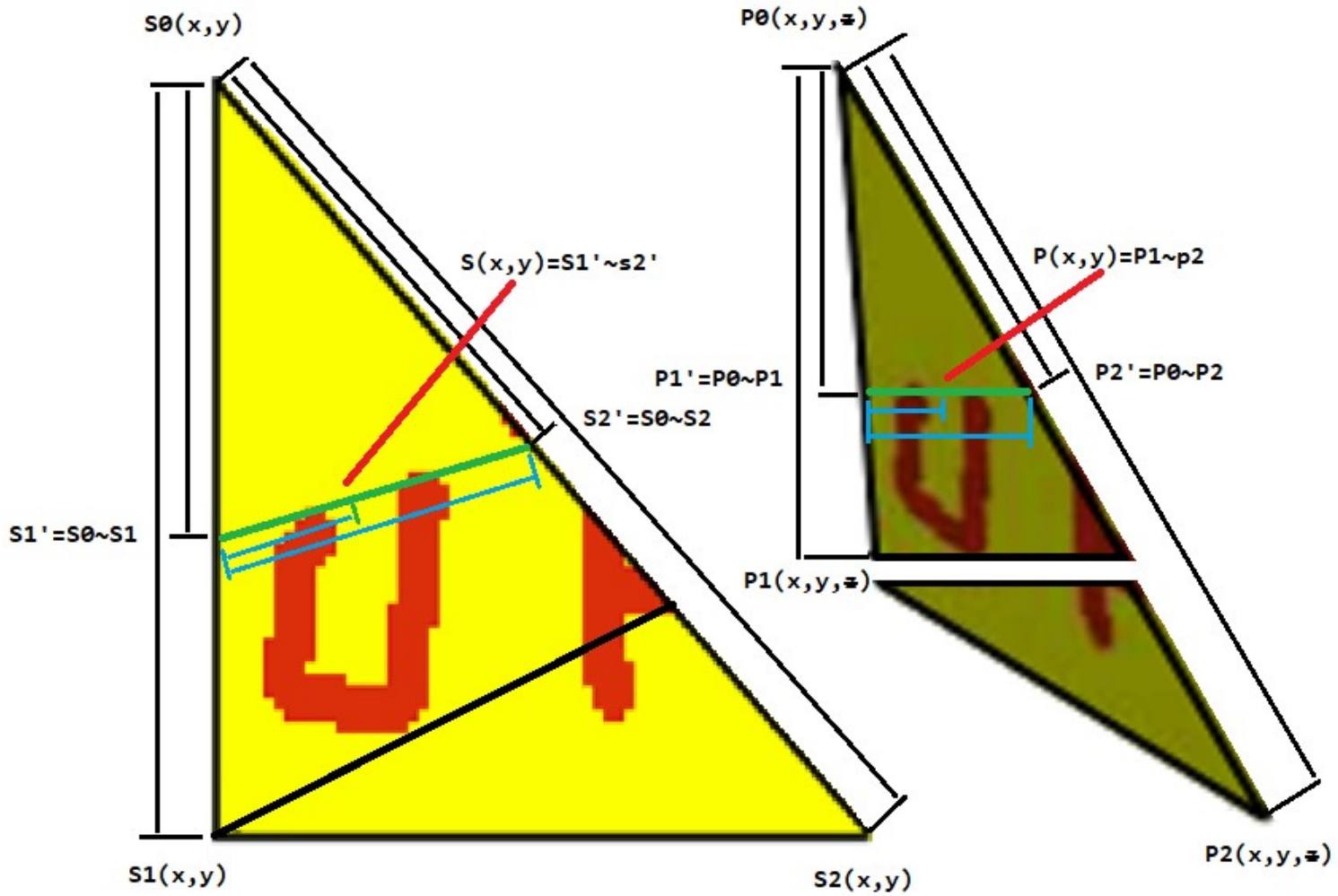
3D Graphics for Dummies

Depth / Texturing / Applying Surfaces



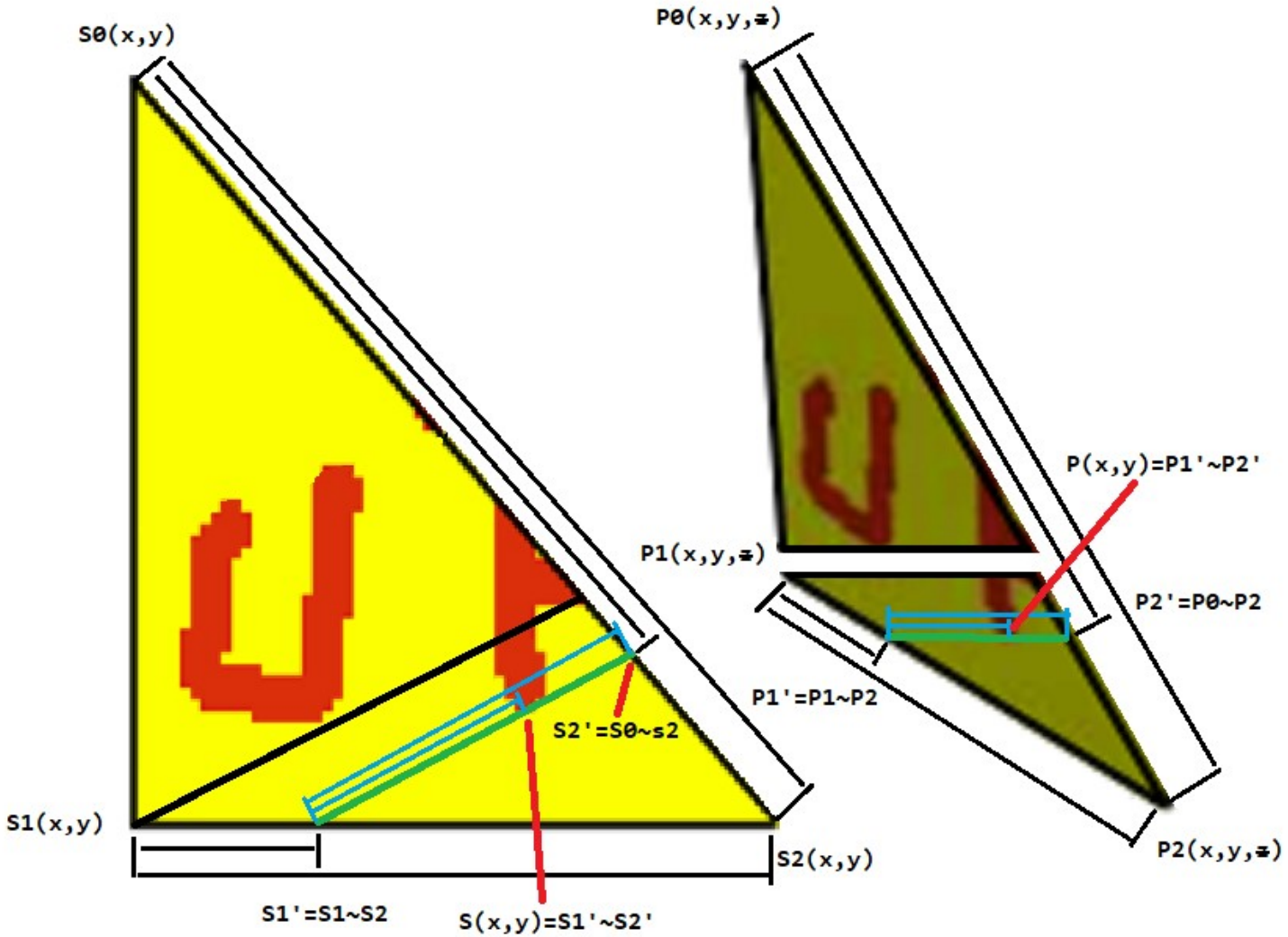
3D Graphics for Dummies

Rasterization: Texturing / Applying Surfaces



