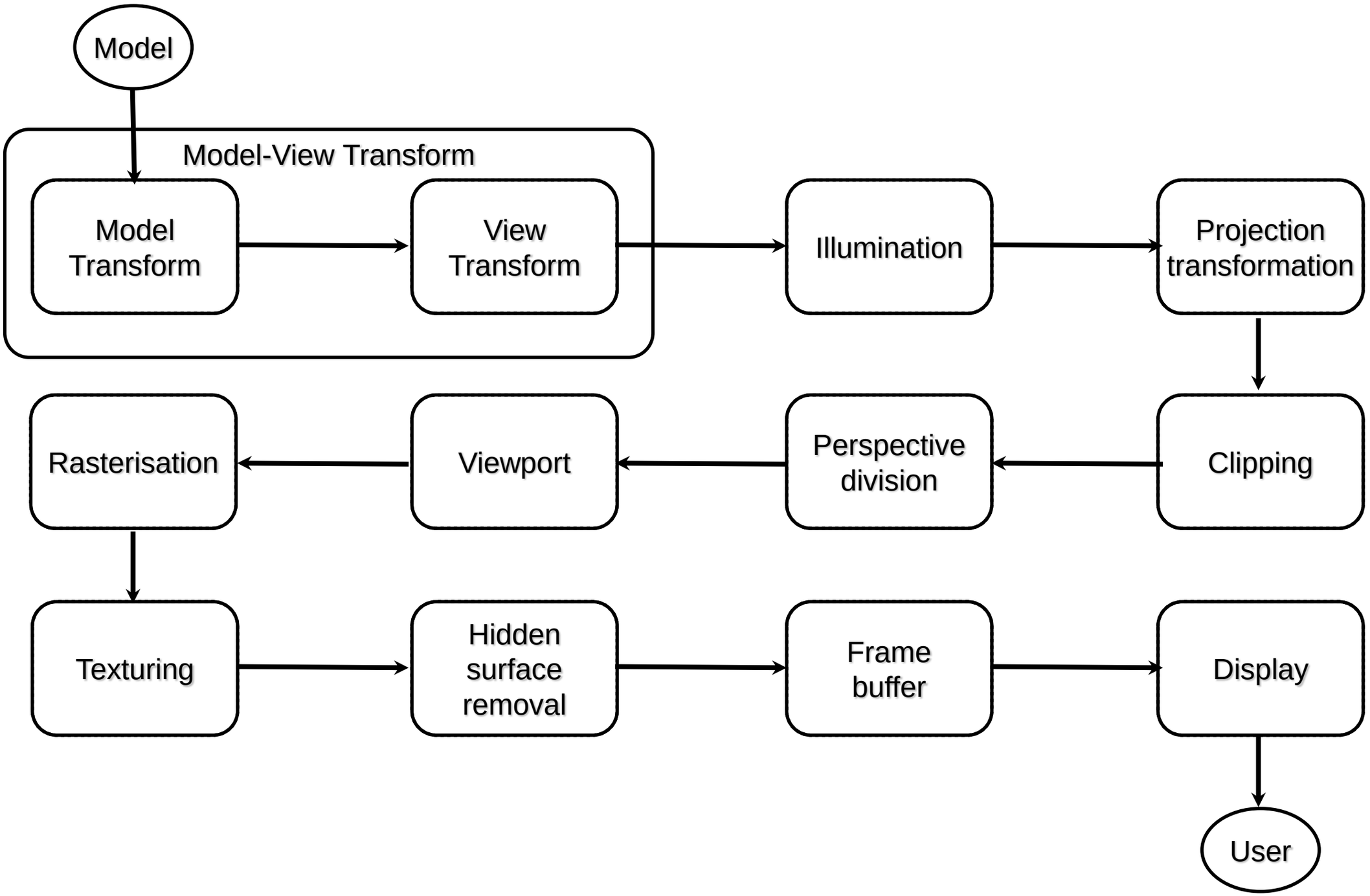
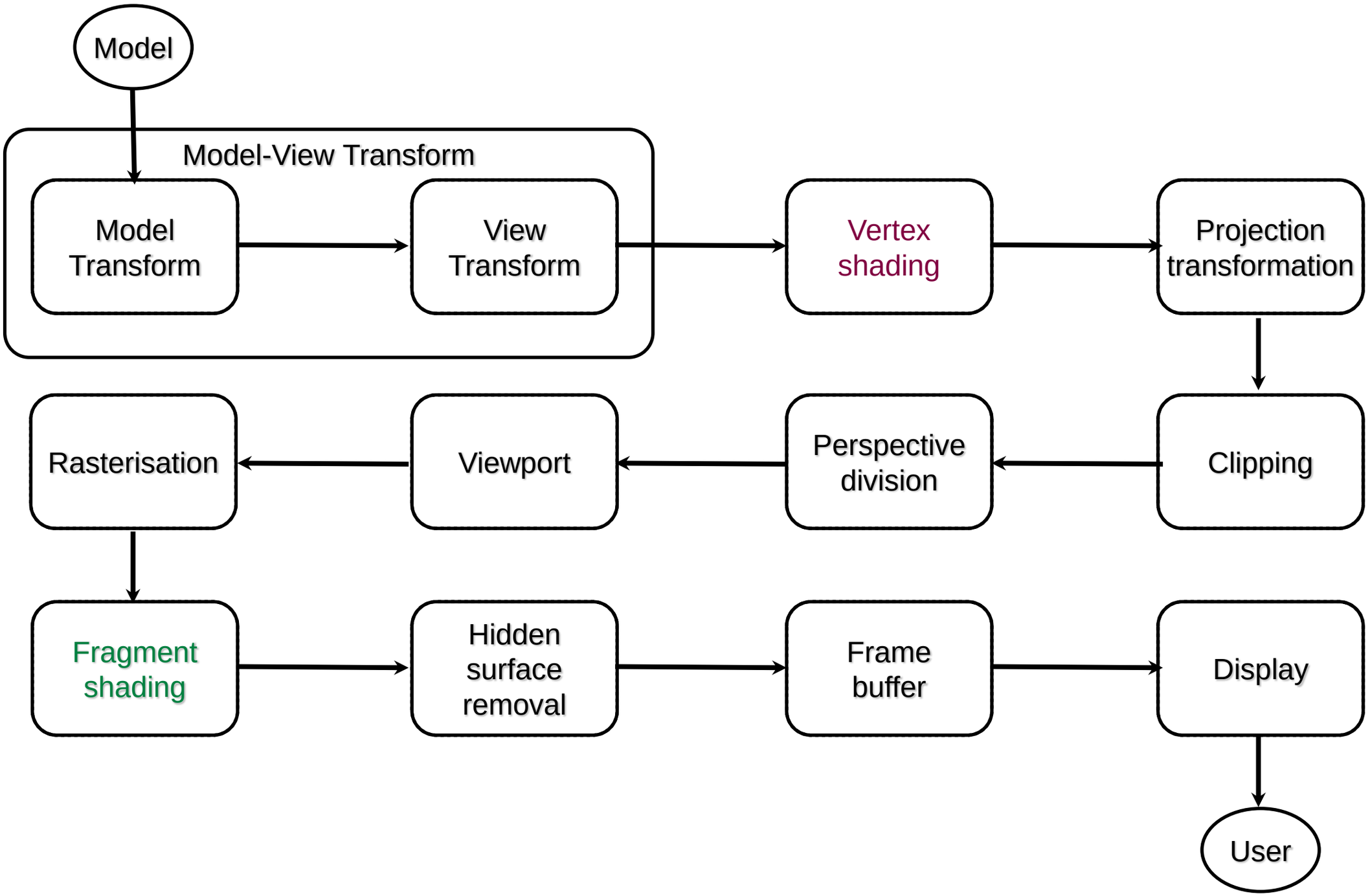
COMP3421

Shading and 3D Modeling

The graphics pipeline



The graphics pipeline



Shading

Illumination (a.k.a shading) is done at two points in the pipeline:

Vertices in the model are shaded before projection.

