

# CE318: Games Console Programming

## Lecture 4: 3D Models, Spaces and Cameras

Philipp Rohlfshagen

[prohlf@essex.ac.uk](mailto:prohlf@essex.ac.uk)

Office 2.529

# Outline

- 1 Intro to 3D: Revision
- 2 3D Models
- 3 More on Transformations
- 4 Concept of Cameras
- 5 Sample Progress Test
- 6 Summary

# Outline

- 1 Intro to 3D: Revision
- 2 3D Models
- 3 More on Transformations
- 4 Concept of Cameras
- 5 Sample Progress Test
- 6 Summary

In the last lecture, we created simple geometries including triangles, quads and (open) cubes. Using primitives is at the very heart of 3D XNA game programming so it is essential you have a good understanding of how it works.

Remember, we need the following to draw a geometry using triangles:

- A view to see the geometry
- A projection to map from 3D to 2D
- A world to position the geometry
- A set of vertices to construct the geometry
  - A primitive type to connect the vertices
  - Possibly indices and buffers
- Supply shader information to the graphics pipeline

To illustrate these aspects, we will now create a 3D cone. We can use standard settings for most of the requirements. The trickiest part is the construction of the cone itself.

The **view** determines how we see things in the 3D world:

```
1 Matrix view = Matrix.CreateLookAt(position, lookat, up)
```

The **projection** determines how the 3D scene is projected onto a 2D surface:

```
1 Matrix proj = Matrix.CreatePerspectiveFieldOfView(angle,
    GraphicsDevice.Viewport.AspectRatio, nearplane, farplane));
```

For the **world**, we simply use `Matrix.Identity`. Remember: each geometry has its own world matrix.

Finally, for the **shading information**, we use `BasicEffect`:

```
1 BasicEffect basicEffect;

1 basicEffect = new BasicEffect(GraphicsDevice);
2 basicEffect.World = world;
3 basicEffect.View = view;
4 basicEffect.Projection = proj;
5 basicEffect.VertexColorEnabled = true;
6
7 basicEffect.CurrentTechnique.Passes[0].Apply();
```

# Intro to 3D: Review

We create a new class for the cone. An abstract super class, `Shape`, and a concrete sub-class to draw the cone.

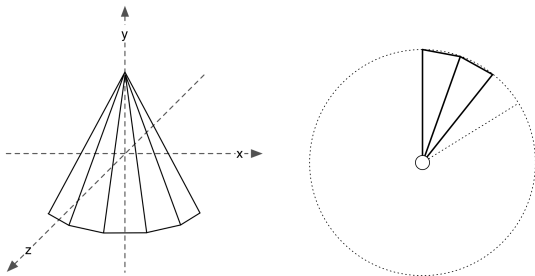
```
1  protected Matrix world = Matrix.Identity;
2
3  public void Scale(Vector3 scale)
4  {
5      world *= Matrix.CreateScale(scale);
6  }
7
8  public void Rotate(Vector3 rotate)
9  {
10     world *= Matrix.CreateFromYawPitchRoll(rotate.X, rotate.Y, rotate.Z);
11 }
12
13 public void Translate(Vector3 translate)
14 {
15     world *= Matrix.CreateTranslation(translate);
16 }
17
18 public abstract void Draw(Vector3 offset, Matrix view, Matrix proj);
19 public abstract void Draw(Matrix view, Matrix proj);
20 public abstract void Update();
21 public abstract void InitEffect(Matrix world, Matrix view, Matrix proj);
```

**Q** How to layout the vertices?

# Intro to 3D: Review

A cone is essentially a circle with a raised centre and the base, if required, is a flat circle with a centre. To draw the cone, we need:

- 1 We need to know how many triangles make up the cone (granularity)
- 2 Set the center vertex
- 3 Calculate the positions of all vertices on the circumference of the circle
- 4 Choose a primitive type. We will use `TriangleList` with indices.