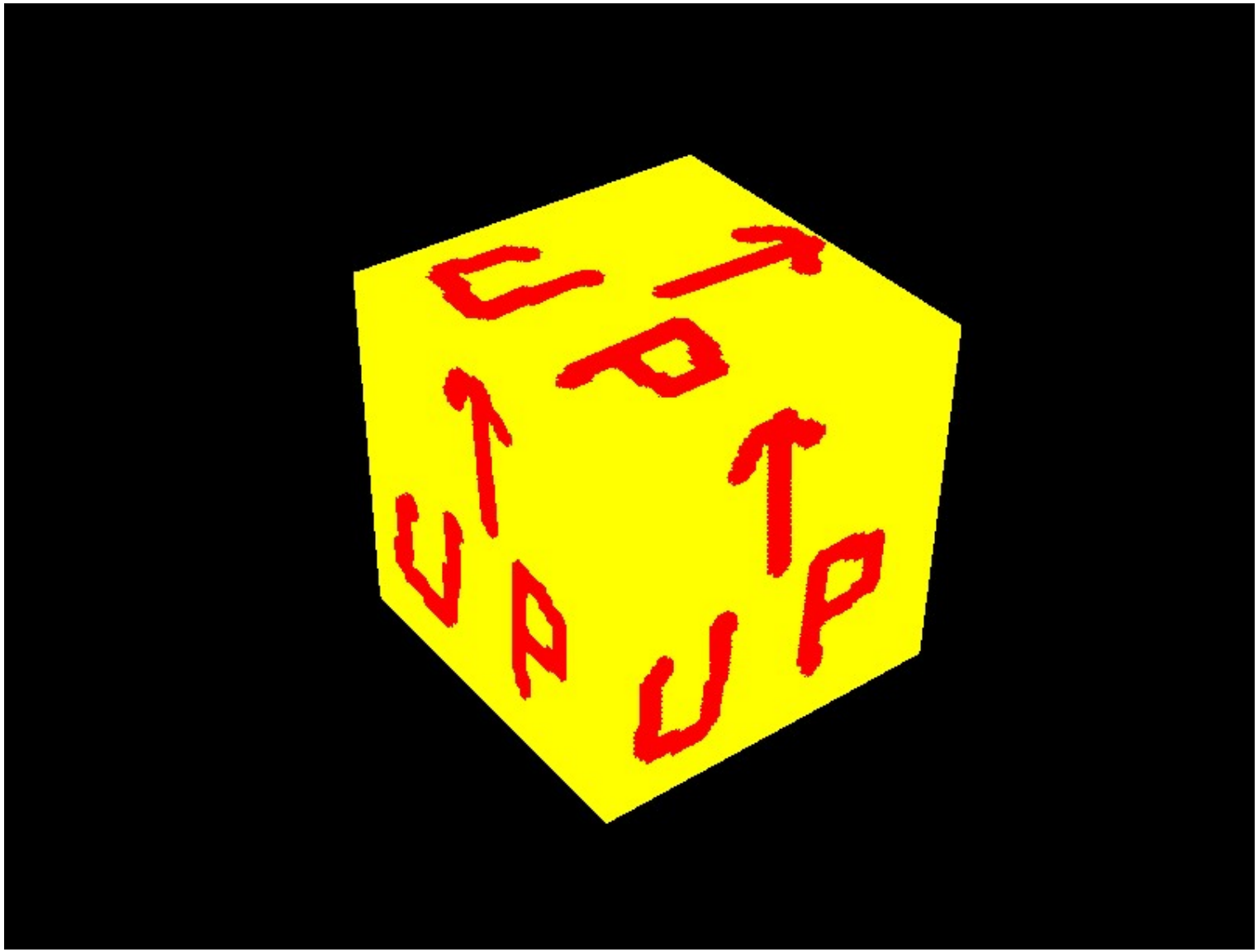
3D Graphics for Dummies

Rasterization: Texturing / Applying Surfaces



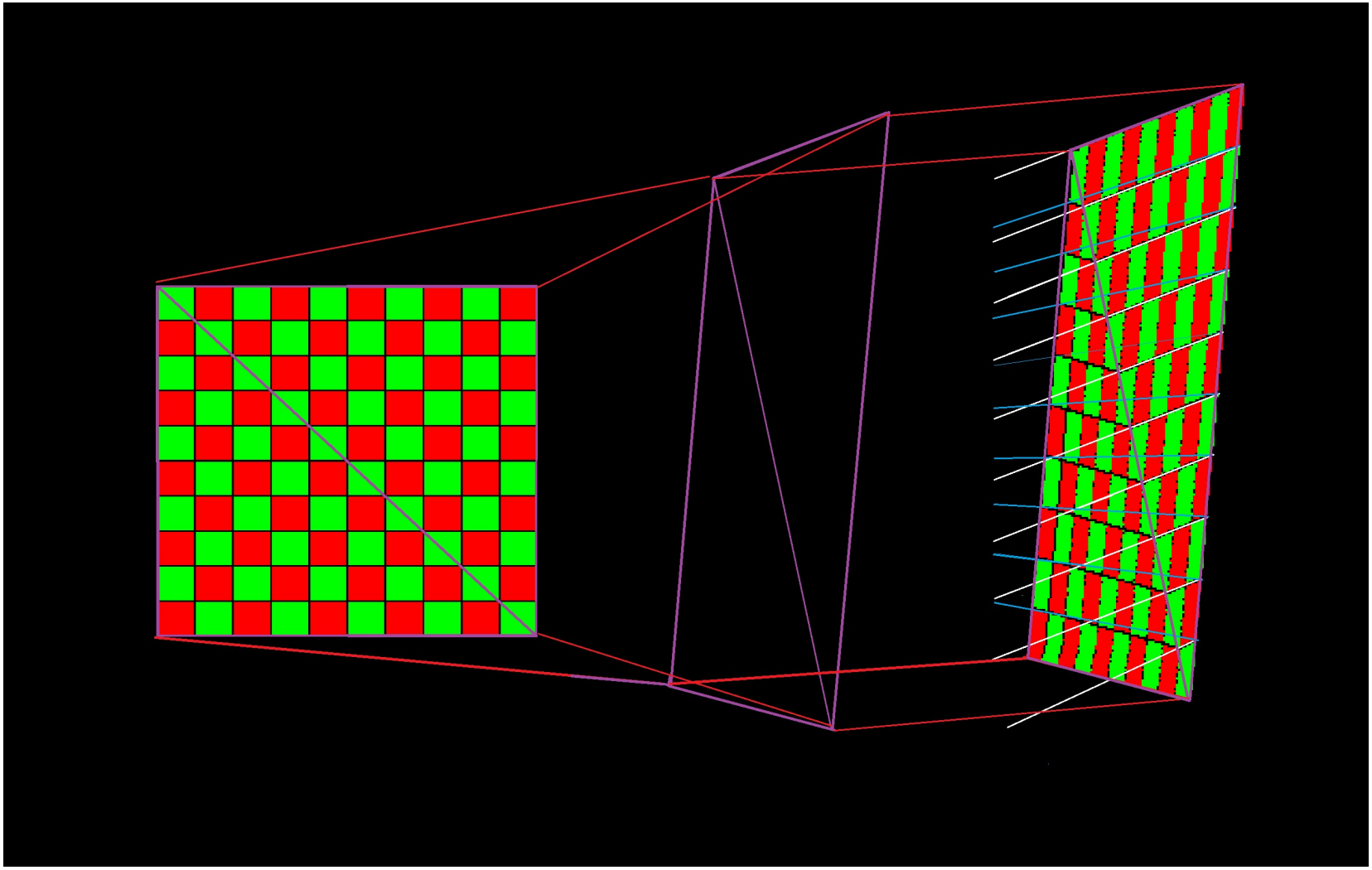
3D Graphics for Dummies

Rasterization: Texturing / Applying Surfaces



