Master – Computer Science
University Paris-Sud

Open GL & GLSL Course

Curve and Surfaces Modeling

Christian Jacquemin

Matrices & Scene Description

(I)

Transformation Matrices in GLM

GLM Transformation Matrices

- C++ library based on OpenGL Shading Language (GLSL)
- Requires header(s) inclusion depending on which classes and functions are used

```
#include <glm.hpp>
#include <gtc/type_ptr.hpp>
#include <gtc/matrix_transform.hpp>...
```

- GTC extensions are stable
- GTX extensions are experimental

OpenGL Mathematics (GLM) Library

- view transformations (4x4 matrices):
  - **lookAt**: eye (point of view), center (target), up direction
  - **frustum**
  - **ortho**
  - **perspective** (angle of view in radians or degrees, aspect ratio, near and far)
    - in radians if GLM_FORCE_RADIANS is defined or degrees otherwise.

```
glm::mat4 Projection

= glm::perspective(45.0f, 4.0f/ 3.0f, 0.1f, 100.f);
```

OpenGL Mathematics (GLM) Library

- geometrical transformations (4x4 matrices):
  - translate
  - rotate
  - scale
- each transformation is defined with respect to a preceding transformation or with respect to Identity matrix otherwise

OpenGL Mathematics (GLM) Library

- geometrical transformations:
  - an example of 3 successive transformations

```
glm::mat4 ViewTranslate = glm::translate(
   glm::mat4(1.0f),// first view transformation:
                   // composed with identity
   glm::vec3(0.0f, 0.0f, -Translate));

glm::mat4 ViewRotateX = glm::rotate(
   ViewTranslate,  // second view transformation:
                   // composed with the preceding one
   angle_y, glm::vec3(-1.0f, 0.0f, 0.0f));
                   // angle and axis of rotation (x)

glm::mat4 View = glm::rotate(
   ViewRotateX,    // third view transformation:
                   // composed with the preceding one
   angle_x, glm::vec3(0.0f, 1.0f, 0.0f));
                   // angle and axis of rotation (y)
```

OpenGL Mathematics (GLM) Library

- other facilities: affineInverse, matrix access (row, columns)...
- wrapping up:
  - Model / View / Projection (MVP) transformation calculation from preceding examples
    (* : matrix product)

```
glm::mat4 Model = glm::scale(
   glm::mat4(1.0f),   // first model transformation:
                      // composed with identity
   glm::vec3(0.5f));  // scale uniformly 0.5 on
                      // all coordinates

glm::mat4 MVP = Projection * View * Model;
                      // matrix product
```

Matrices & Scene Description

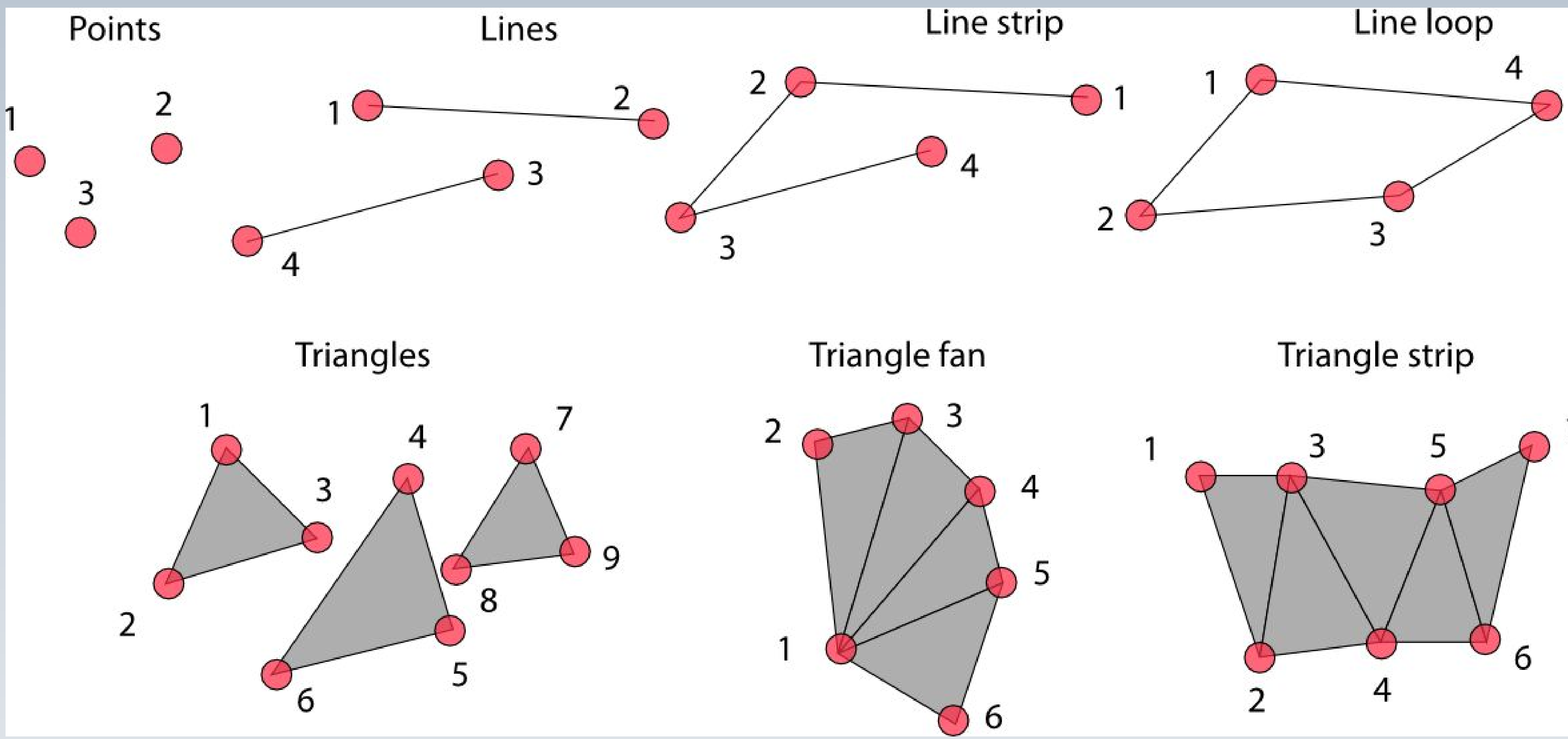
(2)