**Q** If there are to be  $n$  triangles in the cone, how many vertices are required?

# Intro to 3D: Review

Assume `numTrianglesSide` specifies number of triangles (without base). Then, the number of vertices required (using indices) is:

```
1 int numVertices = 3 + (numTrianglesSide - 1);
```

We can also calculate the angle of the triangles at the top:

```
1 float angle = MathHelper.TwoPi / numTrianglesSide;
```

We then start calculating all required vertices:

```
1 VertexPositionColor[] vpc = new VertexPositionColor[numVertices];
2
3 //vertex at peak
4 vpc[0] = new VertexPositionColor(Vector3.Up, Color.Red);
5
6 int radius = 1;
7
8 //vertices around the circle
9 for (int i = 1; i < userPrimitives.Length; i++)
10     vpc[i] = new VertexPositionColor(new Vector3(
11         (float)(Math.Cos(i * angle) * radius),
12         -1,
13         (float)(Math.Sin(i * angle) * radius)),
14         colours[i%colours.Length]);
```

We use a circle of unit length: we can scale it later using transformations.

Next, we need to set up the vertices to specify the triangles:

```
1 short[] indices = new short[3 * totalTriangles];
2
3 int count = 0;
4
5 for (int i = 0; i < indices.Length; i += 3)
6 {
7     indices[i] = 0;
8     indices[i + 1] = (short)(count + 1);
9     indices[i + 2] = (short)(count + 2);
10    count++;
11 }
```

The first vertex of every triangle is the center one. The other two are chosen sequentially to lie along the circumference of the circle.

Note: the vertices are laid out clock-wise and added in the same manner; hence no culling.

And that's it. We only need to draw it:

```
1 device.DrawUserIndexedPrimitives<VertexPositionColor>(PrimitiveType.
    TriangleList, vpc, 0, vpc.Length, indices, 0, numberOfTriangles);
```

The last thing to add is a base.

**Q** How many additional vertices are required?

**Q** How many more triangles do we draw?

Instantiate your vertex array to the appropriate size, then:

```
1 vpc[vpc.Length-1] = new VertexPositionColor(Vector3.Down, Color.Red);
2
3 count = 0;
4
5 for (int i = 0; i < indices.Length / 2; i += 3)
6 {
7     indices[(indices.Length / 2) + i] = (short)(vpc.Length - 1);
8     indices[(indices.Length / 2) + i + 1] = (short)(count + 2);
9     indices[(indices.Length / 2) + i + 2] = (short)(count + 1);
10    count++;
11 }
```

Tip: it is often a bit tricky to get the drawing order right. Try to draw the geometry with culling turned off. Once it displays correctly, turn culling back on and the fix drawing order if required.

Finally, some pointers based on last week's lab:

- Each geometry has its own world matrix
- Use a camera for the view and perspective
  - Make sure to assign the view and projection to the object's effect
- Can use `GameComponent` for camera and geometries
- It is often easiest to declare a new `BasicEffect` for each shape drawn:
  - Do not need to revert previous settings (e.g., lighting, textures)
  - Can easily save settings for future viewing
- Make sure that assets are
  - Loaded where required (e.g., in a `DrawableGameComponent`)
  - Passed from `Game1` to the appropriate class
- Changing the cull mode requires a new `RasterizerState` object

Have a look at the example code to be made available on the course website shortly. This code includes the cone, a closed cube and a model cube as well as a free camera implementation.

# Outline

- 1 Intro to 3D: Revision
- 2 3D Models**
- 3 More on Transformations
- 4 Concept of Cameras
- 5 Sample Progress Test
- 6 Summary

It is possible to create simple constructs using primitives. However, even smaller objects may require large amounts of code, most of which needs to be hand-coded (symmetries etc. may be exploited but asymmetric deviations require additional data).

It is thus common to create complex models in specialised software packages and to import these directly into XNA. These models may vary greatly in complexity from spheres and boxes to entire cities. We will also look at some 3D authoring tools that are free to download.