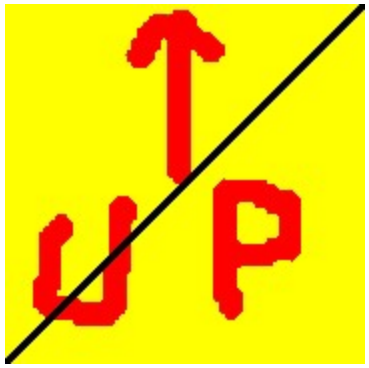
3D Graphics for Dummies

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



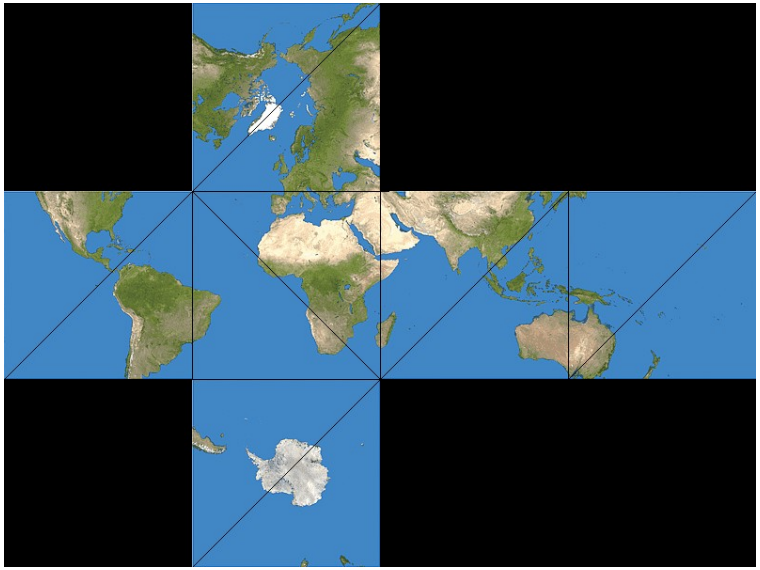
3D Graphics for Dummies

```
