CE318: Games Console Programming

Lecture 3: From 2D to 3D - The Basics

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Outline

1. Introduction to 3D
2. Drawing Simple Geometries
3. Drawing Cubes, Pyramids & Cones
4. Simple World Transformations
5. Summary

Outline

1 Introduction to 3D

2 Drawing Simple Geometries

3 Drawing Cubes, Pyramids & Cones

4 Simple World Transformations

5 Summary

What is 3D?

In lectures 1 and 2 we dealt with 2D graphics which are relatively simple in XNA. 3D graphics are much more involved. At the core of any 3D graphic are many triangles that define its shape. This is known as a **geometry**. In order to bring this geometry to the screen, numerous attributes and processes are required:

- Position in the **world**
- **View** in the world
- **Projection** of the world
- Lighting
- Shadows
- Textures
- Rasterisation
- Pixel tests
- Blending

Why do we need all this? Because 3D graphics are depicted on 2D surfaces, so we need to transform them and create an illusion of depth (and, of course, to make the graphics look nice).

These processes are dealt with in the XNA **graphics pipeline**.

The Graphics Pipeline

3D graphics are composed of many triangles. The **vertices** that make up the triangles may contain numerous items of information, such as colours and textures. Additional information regarding lighting and shading is given by an **effect** which supplies the graphics pipeline with shader information. The graphics pipeline then renders the 3D object on the screen given our view of the world and the position of the geometry within it.

In general, the XNA 3D pipeline requires the following for initialisation:

1. World, view, and projection transforms to transform 3D vertices to 2D
2. A set of vertices that contains the geometry to render
3. An effect that sets the render state necessary for drawing the geometry

We will briefly look at these requirements next - they will be covered in much more depth in today's and upcoming lectures.

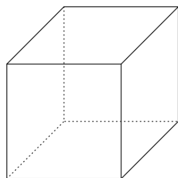
World, View and Projection

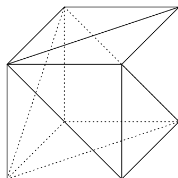
The world, view and projection transforms are required to map a 3D object onto a 2D surface:

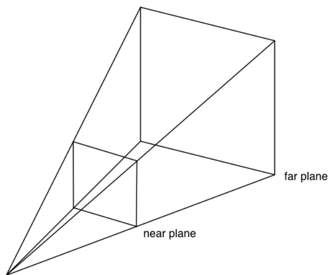
- **World**: describes the position of a particular object in the world. This includes information regarding the geometry's **translation**, **rotation** and **scale**.
- **View**: describes our view within the world. This includes the position of the **camera** and the target the camera points at.
- **Projection**: describes the **viewing frustum**. A viewing frustum is the extend to which we can see along the direction we are looking at.

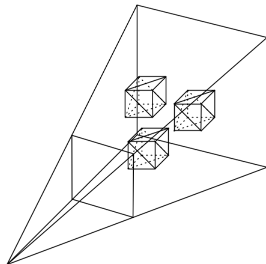
Together, these transforms determine which parts of the object are visible and hence drawn onto the screen. In order to create a realistic depiction (i.e., one that does not look flat), we also make use of **effects**.

What is 3D?









XNA Effects

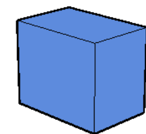
During rendering, the graphics pipeline transforms 3D geometry to a 2D surface, giving you the option of adding lighting, texturing and many other per-vertex or per-pixel visual effects. An effect initialises the pipeline to render 3D geometry using vertex and pixel shaders (although you can also render a 2D sprite with an effect).

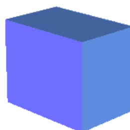
There are two types of effects.

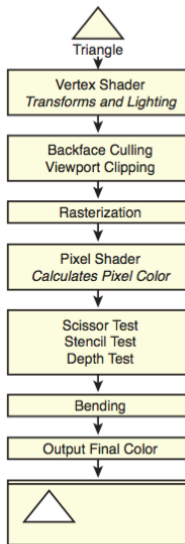
- **Configurable Effects**: this is an optimised rendering effect using a **built-in** object with options for user configuration. XNA provides five of these.
- **Programmable Effects**: A programmable effect is a **general purpose** rendering effect. It is created from vertex and pixel shaders written in the HLSL and is completely customisable.

