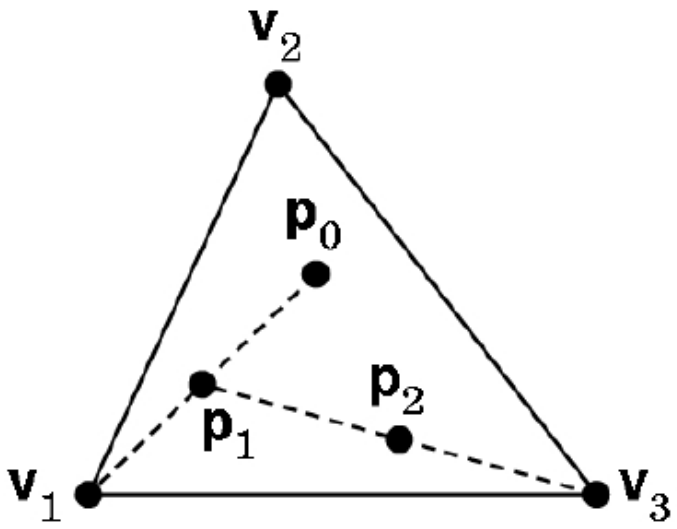
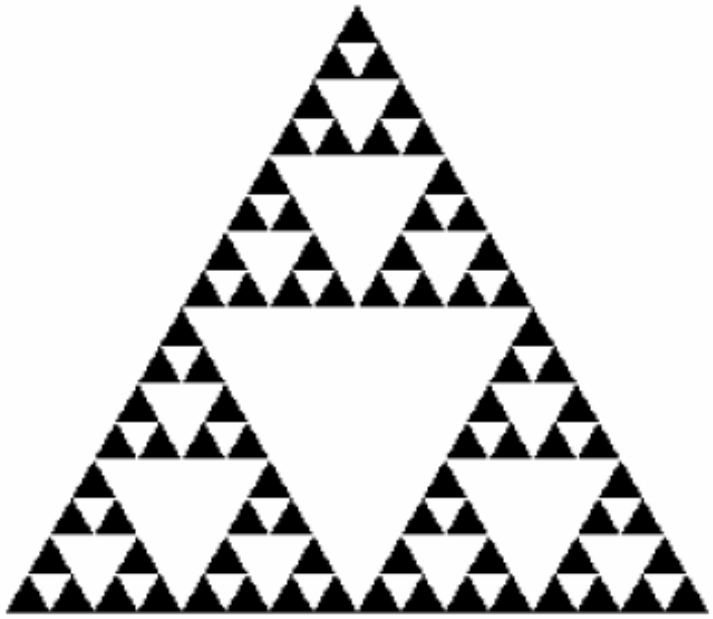
Graphics Programming

527970
Fall 2020
9/17/2020
Kyoung Shin Park
Computer Engineering
Dankook University

Sierpinski Gasket

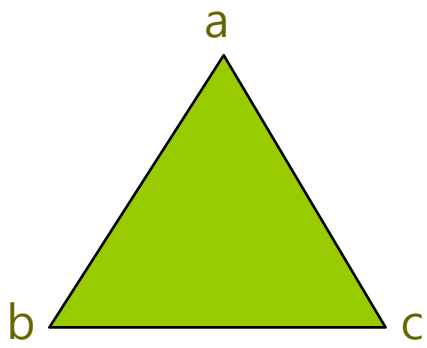
- A sample problem of drawing of the Sierpinski gasket
  1. Pick an initial point at random inside the triangle, P0
  2. Select one of the 3 vertices at random, v1
  3. Find the point halfway, P1
  4. Display this new point
  5. Replace the initial point with this new point
  6. Return to step 2



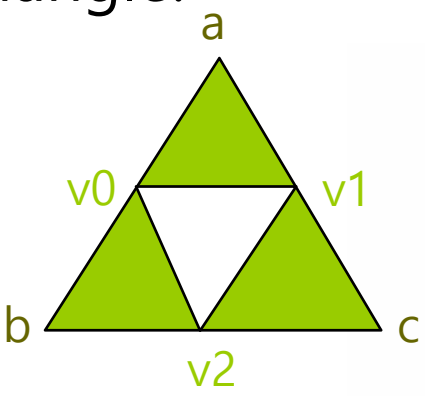


Sierpinski Gasket (2D)

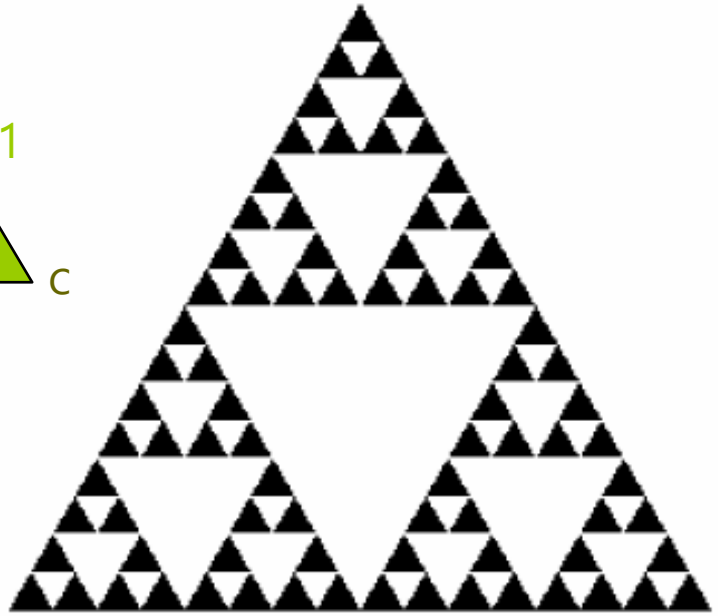
- Start with a triangle



- Connect the bisectors of each side of the triangle and remove the central triangle.