Shape Modeling with Curves

OpenGL primitives

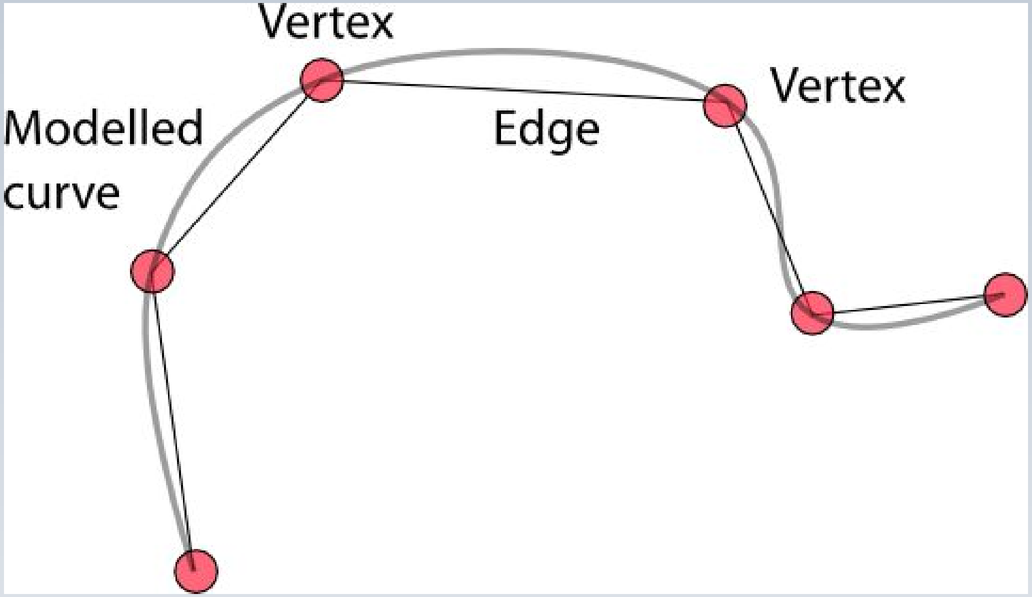
- Points GL_POINTS
- Lines GL_LINES
- Line Strips GL_LINE_STRIP
- Line Loops GL_LINE_LOOP
- Independent Triangles GL_TRIANGLES
- Triangle Strips GL_TRIANGLE_STRIP
- Triangle Fans GL_TRIANGLE_FAN

OpenGL primitives



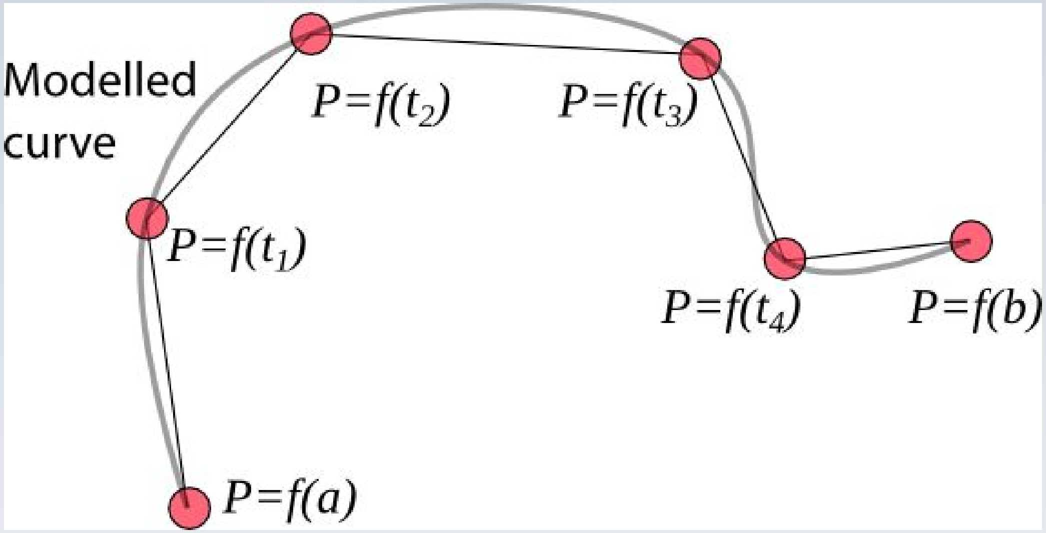
Curve Polygonization Basic Elements

- Vertices (the points)
- Edges (the segments building up the curve)



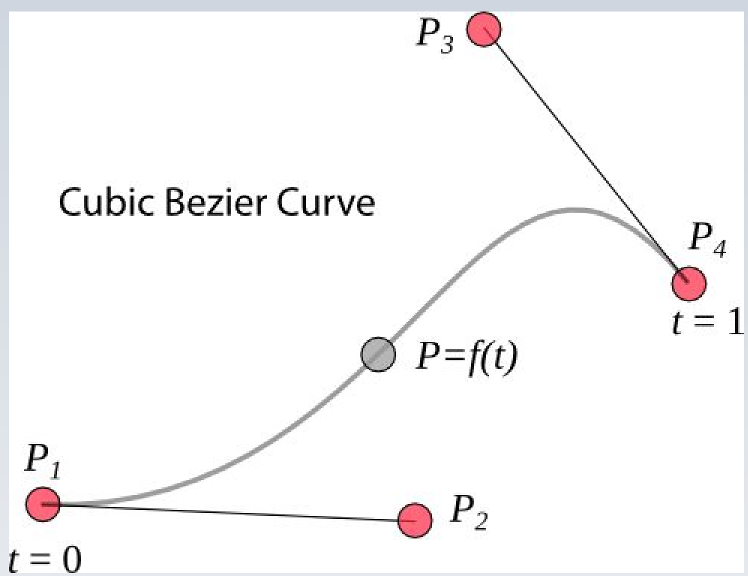
Parametric Curve Polygonization: Piecewise Linear Approximation

- Uniform Polygonization
  - $P_{xyz} = f(t) \quad t \text{ in } [a,b]$
  - The segment $[a,b]$ is divided into n segments of length b-a/n : $[a,t_1]$, $[t_1,t_2]$, ... $[t_{n-1},b]$
  - The curve is evaluated at a, t_1, t_2... t_{n-1}, b and the corresponding segments are drawn: P_0P_1, P_1P_2… $P_{n-1}P_n$



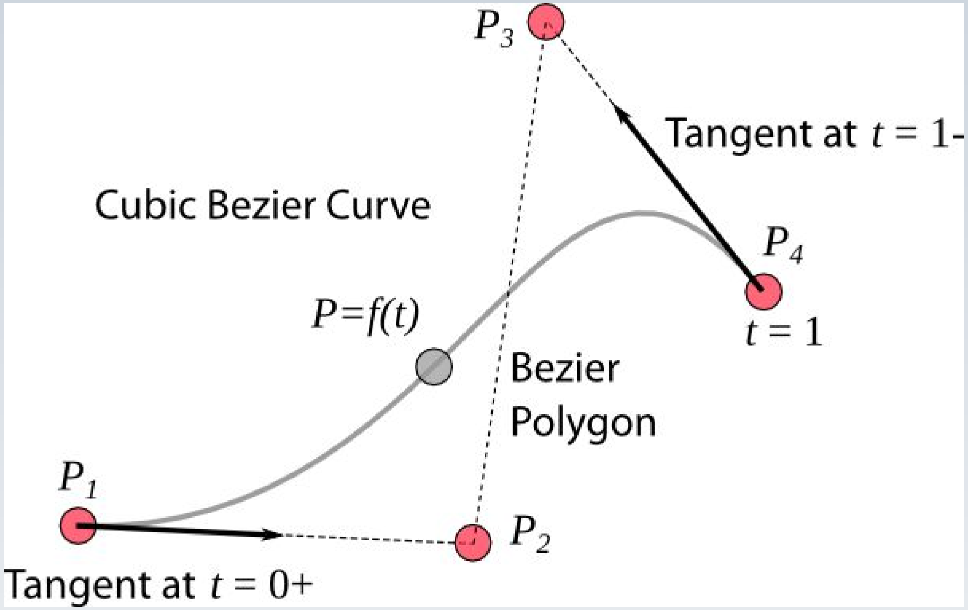
Bezier Curves

- A Bezier curve is a spline: a smooth function defined by control points
- Polynomial function of order n such as:
  - n=3: Cubic Bezier curves defined by 4 control points
  - n=2: Quadratic Bezier curves with 3 control points



