Introduction to Game Graphics

Behrouz Minaei

Iran University of Science and Technology

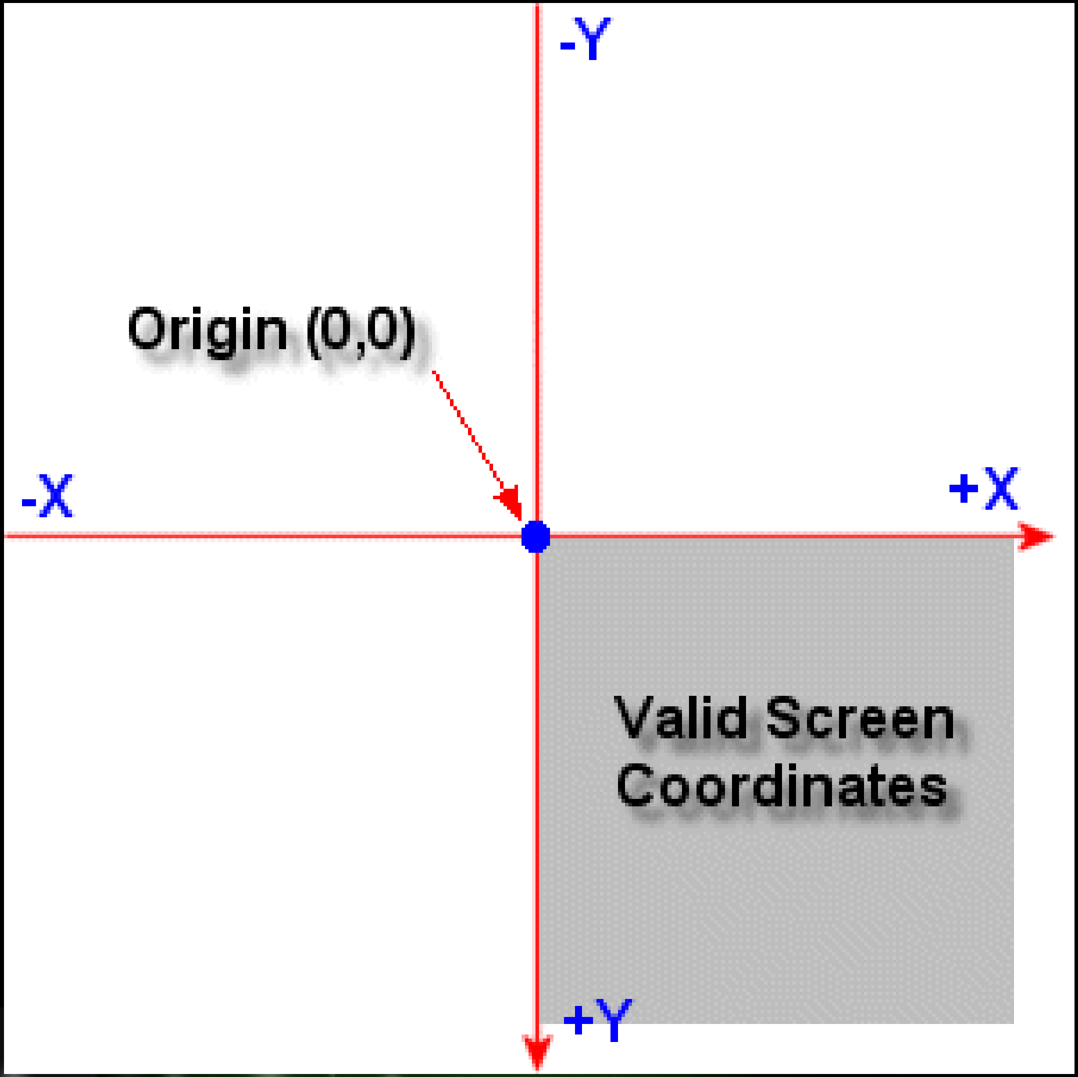
2D V.S 3D

- 2D games are constructed from bitmap units called sprites. By moving these sprites you can enforce the sense of motion

- But instead we have to represent our world mathematically as a series of connected points which are plotted in a 3D coordinate System.

- So 3D games should have something which is called transformation pipeline which converts this mathematical presentation into something that can be rendered and presented in the 2D screen

2D V.S 3D



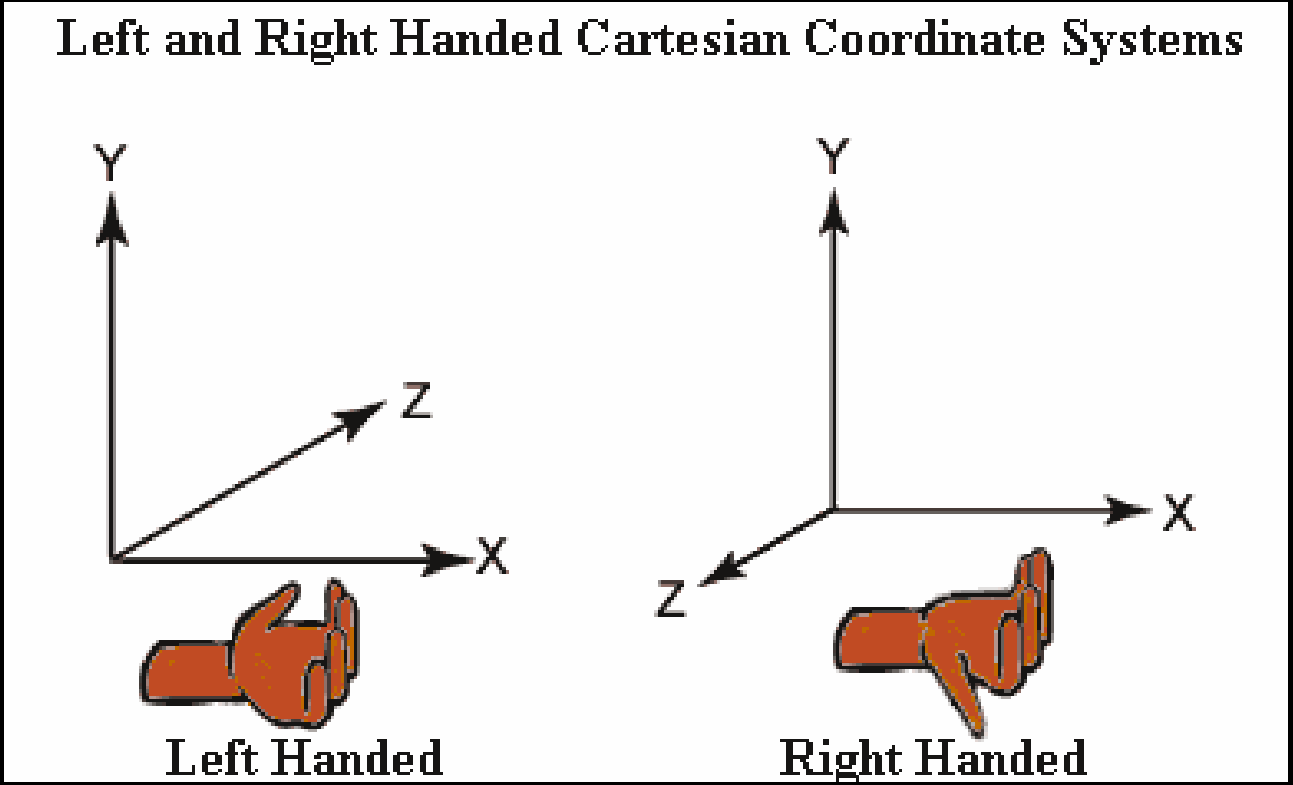
2D V.S 3D

- But before learning the transformation pipeline we have to learn how the 3D world behaves.

- 3D objects are represented in computer games using an area of mathematics called “Geometry”.

- At its simplest geometry is the process of representing an object using a series of points and change the appearance of that object by changing those points.

Coordinate Systems

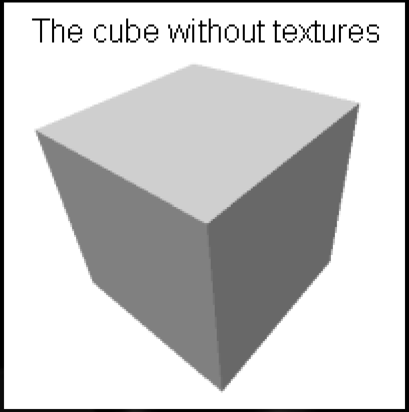


So what coordinate we should use?

- DirectX uses a left handed coordinate system
- OpenGL on the other hand uses a right handed coordinate system

What is a mesh?

- A mesh is a collection of polygons that are joined together to create the outer hull of the object being defined.
- Each polygon in the mesh (often referred to as a face), is created by connecting a collection of points defined in three dimensional space with a series of line segments.
- If desired, we can ‘paint’ the surface area defined between these lines with a number of techniques .For example, data from two dimensional images called texture maps can be used to provide the appearance of complex texture and color



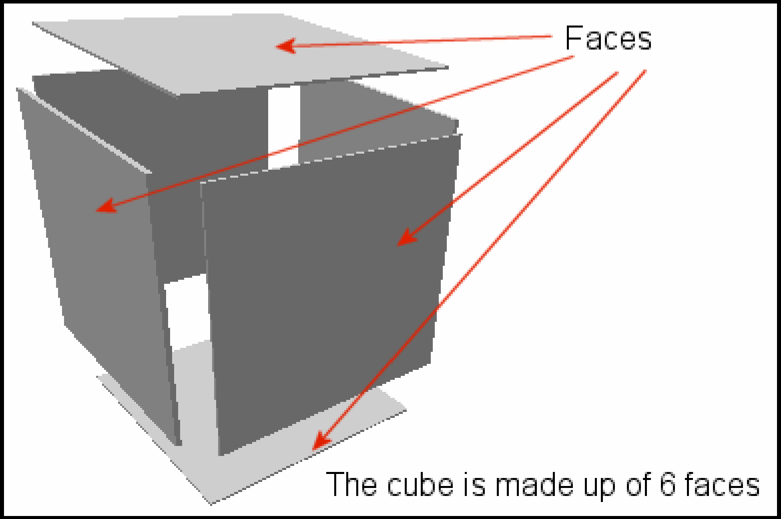
The Mesh of a wooden crate

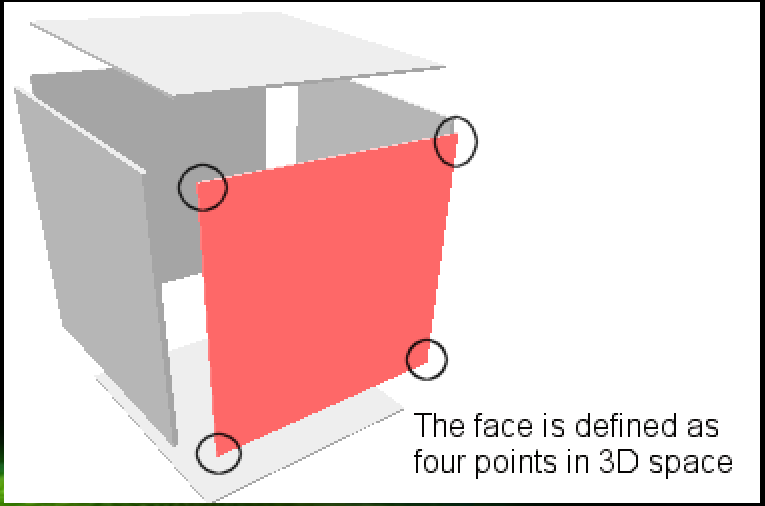
What is a mesh?

```
struct Vertex
{
 float x ; float y; float z;
};
```

```
struct Polygon
{
 UINT    VertexCount;
 Vertex *  VertexList;
};
```

```
struct Mesh
{
  UINT FaceCount;
  Polygon * FaceList;
};
```

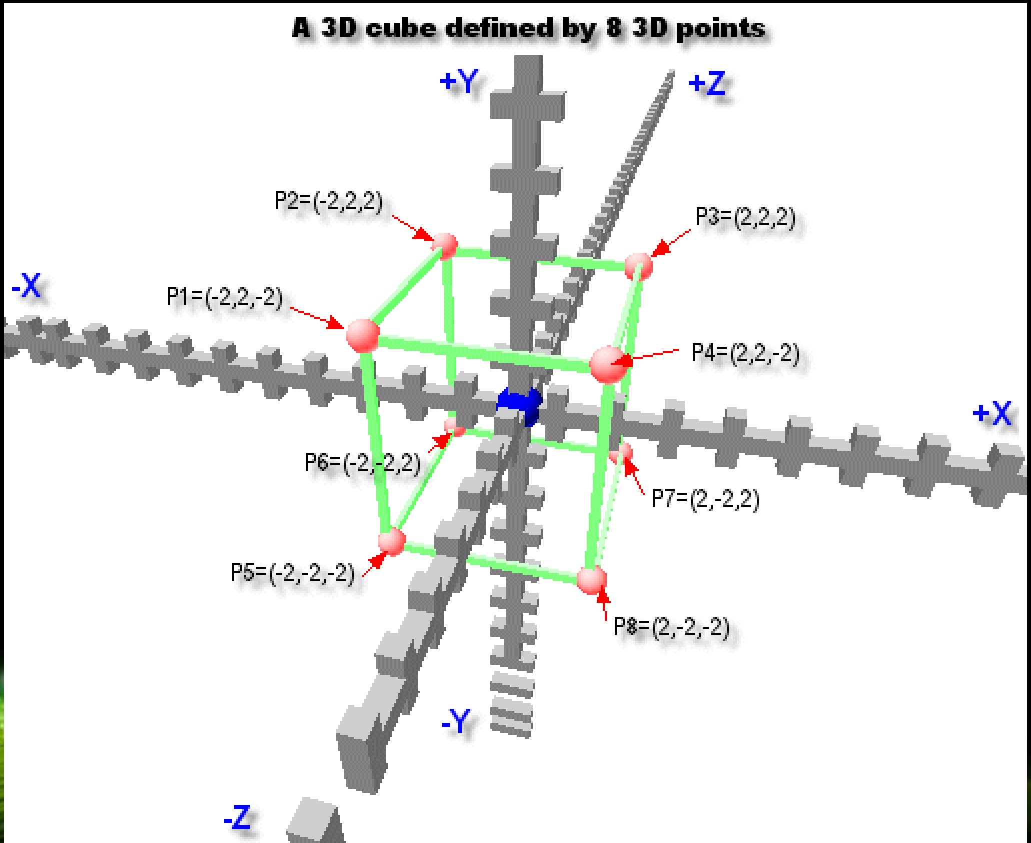




Model Space

•The cubic mesh can be defined by the eight vertex in which their position is defined relative to the origin of the coordinate system.