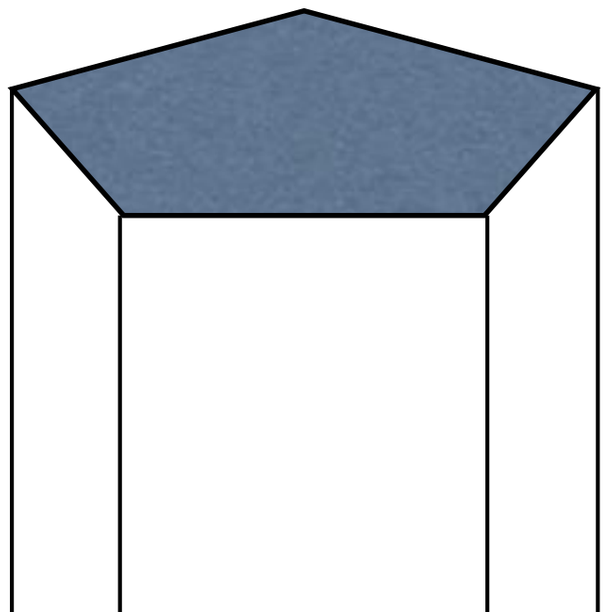
Pixels (fragments) are shaded in the image after rasterisation.

Doing more work at the vertex level the is more efficient.

Vertex shading

The built-in lighting in OpenGL is mostly done as vertex shading.

The lighting equations are calculated for each vertex in the image using the associated vertex normal.

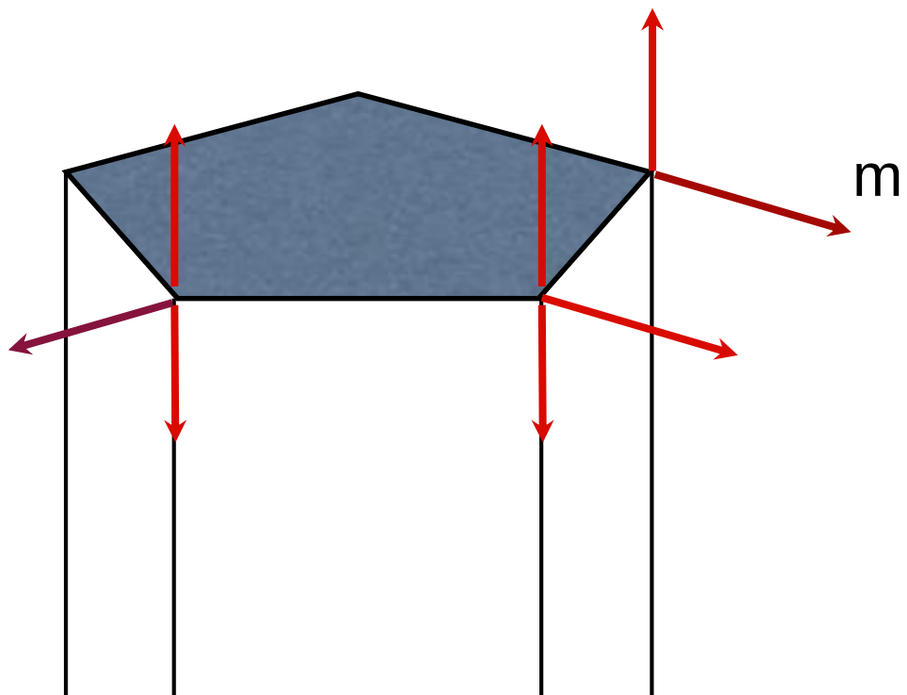


Illumination is calculated at each of these vertices.

Vertex shading

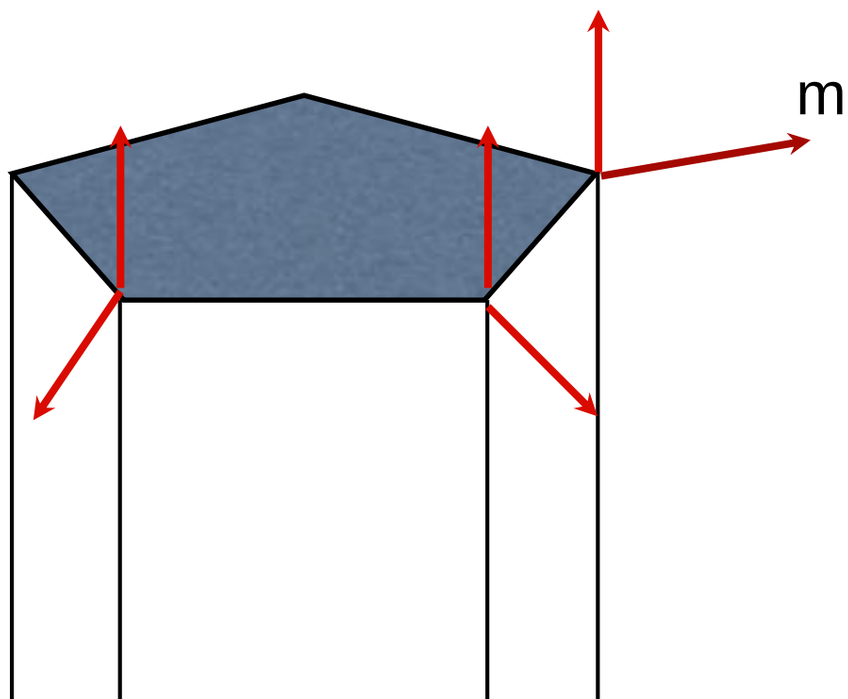
The normal vector **m** used to compute the lighting equation is the normal we specified when creating the vertex using

```
gl.glNormal3d(mx, my, mz);
```



Vertex shading

This is why we use different normals on curved vs flat surfaces, so the vertex may be lit properly.



