Reference Axes

3D space (with 6 DFs) is usually described with a reference axis.

XNA uses a right-hand coordinate system

positive Z towards viewer

Positive rotations about an axis are counter-clockwise looking along the axis to the origin.

Or, clockwise looking from the origin along the axis.

Place right hand around "axis" with thumb pointing along the positive direction and the finger curl is in the positive direction.
Vertex based "shapes"

Models (*.x) loaded contain vertices and indices.

Shapes can be defined/drawn by surface.

This pyramid has 6 surfaces:
- 4 on the sides
- 2 on the bottom

These surfaces are defined by 5 vertices ("points")
- Each vertex has a position – a Vector3 (x, y, z)
- Vertices can also have a color or a texture and a normal

MonoGames defines several vertex formats (see docs)
- VertexPositionColor
- VertexPositionColorTexture
- VertexPositionNormalTexture

MonoGames supports creation of custom vertex formats --
- VertexPositionColorNormal

<< see "Creating a Custom Vertex Declaration"
Declarations and initializations

Consider the following declarations and initializations.

```csharp
BasicEffect effect;
GraphicsDevice display;
int nVertices = 5;
VertexPositionColor[] vertex;
VertexBuffer vb = null;
int nIndices = 18;
IndexBuffer ib = null;  // better declaration later
Vector3 lightDir, lightColor;

// constructor initializes
graphics = new GraphicsDeviceManager(this);
display = graphics.GraphicsDevice;
effect = new BasicEffect(display);
lightColor = Color.White.ToVector3();
// light direction should be a unit normal vector
lightDir = new Vector3(1.0f, 0.0f, 0.0f);
```
// allocate vertex array – in constructor
vertex = new VertexPositionColor[nVertices];
// create vertices
vertex[0] = new VertexPositionColor(
    new Vector3(-10,  0,  10), Color.Red);
vertex[1] = new VertexPositionColor(
    new Vector3(  0, 10,   0), Color.Green);
vertex[2] = new VertexPositionColor(
    new Vector3( 10,  0,  10), Color.Blue);
vertex[3] = new VertexPositionColor(
    new Vector3( 10,  0, -10), Color.Cyan);
vertex[4] = new VertexPositionColor(
    new Vector3(-10,  0, -10), Color.Orange);
Vertices are placed in a VertexBuffer and stored on the GPU. This is done in `LoadContent()`.

```csharp
// create VertexBuffer
VertexBuffer vb = new VertexBuffer(display, typeof(VertexPositionColor),
                                  vertexColored.Length, BufferUsage.WriteOnly);

// and store on GPU
vb.SetData<VertexPositionColor>(vertex, 0, vertex.Length);
```

The pyramid has 5 vertices and 6 surfaces. These surfaces can be drawn with the Primitive Type `TriangleList`. This would require duplicate copies of vertices in the VertexBuffer, since each `TriangleList` would have 3 vertex values.

\[ 18 \text{ vertices} = 3 \div \text{surface} \times 6 \text{ surfaces} \]
Duplicate vertices can be eliminated from the VertexBuffer by having an IndexBuffer reference VertexBuffer vertices.

Winding order (clockwise) determines visibility of face from view. Visible when surface unit normal + view normal <= 0

```csharp
private IndexBuffer ib = null;
// define face vertex index clockwise from camera view
private int[] indices = {
    0, 1, 2, // Front Face
    2, 1, 3, // Right Face
    3, 1, 4, // Back Face
    4, 1, 0  // Left Face
    0, 2, 4, // Bottom Face
    2, 3, 4, // Bottom Face
};
...
// create IndexBuffer and store on GPU in LoadContent()
ib = new IndexBuffer(display, typeof(int), indices.Length,
    BufferUsage.WriteOnly);
ib.SetData<int>(indices);
```
public override void Draw(GameTime gameTime) {
    effect.VertexColorEnabled = true;
    // set effect lights and other effect attributes
    // set effect matrices: World, View, Projection
    foreach (EffectPass pass in
        effect.CurrentTechnique.Passes) {
        pass.Apply();
        display.SetVertexBuffer(vb);
        display.Indices = ib;
        display.DrawIndexedPrimitives(
            PrimitiveType.TriangleList, 0, 0, nVertices,
            0, nIndices/3);
    }
}

<<see Drawing with Vertices >>
Normals

Normals are vectors orthogonal to a plane. The cross product of 2 vectors sharing a vertex is its normal.

\[
\text{Vector3.Cross(vector1, vector2);}
\]

MonoGames has a [VertexPositionNormalTexture format](#).

```
VertexPositionNormalTexture vertexPNT =
    new VertexPositionNormalTexture(
        new Vector3(x, y, z), // position
        Vector3.Up,          // normal "up" (0, 1, 0)
        new Vector2(u, v));  // texture coordinate
```

Normals used for:

- surface visibility
- surface shading (surface lighting reflection)

A normalized vector has a length of 1

\[
\text{Vector3.Normalize(aVector3);} 
\]
Textures

Textures are 2D images that can be used as a surface’s “material”.

Texture coordinates are (u,v) and range from 0.0 … 1.0

A triangle with vertices having (0,0), (0, 1), (1,1) u,v coordinates and the DirectX puck.bmp texture.

Texture2D puckTexture = Content.Load<Texture2D>("puck");
Lights

A point on a surface is seen as reflected light from a view.

Light has a position and color that illuminates the surface. Some light sources have attenuation – intensity dimishes w/ distance.

A surface has material (color, texture, and reflective) properties.

BasicEffect light has direction and diffuse color.
Shading

4 contributions to the shading of a point (not BasicEffect)

diffuse   light reflected in all directions ( !V )
specular  shininess of reflections from smooth surface (R V)
ambient   sum of all light source (intesity of light, !R !V)
emissive  glowing – light source (!L, lights ! visible)

Imagine a white light in room with green walls
   diffuse and specular parameters are white
   reflections are green – so the ambient parameter is green

Light properties can be set in MonoGames’ BasicEffect, or can be programmed in effect shaders.

recall BasicEffect has 3 lights – 0, 1, 2

effect.DirectionalLight0.DiffuseColor = lightColor;
effect.DirectionalLight0.Direction = lightDir;
Pyramids example

Left pyramid
  VertexPositionColor
  vertex unique colors, shading

Right pyramid
  VertexPositionNormalTexture
  puck.bmp texture

Each pyramid needs its own
  BasicEffect, and
  VertexBuffer

Pyramids can
  share an IndexBuffer

See PyramidsDistro.zip
posted example.