Bezier Curves

- A Bezier curve has n control points $P_0 \ldots P_n$ which build the Bézier polygon
  - The curve begins at P_0 and ends at P_n
  - The curve is a line if and only if all the control points are collinear.
  - The start (resp. end) of the curve is tangent to the first (resp. last) edge of the Bezier polygon



Bezier Curves Equations

- Bezier curve general parametric equation

$$\mathbf{B}(t) = \sum_{i=0}^{n} b_{i,n}(t)\mathbf{P}_i, \quad t \in [0,1]$$

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n$$

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

From Wikipedia,
the free encyclopedia

- Quadratic and Cubic Bezier curve equations

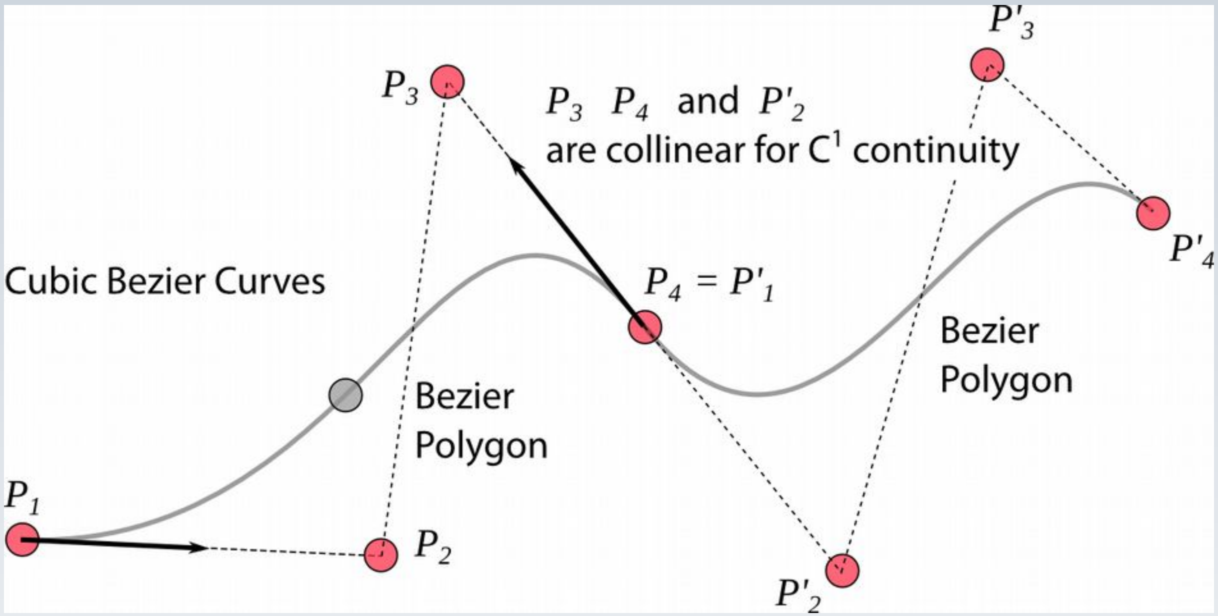
$$P_3(t) = (1 - t)^2 P_0 + 2\, t(1 - t)\, P_1 + t^2 P_2$$

$$P_4(t) = (1 - t)^3 P_0 + 3\, t^2(1 - t)\, P_1 + 3\, t(1 - t)^2 P_2 + t^3 P_3$$

Bezier Curves Polygonization

- The parameter t varies from 0 to 1:
  - for a n-edge polygon, use the n+1 parameter values

    i/n with i in $\{0, 1, .., n\}$
- Bezier curve can be used for approximating complex curves by joining together several Bezier curves

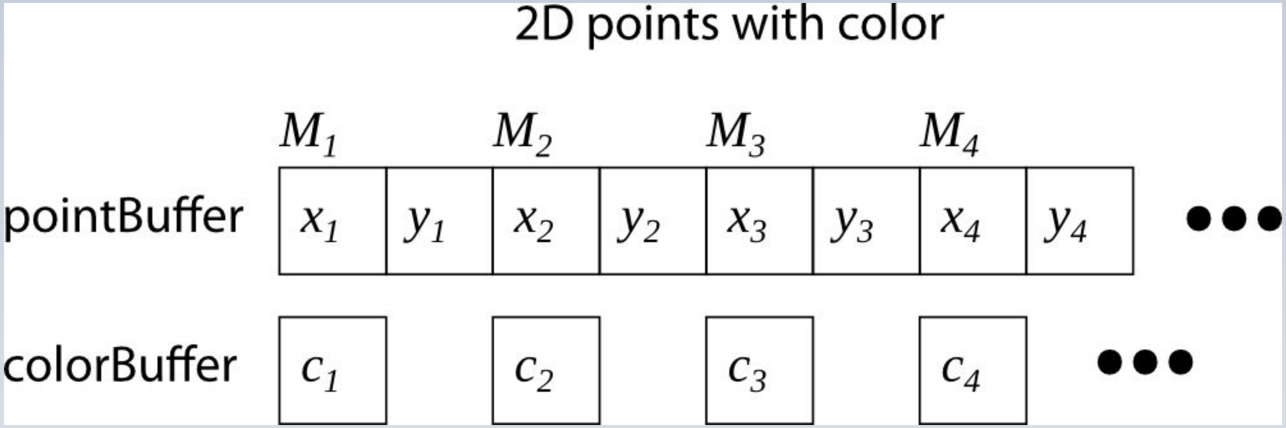


Bezier Curves Applications

- Fonts
- Computer Aided Modelling (were made popular during the 50s-60s in the automotive industry)
- Interpolation...
- Can be found in all the vector graphics tools
- There are several other types of splines with different properties (B-spline, T-spline...)

OpenGL Vertex Array Objects (VAO)

- Vertex stream for a colored curve
  - 2D vertices (nbEdges + 1) for the curve + 4 for the control polygon
  - 1D float color (one channel, here red)



→ VAOs store all the state data to render vertices

- format of the vertex data
- Vertex Buffer Objects (VBO)

OpenGL Vertex Array Object binding

- as for any OpenGL resource, requires acquiring an ID and binding

```
unsigned int vao = 0;
glGenVertexArrays (1, &vao);     // gets VAO ID
glBindVertexArray (vao);         // VAO binding
```

OpenGL Vertex Buffer Objects (VBO)

- a buffer object used to store vertex data
- `glBufferData` copies client memory (CPU) to server memory (GPU)
- vertex geometry data are defined by `glBufferData` after getting an ID and binding

```
unsigned int vbo = 0;
glGenBuffers(1, &vbo);  // gets an ID for the VBO
glBindBuffer(GL_ARRAY_BUFFER, vbo);  // binds it
glBufferData(GL_ARRAY_BUFFER, // data target
             2 * (nbEdges+1 + 4) * sizeof(float),
                              // data size
             pointBuffer,     // geometry data buffer
             GL_STATIC_DRAW); // data usage (w/wo modifs)
```

OpenGL Vertex Buffer Objects (VBO)