Vertex shading

Illumination values are attached to each vertex and carried down the pipeline until we reach the fragment shading stage.

```
struct vert {

   float[3] pos; // vertex coords
   float[4] light; // rgba colour
                   // other info...

}
```

Fragments

Rasterisation converts polygons into collections of fragments. A fragment is a single image pixel with extra information attached.

```
struct frag {

    float[3] pos; // pixel coords
    float[4] color; // rgba colour

    // other info...

}
```

Fragment shading

We need to translate vertex illumination values into appropriate colours for every pixel that makes up the polygon.

There are three common options:

- Flat shading
- Gouraud shading
- Phong shading

In OpenGL

```
// Flat shading :
gl.glShadeModel(GL2.GL_FLAT);
// Gouraud shading (default):
gl.glShadeModel(GL2.GL_SMOOTH);
// Phong shading:
// No built-in implementation
```

Flat shading

The simplest option is to shade the entire face the same colour:

Choose one vertex (first for a polygon, third for a triangle)

Take the illumination of that vertex

Set every pixel to that value.

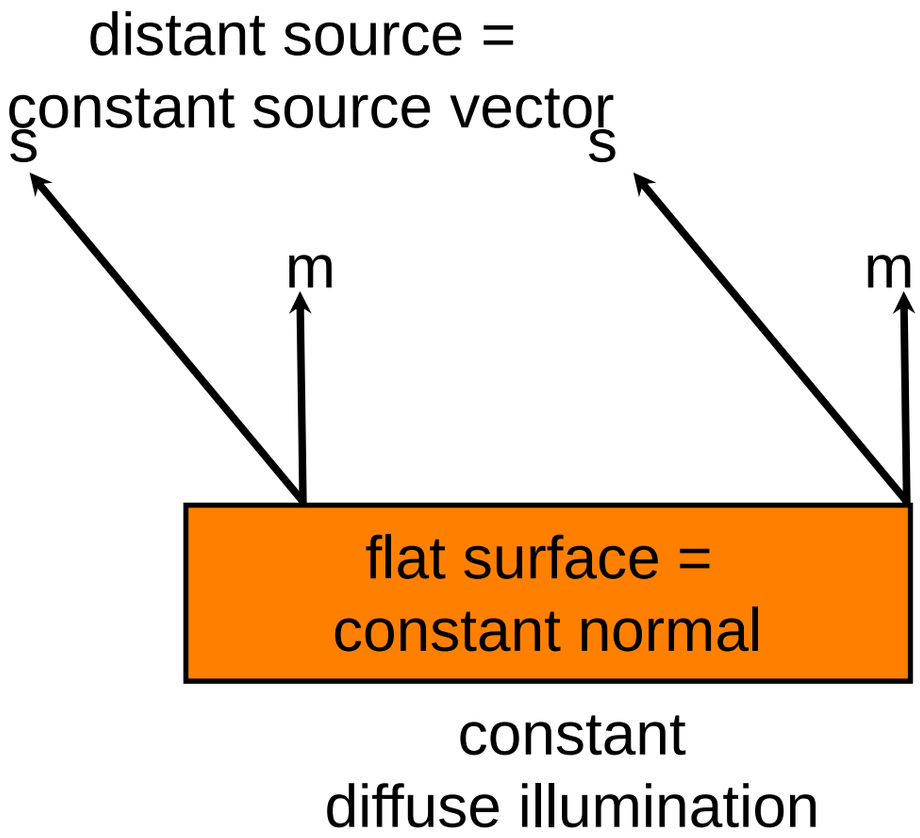
Flat shading

Flat shading is good for:

flat surfaces

distant light sources

diffuse illumination



constant
diffuse illumination

It is the fastest shading option.

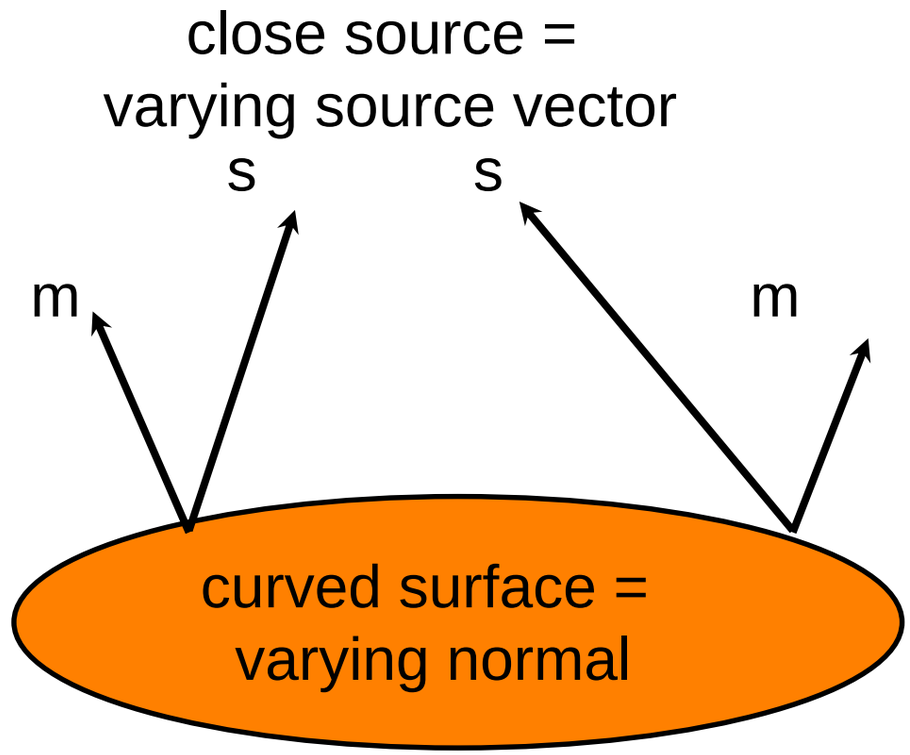
Flat shading

Flat shading is bad for:

curved surfaces

close light sources

specular shading

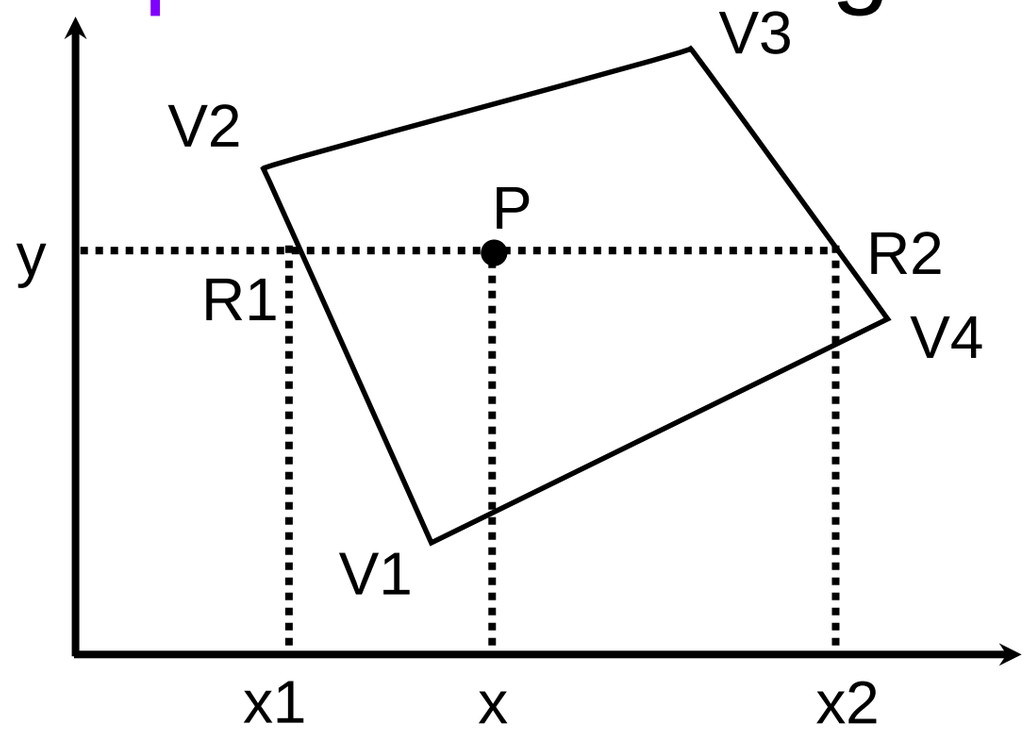


varying
diffuse + specular illumination

Gouraud shading

Gouraud shading is a simple smooth shading model.

We calculate fragment colours by bilinear interpolation on neighbouring vertices.



$$
\begin{aligned}
colour(R_1) &= lerp(V_1, V_2, \tfrac{y-y_1}{y_2-y_1}) \\
colour(R_2) &= lerp(V_3, V_4, \tfrac{y-y_3}{y_4-y_3}) \\
colour(P) &= lerp(R_1, R_2, \tfrac{x-x_1}{x_2-x_1})
\end{aligned}
$$

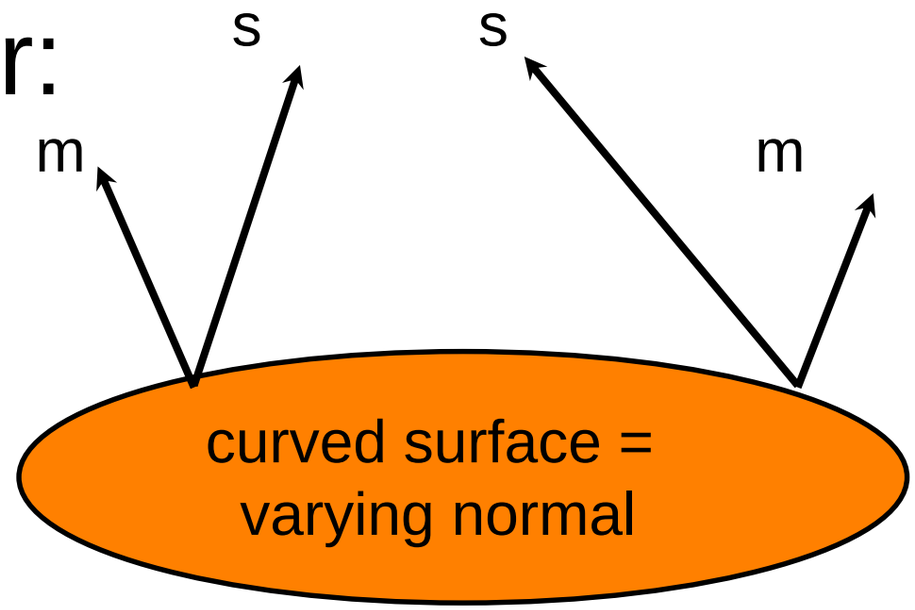
Gouraud shading

Gouraud shading is good for:

curved surfaces

close light sources

diffuse shading



varying
diffuse illumination

Gouraud shading

Gouraud shading is only slightly more expensive than flat shading.

It handles specular highlights poorly.

It works if the highlight occurs at a vertex.

If the highlight would appear in the middle of a polygon it disappears.

Phong shading

Phong shading is designed to handle specular lighting better than Gouraud.

It works by deferring the illumination calculation until the fragment shading step.

So illumination values are calculated per pixel rather than per vertex.

Phong shading

For each pixel we need to know:

source vector **s**

eye vector **v**

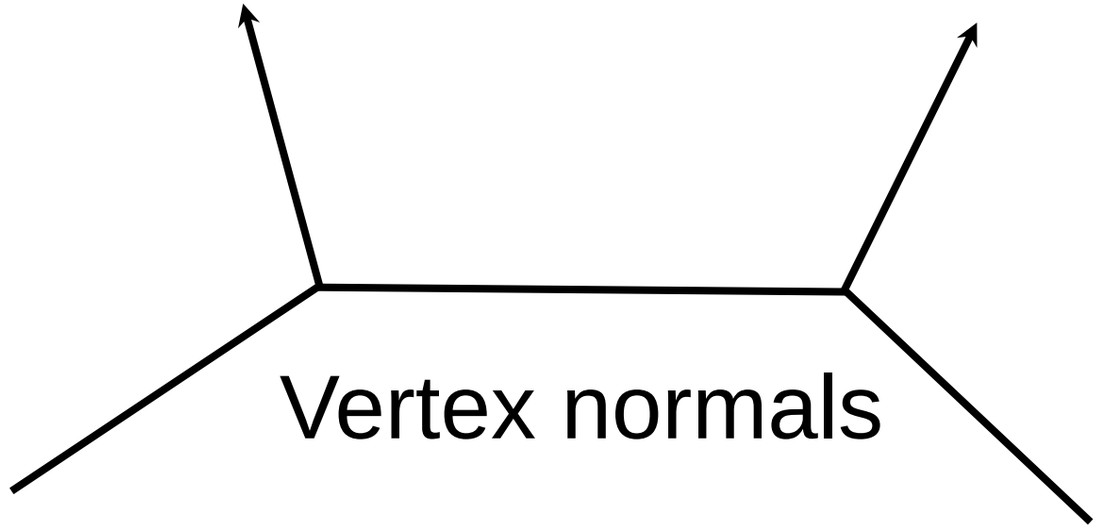
normal vector **m**

Knowing the source location, camera location and pixel location we can compute **s** and **v**.

What about **m**?