The built-in effects will be covered in the next lecture, programmable effects and the High Level Shading Language (HLSL) will be covered in lecture 6.

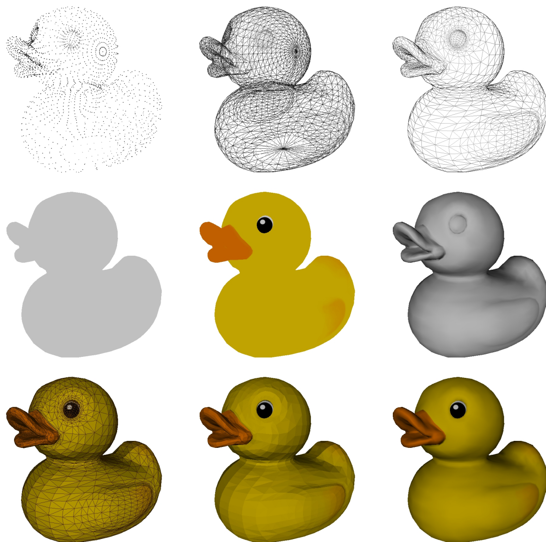
Graphics Pipeline & Effects







What is 3D?



Outline

1. Introduction to 3D
2. Drawing Simple Geometries
3. Drawing Cubes, Pyramids & Cones
4. Simple World Transformations
5. Summary

Drawing Simple Geometries

The following key components are required to display a 3D object:

1. **Geometry**: object to be rendered (vertices)
2. **World**: translation, rotation and scale of the object
3. **View**: position of camera and target
4. **Projection**: how the world is viewed (perspective)
5. **Effect**: how the object is displayed in 2D

2-5 essentially transform the geometry as follows:

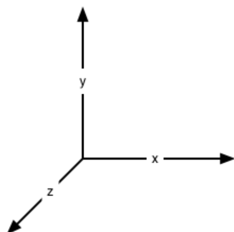
Model space $\longrightarrow$ World space $\longrightarrow$ View space $\longrightarrow$ Screen space

Today we use a simple stationary camera (more next lecture) and utilise XNA's `BasicEffect` (more in lecture 6). We will start off with a world matrix centred at the origin. Once we have drawn some geometries, we will look at transformations.

First, however, we need to understand 3D spaces, vectors and matrices.

3D Spaces & Vectors

The crucial difference between 2D and 3D is, of course, the extra dimension. In 2D, we have x and y axes. In 3D, we have x, y and z:



This coordinate system is known as a **right-handed** coordinate system and is the default in XNA.

Use your right hand to emulate the three axes.

To specify a point in the 3D space, we can use **vectors**:

- `Vector2`: for 2D spaces - (x, y)
- `Vector3`: for 3D spaces - (x, y, z)
- `Vector4`: for 3D spaces and matrix multiplications - (x, y, z, w)

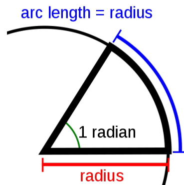
Q What units are vectors in?

XNA Vectors & Basic Geometry

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. Vectors in XNA have many properties and methods. We will cover most of these over the next few weeks.

Angles in XNA are specified in *radians* rather than *degrees*. Use `MathHelper.ToDegrees()` and `MathHelper.ToRadians()` to convert between the two.

Q How many radians are 360°?



Note: other useful functions found in `MathHelper` include `Clamp()`, `Max()`, `Min()` and `WrapAngle()`. Useful constants include `Pi`, `TwoPi`, `PiOver2`, `PiOver4`.

Background in Maths – Matrices (1/2)

Matrices form the second important aspect of 3D. XNA has a built-in data type `Matrix`, which is 4×4:

$$\begin{bmatrix} M11 & M12 & M13 & M14 \\ M21 & M22 & M23 & M24 \\ M31 & M32 & M33 & M34 \\ M41 & M42 & M43 & M44 \end{bmatrix}$$

Matrices represent linear transformations used to transform vectors. Using a 4×4 matrix allows one to combine all these transformations into a single matrix.

Q What transformations are there?

To draw objects in 3D spaces, some basic mathematical understanding of 3D, vectors and matrices is required. We will cover the different principles required as we go along.