•***Origin of the coordinate system is at the center of the mesh.***



Model Space Coordinate System

•This is what is known as defining the mesh at "Model space coordinates".

•***The 3D mathematical representation of the mesh is defined at model space where it has no formal relation to any other meshes that the application may have created.***

•***Each mesh when gets created is defined at the model space initially.***

•***We will see that one of the jobs of transformation pipeline is to transform all the meshes from the model space coordinate system to a shared global coordinate space call "The world space"***

Back face Culling and winding order of vertices

•But we cannot specify vertices in any desired form!

•Because otherwise the renderer won’t be able to determine how to connect two vertices with a line (what vertex will be the origin of the line and what vertex will be the end of the line!)

•The answer is that the winding order (the order that you define your vertices to represent a face) should be clockwise

Back face Culling and winding order of vertices

•3D models will not usually be created programmatically but will be created within a modeling package such as 3D Studio Max™.

•This allows us to create scenes with thousands or even millions of polygons. Very high polygon counts often correlate to a reduction in application performance due to the increased volume of calculations that need to be performed when drawing them.

•As a graphics developer you will use a number of techniques to keep the number of polygons that need to be drawn in a given frame to a minimum.

•Certainly you would not want to render polygons that the user could not possibly see from their current position in the virtual world.

•One such optimization discards polygons that are facing away from the viewer; this technique is called ***back face culling.***

Back face Culling and winding order of vertices

•For this reason, 3D rendering engines normally perform a fast and cheap test before rendering a polygon to see if it is facing the viewer. When it is not it can be discarded.

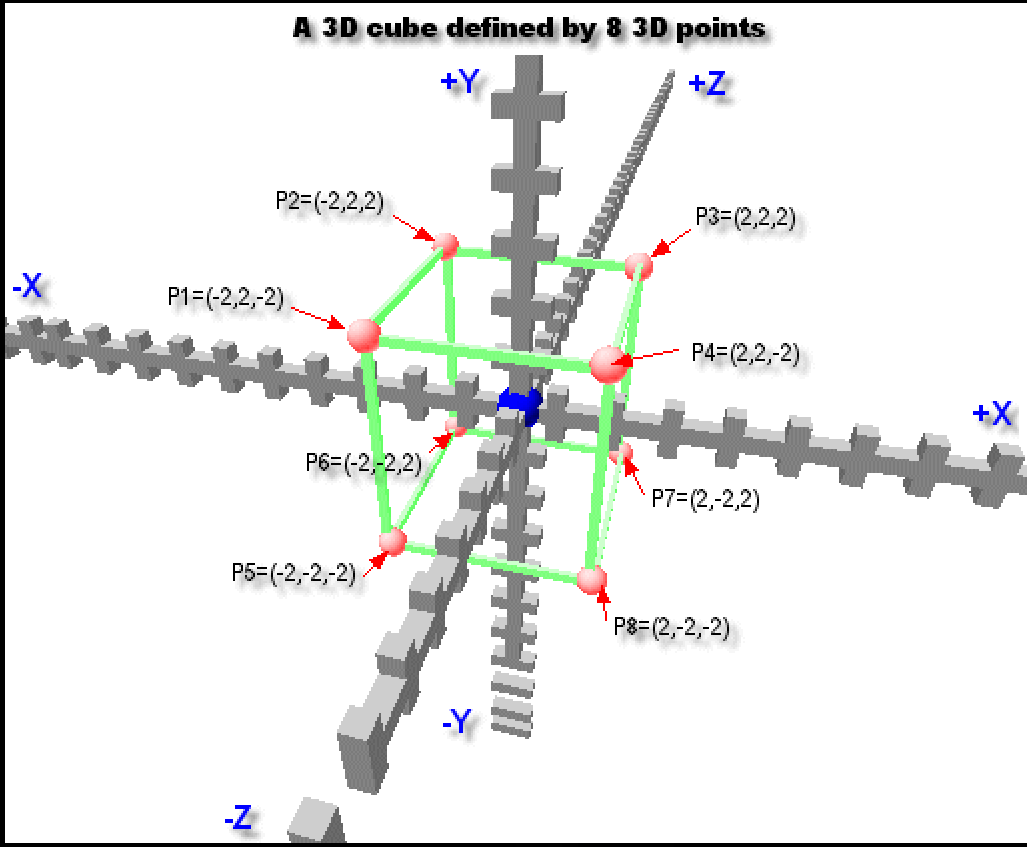
•So the point of view is really important in this check. Because being clockwise or counter clock wise is dependent on from what point we see the mesh.

Back face Culling and winding order of vertices

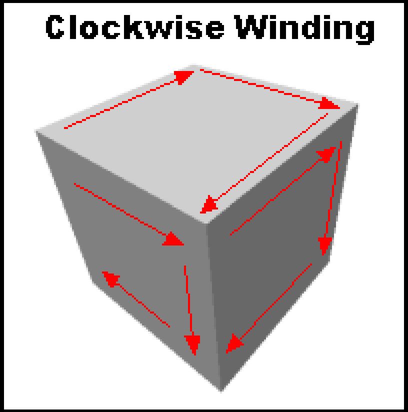
•The coordinate P1 is used to create a vertex in the left face, the top face and the front face. And so on for the other coordinates.

•Also note that the vertices are specified in an ordered way so that lines can be drawn between each pair of points in that polygon until the polygon is finally complete.

•The order in which we specify the vertices is significant and is known
•as the **winding order.**



Back face Culling and winding order of vertices

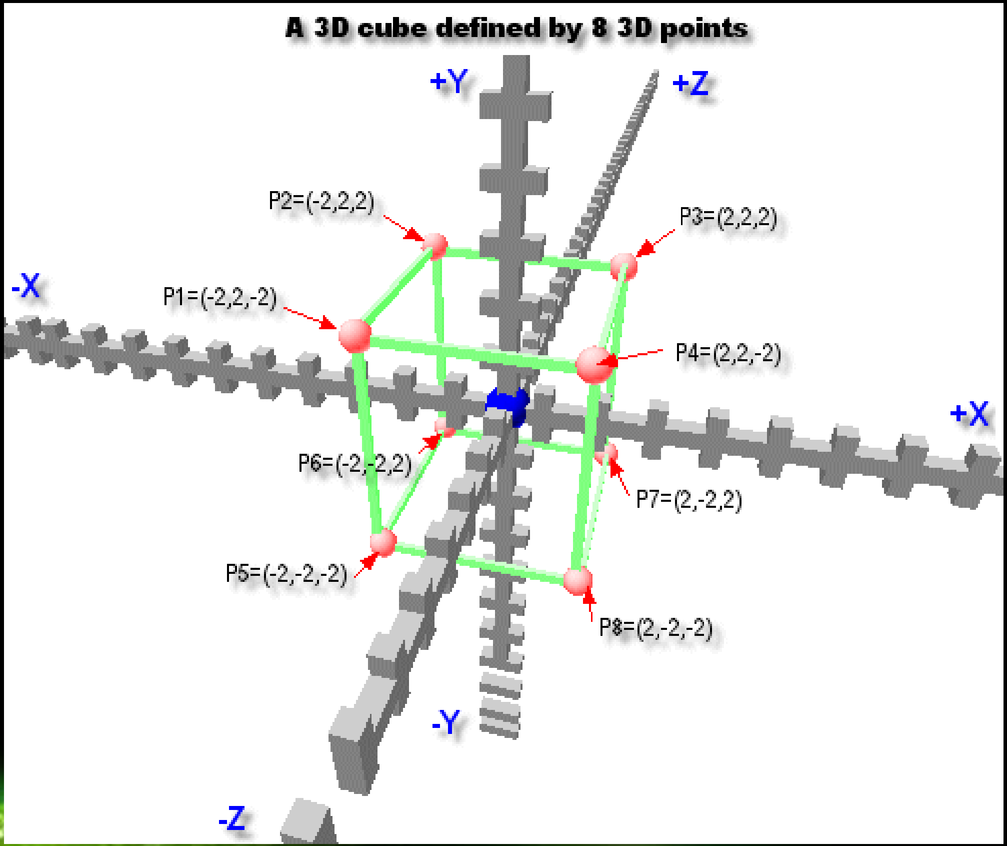


Front Face (P1 , P4 , P8 , P5)

```
pPoly = m_Mesh.m_pPolygon[0];
if ( pPoly->AddVertex( 4 ) < 0 ) return false;
pPoly->m_pVertex[0] = CVertex( -2,  2, -2 );  // P1
pPoly->m_pVertex[1] = CVertex(  2,  2, -2 );  // P4
pPoly->m_pVertex[2] = CVertex(  2, -2, -2 );  // P8
pPoly->m_pVertex[3] = CVertex( -2, -2, -2 );  // P5
```

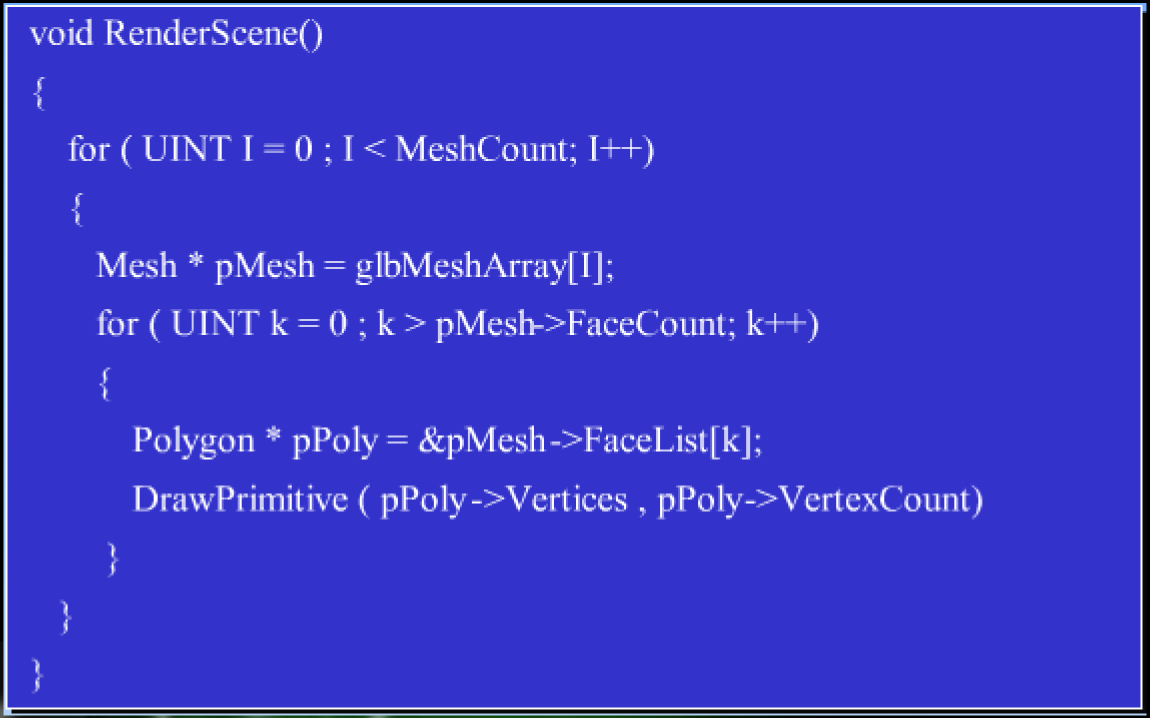
Back Face (P6 , P7 , P3 , P2)

```
pPoly = m_Mesh.m_pPolygon[2];
if ( pPoly->AddVertex( 4 ) < 0 ) return false;
pPoly->m_pVertex[0] = CVertex( -2, -2,  2 );  // P6
pPoly->m_pVertex[1] = CVertex(  2, -2,  2 );  // P7
pPoly->m_pVertex[2] = CVertex(  2,  2,  2 );  // P3
pPoly->m_pVertex[3] = CVertex( -2,  2,  2 ),  // P2
```



Scene Rendering

- The game world (scene) is made up of a collection of meshes.
- At first, displaying the world seems like the easy task of looping through each mesh and then looping through each polygon in that mesh and drawing them!



