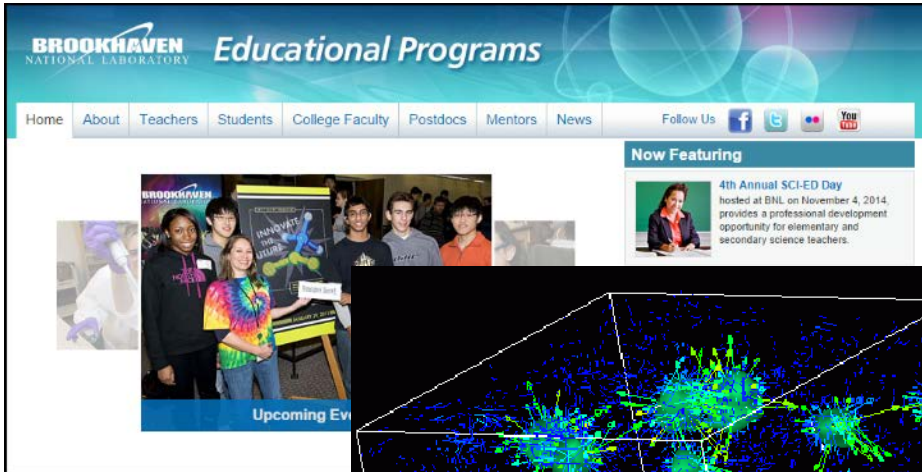
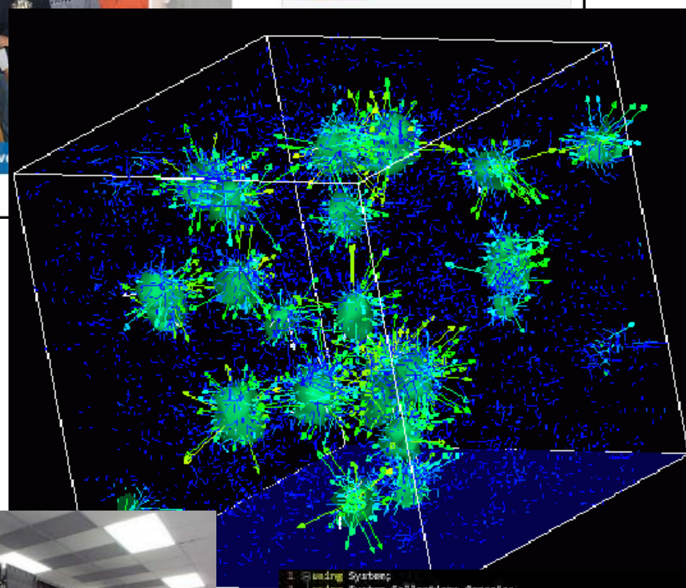
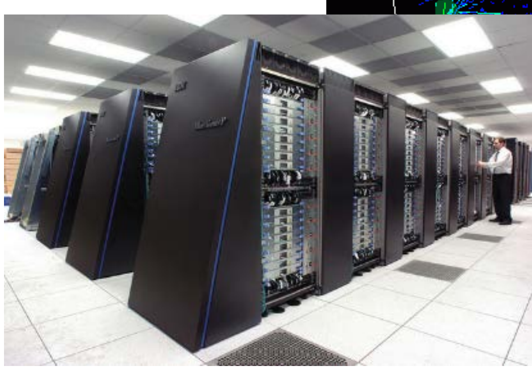






Survey of Scientific Computing (SciComp 301)

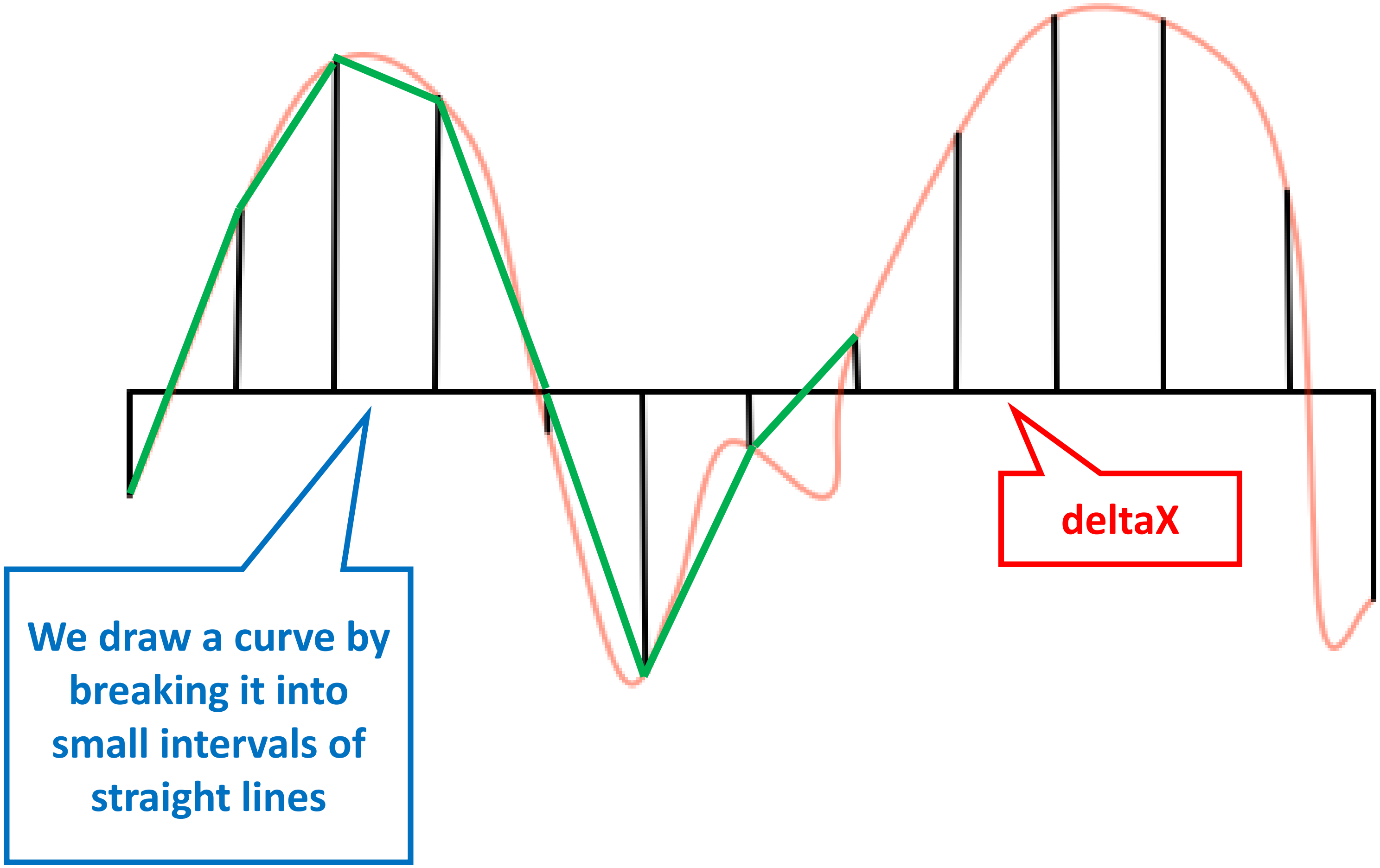
Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 16
3D Graphics, Vector Algebra

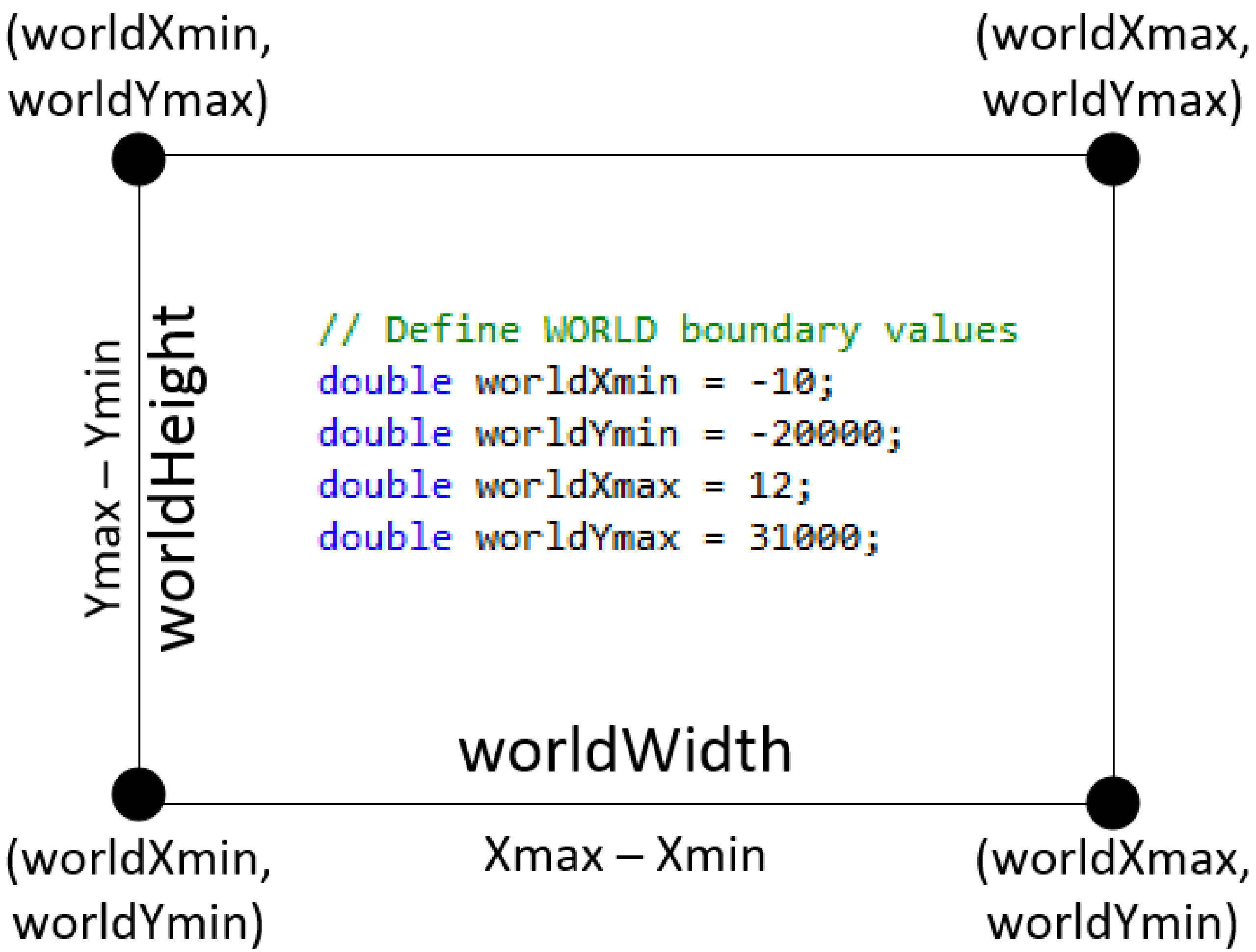
Session Goals

- Sizing the **World Rectangle** to frame polynomials
- Review 3D Cartesian coordinates and **oblique projection**
- Create a **vertex** array from a set of **Point3D** elements
- Create a **facet** from a set of vertices
- Draw a wireframe **monolith** and **pyramid**
- Introduce **spherical coordinates** (θ and ϕ)
- Draw a wireframe **sphere** and **torus**
- Use vector **cross product** to perform **back face culling**
- Use vector **dot product** to perform **facet shading**

Drawing Curves using Intervals



Bounding World Rectangle



Drawing a Polynomial

- Your task is to graph this polynomial using Allegro:

$$y = x^5 - 2x^4 - 120x^3 + 22x^2 + 2119x + 1980$$

- First determine the appropriate World bounding rectangle values to "see" the full polynomial
  - Hint: $y = (x+9)(x+4)(x+1)(x-5)(x-11)$
  - Hint: Find the roots of $\frac{dy}{dx} = 0$ to locate **extrema points**
- What does the **Fundamental Theorem of Algebra** tell us about the **maximum** number of places **y(x)** *could* cross the **x-axis**?

Finding Extrema

$$\frac{dy}{dx} = 5x^4 - 8x^3 - 360x^2 + 44x + 2119$$

WolframAlpha | PRO PREMIUM

Input interpretation:

factor	$5x^4 - 8x^3 - 360x^2 + 44x + 2119$

Irreducible factorization:

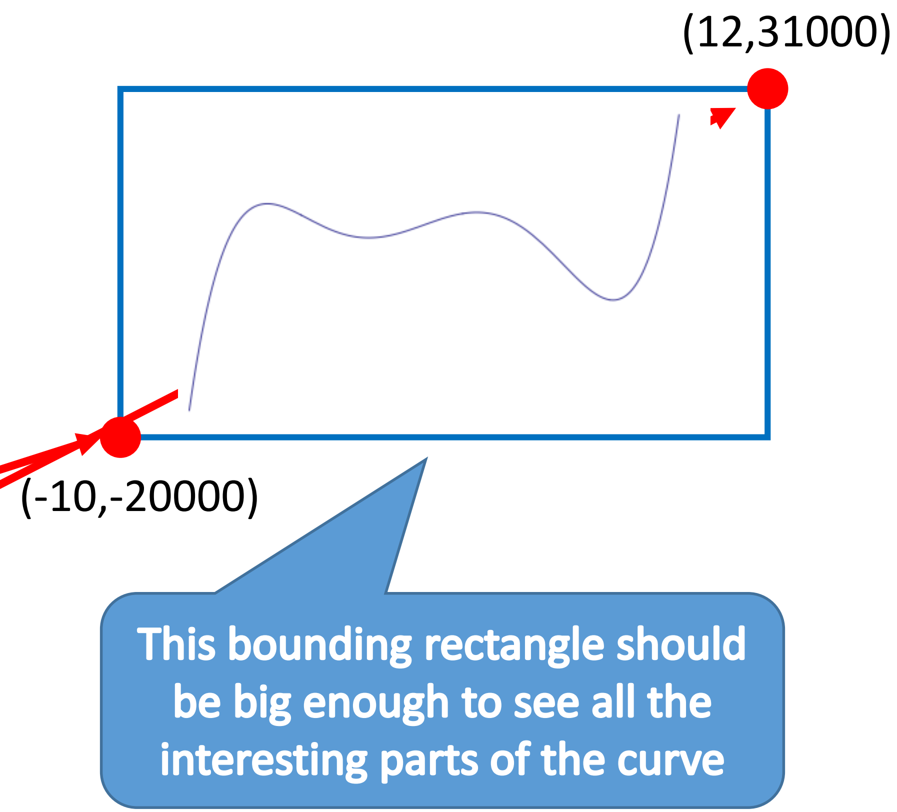
Exact form

$5(x - 8.94624)(x - 2.53081)(x + 2.55715)(x + 7.31989)$

[illegible]	[illegible]	[illegible]
-2.5572	-1483.0510	Extrema
-1.0000	0.0000	Zero
2.5308	5560.2920	Extrema
5.0000	0.0000	Zero
8.9462	-18728.7695	Extrema
11.0000	0.0000	Zero
12.0000	30576.0000	

Finding Extrema

Zeros and Extrema of		
y=(x+9)(x+4)(x+1)(x-5)(x-11)		
X	**Y**	
-10.0000	-17010.0000	
-9.0000	0.0000	Zero
-7.3199	7956.1071	Extrema
-4.0000	0.0000	Zero
-2.5572	-1483.0510	Extrema
-1.0000	0.0000	Zero
2.5308	5560.2920	Extrema
5.0000	0.0000	Zero
8.9462	-18728.7695	Extrema
11.0000	0.0000	Zero
12.0000	30576.0000	