# What are Models?

Models are really just geometries (or collections of geometries) like the cone we drew earlier. However, most models tend to be too complex to specify using primitives and often contain additional aspects such as colours, textures or even information regarding animations.

Models in XNA are stored as type `Model` and have the following composition:

- `Meshes`: a collection of `ModelMesh` objects, each of which has:
  - `MeshParts`: collection of `ModelMeshPart` objects with:
    - `Effect`, `IndexBuffer`, `VertexBuffer`, `NumVertices`, `PrimitiveCount`
  - `BoundingSphere`: bounding sphere for the mesh
  - `Effects`: collection of effects (one for each mesh part)
  - A `Draw()` method (to draw all mesh parts)
  - `ParentBone`: contains transformation matrix with relative position
- `Bones`: a collection of `ModelBone` objects which have
  - `Parent`, `Children`, `Transform`
- `Root`: the root bone

`Model` also has some methods to efficiently obtain the transformations (bones).

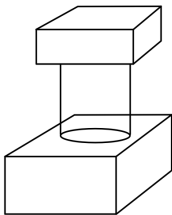
# Meshes, Mesh Parts & Bones

A `Model` consists of one or more `ModelMesh` objects. The cube we saw in the last lecture contains only one `ModelMesh`. Each `ModelMesh` object of a `Model` may be moved independently and may be composed of multiple **materials**. A material is a combination of `Effect` and texture and is stored as a `ModelMeshPart` object.

Note: the effects stored in each `ModelMeshPart` may be accessed directly via the collection of effects stored in the `ModelMesh`. The `Draw()` method of `ModelMesh` does not take any arguments: it uses the appropriate effect from the `ModelMeshPart` currently drawn.

**Bones** connect the individual meshes of the model. In particular, each bone contains a transformation that places itself relative to its parent (and thus implicitly to its children). The `Root` is the root bone without a parent. All other bones are positioned relative to this.

# Meshes, Mesh Parts & Bones



```
top = Matrix.CreateRotationX(0.3f) *  
      Matrix.CreateTranslation(new Vector3(0, 3, 0))
```

```
center = Matrix.CreateTranslation(new Vector3(0, 3, 0))
```

```
bottom = Matrix.Identity
```

Note: all transformations are **relative** to one another.

We can use 3 methods in `Model` to access all transforms:

- `CopyBoneTransformsTo()`, `CopyBoneTransformsFrom()`, `CopyAbsoluteBoneTransformsTo()`

```
1 Matrix[] transforms = new Matrix[model.Bones.Count];  
2 model.CopyAbsoluteBoneTransformsTo(transforms);
```

The first method retrieves a copy of the model's transforms while the second method sets the transforms. The third method retrieves all transforms but combines each transform with that of its parent (i.e., no longer relative).

# Drawing Models

At the end of the last lecture, we drew a simple model (a box). We needed to add the model to the content project, then load and draw it.

```
1 Model model;
```

In `Game1.LoadContent()`, load the model and set default lighting:

```
1 model = Content.Load<Model>("myModel");  
2 (model.Meshes[0].Effects[0] as BasicEffect).EnableDefaultLighting();
```

In `Game1.Draw()`:

```
1 model.Draw(world, view, proj);
```

We can make 3 observations:

- We have no control over the individual meshes
- We set the default light only for the first mesh
- We did not need to apply the effect

It is common to write your own draw method to have a variable degree of flexibility with regards to the individual components of the model. We will do this next and thereby highlighting the different aspects of `Model`.

# Drawing Models

The first draw method goes through all meshes, sets the individual effects for the mesh parts and then calls the method `ModelMesh.Draw()`.

```
1 public void DrawModel(Matrix view, Matrix proj)
2 {
3     Matrix[] transforms = new Matrix[model.Bones.Count];
4     model.CopyAbsoluteBoneTransformsTo(transforms);
5
6     foreach (ModelMesh mesh in model.Meshes)
7     {
8         foreach (BasicEffect effect in mesh.Effects)
9         {
10            effect.EnableDefaultLighting();
11            effect.Projection = proj;
12            effect.View = view;
13            effect.World = world * mesh.ParentBone.Transform;
14        }
15
16        mesh.Draw();
17    }
18 }
```

`ModelMesh.Draw()` goes through all model mesh parts and draws them according to the specification of their effect. Note: no need to apply effect explicitly.

