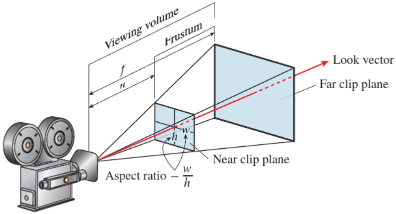
Algebra and perspective

Computer Graphics (DT3025)

Martin Magnusson
November 1, 2016



Hughes et al. 2013

Last time

- GPU programming
  - vertex shaders
  - fragment shaders
- Colour fundamentals
- Rasterised vector graphics

Vertex shader code

```
layout(location=0) in vec4 inPosition;
out vec3 myColour;

void main() {
  gl_Position = inPosition;
  myColour.rg = inPosition.xy;
  myColour.b = 1.0;
}
```

What does this vertex shader do?

1. Perspective correction
2. Apply gradient colour to vertices
3. All of the above

Fragment shader code

```
in vec3 myColour;
out vec4 pixel;

void main() {
  pixel.rgb = myColour;
  pixel.a = 0.0;
}
```

What does this fragment shader do?

1. Interpolate (linearly) between vertex colours
2. Pin all depth coordinates to zero
3. Pass through colours from vertex shader

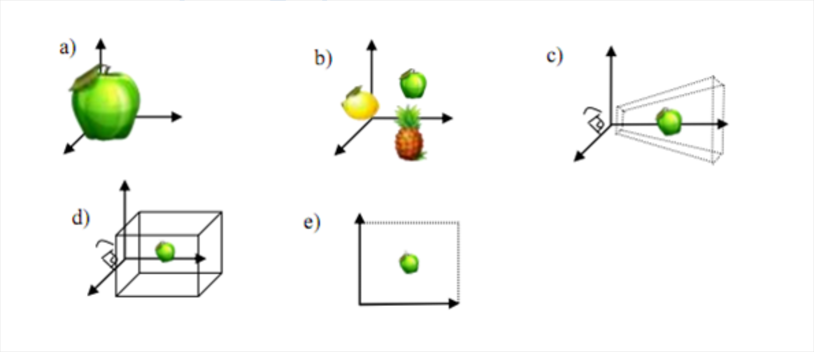
Today

- How to compute where an object ends up on the screen.
- How to check which objects occlude each other.

Reading material

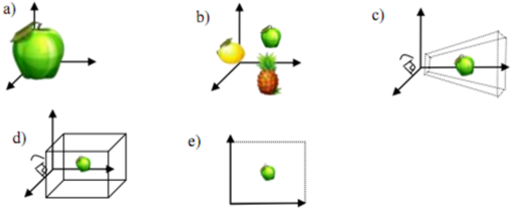
- Hughes et al.:
  - 7.1–7.6.6
  - 10 (mostly 10.6 and 10.13)
  - 11.1–11.2.1
  - 13

Spaces in computer graphics



- a) Object space (local coordinates, per model)
- b) World space (global coordinates, complete scene)
- c) Eye space (camera-local coordinates)
- d) Image space (perspective)
- e) Screen space (2D)

Transforming between spaces



1. object → world space: translate and rotate from world pose
2. world → eye space:
   - translate so that camera is at origin
   - rotate (around origin) so camera looks along $-z$ and y is up
3. eye → image space: perspective transform (scale by $1/z$)
4. image → screen space: remove z component

Vectors

3D vector $\mathbf{v} = \left[v_x, v_y, v_z\right]^{\mathrm{T}} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$

Norm $\|\mathbf{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$

Addition $\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_x + v_x \\ u_y + v_y \\ u_z + v_z \end{bmatrix}$

Scalar (dot) product $\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z = \|\mathbf{u}\| \cdot \|\mathbf{v}\| \cdot \cos\theta$

Vector (cross) product $\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix} = \text{scaled normal}$

Matrices

Matrix $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{13} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$

Multiplication $\mathbf{AB} = \mathbf{C}$, with $c_{ik} = a_{i1}b_{1k} + a_{i2}b_{2k} + \cdots + a_{im}b_{mk}$

Transpose $\mathbf{A}^{\mathrm{T}} = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{31} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$