Lab 1 – Draw Polynomial

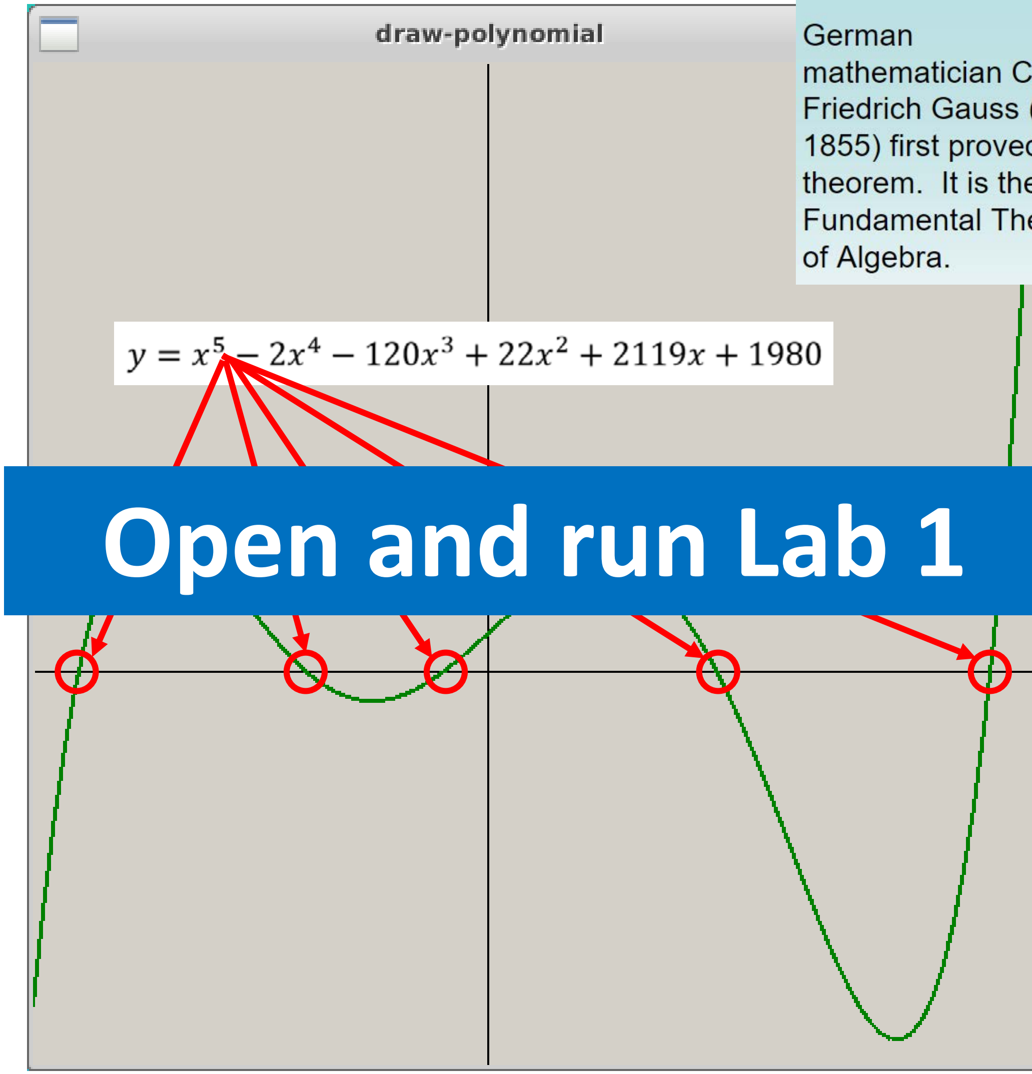
```
void draw(SimpleScreen& ss)
{
    ss.DrawAxes();

    double x= ss.worldXmin ;
    const double dx = ss.worldWidth / 97;

    PointSet ps;
    while (x <= ss.worldXmax) {
        double y = (x + 9) * (x + 4)
            * (x + 1) * (x - 5) * (x - 11);
        ps.add(x, y);
        x += dx;
    }

    ss.DrawLines(&ps, "green", 2, false);
}

int main()
{
    SimpleScreen ss(draw);
    ss.SetWorldRect(-10, -20000, 12, 31000);
    ss.HandleEvents();
    return 0;
}
```

Prime number to minimize the chance of aliasing

Approximate the curve using small line segments

From the roots & extrema

Lab 1 – Draw Polynomial



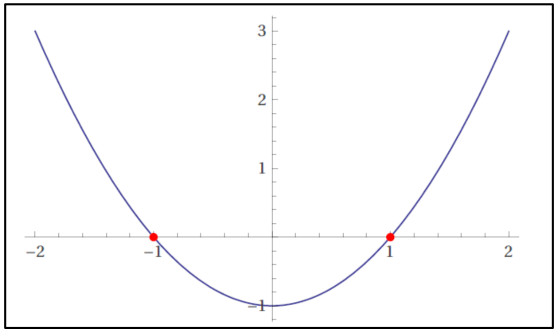
$$y = x^5 - 2x^4 - 120x^3 + 22x^2 + 2119x + 1980$$

German mathematician Carl Friedrich Gauss (1777-1855) first proved this theorem. It is the Fundamental Theorem of Algebra.

Open and run Lab 1

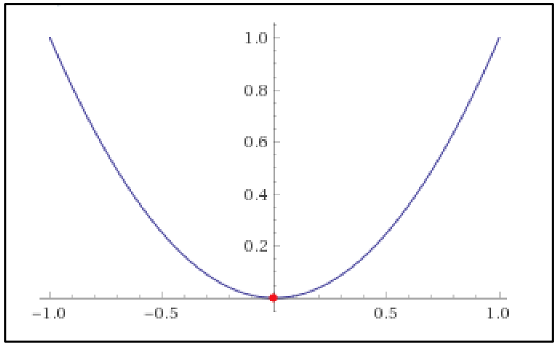
Polynomial Power & Complex Roots

$y = x^2 - 1$



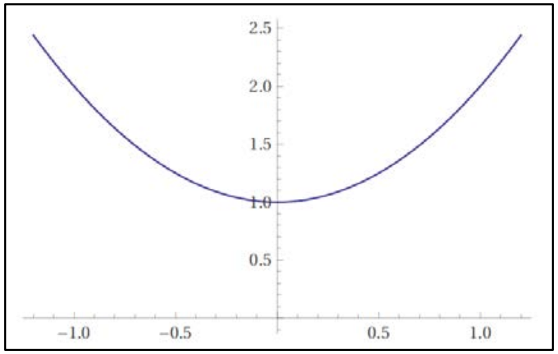
Two real roots: $x = \pm 1$

$y = x^2$



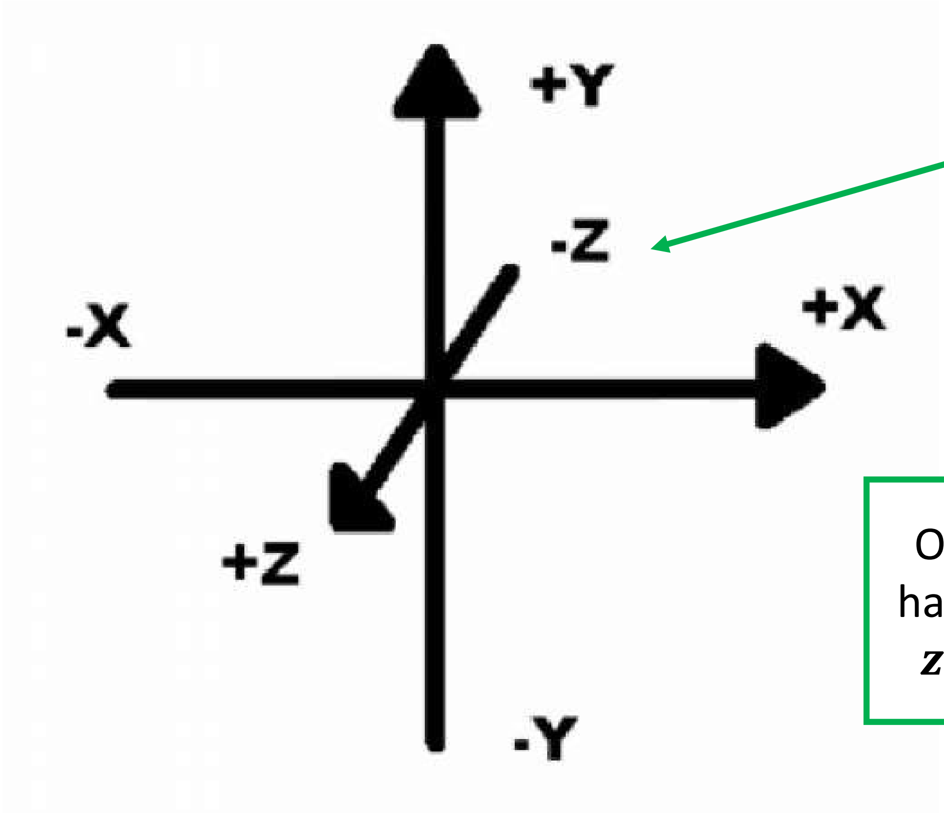
"Two" real roots: $x = 0{,}0$

$y = x^2 + 1$



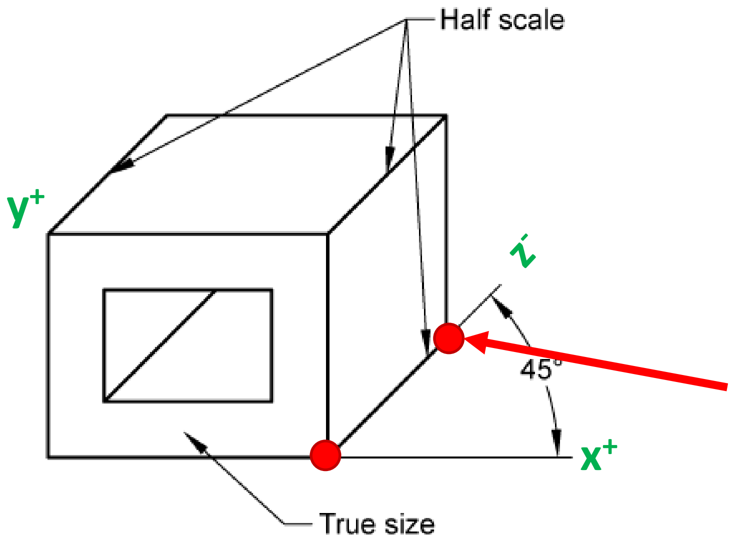
Two $\mathbb{C}$ roots: $x = \pm i$

Axis Orientation in 3D



Objects father away have a more **negative** z coordinate value!

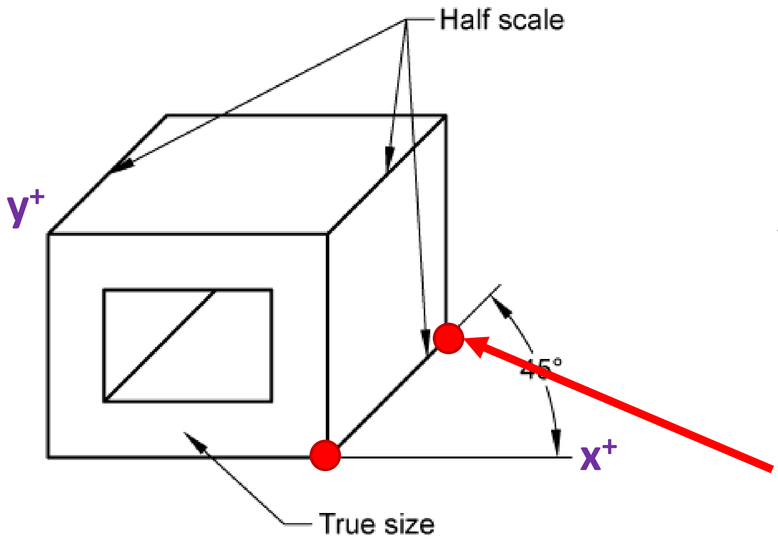
Oblique Projection



In 3-space (**world coordinates**), these two **red** points have the **same** X & Y coordinate values. They only differ in their Z coordinate values.

The **back** red point has **only** a more *negative* Z coordinate value than the **front** red point.

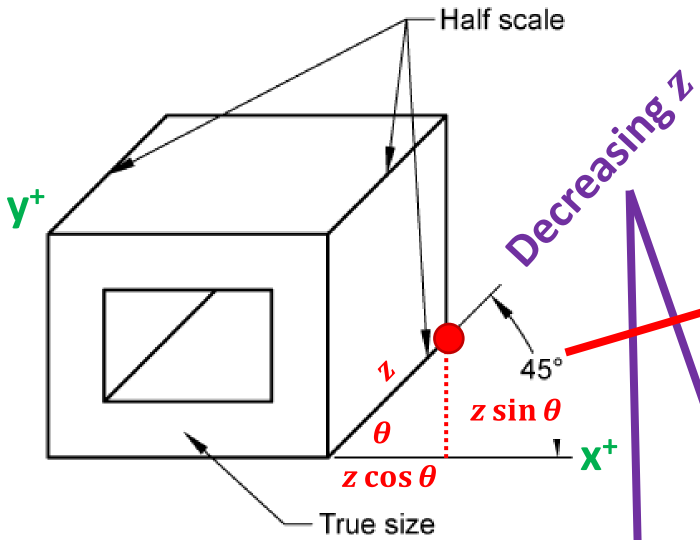
Oblique Projection



When drawn in 2-space (**screen coordinates**), to provide a sense of depth, the **back** red point needs *artificially* **larger** X & Y coordinate values than the **front** red point X & Y coordinate values.

We adjust the perceived 2D screen X & Y locations of the **back** red point according to the value of its Z coordinate.

