



OpenGL buffers

- frame buffer** consists of pixel (RGB values)
- alpha buffer** alpha is the pixel's transparency
0 - opaque, 255 fully transparent
- z-buffer (depth)** z is the distance
of a pixel from the viewer
z-test is used for visibility



OpenGL buffers

- stencil buffer** controls special writing operations
- accumulation buffer** used to store intermediate images
(antialiasing, motion blur, etc.)
- index color buffer** obsolete, 1 byte per color value
- Some values are set explicitly by the user
Some are set by rendering



Clearing a buffer

- 1) specify the value (once in the program)
- 2) clear the buffer

ad 1) specify the value

```
glClearColor(GLclampf r, GLclampf g,
             GLclampf b, GLclampf a)
glClearDepth(GLclampd d)
glClearStencil(GLint s)
glClearAccum(GLfloat r, GLfloat g, GLfloat b, GLfloat a)
glClearIndex(GLfloat index)
```



Clearing a buffer

ad 2) clear the buffer

```
void glClear(GLbitfield mask)
```

where mask is a logical or consisting of:

color	GL_COLOR_BUFFER_BIT
depth	GL_DEPTH_BUFFER_BIT
stencil	GL_STENCIL_BUFFER_BIT
accumulation	GL_ACCUM_BUFFER_BIT

typically:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```



Forcing completion of drawing

due to optimization some command may be stuck
there are two “chimney-sweeper” commands

```
void glFlush()  
void glFinish()
```

the `glFlush()` command forces the OpenGL to finish all previously executed commands in finite time

the `glFinish()` command forces the OpenGL to finish all previously executed commands and waits for the result
- if you think of CPU and OpenGL as a parallel processes

what kind of synchronization task is it?



- everything in the OpenGL is composed of vertices
- the most important calculations are done on vertices (lighting, transforms)
- vertex is specified in *homogenous coordinates* [x, y, z, w]

- *note:* for $w < 0$ the result is not guarantee
- vertex specification:

```
void glVertex{234}{sifd}[v](TYPE coords)
```



if the w coordinate is not specified 1.0 is the default
if the z coordinate is not specified 0.0 is the default
i.e., all internal calculations are done in 3D!

example:

```
GLfloat foo[]={30.0, 20.0, 0, 10};  
glVertex4fv(foo);
```

and

```
glVertex2f(3,2);  
are equal (Why?)
```



`glBegin` command sets the OpenGL to the state
“now draw something”

`glEnd` command says that this is the end
think of it as *semantic parenthesis*

an example:

```
glBegin(GL_POINTS); // set the GL state  
    glVertex3f(0.0, 0.0, 0.0);  
    glVertex3f(1.0, 0.0, 0.0);  
    glVertex3f(1.0, 1.0, 0.0);  
glEnd(); //end of rendering
```



OpenGL supports:

points GL_POINTS

lines GL_LINES

polylines GL_LINE_STRIP

closed polylines GL_LINE_LOOP

triangles GL_TRIANGLES

triangle strips GL_TRIANGLE_STRIP

triangle fans GL_TRIANGLE_FAN



OpenGL supports:

quadrilaterals GL_QUADS

quadrilateral strips GL_QUAD_STRIP

polygons GL_POLYGON



Examples:

`glBegin(GL_POINTS)`



```
glBegin(GL_POINTS);
glColor3f(1.0,0.0,0.0);glVertex2i(0,0);
glColor3f(0.0,1.0,0.0);glVertex2i(100,0);
glColor3f(0.0,0.0,1.0);glVertex2i(100,100);
glColor3f(1.0,1.0,0.0);glVertex2i(0,100);
glEnd();
```

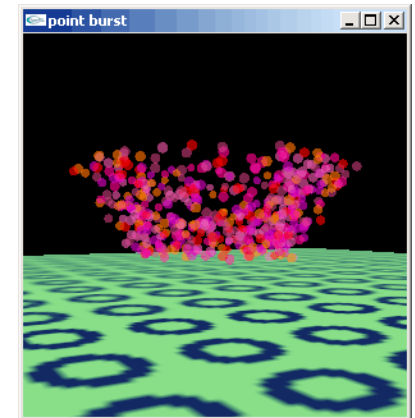


Examples:

Can we do something interesting with points?

Of course YES!

Look at the example
of an explosion!

