CS 432 – Interactive Computer Graphics

Lecture 5 – Part 2

Parametric Surfaces

Surfaces

- We know how to create objects with flat faces
  - Specify vertices and make triangles/polygons from them.
- Often objects are *smooth/curved*
- One way to specify such objects is *parametrically*
  - Then we can discretize their values to create vertices

Parametric Representation

- Parametric representations allow us to trace out an object as a function of some parameter.
- We just did this with lines.
  - Given two endpoints P_0, P_1:
  $$P(u) = P_0 + u(P_1 - P_0)$$
  - We can also decompose this into x, y, z components.
  - Let $P_0.x$ be the x component of point P_0. Then we can define points $P(u) = \big(x(u), y(u), z(u)\big)$ as:
  $$x(u) = P_0.x + u(P_1.x - P_0.x)$$
  $$y(u) = P_0.y + u(P_1.y - P_0.y)$$
  $$z(u) = P_0.z + u(P_1.z - P_0.z)$$
  - For $0 \leq u \leq 1$

Parametric Representation

- You also did this in HW1 to determine the vertices of a circle:

$$For\ 0 \leq \theta \leq 360$$
$$y(\theta) = \sin(\theta)$$
$$x(\theta) = \cos(\theta)$$

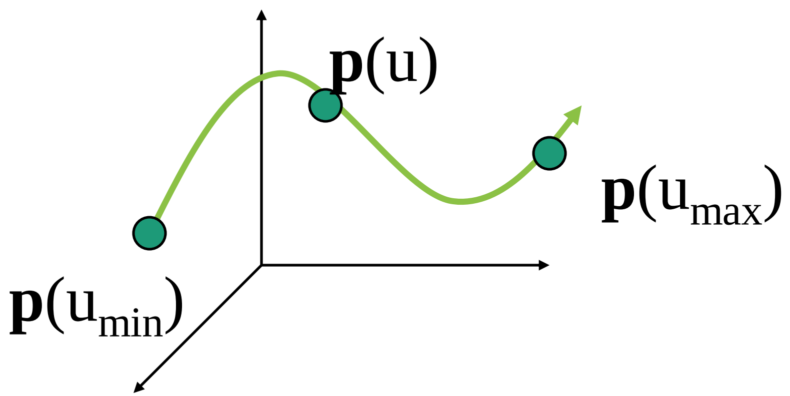
- Perhaps you did this for HW3 as well!?

Parametric Curves

- Moving to 3D, we can define a generic parametric curve as:

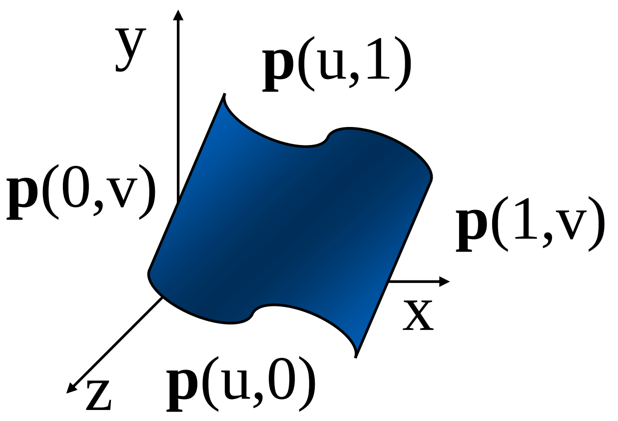
$$P(u) = [x(u), y(u), z(u)]$$

  - Where $x(u), y(u), z(u)$ are some functions
- Then to trace the curve we just vary $u_{min} \leq u \leq u_{max}$