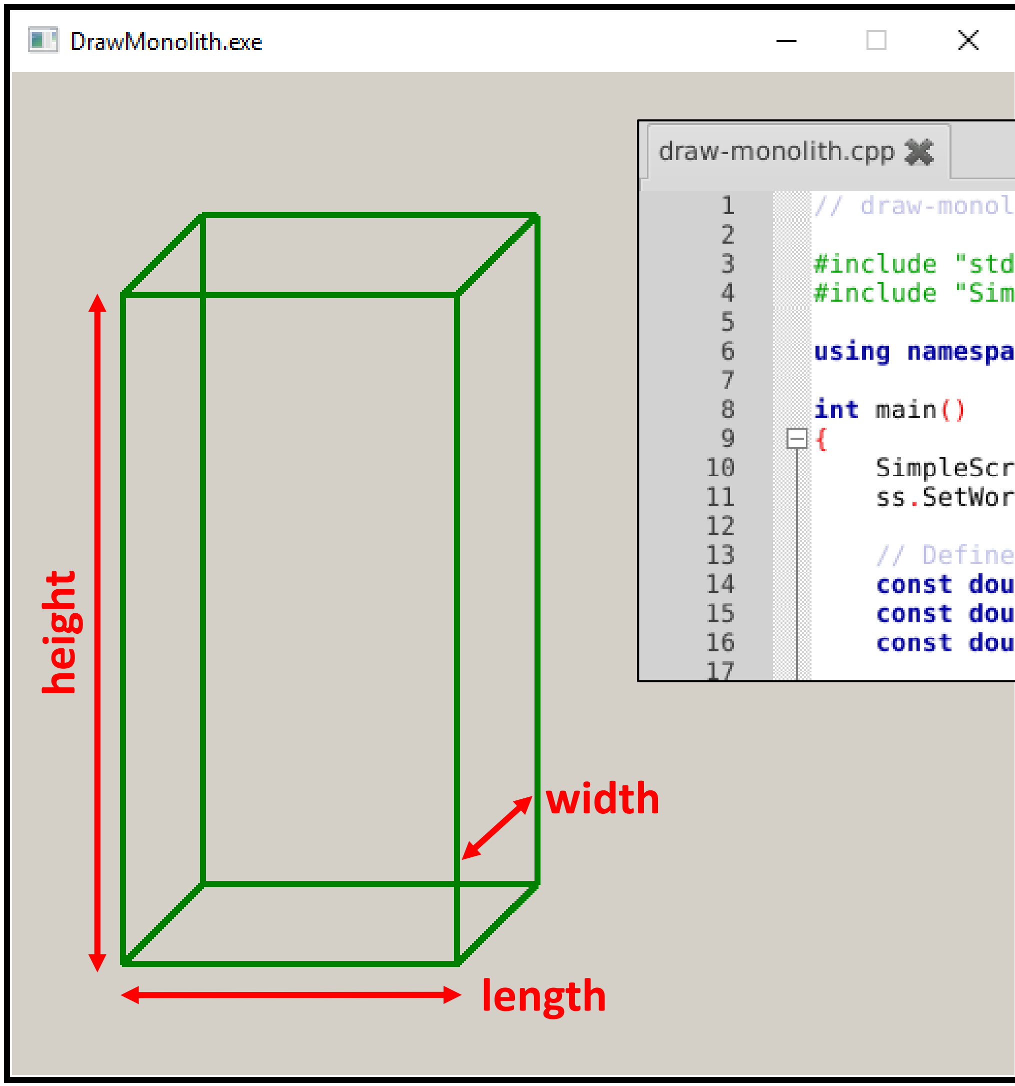
Oblique Projection




```
private void SetProjection(double degrees = 45,
                           double correction = 1)
{
    obliqueAngle = degrees * Math.PI / 180;
    obliqueCos = Math.Cos(obliqueAngle);
    obliqueSin = Math.Sin(obliqueAngle);
    aspectCorrection = correction;
}
```

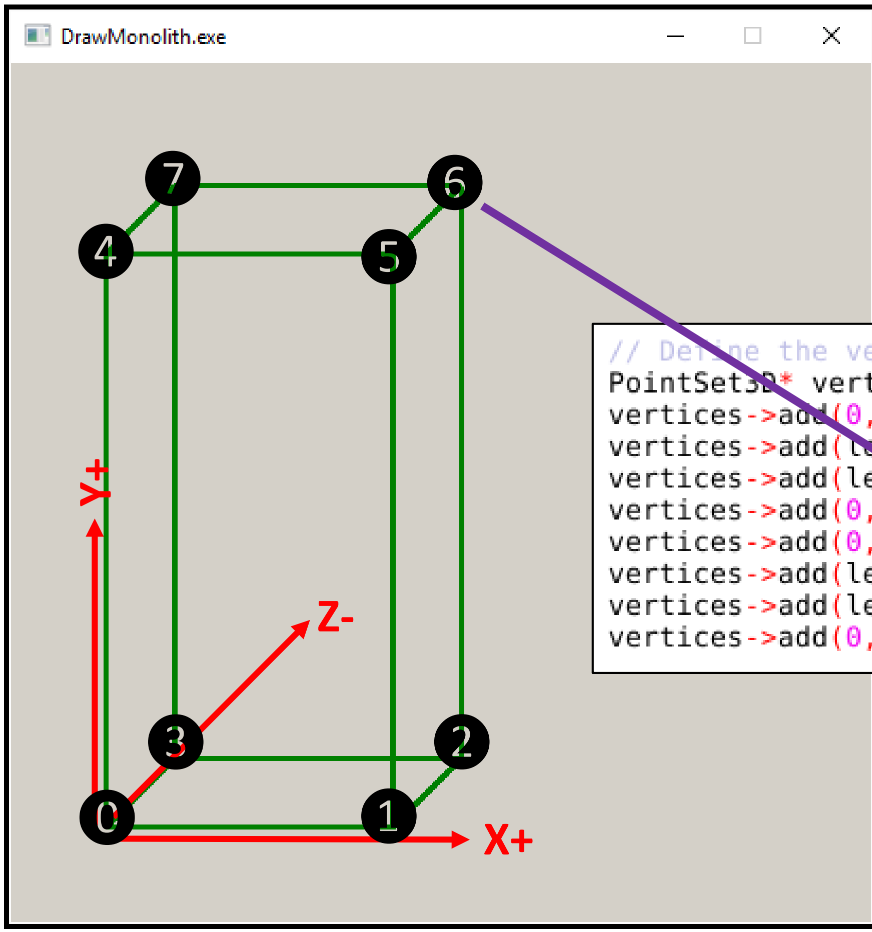
```
void SimpleScreen::CalcScreenPoints3D(PointSet3D* ps3d)
{
    points->clear();
    for (size_t i{}; i < ps3d->size(); ++i) {
        // Convert WORLD coordinates to SCREEN coordinates
        double screenX = (ps3d->at(i)->x -
            ps3d->at(i)->z * obliqueCos) * aspectCorrection - worldXmin) * scaleX;
        double screenY = screenHeight -
            (ps3d->at(i)->y - ps3d->at(i)->z * obliqueSin - worldYmin) * scaleY;
        // Add this SCREEN coordinate to array of points
        points->push_back(new Point2D(screenX, screenY));
    }
}
```

Oblique Projection – Monolith




```cpp
// draw-monolith.cpp

#include "stdafx.h"
#include "SimpleScreen.h"

using namespace std;

int main()
{
    SimpleScreen ss;
    ss.SetWorldRect(-50, -50, 400, 400);

    // Define the size of the monolith
    const double length = 150;    // In X direction
    const double height = 300;    // In Y direction
    const double width = 50;      // In Z direction
```

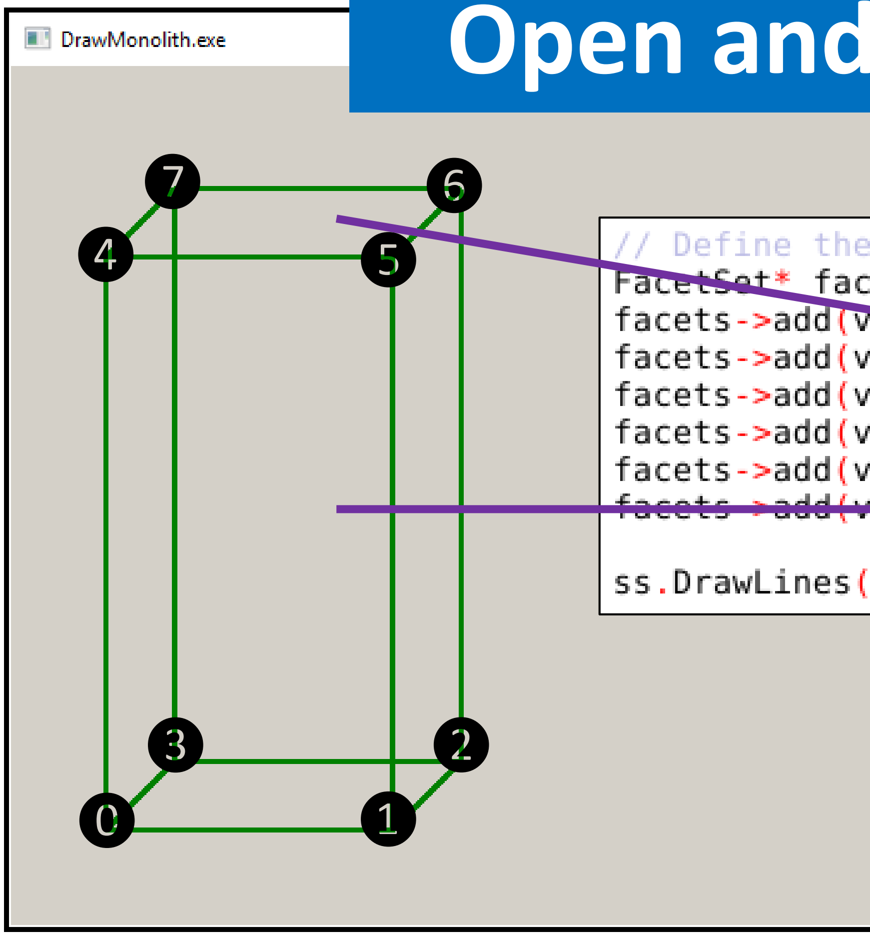
Oblique Projection – Vertex Array




```
// Define the vertices of the monolith by 3D world coordinates
PointSet3D* vertices = new PointSet3D();
vertices->add(0, 0, 0);                    // Front Left Bottom
vertices->add(length, 0, 0);               // Front Right Bottom
vertices->add(length, 0, -width);          // Back Right Bottom
vertices->add(0, 0, -width);               // Back Left Bottom
vertices->add(0, height, 0);               // Front Left Top
vertices->add(length, height, 0);          // Front Right Top
vertices->add(length, height, -width);     // Back Right Top
vertices->add(0, height, -width);          // Back Left Top
```

Oblique Projection – Facet Array

Open and run Lab 2

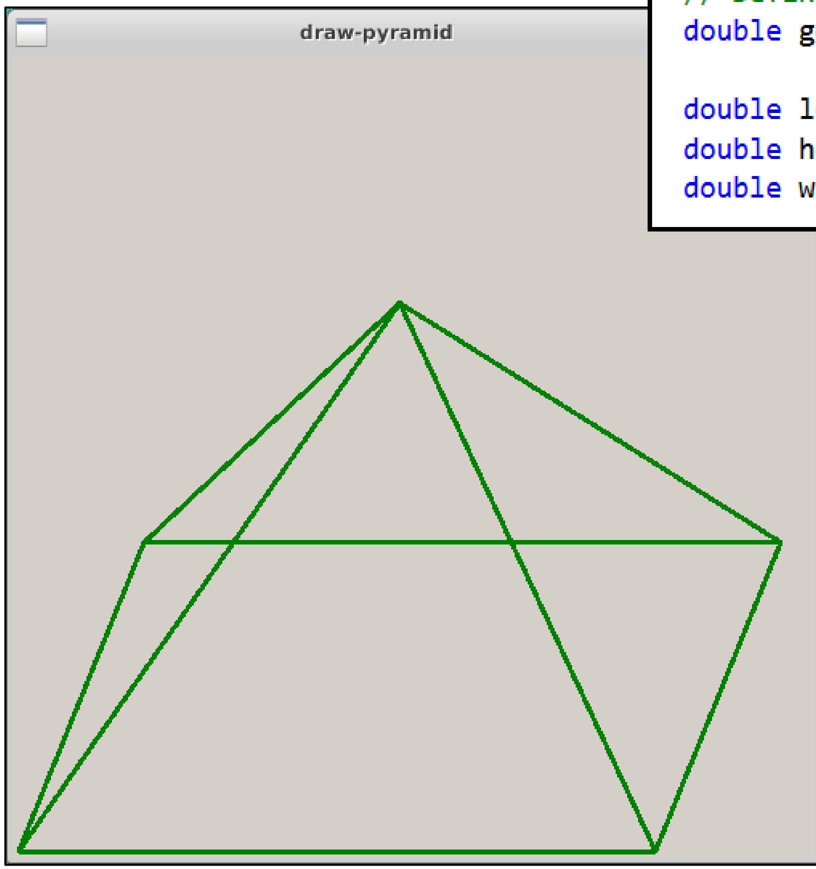


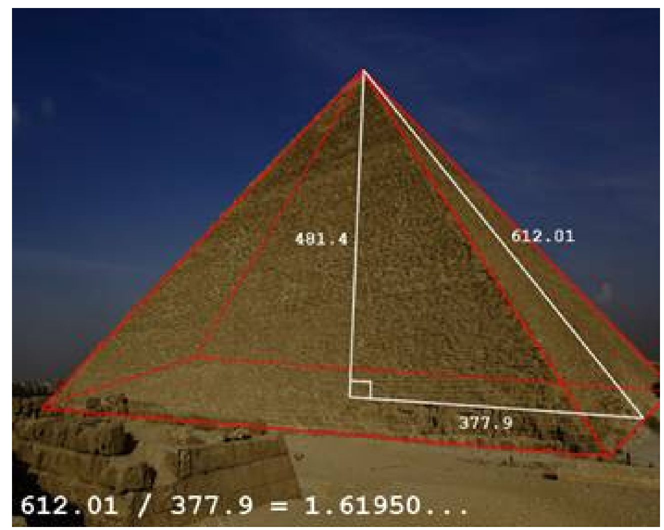

```
// Define the facets of the monolith by vertex numbers
FacetSet* facets = new FacetSet();
facets->add(vertices, { 0,1,2,3 });     // Bottom
facets->add(vertices, { 4,5,6,7 });     // Top
facets->add(vertices, { 0,4,7,3 });     // Left
facets->add(vertices, { 1,2,6,5 });     // Right
facets->add(vertices, { 0,1,5,4 });     // Front
facets->add(vertices, { 2,3,7,6 });     // Back

ss.DrawLines(facets, "green", 3, false, 2000);
```

Delay Count

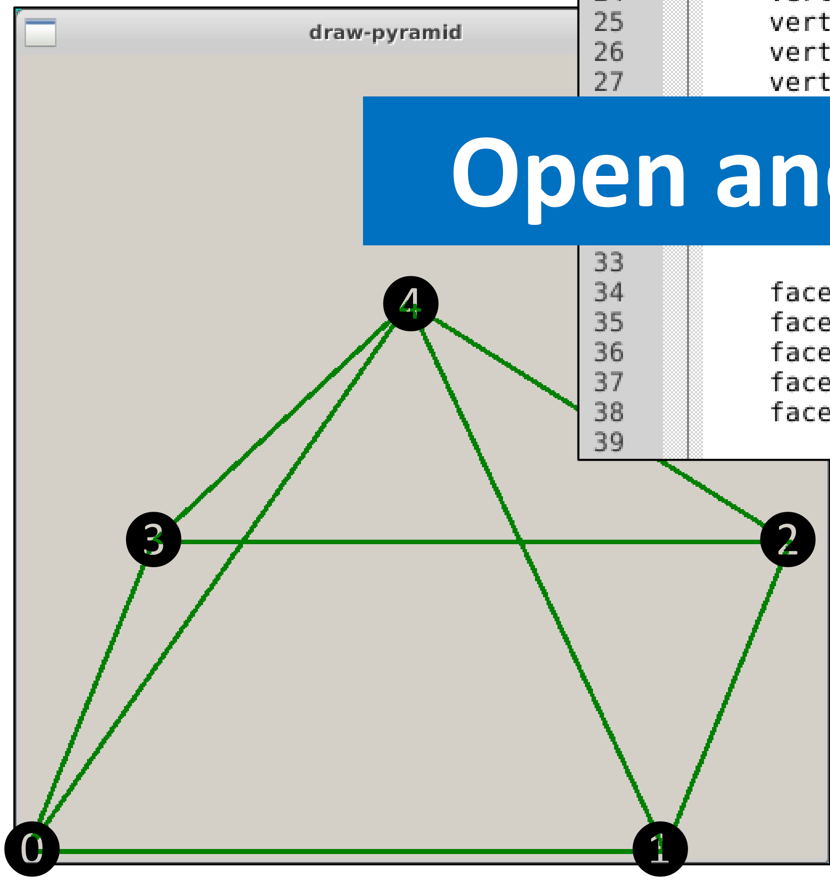
Oblique Projection – Pyramid

```
// Define the size of the pyramid
double goldenRatio = (1.0 + sqrt(5)) / 2.0;

double length = 400;                          // In X direction
double height = length / goldenRatio;         // In Y direction
double width = 400;                           // In Z direction
```