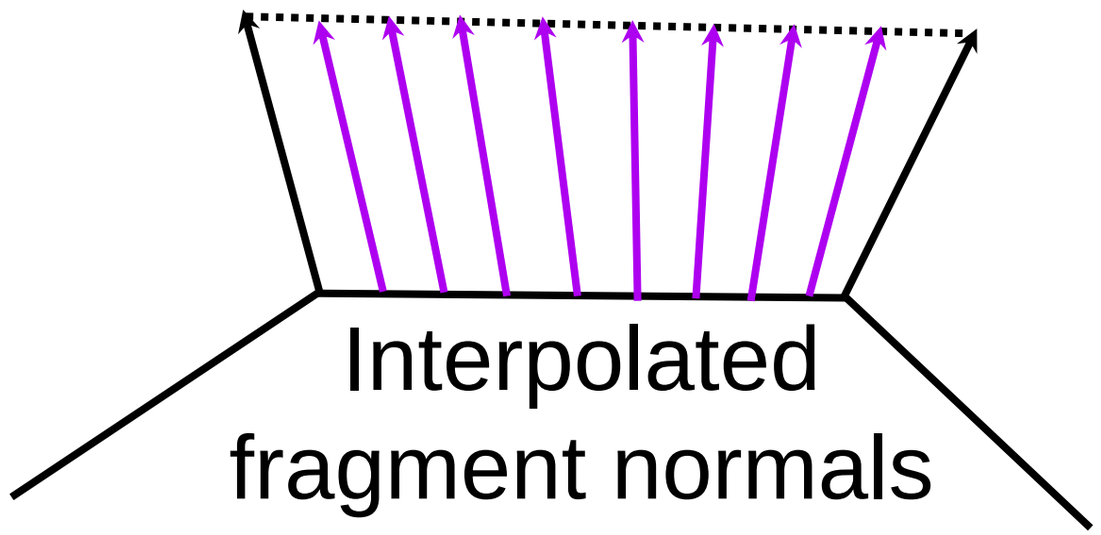
Normal interpolation

Phong shading approximates **m** by interpolating the normals of the polygon.



Normal interpolation

Phong shading approximates **m** by interpolating the normals of the polygon.



Normal interpolation

In a 2D polygon we do this using (once again) bilinear interpolation.

However the interpolated normals will vary in length, so they need to be normalised (set length = 1) before being used in the lighting equation.

Phong shading

Pros:

Handles specular lighting well.

Improves diffuse shading

More physically accurate

Phong shading

Cons:

Slower than Gouraud . Normals and illumination values have to be calculated per pixel rather than per vertex.

Shaders

Later we will discuss vertex and fragment shaders.

These are user-written programs to implement various vertex and fragment shading techniques on the graphics card.

Phong shading can be implemented as a fragment shader.

More...

There are many more shading algorithms design to implement different lighting techniques with different levels of speed and accuracy.

Check out the Graphics Gems and GPU Gems books for lots of ideas.

3D Modeling

What if we are sick of teapots?

How can we make our own 3d meshes that are not just cubes?

We will look at simple examples along with some clever techniques such as

- Extrusion
- Revolution

Exercise: Cone

How can we model a cone?

There are many ways.

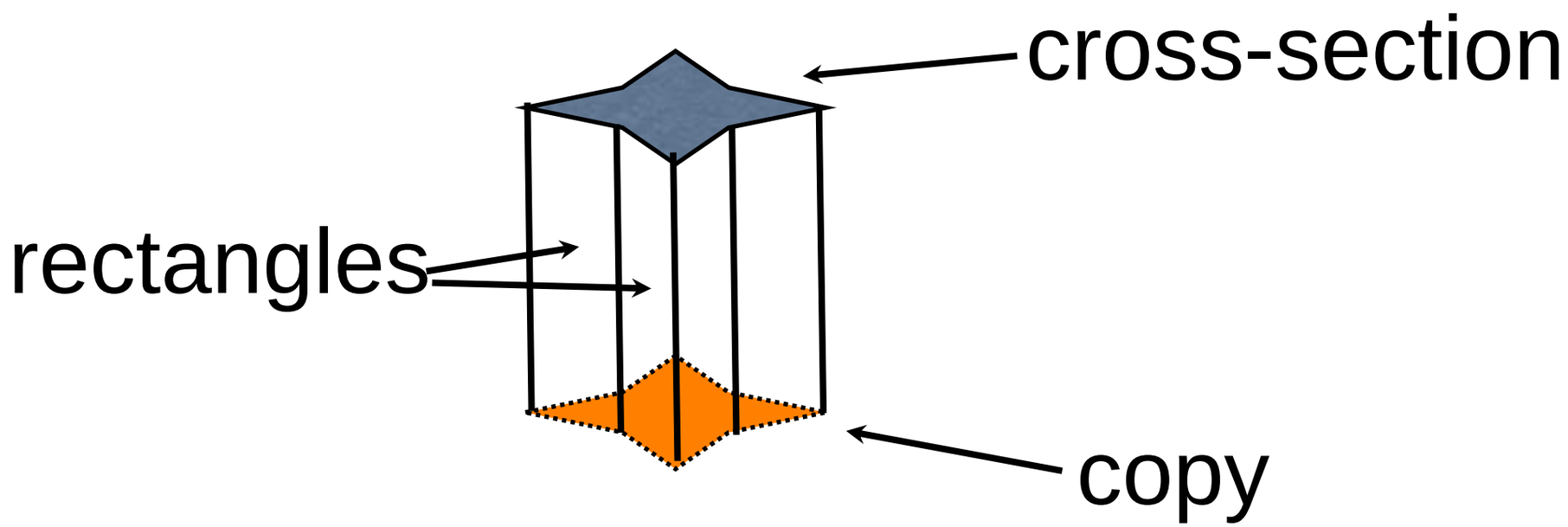
Simple way: Make a circle using a triangle fan parallel to the x-y plane. For example at z = -3

Change to middle point to lie at a different z-point for example z = -1.

Extruding shapes

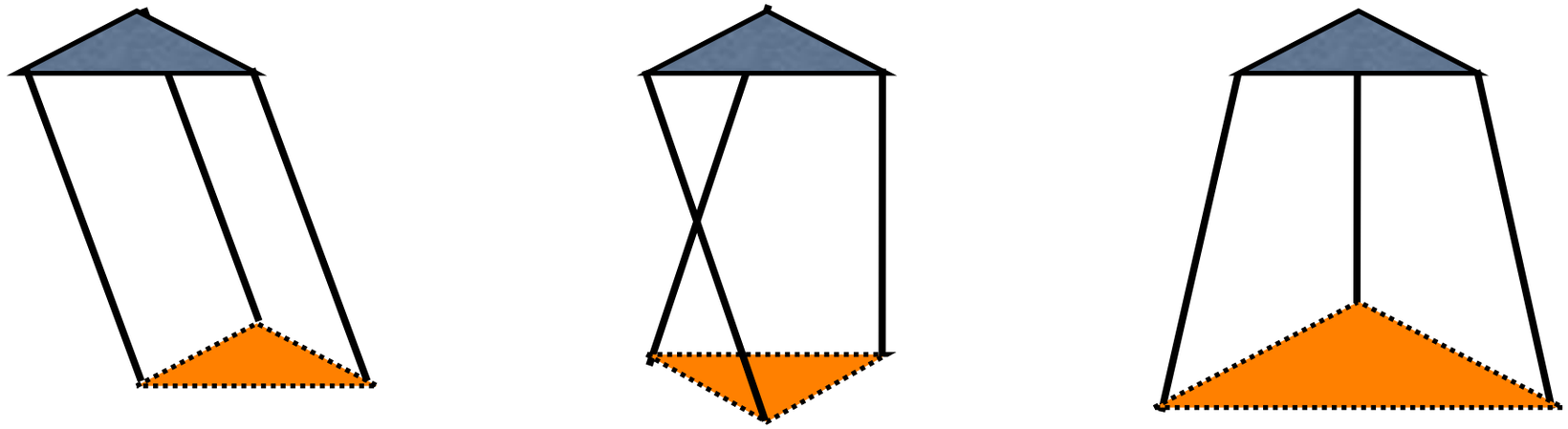
Extruded shapes are created by sweeping a 2D polygon along a line or curve.