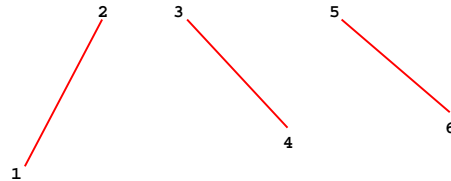


Example by Mark Kilgard

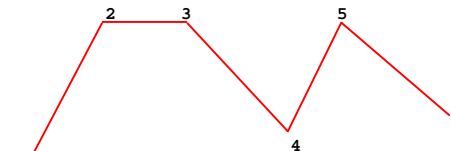


Examples:

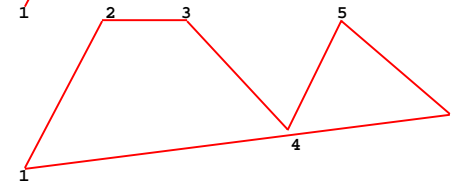
`glBegin(GL_LINES)`



`glBegin(GL_LINE_STRIP)`



`glBegin(GL_LINE_LOOP)`



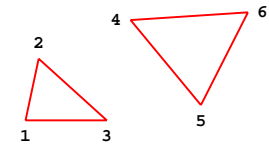
(C) Bedrich Benes

13

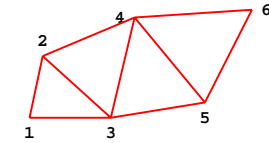


Examples:

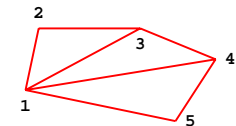
`glBegin(GL_TRIANGLES)`



`glBegin(GL_TRIANGLE_STRIP)`



`glBegin(GL_TRIANGLE_FAN)`



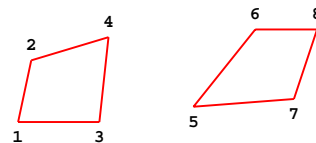
(C) Bedrich Benes

14

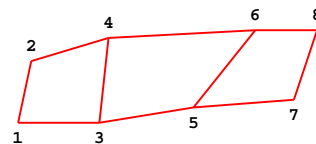


Examples:

`glBegin(GL_QUADS)`



`glBegin(GL_QUAD_STRIP)`



(C) Bedrich Benes

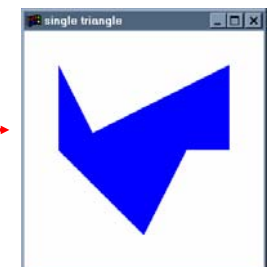
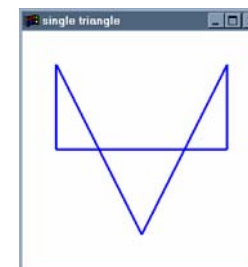
15



Examples:

`glBegin(GL_POLYGON)`

must be convex, no holes, must not intersect itself
otherwise unpredictable results!



(C) Bedrich Benes

16



Triangle strips

less than three vertices per Δ !
 a sequential triangle strip is defined by
 an ordered vertex list $[v_0, v_1, \dots, v_n]$
 the i -th triangle $\Delta v_{i-1} v_i v_{i+1}$
 the vertex list includes $n-2$ triangles
 average # of vertices per triangle



Triangle strips

average # of vertices per triangle v_a
 $v_a = 3 + (m-1)/m = 1 + 2/m$
 where m is the length of the list

$$\lim_{m \rightarrow \infty} v_a = 1$$

imagine $m=100$, $v_a = 1.02$ (wow!)
 we save 2/3 of space



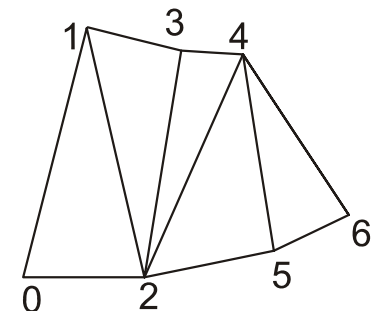
Triangle strips vs fans

the same stands for triangle fans,
 many providers (NVIDIA) suggest using strips
 they are better optimized



Generalized triangle strips

the sequence of Δ s is *not* strictly sequential
 to use them, we need a vertex cache
 or the *swap* operation
 $[0, 1, 2, 3, \text{swap } 4, 5, 6]$
 IrisGL does it (for example)
 In OpenGL the vertex must
 be resend
 $[0, 1, 2, 3, 2, 4, 5, 6]$





Triangle fans

a general convex polygon can be converted into one Δ fan or a simple Δ strip

v_a is the same as for Δ strips (the same startup)

any Δ fan can be converted into generalize Δ strip (repeat the center or many swaps)

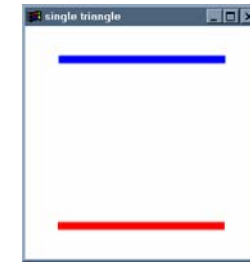
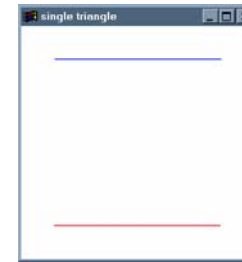
not vice versa



Attributes

Points `glPointSize(GLfloat size)`

Lines `glLineWidth(GLfloat size)`



Attributes

Line stippling

`glLineStipple(GLint n, GLushort pattern)`

pattern:

0 does not draw a fragment

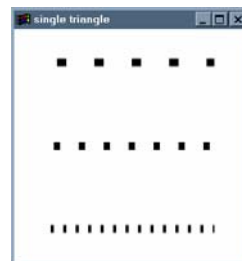
1 draw a fragment

line stippling must be enabled

`glEnable(GL_LINE_STIPPLE)`

or can be disabled

`glDisable(GL_LINE_STIPPLE)`



Attributes

Polygon stippling

`glPolygonStipple(const GLubyte *mask)`

mask:

32x32 bits

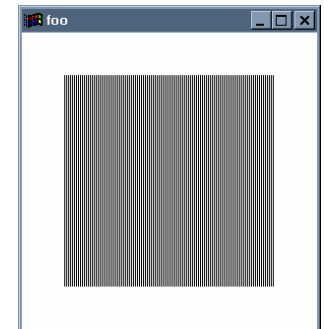
example 0x55

polygon stippling must be enabled

`glEnable(GL_POLYGON_STIPPLE)`

or can be disabled

`glDisable(GL_POLYGON_STIPPLE)`





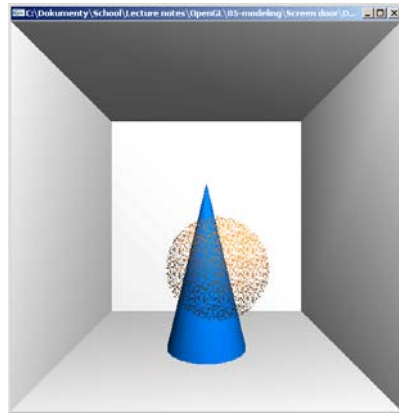
Attributes

Polygon stippling example

using *random* mask we can define so called

screen door transparency

every fragment is 0 or 1,
but the area has
n% of transparency



Example by Celeste Fowler



Model of a cube 1

```
#define A {0,0,0}
```

```
#define B {1,0,0}
```

```
GLfloat vert[8][3]={A,B,C,D,E,F,G,H};
```

```
glBegin(GL_QUAD_STRIP);
```

```
glVertex3fv(vert[0]); //A
```

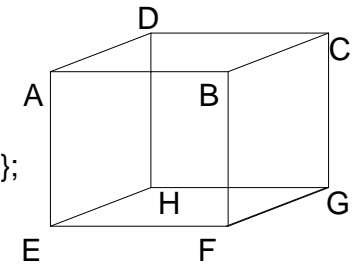
```
glVertex3fv(vert[3]); //D
```

```
glVertex3fv(vert[1]); //B
```

```
glVertex3fv(vert[2]); //C
```

```
etc...
```

```
glEnd();
```



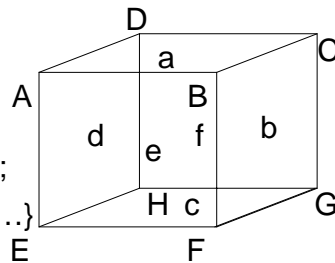
Model of a cube 2

```
#define A {0,0,0}
```

```
#define B {1,0,0}
```

```
GLfloat vert[8][3]={A,B,C,D,E,F,G,H};
```

```
GLint faces[4][6]={0,1,2,3},{1,2,5,6}...
```



```
glBegin(GL_QUADS);
```

```
for (i=0;i<6;i++)
```

```
for (j=0;j<4;j++)
```

```
glVertex3fv(vert[face[i][j]]); //ufff.....
```

```
glEnd();
```

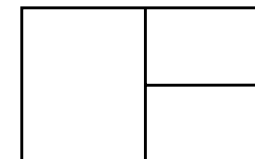


In general

- it is good to keep a list of smallest versatile elements
- in general the best is a list of triangle strips
- OpenGL is a RENDERER not a modeler
models can be imported
- Avoid T-vertices!

They can crack when transformed

They cause bad illumination artifacts





Tessellation is a process of converting a complex object into simpler primitives such as triangles or quadrilaterals

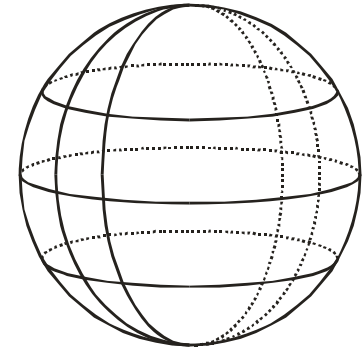
Tessellation is the most frequently wrongly spelled word in Computer Graphics



Sphere stacks and slices:

not a very good approach:

- triangles have different size
- involves cos and sin functions
- some triangles are be very long

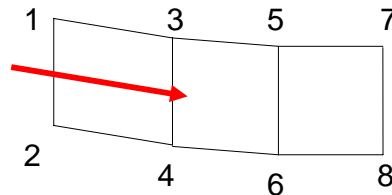


Sphere stacks and slices - source code:

- one quad strip corresponds to one stack (or slice)
- program it as double cycle
- sphere equation in 3D
 $x^2+y^2+z^2=r^2$... is useless

$x(u,v)=r \cos(u)\sin(v)$
 $y(u,v)=r \cos(u)\cos(v)$
 $z(u,v)=r \sin(u)$

$0 \leq u \leq \pi; 0 \leq v \leq 2\pi$



Sphere stacks and slices - source code:

```
#define PI 3.1415926535897932384626433832795 //does it make sense?
```