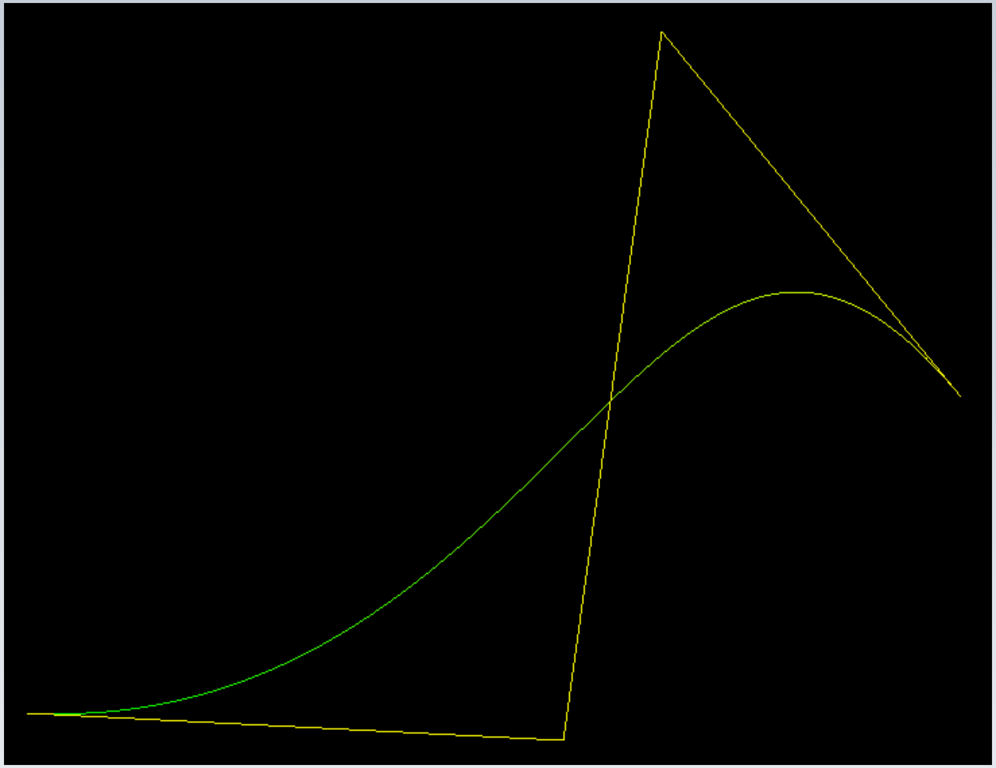
- vertex format defined by:

```
glVertexAttribPointer(
     0,                 // first parameter in
                        // the vertex shader
     2,                 // size: number of floats
                        // for this parameter
     GL_FLOAT,          // parameter type:
                        // here 32-bit float value
     GL_FALSE,          // optional data normalization
     0,                 // possible byte offset
                        // between 2 values (0=packed)
                        // allows for interleaved values
     (GLubyte*)NULL);   // offset to the first value
                        // (0 = initial value)

glEnableVertexAttribArray(0); // enables array access
                              // for the first param
```

OpenGL Rendering of a Bezier Curve + Bezier Polygon

- the vertex geometry and color VBOs store first the curve points and second the 4 control points



OpenGL Vertex Array Object drawing

- drawing of a VAO by using the VBOs which it is pointing to

```
glBindVertexArray (vao);            // VAO binding

// draws the curve
glDrawArrays(GL_LINE_STRIP,         // primitive
             0 ,                    // offset
             (nbEdges + 1) );       // size

// draws the control polygon
glDrawArrays(GL_LINE_STRIP,         // primitive
             (nbEdges + 1) ,        // offset
             4 );                   // size
```

GLSL Vertex Shader and its 2 attributes

- The vertex shader should have the 2 parameters with the structure defined by **`glVertexAttribPointer`**

```
in vec2 vp;           // 2-float position
in float color;       // 1-float color
```

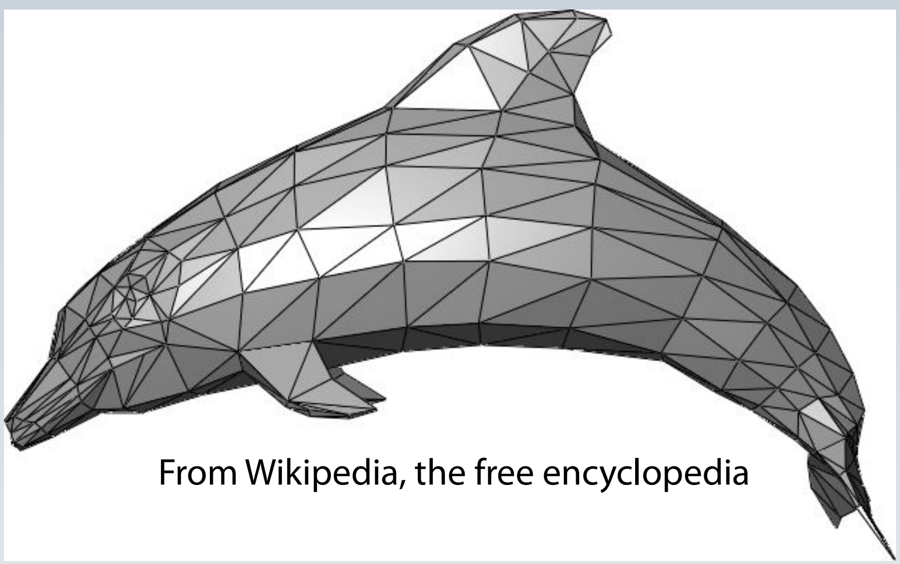
Matrices & Scene Description

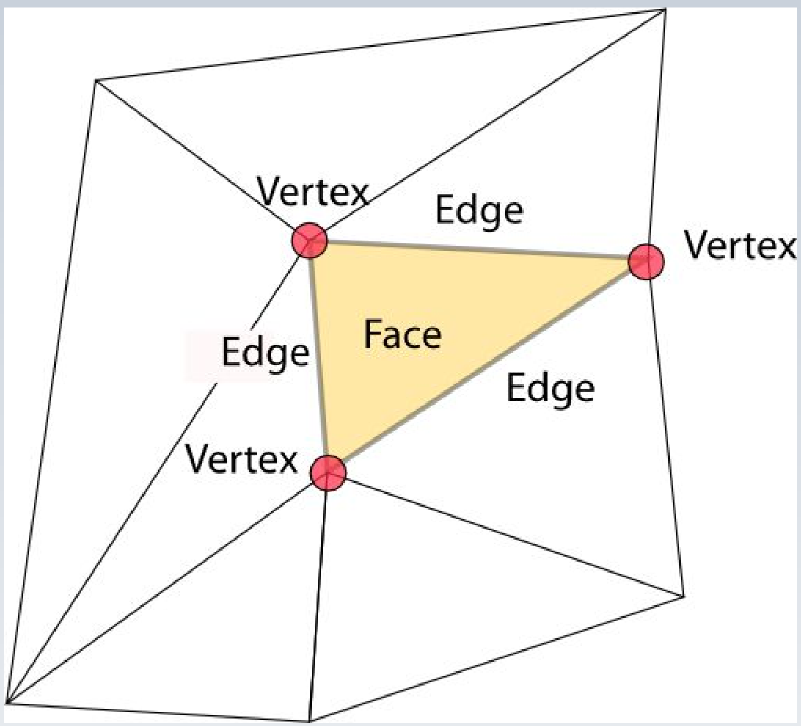
(3)

Shape Modeling with Meshes

Mesh (Surface Polygonization) Basic Elements

- Vertices (the points)
- Edges (the segments connecting the points)
- Faces (the triangles (or quads) building up the mesh)