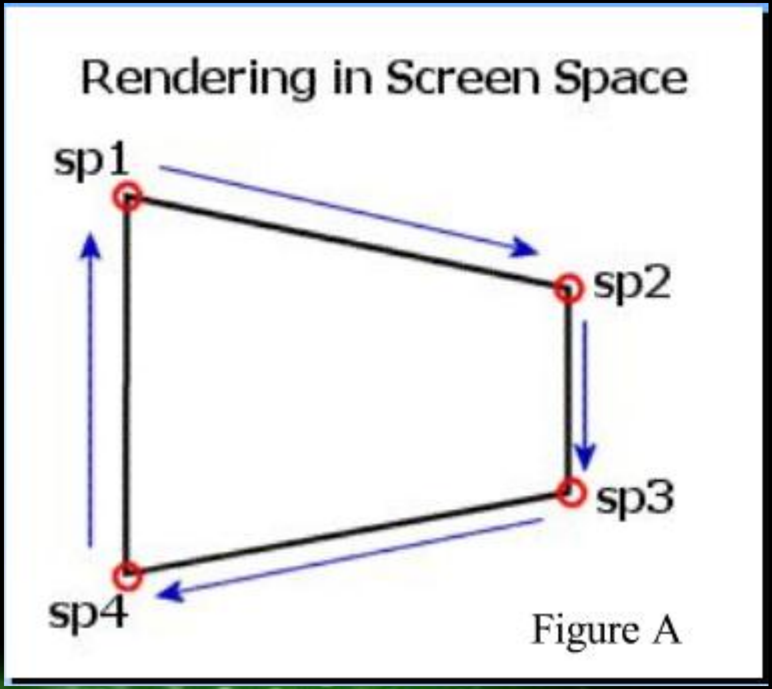
```
void RenderScene()
{
   for ( UINT I = 0 ; I < MeshCount; I++)
   {
     Mesh * pMesh = glbMeshArray[I];
     for ( UINT k = 0 ; k > pMesh->FaceCount; k++)
     {
        Polygon * pPoly = &pMesh->FaceList[k];
        DrawPrimitive ( pPoly->Vertices , pPoly->VertexCount)
     }
   }
}
```

Scene Rendering

- While this is partially true, remember that the meshes at this point means nothing at the screen coordinate system.
- The DrawPrimitive function in the code needs to be rewritten such that it converts the model 3D coordinate space to the screen 2D coordinate System.
- This is what transformation pipeline of our code needs to do

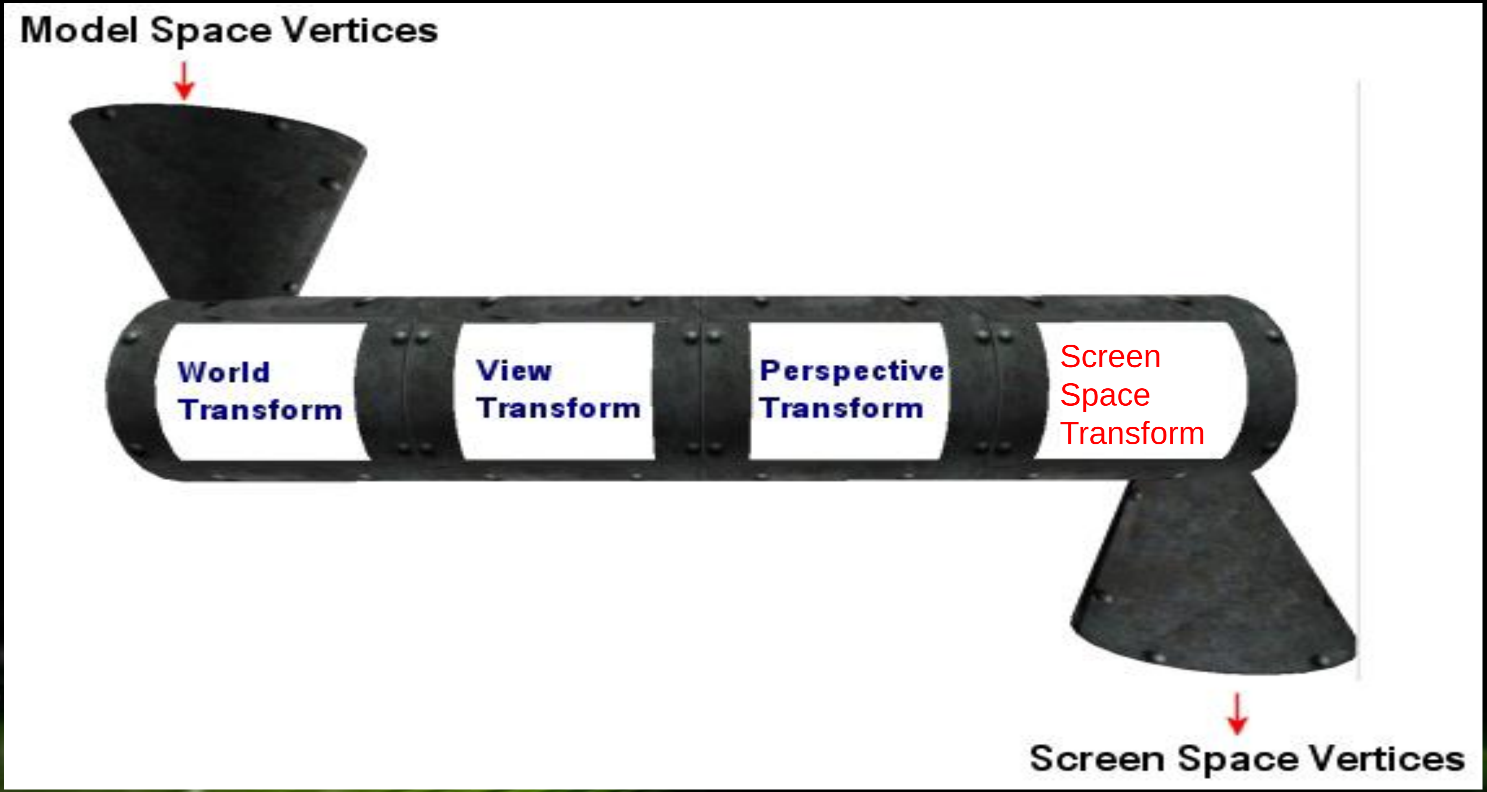
Scene Rendering

- Once we have the vertices of the polygon transformed from 3D coordinates into screen space coordinates, each vertex describes the location of a pixel in the screen. Rendering the wireframe polygon is done by drawing lines in a clockwise manner between vertices.



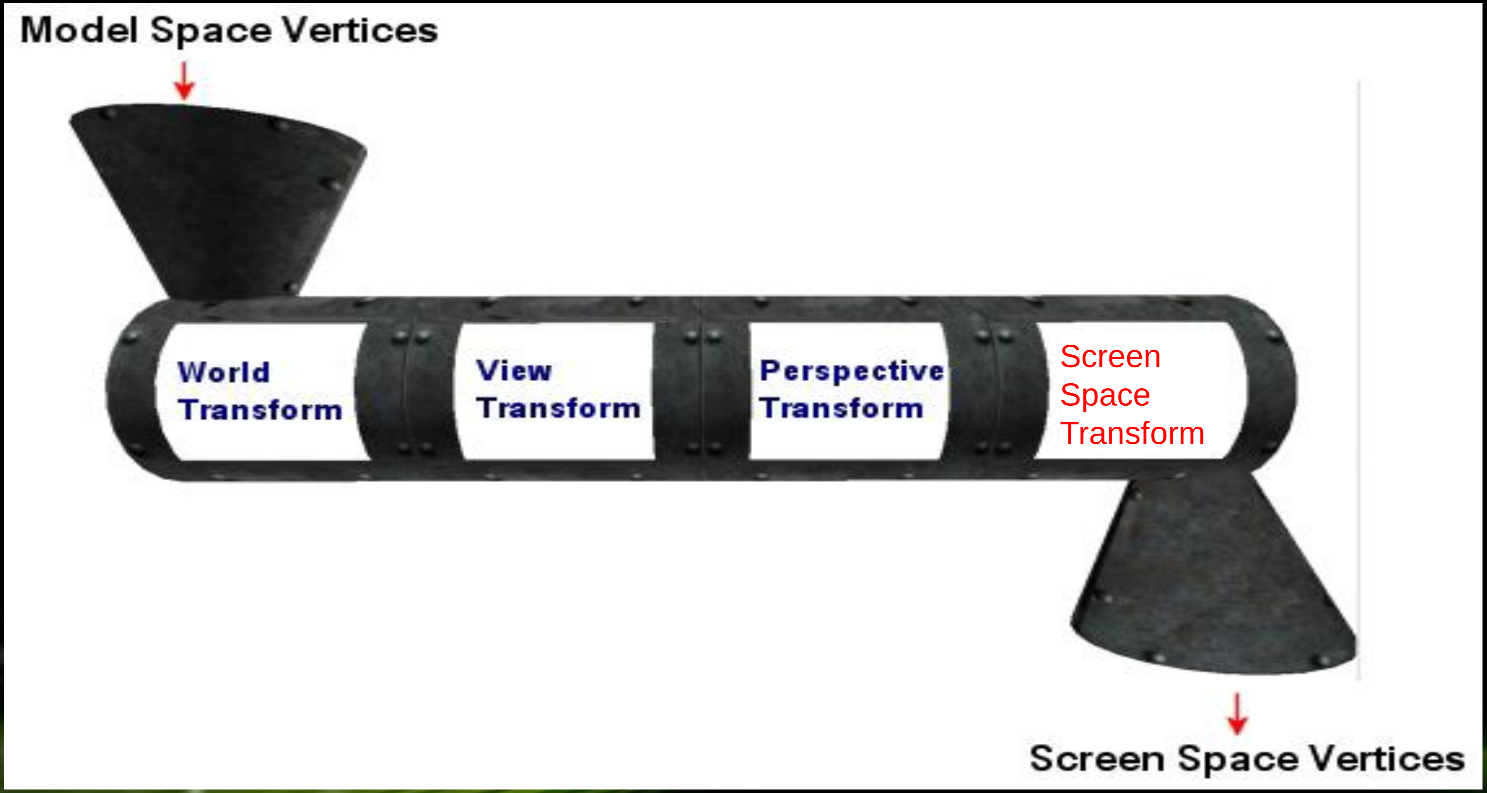
Transformation Pipeline

- The code that is responsible to convert vertices from 3D coordinate system to screen space coordinates is called transformation pipeline
- We call it a pipe line because steps are done sequentially in it.
- The outputted vertices of each step is injected to the next one.



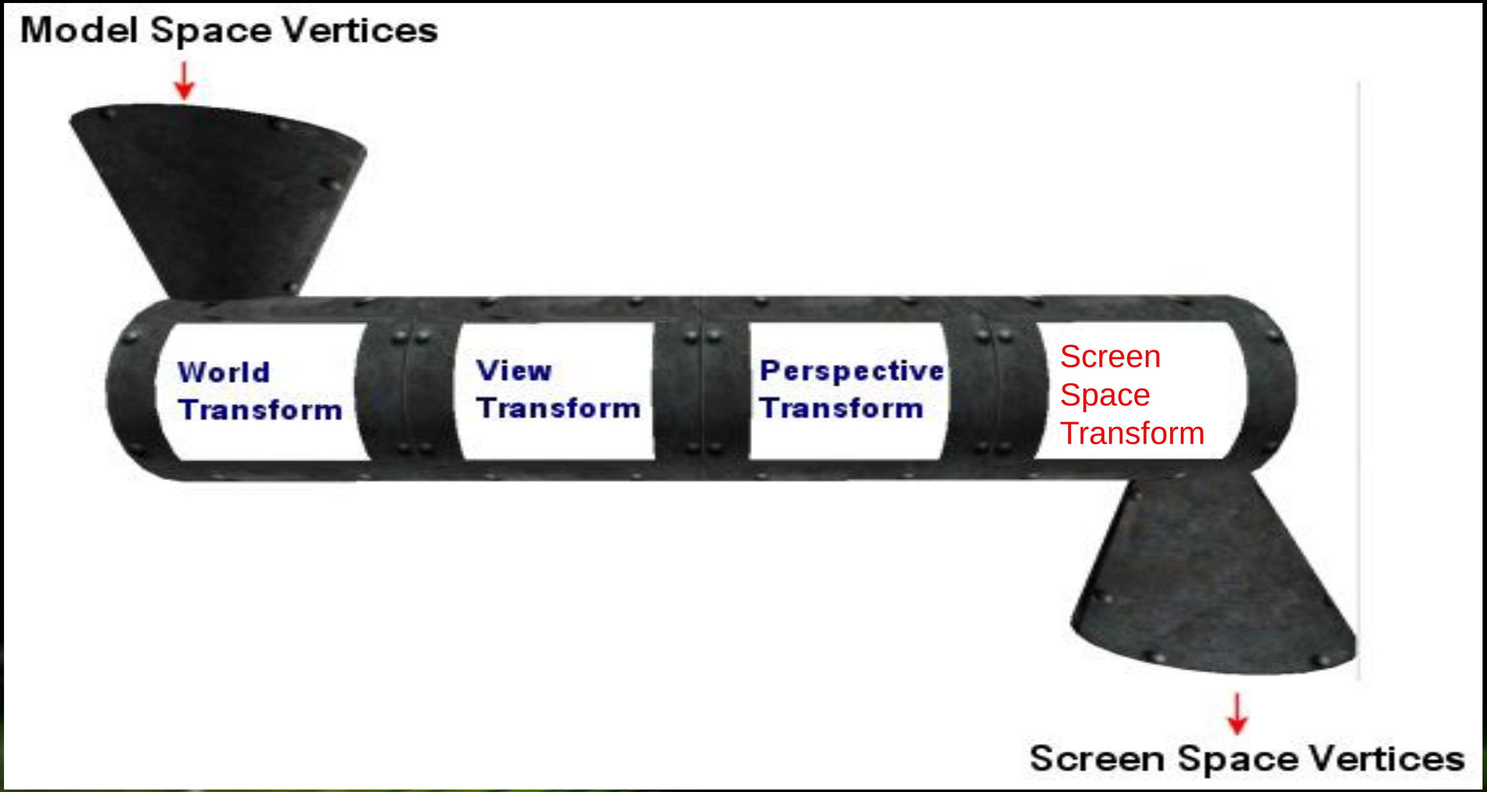
Transformation Pipeline

- First the vertices in the model space should get converted into the world space in a process called “world transformation”.
- In the model space the origin of each mesh is at its center point and thus the position of each vertex is relative to that location.
- In world transformation the origin will be the origin of the world. In other words we are placing the model into the world.
- If we don’t do this all the meshes will be at (0,0,0) !