- Repeat



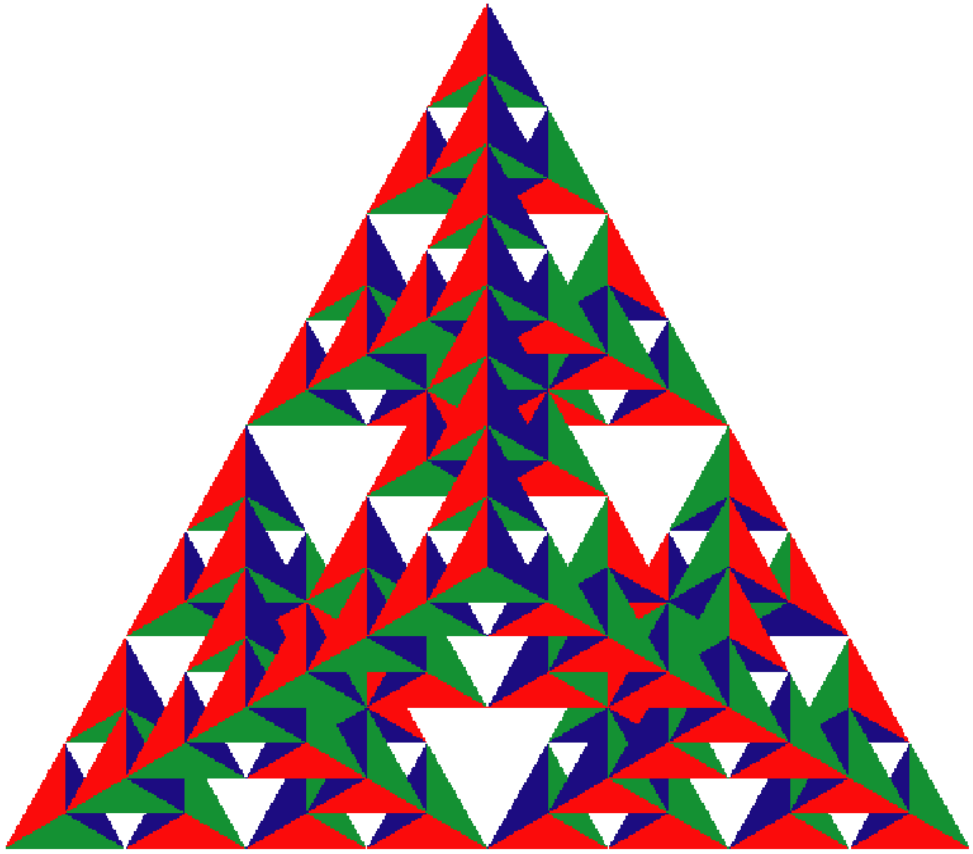
5 subdivisions

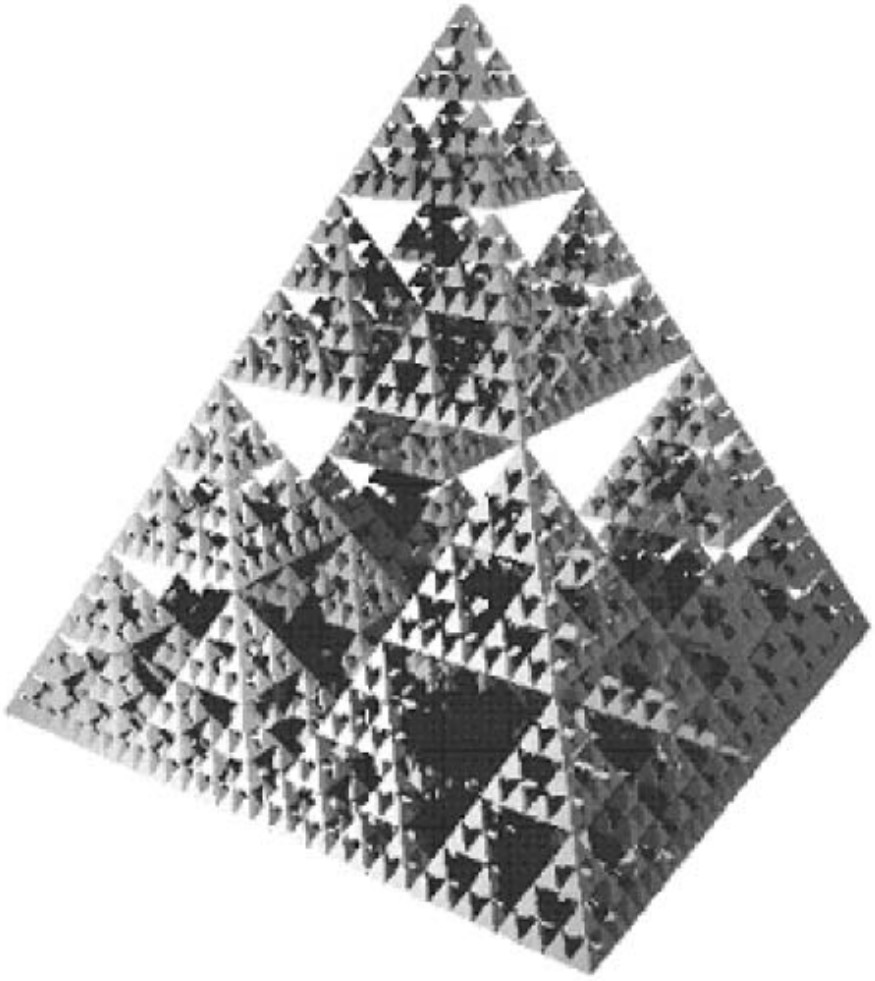
Sierpinski Gasket (2D)

```
/* recursive subdivision of triangle to form Sierpinski gasket*/
#include "Angel.h"
/* initial triangle */
vec2 v[3]={vec2(-1.0, -0.58), vec2(1.0, -0.58), vec2(0.0, 1.15)};
const int NumTimesToSubdivide = 5;
const int NumTriangles = 243; // 3^5
const int NumVertices = 3 * NumTriangles;
vec2 points[numVertices];
int Index = 0;

void triangle(vec2& a, vec2& b, vec2& c)
{ /* specify one triangle */
  points[Index++] = a;
  points[Index++] = b;
  points[Index++] = c;
}

void divide_triangle(vec2& a, vec2& b, vec2& c, int count)
{ /* triangle subdivision using vertex numbers */
  if(count>0)
  {
      vec2 v0=(a+b)/2;
      vec2 v1=(a+c)/2;
      vec2 v2=(b+c)/2;
      divide_triangle(a, v0, v1, count-1);
      divide_triangle(c, v1, v2, count-1);
      divide_triangle(b, v2, v0, count-1);
  }
  else triangle(a,b,c); /* draw triangle at end of recursion */
}
```