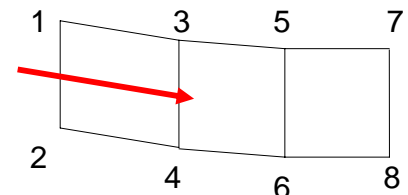
```
#define K PI/10.f
```

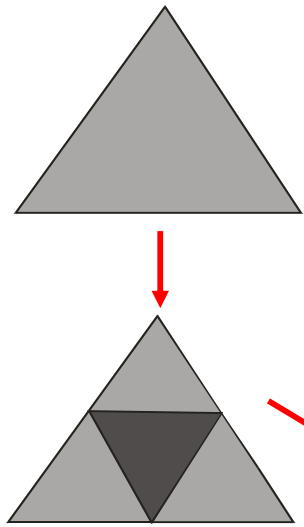
```
#define R 1.0
```

```
GLfloat u,v,x,y,z;
```

```
for (u=0;u<=2*PI;u+=K){  
  glBegin(GL_QUAD_STRIP); //start a new strip
```

```
  for (v=0;v<=PI+K;v+=K) {  
    x=R*cos(u)*sin(v);  
    y=R*sin(u)*sin(v);  
    z=R*cos(v);  
    glVertex3f(x,y,z); //first vertex  
    x=R*cos(u+K)*sin(v);  
    y=R*sin(u+K)*sin(v);  
    glVertex3f(x,y,z); //second vertex  
  }  
  glEnd(); //end of the strip  
}
```



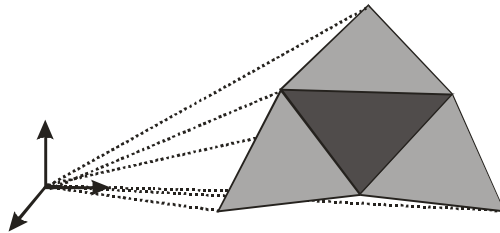


Recursive subdivision

much better approach

the principal idea:

- 1) take a triangle
- 2) Subdivide it recursively
- 3) normalize the distance of each vertex from the origin (so they will be on a unit sphere)



(C) Bedrich Benes

33



Subdivision of a triangle data structures:

```
typedef struct TriangleH
{
    GLfloat a[3];
    GLfloat b[3];
    GLfloat c[3];
} TriangleT;
```

two advantages:

- 1) Encapsulated
- 2) Vertices are arrays

(C) Bedrich Benes

34



Subdivision of a triangle - procedure

we need one macro:

```
#define MIDDLE(a,b,c) {
    c[0]=(a[0]+b[0])/2.0;
    c[1]=(a[1]+b[1])/2.0;
    c[2]=(a[2]+b[2])/2.0;
}
```

it could be also an inline procedure....

It takes points a and b and returns c as a mid-point

(C) Bedrich Benes

35



Subdivision of a triangle - procedure

```
void Normalize(GLfloat *n)
{
    static GLfloat size;

    size = (n[0]*n[0]+n[1]*n[1]+n[2]*n[2]);
    size=sqrt(size);
    n[0] /= size;
    n[1] /= size;
    n[2] /= size;
}
```

(C) Bedrich Benes

36



Subdivision of a triangle - procedure

```
void SubdivideTriangle(TriangleT *p, int n){
    TriangleT up, lowleft, lowright, mid;

    //normal vector and point has the same coordinates
    //on the unit sphere
    //if a is zero, we do not want to subdivide anymore
    if (n==0) { //kernel of recursion
        glNormal3fv(p->a);glVertex3fv(p->a);
        glNormal3fv(p->b);glVertex3fv(p->b);
        glNormal3fv(p->c);glVertex3fv(p->c);
        return;
    }
```

//the code goes on the next slide

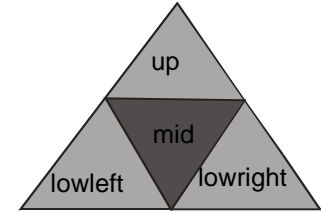
(C) Bedrich Benes

37



Subdivision of a triangle procedure (contd.)

```
up.a[0]=p->a[0];up.a[1]=p->a[1];up.a[2]=p->a[2];
MIDDLE(p->a,p->b,up.b);
Normalize(up.b);
MIDDLE(p->a,p->c,up.c);
Normalize(up.c);
SubdivideTriangle(&up,n-1);
```



```
lowleft.a[0]=up.b[0];lowleft.a[1]=up.b[1];lowleft.a[2]=up.b[2];
lowleft.b[0]=p->b[0];lowleft.b[1]=p->b[1];lowleft.b[2]=p->b[2];
MIDDLE(p->b,p->c,lowleft.c);
Normalize(lowleft.c);
SubdivideTriangle(&lowleft,n-1);
```