# Drawing Models

Very similar to the previous method. But now we go through all model mesh parts directly and do not access the effects via the array in the mesh.

```
1 public void DrawModel(Matrix view, Matrix proj)
2 {
3     Matrix[] transforms = new Matrix[model.Bones.Count];
4     model.CopyAbsoluteBoneTransformsTo(transforms);
5
6     foreach (ModelMesh mesh in model.Meshes)
7     {
8         foreach (ModelMeshPart mmp in mesh.MeshParts)
9         {
10            BasicEffect effect = mmp.Effect as BasicEffect;
11            effect.EnableDefaultLighting();
12            effect.Projection = proj;
13            effect.View = view;
14            effect.World = world * mesh.ParentBone.Transform;
15        }
16
17        mesh.Draw();
18    }
19 }
```

Like before, we need to apply the parent bone transform to get the absolute position of the mesh by its relative transformation.

# Drawing Models

```
1 public void DrawModel(Matrix view, Matrix proj)
2 {
3     Matrix[] transforms = new Matrix[model.Bones.Count];
4     model.CopyAbsoluteBoneTransformsTo(transforms);
5
6     foreach (ModelMesh mesh in model.Meshes)
7     {
8         foreach (ModelMeshPart mmp in mesh.MeshParts)
9         {
10            BasicEffect effect = mmp.Effect as BasicEffect;
11            effect.EnableDefaultLighting();
12            effect.Projection = proj;
13            effect.View = view;
14            effect.World = world * mesh.ParentBone.Transform;
15
16            effect.GraphicsDevice.SetVertexBuffer(mmp.VertexBuffer, mmp.
17                VertexOffset);
18            effect.GraphicsDevice.Indices = mmp.IndexBuffer;
19
20            foreach (EffectPass ep in effect.CurrentTechnique.Passes)
21            {
22                ep.Apply();
23                device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
24                    mmp.NumVertices, mmp.StartIndex, mmp.PrimitiveCount);
25            }
26        }
27    }
28 }
```

Note: same approach we used for the primitives. Also, need to apply effect.

Note: BasicEffect only has one pass so third loop not really required.

# Drawing Models

The last draw method nicely illustrates the relationship between the primitive types you drew last week (and the cone from today) and models. In the past, we did 2 things when dealing with primitives and models.

In the case of the primitives, we initialised our effect as follows:

```
1 effect.World = world;
2 effect.View = view;
3 effect.Projection = proj;
4 effect.VertexColorEnabled = true;
5 effect.CurrentTechnique.Passes[0].Apply();
```

We need to apply the effect ourselves. As `BasicEffect` only has a single pass, we can access the pass directly. We will cover these aspects in more depth in lecture 6.

In the case of models, we did the following:

```
1 (model.Meshes[0].Effects[0] as BasicEffect).EnableDefaultLighting();
```

This prevents the model from looking flat.

**Q** What if there are more than one mesh in the model?

Solution: loop through effects or use extended draw method.

# DCC Software Packages

A game's assets (including models) are created using a variety of digital content creation (DCC) packages. The XNA content pipeline makes it easy and efficient to integrate externally created artwork into XNA games.

We will look at the following (free) DCC tools (installed on lab machines):

- Paint.NET - for textures and images
- Blender - for 3D models and animated models
- Google SketchUp - for 3D models (warehouse)
- MeshLab - analyse and correct 3D meshes

Many applications offer a wide array of functionality but you will find specialised packages to deal with: 3D models, animations, physics modelling, drawing/painting, terrain generation, level design and shading.

We will be using Google SketchUp mostly.