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}

void init()
{
    divide_triangle(v[0], v[1], v[2], NumTimesToSubdivide);
   // VAO & VBO ..
   // init the vertex position attribute from the vertex shader
    GLuint loc = glGetAttribLocation(program, "vPosition");
    glEnableAttribPointer(loc, 2, GL_FLOAT, GL_FALSE, 0, 0);
    glClearColor (1.0, 1.0, 1.0, 1.0);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Sierpinski Gasket");
    glewInit();
    init();
    glutDisplayFunc(display);
    glutMainLoop();
}
```

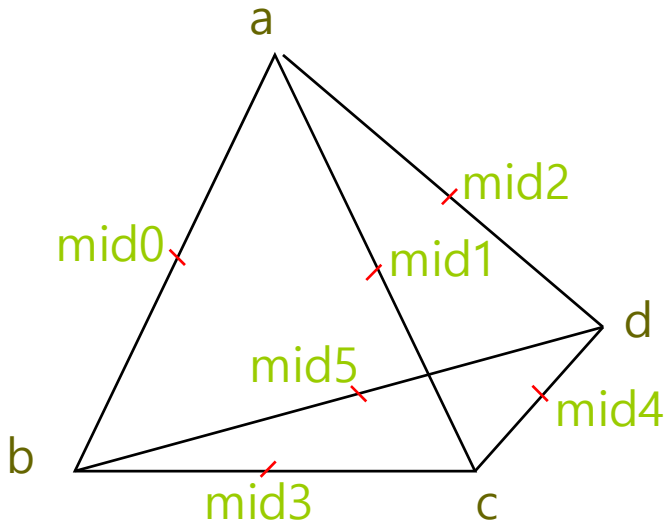
3D Gasket

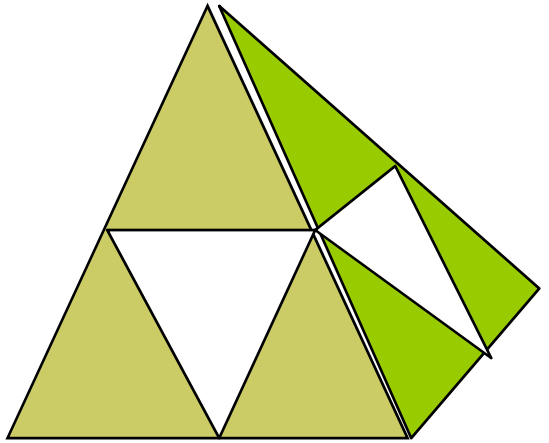




3D Gasket

- 3D Gasket is subdivision on each of the four faces.





- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedral.

3D Gasket

```
/* recursive subdivision of a tetrahedron to form 3D
Sierpinski gasket */
#include "Angle.h"
/* initial tetrahedron */
vec3 v[4]={vec3(0.0, 0.0, -1.0), vec3(0.0, 0.9428, 0.3333),
vec3(-0.8165, -0.4714, 0.3333), vec3(0.8165, -0.4714, 0.3333)};
vec3 base[4] = {vec3(1, 0, 0), vec3(0, 1, 0),
vec3(0, 0, 1), vec3(0, 0, 0)};
const int NumTimesToSubdivide = 5;
const int NumTetrahedrons = 1024; // 4^5
const int NumTriangles = 4*NumTetrahedrons;
const int NumVertices = 3 * NumTriangles;
vec3 points[numVertices];
vec3 colors[numVertices];
int Index = 0;

void triangle(vec3& a, vec3& b, vec3& c, int color)
{ /* specify one triangle */
   points[Index]=a; colors[Index]=base[color]; Index++;
   points[Index]=b; colors[Index]=base[color]; Index++;
   points[Index]=c; colors[Index]=base[color]; Index++;
}

void tetra(vec3& a, vec3& b, vec3& c, vec3& d)
{
   triangle(a, b, c, 0);
   triangle(a, c, d, 1);
   triangle(a, d, b, 2);
   triangle(b, d, c, 3);
}
```