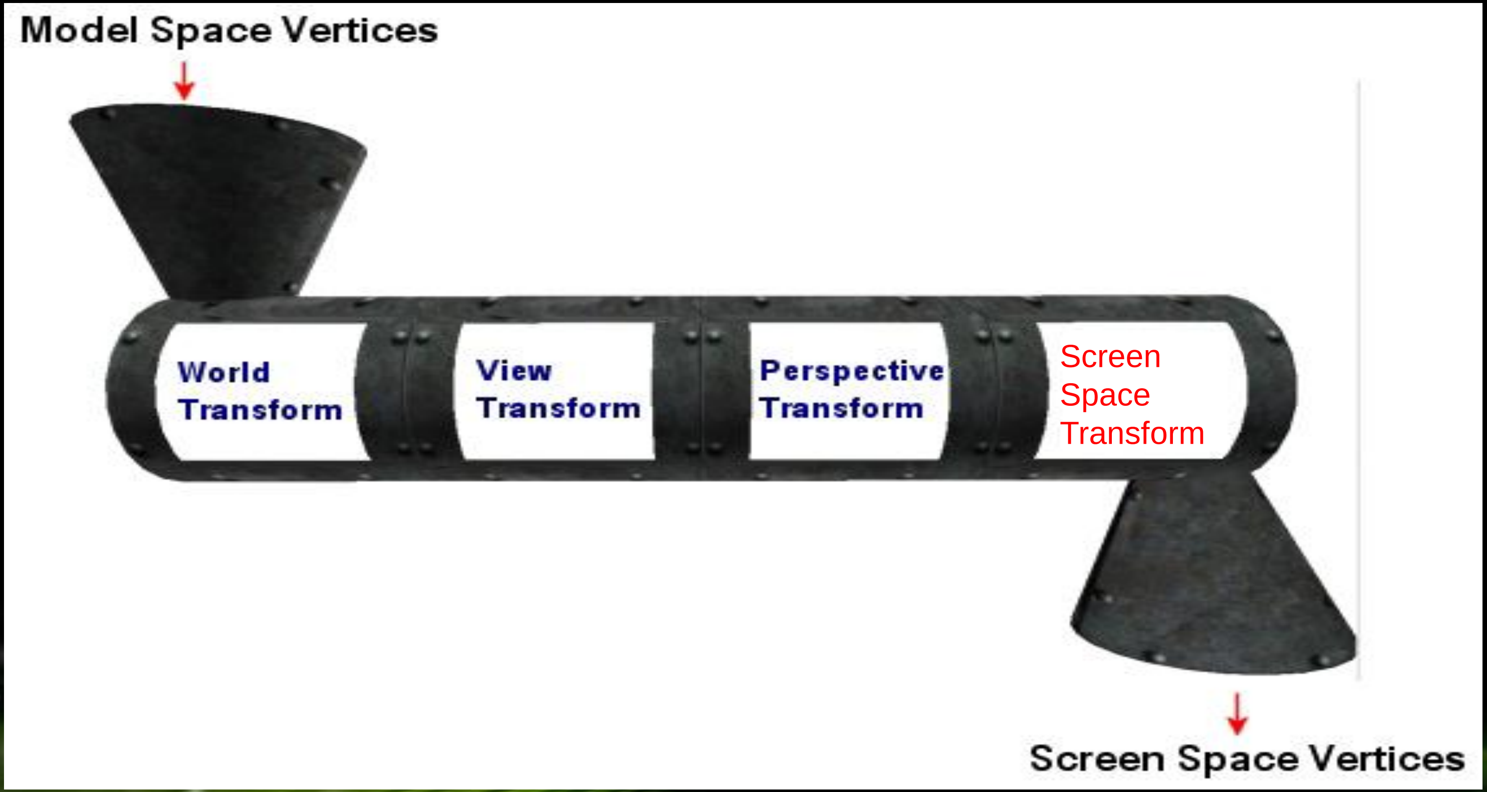
Transformation Pipeline

- The view transformation converts the coordinates from the world space to the coordinates in the view space
- This enables us to place a camera at any location in our world (not just the origin) and see the world from that location.
- The view space transformation would allow us to fake the viewer into thinking that he or she is able to move in 3d space around the world. Whereas the opposite is true! The world around the player will transform and move and the player will stand still.



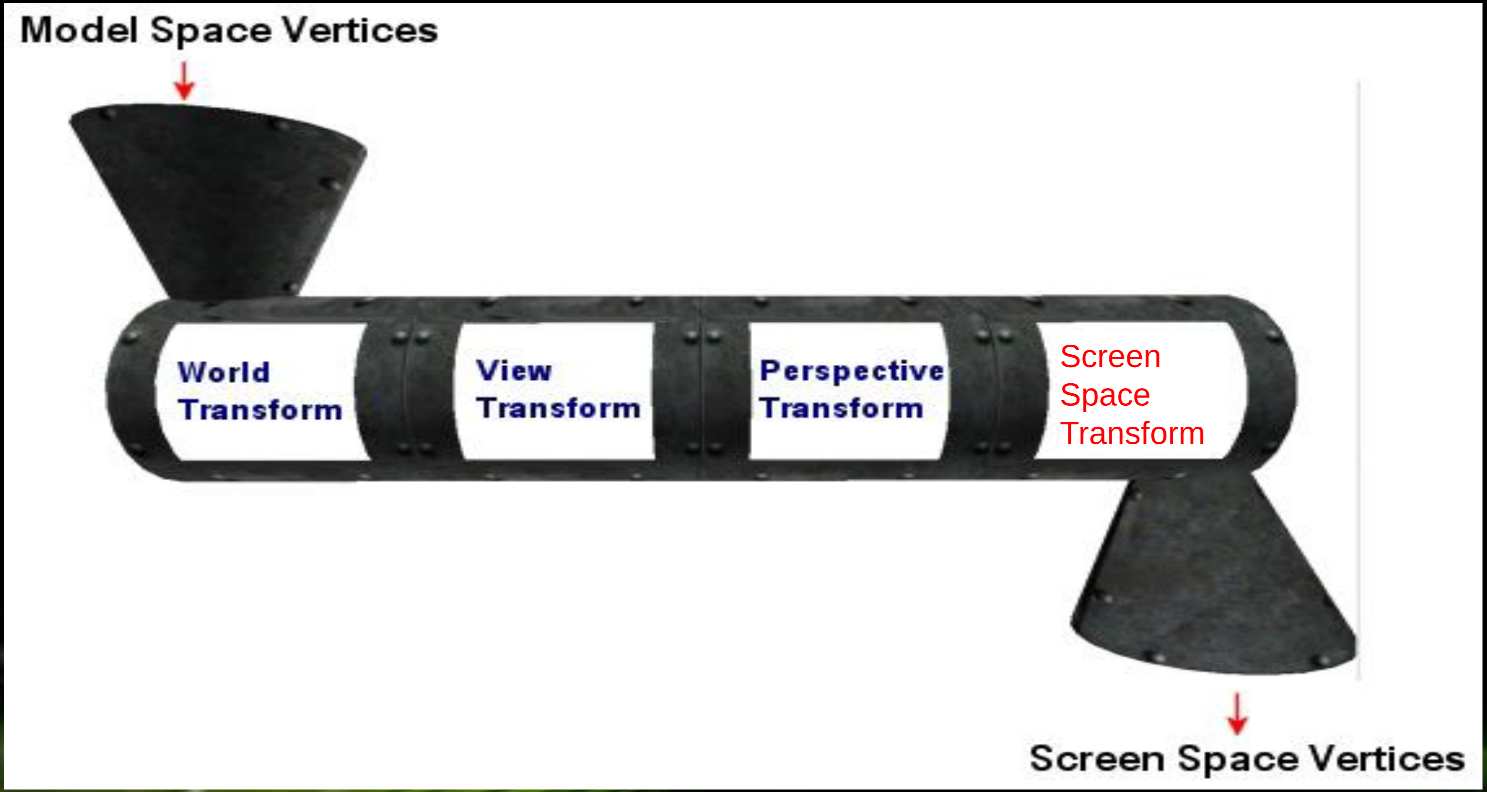
Transformation Pipeline

- Once the view space transformation has positioned the vertices relative to the camera the perspective projection transforms the 3D view space vertices into 2D projection space vertices.
- This space is in the range of -1 to 1 along x and y axis for all the vertices that are visible and are determined to be in the field of view.



Transformation Pipeline

- All that is left to do now is to map the -1 to 1 coordinates to the range and coordinate space of the screen (for example converting (1-,1) to (1024,768))
- Each vertex then has a 2D pixel coordinate describing where it should be rendered on the screen



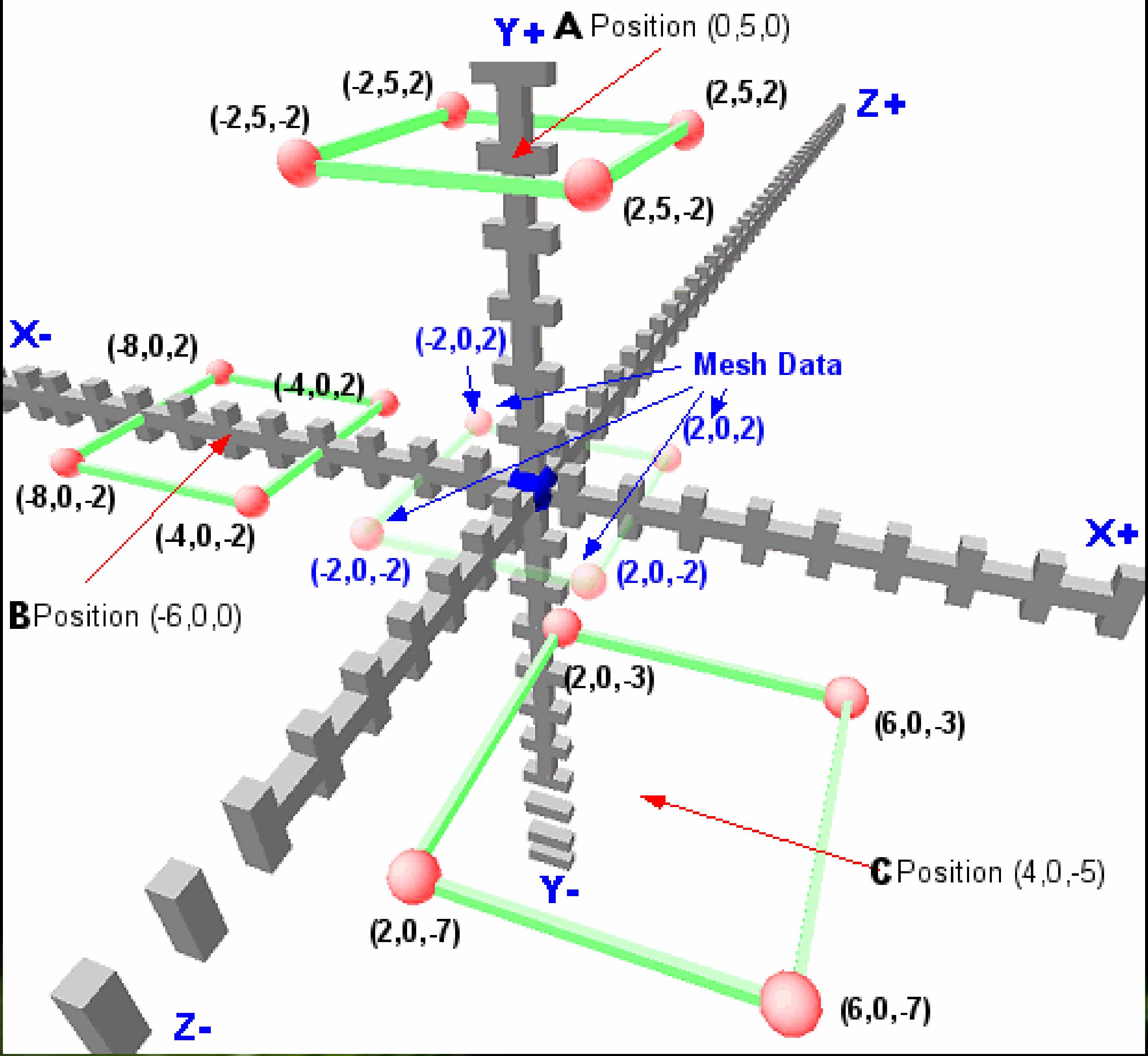
Question

- Why are we defining meshes in model space and then convert them into world space? Why not define them in the world space and remove one stage of the pipeline altogether?

Answer -> Instancing

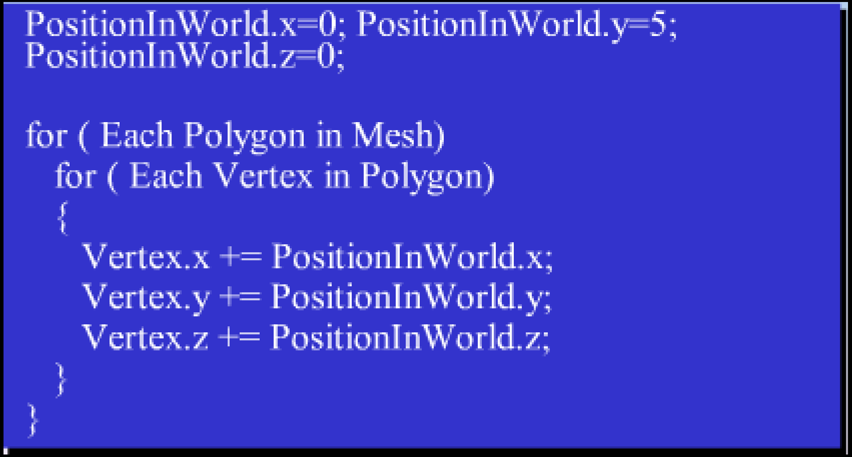
- The answer is simple! (well , no that simple ☺)
- Consider we have a road mesh and a lot of houses are around it. Let’s say all the houses were exactly the same. So we have one mesh placed multiple times in multiple locations.
- If we defined our meshes in world space, every time we wanted to draw another house mesh we had to reset all the vertices location (because we need their offset to be relative to their origin)
- **This process of decoupling mesh data from positional data is call “Instancing”**

Instances

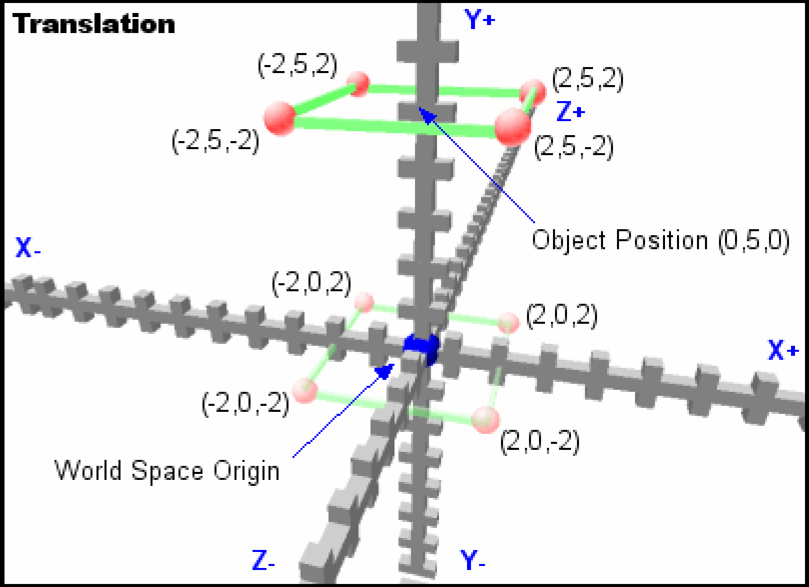

Y+
A Position (0,5,0)
(-2,5,2)
(2,5,2)
(-2,5,-2)
(2,5,-2)
Z+
X-
(-8,0,2)
(-4,0,2)
(-2,0,2)
Mesh Data
(2,0,2)
(-8,0,-2)
(-4,0,-2)
(-2,0,-2)
(2,0,-2)
X+
B Position (-6,0,0)
(2,0,-3)
(6,0,-3)
C Position (4,0,-5)
Y-
(2,0,-7)
(6,0,-7)
Z-