Background in Maths – Matrices (2/2)

XNA provides many functions to construct matrices from one or more vectors. The `Vector4` may be used to transform a vector by a matrix. XNA automatically converts between `Vector3` and `Vector4`.

$$\begin{bmatrix} a & b \end{bmatrix} \times \begin{bmatrix} c & d \\ e & f \end{bmatrix} = \begin{bmatrix} ac + be & ad + bf \end{bmatrix}$$

Q What happens to a vector transformed by the identity matrix?

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We will use `Matrix.Identity` for now.

Create View and Projection

Finally we get to do some drawing. First, we need to be able to view the geometries we will be drawing. For this, we will use a simple, stationary **camera**.

The concept of a camera may be used to implement the View and Projection:

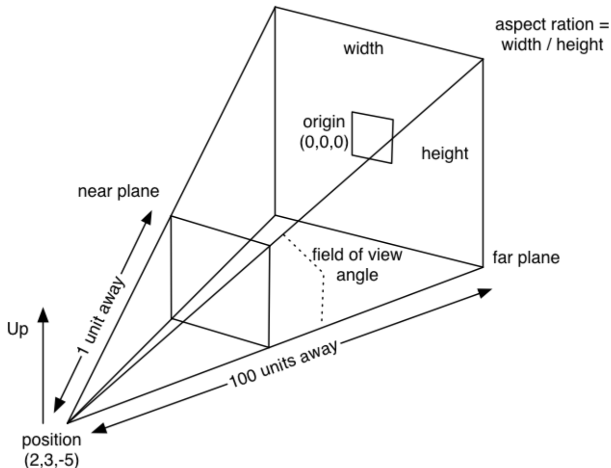
```
1 Matrix View = Matrix.CreateLookAt(new Vector3(1, 2, 7), Vector3.Zero,
      Vector3.Up)
```

This creates the view. The camera is at position $(1, 2, 7)$ and looks at point $(0, 0, 0)$. We also define which way is up.

```
1 Matrix Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.
      PiOver4, GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f));
```

This creates the viewing frustum: the angle specifies the field of view angle, the aspect ration determines the dimensions of the viewing rectangle, the value of 1 is the distance of the near field and 100 is the distance of the far field - see next slide.

The Viewing Frustum



We will cover different camera implementations in lecture 4.

3D: How to Draw Primitives

There are 4 primitive types in XNA define by the enumeration `PrimitiveType`:

1. `LineList`
2. `LineStrip`
3. `TriangleList`
4. `TriangleStrip`

These types determine how the **vertices** are combined to form a geometry. XNA has several different built-in vertex types:

1. `VertexPositionColor` – holds position and color
2. `VertexPositionColorTexture` – holds position, color and texture
3. `VertexPositionTexture` – holds position and texture
4. `VertexPositionNormalTexture` – holds position, normal and texture

It is also possible to specify custom vertex types.

How to Draw Primitives

There are several ways to draw geometries. Not only can we use different primitive types and vertices, we can also make use of indices (to save memory) and buffers (to improve speed). The `GraphicsDevice` provides the following four methods to draw primitives:

1. `DrawUserPrimitives`
2. `DrawIndexedUserPrimitives` – indexed
3. `DrawPrimitives` – buffered
4. `DrawIndexedPrimitives` – indexed and buffered

Finally, we also need an effect to tell the graphics pipeline how to deal with vertex and pixel shading. For now we will use the built-in effect `BasicEffect`.

In the following examples, we will use the View and Projection defined earlier and for the World we use `Matrix.Identity`.

Lines

There are two `PrimitiveType` to draw lines:

1. `LineList`: The data is ordered as a sequence of line segments; each line segment is described by two new vertices.
2. `LineStrip`: The data is ordered as a sequence of line segments; each line segment is described by one new vertex and the last vertex from the previous line segment.

First, we create some vertices:

```
1 VertexPositionColor[] vpc;
```

```
1 vpc = new VertexPositionColor[4];
2 vpc[0] = new VertexPositionColor(new Vector3(-1,  1, 0); Color.White);
3 vpc[1] = new VertexPositionColor(new Vector3(-1, -1, 0); Color.White);
4 vpc[2] = new VertexPositionColor(new Vector3( 1,  1, 0); Color.White);
5 vpc[3] = new VertexPositionColor(new Vector3( 1, -1, 0); Color.White);
```

Arguments: position, colour.