Oblique Projection – Pyramid




```
// Define the vertices of the pyramid by 3D world coordinates
PointSet3D* vertices = new PointSet3D();
vertices->add(0, 0, 0);                          // Base Front Left
vertices->add(length, 0, 0);                     // Base Front Right
vertices->add(length, 0, -width);                // Base Back Right
vertices->add(0, 0, -width);                     // Base Back Left
                                                 // Apex

                                                 rtex numbers

facets->add(vertices, { 0,1,2,3 });         // Base
facets->add(vertices, { 0,3,4 });           // Left
facets->add(vertices, { 0,1,4 });           // Front
facets->add(vertices, { 1,2,4 });           // Right
facets->add(vertices, {/*TODO*/ });         // Back
```

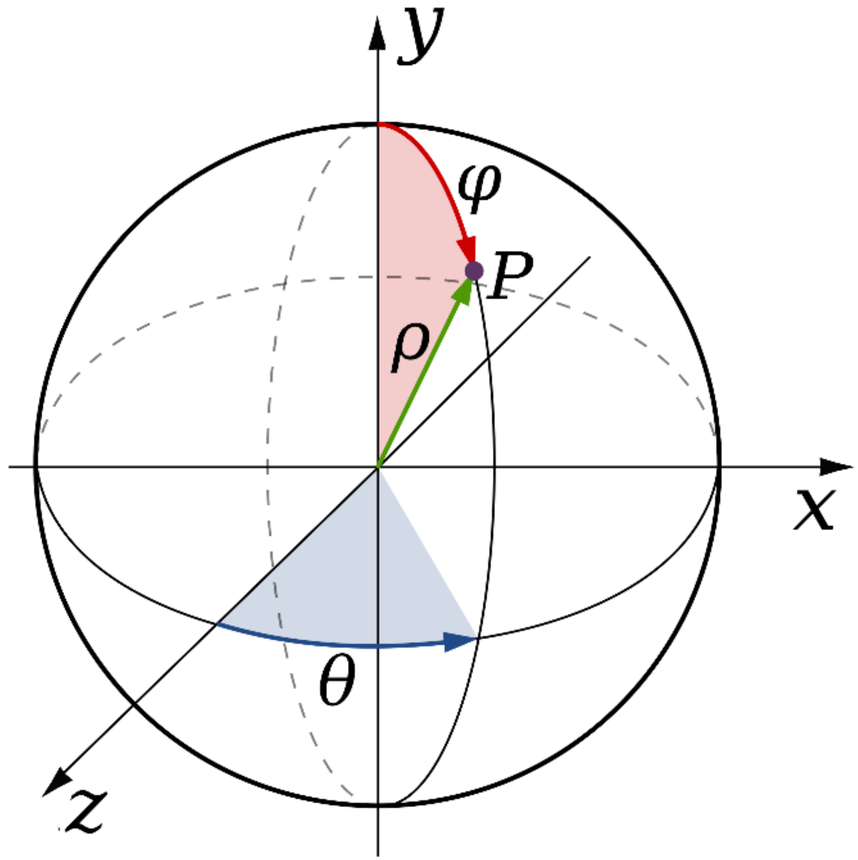
Open and **edit** Lab 3

Remove the **two** /* TODO */ comments in the definitions of both **vertices** and **facets**, and enter the correct code

Oblique Projection – Pyramid

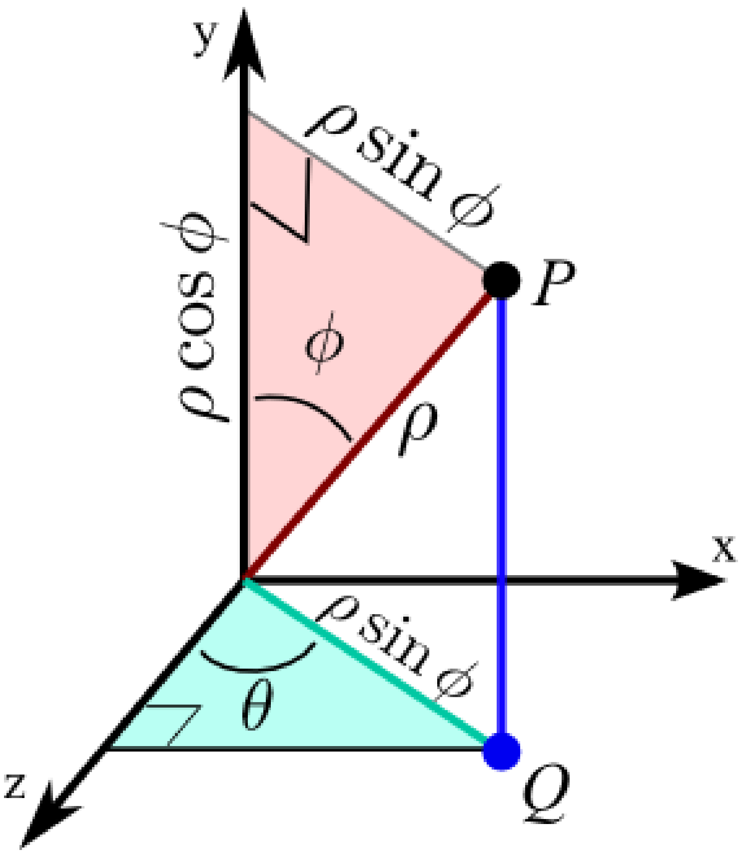
```
// Define the vertices of the pyramid by 3D world coordinates
PointSet3D* vertices = new PointSet3D();
vertices->add(0, 0, 0);                          // Base Front Left
vertices->add(length, 0, 0);                     // Base Front Right
vertices->add(length, 0, -width);                // Base Back Right
vertices->add(0, 0, -width);                     // Base Back Left
vertices->add(length/2, height, -width/2);       // Apex


// Define the facets of the pyramid by vertex numbers
FacetSet* facets = new FacetSet();

facets->add(vertices, { 0,1,2,3 });      // Base
facets->add(vertices, { 0,3,4 });        // Left
facets->add(vertices, { 0,1,4 });        // Front
facets->add(vertices, { 1,2,4 });        // Right
facets->add(vertices, { 2,3,4 });        // Back
```

Spherical Coordinates

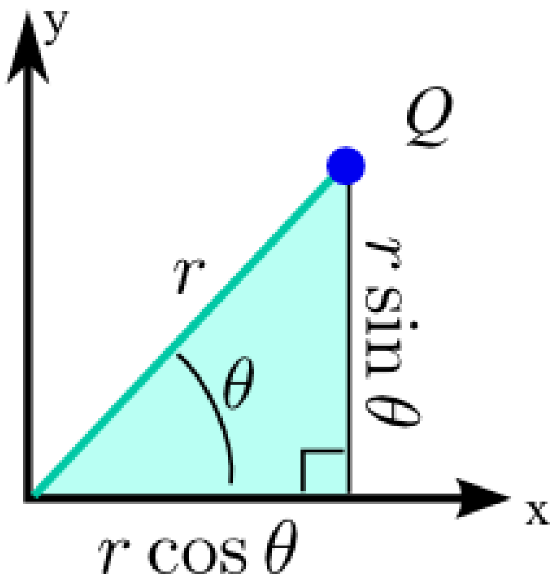


Spherical Coordinates

3D Spherical

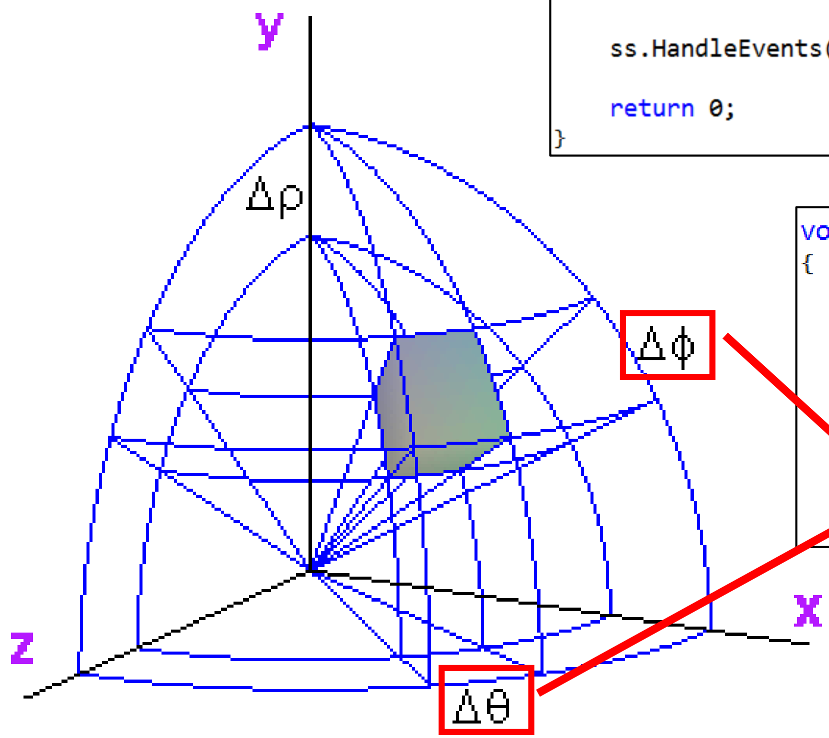


2D Polar



Spherical Coordinates

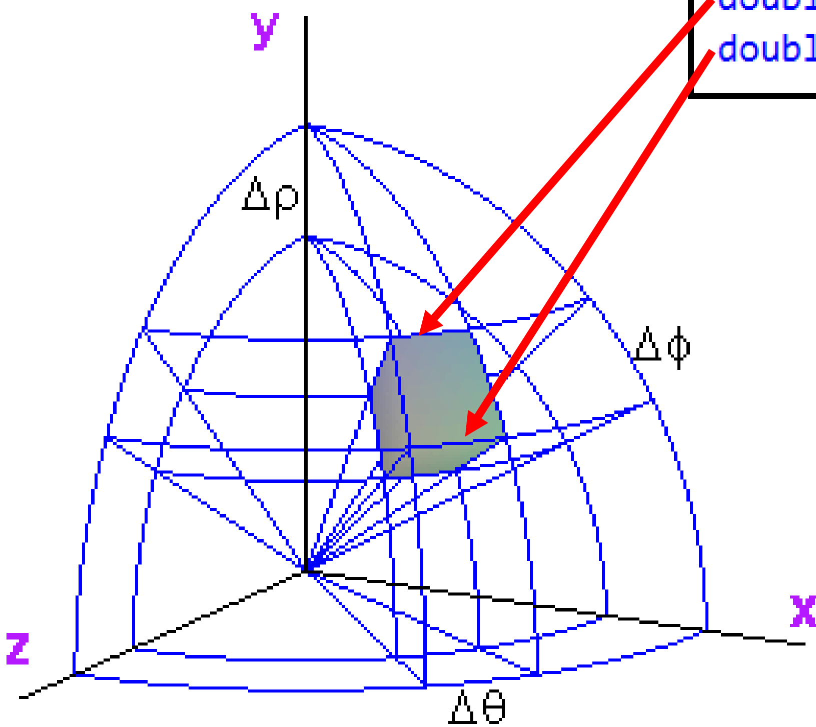
```
int main()
{
    SimpleScreen ss(draw);
    ss.SetZoomFrame("white", 3);

    ss.SetWorldRect(-200, -200, 200, 200);

    ss.SetProjection(29, 0.225);

    ss.SetCameraLocation(30000, 60000, 120000);

    ss.HandleEvents();

    return 0;
}
```

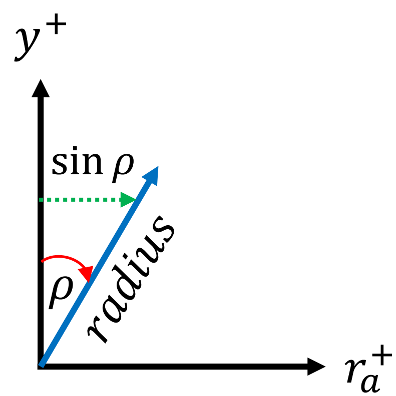



```
void draw(SimpleScreen& ss)
{
    // Set the radius of the sphere
    double radius = 175;

    // Calculate the angle deltas
    double intervals = 37;
    double deltaPhi = M_PI / intervals;         // Latitudes
    double deltaTheta = 2 * M_PI / intervals;   // Longitudes
```

Spherical Coordinates

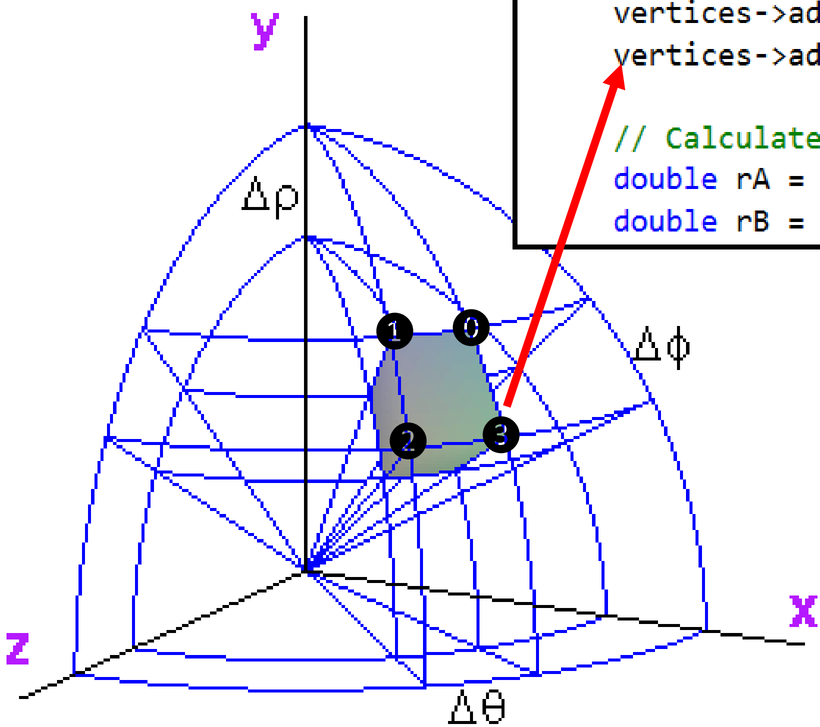
```
// Calculate radius of each line of latitude
double rA = radius * sin(phi);
double rB = radius * sin(phi + deltaPhi);
```