**Paint.NET** is a popular Gimp clone for Windows that allows one to create layered graphics and textures. It is available from

[www.getpaint.net](http://www.getpaint.net)

**Blender** allows for the creation and animation of complex models:

[www.blender.org](http://www.blender.org)

Due to its complexity, Blender is not straightforward to use but there are many tutorials online.

**MeshLab** is a tool for the processing and editing of 3D meshes:

[meshlab.sourceforge.net](http://meshlab.sourceforge.net)

It is particularly useful for examine complex meshes and to perform minor operations on these to fix errors.

# Google SketchUp (8.0)

Google SketchUp is probably one of the simplest 3D packages out there.

`sketchup.google.com`

SketchUp comes with a 3D Warehouse, an exchange of free 3D models that may be imported. The Warehouse has an impressive collection of models, ranging from objects like chairs to football stadia.

Free version only allows exports into

- COLLADA (.dae)
- Google Earth (.kmz)

Use the plugin XEXPORTER to export to .x:

`http://edecadoudal.googlepages.com/xExporter.rb`

This does not seem to work well for complex models. Instead, export into .dae, import into Blender and export from there to .fbx.

# Calculating Bounding Spheres

Similar to collision detection in 2D using bounding rectangles, we can use bounding spheres to get a rough idea as to which portions of a 3D world a particular model occupies.

```
1 BoundingBox boundingSphere;
```

Create an all-encapsulating bounding sphere from all the model's meshes:

```
1 private void buildBoundingBox()
2 {
3     Matrix[] transforms = new Matrix[model.Bones.Count];
4     model.CopyAbsoluteBoneTransformsTo(transforms);
5
6     boundingSphere = new BoundingBox(Vector3.Zero, 0);
7
8     foreach (ModelMesh mesh in model.Meshes)
9     {
10        BoundingBox transformed = mesh.BoundingBox.Transform(
11            transforms[mesh.ParentBone.Index]);
12        boundingSphere = BoundingBox.CreateMerged(boundingBox,
13            transformed);
14    }
15 }
```

# Calculating Bounding Spheres

It is necessary to scale and translate the bounding sphere in accordance to the model to match the two. We can use properties for this:

```
1 public BoundingBox BoundingBox
2 {
3     get
4     {
5         Matrix worldTransform = Matrix.CreateScale(scale) * Matrix.
            CreateTranslation(position);
6         BoundingBox transformed = boundingSphere;
7         transformed = transformed.Transform(worldTransform);
8
9         return transformed;
10    }
11 }
```

**Q** Why do we not apply rotations as well?

Once we have the `BoundingBox`, we can use it in a similar fashion to the bounding rectangles to detect collisions etc. We can use

- `BoundingBox.Contains()`
- `BoundingBox.Intersects()`

# Scaling Models

As content design is spread across many people, it is vital (during design) to establish a unit of measurement. In some cases, models may be too large/small to fit into the game world created. In that case, they need to be scaled:

```
1 private float getScale(Model m)
2 {
3     float radius = 0.0f;
4
5     foreach (ModelMesh mm in m.Meshes)
6     {
7         if (mm.BoundingBox.Radius > radius)
8         {
9             radius = mm.BoundingBox.Radius;
10        }
11    }
12
13    return 1.0f / radius;
14 }
```

```
1 private void DrawModel(Model m, float radius, Matrix prj, Matrix view)
2 {
3     m.Draw(Matrix.CreateScale(getScale(m)), view, prj);
4 }
```

Note: it is also possible to set the scale in VS2010, under *Properties* of the model.

# Outline

- 1 Intro to 3D: Revision
- 2 3D Models
- 3 More on Transformations**
- 4 Concept of Cameras
- 5 Sample Progress Test
- 6 Summary

# Transformations

In the simplest case, we have a stationary object at the origin; we can use `Matrix.Identity`. We have already covered how to transform the object's world using rotations, translations and scale.

$$\mathbf{world} = \mathbf{scale} \times \mathbf{rotation} \times \mathbf{translation}$$

We now look in more depth at how to move an object through 3D space. The most important concepts include:

- The distinction between global and local coordinate systems
- The relationship between the local axes
- How one transformation affects multiple axes simultaneously

For this, we need to know a little bit more about vectors and matrices.