Lines

We then need to specify the `BasicEffect` (we use this code for all drawings).

```
1 BasicEffect basicEffect;
```

```
1 basicEffect = new BasicEffect(GraphicsDevice);
2 basicEffect.World = Matrix.Identity;
3 basicEffect.View = View;
4 basicEffect.Projection = Projection;
5 basicEffect.VertexColorEnabled = true;
```

Before drawing the primitives, we need to apply the effect:

```
1 basicEffect.CurrentTechnique.Passes[0].Apply();
```

Then draw the lines using

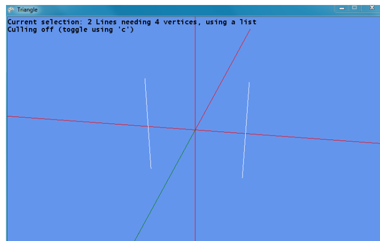
```
1 GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.
      LineList, vpc, 0, 2);
```

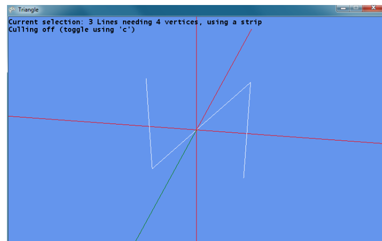
```
1 GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.
      LineStrip, vpc, 0, 3);
```

Arguments: `PrimitiveType`, actual primitives, offset, number of constructs.

What will the output be?

Lines





Triangles

We now move on to more exciting geometries: a triangle.

1. `TriangleList`: The data is ordered as a sequence of triangles; each triangle is described by three new vertices. Backface culling is affected by the current winding-order render state.

2. `TriangleStrip`: The data is ordered as a sequence of triangles; each triangle is described by two new vertices and one vertex from the previous triangle. The backface culling flag is flipped automatically on even-numbered triangles.

What is **backface culling**?

In order to improve speed, triangles that can't be seen are not drawn. This is called culling. By default, triangles are define clockwise in XNA. Counter-clockwise triangles are culled. Culling can be turned off using the `CullMode` of `GraphicsDevice.RasterizerState`.

Triangles

To draw a triangle, we first, we create some vertices:

```
1 VertexPositionColor[] vpc;
```

```
1 vpc = new VertexPositionColor[3];
2 vpc[0] = new VertexPositionColor(new Vector3( 0,  1, 0); Color.Red);
3 vpc[1] = new VertexPositionColor(new Vector3( 1, -1, 0); Color.Green);
4 vpc[2] = new VertexPositionColor(new Vector3(-1, -1, 0); Color.Blue);
```

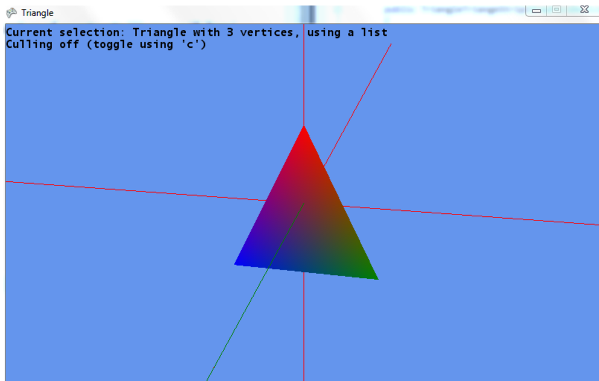
We then apply the effect and draw the triangle as follows:

```
1 device.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.
      TriangleList, vpc, 0, 1);
```

```
1 device.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.
      TriangleStrip, vpc, 0, 1);
```

Q Will there be any differences between the two draw methods?

Triangles



Triangles: Order of Vertices Matters

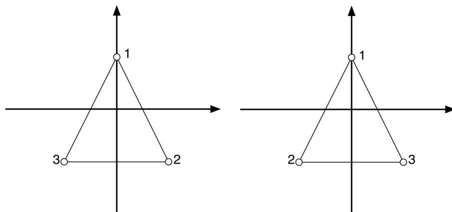
Compare these two implementations:

```
1 vpc[0] = new VertexPositionColor(new Vector3(0, 1, 0), Color.Red);
2 vpc[1] = new VertexPositionColor(new Vector3(1, -1, 0), Color.Green);
3 vpc[2] = new VertexPositionColor(new Vector3(-1, -1, 0), Color.Blue);
```

and

```
1 vpc[0] = new VertexPositionColor(new Vector3(0, 1, 0), Color.Red);
2 vpc[1] = new VertexPositionColor(new Vector3(-1, -1, 0), Color.Blue);
3 vpc[2] = new VertexPositionColor(new Vector3(1, -1, 0), Color.Green);
```

Q What is the difference?