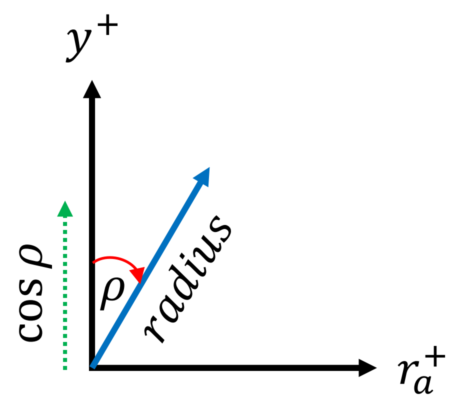




Spherical Coordinates

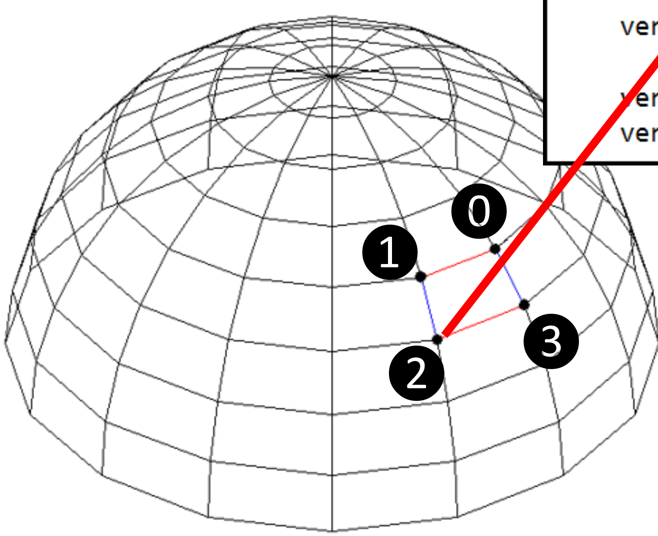
```
// Step phi around a half-circle to set each vertex "Y" coordinate
for (double phi = 0; phi < M_PI; phi += deltaPhi)
{
    PointSet3D* vertices = new PointSet3D();
    vertices->add(0, radius * cos(phi), 0);
    vertices->add(0, radius * cos(phi), 0);
    vertices->add(0, radius * cos(phi + deltaPhi), 0);
    vertices->add(0, radius * cos(phi + deltaPhi), 0);

    // Calculate radius of each line of latitude
    double rA = radius * sin(phi);
    double rB = radius * sin(phi + deltaPhi);
```

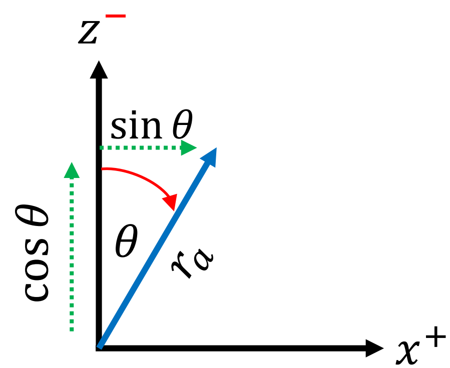




Vertices and Facets

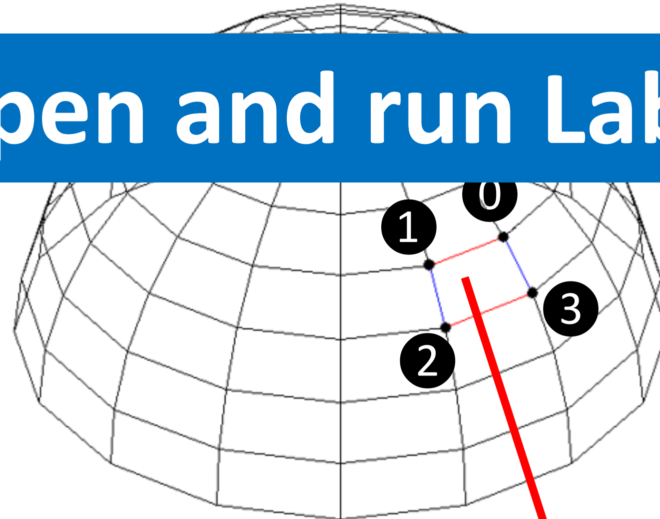
```
// Step theta around a full circle to set each vertex "X" and "Z" coordinate
for (double theta = 0; theta < M_PI * 2; theta += deltaTheta)
{
    vertices->at(0)->x = rA * sin(theta);
    vertices->at(0)->z = -rA * cos(theta);

    vertices->at(1)->x = rA * sin(theta + deltaTheta);
    vertices->at(1)->z = -rA * cos(theta + deltaTheta);

    vertices->at(2)->x = rB * sin(theta + deltaTheta);
    vertices->at(2)->z = -rB * cos(theta + deltaTheta);

    vertices->at(3)->x = rB * sin(theta);
    vertices->at(3)->z = -rB * cos(theta);
```





Vertex Winding Order

Open and run Lab 4

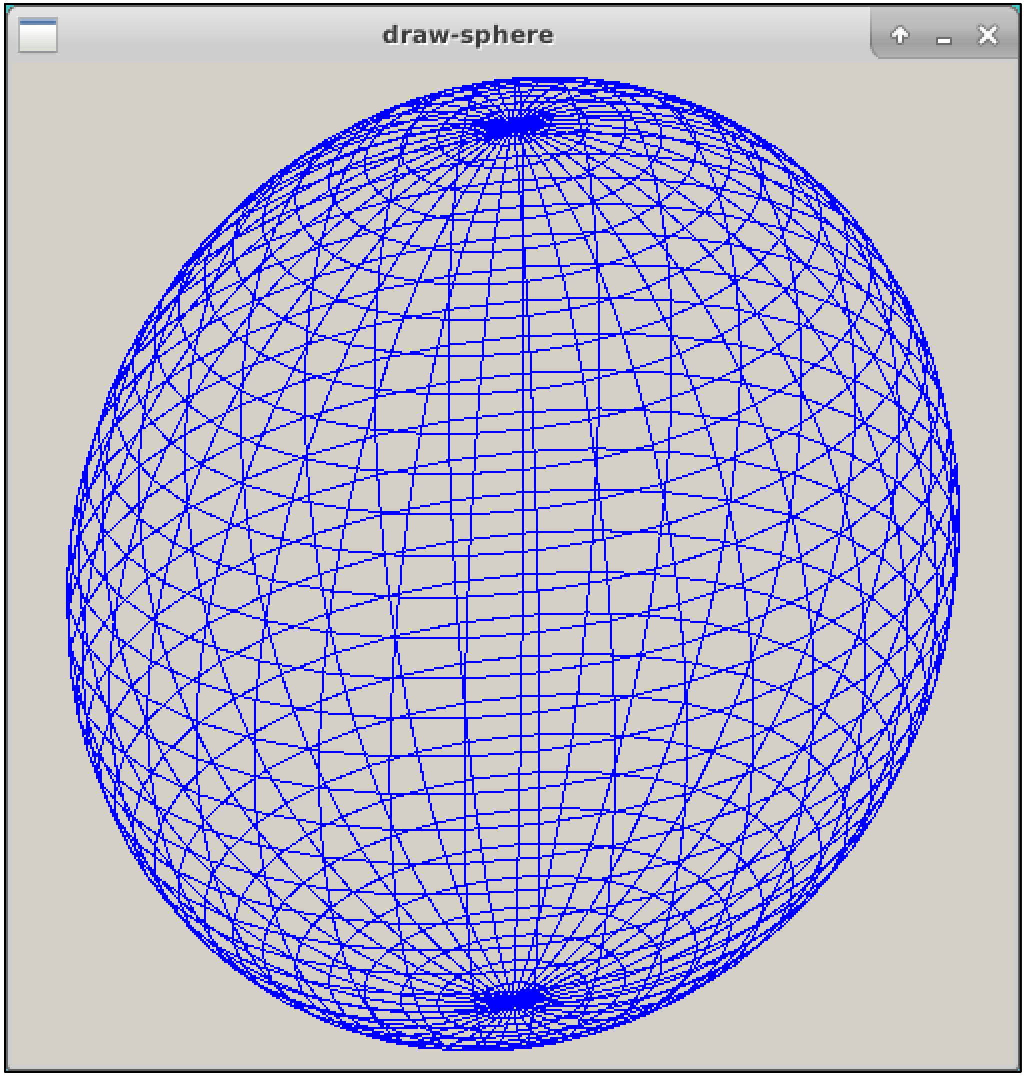



```
            // At the North pole (phi == 0) vertex 0 and 1 are the same points,
            // so we use a different vertex number ordering to designate a
            // more meaningful surface normal for those particular facets
            Facet* f = (phi > 0) ? new Facet(vertices, { 0, 1, 2, 3 })
                : new Facet(vertices, { 2, 3, 0, 1 });

            ss.DrawFacet(f, al_map_rgb(0, 0, 0), al_map_rgb(0, 0, 255), 1, false, false, 0);
        }
    }
}
```

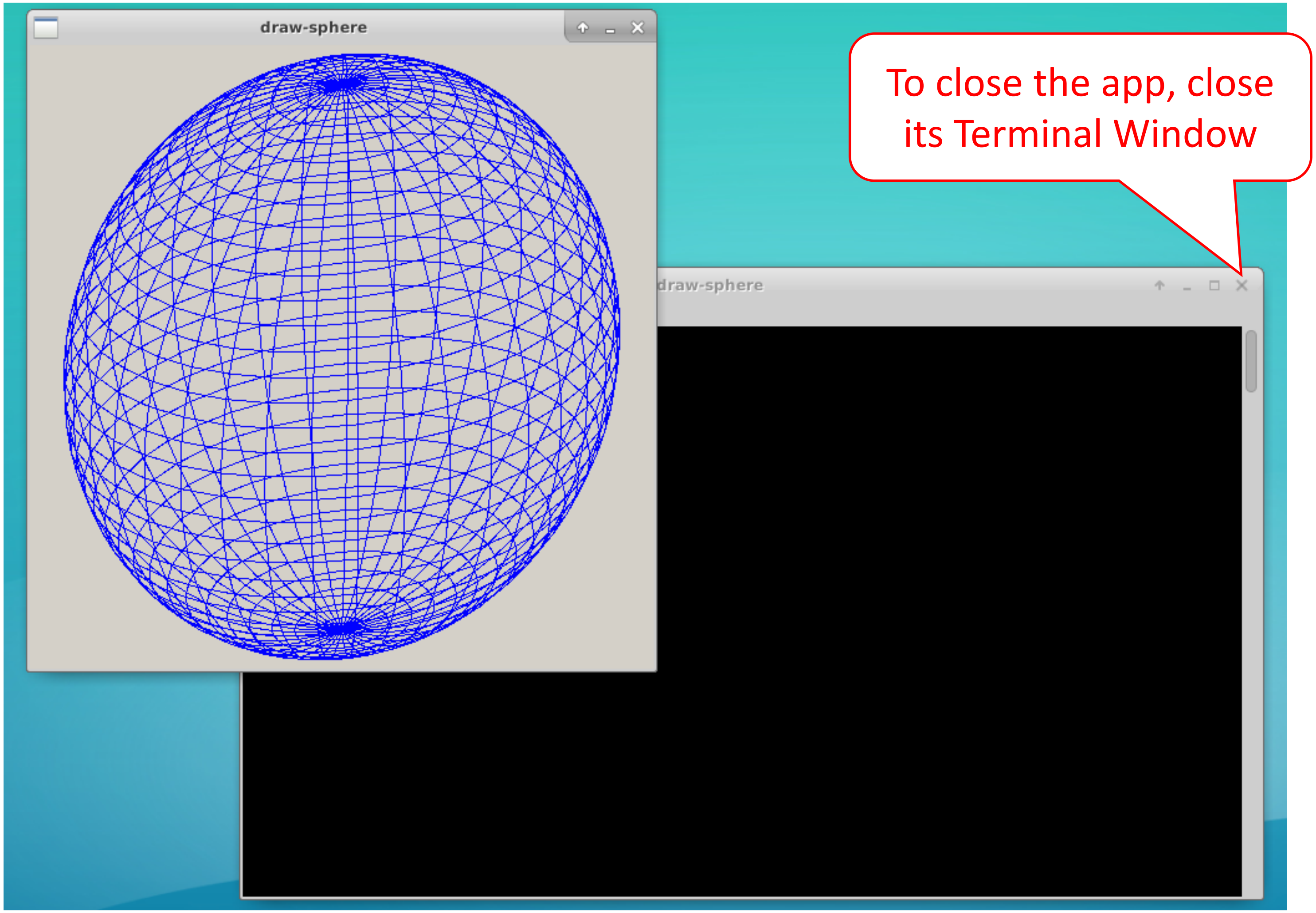
RGB Min

RGB Max

Wireframe Sphere



Wireframe Sphere



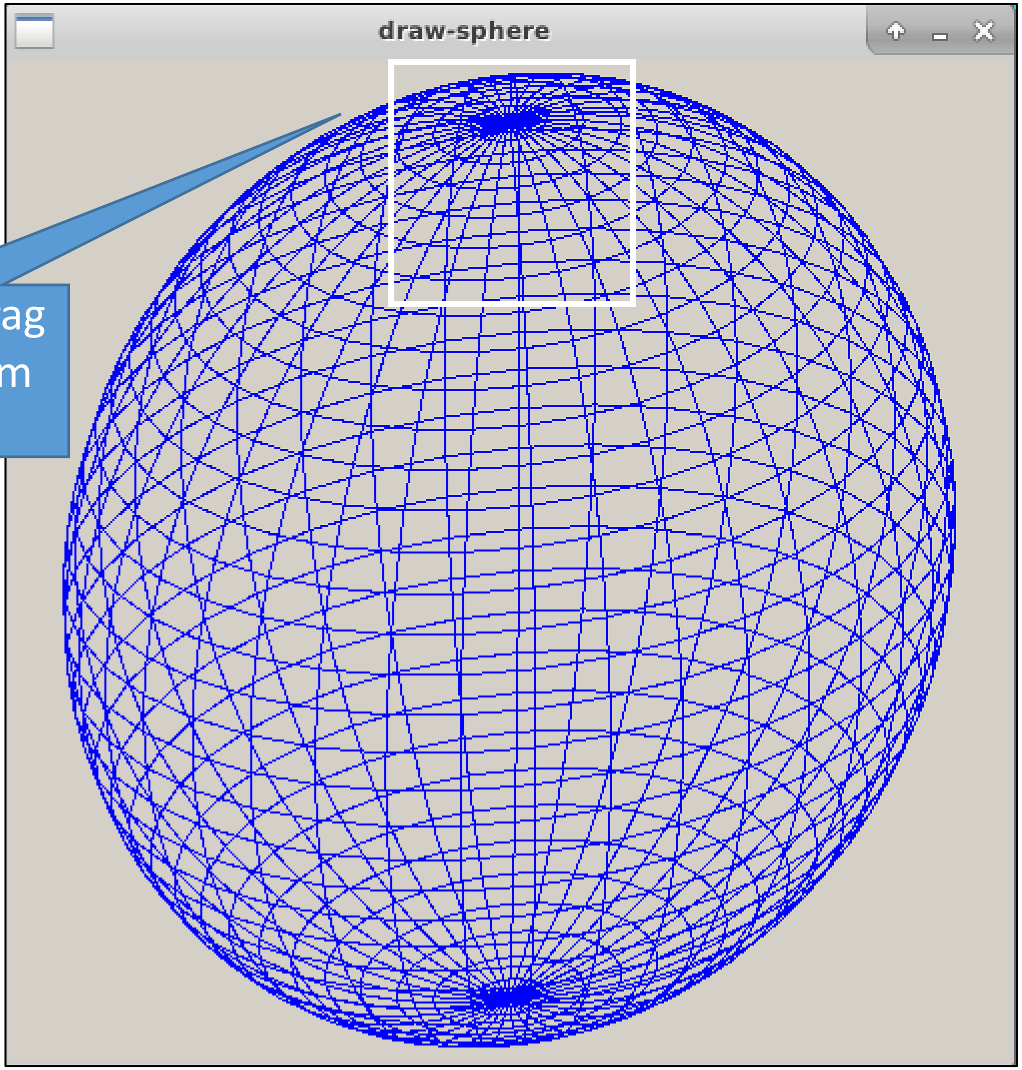
Wireframe Sphere

Delay Count

```
            // At the North pole (phi == 0) vertex 0 and 1 are the same points,
            // so we use a different vertex number ordering to designate a
            // more meaningful surface normal for those particular facets
            Facet* f = (phi > 0) ? new Facet(vertices, { 0, 1, 2, 3 })
                : new Facet(vertices, { 2, 3, 0, 1 });

            ss.DrawFacet(f, al_map_rgb(0, 0, 0), al_map_rgb(0, 0, 255), 1, false, false, 0);
        }
    }
}
```