Transformations : Translation

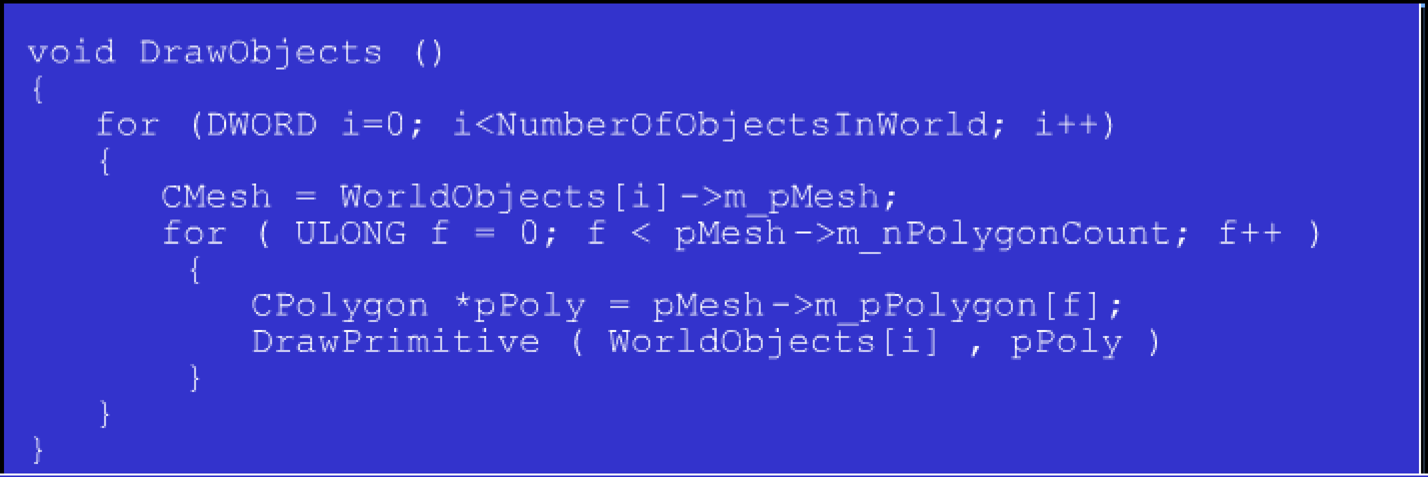
- Translating or moving vertices from one coordinate to another is called translation.
- Remember when you translate a mesh it will get translated from model space to world space.
- More on translation in math part!



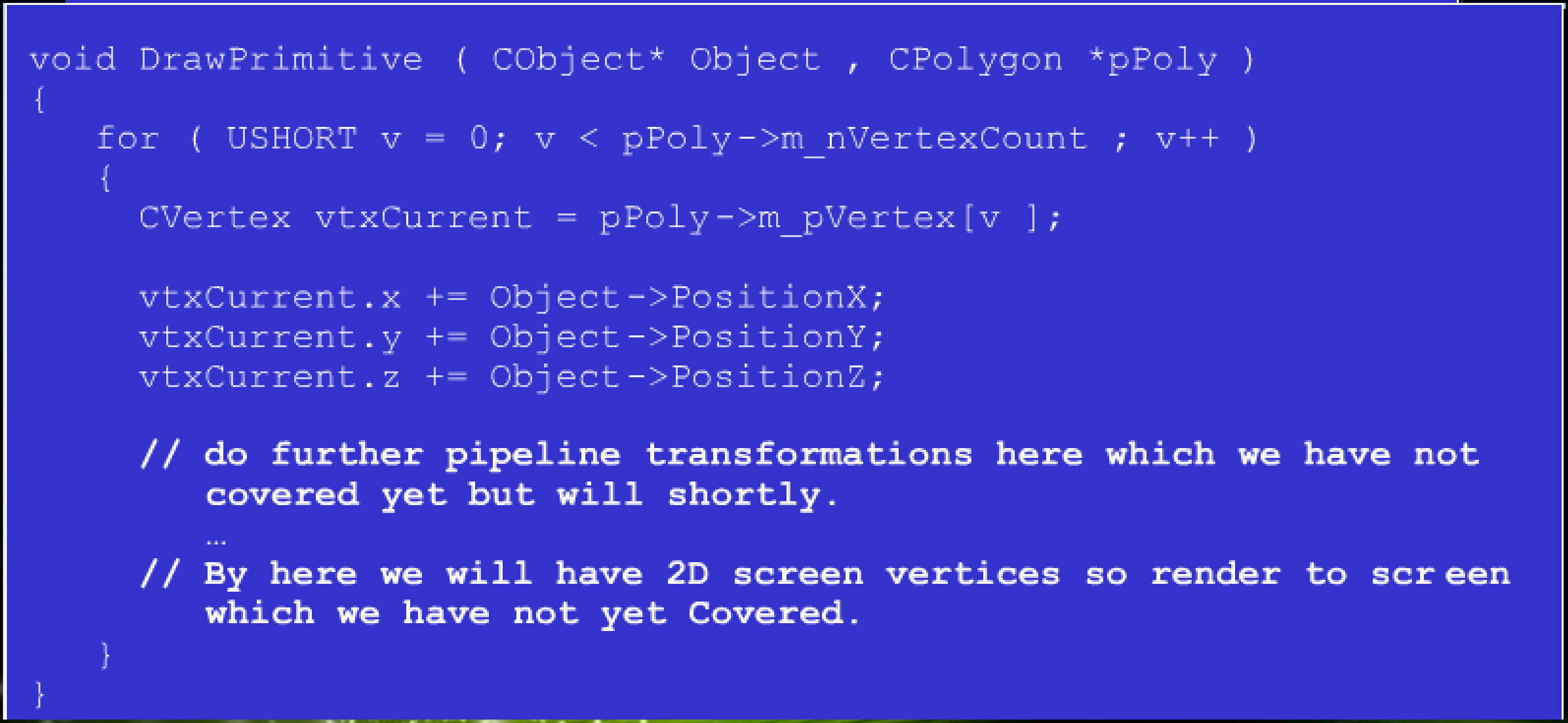
```
PositionInWorld.x=0; PositionInWorld.y=5;
PositionInWorld.z=0;

for ( Each Polygon in Mesh)
   for ( Each Vertex in Polygon)
   {
      Vertex.x += PositionInWorld.x;
      Vertex.y += PositionInWorld.y;
      Vertex.z += PositionInWorld.z;
   }
}
```



Transformation Pipeline So Far!



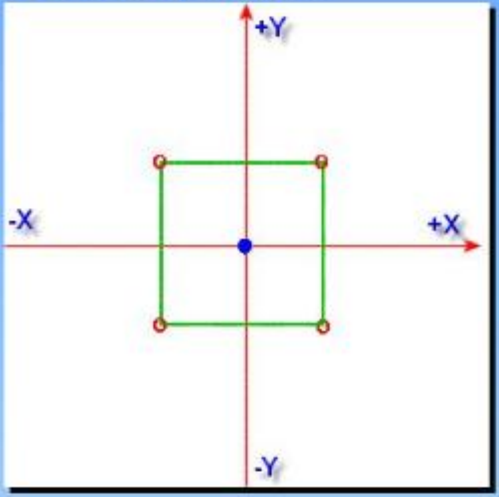
```
void DrawObjects ()
{
   for (DWORD i=0; i<NumberOfObjectsInWorld; i++)
   {
      CMesh = WorldObjects[i]->m_pMesh;
      for ( ULONG f = 0; f < pMesh->m_nPolygonCount; f++ )
       {
          CPolygon *pPoly = pMesh->m_pPolygon[f];
          DrawPrimitive ( WorldObjects[i] , pPoly )
       }
   }
}
```



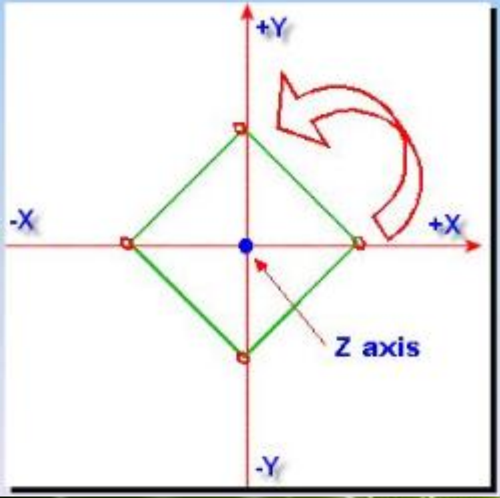
```
void DrawPrimitive ( CObject* Object , CPolygon *pPoly )
{
   for ( USHORT v = 0; v < pPoly->m_nVertexCount ; v++ )
   {
     CVertex vtxCurrent = pPoly->m_pVertex[v ];

     vtxCurrent.x += Object->PositionX;
     vtxCurrent.y += Object->PositionY;
     vtxCurrent.z += Object->PositionZ;

     // do further pipeline transformations here which we have not
        covered yet but will shortly.
        …
     // By here we will have 2D screen vertices so render to screen
        which we have not yet Covered.
   }
}
```

Transformations : Rotation

Before Rotation



After Rotation



X Axis Rotation

$$NewY = OldY \times \cos(\theta) - OldZ \times \sin(\theta)$$
$$NewZ = OldY \times \sin(\theta) + OldZ \times \cos(\theta)$$

Y Axis Rotation

$$NewX = OldX \times \cos(\theta) + OldZ \times \sin(\theta)$$
$$NewZ = OldX \times -\sin(\theta) + OldZ \times \cos(\theta)$$

Z Axis Rotation

$$NewX = OldX \times \cos(\theta) - OldY \times \sin(\theta)$$
$$NewY = OldX \times \sin(\theta) + OldY \times \cos(\theta)$$

$\theta =$ Angle to Rotate in Radians

```
class CObject
{
  public:
      CMesh *m_pMesh;

      float PositionX;
      float PositionY;
      float PositionZ;

      float RotationX;
      float RotationY;
      float RotationZ;
}
```

There are approximately 6.28 radians in a full circle. This means that 45 degrees is equivalent to pi/2 (1.1570796) radians and so on.

To convert degrees to radians we multiply the angle by PI divided by 180

```
DegToRad(x) ( x *( pi/18))
```

Important Notes About rotation

- When you rotate around an axis only the other two parameters should get calculated for vertices (for example if you rotate around x axis only the y and z parameters should get calculated)

- When a polygon (in 2D or 3D) is rotated it is always rotated around the center of coordinate system

Translation and rotation order

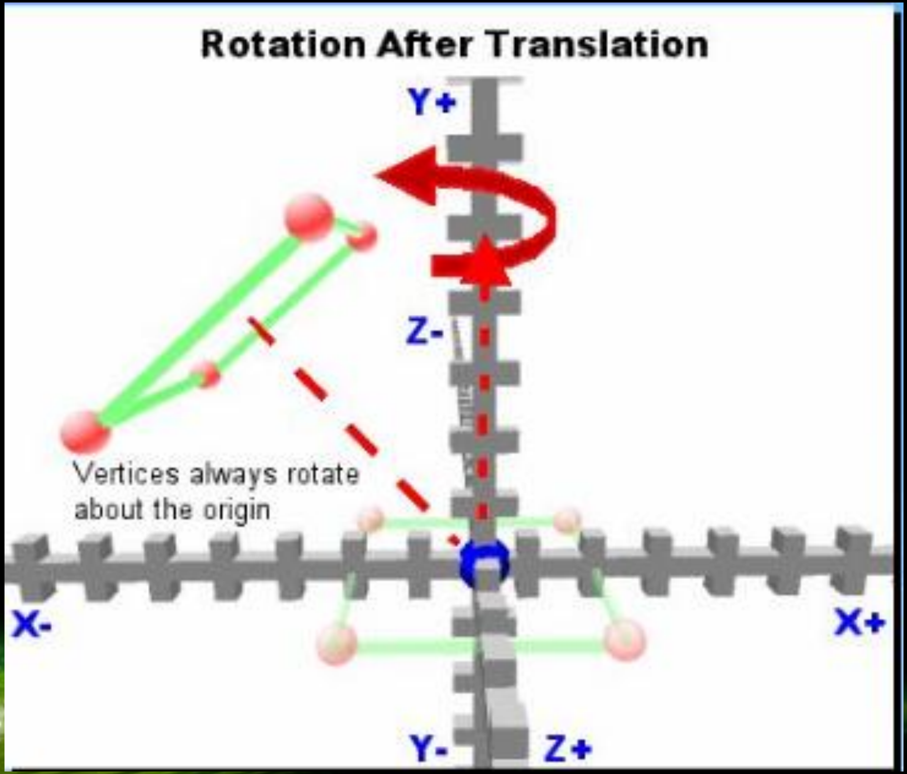
- Be careful! The order of doing the rotation and translation matters!

- Lets say we want to move a square 5 units up and rotate it 45 degree at the same time !

- So what can we do?