Rectangle: Using List

Triangles may, of course, be used to draw arbitrary shapes.

Next: a rectangle.

```
1 VertexPositionColor[] vpc;
```

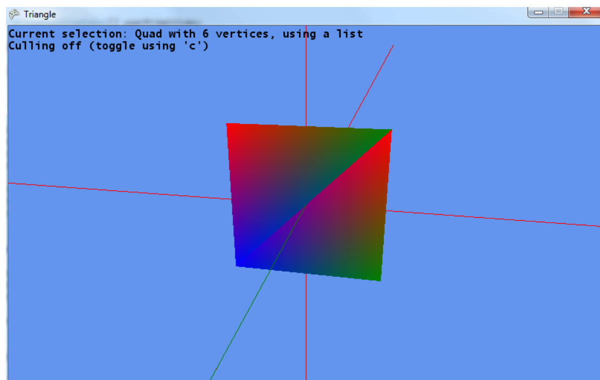
```
1 vpc = new VertexPositionColor[6];
2 //Triangle 1
3 vpc[0] = new VertexPositionColor(new Vector3(-1, 1, 0), Color.Red);
4 vpc[1] = new VertexPositionColor(new Vector3( 1, 1, 0), Color.Green);
5 vpc[2] = new VertexPositionColor(new Vector3(-1,-1, 0), Color.Blue);
6 //Triangle 2
7 vpc[3] = new VertexPositionColor(new Vector3( 1, 1, 0), Color.Red);
8 vpc[4] = new VertexPositionColor(new Vector3( 1,-1, 0), Color.Green);
9 vpc[5] = new VertexPositionColor(new Vector3(-1,-1, 0), Color.Blue);
```

Note: duplicate vertices.

```
1 device.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.
      TriangleList, vpc, 0, 2);
```

Note: primitive count is now 2. We are drawing two adjacent triangles.

Rectangle: Using List



Q How many vertices do we need if we use a triangle strip?

Rectangle: Using Strip

Note: culling order changes from triangle to triangle when using striping.

```
vpc[] vpc;
```

```
vpc = new VertexPositionColor[4];
//Triangle 1
vpc[0] = new VertexPositionColor(new Vector3(-1, 1, 0), Color.Red);
vpc[1] = new VertexPositionColor(new Vector3(1, 1, 0), Color.Green);
vpc[2] = new VertexPositionColor(new Vector3(-1, -1, 0), Color.Blue);
//Triangle 2
vpc[3] = new VertexPositionColor(new Vector3(1, -1, 0), Color.Blue);
```

We now only use 4 vertices but still have a primitive count of 2:

```
device.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.
     TriangleStrip, vpc, 0, 2);
```

Using a triangle strip is often more efficient as fewer vertices are required. However, it is possible to improve the memory requirements of lists using indices that allow re-use of already defined vertices. Also, buffers may be used to improve the speed of drawing. These two concepts are explored next.

Rectangle: Using List & Indices

```
VertexPositionColor[] vpc;
short[] indices;
```

Note: C# short is 16 bit.

```
vpc = new VertexPositionColor[4];
//Triangle 1
vpc[0] = new VertexPositionColor(new Vector3(-1, 1, 0), Color.Red);
vpc[1] = new VertexPositionColor(new Vector3( 1, 1, 0), Color.Green);
vpc[2] = new VertexPositionColor(new Vector3(-1,-1, 0), Color.Blue);
//Triangle 2
vpc[3] = new VertexPositionColor(new Vector3( 1,-1, 0), Color.Blue);

indices = new short[6];
//Triangle 1
indices[0] = 0;
indices[1] = 1;
indices[2] = 2;
//Triangle 2
indices[3] = 1;
indices[4] = 3;
indices[5] = 2;
```

We now have to use the draw method `DrawUserIndexedPrimitives()`:

```
device.DrawUserIndexedPrimitives<VertexPositionColor>(PrimitiveType.
    TriangleList, vpc, 0, 4, indices, 0, 2);
```

Arguments: `PrimitiveType`, actual primitives, vertex offset, number of vertices, index data, index offset, primitive count.