Mesh Basic Elements

- In OpenGL, meshes are defined by faces which are in turn referring to vertices

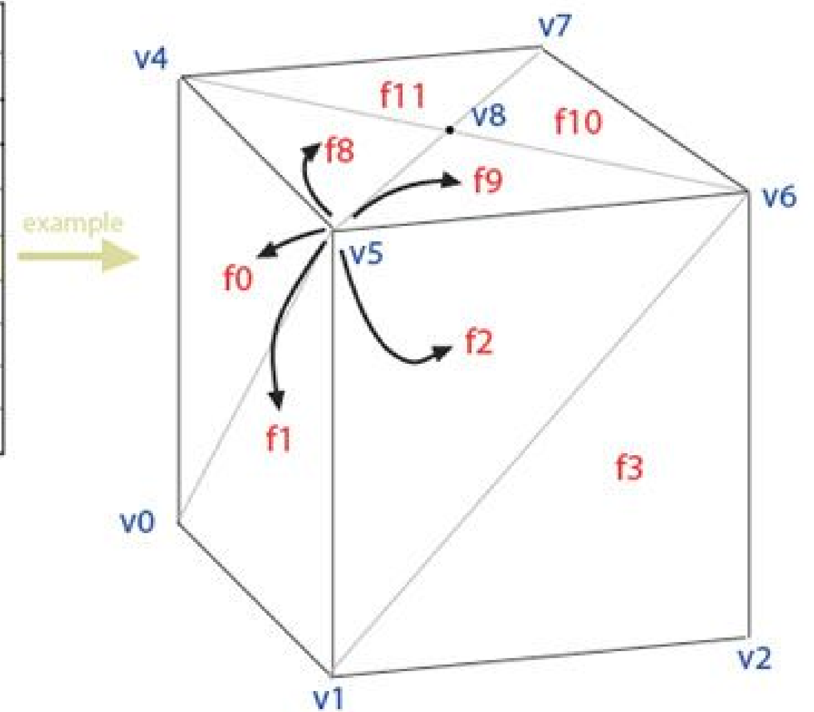
Face-Vertex Meshes

Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

Vertex List

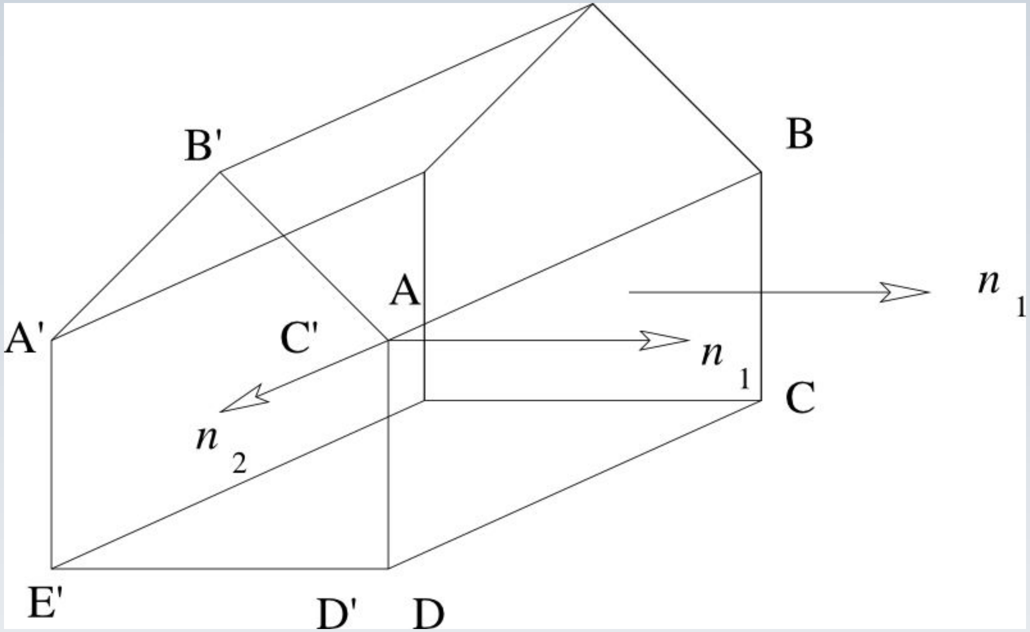
v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15



From Wikipedia, the free encyclopedia

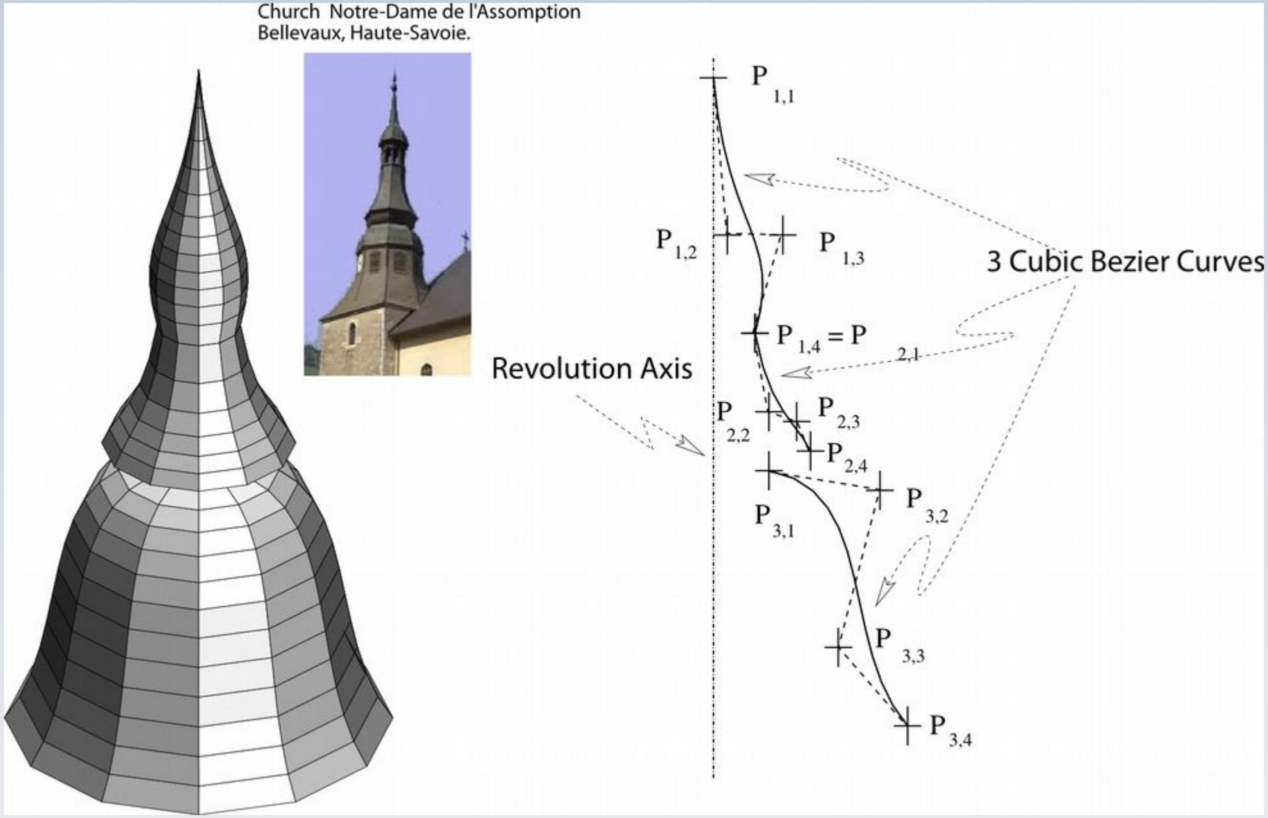
Mesh (Surface Polygonization)
Basic Elements