Translation and rotation order

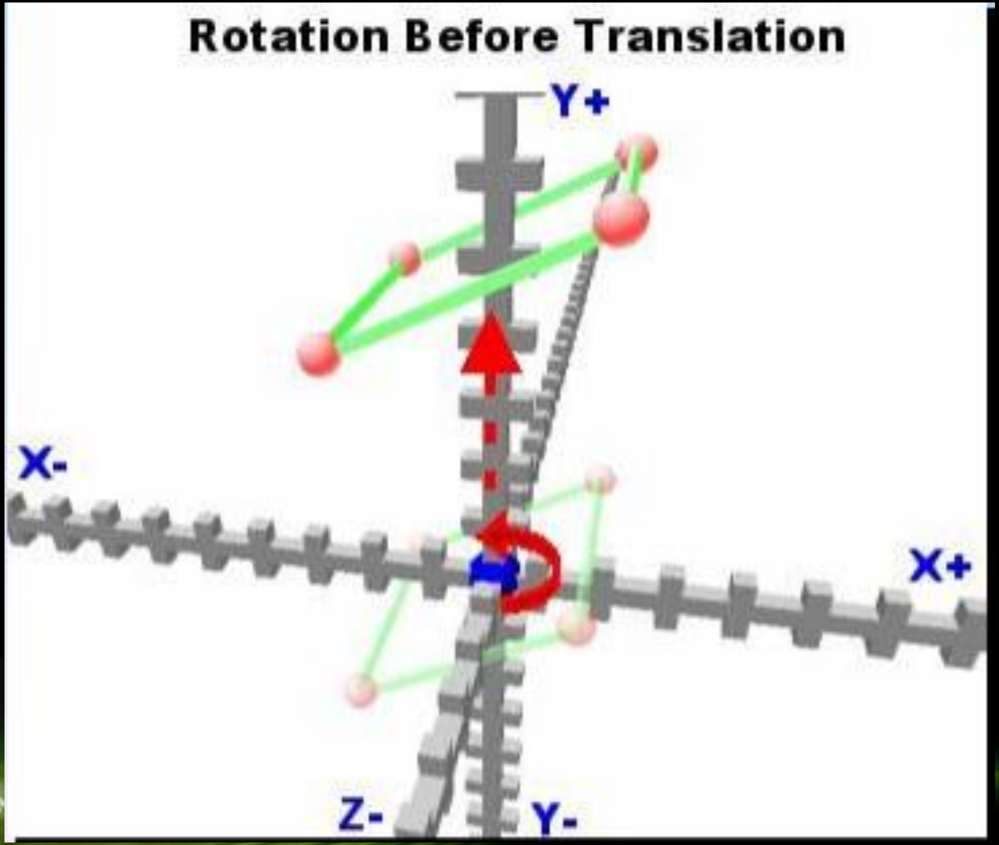
- If we rotate the vertices after we have translated them then the vertices is in the world space prior to the rotation.
- This means that the origin of the object is no longer at its center point but is the world space origin.
- Since the rotation always happens around the current center point the rotation will be aournd the world origin


Rotation After Translation
Y+
Z-
Vertices always rotate about the origin
X-
X+
Y-
Z+

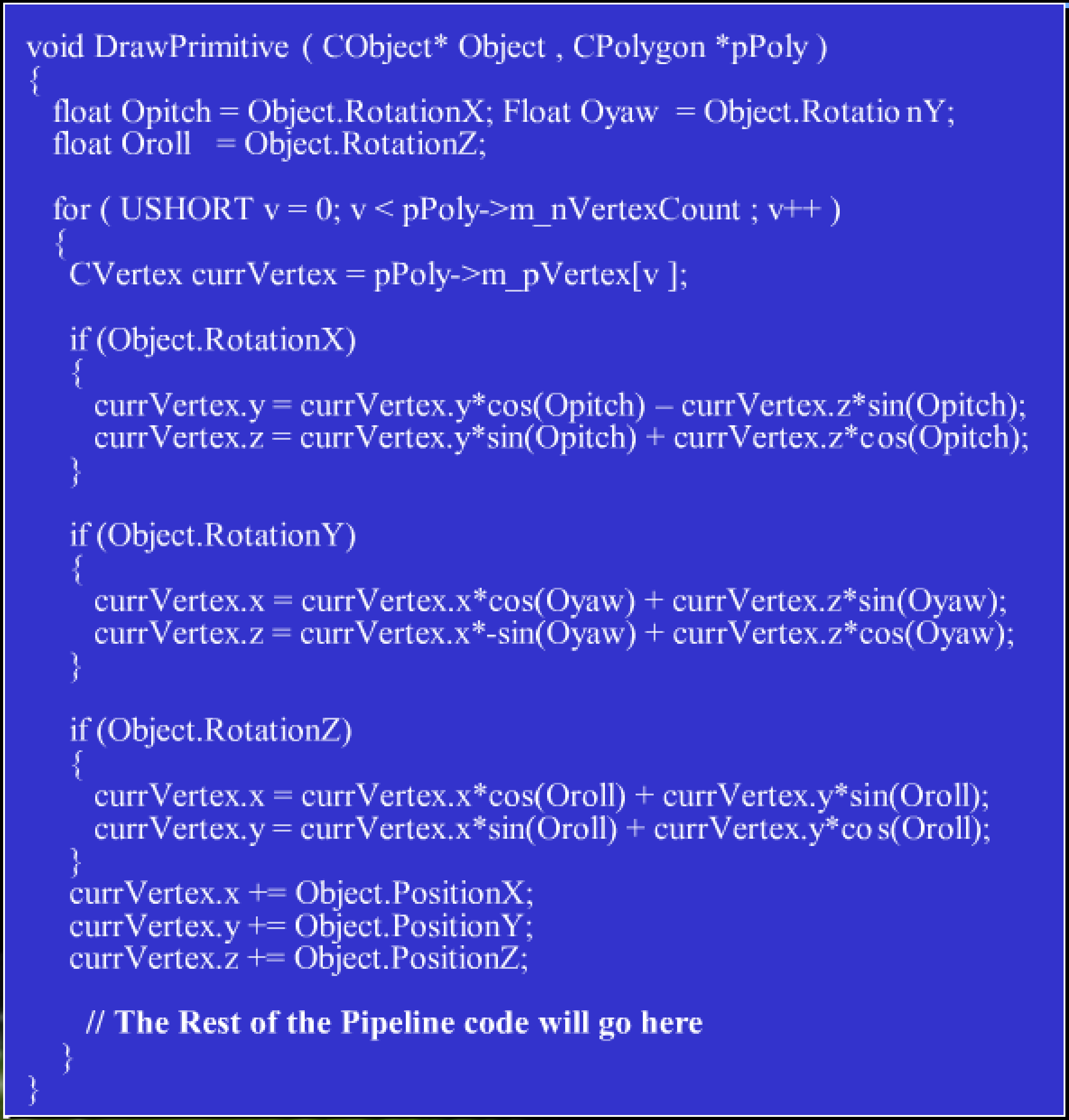
Translation and rotation order

- But if we rotate the vertices before translating them, we are performing the rotation in model space.

- Remember ! Most of the time you want to rotate any object around its own center point and not around the world center point!
- Of course exceptions exists ! Can you name one?



The DrawPrimitive So far!

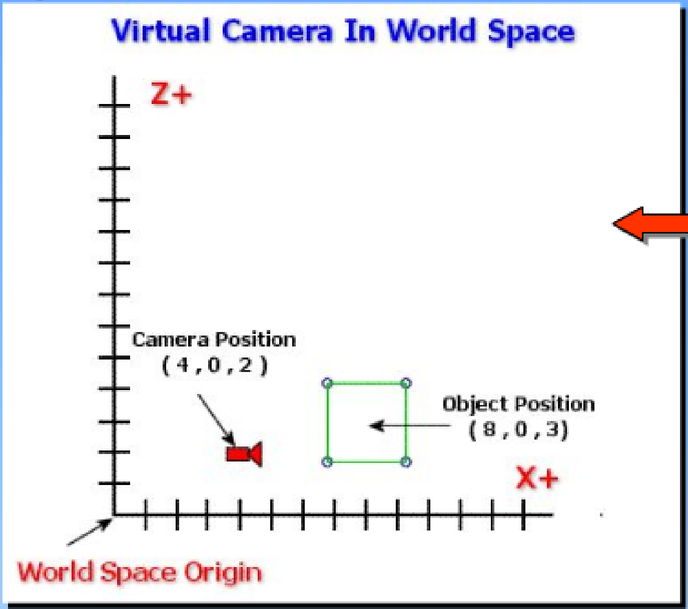


```
void DrawPrimitive ( CObject* Object , CPolygon *pPoly )
{
   float Opitch = Object.RotationX; Float Oyaw  = Object.Rotatio nY;
   float Oroll   = Object.RotationZ;

   for ( USHORT v = 0; v < pPoly->m_nVertexCount ; v++ )
   {
     CVertex currVertex = pPoly->m_pVertex[v ];

     if (Object.RotationX)
     {
        currVertex.y = currVertex.y*cos(Opitch) – currVertex.z*sin(Opitch);
        currVertex.z = currVertex.y*sin(Opitch) + currVertex.z*c os(Opitch);
     }

     if (Object.RotationY)
     {
        currVertex.x = currVertex.x*cos(Oyaw) + currVertex.z*sin(Oyaw);
        currVertex.z = currVertex.x*-sin(Oyaw) + currVertex.z*cos(Oyaw);
     }

     if (Object.RotationZ)
     {
        currVertex.x = currVertex.x*cos(Oroll) + currVertex.y*sin(Oroll);
        currVertex.y = currVertex.x*sin(Oroll) + currVertex.y*co s(Oroll);
     }
     currVertex.x += Object.PositionX;
     currVertex.y += Object.PositionY;
     currVertex.z += Object.PositionZ;

       // The Rest of the Pipeline code will go here
    }
}
```

View space transformation

- The idea is we have to be able to see different sections of the world by placing the camera and moving it around the world.
- But mathematically speaking no such entity as camera exist. Because all we have right now along our pipeline is a mathematical representation of our world stored as a collection of vertices.
- A 3D scene can always and only be viewed from the origin looking down the positive z axis.
- But how about FPS games where player can walk around the world and view at any angle he wants?
- The answer is : assuming that we have a hypothetical camera stuck at the origin which can not move! If we want to move the camera forward we have to move all the world object backward.
- If we want to rotate the camera to the right to see the right section of the world then we have to rotate everything in the world to the left.
- So we are making all the vertices camera relative !

Figure A

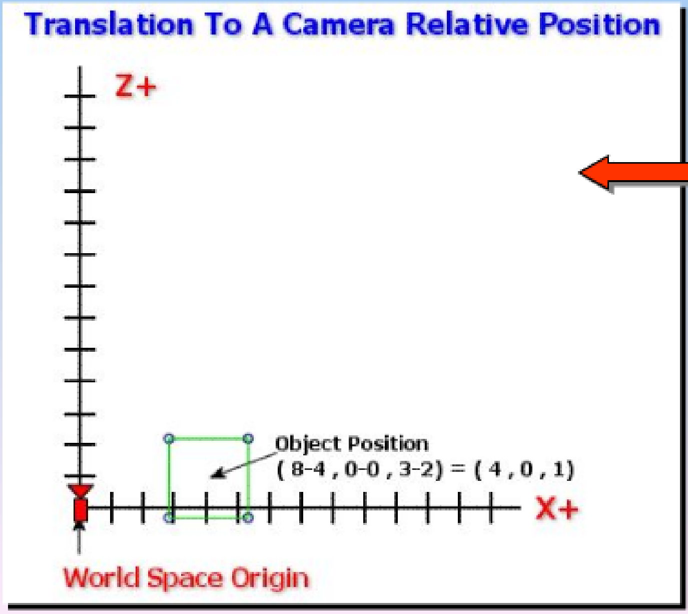


A camera doesn't actually exist, but we can store the world space position and orientation of an imaginary camera in our CCamera class.

```
class CCamera
{
 public:
    float PositionX;
    float PositionY;
    float PositionZ;

    float RotationX;  // Pitch
    float RotationY;  // Yaw
    float RotationZ;  //  Roll
};
```

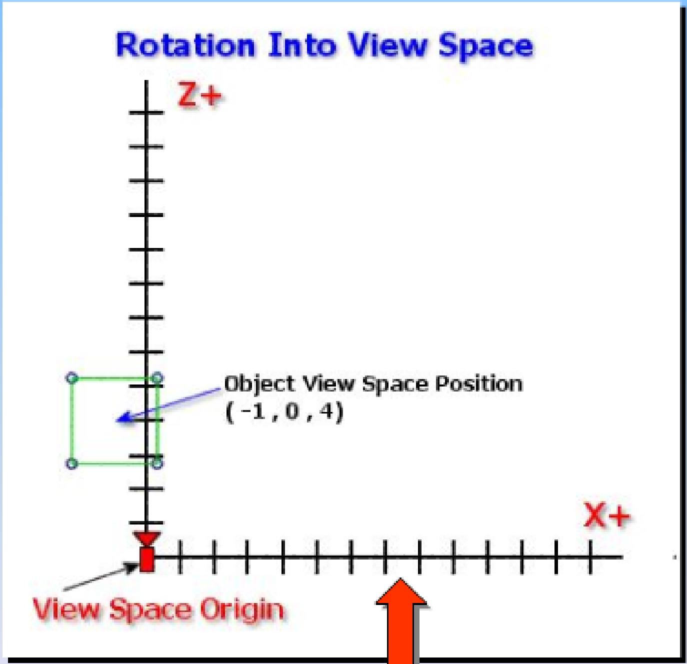
Figure A) shows an imaginary camera placed in the world at position (4,0,2) and rotated 90 degrees so that it is facing down the positive X axis.

Figure B



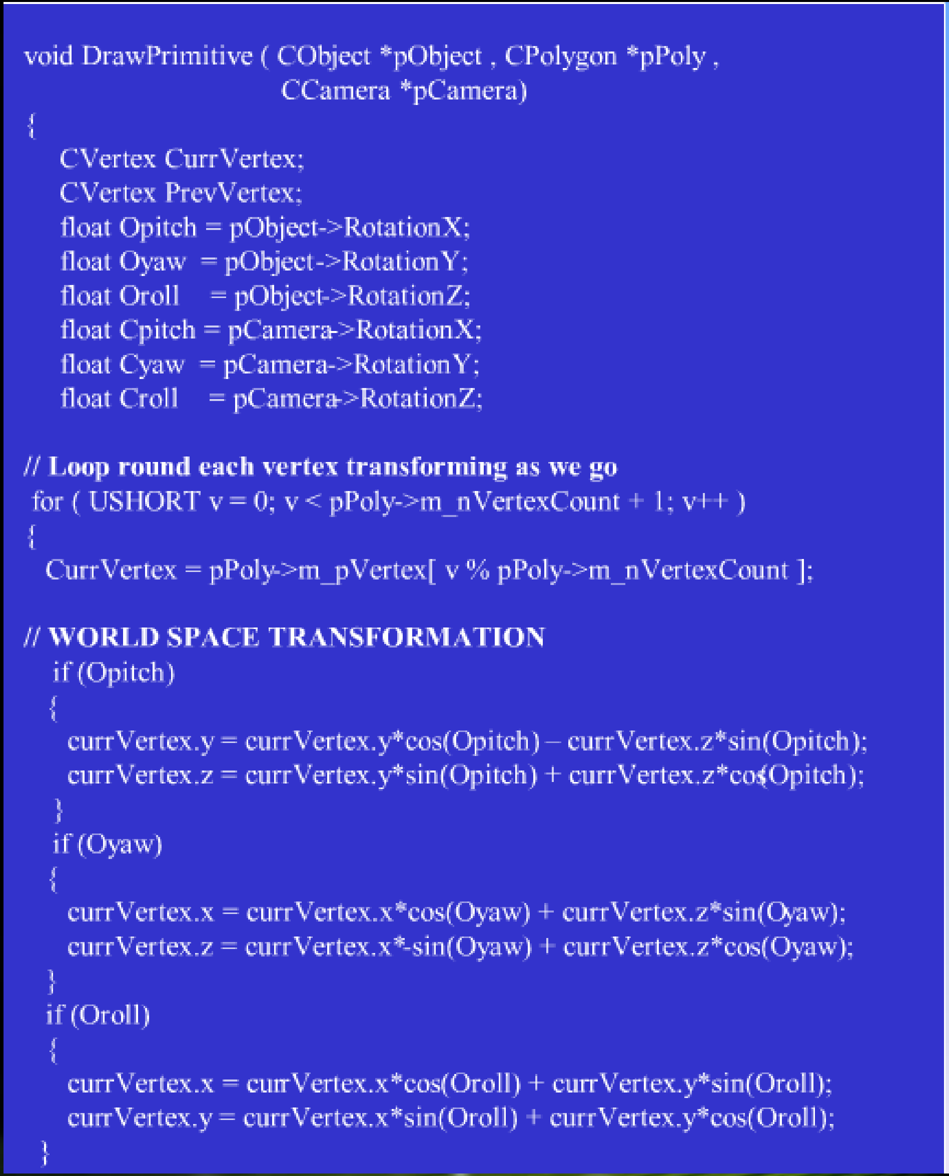
We first subtract the position of the camera from the position of each world space vertex such that the vertex is now specified relative to the camera at the origin. Each vertex is now a coordinate relative to the view space origin instead of the world space origin.

