3D object representation

Most common for real-time:

Polygon surfaces

- **Supported by graphics accelerators**
- **Easy to handle**
- **Not very space efficient**
- **Often triangle-based**

How do we store that?

First try: A simple format

Vertex = (x, y, z)

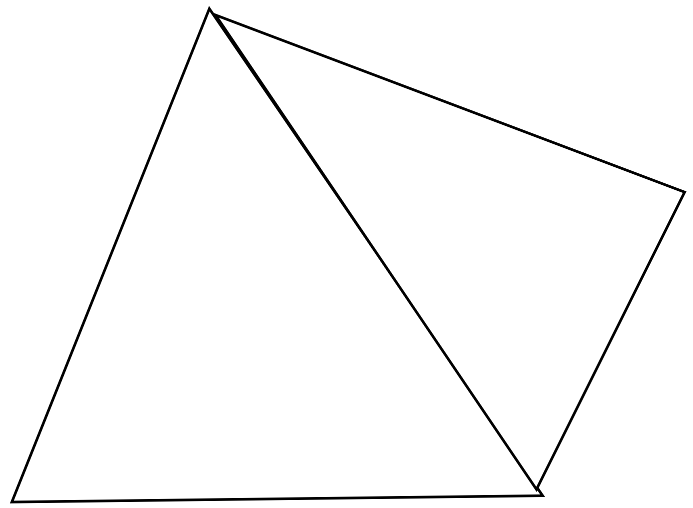
Triangle = array of Vertex

3DObject = array of Triangle

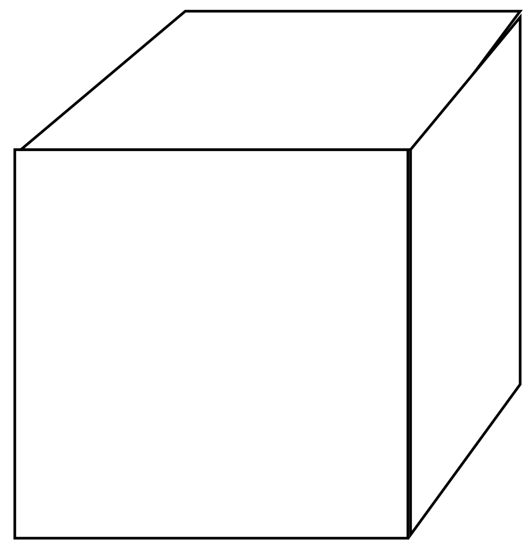
Example

Triangle[1]: (10, 10, 10), (10, 20, 10) (10, 20, 20)
Triangle[2]: (10, 10, 10), (10, 20, 10), (10, 10, 20)
Triangle[3]: (. . .), (. . .), (. . .)
. . .

...but why is this not quite good?



Pyramid:
4 triangles
4 corners
12 vertices!



Cube:
6 squares or 12 triangles
8 corners
24 or 36 vertices!

Several times too many vertices to process!

A better format

Vertex = (x, y, z)

Vertex table = array of Vertex

Triangle = array of integers

Triangle table = array of Triangles

3DObject = Vertex table + Triangle table

Example

Vertex table:

(10, 10, 10)
(10, 20, 10)
(10, 20, 20)
(10, 10, 20)
. . .

All vertices in the object in one list

Triangle table:

(1, 2, 3)
(1, 2, 4)
. . .

Indexes to the vertex table

Modelling in OpenGL

GLfloat vertices[] ={{-1,-1,-1}, {1,-1,-1}, {1,1,-1}, {-1,1,-1},{-1,-1,1}, {1,-1,1}, {1,1,1}, {-1,1,1}};

GLubyte cubeIndices[24] ={0,3,2,1, 2,3,7,6, 0,4,7,3, 1,2,6,5, 4,5,6,7, 0,1,5,4};

glDrawElements can draw the entire shape with one call (almost)!