The simplest example is a prism.



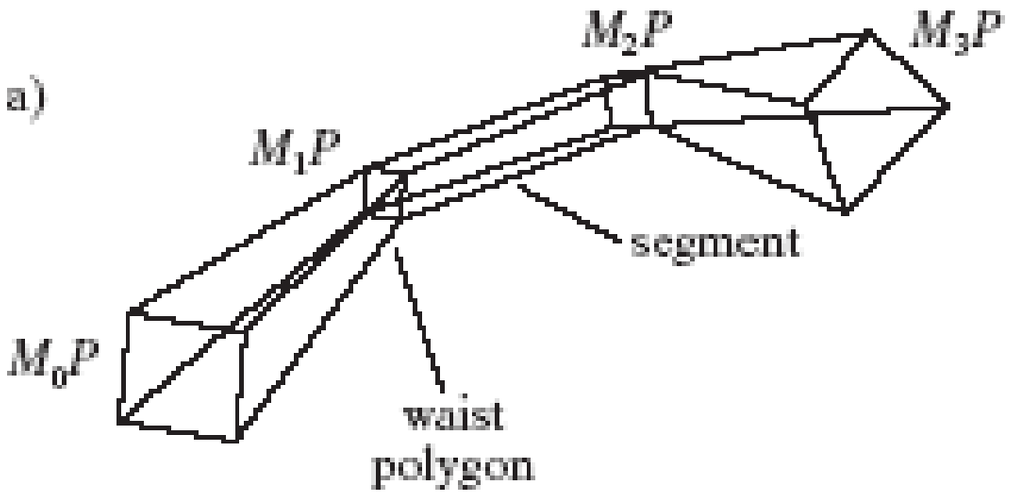
Variations

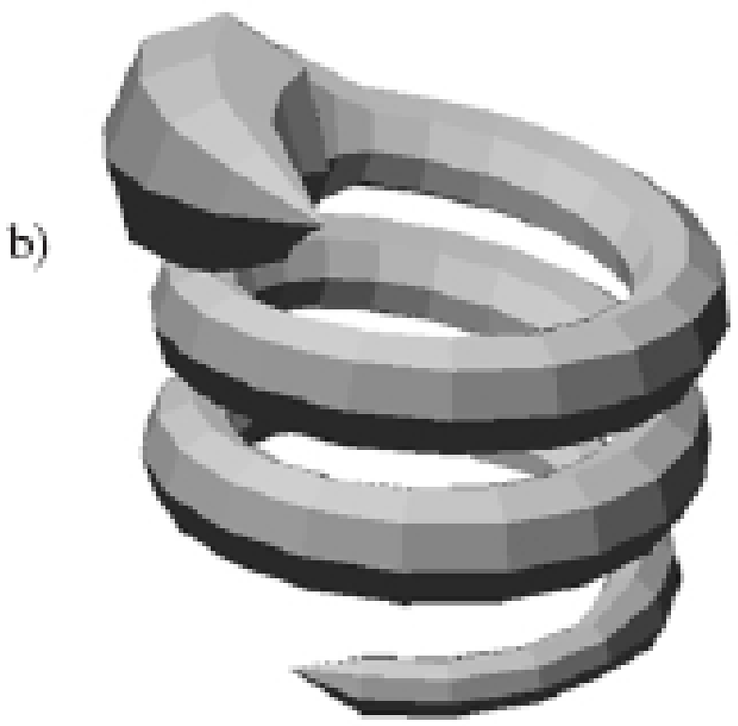
One copy of the prism can be translated, rotated or scaled from the other.



Segmented Extrusions

A square P extruded three times, in different directions with different tapers and twists. The first segment has end polygons M_0P and M_1P, where the initial matrix M_0 positions and orients the starting end of the tube. The second segment has end polygons M_1P and M_2P, etc.





Segmented extrusions

We can extrude a polygon along a path by specifying it as a series of transformations.

$$poly = P_0, P_1, \ldots, P_k$$

$$path = \mathbf{M_0}, \mathbf{M_1}, \ldots, \mathbf{M_n}$$

At each point in the path we calculate a cross-section:

$$poly_i = \mathbf{M_i}P_0, \mathbf{M_i}P_1, \ldots, \mathbf{M_i}P_k$$

Segmented Extrusion

Sample points along the spine using different values of t

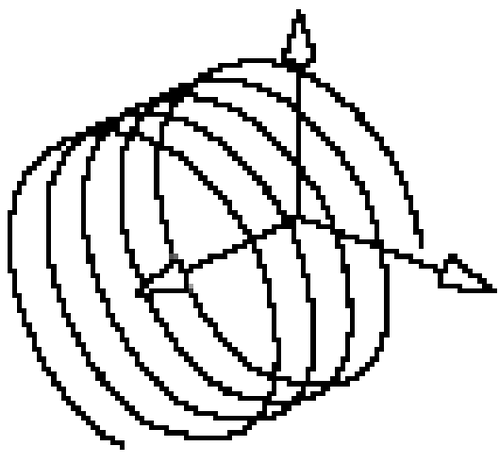
For each t:

- generate the current point on the spine
- generate a transformation matrix
- multiply each point on the cross section by the matrix.
- join these points to the next set of points using quads/triangles.

Segmented Extrusion Example

For example we may wish to extrude a circle cross-section around a helix spine.

helix $C(t) = (cos(t), sin(t), bt)$).