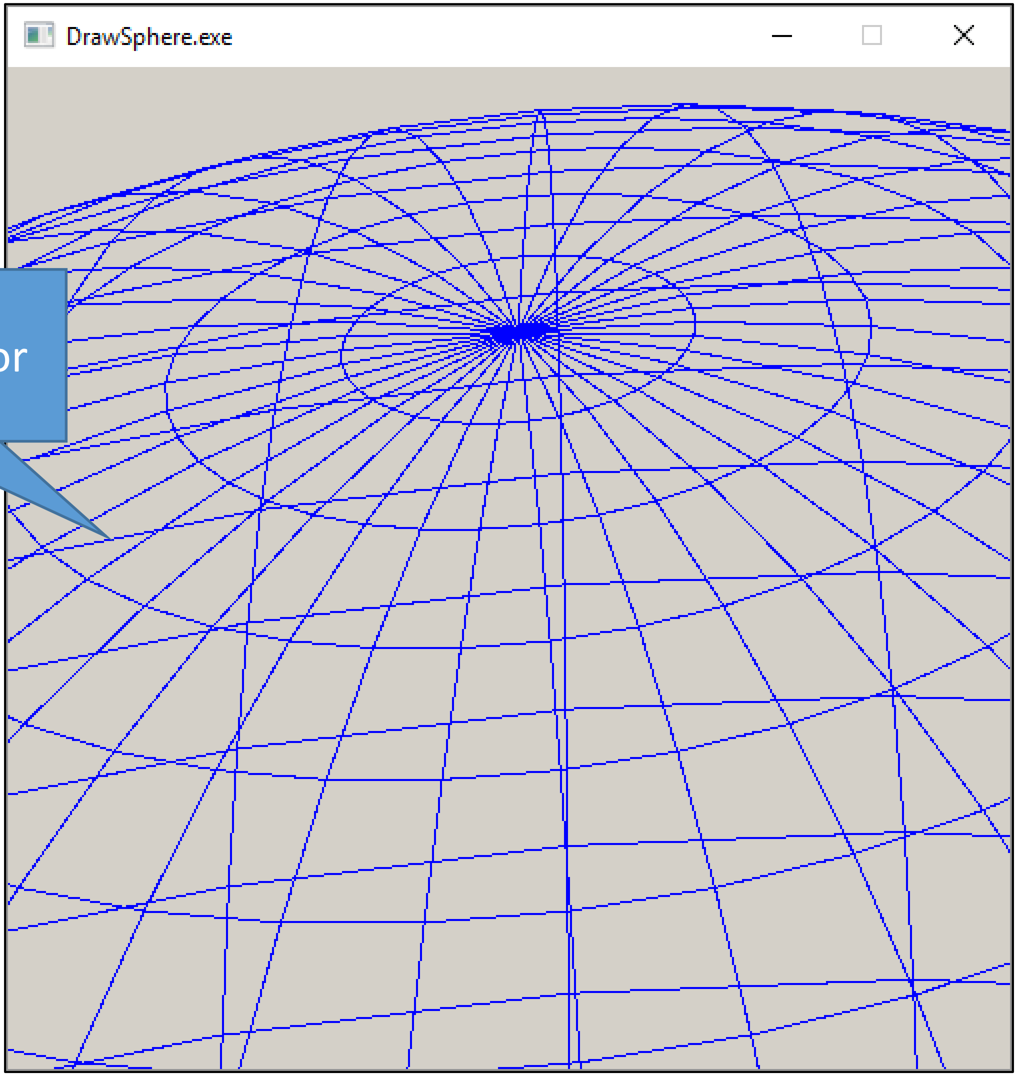
Re-run Lab 4

Wireframe Sphere



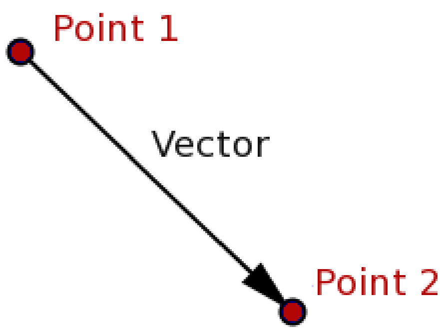
Left click and drag to create a zoom rectangle

Wireframe Sphere

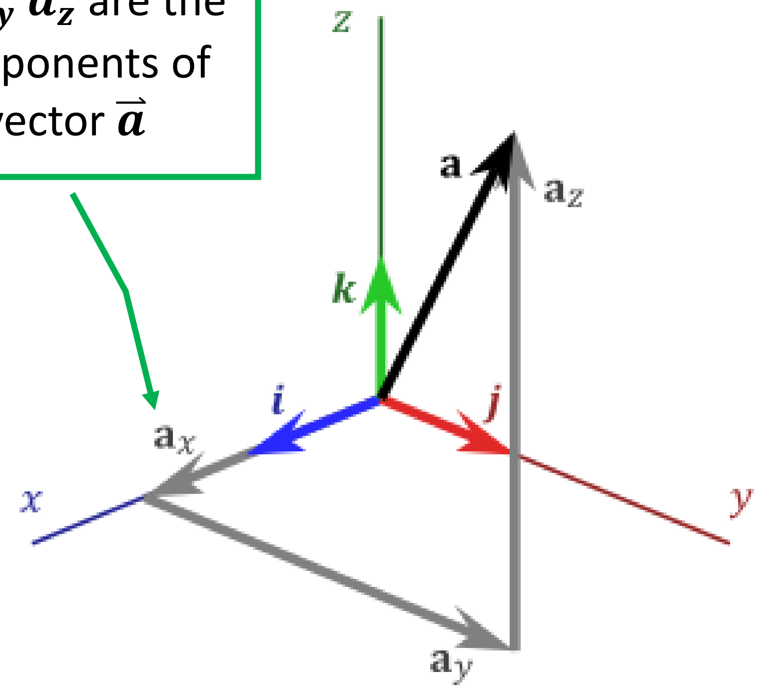


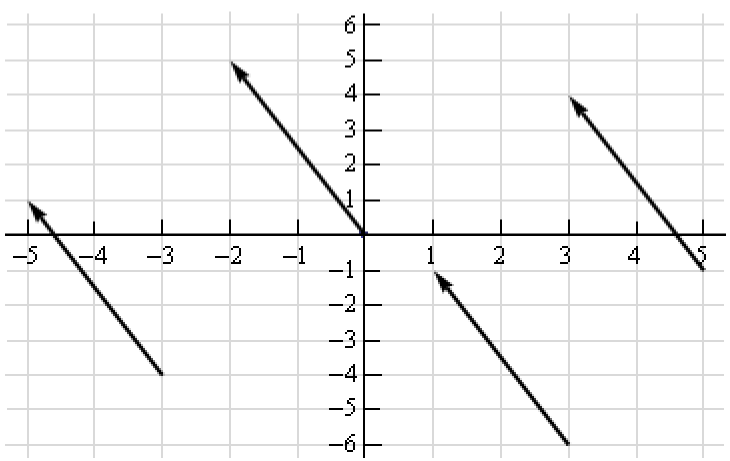
Right click to restore the prior zoom state

What is a vector?



$\boldsymbol{a_x}$ $\boldsymbol{a_y}$ $\boldsymbol{a_z}$ are the components of vector $\vec{\boldsymbol{a}}$

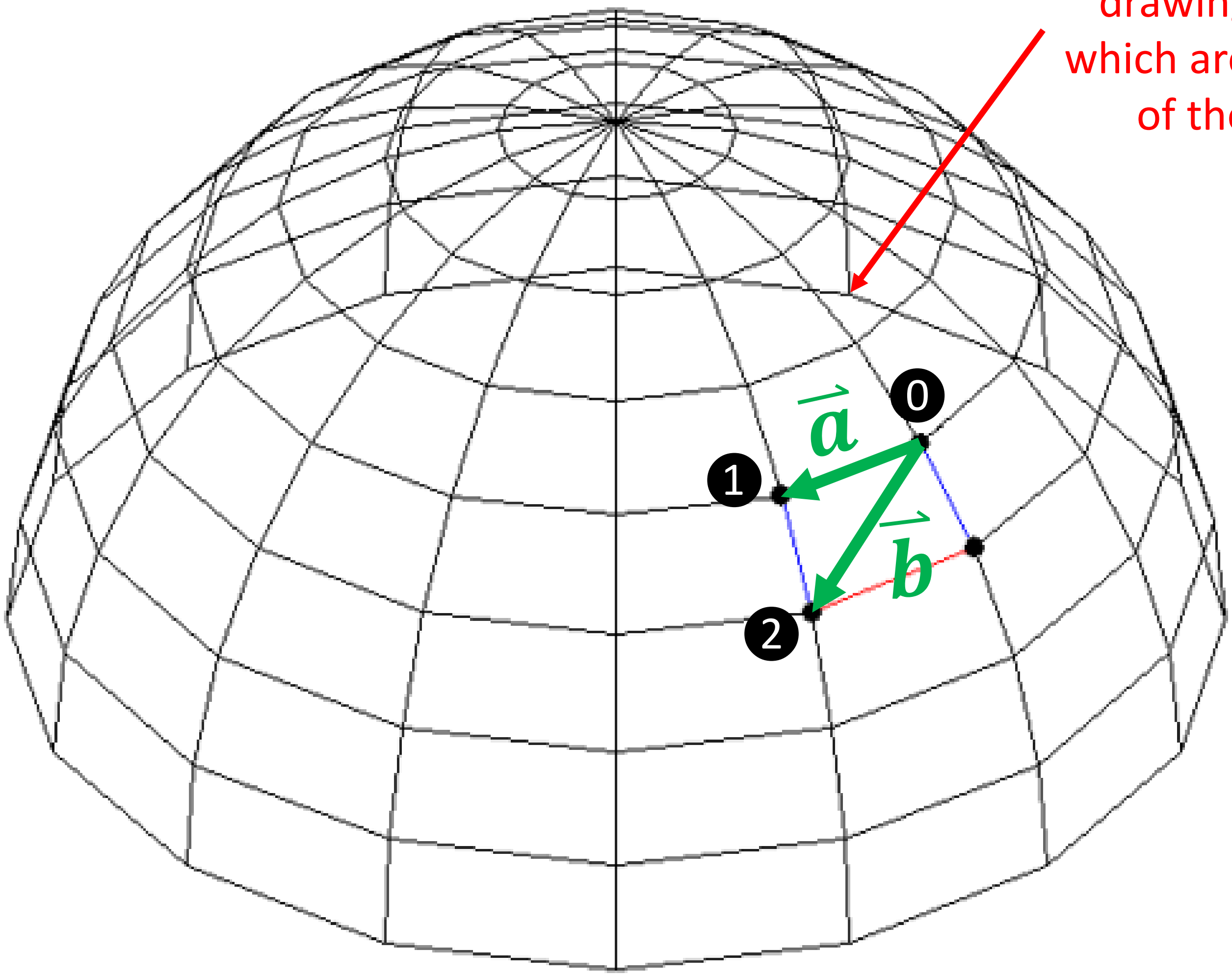




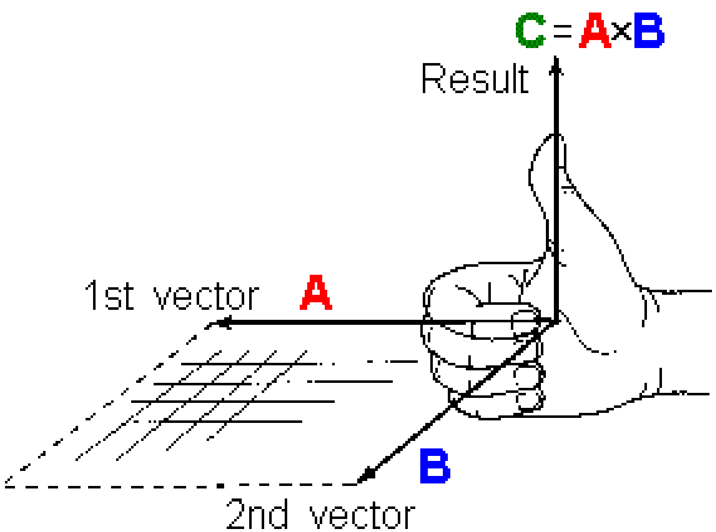
$$\boldsymbol{a_x} = P2_x - P1_x$$
$$\boldsymbol{a_y} = P2_y - P1_y$$
$$\boldsymbol{a_z} = P2_z - P1_z$$

What is a vector?

How do we avoid drawing the facets which are on the <u>back</u> of the sphere?

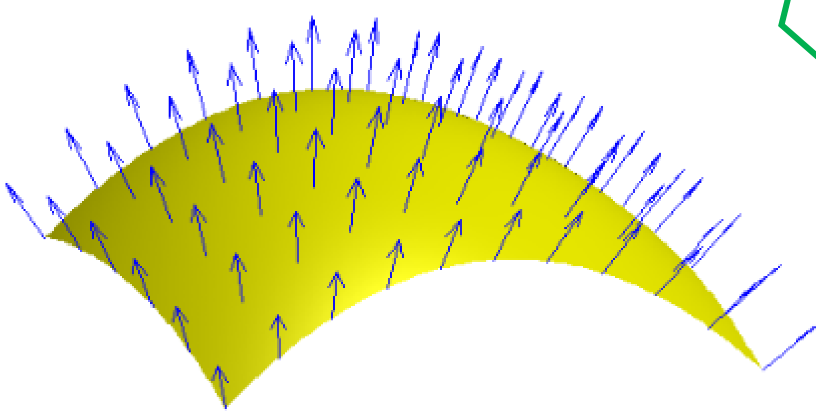


Vector Cross Product



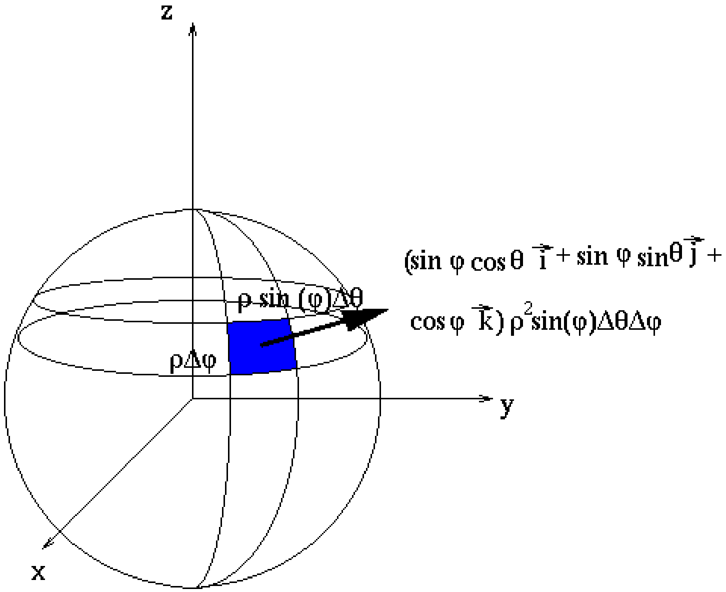
$$c = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