Models on disk

Wavefront .obj format. Simple, text-based mesh format. Example: A cube:

```
# Exported from Wings 3D 0.98.26b
mtllib cube.mtl
o cube1
#8 vertices, 6 faces
v -1.00000000 -1.00000000 1.00000000
v -1.00000000 1.00000000 1.00000000
v 1.00000000 1.00000000 1.00000000
v 1.00000000 -1.00000000 1.00000000
v -1.00000000 -1.00000000 -1.00000000
v -1.00000000 1.00000000 -1.00000000
v 1.00000000 1.00000000 -1.00000000
v 1.00000000 -1.00000000 -1.00000000
```

Vertex list

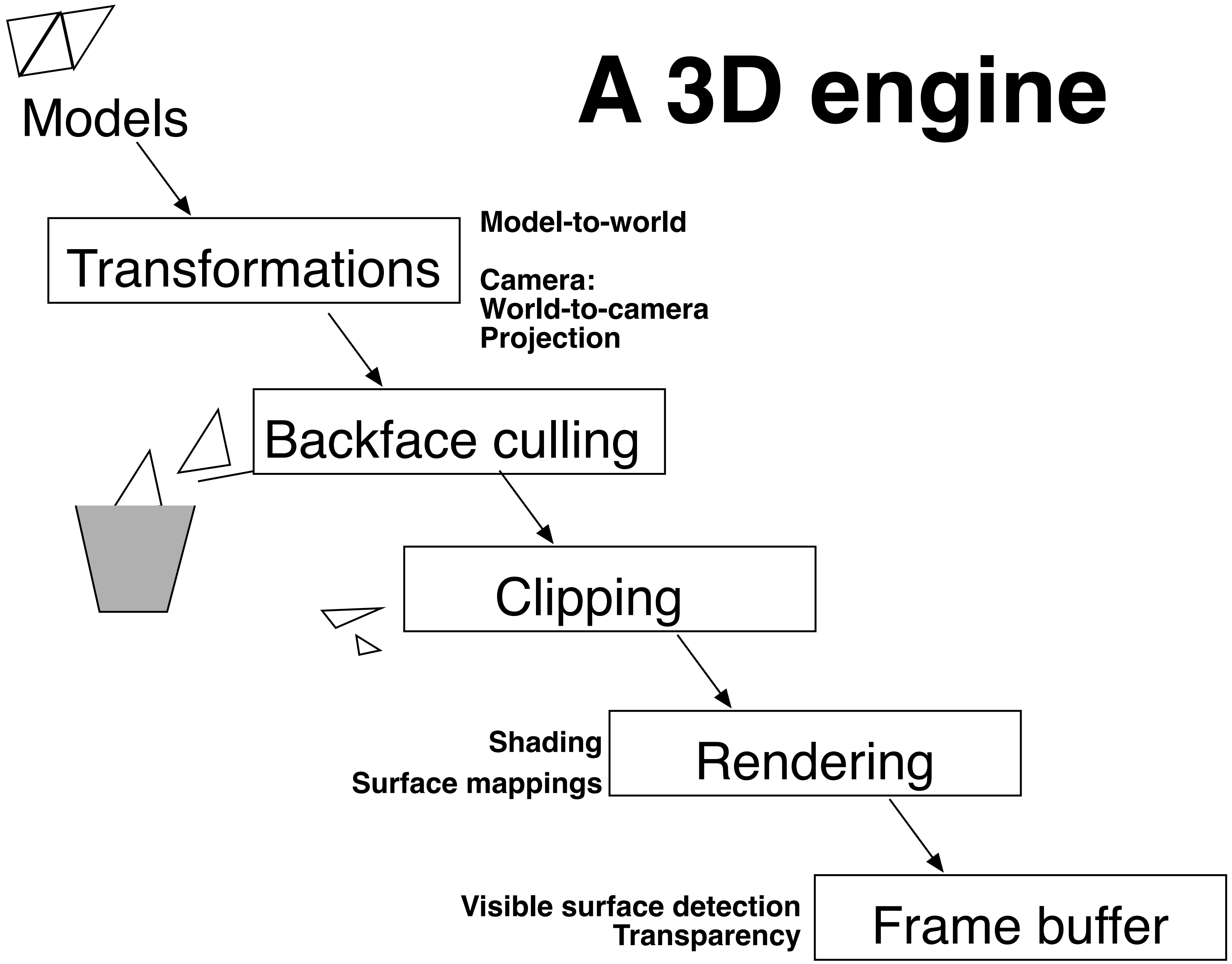
```
vn -0.57735027 -0.57735027 0.57735027
vn -0.57735027 0.57735027 0.57735027
vn 0.57735027 0.57735027 0.57735027
vn 0.57735027 -0.57735027 0.57735027
vn -0.57735027 -0.57735027 -0.57735027
vn -0.57735027 0.57735027 -0.57735027
vn 0.57735027 0.57735027 -0.57735027
vn 0.57735027 -0.57735027 -0.57735027
```