Figure C

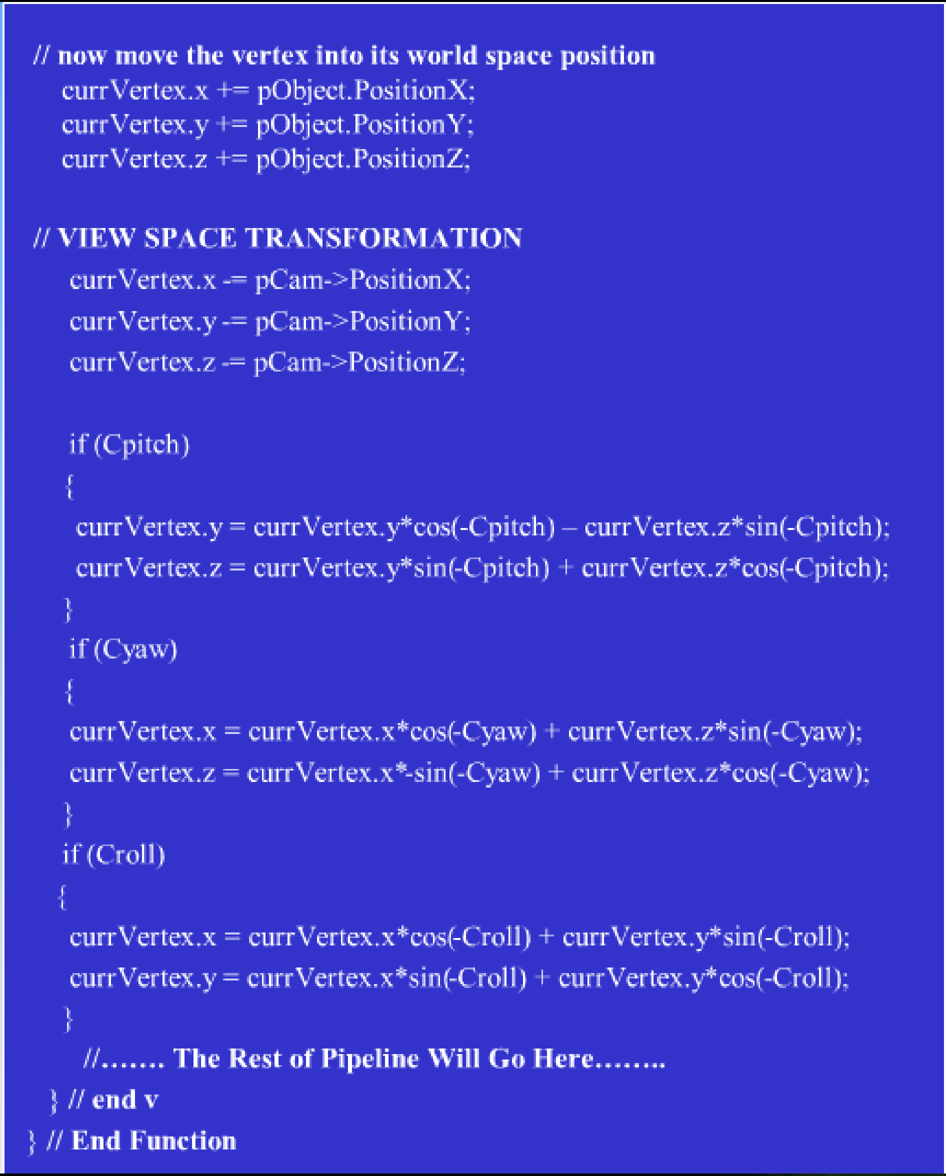


Finally, if the camera is rotated we must perform the inverse rotations to the vertices. In this example the camera was rotated right 90 degrees so we rotate the world space vertices left 90 degrees.

Our transformation code so far !



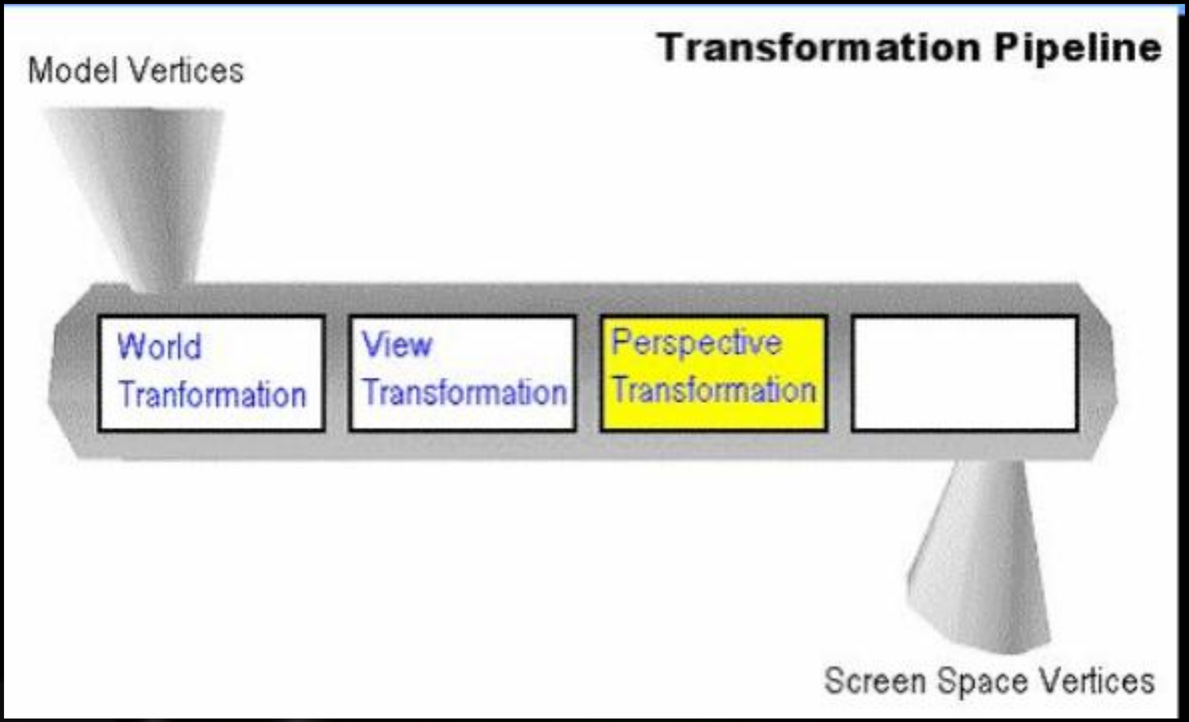
```
void DrawPrimitive ( CObject *pObject , CPolygon *pPoly ,
                              CCamera *pCamera)
{
    CVertex CurrVertex;
    CVertex PrevVertex;
    float Opitch = pObject->RotationX;
    float Oyaw   = pObject->RotationY;
    float Oroll    = pObject->RotationZ;
    float Cpitch = pCamera->RotationX;
    float Cyaw   = pCamera->RotationY;
    float Croll    = pCamera->RotationZ;

// Loop round each vertex transforming as we go
 for ( USHORT v = 0; v < pPoly->m_nVertexCount + 1; v++ )
{
  CurrVertex = pPoly->m_pVertex[ v % pPoly->m_nVertexCount ];

// WORLD SPACE TRANSFORMATION
   if (Opitch)
   {
     currVertex.y = currVertex.y*cos(Opitch) – currVertex.z*sin(Opitch);
     currVertex.z = currVertex.y*sin(Opitch) + currVertex.z*cos(Opitch);
   }
   if (Oyaw)
   {
     currVertex.x = currVertex.x*cos(Oyaw) + currVertex.z*sin(Oyaw);
     currVertex.z = currVertex.x*-sin(Oyaw) + currVertex.z*cos(Oyaw);
   }
  if (Oroll)
   {
     currVertex.x = currVertex.x*cos(Oroll) + currVertex.y*sin(Oroll);
     currVertex.y = currVertex.x*sin(Oroll) + currVertex.y*cos(Oroll);
   }
```



```
// now move the vertex into its world space position
    currVertex.x += pObject.PositionX;
    currVertex.y += pObject.PositionY;
    currVertex.z += pObject.PositionZ;

// VIEW SPACE TRANSFORMATION
     currVertex.x -= pCam->PositionX;
     currVertex.y -= pCam->PositionY;
     currVertex.z -= pCam->PositionZ;


     if (Cpitch)
     {
      currVertex.y = currVertex.y*cos(-Cpitch) – currVertex.z*sin(-Cpitch);
      currVertex.z = currVertex.y*sin(-Cpitch) + currVertex.z*cos(-Cpitch);
     }
     if (Cyaw)
     {
     currVertex.x = currVertex.x*cos(-Cyaw) + currVertex.z*sin(-Cyaw);
     currVertex.z = currVertex.x*-sin(-Cyaw) + currVertex.z*cos(-Cyaw);
     }
    if (Croll)
    {
     currVertex.x = currVertex.x*cos(-Croll) + currVertex.y*sin(-Croll);
     currVertex.y = currVertex.x*sin(-Croll) + currVertex.y*cos(-Croll);
     }
      //……. The Rest of Pipeline Will Go Here……..
  } // end v
} // End Function
```

Perspective Transformation

- This transformation is responsible for taking a 3D space coordinate and converting it into 2D coordinate taking the perspective into account.
- Objects that are farther away from the camera will appear smaller in screen space and vice versa

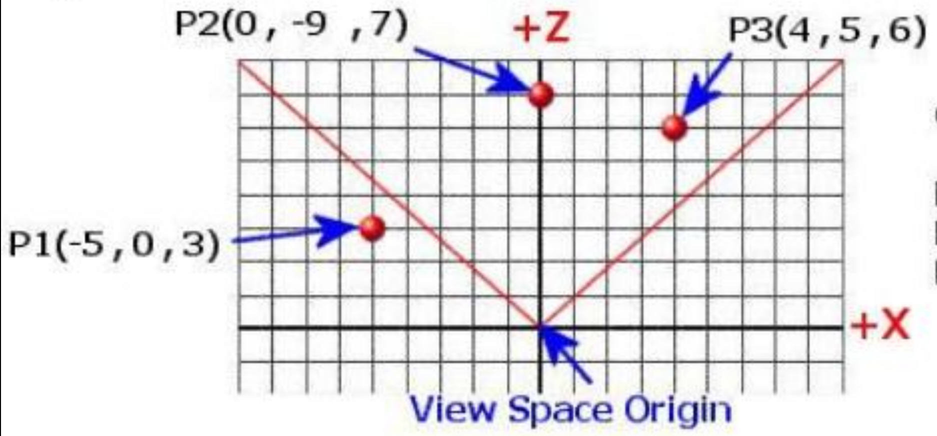


Perspective Transformation

- In real life when objects gets farther away from us they appear smaller to us
- That’s because Our view (Human eyes) is a cone !
- So the camera should have a view cone too !
- Every vertex that falls inside the imaginary view cone will get projected inside a -1 to +1 range of projection plane.
- With more distance (the larger the z component) the more world objects should fit in the projection plane and thus more squashing will take place and therefore :
    - The farther objects appear smaller

Figure A

The XZ View (Top Down View)

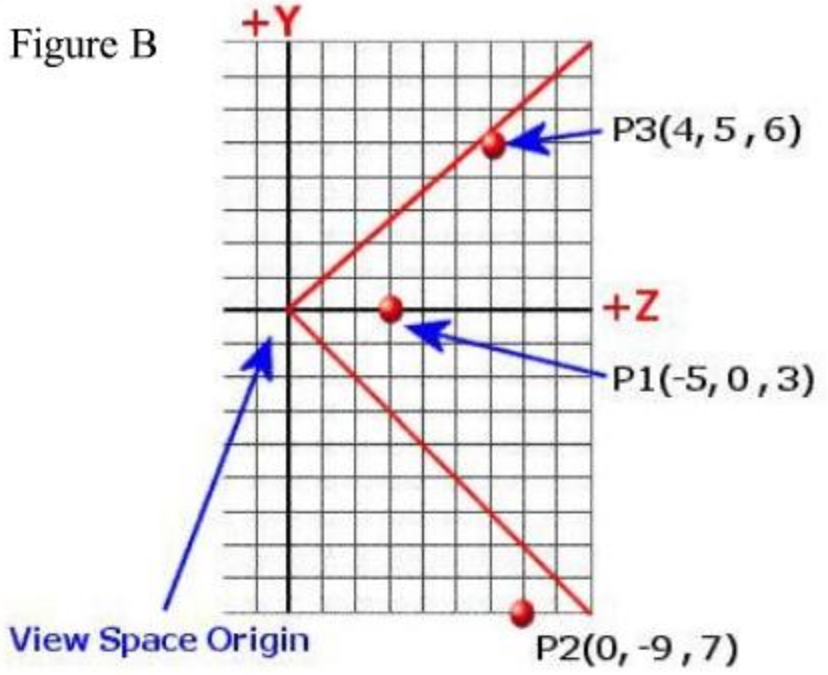


Calculating Projected X

ProjX1 = P1.x / P1.z = -5 / 3 = -1.666
ProjX2 = P2.x / P2.z = 0 / 7 = 0.0
ProjX3 = P3.x / P3.z = 4 / 6 = 0.666