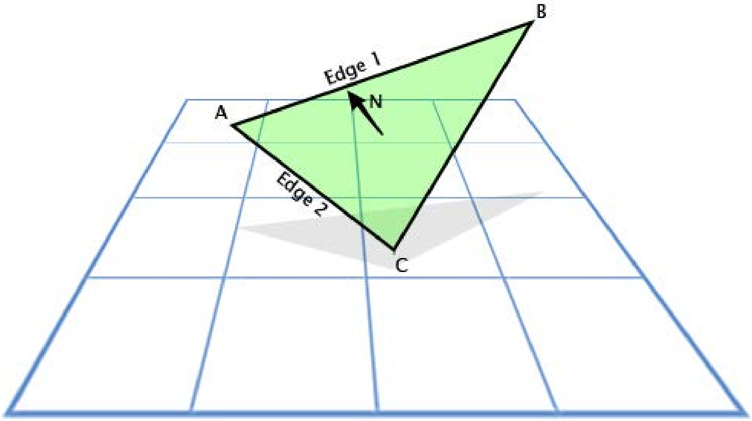
$$c = \begin{matrix} [(a_2 \times b_3) - (a_3 \times b_2)]\, i \; + \\ [(a_3 \times b_1) - (a_1 \times b_3)]\, j \; + \\ [(a_1 \times b_2) - (a_2 \times b_1)]\, k \end{matrix}$$

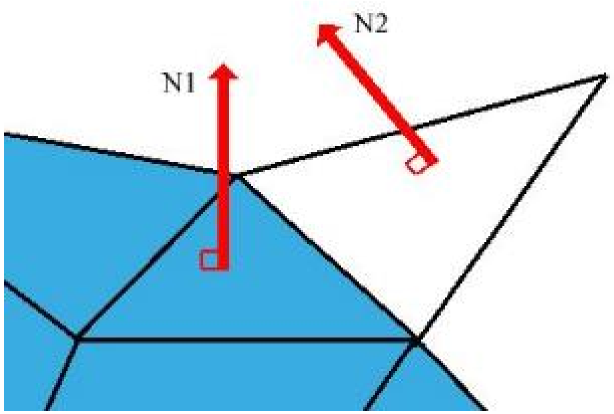


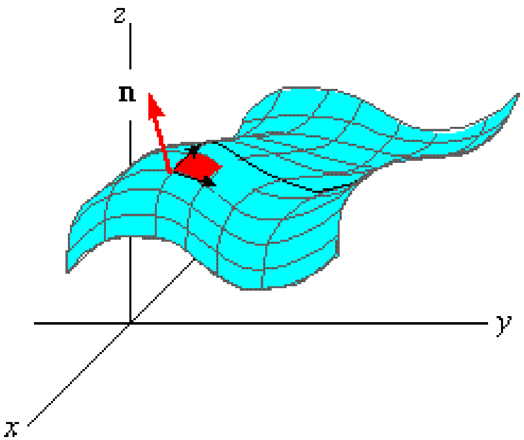
The **cross product** of two vectors is another **vector** which is perpendicular to both vectors **A** and **B**

Every Facet has a Surface Normal Vector

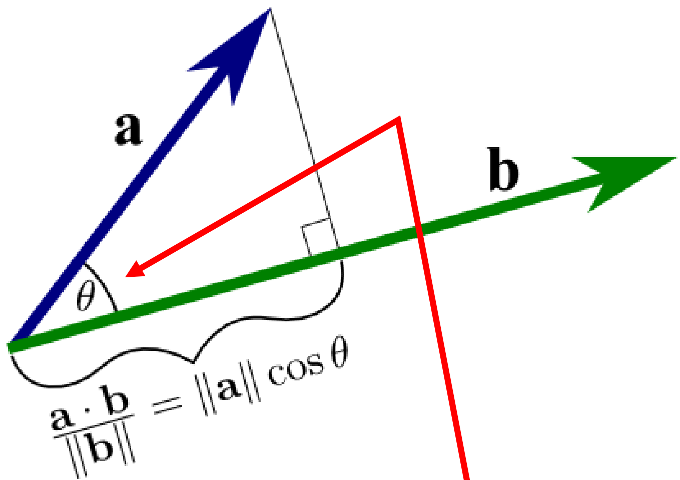








Vector Dot Product



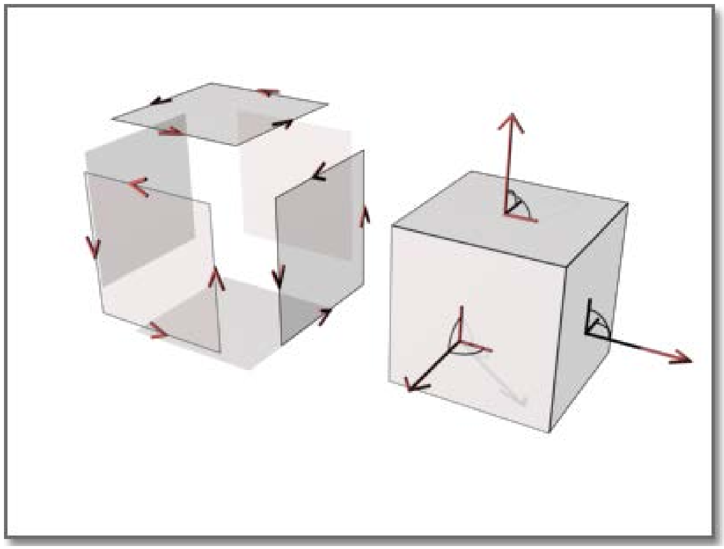
$$\mathbf{A}\cdot\mathbf{B} = \|\mathbf{A}\|\,\|\mathbf{B}\|\cos\theta$$

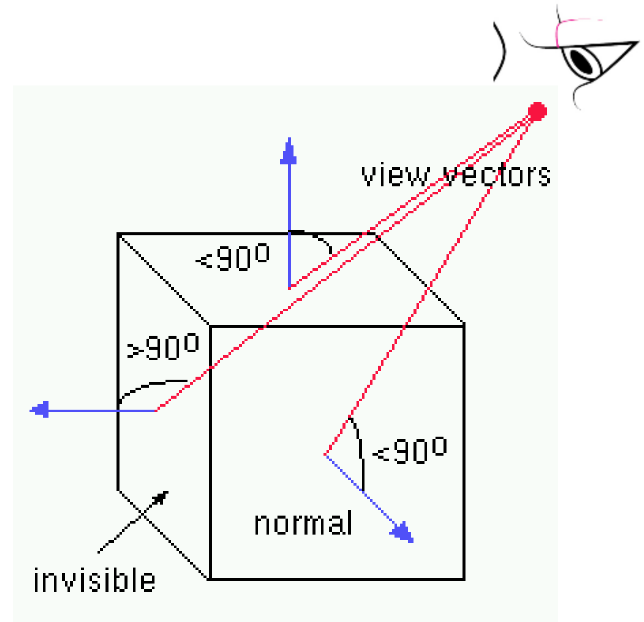
$$\cos\theta = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\,\|\mathbf{B}\|}$$

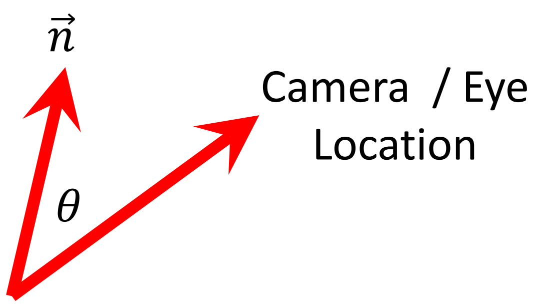
$$\theta = \operatorname{acos}\left(\frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\,\|\mathbf{B}\|}\right)$$

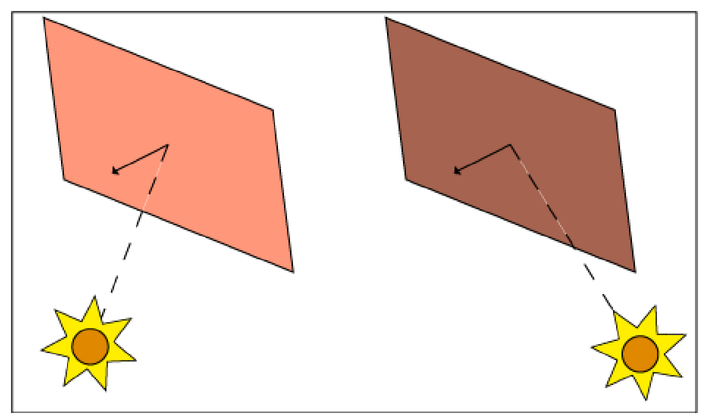
The **dot product** gives the angle *between* two **vectors**

Back Face Culling and Facet Shading









Back Face Culling

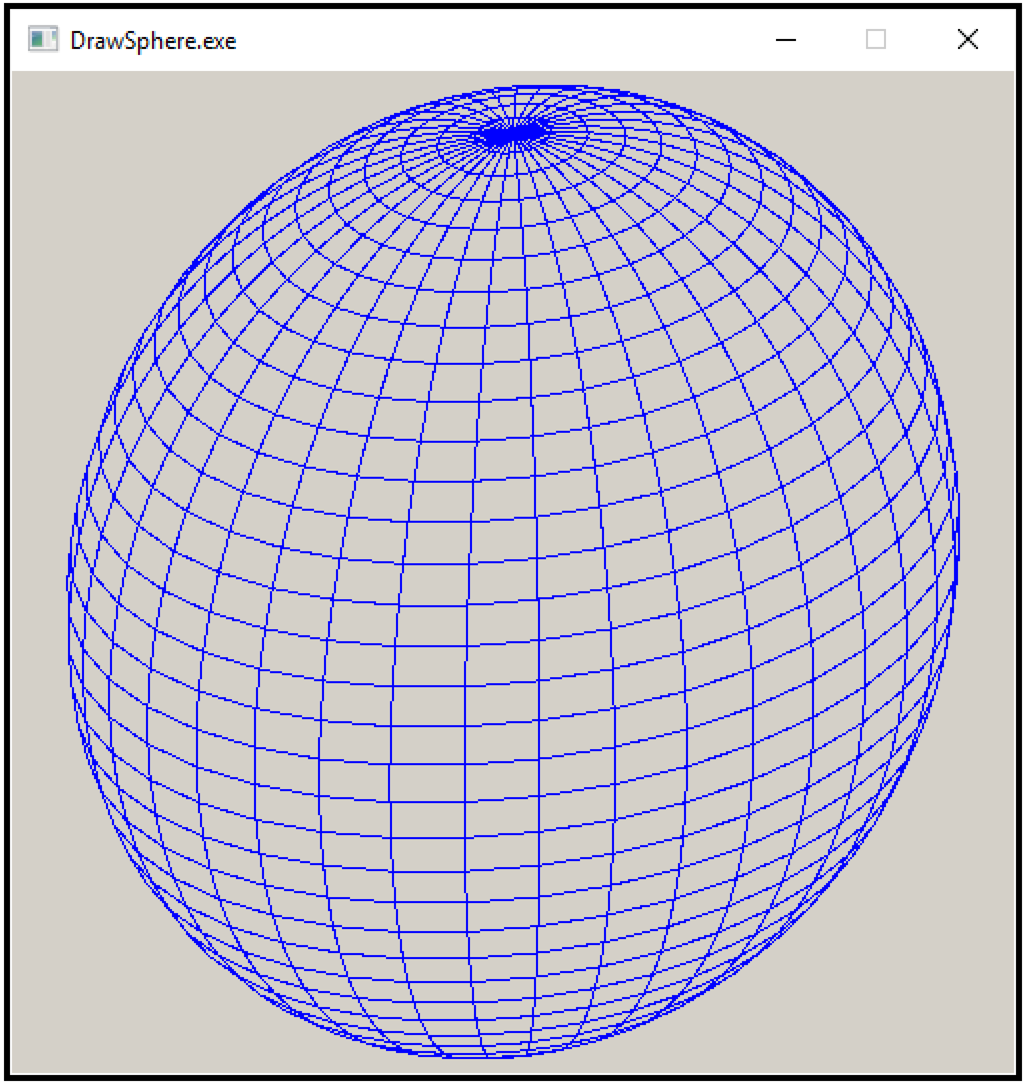
```
void SimpleScreen::DrawFacet(Facet* f, ALLEGRO_COLOR clrMin, ALLEGRO_COLOR clrMax,
    float width, bool culled, bool shaded, long delay) {
    if (shaded) culled = true;
    UnitVector* cameraVector = new UnitVector(f->center(), cameraLocation);
    double dotProduct = cameraVector->dotProduct(f->surfaceNormal());
    ALLEGRO_COLOR clr = clrMax;
    if (shaded && dotProduct >= 0) {
        // Adjust the brightness of this facet based upon dotProduct
        float red = (clrMax.r - clrMin.r) * dotProduct + clrMin.r;
        float green = (clrMax.g - clrMin.g) * dotProduct + clrMin.g;
        float blue = (clrMax.b - clrMin.b) * dotProduct + clrMin.b;
        clr = al_map_rgb_f(red, green, blue);
    }
    if (!culled || dotProduct >= 0) {
        DrawLines(f, clr, width, shaded, delay);
    }
}
```

Back Face Culling

```
            // At the North pole (phi == 0) vertex 0 and 1 are the same points,
            // so we use a different vertex number ordering to designate a
            // more meaningful surface normal for those particular facets
            Facet* f = (phi > 0) ? new Facet(vertices, { 0, 1, 2, 3 })
                : new Facet(vertices, { 2, 3, 0, 1 });

            ss.DrawFacet(f, al_map_rgb(0, 0, 0), al_map_rgb(0, 0, 255), 1, true, false, 0);
        }
    }
}

int main()
{
    SimpleScreen ss(draw);
    ss.SetZoomFrame("white", 3);
```

Re-run Lab 4

Back Face Culling



Facet Shading

```
void SimpleScreen::DrawFacet(Facet* f, ALLEGRO_COLOR clrMin, ALLEGRO_COLOR clrMax,
    float width, bool culled, bool shaded, long delay){
    if (shaded) culled = true;
    UnitVector* cameraVector = new UnitVector(cameraLocation, f->center());
    double dotProduct = cameraVector->dotProduct(f->surfaceNormal());
    ALLEGRO_COLOR clr = clrMax;
    if (shaded && dotProduct <0){
        // Adjust the brightness of this facet based upon dotProduct
        float red = (clrMax.r - clrMin.r) * abs(dotProduct) + clrMin.r;
        float green = (clrMax.g - clrMin.g) * abs(dotProduct)+ clrMin.g;
        float blue = (clrMax.b - clrMin.b) * abs(dotProduct)+ clrMin.b;
        clr = al_map_rgb_f(red, green, blue);
    }
    if (!culled || dotProduct < 0) {
        DrawLines(f, clr, width, shaded, delay);
    }
}
```

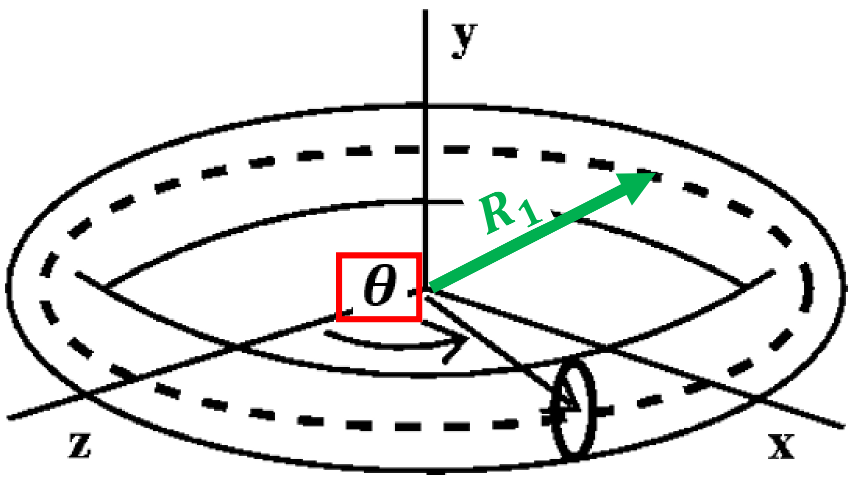
Back Face Culling

```cpp

            // At the North pole (phi == 0) vertex 0 and 1 are the same points,
            // so we use a different vertex number ordering to designate a
            // more meaningful surface normal for those particular facets
            Facet* f = (phi > 0) ? new Facet(vertices, { 0, 1, 2, 3 })
                : new Facet(vertices, { 2, 3, 0, 1 });

            ss.DrawFacet(f, al_map_rgb(0, 0, 0), al_map_rgb(0, 0, 255), 1, true, true, 0);
        }
    }
}

int main()
{
    SimpleScreen ss(draw);
    ss.SetZoomFrame("white", 3);
```