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

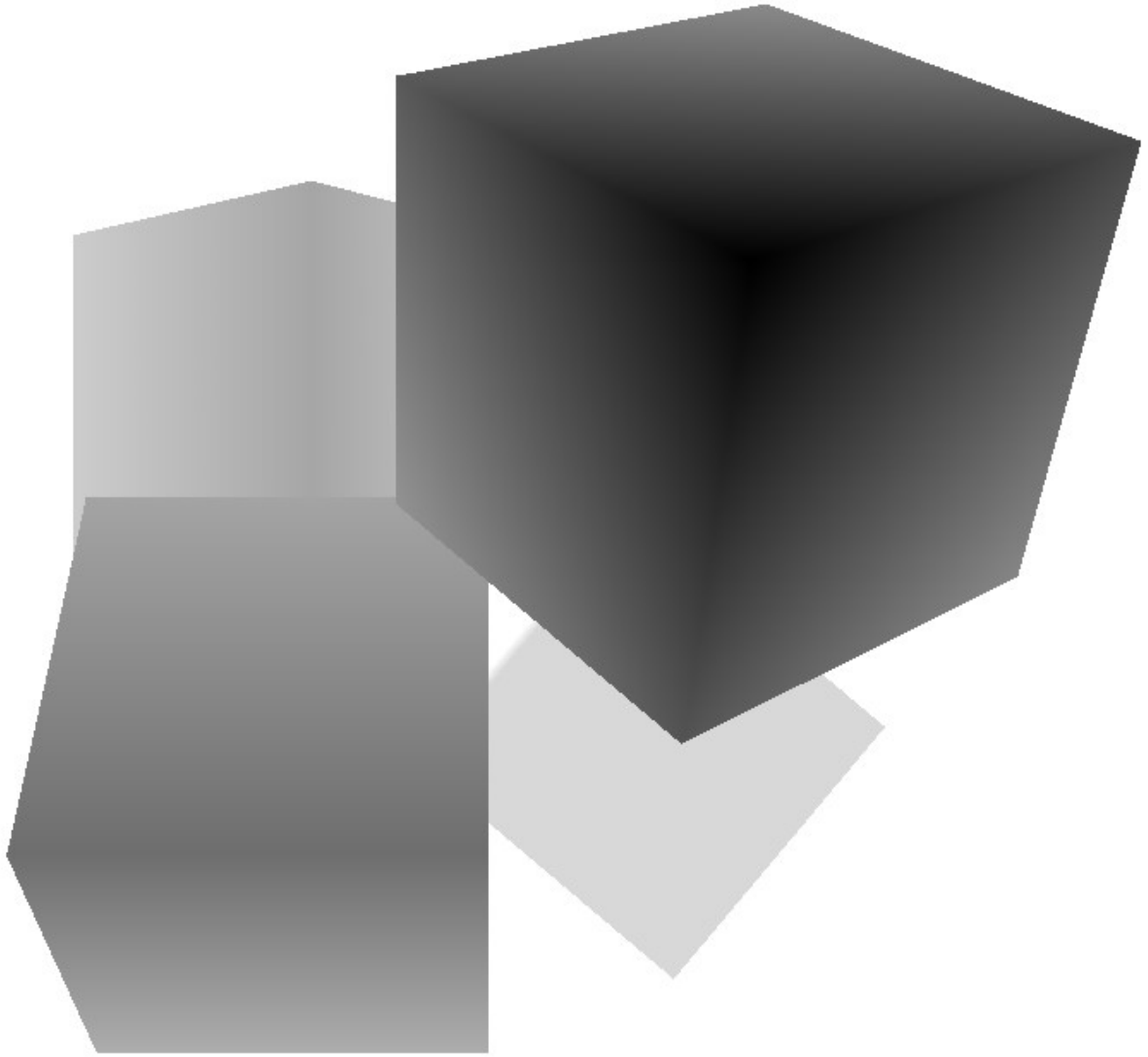
```
modelEarth =
{
    { {{  1,  1,  1 }, { -1,  1,  1 }, { -1,  1, -1 }}, IDB_EARTH, {{ 200, 400 }, { 200, 600 }, { 400, 600 }} },