Inverse $\mathbf{A}^{-1}$, such that $\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Homogenous coordinates

- Add an extra (fourth) element $[x, y, z, w]^{\mathrm{T}}$
- Represents the point $[x/w, y/w, z/w]^{\mathrm{T}}$
- Typically: $[x, y, z, 1]^{\mathrm{T}}$

Vectors vs. points

- So, our 3D world is the slice of the 4D space where $w = 1$.
- Points in space have $w = 1$.
- Vectors have $w = 0$.
- Why?
  - Vectors have no “place” in space, but points do.
  - We can add vector + vector (= a new vector with $w = 0$), and vector + point (= a new point with $w = 1$)
  - but we can’t add two points. (“This corner plus that corner” doesn’t mean anything.)

Transformations

- How do we express transformations on vectors in 2D and 3D space?
- Matrix/vector multiplication is convenient.
- But we could also *add* vectors, use quaternion algebra, etc.

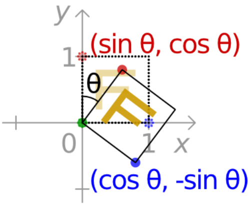
Scaling

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ 1 \end{bmatrix}$$

Rotation

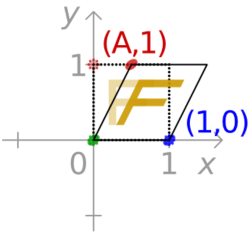
Rotate by angle θ:

- "around z axis,"
- equivalently: "in xy plane."

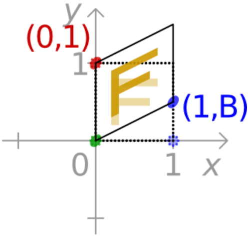


$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \\ 1 \end{bmatrix}$$

Shear



2D shear along x



2D shear along y

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + az \\ y + bz \\ z \\ 1 \end{bmatrix}$$

Translation (moving)

- In Cartesian coordinates, no matrix exists that can do translation.
- We'd need to *add a vector*, not *multiply by matrix*.

$$\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \end{bmatrix}$$

- Can we make it fit our matrix multiplication framework anyway?
- Cue: homogeneous coordinates.

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

Perspective

If z is "out of the screen" in eye space (so $-z$ is "into the image"), and the image plane is at $z = d$:

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ dz \\ z \end{bmatrix} = \begin{bmatrix} dx/z \\ dy/z \\ d \\ 1 \end{bmatrix}$$

homogenisation

If $d = 1$:

$$\cdots = \begin{bmatrix} x/z \\ y/z \\ 1 \\ 1 \end{bmatrix}$$

Combinations

- Transformations can be combined using matrix multiplication.
- *Order is important* (not commutative).

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{translate}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{rotate around } x} \underbrace{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}_{\text{point}} = \underbrace{\begin{bmatrix} xs_x + t_x \\ ys_y\cos\theta - zs_z\sin\theta + t_y \\ ys_y\sin\theta + zs_z\cos\theta + t_z \\ 1 \end{bmatrix}}_{\text{transformed point}}$$

Rotation matrices

- A 3×3 matrix (or homogeneous 4×4) can represent all possible rotations — but all matrices are not rotations!
- Requirements:
  - square
  - $|\mathbf{R}| = +1$
  - orthogonal: $\mathbf{R}^{\mathrm{T}} = \mathbf{R}^{-1}$
- In other words:
  - $\mathbf{R}$ is normalized: the squares of the elements in any row or column sum to 1.
  - $\mathbf{R}$ is orthogonal: the dot product of any pair of rows or any pair of columns is 0.
- The *rows* of $\mathbf{R}$ represent the axes in the *original* space of unit vectors along the axes of the *rotated* space.
- The *columns* of $\mathbf{R}$ represent the axes in the *rotated* space of unit vectors along the axes of the *original* space.

Orthonormalization

- What to do if rotation matrix is not orthogonal and with determinant 1?
- If we know that it should be a rotation matrix (only), we can “massage” it into being orthonormalised again.