Note: you will notice that there are many different approaches in the literature to implement rotations and translations. Many of them are equally valid. It is thus important to understand how these transforms work to choose the best implementation for your game.

# Operations on Vectors

## Addition:

$$(1, 3) + (2, 2) = (3, 5)$$

$$a + b \equiv b + a$$

## Scalar multiplication:

$$x(2, 3) = (2, 3)x = (2x, 3x)$$

$$2(2, 3) = (4, 6)$$

## Dot product:

$$a \cdot b = b \cdot a = \sum_{i=1}^n a_i b_i$$

$$a \cdot b = |a||b| \cos(\theta)$$

## Subtraction:

$$(4, 3) - (3, 1) = (1, 2)$$

$$a - b \neq b - a$$

## Negation:

$$a = (3, 2), -a = (-3, -2)$$

$$b = (2, -2), -b = (-2, 2)$$

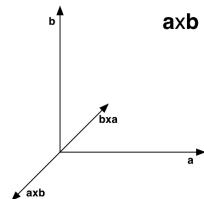
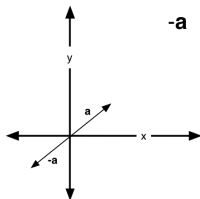
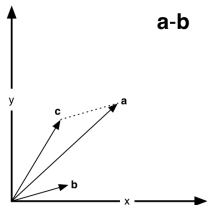
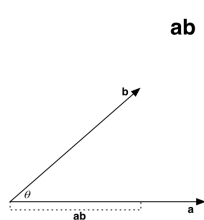
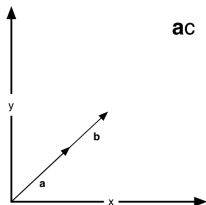
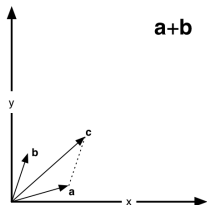
## Cross product:

$$a \times b = |a||b| \sin(\theta)n$$

$$a \times b \equiv -(b \times a)$$

$n$  is the unit vector perpendicular to the plane containing  $a$  and  $b$ .

# Operations on Vectors



Notes: vectors always emanate from origin.

# Matrices & Vectors

Vectors can be used to represent a **point in space** or a **direction with a magnitude**. The magnitude of a vector can be determined by the `Length` attribute. Sometimes, only the direction is relevant (and not the magnitude). In this case it is often useful to normalise the vector:  $v/|v| = v/\sqrt{v_1^2, \dots, v_n^2}$ :

```
1 Vector3 v = new Vector3(2, 4, 8);  
2 v.Normalize();
```

Matrices represent linear transformations used to transform vectors. Using a  $4 \times 4$  matrix allows one to combine all these transformations into a single matrix. Matrices in XNA are **row major**:

$$\begin{bmatrix} R_x & R_y & R_z & R_w \\ U_x & U_y & U_z & U_w \\ -F_x & -F_y & -F_z & -F_w \\ T_x & T_y & T_z & T_w \end{bmatrix} \begin{array}{l} \text{Right (direction)} \\ \text{Up (direction)} \\ \text{-Forward = Backward (direction)} \\ \text{Translation (position)} \end{array}$$

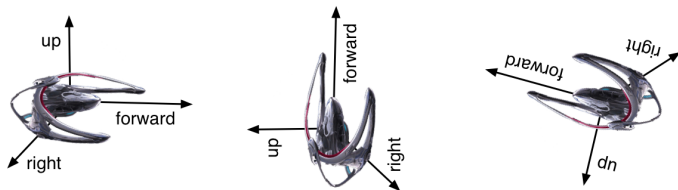
The following vectors may be obtain from a given matrix:

Right, Left, Up, Down, Forward, Backward, Translation

# World Transforms

Remember: each object has its own world matrix. The first three rows specify its **local** coordinate system. Usually, all three axes are at right angles to each other: they are *orthogonal*. Furthermore, unless scaled, these directions are usually represented by unit vectors; they are *normal*. Such a matrix is known as *orthonormal*.