```
void divide_tetra(vec3& a, vec3& b, vec3& c, vec3& d, int count)
{
   if(count>0)
   {
      /* compute six midpoints */
      vec3 v0 = (a+b)/2;
      vec3 v1 = (a+c)/2;
      vec3 v2 = (a+d)/2;
      vec3 v3 = (b+c)/2;
      vec3 v4 = (c+d)/2;
      vec3 v5 = (b+d)/2;

      /* create 4 tetrahedrons by subdivision */
      divide_tetra(a, v0, v1, v2, count-1);
      divide_tetra(v0, b, v3, v5, count-1);
      divide_tetra(v1, v3, c, v4, count-1);
      divide_tetra(v2, v4, d, v5, count-1);
   }
   else(tetra(a,b,c,d)); /* draw tetrahedron at end of recursion */
}
```

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}

void init()
{
    divide_tetra(v[0], v[1], v[2], v[3], NumTimesToSubdivide);
    // VAO & VBO ..
    // init the vertex position attribute from the vertex shader
    GLuint loc = glGetAttribLocation(program, "vPosition");
    glEnableAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
    // init the vertex color attribute from the vertex shader
    GLuint col = glGetAttribLocation(program, "vColor");
    glEnableAttribPointer(col, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(points)));
    glEnable(GL_DEPTH_TEST);
    glClearColor (1.0, 1.0, 1.0, 1.0);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("3D Gasket");
    glewInit();
    init();
    glutDisplayFunc(display);
    glutMainLoop();
}
```

Using Z-buffer algorithm

Using the Z-buffer

- Hidden surface removal
  - Z-buffer (depth buffer) algorithm
    - The value of the plane with the smallest z (depth) value of a geometry object in pixels is drawn on the screen.
    - Need the depth buffer that stores depth information per pixel
- Z-buffer algorithm
  - Initialize the depth buffer
    - glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
  - Enable depth test
    - glEnable(GL_DEPTH_TEST);
  - Clear the depth buffer in the display callback
    - glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

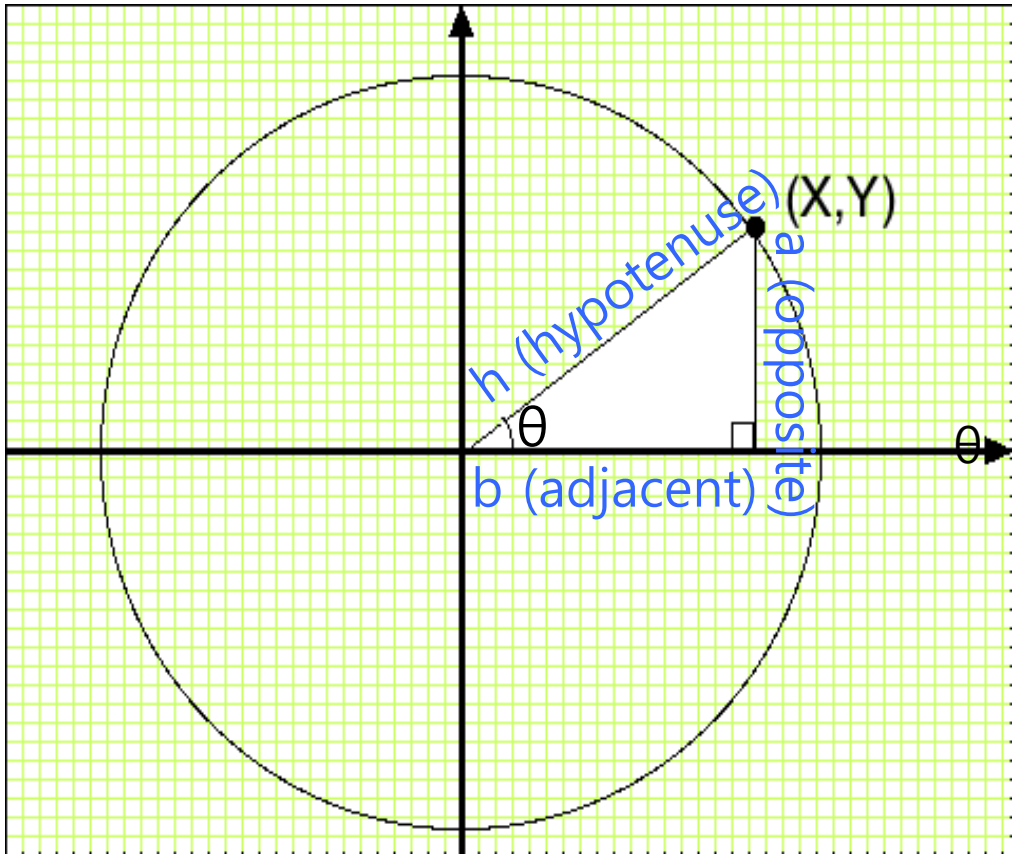
Geometric Primitives

Angles, Degrees, and Radians

- General math library functions uses radians.
- 360 degrees(°) = 1 full circle = 2 πradians
- 1 radian = 180.0/π degree ≈ 57.29578 degree
  or 1 degree = π/180.0 radian ≈ 0.01745329 radian

```
#ifndef M_PI
#define M_PI 3.141592654f
#endif
#define DegreesToRadians(degree) ((degree) * (M_PI / 180.0f))
#define RadiansToDegrees(radian) ((radian) * (180.0f / M_PI))
```