- for lighting, it is necessary to define normals
  - if attached to faces, results in flat shading
  - if attached to vertices, can be used for smooth shading
- normals can be computed through cross product for faces, and averaged over faces for vertex normals



Examples of Meshes: Revolution Surface

- A revolution surface in 3D is defined by
  - an axis (we use z as axis)
  - a curve in a plane containing the axis

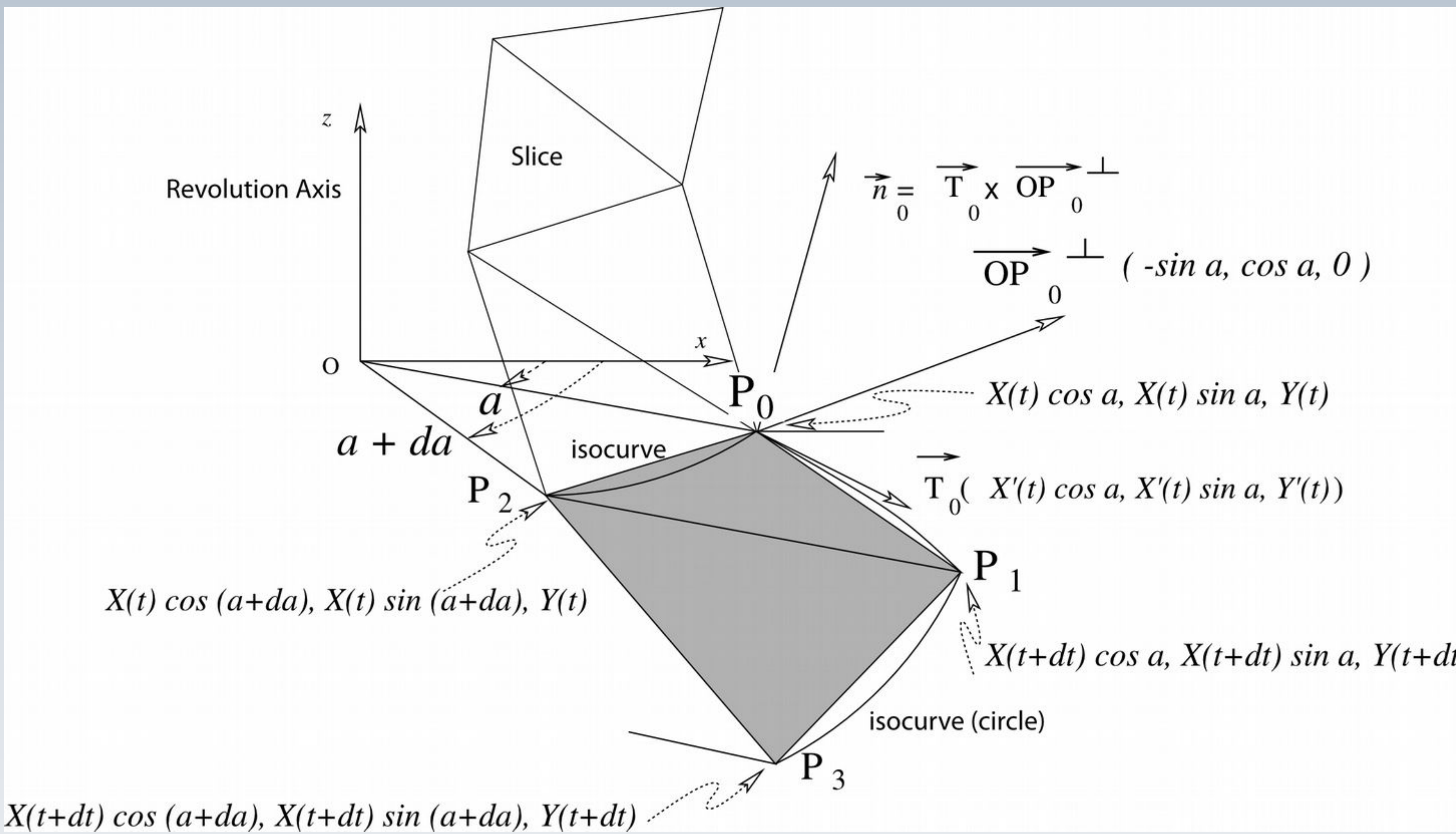


Examples of Meshes: Revolution Surface

- The polygonization of the surface is made by
  - **slices** (surface between two planes containing z)
  - and **stacks** (surface between two z isocurves (circles))
- Each slice or stack is rendered in OpenGL through a triangle strip (or a triangle fan around poles)
- For multiple triangle strips you can use
  - indexed rendering (index to vertices instead of vertices)
  - primitive restart: a special index that restarts a new primitive