Transformation Matrix

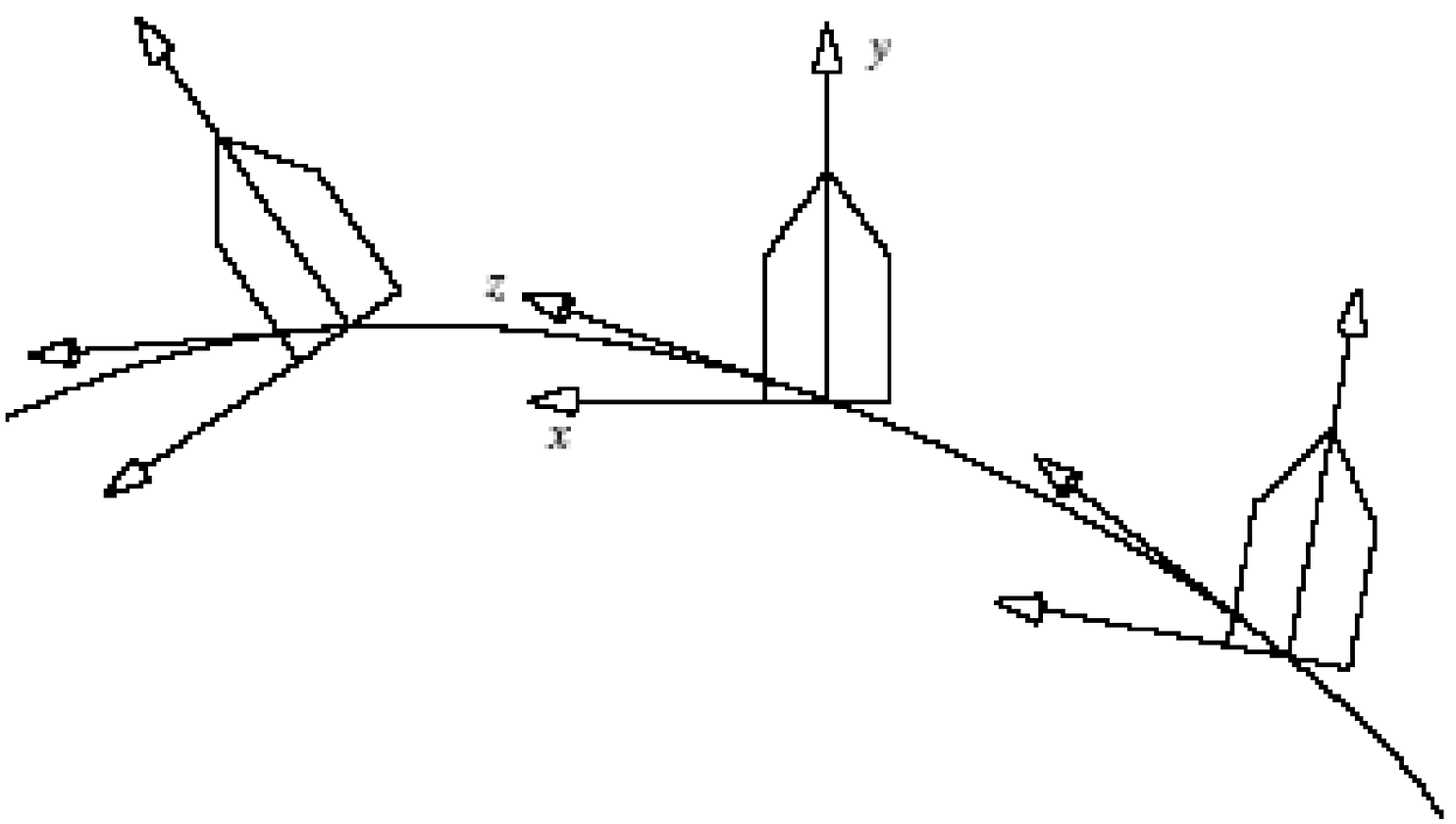
How can we automatically generate a matrix to transform our cross-section by?

We need the origin of the matrix to be the new point on the spine. This will translate our cross-section to the correct location.

Which way will our cross-section be oriented? What should i, j and k of our matrix be?

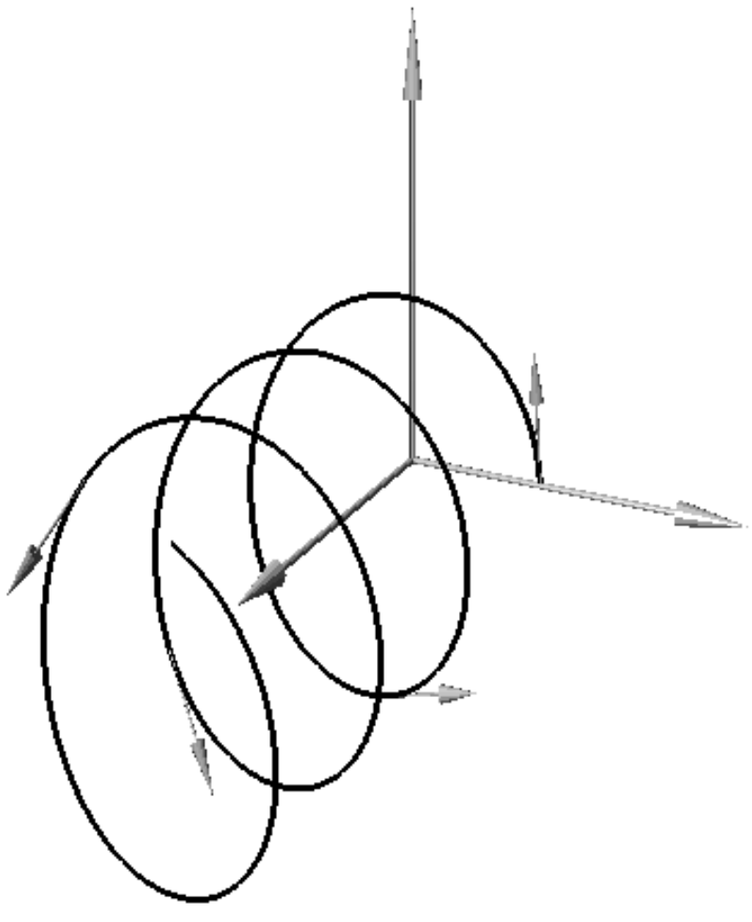
Frenet Frame

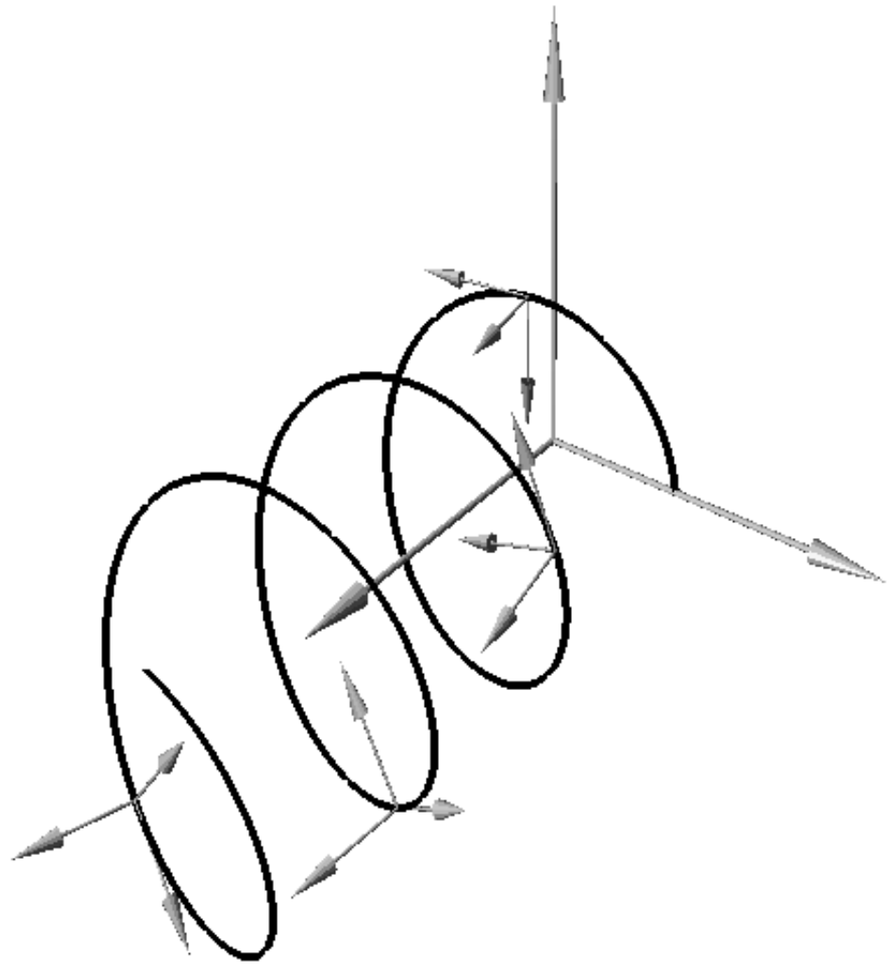
We can get the curve values at various points t_i and then build a polygon perpendicular to the curve at $C(t_i)$ using a Frenet frame.