Rectangle: Using List & Indices & Buffers (1/2)

A buffer stores vertices on the graphics card forr quick access. Vertices do not need to be send to the graphics card from the main memory every time the geometry is drawn. Can use buffers for vertices (alone) and indices.

```
1 vpc[] primitives;
2 short[] indices;
3
4 VertexBuffer vertexBuffer;
5 IndexBuffer indexBuffer;
```

First, we define the vertices and indices.

```
1 vpc = new VertexPositionColor[4];
2 //Triangle 1
3 vpc[0] = new VertexPositionColor(new Vector3(-1, 1, 0), Color.Red);
4 vpc[1] = new VertexPositionColor(new Vector3(1, 1, 0), Color.Green);
5 vpc[2] = new VertexPositionColor(new Vector3(-1, -1, 0), Color.Blue);
6 //Triangle 2
7 vpc[3] = new VertexPositionColor(new Vector3(1, -1, 0), Color.Blue);
8
9 indices = new short[6];
10 //Triangle 1
11 indices[0] = 0;
12 indices[1] = 1;
13 indices[2] = 2;
14 //Triangle 2
15 indices[3] = 1;
16 indices[4] = 3;
17 indices[5] = 2;
```

Rectangle: Using List & Indices & Buffers (2/2)

Each buffer needs to be initialised once the vertices have been set.

```
1 vertexBuffer = new VertexBuffer(device, typeof(VertexPositionColor),
      vpc.Length, BufferUsage.None);
2 vertexBuffer.SetData<VertexPositionColor>(vpc);
3
4 indexBuffer = new IndexBuffer(device, IndexElementSize.SixteenBits,
      indices.Length, BufferUsage.None);
5 indexBuffer.SetData<short>(indices);
```

Finally, we can draw the geometry using the vertices, indices and buffers:

```
1 public override void Draw()
2 {
3   effect.VertexColorEnabled = true;
4
5   device.SetVertexBuffer(vertexBuffer);
6   device.Indices = indexBuffer;
7   effect.CurrentTechnique.Passes[0].Apply();
8
9   device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0, 4, 0,
         2);
10 }
```

Arguments: `PrimitiveType`, base vertex, min vertex index, number of vertices, starting index, primitive count.

Textured Quads (1/2)

It is possible to apply a texture to a quad drawn using triangles. For this we use the primitive type `VertexPositionTexture` and define where the texture is to be applied (this example uses a TriangleStrip):

```
1 VertexPositionTexture[] userPrimitives;
2 Texture2D texture;
```

We assume the texture has been loaded.

```
1 userPrimitives = new VertexPositionTexture[4];
2 //Triangle 1
3 userPrimitives[0] = new VertexPositionTexture(new Vector3(-1, 1, 0),
      new Vector2(0, 0));
4 userPrimitives[1] = new VertexPositionTexture(new Vector3(1, 1, 0),
      new Vector2(1, 0));
5 userPrimitives[2] = new VertexPositionTexture(new Vector3(-1, -1, 0),
      new Vector2(0, 1));
6 //Triangle 2
7 userPrimitives[3] = new VertexPositionTexture(new Vector3(1, -1, 0),
      new Vector2(1, 1));
```

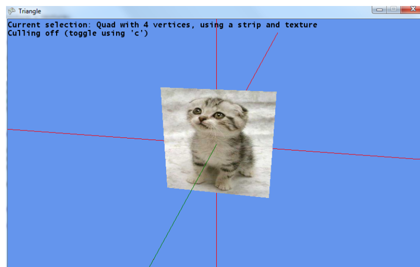
Note: we no longer supply a colour but instead a 2D vector.

Q What do the 2D coordinates represent?