Trignometry



- $\sin\theta = a/h$
  $\cos\theta = b/h$
  $\tan\theta = a/b$
- $b = h*\cos\theta$
  $a = h*\sin\theta$
- $x^2 + y^2 = 1$
  $x = \cos\theta$
  $y = \sin\theta$
  $y/x = \sin\theta/\cos\theta = \tan\theta$
- $x = \text{distance} * \cos\theta$
  $y = \text{distance} * \sin\theta$

Trignometry

- Multiplicative inverse:
  $\csc\theta = 1/\sin\theta$
  $\sec\theta = 1/\cos\theta$
  $\cot\theta = 1/\tan\theta = \cos\theta/\sin\theta = x/y$
- Inverse:
  $\arcsin(x) = \sin^{-1}(x)$
  where $y=\arcsin(x)$ $x:[-1, 1] \rightarrow y:[-\pi/2, \pi/2]$
  $\arccos(x) = \cos^{-1}(x)$
  where $y=\arccos(x)$ $x:[-1, 1] \rightarrow y:[0, \pi]$
  $\arctan(x) = \tan^{-1}(x)$
  where $y=\arctan(x)$ $x:[-\infty, \infty] \rightarrow y:[-\pi/2, \pi/2]$

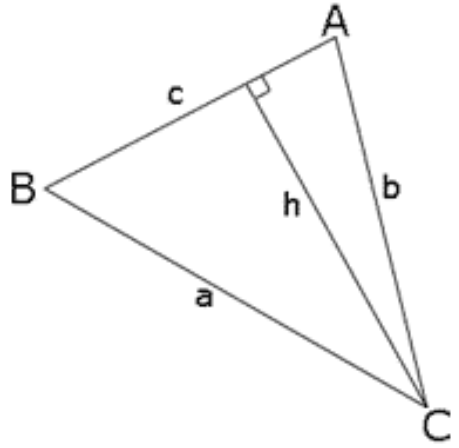
Trignometric Identity

- $\sin^2\theta + \cos^2\theta = 1$
  $1 + \tan^2\theta = \sec^2\theta$
  $1 + \cot^2\theta = \csc^2\theta$
- $\sin(\pi/2 - \theta) = \cos\theta$
  $\cos(\pi/2 - \theta) = \sin\theta$
  $\tan(\pi/2 - \theta) = \cot\theta$
- $\sin(x+y) = \sin x \cos y + \cos x \sin y$
  $\sin(x-y) = \sin x \cos y - \cos x \sin y$
  $\cos(x+y) = \cos x \cos y - \sin x \sin y$
  $\cos(x-y) = \cos x \cos y + \sin x \sin y$
- $\sin 2\theta = 2\sin\theta\cos\theta$
  $\cos 2\theta = \cos^2\theta - \sin^2\theta = 2\cos^2\theta - 1 = 1 - 2\sin^2\theta$