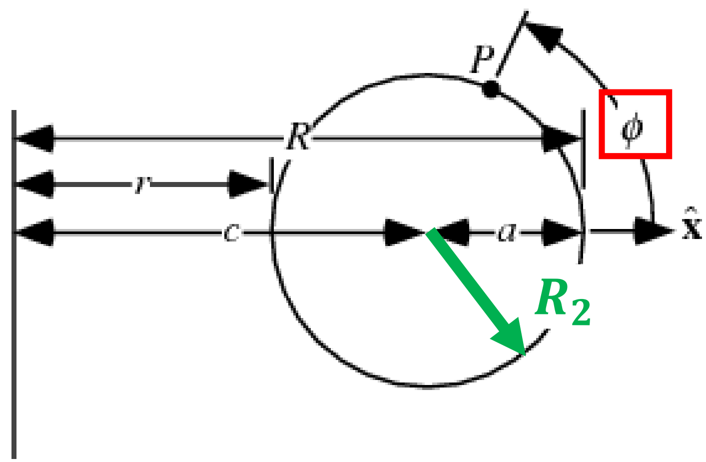
Re-run Lab 4

Facet Shading

Drawing a Torus



A torus and a sphere are not **homeomorphic** as we need **two** defining **radii** for a torus



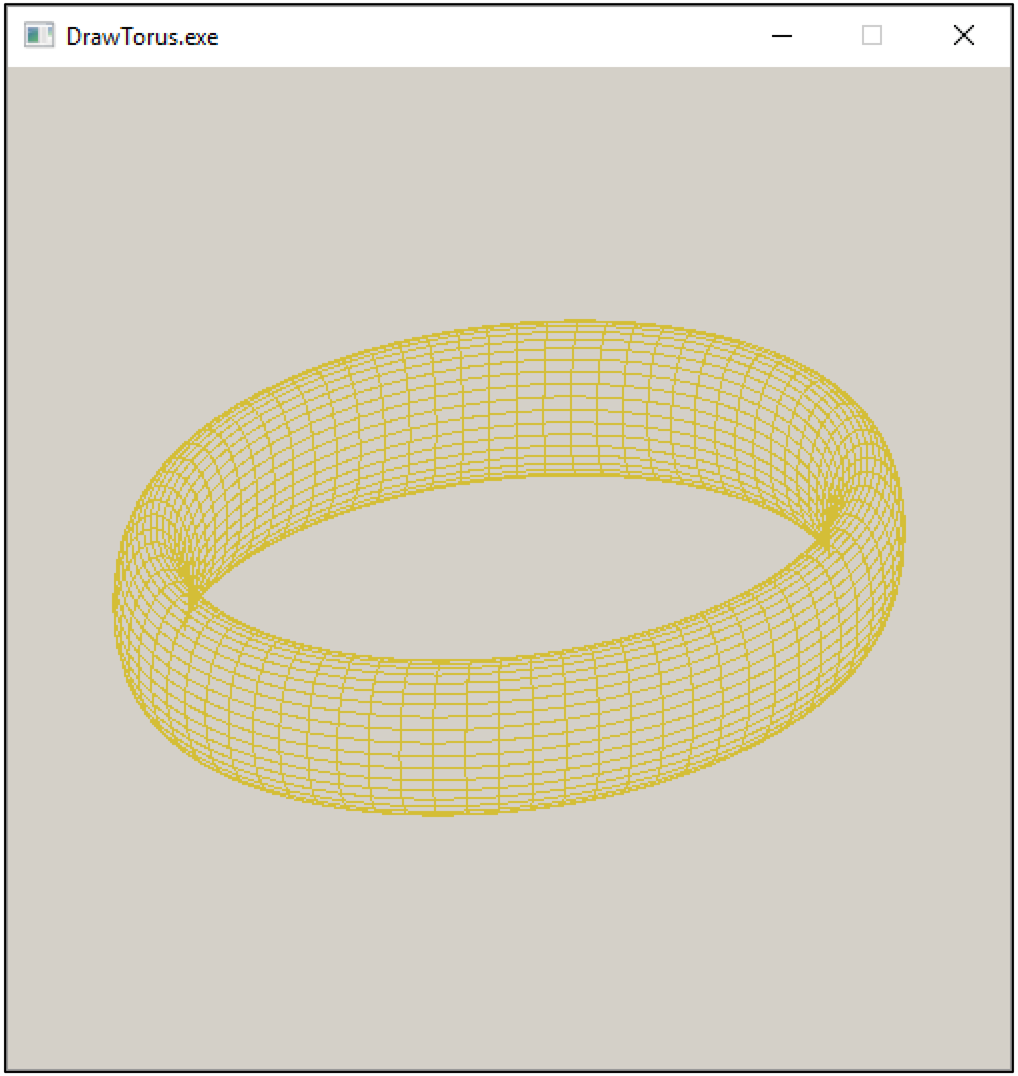
Drawing a Torus

```
void draw(SimpleScreen& ss)
{
    // Define the two radii of the torus
    double r1 = 140;     // "Donut hole" radius
    double r2 = 30;      // "Cross-sectional" radius

    // Calculate the angle deltas
    double intervals = 37;
    double deltaPhi = M_PI / intervals;         // Latitudes
    double deltaTheta = 2 * M_PI / intervals;   // Longitudes

    // Step the phi angle counter-clockwise through a full circle (expressed in radians)
    for (double phi = 0; phi < M_PI * 2; phi += deltaPhi) {

        // S
        for
```

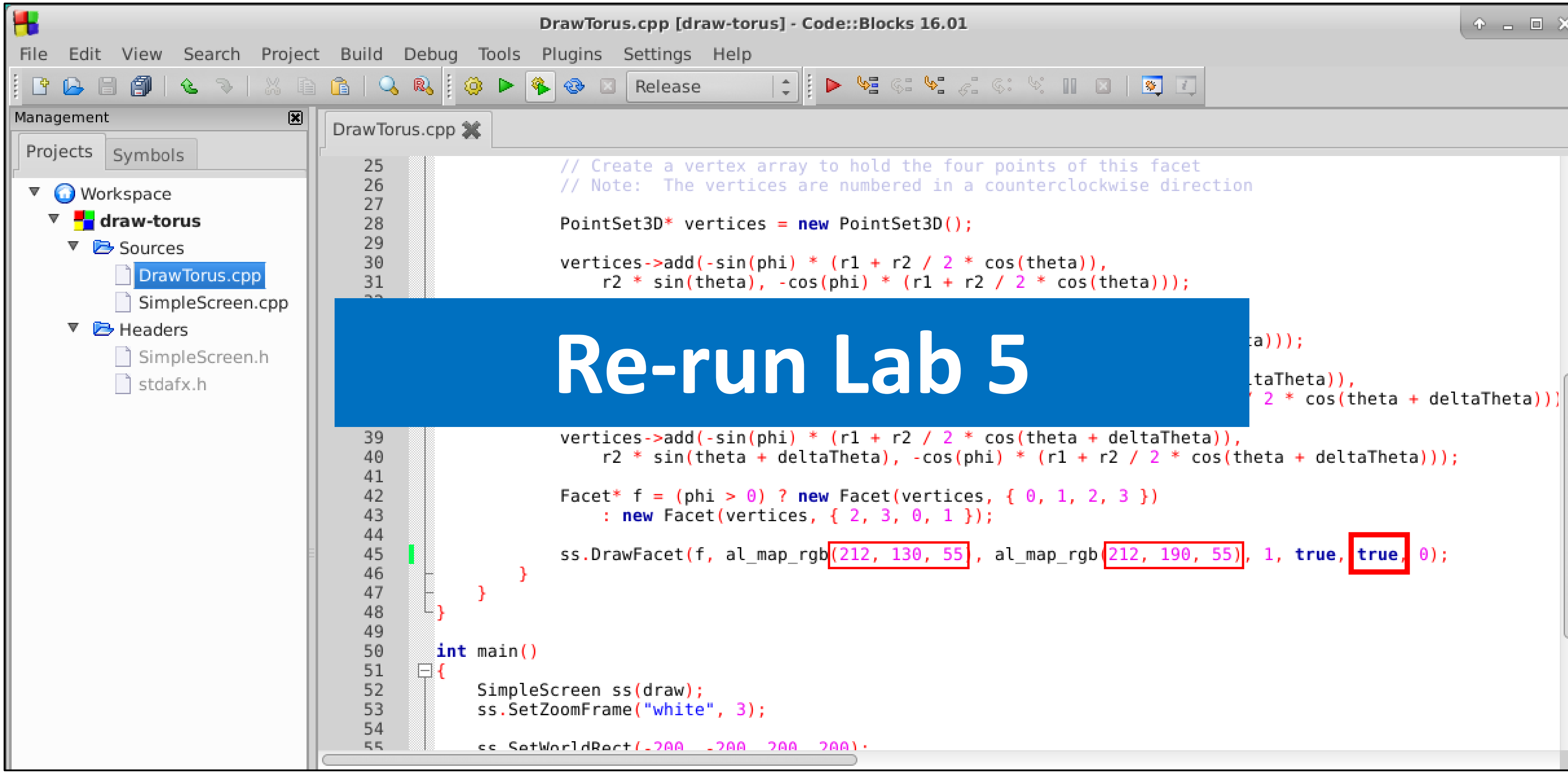
Open and run Lab 5

```
            PointSet3D* vertices = new PointSet3D();

            vertices->add(-sin(phi) * (r1 + r2 / 2 * cos(theta)),
                r2 * sin(theta), -cos(phi) * (r1 + r2 / 2 * cos(theta)));

            vertices->add(-sin(phi + deltaPhi) * (r1 + r2 / 2 * cos(theta)),
                r2 * sin(theta), -cos(phi + deltaPhi) * (r1 + r2 / 2 * cos(theta)));

            vertices->add(-sin(phi + deltaPhi) * (r1 + r2 / 2 * cos(theta + deltaTheta)),
                r2 * sin(theta + deltaTheta), -cos(phi + deltaPhi) * (r1 + r2 / 2 * cos(theta + deltaTheta)));

            vertices->add(-sin(phi) * (r1 + r2 / 2 * cos(theta + deltaTheta)),
                r2 * sin(theta + deltaTheta), -cos(phi) * (r1 + r2 / 2 * cos(theta + deltaTheta)));

            Facet* f = (phi > 0) ? new Facet(vertices, { 0, 1, 2, 3 })
                : new Facet(vertices, { 2, 3, 0, 1 });

            ss.DrawFacet(f, al_map_rgb(212, 130, 55), al_map_rgb(212, 190, 55), 1, true, false, 0);
        }
    }
}
```

Drawing a Torus

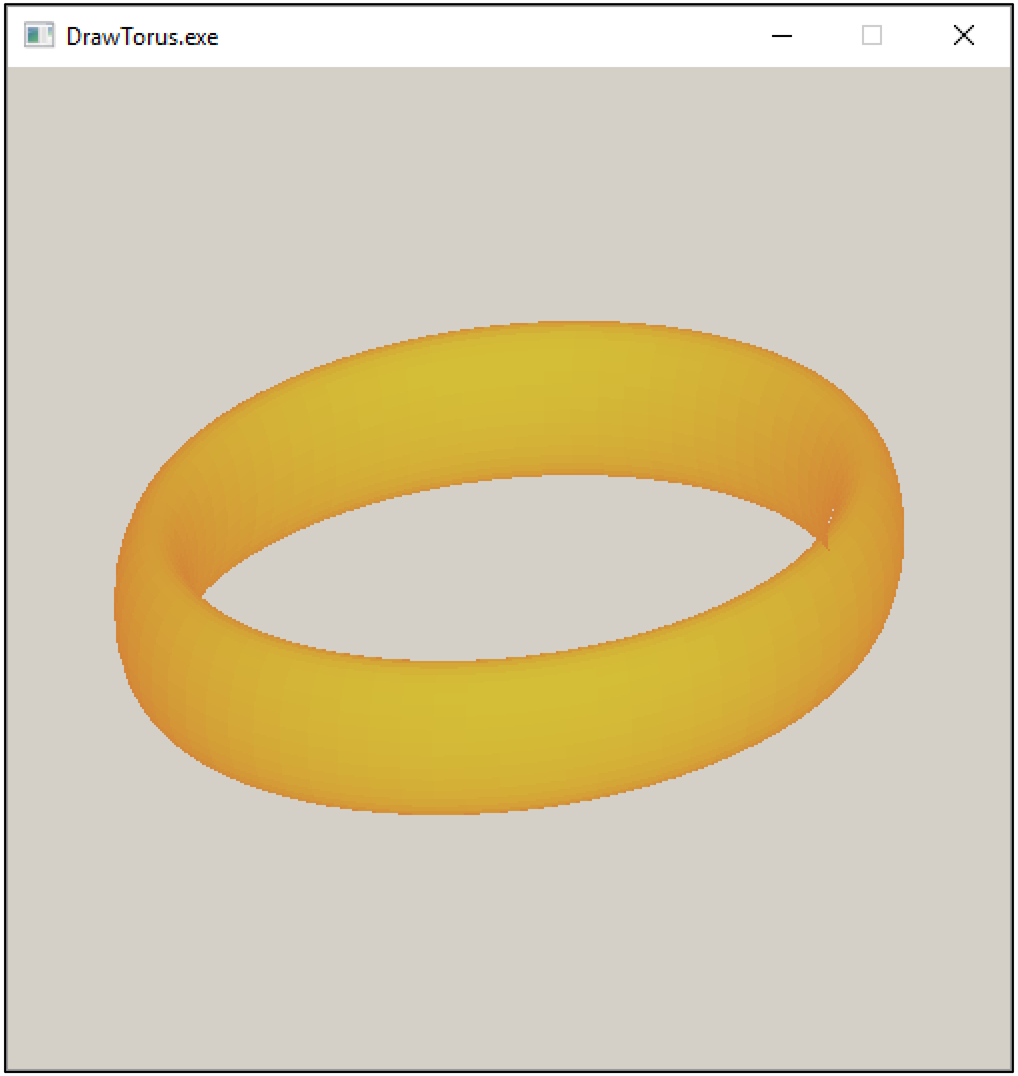


Drawing a Torus



Re-run Lab 5

Drawing a Torus



Now you know…

- We can use the **zeroes** and **extrema** of a **polynomial** to determine an appropriate **world rectangle** to frame it
- 3D **Cartesian** coordinates use (x, y, z) while 3D **Spherical** coordinates use (r, θ, ϕ)
  - An **oblique projection** (2.5D) shows a ¾ "side" view
  - Sets of **vertices** are assembled into **facets**
- A surface **normal** vector is a **cross product** of two facet edge vectors following the right-hand rule
  - A **dot product** expresses the angle <u>between</u> two vectors
  - The dot product between the **camera vector** and each facet normal enables **back face culling** and **facet shading**