When you transform this matrix with another matrix (e.g., rotation matrix), all axes will be rotated to reflect the change while maintaining their relationship to one another.



Note: local coordinate systems need not be aligned with the global coordinate system. It is thus more accurate to say that you rotate around the object's forward vector, rather than the z-axis etc.

# World Transforms

The world matrix specifies how each individual object is positioned in the world. You can transform the world matrix of each object to induce **rotation**, **translation** or **scale**.

**Translation:**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

**Scale:**

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**X-Rotation:**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \ominus & -\sin \ominus & 0 \\ 0 & \sin \ominus & \cos \ominus & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Y-Rotation:**

$$\begin{bmatrix} \cos \ominus & 0 & \sin \ominus & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \ominus & 0 & \cos \ominus & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Z-Rotation:**

$$\begin{bmatrix} \cos \ominus & -\sin \ominus & 0 & 0 \\ \sin \ominus & \cos \ominus & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Note on Vector4

When you multiply a vector by a matrix, the number of components in the vector must equal the number of rows in the matrix - this is where `Vector4` comes in. The fourth component,  $w$ , allows multiplication by the built-in type `Matrix` (the vector is treated as a  $4 \times 1$  matrix).

You don't need to work with `Vector4` directly - XNA converts a `Vector3` automatically into a `Vector4`. However, it is important to understand the role played by the fourth value:

- If your vector is a position vector:
  - $w = 1$
  - Use `Vector3.Transform()`.
- If you vector is a direction vector:
  - $w = 0$
  - Use `Vector3.TransformNormal()`

E.g., `Vector3.Transform(vector, matrix)` and `Vector3.TransformNormal(vector, matrix)`.

# Transformations

Enough theory. Let's do some transformations and observe how the matrices and vectors behave. Assume we have an object at the origin; no scaling or rotations have been applied. We declare its world as:

```
1 Matrix m = Matrix.Identity;
```

We now tilt the object forward by 45 degrees:

```
1 m *= Matrix.CreateRotationX(MathHelper.ToRadians(45));
```

Note: same as `m *= Matrix.CreateFromYawPitchRoll(0f, MathHelper.ToRadians(45), 0f);`

Now `m` is approximately: 
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.71 & 0.71 & 0 \\ 0 & -0.71 & 0.71 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
 **Q** What does this mean?

The vector `m.Up` is now 0.71 0.71. We could get this also this way:

```
1 Vector3.Transform(Vector3.Up, Matrix.CreateRotationX(MathHelper.ToRadians(45)));
```

# Outline

- 1 Intro to 3D: Revision
- 2 3D Models
- 3 More on Transformations
- 4 Concept of Cameras**
- 5 Sample Progress Test
- 6 Summary

# (World), View & Projection

The world matrix positions an object in 3D space. The view and projection are responsible for how we perceive said object. So far, we have simply used a standard view and projection. We will now see how to define different views and projections and then develop different camera types suitable for a variety of games.

The view matrix is very simple. We can create it as follows:

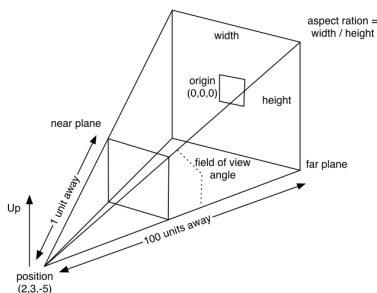
```
1 public static Matrix CreateLookAt (  
2     Vector3 cameraPosition,  
3     Vector3 cameraTarget,  
4     Vector3 cameraUpVector )
```

This simply specified where the camera is, where the camera points at and what direction is up. Typical values we have used so far include:

- `cameraPosition = new Vector3(3, 4, 8);`
- `cameraTarget = Vector3.Zero;`
- `cameraUpVector = Vector3.Up;`

# View & Projection

Defining the viewing frustum is much more involved and XNA provides numerous built-in methods to define different shapes for the “pyramid”:



You have the following options:

`CreatePerspectiveFieldOfView()`

`CreatePerspective()`

`CreatePerspectiveOffCenter()`

`CreateOrthographic()`

`CreateOrthographicOffCenter()`

**Q** What impact has the pyramid shape on how we perceive objects of different distance to the camera?

# Different Projections: Perspective

These matrices produce pyramid-shape viewing frustums.

CreatePerspectiveFieldOfView()

```
1 public static Matrix CreatePerspectiveFieldOfView (
2     float fieldOfView,
3     float aspectRatio,
4     float nearPlaneDistance,
5     float farPlaneDistance )
```

CreatePerspective()

```
1 public static Matrix CreatePerspective (
2     float width,
3     float height,
4     float nearPlaneDistance,
5     float farPlaneDistance )
```

CreatePerspectiveOffCenter()

```
1 public static Matrix CreatePerspectiveOffCenter (
2     float left,
3     float right,
4     float bottom,
5     float top,
6     float nearPlaneDistance,
7     float farPlaneDistance)
```

# Different Projections: Orthographic

Orthographic viewing frustums are more like a cube than a pyramid. In particular, the size of the near plane and the far plane is the same. Thus all objects of equal size are rendered the same size independent of distance to the camera.

CreateOrthographic()

```
1 public static Matrix CreateOrthographic (
2     float width,
3     float height,
4     float zNearPlane,
5     float zFarPlane )
```

CreateOrthographicOffCenter()

```
1 public static Matrix CreateOrthographicOffCenter (
2     float left,
3     float right,
4     float bottom,
5     float top,
6     float zNearPlane,
7     float zFarPlane )
```

We will now see how these viewing frustums differ in practice.

# Outline

- 1 Intro to 3D: Revision
- 2 3D Models
- 3 More on Transformations
- 4 Concept of Cameras
- 5 Sample Progress Test**
- 6 Summary

# Sample Progress Test

10% of your final mark will be determined by a 20 question progress test for which you will have 30 minutes to complete. Two example questions are as follows:

Given  $n$  vertices, how many triangles could be drawn using a `TriangleStrip`?

- ①  $3n$
- ②  $2n$
- ③  $n - 1$
- ④  $n - 2$

In XNA, a change in *yaw* would be represented as which one of the following?

- ① Rotation about the local  $x$  axis
- ② Rotation about the local  $y$  axis
- ③ Rotation about the local  $z$  axis
- ④ A translation along the local  $x$  axis

You are encouraged to go through the lecture notes, lab exercises and primary literature to review for this. Test will take place 2pm next Thursday.

Note: the test will only ask questions regarding topics covered so far.

# Outline

- 1 Intro to 3D: Revision
- 2 3D Models
- 3 More on Transformations
- 4 Concept of Cameras
- 5 Sample Progress Test
- 6 Summary**

# Summary

In this lecture you have been introduced to 3D models and DCC packages to create them. We animated the objects using various compound transformations and explored the 3D space using a range of camera implementations.

We covered

- 3D models
- Meshes, mesh parts and bones
- DCC packages
- Scaling models
- Bounding spheres
- More matrix and vector maths
- Rotations and translations
- Local & global coordinate systems
- Compound World transformations
- Different camera implementation

In the lab you will

- First you will create a simple model in Google SktechUp. Feel free to add textures to this model and to be creative with its shape. You can also have a look at the 3D warehouse for ideas.
- If you did not finish off the primitives from the last lab, you should try to get to the stage of drawing cubes. You should then have triangles, quads and cubes.
- Use the 3D world provided (from last lab) and populate it with automatically animated models and primitive geometries. You can use the camera provided to explore the world.

Next lecture: a free camera and an articulated robotic arm (+ progress test).