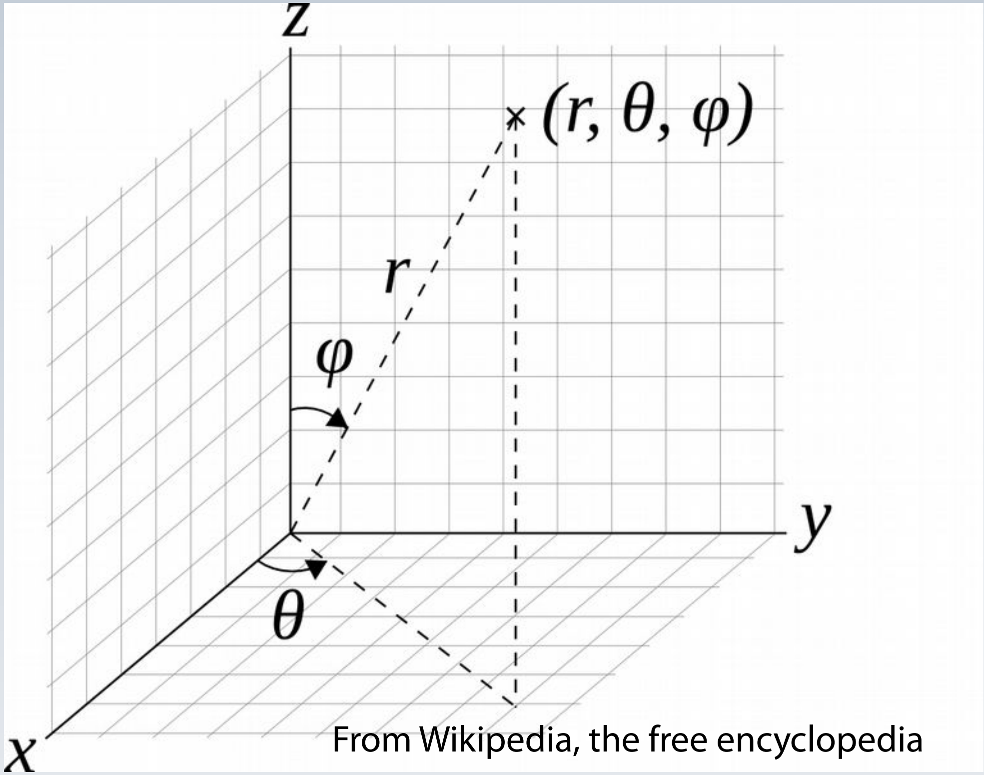
Revolution Surface polygonization

- Vertex 3D position and normals



Special cases of Revolution Surfaces

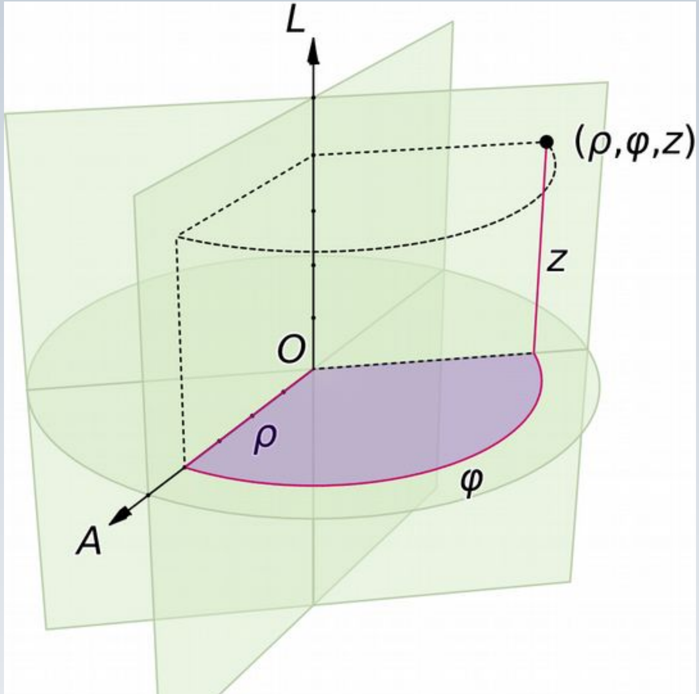
- **Sphere**: use spherical coordinates with angles as parameters: azimuthal angle θ, and polar angle φ.
  - $M(r,\theta,\varphi) = (r \cos\theta \sin\varphi, r \sin\theta \sin\varphi, r \cos\varphi)$



From Wikipedia, the free encyclopedia

Special cases of Revolution Surfaces

- **Cylinder**: use cylindrical coordinates with the angle as parameter:
  - $M(r,\theta,z) = (r \cos\theta, r \sin\theta, z)$
- One triangle strip can model the whole cylinder



From Wikipedia, the free encyclopedia

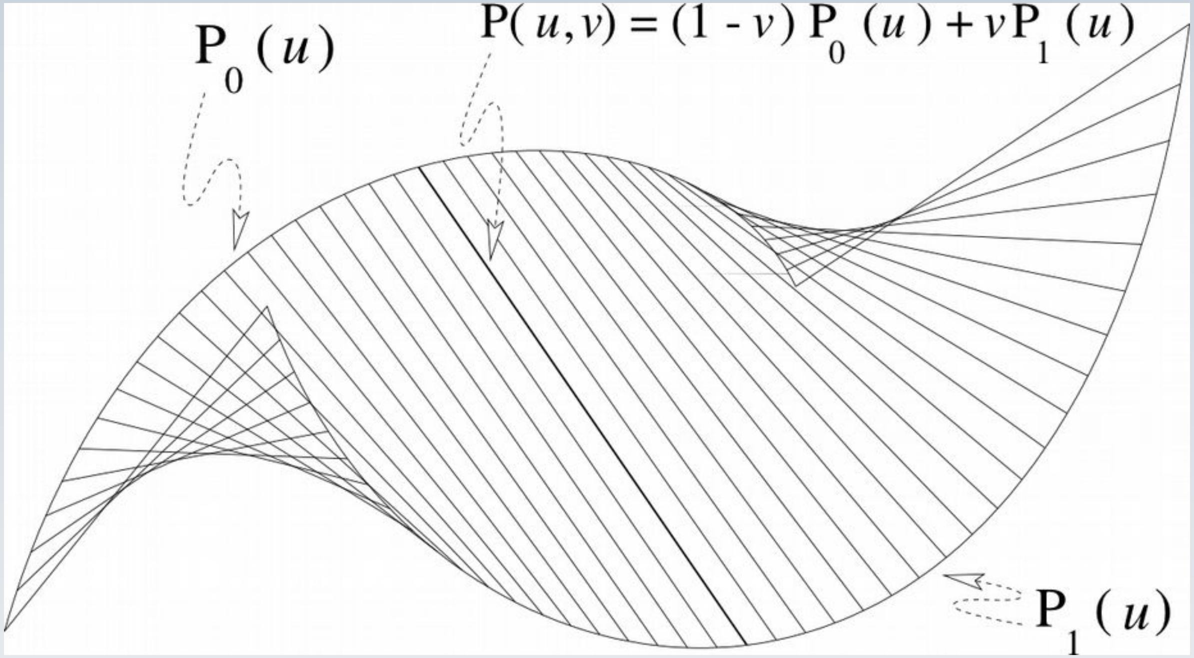
Special cases of Revolution Surfaces

- **Cone**: also use cylindrical coordinates for the base circle
- One triangle fan can model the whole cone
- Cone and cylinder are also 2 cases of **ruled surfaces**
  - the cone is defined by a point and a circle
  - the cylinder is defined by two circles