(C) Bedrich Benes

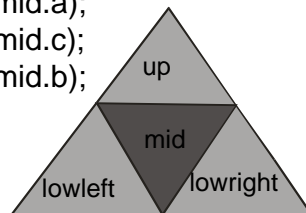
38



Subdivision of a triangle procedure (contd.)

```
lowright.a[0]=up.c[0];lowright.a[1]=up.c[1];lowright.a[2]=up.c[2];
lowright.b[0]=lowleft.c[0];lowright.b[1]=lowleft.c[1];
lowright.b[2]=lowleft.c[2];
lowright.c[0]=p->c[0];lowright.c[1]=p->c[1];lowright.c[2]=p->c[2];
SubdivideTriangle(&lowright,n-1);
```

```
MIDDLE(p->b, p->c, mid.a);Normalize(mid.a);
MIDDLE(p->a, p->b, mid.c);Normalize(mid.c);
MIDDLE(p->a, p->c, mid.b);Normalize(mid.b);
SubdivideTriangle(&mid,n-1);
}
```



(C) Bedrich Benes

39

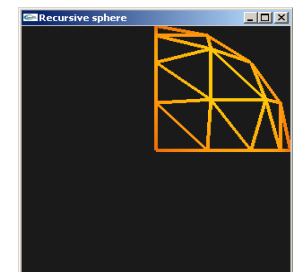
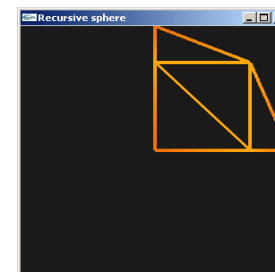
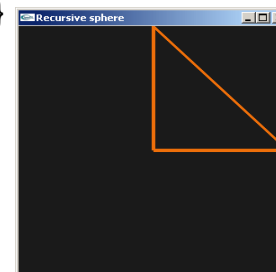


Let's try it with one triangle

```
TriangleT x={{0,1,0},{1,0,0},{0,0,1}};
```

```
void DisplayPieceofASphere(int n)
```

```
{
    glBegin(GL_TRIANGLES);
        SubdivideTriangle(&x,n);
    glEnd();
}
```



(C) Bedrich Benes

40



Subdivision of a triangle procedure (contd.)

The entire sphere:

```
void DisplaySphere(int n)
{
    glBegin(GL_TRIANGLES);
    SubdivideTriangle(&p1,n); //p1,p2,...,r2 must be defined
    SubdivideTriangle(&q1,n);
    SubdivideTriangle(&r1,n);
    SubdivideTriangle(&p2,n);
    SubdivideTriangle(&q2,n);
    SubdivideTriangle(&r2,n);
    glEnd();
}
```

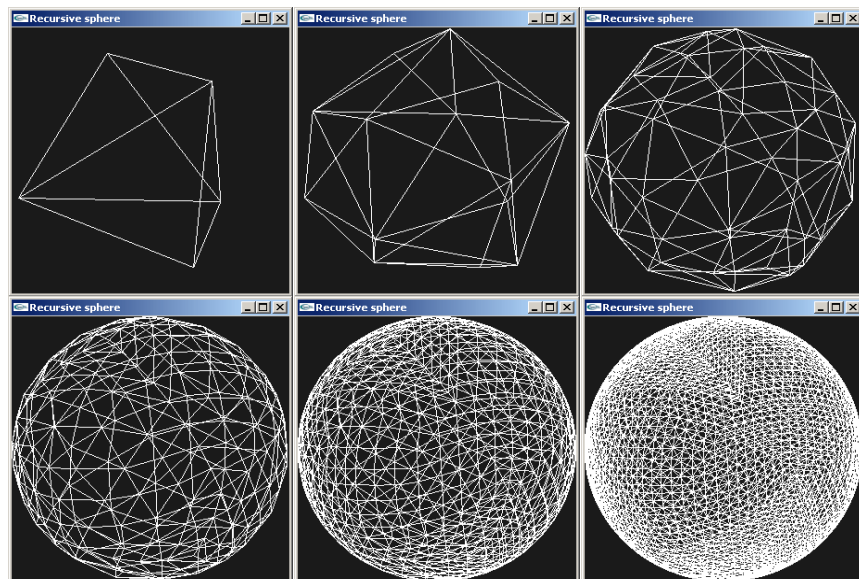


Subdivision of a triangle procedure (contd.)

```
#define HLP1 0.86603
#define HLP2 0.5

//two tetrahedra
TriangleT p1 = {{0,1,0},{0,0,1}, {HLP1, 0,-HLP2}};
TriangleT q1 = {{0,1,0},{HLP1,0,-HLP2},{-HLP1,0,-HLP2}};
TriangleT r1 = {{0,1,0},{-HLP1,0,-HLP2},{0,0,1}};

TriangleT p2 = {{0,-1,0},{0,0,1}, {HLP1, 0,-HLP2}};
TriangleT q2 = {{0,-1,0},{HLP1,0,-HLP2},{-HLP1,0,-HLP2}};
TriangleT r2 = {{0,-1,0},{-HLP1,0,-HLP2},{0,0,1}};
```



Subdivision of a triangle procedure (contd.)

advantages:

- nice and uniform tessellation
- equal triangles
- the depth we wish

disadvantages:

- triangle strips?