    { {{ -1,  1, -1 }, {  1,  1, -1 }, {  1,  1,  1 }}, IDB_EARTH, {{ 400, 600 }, { 400, 400 }, { 200, 400 }} },

    { {{ -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 }} },
};
```



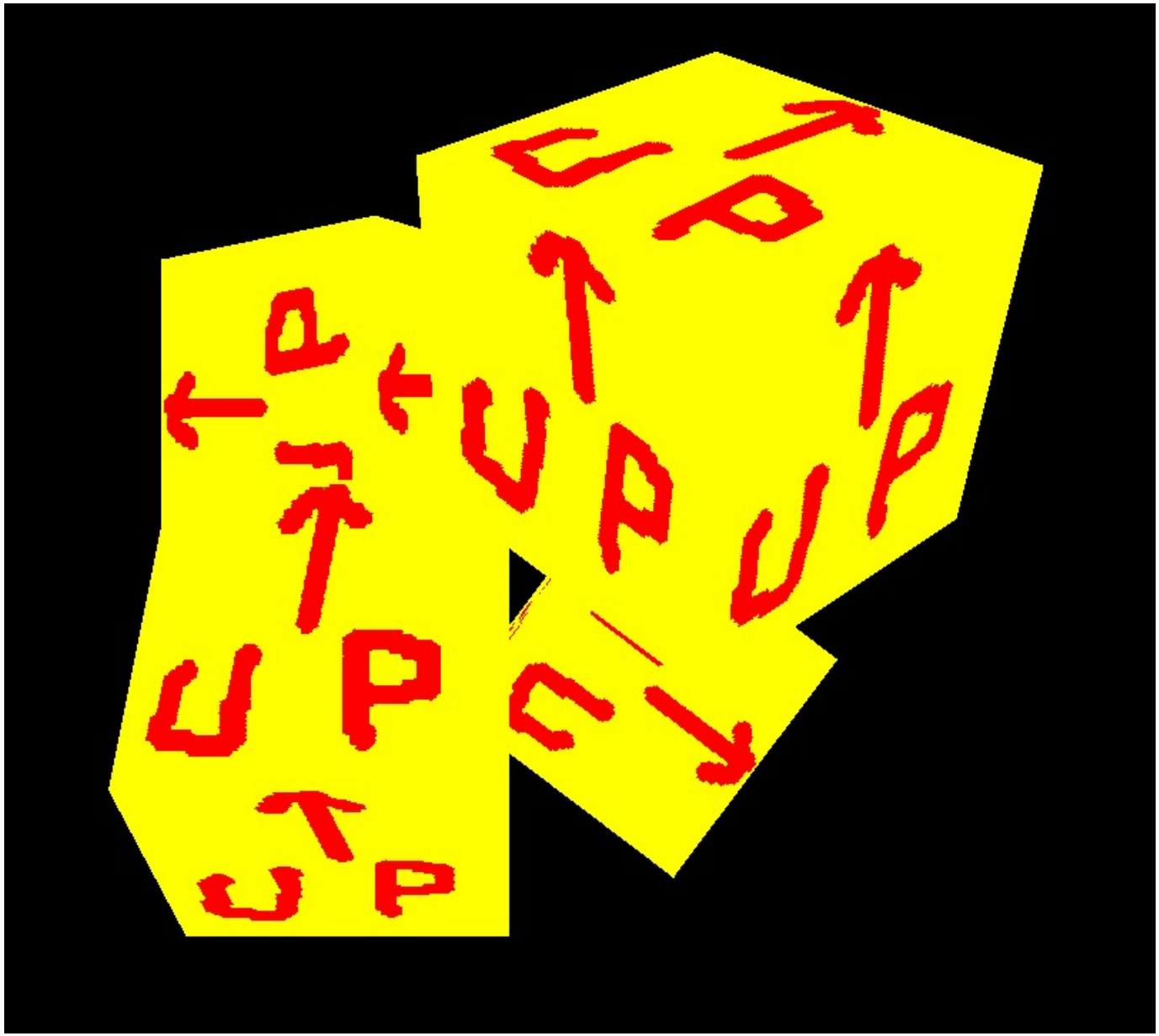