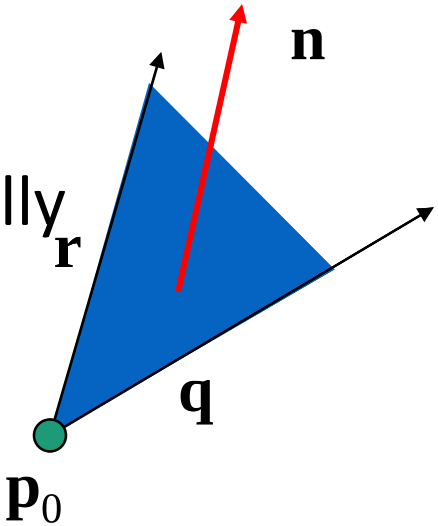
Parametric Surfaces

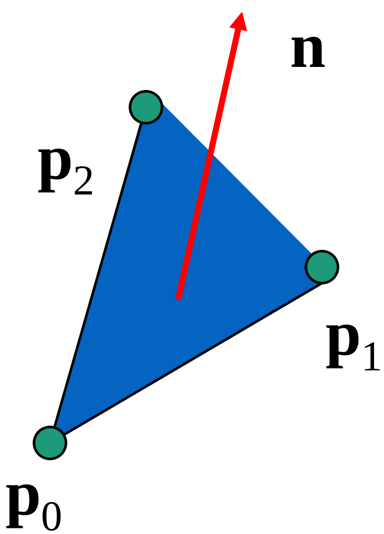
- 3D Surface requires 2 parameters
  - $x = x(u, v)$
  - $y = y(u, v)$
  - $z = z(u, v)$
  - $P(u, v) = [x\ y\ z]$



Parametric Planes

- We can find points on a plane parametrically depending on how the plane is specified:
- Point-vector form
  - $P(u, v) = P_0 + uq + vr$
- Three-point form
  - $q = P_1 - P_0$
  - $r = P_2 - P_0$
  - Then do point-vector form





Creating Vertices

- Simplest way to approximate a surface is to evaluate the polynomial at many points (parameter locations).
- For surfaces this forms an approximating mesh quadrilaterals each of which we can divide into triangles.
- So let's look at how we can make a parametric mesh for a cylinder!
  - The equations for this are
    - $x^2 + y^2 = 1$
    - For $0 \leq z \leq 1$

Creating Parametric Meshes

- Algorithm
  1. Define the grid we want to make our mesh on
     - Let's use the $x - z$ plane and evaluate values of y
  2. Define the resolution of the grid
     - Let's do 50 values along the x-axis and just the two (0,1) on the z-axis
     - Therefore 50*2 total vertices for the mesh
  3. Compute the 3rd dimensions value over the entire mesh according to the equation
  4. Use these vertices to make primitives (quads? triangles?)
     - We would need 50 quads, each defined by 4 points = 200 vertices
     - We would need 50*2 triangles, each defined by 3 points = 300 vertices
     - We can also make the decision to send all the vertices to the GPU or the unique set of vertices and the indices
       - Depends on how much repetition

Cylinder Code

```
//////CYLINDER////////
//Cyl = (x,y,z) where x^2+y^2=1, 0<=z<=1
//In other words a cylinder of radius 1 and height 1
//Mesh: Compute y on x-z plane as 50x2 vertices
//      -1<=x<=1, z={0,1}
//      Therefore 50*2 vertices
//Triangles:    There will be 50 "faces" each of which are rendered as 2 triangles, so total # triangle vertices = 50*2*3=300
//              Opt to use indices to vertices due to vertex redundancy


const GLuint numCylinderVertices = 50*2;
vec4 cylinderVertices[numCylinderVertices];
vec4 cylinderColors[numCylinderVertices];
GLubyte cylinderIndices[2*3*(numCylinderVertices/2)];
```

Cylinder Code: Create Vertices for the Mesh

```
void buildCylinder(){

    //create the vertices using equation x^2+y^2=1 for -1<=z<=1.
    //could also do this by varying 0<=theta<=380, using sin and cos

    index=0;

    //determine the step for the -1<=x<=1.  Need to span distance of 2 with our numCylinderVertices/4 vertices
    //(since we generate 4 vertices for each value of x)
    float step = 2.0/(numCylinderVertices/4.0);

    //make the vertices///////////////////
    //each x location creates 4 vertices:
    //(x,y,0), (x,y,1), (x,-y,0), (x,-y,1);
    for(float x=-1.0; x<1.0;x+=step){
        if(index>=numCylinderVertices/2)
            break;

        float y=sqrt(1-x*x);
        cylinderVertices[index]=vec4(x,y,0.0,1.0); cylinderColors[index] = cubeColors[index%8];
        cylinderVertices[index+1]=vec4(x,y,1.0,1.0); cylinderColors[index+1] = cubeColors[(index+1)%8];

        cylinderVertices[numCylinderVertices-index-2]=vec4(x,-y,0.0,1.0); cylinderColors[numCylinderVertices-index-2] = cubeColors[(numCylinderVertices-index-2)%8];
        cylinderVertices[numCylinderVertices-index-1]=vec4(x,-y,1.0,1.0); cylinderColors[numCylinderVertices-index-1] = cubeColors[(numCylinderVertices-index-1)%8];

        index+=2;
    }
```