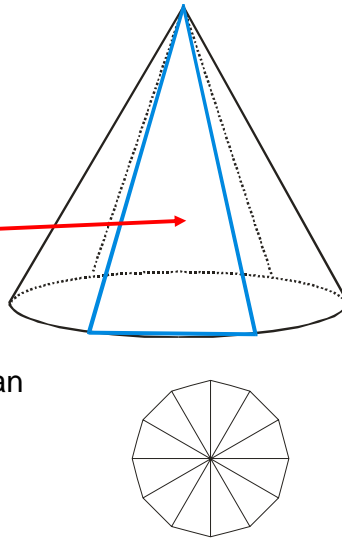


Circle to point ruled surface

idea:
make tessellation of a circle
and connect each line with
the apex
(as depicted here)

this leads to triangles

can be rendered as a triangle fan



The source code:

```
void Cone(GLfloat h, GLfloat rad){
    #define STEP 2*PI/10.0
    GLfloat angle;
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(0,h,0); //the apex of the cone
    for (angle=0;angle<=2*PI+STEP;angle+=STEP)
        glVertex3f(rad*sin(angle),0,rad*cos(angle));
    glEnd();
}
```

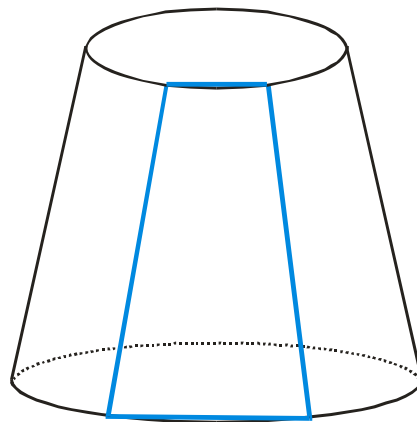


Circle to circle ruled surface

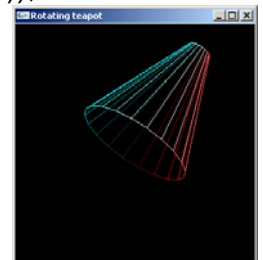
idea

quad strips connecting

the upper and the lower discs



```
void Cylinder(GLfloat baseRad, GLfloat topRad, GLfloat h){
    #define PI 3.141562
    #define K PI/10
    GLfloat t;
    glBegin(GL_QUAD_STRIP);
    for (t=0;t<=2*PI+K;t+=K) {
        glVertex3f(baseRad*sin(t),0,baseRad*cos(t));
        glVertex3f(topRad*sin(t),h,topRad*cos(t));
    }
    glEnd();
}
```





the first curve has equation $P(t)$
the second $S(t)$
the resulting surface has equation $Q(u,t)$

curves connecting $P(t)$ and $S(t)$ are lines,
so the

$$Q(u,t) = (1-u)P(t) + uS(t)$$

the parameter t must be uniform

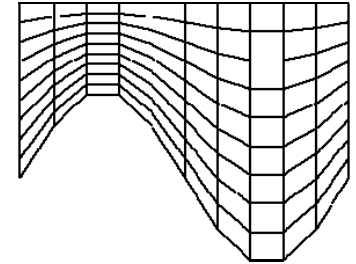


Example:

$$S(t) = [t, \sin(2\pi t), 1], P(t) = [t, 0, 0] \quad 0 \leq t \leq 1$$

$$Q(u,r) = (1-u)P(r) + uS(r) = [t-ut, 0, 0] + [ut, u \sin(2\pi t), u] = [t, u \sin(2\pi t), u]$$

```
void SinLine(){
  GLfloat t,u;
  for(t=0;t<=1;t+=K){
    glBegin(GL_QUAD_STRIP);
    for(u=0;u<=1;u+=K){
      glVertex3f(t,0.5*u*sin(t*2*PI),u);
      glVertex3f(t+K,0.5*u*sin((t+K)*2*PI),u);
    } //of for u
    glEnd();
  } //of for t
} //of procedure
```



Any curve that is rotated defines surface of revolution

Suppose the defining curve is planar and in xy

$$P(u) = [f(u), u, 0]; \quad 0 \leq u \leq 1$$

Equation of circle (in the plane xz):

$$S(v) = [r \sin(2\pi v), 1, r \cos(2\pi v)]$$

The equation of the surface of revolution is

$$Q(u,v) = [f(u) r \sin(2\pi v), u, f(u) r \cos(2\pi v)]$$

meaning:

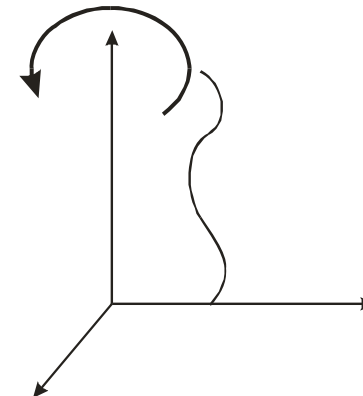
$f(u)$ modifies radius of the circle defined in xz