3D Graphics for Dummies

```
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;
}
```

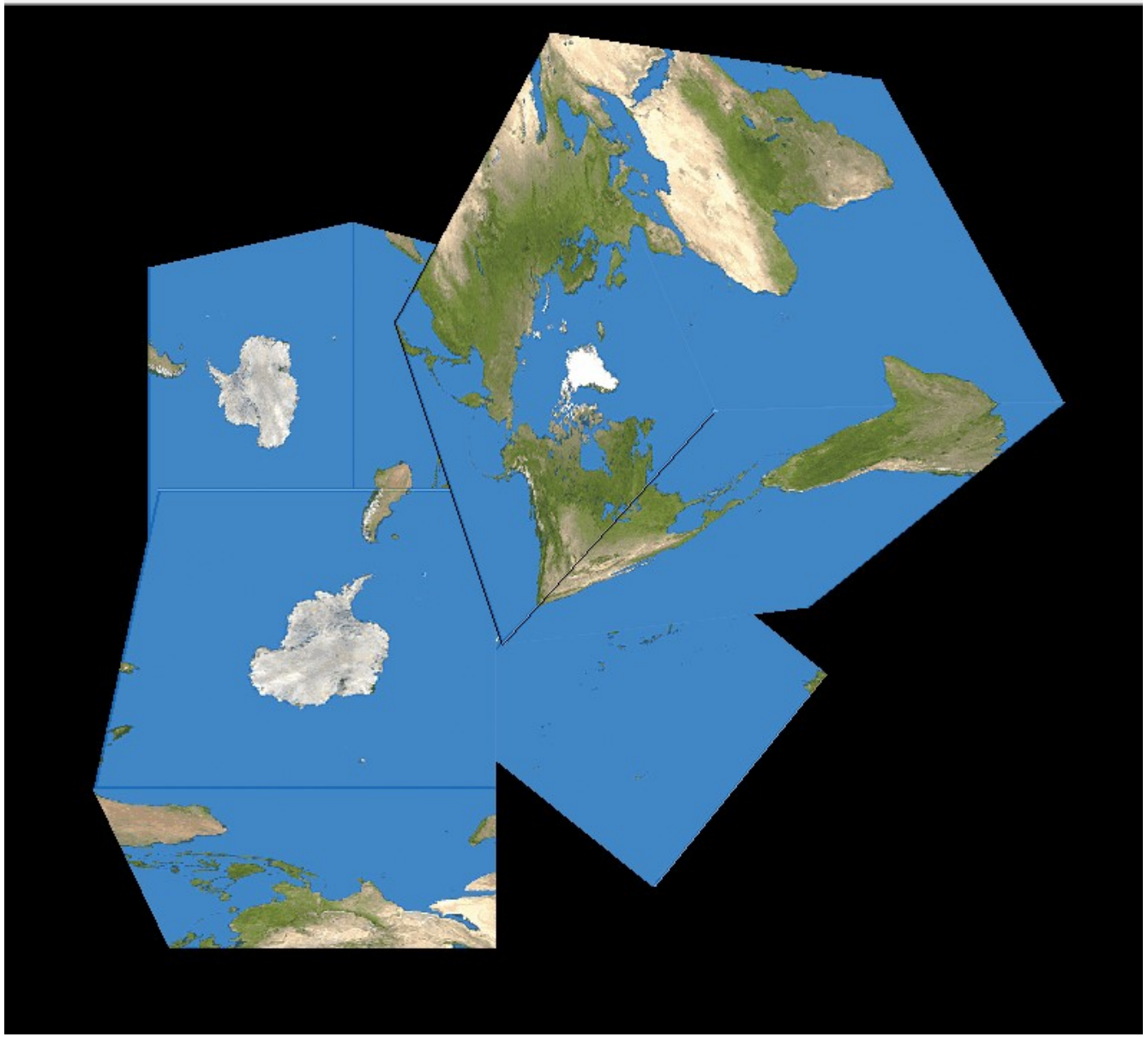
3D Graphics for Dummies



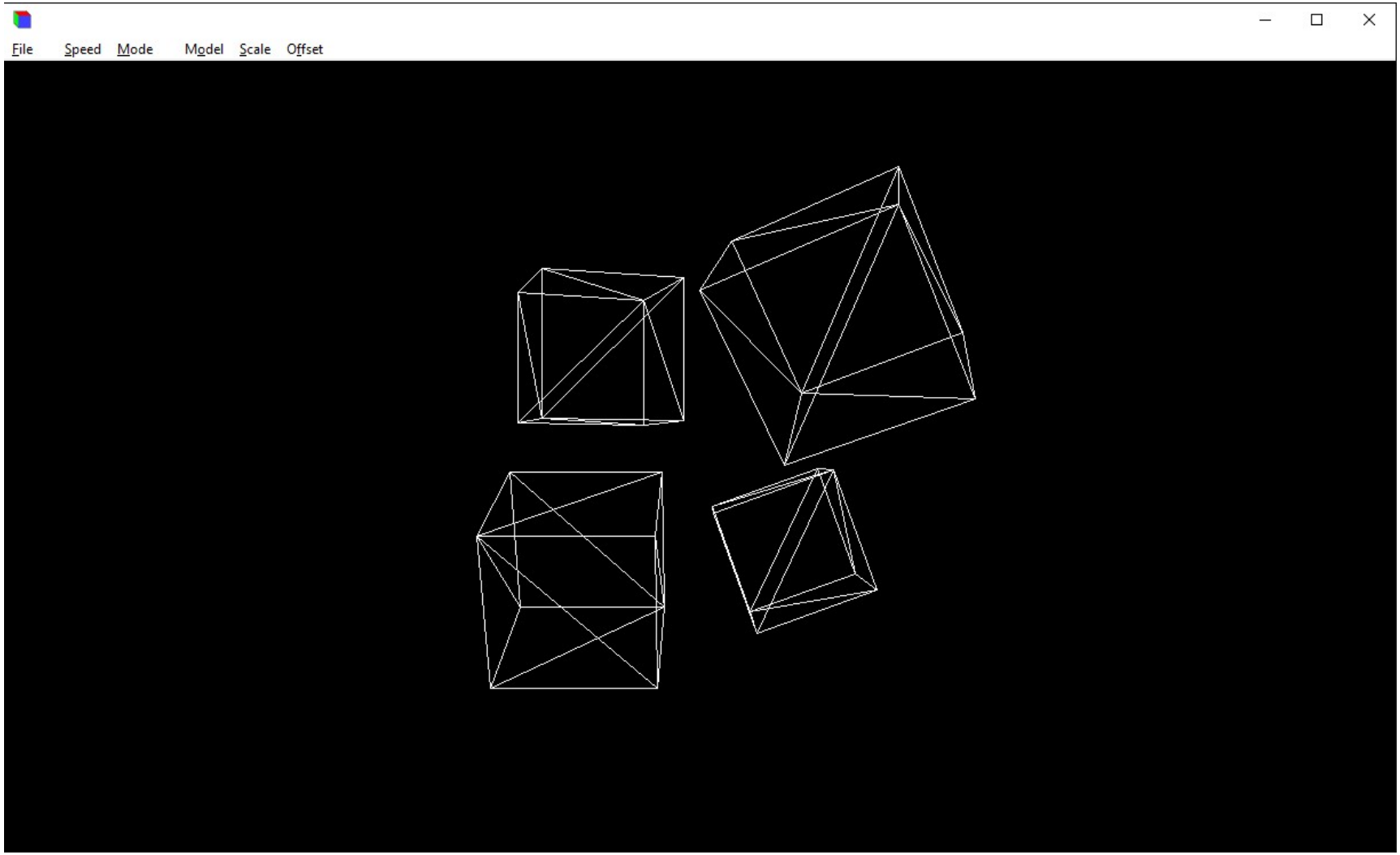