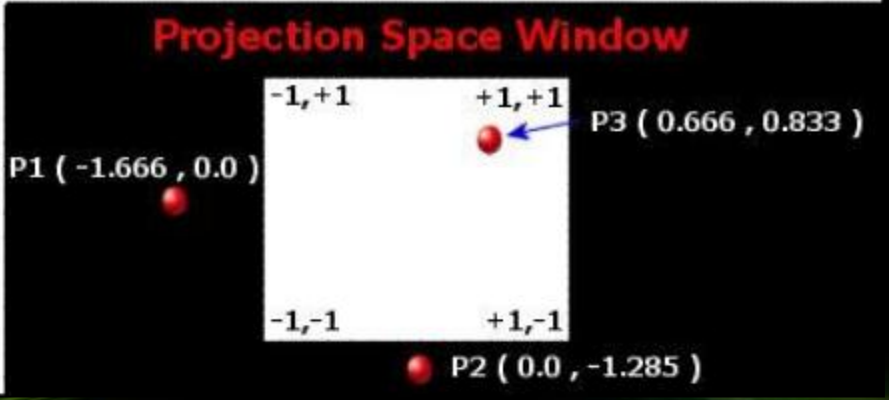
Figure B

The YZ View (Side View)

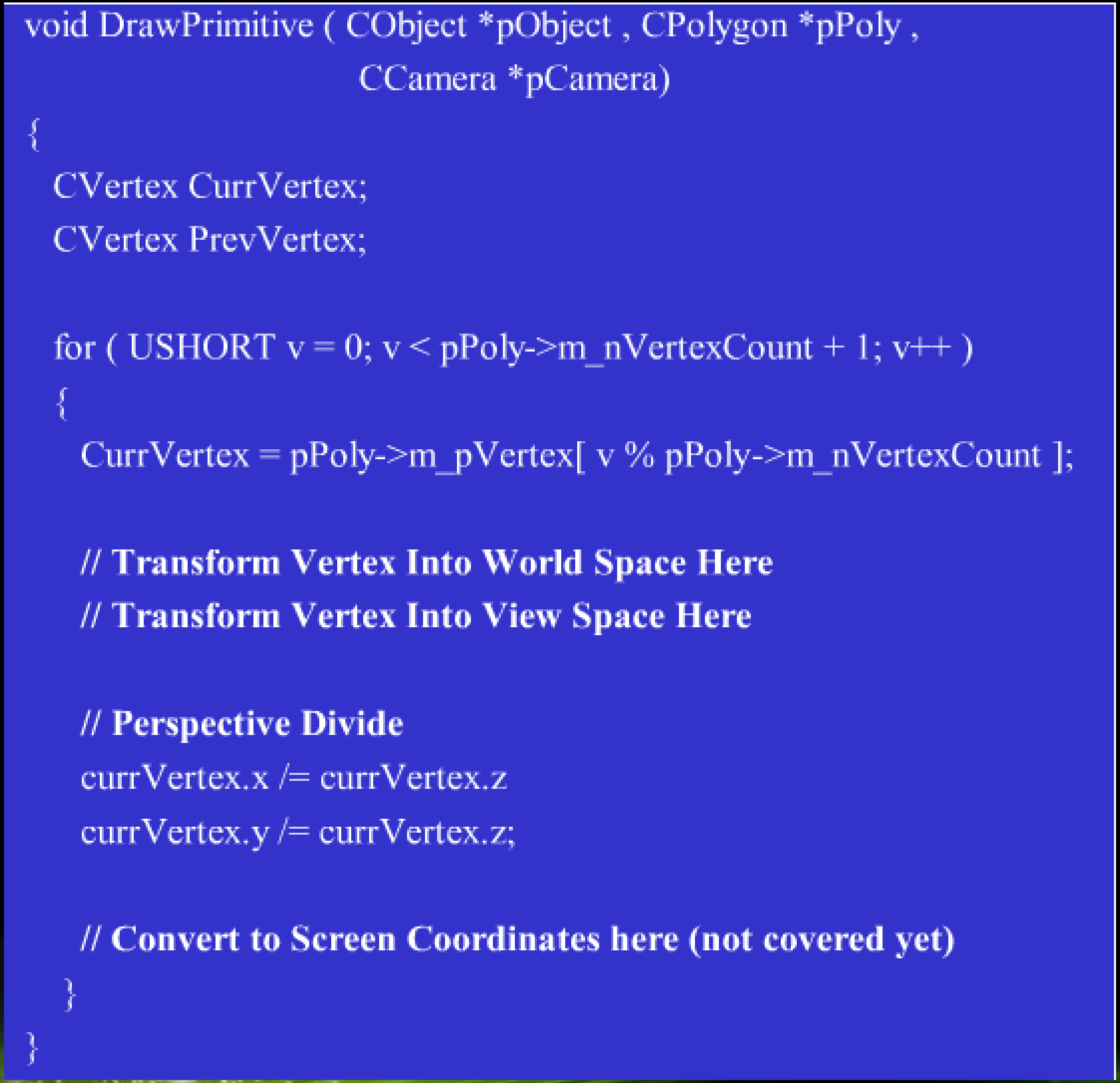


Calculating Projected Y

ProjY1 = P1.y / P1.z = 0 / 3 = 0.0
ProjY2 = P2.y / P2.z = -9 / 7 = -1.285
ProjY3 = P3.y / P3.z = 5 / 6 = 0.833



Our transformation code so far !



```
void DrawPrimitive ( CObject *pObject , CPolygon *pPoly ,
                                    CCamera *pCamera)
{
   CVertex CurrVertex;
   CVertex PrevVertex;

   for ( USHORT v = 0; v < pPoly->m_nVertexCount + 1; v++ )
   {
      CurrVertex = pPoly->m_pVertex[ v % pPoly->m_nVertexCount ];

      // Transform Vertex Into World Space Here
      // Transform Vertex Into View Space Here

      // Perspective Divide
      currVertex.x /= currVertex.z
      currVertex.y /= currVertex.z;

      // Convert to Screen Coordinates here (not covered yet)
    }
}
```

Screen Space Transformation

- All that is left is to transform the vertices into the screen space (or viewport space) coordinates.

- That’s because our rendering target may not be the whole monitor space but just a window or even a rectangular portion of the window

Screen Space Transformation Formula

$$\text{ScreenX} = \text{Vertex.x} * \frac{\text{ViewportWidth}}{2} + \text{ViewportLeft} + \frac{\text{ViewportWidth}}{2}$$

$$\text{ScreenY} = -\text{Vertex.y} * \frac{\text{ViewportHeight}}{2} + \text{ViewportTop} + \frac{\text{ViewportHeight}}{2}$$

Example)

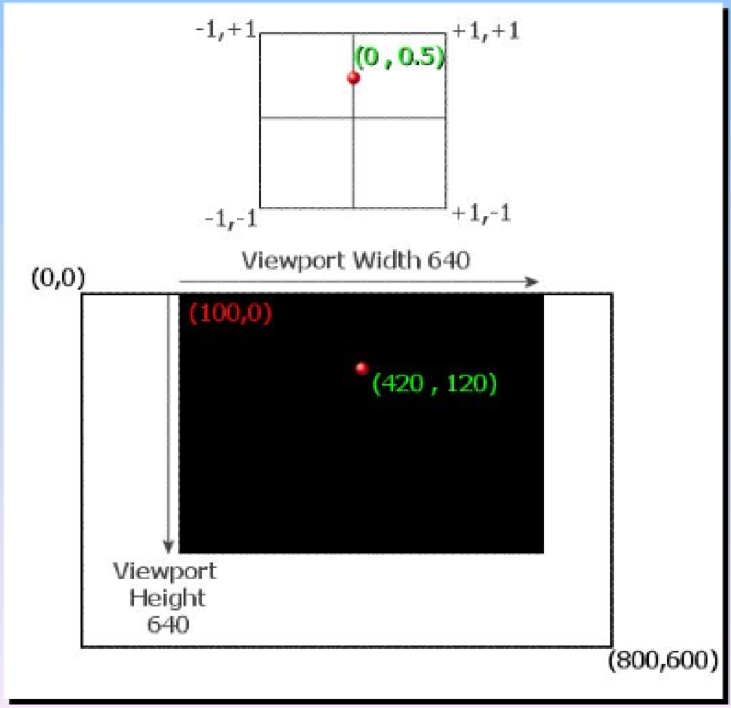
View Port (Left=100 , Top=0 , Width =640 , Height = 480)

Projection Space Vertex = (0.0 , 0.5)

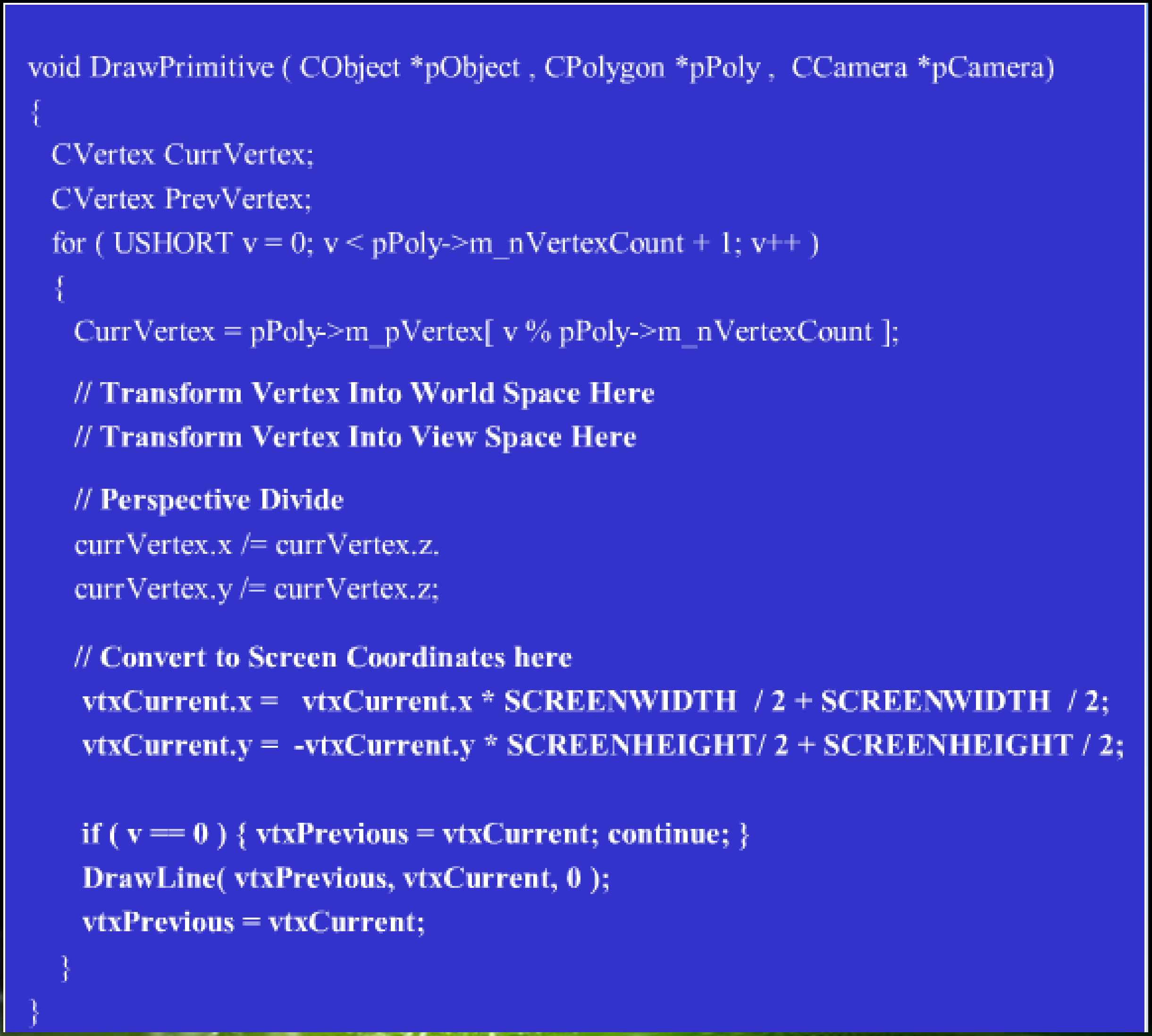
ScreenX = 0.0 * (640 / 2) + 100 + (640 / 2) = 420

ScreenY =- 0.5 * (480 / 2) + 0 + (480 / 2) = 120

Finally we have a screen space vertex ☺



Our Final Transformation Pipeline!



```
void DrawPrimitive ( CObject *pObject , CPolygon *pPoly ,  CCamera *pCamera)
{
  CVertex CurrVertex;
  CVertex PrevVertex;
  for ( USHORT v = 0; v < pPoly->m_nVertexCount + 1; v++ )
  {
    CurrVertex = pPoly->m_pVertex[ v % pPoly->m_nVertexCount ];

    // Transform Vertex Into World Space Here
    // Transform Vertex Into View Space Here

    // Perspective Divide
    currVertex.x /= currVertex.z.
    currVertex.y /= currVertex.z;

    // Convert to Screen Coordinates here
     vtxCurrent.x =   vtxCurrent.x * SCREENWIDTH  / 2 + SCREENWIDTH  / 2;
     vtxCurrent.y =  -vtxCurrent.y * SCREENHEIGHT/ 2 + SCREENHEIGHT / 2;

     if ( v == 0 ) { vtxPrevious = vtxCurrent; continue; }
     DrawLine( vtxPrevious, vtxCurrent, 0 );
     vtxPrevious = vtxCurrent;
  }
}
```

What do you think is wrong with the previous transformation pipeline?

It is extremely slow! So many cos() and sin () functions for each polygons.

This rendering loop will not work in a game ! ☺