Example:

$$P(u) = 1 + 0.5 \sin(4\pi u)$$

$$Q(u,v) = [r \sin(2\pi v)(1 + 0.5 \sin(4\pi u)), u, r \cos(2\pi v)(1 + 0.5 \sin(4\pi u))]$$



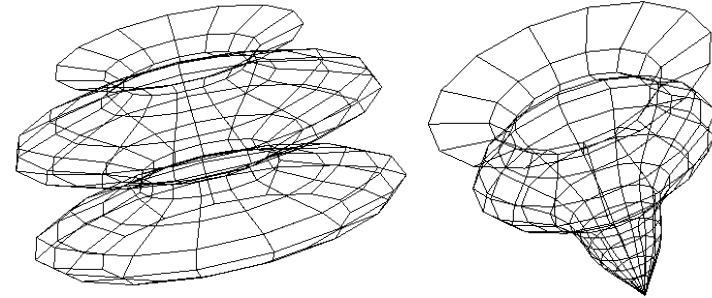


Source code

```
void SinRuled() {
    GLfloat u,v;
    for(v=0;v<=1;v+=K) {
        glBegin(GL_QUAD_STRIP);
        for(u=0;u<=1;u+=K) {
            glVertex3f(R*sin(v*2*PI)*(1.0+0.5*sin(u*4*PI)),
                u,
                R*cos(v*2*PI)*(1.0+0.5*sin(u*4*PI)));
            glVertex3f(R*sin((v+K)*2*PI)*(1.0+0.5*sin(u*4*PI)),
                u,
                R*cos((v+K)*2*PI)*(1.0+0.5*sin(u*4*PI)));
        } //of for u
        glEnd();
    } //of for v
}
```

(C) Bedrich Benes

53



$$f(u) = 1.0 + 0.5 \cdot \sin(u \cdot 4 \cdot \pi)$$

$$f(u) = u \cdot (1.0 + 0.5 \cdot u \cdot \sin(u \cdot 4 \cdot \pi))$$

(C) Bedrich Benes

54



Back face culling

each polygon has two sides
the *front face* and the *back face*

we can discard invisible faces before rasterization (Z-buffer!)

it is called *back face culling*

default OpenGL value:
both faces are rendered - time consuming!

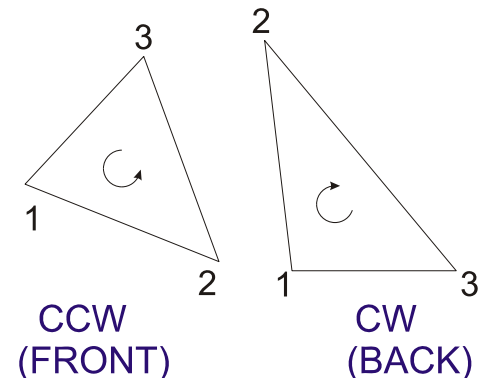
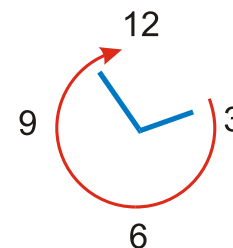
(C) Bedrich Benes

55



Polygon details - face culling (contd.)

- front face ~ projected vertices are in a counterclockwise order
- back face ~ vertices in clockwise order



(C) Bedrich Benes

56



Polygon details - face culling (contd.)

- what is culled can be set by special command

```
void glCullFace(GLenum mode);
```

where

mode is GL_FRONT or GL_BACK or GL_FRONT_AND_BACK

the face culling must be enabled by

```
glEnable(GL_CULL_FACE);
```

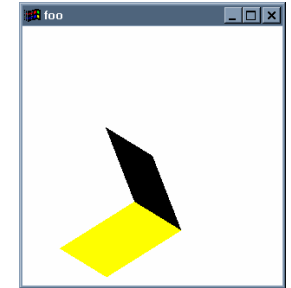
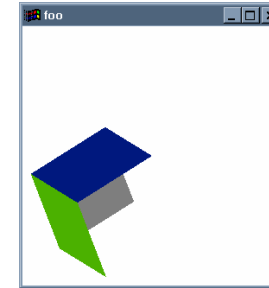
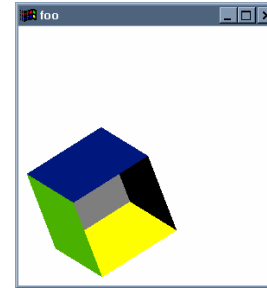
or can be disabled by

```
glDisable(GL_CULL_FACE);
```



Polygon details - face culling (contd.)