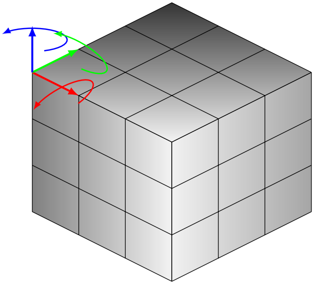
1. Normalise first row (or column).
2. Cross product of first and second row, normalise, use result as third row.
3. Cross product of first and third row, use result as second row.

- May not be “the correct” rotation anymore, but at least it will be a rotation.

Euler angles (fixed angles)

$$(\theta_x, \theta_y, \theta_z)$$

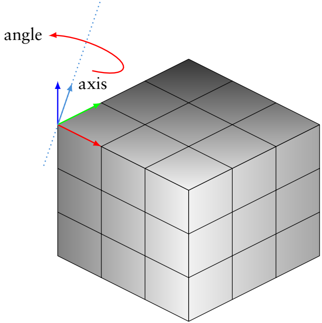
- Rotation orders: (x, y, z), (z, y, x), (x, y, x), etc.
- Problem: gimbal lock.
- Problem: interpolation $\rightarrow$ "detour".



Axis and angle

$$([x, y, z], \theta)$$

- Easy to read
- No gimbal lock
- Hard to concatenate
- Non-trivial to interpolate



Quaternions

$$[s, x, y, z]$$

- (Unit-length) quaternions : $\mathbf{q} = [s, x, y, z]$
- Generalisation of complex numbers
- NB: *not* homogeneous coordinates.
- NB: *not* axis+angle.
- "Vector part" (imaginary) is $\sin(\theta/2) \cdot \text{axis}$
- "Scalar part" (real) is $s = \cos(\theta/2)$
- No gimbal lock, easy to combine, easy to interpolate

Vector products

What does the dot product between two vectors represent?

1. The cosine of the angle between the vectors.
2. A third vector, perpendicular to the two.
3. None of the above.

Link

Matrix inverse

What is the inverse of

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

1. $\mathbf{A}^{-1}$ does not exist
2. $\mathbf{A}^{-1}$ is the identity matrix
3. $\mathbf{A}^{-1} = \mathbf{A}^{\mathrm{T}}$

Link

Transformations

For a point $\mathbf{x} = (10, 0)$ how to rotate it 5 degrees around $(8, 2)$?

- T_1: translate $(8, 2)$
- T_2: translate $(-2, 2)$
- $\mathbf{R}$: rotate 5 degrees

1. $\mathbf{x}' = \mathbf{T}_1^{-1}\mathbf{R}\mathbf{T}_1\mathbf{x}$
2. $\mathbf{x}' = \mathbf{T}_1\mathbf{R}\mathbf{T}_1^{-1}\mathbf{x}$
3. $\mathbf{x}' = \mathbf{T}_2^{-1}\mathbf{R}\mathbf{T}_2\mathbf{x}$
4. $\mathbf{x}' = \mathbf{T}_2\mathbf{R}\mathbf{T}_2^{-1}\mathbf{x}$
5. $\mathbf{x}' = -\mathbf{T}_1\mathbf{R}\mathbf{x}$

Comb.Trans. | Trans.

Rotation matrices

- Requirements for a rotation matrix:
  - square
  - $|\mathbf{R}| = +1$
  - orthogonal: $\mathbf{R}^{\mathrm{T}} = \mathbf{R}^{-1}$

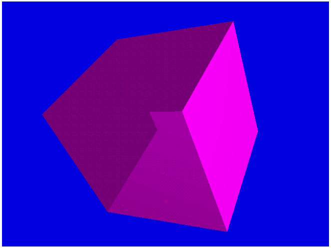
What happens if the determinant $|\mathbf{R}| = -1$?

1. $\mathbf{R}$ is a reflection
2. $\mathbf{R}$ is a rotation with a negative angle
3. $\mathbf{R}$ does not exist

Link

Handling occlusion

- How do we compute which objects should be visible?