3D Graphics for Dummies



3D Graphics for Dummies

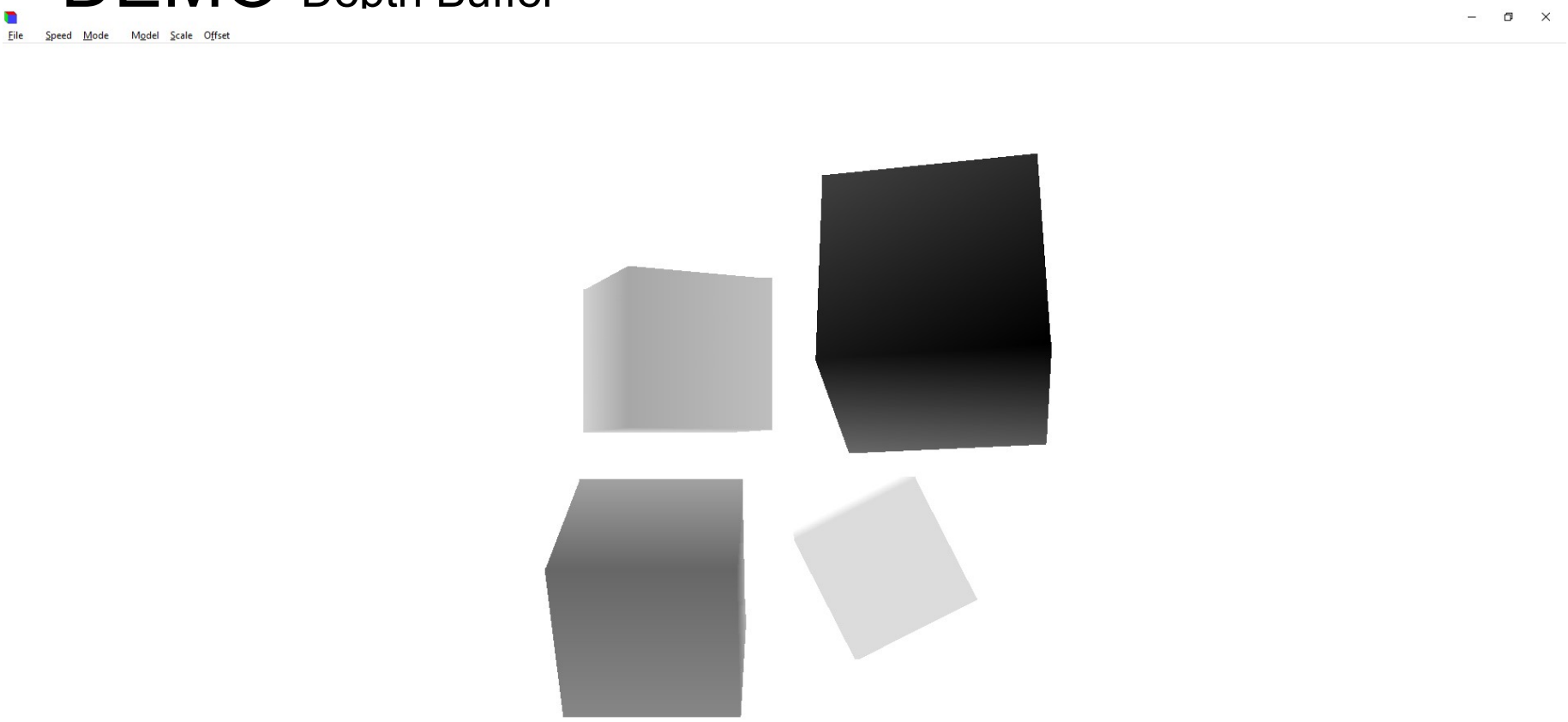


DEMO Wire Frame



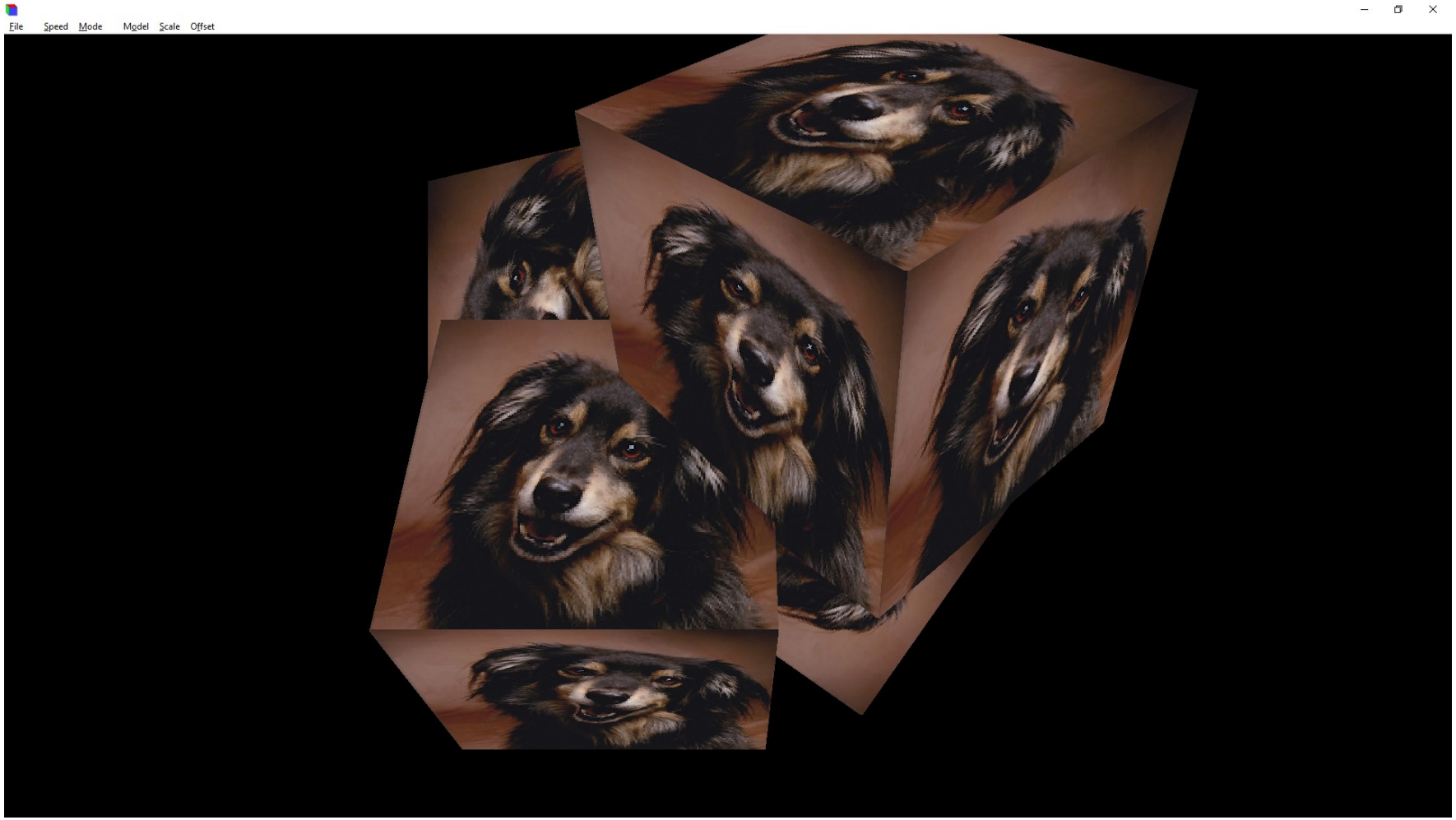
3D Graphics for Dummies

DEMO Depth Buffer