From Wikipedia, the free encyclopedia

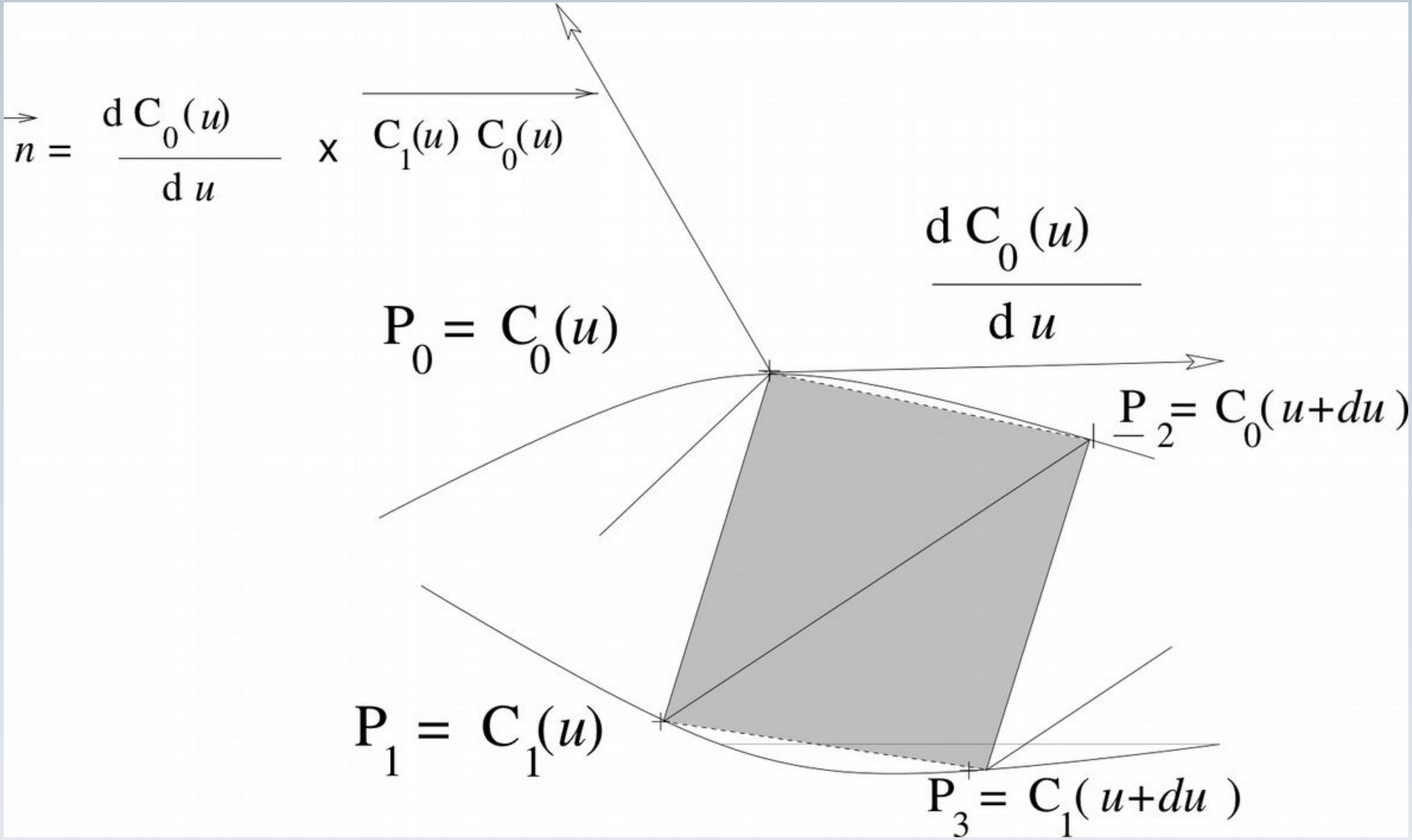
Ruled Surfaces

- Ruled surfaces are such that for every point on the surface there is a line that contains this point and that lies on the surface.
- We consider the case of ruled surfaces defined by two parametric curves



Ruled Surfaces polygonization

- Ruled surfaces can be polygonized with triangle strips joining the two curves.

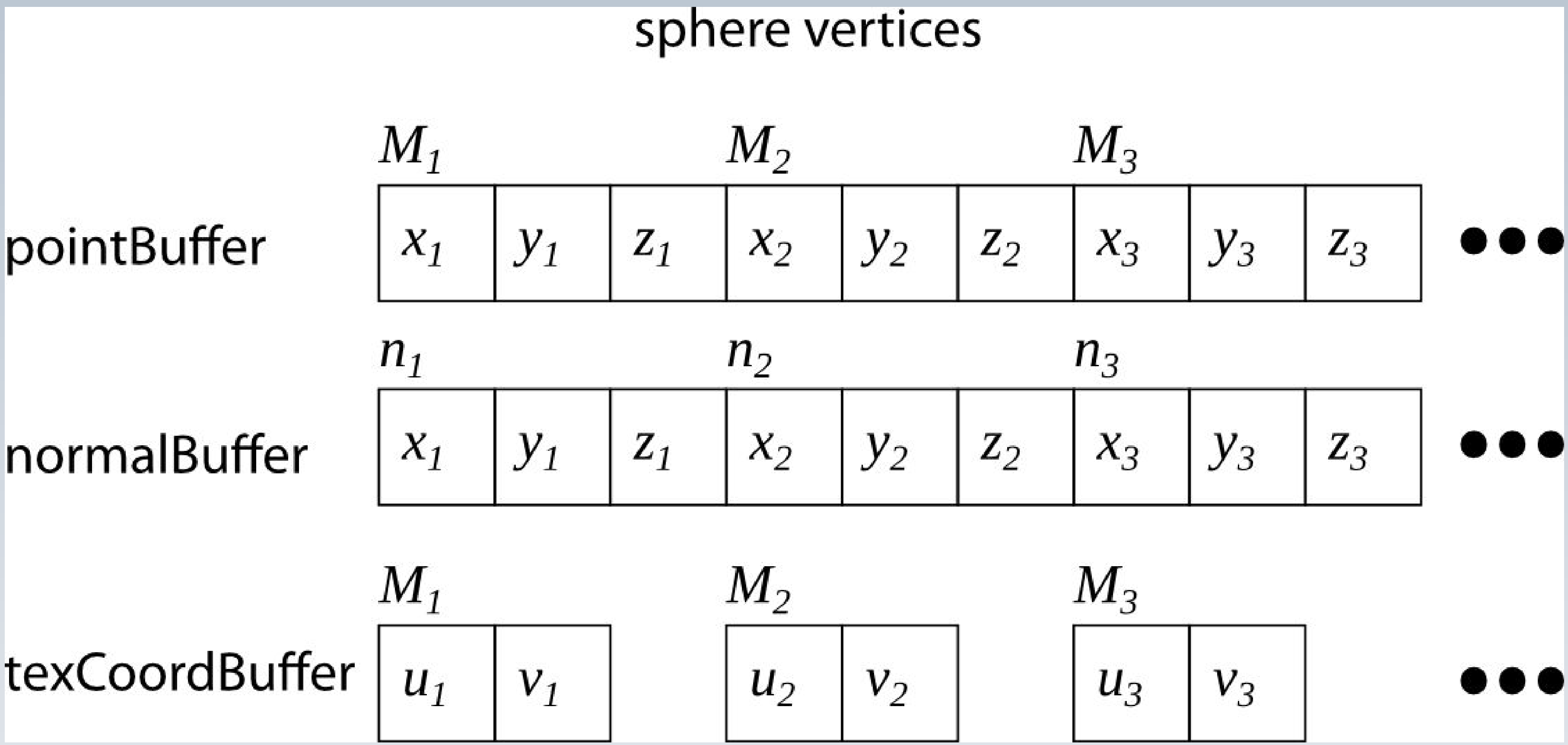


Modeled Surfaces polygonization

- Use Blender, export in Wavefront .obj format
- Parse file:
  - vertex coordinates table
  - normal coordinates table
  - texture coordinates table
  - faces made of 3 triplets of pointers to the preceding tables
- and perform indexed rendering of triangles
- caution: triangulate, apply transformations... before exporting

OpenGL sphere rendering

- data structure



OpenGL sphere rendering

- code for vertex geometry data (VBOs)

```
unsigned int vbo = 0;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, 3 * nbFaces * 3 * sizeof (float),
             pointBuffer, GL_STATIC_DRAW);

unsigned int vboNormals = 0;
glGenBuffers(1, &vboNormals);
glBindBuffer(GL_ARRAY_BUFFER, vboNormals);
glBufferData(GL_ARRAY_BUFFER, 3 * nbFaces * 3 * sizeof (float),
             normalBuffer, GL_STATIC_DRAW);

unsigned int vboTex = 0;
glGenBuffers(1, &vboTex);
glBindBuffer(GL_ARRAY_BUFFER, vboTex);
glBufferData(GL_ARRAY_BUFFER, 3 * nbFaces * 2 * sizeof (float),
             texCoordBuffer, GL_STATIC_DRAW);
```

OpenGL sphere rendering

- code for vertex format

```
// vertex positions are at location 0
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                      0, (Glubyte*)NULL);
glEnableVertexAttribArray(0);

// normal positions are at location 1
glBindBuffer(GL_ARRAY_BUFFER, vboNormals);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
                      0, (Glubyte*)NULL);
glEnableVertexAttribArray(1); // don't forget this!

// texture coordinates positions are at location 2
glBindBuffer(GL_ARRAY_BUFFER, vboTex);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
                      0, (Glubyte*)NULL);
glEnableVertexAttribArray(2); // don't forget this!
```

OpenGL sphere rendering

- code for geometry rendering

```
int nbStackTriangles = (nbSlices + 1) * 2;
// renders each stack as a triangle strip
for(int i = 0 ; i < nbStacks ; i++)
     glDrawArrays(GL_TRIANGLE_STRIP,
                  i * nbStackTriangles ,
                  nbStackTriangles);
```

- alternative (better technique):
  - indexed rendering
  - primitive restart at specific index value

OpenGL sphere rendering

- textured + ambient + diffuse lighting