Mathias Broxvall

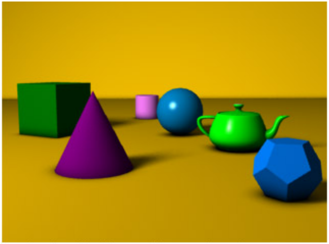
Painter's algorithm

1. Sort primitives by distance to camera
2. Draw most distant primitives first
3. Paint nearby objects on top of old ones

+ Simple to implement
– Expensive to sort all objects
– Expensive to draw pixels that will be overwritten
– Still cannot handle all scenes

z-buffer

- In addition to framebuffer, use a z-buffer: an array with a z value for every *pixel*.
- Only update pixel if new z is closer than the buffer's value.



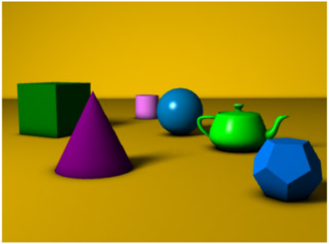
A simple three dimensional scene

Z-buffer representation

Wikimedia Commons

z-buffer outline

1. Reset values in z-buffer to infinity.
2. Draw objects *in any order*.
3. When drawing, compute z value for every pixel.
4. Only update buffer (colour and z-buffer) if new z value is smaller than old value.



A simple three dimensional scene

Z-buffer representation

Wikimedia Commons

z-buffer pros and cons

Disadvantages:

- Memory usage
- Have to compute z value for all pixels
- Precision problem when many objects compete for same pixel (e.g., at edges)

Advantages:

+ Simple algorithm
+ Efficient hardware implementations
+ Works for all *nontransparent* objects / scenes

What about transparent objects, then?

- A-buffer: “the anti-aliased, area-averaged, accumulation buffer” (Carpenter 1984)
- Instead of storing single z value, build a *linked list* for each pixel.
- Fragment shader “draws” all pixels, adding to the list.
- Keep list sorted on depth.
- Post processing: traverse list, compute final colour with blending.

Carpenter's A-Buffers

Clipping

- During *primitive assembly* the vertices are *clipped* to fit a volume

$$-c \leq x \leq c$$
$$-c \leq y \leq c$$
$$-c \leq z \leq c$$

- Why clip on z axis?
  - Don't draw objects behind the camera
  - Avoid numerical problems (div by zero)
  - Avoid drawing the whole world to infinity

Transformations in OpenGL

- Compute final transformation matrix as *product* of a sequence of *primitive* transformations.
- Pass to vertex shader (as a `uniform`) variable.
- Vertex shader (typically) multiplies each vertex position with this matrix.
- In legacy OpenGL, we could use `gluPerspective` to automatically set up projection matrix.
- Since OpenGL 3.3, we need to set it up ourselves and pass it as input to the vertex shader. (E g, the GLM or glhlib projects)
- (See Lab 1.6!)

Projection caveat

- Applying the projection matrix from before works, but…
- after flattening the scene onto the $z = 1$ plane, we lost all depth info.

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

- We will need the distance for z-buffering (to determine what is in front of what).

Making a view volume

- Instead of flattening onto $z = 1$ *plane*, make view frustum *box* instead.
- We want to map the full range of z values to $[-1, +1]$ and keep *relative distances*.
- $\text{pseudo}(z) = A + B/z$
- Choose A, B so that clipping planes are at $+1$ and -1.
- (Near clipping plane: $z = +1$, far clipping plane: $z = -1$.)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ Az + B \\ z \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ A + B/z \\ 1 \end{bmatrix}$$

Accounting for aspect ratio

- One more thing: we also need to account for non-square view volumes.
- If window aspect ratio is x/y (width/height), we'll need to scale x with y/x.

$$\begin{bmatrix} H/W & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

(Where H/W would be $9/16$ for a normal 16:9 display.)

Transforming z to fit [-1,+1]

- Near clipping plane (e g, $n = -1$) should map to $+1$.
- Far clipping plane (e g, $f = -10$) should map to -1.

$$\begin{aligned} 1 &= A - B/n \\ -1 &= A - B/f \end{aligned}$$

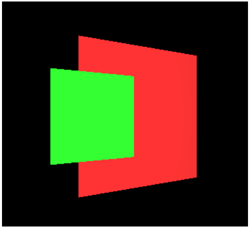
- Solve for A, B:

$$\begin{aligned} A &= -\frac{f+n}{f-n} \\ B &= -\frac{2fn}{f-n} \end{aligned}$$

Putting it all together

1. Set up your model matrix (move/rotate object to where it is in the world).
2. Set up your view matrix (move/rotate world to camera's point of view).
3. Set up your projection matrix (as in the previous slide).
4. Combine the three to a MVP (modelViewProjection) matrix.
5. Pass the matrix to the vertex shader (with a `uniform`).