Law of Sines and Law of Cosines

- Law of sines

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

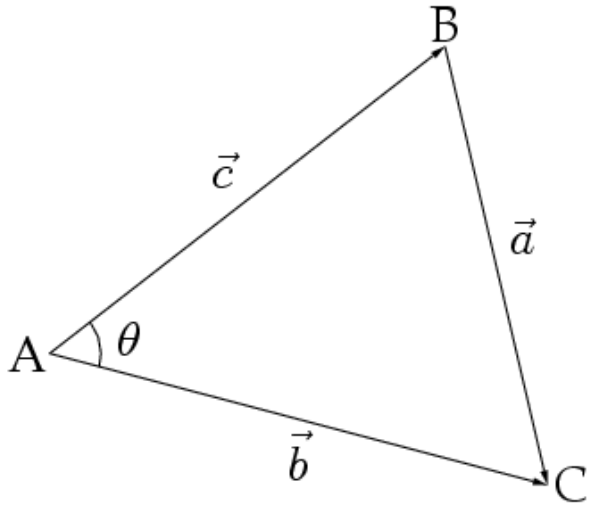


- Law of cosines

$$a^2 = b^2 + c^2 - 2bc\cos A$$

$$b^2 = a^2 + c^2 - 2ac\cos B$$

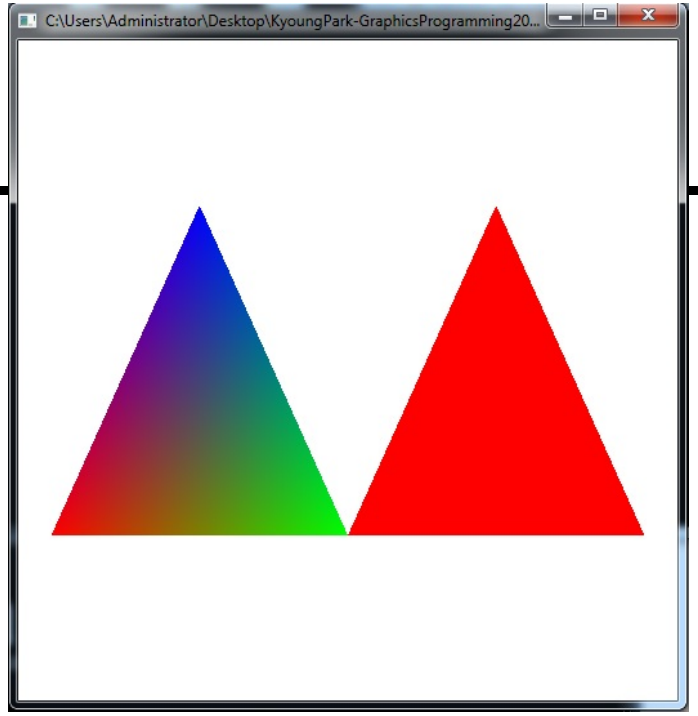
$$c^2 = a^2 + b^2 - 2ab\cos C$$



Geometric Primitives

- The most basic elements in expressing object
- In real-time graphics, linear primitives are used
  - Point
  - Line, Line Segment, Ray
  - Sphere, Cylinder, Cone
  - Cube (Box)
  - Triangle
  - Polygon, …
- Requirements for polygons in OpenGL
  - The polygon specified must **not intersect** itself.
  - Must be **convex.**
  - Its vertices are co-planar.

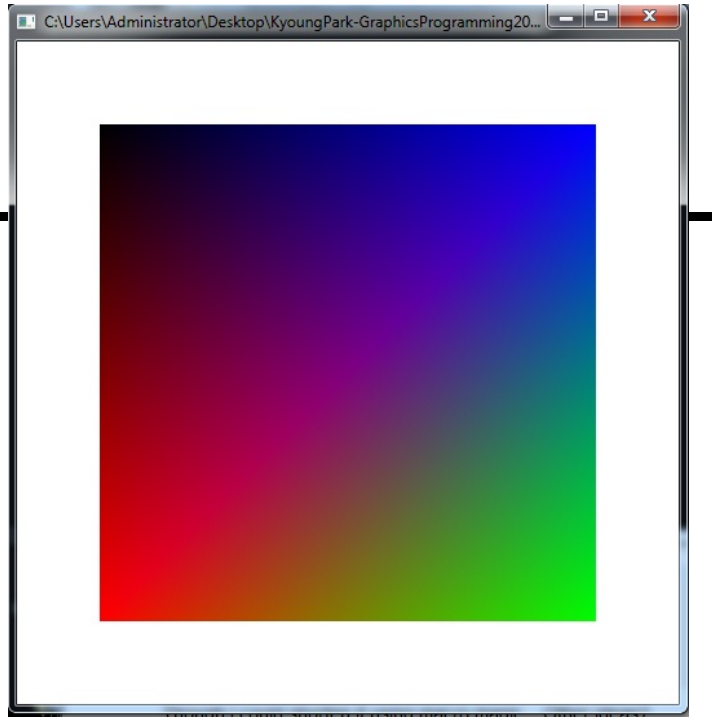
2 Triangles



- Draw 2 triangles
  (using color buffer or not)
  - GL_TRIANGLES

```
void setTriangles()
{
    std::vector<glm::vec4> vertexColors;
    std::vector<glm::vec4> vertexPositions;
    vertexColors.push_back(glm::vec4(1, 0, 0, 1)); // first triangle color
    vertexColors.push_back(glm::vec4(0, 1, 0, 1));
    vertexColors.push_back(glm::vec4(0, 0, 1, 1));
    vertexPositions.push_back(glm::vec4(-0.9, -0.5, 0, 1)); // first triangle
    vertexPositions.push_back(glm::vec4(0, -0.5, 0, 1));
    vertexPositions.push_back(glm::vec4(-0.45, 0.5, 1, 1));
    vertexPositions.push_back(glm::vec4(0, -0.5, 0, 1)); // second triangle
    vertexPositions.push_back(glm::vec4(0.9, -0.5, 0, 1));
    vertexPositions.push_back(glm::vec4(0.45, 0.5, 1, 1));
}
```