Cylinder Code: Create Primitives from the Vertices

```
//assign the indices
//numCylinderVertices/2 quads will be made, each rendered as 2 triangles
//vertex i will generate a quad with vertices (i,i+2,i+3,i+1)
index=0;
int totalQuads = 0;
int bottomLeftVertex=0;
while(bottomLeftVertex+3<numCylinderVertices){
    //first triangle
    cylinderIndices[index]=bottomLeftVertex; index++;
    cylinderIndices[index]=bottomLeftVertex+2; index++;
    cylinderIndices[index]=bottomLeftVertex+3; index++;

    //second triangle
    cylinderIndices[index]=bottomLeftVertex; index++;
    cylinderIndices[index]=bottomLeftVertex+3; index++;
    cylinderIndices[index]=bottomLeftVertex+1; index++;

    totalQuads++;
    bottomLeftVertex+=2;

}
```

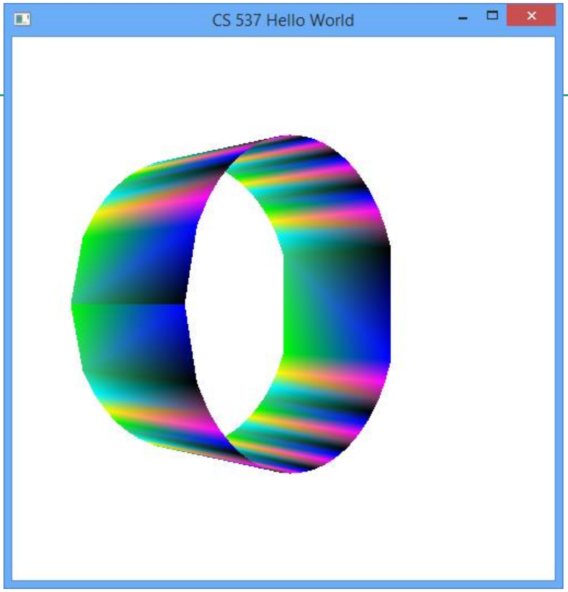
Display Primitives

```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );


    glUseProgram(program);
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES,(2*3*numCylinderVertices/2),GL_UNSIGNED_BYTE,BUFFER_OFFSET(0));

    glutSwapBuffers();


}
```



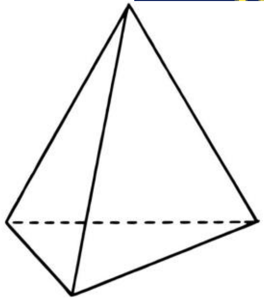
Parametric Sphere

- For the remaining assignments we're going to have spheres.
- We can parametrically find the vertices for a sphere as follows:
  - $x(\theta, \Phi) \; = \; r \cos\theta \sin\Phi$
  - $y(\theta, \Phi) \; = \; r \sin\theta \sin\Phi$
  - $z(\theta, \Phi) \; = \; r \cos\Phi$
  - For (if in degrees)
    - $0 \geq \theta \geq \; 360$
    - $0 \geq \Phi \geq 180$

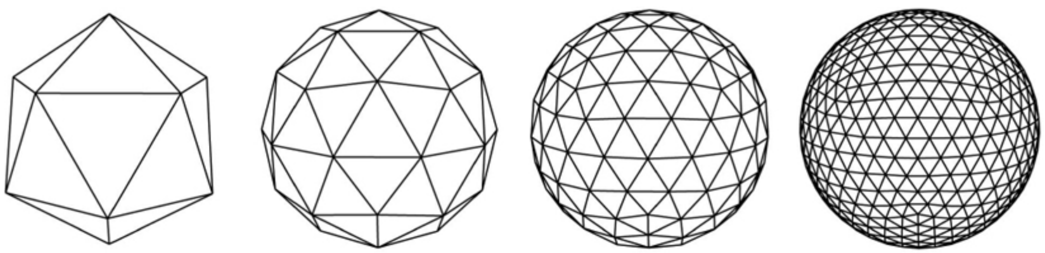
Example: Building a Sphere

- By fixing θ and varying Φ we get circles of constant longitude
- By fixing Φ and varying θ we get circles of constant latitude
- Together these can generate a set of vertices

Example: Building a Sphere



- Another way we can do this is with *recursive division*
- We first approximate the sphere as a tetrahedron whose vertices lie on the unit sphere
- We can get a closer approximation by subdividing each face of the tetrahedron into smaller triangles and move their new vertices onto the unit sphere