Warning!



```
glDisable(GL_CULL_FACE);
```

```
glEnable(GL_CULL_FACE);  
glCullFace(GL_BACK);
```

```
glEnable(GL_CULL_FACE);  
glCullFace(GL_FRONT);
```



Front and Back faces can be rendered in different modes

- determining the drawing mode of a face

```
void glPolygonMode(GLenum face, GLenum mode)
```

where

face is GL_FRONT or GL_BACK or GL_FRONT_AND_BACK

mode is GL_POINT or GL_LINE or GL_FILL

the polygon is either drawn as points, or outlined, or filled

- default:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)
```



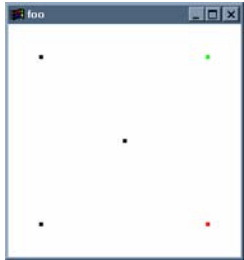
Front and Back faces can be rendered in different modes

example:

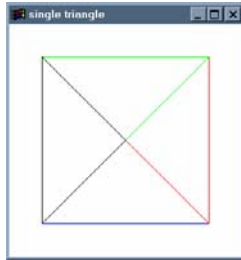
```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);  
glBegin(GL_TRIANGLE_FAN);  
    glColor3f(0.0,0.0,1.0); //color of first  
    glVertex3f(5, 5, 5);  
    glVertex3f(0, 0, 0);  
    glVertex3f(10, 0, 0);  
    glColor3f(1.0,0.0,0.0); // vertex and  
    glVertex3f(10, 10, 0); // color of next  
    glColor3f(0.0,1.0,0.0);  
    glVertex3f(0, 10, 0); // last  
    glColor3f(0.0,0.0,0.0);  
    glVertex3f(0, 0, 0);  
glEnd();
```



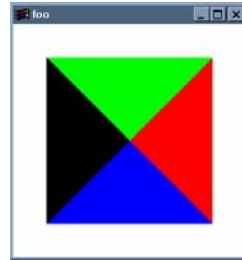
Front and Back faces can be rendered in different modes



GL_POINT



GL_LINE

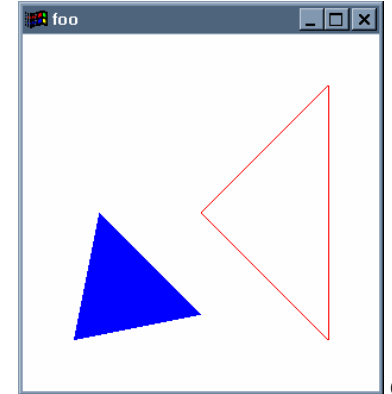


GL_FILL



Another example source code

```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
glBegin(GL_TRIANGLES);
glColor3f(0.0,0.0,1.0);
glVertex2f(0, 0);
glVertex2f(5, 1);
glVertex2f(1, 5);
glColor3f(1.0,0.0,0.0);
glVertex2f(5, 5);
glVertex2f(10, 10);
glVertex2f(10, 0);
glEnd();
```



Polygon details - face culling (contd.)

- changing the orientation

- a) change vertex order ☹️
- a) change polygon mode ☺️

```
void glFrontFace(GLenum mode);
```

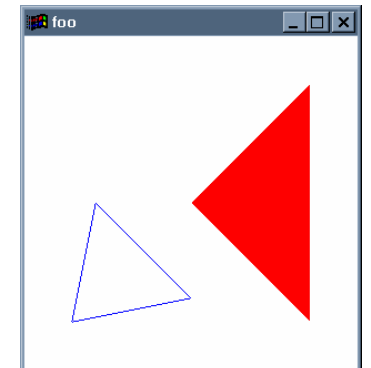
where

mode is GL_CCW (counterclockwise) or
 GL_CW (clockwise)



Another example source code (contd.)

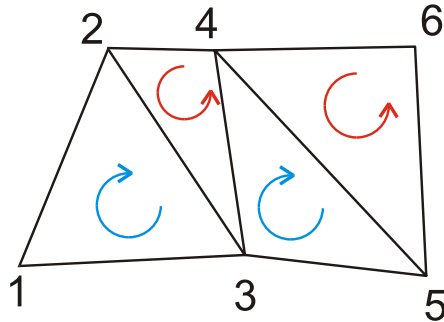
```
glFrontFace(GL_CW);
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
glBegin(GL_TRIANGLES);
glColor3f(0.0,0.0,1.0);
glVertex2f(0, 0);
glVertex2f(5, 1);
glVertex2f(1, 5);
glColor3f(1.0,0.0,0.0);
glVertex2f(5, 5);
glVertex2f(10, 10);
glVertex2f(10, 0);
glEnd();
```





Polygon details - face culling (contd.)

- GL_*_STRIP and GL_TRIANGLE_FAN orientation are automatically set according to the first object
- (if the first triangle in strip is CCW the others are CCW as well)



(C) Bedrich Benes

65



- OpenGL buffers
- OpenGL elements
- Attributes
- Back - front faces
- Back face culling
- Polygon rendering modes

(C) Bedrich Benes

66



- Jackie Neider, Tom Davis, Mason Woo
OpenGL Programming Guide,
Addison-Wesley Publication Company
ON LINE at <http://www.opengl.org.ru/docs/>
- www.opengl.org/developers/code/tutorials.html
- SIGGRAPH 2001
An Interactive Introduction To OpenGL Programming
www.opengl.org/developers/code/s2001/index.html
- SIGGRAPH '99
Lighting and Shading Techniques for Interactive Applications
www.opengl.org/developers/code/sig99/index.html

(C) Bedrich Benes

67