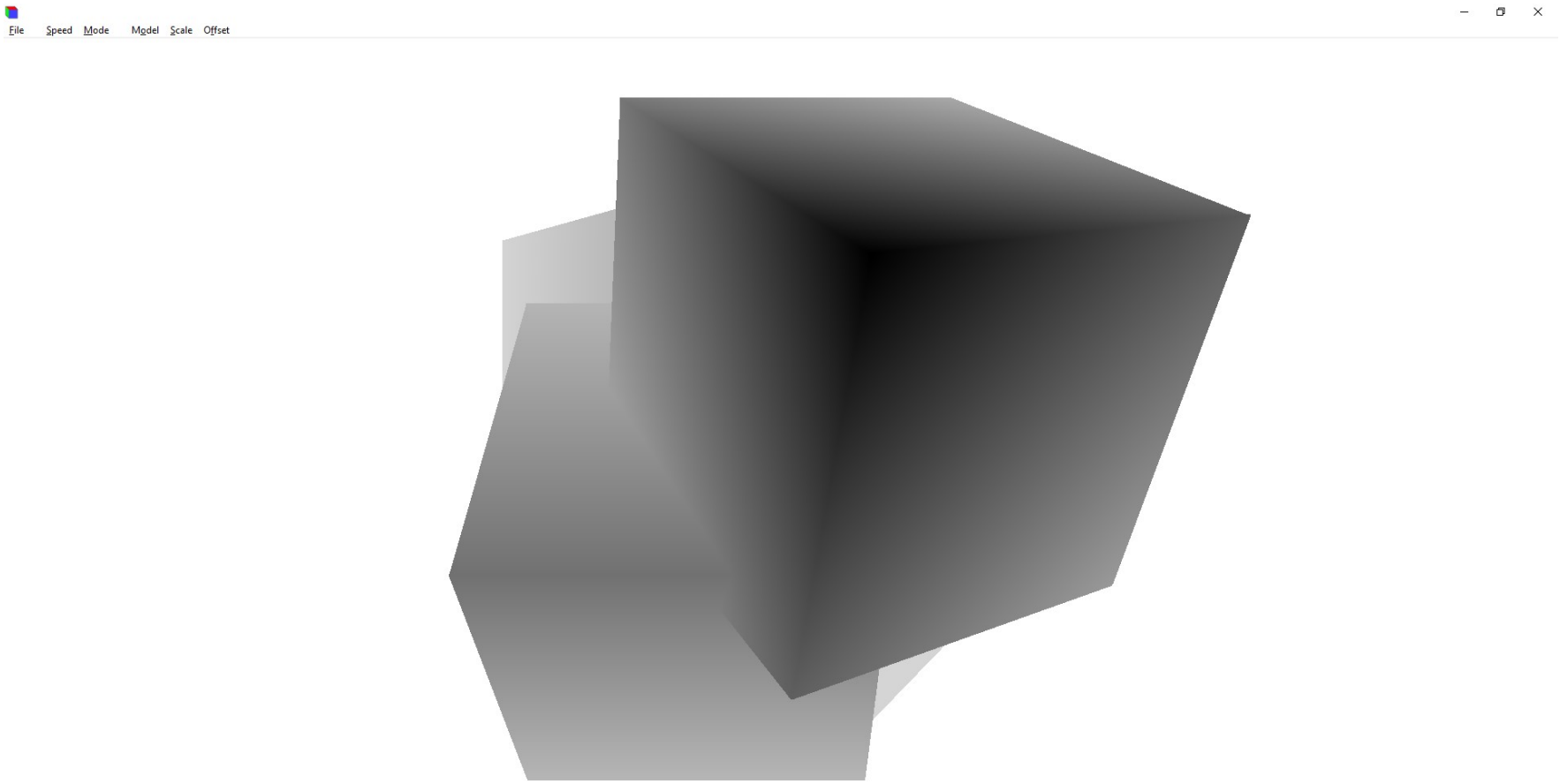
3D Graphics for Dummies

DEMO Model: “Frankie” large scale, with surface collision



3D Graphics for Dummies

DEMO Depth Buffer, large scale, with surface collision



Questions ?

Thank You