Square

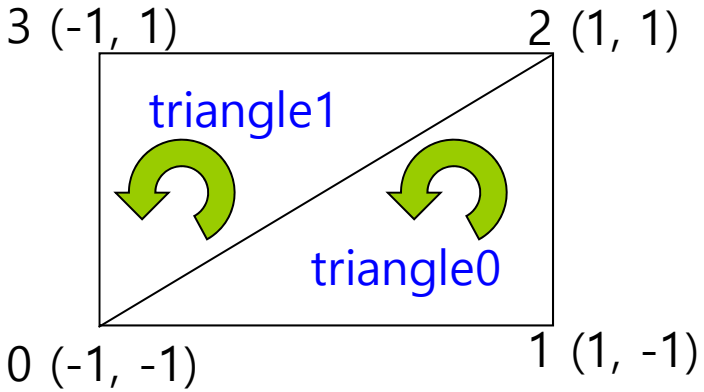


- Draw a square using **Index Buffer** & **glDrawElements(…)**
  - GL_LINE_LOOP & GL_TRIANGLES

```
void setSquare() {
    unsigned int Indices[6] = { 0, 1, 2, 0, 2, 3 };
    std::vector<glm::vec4> vertexColors;
    std::vector<glm::vec4> vertexPositions;
    vertexColors.push_back(glm::vec4(1, 0, 0, 1)); // square vertex color
    vertexColors.push_back(glm::vec4(0, 1, 0, 1));
    vertexColors.push_back(glm::vec4(0, 0, 1, 1));
    vertexColors.push_back(glm::vec4(0, 0, 0, 1));
    vertexPositions.push_back(glm::vec4(-0.75, -0.75, 0, 1)); // position
    vertexPositions.push_back(glm::vec4(0.75, -0.75, 0, 1));
    vertexPositions.push_back(glm::vec4(0.75, 0.75, 1, 1));
    vertexPositions.push_back(glm::vec4(-0.75, 0.75, 0, 1));
}
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Square

- Draw a square (2 triangles) using Vertex buffer



Vertex buffer

0	(-1, -1)
1	(1, -1)
2	(1, 1)
3	(-1, -1)
4	(1, 1)
5	(-1, 1)

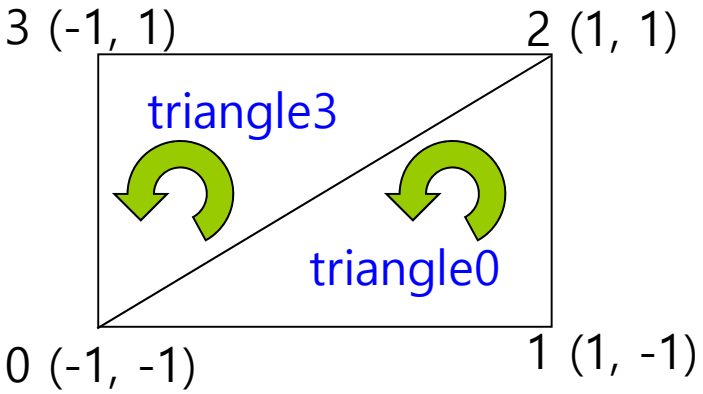
first=0

count=6

glDrawArrays(GL_TRIANGLES, 0, 6);

Square

- Draw a square (2 triangles) using Vertex buffer & Index buffer



	Index buffer
0	0
1	1
2	2
3	0
4	2
5	3

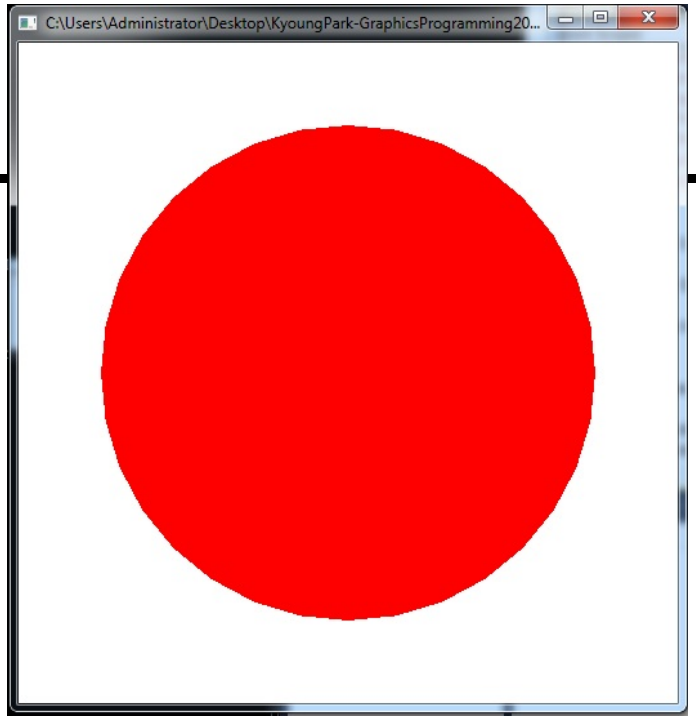
count=6

Vertex buffer
(-1, -1)
(-1, 1)
(1, 1)
(1, -1)

glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

Circle

- Draw a wireframe circle
  - GL_LINE_LOOP
- Draw a solid circle
  - GL_TRIANGLE_FAN

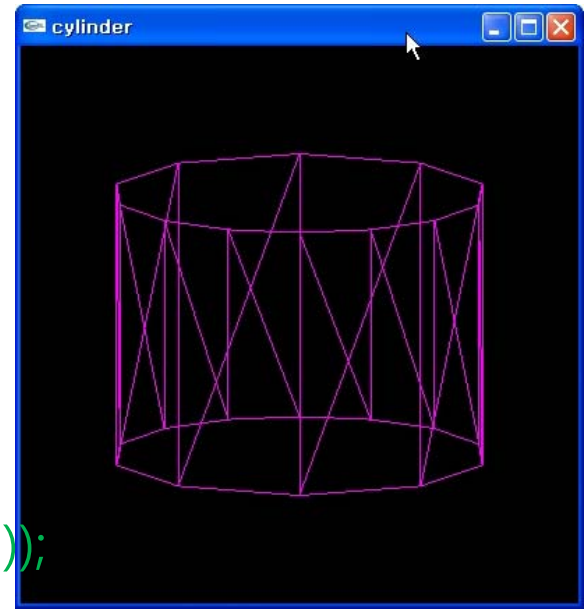