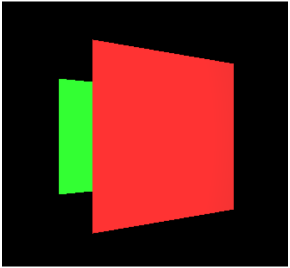
```
layout(location=0) in vec4 inPosition;
uniform mat4 projectionMatrix;

void main() {
  gl_Position = projectionMatrix * inPosition;
}
```



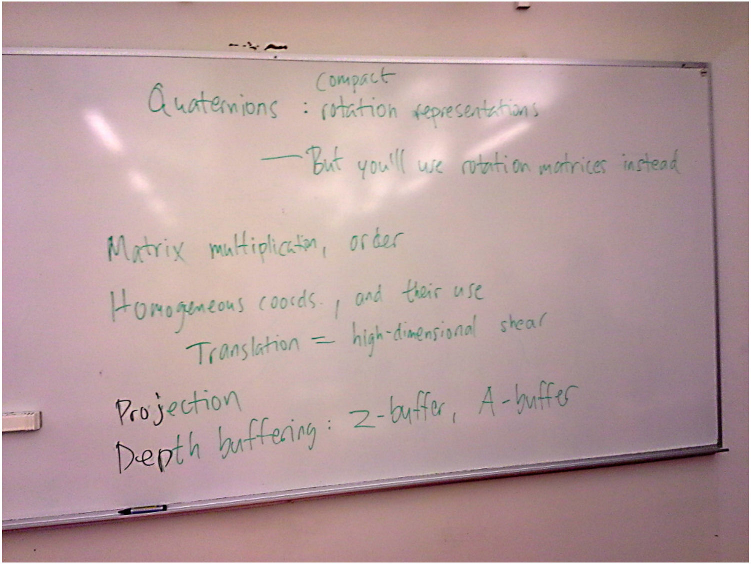
Using z-buffering in GL

- Enable z-buffer (depth test), and specify sorting (keep smaller or larger z values).

```
glEnable(GL_DEPTH_TEST); // happens after fragment shader
glDepthFunc(GL_LESS);
```



Summary


Quaternions : compact rotation representations
— But you'll use rotation matrices instead
Matrix multiplication, order
Homogeneous coords., and their use
Translation = high-dimensional shear
Projection
Depth buffering : z-buffer, A-buffer

What's next

Next lecture: lighting and materials

- Mon Nov 7, 13.15–15.00
- T-141
- Hughes et al.:
  - 6.2.2–6.3, 6.5,
  - 14.9,
  - 1.13.1–2.
  - (Chapter 27 is great, but perhaps overly detailed. I recommend to look through it, but never mind solid angles and integrals for now. The chapters listed above are more to the point.)

What's next

Next next lecture: textures

- Tue Nov 8, 14.15–17.00
- T-211
- Hughes et al.:
  - 7.9–7.9.1,
  - 9.6,
  - 20.1–20.8.2.

Study questions (for lecture #3)

1. The irradiance (incoming light energy per area) at a surface patch is proportional to the cosine of the angle of the incoming light. Why?
2. How many dimensions (input and output values) does a BRDF function have?
3. How many dimensions does a BSDF function have?
4. Search for the most *Lambertian* surface you can see (if there is one).
5. Search for the surface with the highest *specular* exponent (if there is one).
6. Search for the surface with the highest *ambient* component (if there is one).

References

- Loren Carpenter (1984). “The A-buffer, an Antialiased Hidden Surface Method”. In: 18.3, pp. 103–108.
- John F. Hughes et al. (2013). *Computer graphics: principles and practice (3rd ed.)* Boston, MA, USA: Addison-Wesley Professional, p. 1264. ISBN: 0321399528.