Normal vectors list

```
g cube1_default
usemtl default
f 3//3 2//2 1//1 4//4
f 5//5 1//1 2//2 6//6
f 6//6 2//2 3//3 7//7
f 7//7 3//3 4//4 8//8
f 8//8 4//4 1//1 5//5
f 8//8 5//5 6//6 7//7
```

Polygon list

A 3D engine


Models
Transformations
Model-to-world
Camera:
World-to-camera
Projection
Backface culling
Clipping
Shading
Surface mappings
Rendering
Visible surface detection
Transparency
Frame buffer

Visible surface detection

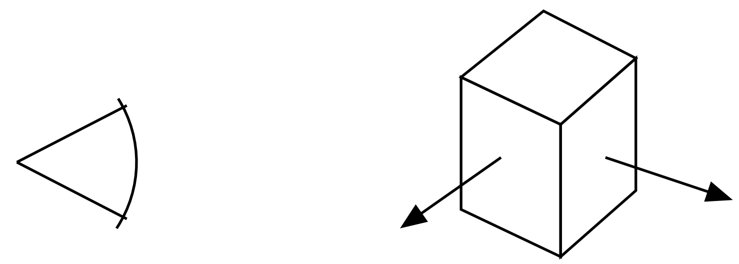
Backface culling
Z-buffer
Painter’s algorithm
BSP trees
Scan-line method

Ray-casting
A-buffer
Area subdivision
Octrees
Portals

Backface culling

Object space method

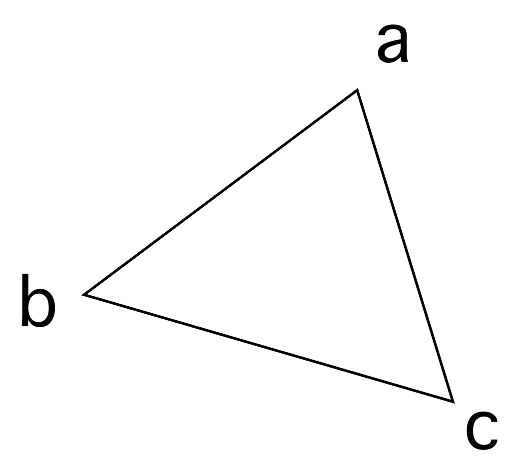
Removes all polygons that are "looking away" from the camera.



Removes ≈50% of all polygons that would otherwise be in view!

Backface culling

Based on vertex order of the projected polygon.



Inspect sign of z component of ab x ac!

Back-face culling can be configured to cull clockwise or counter-clockwise. Default is counter clockwise!

Z-buffer

Depth-buffer method

Image-space method

“Z” since we usually look along Z axis.

An extra “depth image” buffer is used, the Z buffer.

The Z buffer holds a Z value for every pixel in the image. The Z value corresponds to the nearest object written so far, that has touched that pixel.

Z-buffer algorithm

Initialize Z-buffer to infinite distance and the image buffer to background.

```
for each polygon
 for each pixel (x,y) in the polygon
  calculate z value
  if z closer than the current z-buffer value Z(x,y)
   write pixel to image (x,y)
   write z to z-buffer Z(x,y)
```

Calculation of Z

Z values must survive the projection - as specified in previous lecture

Z values are calculated for each vertex and interpolated over the surface

Culling and Z-buffer in OpenGL

```
glEnable(GL_CULL_FACE);

glCullFace(GL_BACK);
glCullFace(GL_FRONT);
```

Rarely used:

```
glFrontFace(GL_CW);
glFrontFace(GL_CCW);
```

Initialize context with depth buffer, e.g. GLUT_DEPTH

```
glEnable(GL_DEPTH_TEST);

glClear(GL_DEPTH_BUFFER_BIT);
```

Performance

Back-face culling removes polygons before fragment shader. Significant speedup!

Z-buffer test done after fragment shader. Limited performance improvement if any (saves some writes to the frame buffer, costs reads from the Z buffer.

When will culling + Z-buffering not be enough?

- Can not handle transparency
- High-level methods are needed to reduce the amount of data, to avoid passing unseen surfaces to the OpenGL pipeline