```
std::vector<glm::vec4> circleVertices;
void setCircle(float radius, int step)
{
    float x, y, theta;
    theta = (float) (2*M_PI/step);
    for (int i=0; i<step; i++) {
        x = radius * cos(theta * i);
        y = radius * sin(theta * i);
        circleVertices.push_back(glm::vec4(x, y, 0, 1));
    }
}
```

Cylinder

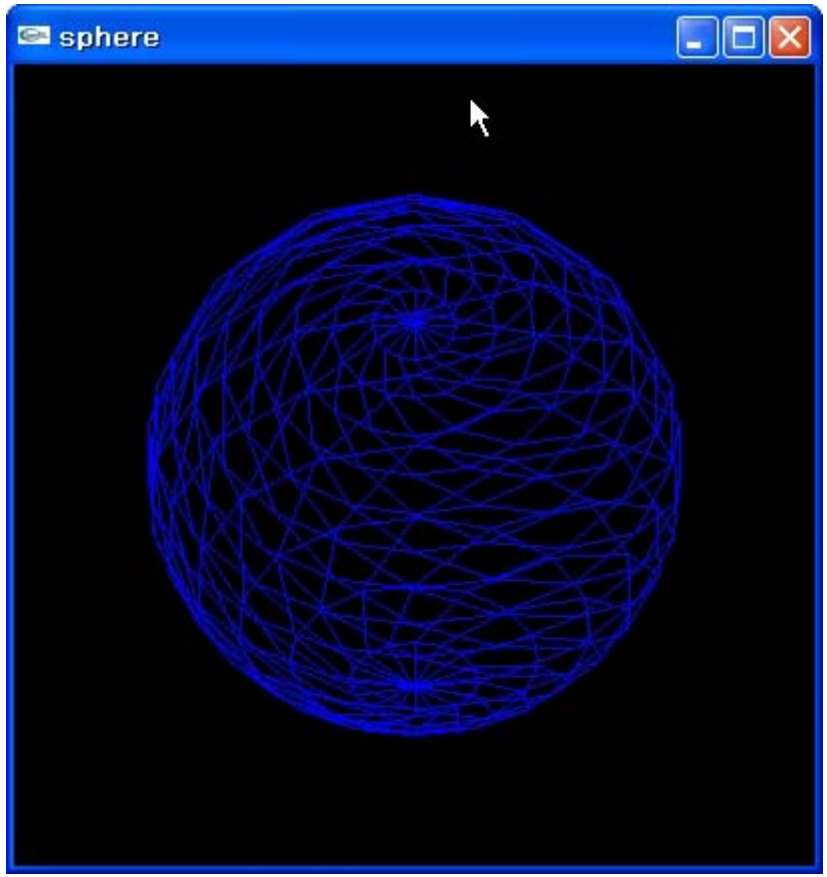
- Draw a wireframe cylinder
  - glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
- Draw a solid cylinder
  - glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)

```
std::vector<glm::vec4> cylinderVertices;
void setCylinder(float h, float r, int step)
{
    float theta = (float) (2*M_PI/step);
    for (int i=0; i<=step; i++) {
        float x = r * cos(theta * i);
        float y = -h/2;
        float z = r * sin(theta * i);
        cylinderVertices.push_back(glm::vec4(x, y, z, 1));
        y = h/2;
        cylinderVertices.push_back(glm::vec4(x, y, z, 1));
    }
}
```



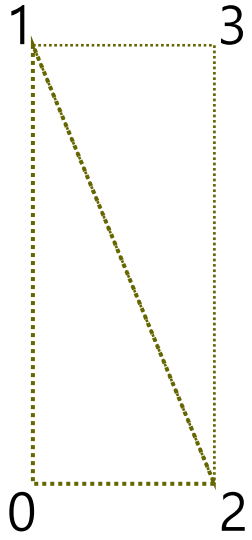
Sphere

- Draw a wireframe sphere
  - GL_LINE_STRIP
- Draw a solid sphere
  - GL_TRIANGLE_STRIP



Sphere

```
std::vector<glm::vec4> sphereVertices;
void drawSphere(float radius, int stacks, int slices)
{
    float lonstep = M_PI/stacks; float latstep = M_PI/slices;
    for (lon = 0.0; lon<=2*M_PI; lon+= (lonstep)) {
        for (lat=0.0; lat<=M_PI+latstep; lat += (latstep)) {
            x = cosf(lon)*sinf(lat)*radius;
            y = sinf(lon)*sinf(lat)*radius;
            z = cosf(lat)*radius;
            sphereVertices.push_back(glm::vec4(x, y, z, 1));
            x = cosf(lon+lonstep)*sinf(lat)*radius;
            y = sinf(lon+lonstep)*sinf(lat)*radius;
            z = cosf(lat)*radius;
            sphereVertices.push_back(glm::vec4(x, y, z, 1));
        }
    }
}
```



경도(lon) 위도(lat)

x = cosφ *cosθ
y = sinθ
z = sinφ * cosθ
where 0 ≤ φ ≤ 2π, -π/2 ≤θ≤ π/2

Cube

- Draw a wireframe cube
  - GL_LINE_LOOP
- Draw a solid cube
  - GL_TRIANGLES