Example

a). Tangents to the helix. b). Frenet frame at various values of *t*, for the helix.





Frenet Frame

Once we calculate the tangent to the spine at the current point, we can use this to calculate normals.

We then use the tangent and the 2 normals as i, j and k vectors of a co-ordinate frame.

We can then build a matrix from these vectors, using the current point as the origin of the matrix.

Frenet frame

We align the **k** axis with the (normalised) tangent, and choose values of **i** and **j** to be perpendicular.

$$
\begin{aligned}
\phi &= \Phi(t) \\
\mathbf{k} &= \hat{\Phi}'(t) \\
\mathbf{i} &= \begin{pmatrix} -k_2 \\ k_1 \\ 0 \end{pmatrix} \\
\mathbf{j} &= \mathbf{k} \times \mathbf{i}
\end{aligned}
$$

Frenet Frame Calculation

Finding the tangent (our k vector):

1. Using maths. Eg for

$C(t) = (\cos(t), \sin(t), bt)$

$T(t) \equiv \text{normalise}(-\sin t(t), \cos(t), b)$

2. Or just approximate the tangent

$T(t) = \text{normalise}(C(t+1) - C(t-1))$

Frenet Frame Calculation

If our tangent at t is the vector

T(x,y,z)

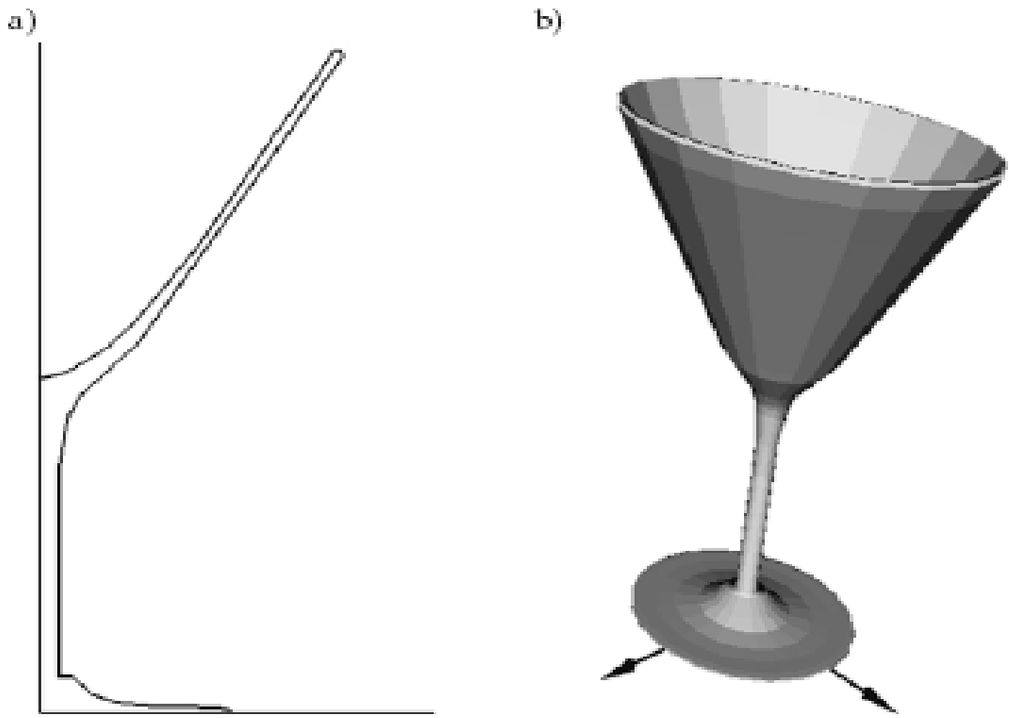
We can use the normal

N(-y,x,0). This will be our i vector

To find the other normal we simply do k x i

Revolution

A surface with radial symmetry (i.e. a round object, like a ring, a vase, a glass) can be made by sweeping a half cross-section around an axis.

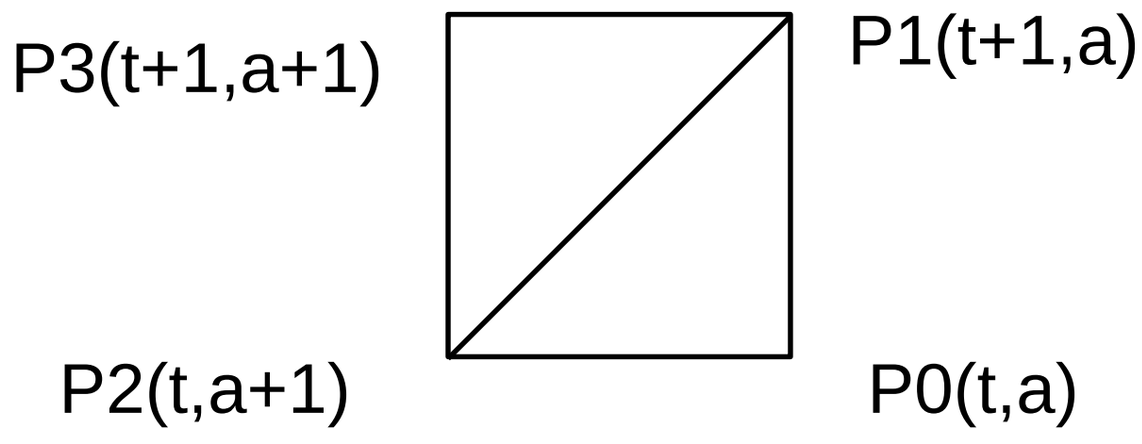


Revolution

Take your 2d function which can generate points for X(t) and Y(t) and sample them for different values of t and angles of a.

//Revolution around the Y-axis

P(t,a) = (X(t)cos a, Y(t), X(t)sin a)



Normals To Surface

If we do not know a precise function for our normals, we can approximate them. For example, one common way:

For each vertex:

Take an average of the surrounding face normals for the vertex. Normalise it and use that as the normal for the vertex.