Example: Building a Sphere

- The first method is fast but has higher density of vertices at the poles
- The second is slower but generates triangles of the same size everywhere
  - And only need to pre-generate this model once (and use often!)
  - See Angel Appendix A, Problems 6 and 7

Example: Building a Sphere (header file)

```
#ifndef __SPHERE_H__
#define __SPHERE_H__
#include "Drawable.h"

class Sphere:public Drawable {
public:
    Sphere();
    void draw(Camera);
    ~Sphere();
private:
    //(4 triangular faces per tetrahedron)^(numDivisions+1)*3 vertices per triangle
    static const unsigned int numVertices = 3072;

    void build();
    unsigned int index;
    GLuint vpos, cpos, mmpos,vmpos,pmpos;

    vec4 vertexLocations[numVertices];
    vec4 vertexColors[numVertices];
    float sqrt2, sqrt6;
    void tetrahedron(int);
    void divideTriangle(vec4, vec4, vec4,int);
    void triangle(vec4, vec4, vec4);
    vec4 unit(vec4);

};
#endif
```

Example: Building a Sphere (Building)

```
void Sphere::tetrahedron(int count) {

    vec4 v[4] = {
        vec4(0,0,1,1),
        vec4(0,2 * sqrt2 / 3,-1.0f / 3.0f,1),
        vec4(-sqrt6 / 3.0f, -sqrt2 / 3.0f, -1.0f / 3.0f,1.0f),
        vec4(sqrt6 / 3.0f, -sqrt2 / 3.0f, -1.0f / 3.0f,1.0f)
    };

    //subdivide each of the 4 faces
    divideTriangle(v[0], v[1], v[2], count);
    divideTriangle(v[3], v[2], v[1], count);
    divideTriangle(v[0], v[3], v[1], count);
    divideTriangle(v[0], v[2], v[3], count);
```

```
void Sphere::build() {

    sqrt2 = (float)sqrt(2.0);
    sqrt6 = (float)sqrt(6.0);

    index = 0;

    tetrahedron(4);

    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexLocations) + sizeof(vertexColors), NULL, GL_STATIC_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertexLocations), vertexColors);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertexLocations), sizeof(vertexColors), vertexColors);
}
```

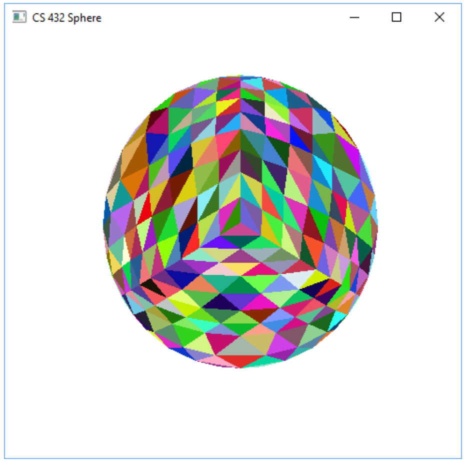
Example: Building a Sphere (Building)

```
void Sphere::divideTriangle(vec4 a, vec4 b, vec4 c, int count) {
    if (count>0) {
        vec4 v1 = unit(a + b);
        v1.w = 1.0;
        vec4 v2 = unit(a + c);
        v2.w = 1.0;
        vec4 v3 = unit(b + c);
        v3.w = 1.0;

        divideTriangle(a, v1, v2, count - 1);
        divideTriangle(c, v2, v3, count - 1);
        divideTriangle(b, v3, v1, count - 1);
        divideTriangle(v1, v3, v2, count - 1);
    }
    else
        triangle(a, b, c);

}
```

```
vec4 Sphere::unit(vec4 p) {
    float len = p.x*p.x + p.y*p.y + p.z*p.z;
    vec4 t;
    if (len > DivideByZeroTolerance) {
        t = p / sqrt(len);
        t.w = 1.0;
    }
    return t;
}
```




```
void Sphere::triangle(vec4 a, vec4 b, vec4 c) {
    vec4 color(1.0*rand() / RAND_MAX, 1.0*rand() / RAND_MAX, 1.0*rand() / RAND_MAX, 1.0);
    vertexLocations[index] = a;
    vertexColors[index] = color;
    index++;

    vertexLocations[index] = b;
    vertexColors[index] = color;
    index++;

    vertexLocations[index] = c;
    vertexColors[index] = color;
    index++;
}
```