Textured Quads (2/2)

We can then draw the textured quad:

```
1 public override void Draw()
2 {
3   effect.Texture = texture;
4   effect.TextureEnabled = true;
5   effect.CurrentTechnique.Passes[0].Apply();
6
7   device.DrawUserPrimitives<VertexPositionTexture>(PrimitiveType.
        TriangleStrip, userPrimitives, 0, 2);
8 }
```



Outline

1 Introduction to 3D

2 Drawing Simple Geometries

3 Drawing Cubes, Pyramids & Cones

4 Simple World Transformations

5 Summary

Drawing a Cube (1/4)

It is possible to draw any shapes using nothing but triangles. The simplest 3-dimensional shape is a cube. If we want to draw a cube without top or bottom, then:

- How many faces does it have?
- How many triangles are required?
- How many vertices are required?
  - No indexing using a list
  - No indexing using a strip
  - Indexing using a strip
  - Indexing using a list

We turn culling off for this:

```
1 RasterizerState rs;
2
3 rs = new RasterizerState();
4 rs.CullMode = CullMode.None;
5
6 GraphicsDevice.RasterizerState = rs;
```

Next we will draw the cube using a triangle strip and indices.

Drawing a Cube (2/4)

First, declare the class variables for the vertices and indices:

```
1 VertexPositionColor[] vpc;
2 short[] indices;
```

Then define the vertices:

```
1  userPrimitives = new VertexPositionColor[8];
2
3 vpc[0] = new VertexPositionColor(new Vector3(-1, -1, 1), Color.Blue);
4 vpc[1] = new VertexPositionColor(new Vector3(-1, 1, 1), Color.Blue);
5 vpc[2] = new VertexPositionColor(new Vector3(1, -1, 1), Color.Blue);
6 vpc[3] = new VertexPositionColor(new Vector3(1, 1, 1), Color.Blue);
7 vpc[4] = new VertexPositionColor(new Vector3(1, -1, -1), Color.Blue);
8 vpc[5] = new VertexPositionColor(new Vector3(1, 1, -1), Color.Blue);
9 vpc[6] = new VertexPositionColor(new Vector3(-1, -1, -1), Color.Blue);
10 vpc[7] = new VertexPositionColor(new Vector3(-1, 1, -1), Color.Blue);
```

Drawing a Cube (3/4)

Next define the indices:

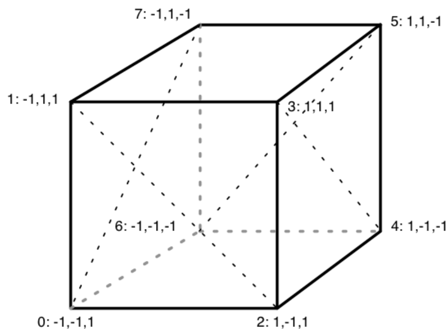
```
1  indices = new short[10];
2  //front
3  indices[0] = 0;
4  indices[1] = 1;
5  indices[2] = 2;
6  indices[3] = 3;
7  //right side
8  indices[4] = 4;
9  indices[5] = 5;
10 //back
11 indices[6] = 6;
12 indices[7] = 7;
13 //left side
14 indices[8] = 0;
15 indices[9] = 1;
```

Note: we reuse the first two vertices for the left face of the cube.

```
1  device.DrawUserIndexedPrimitives(PrimitiveType.TriangleStrip, vpc, 0,
       8, indices, 0, 8);
```

In the lab: you will be asked to draw a closed cube (i.e., with top and bottom) and a pyramid. Can you draw a cone also?

Drawing a Cube (4/4)



Outline

1. Introduction to 3D
2. Drawing Simple Geometries
3. Drawing Cubes, Pyramids & Cones
4. Simple World Transformations
5. Summary

World Transforms

The world matrix specified 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:

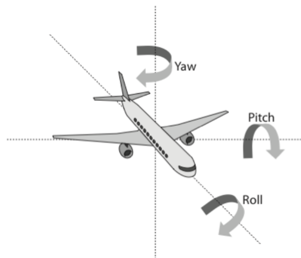
$$\begin{bmatrix} \cos \ominus & -\sin \ominus & 0 & 0 \\ \sin \ominus & \cos \ominus & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

XNA Matrices

XNA provides numerous method and properties (similar to Vector) to obtain specific matrices and to perform operations on these. To obtain the matrices from the previous slide:

- `Matrix.CreateTranslation()`
- `Matrix.CreateScale()`
- `Matrix.CreateRotationX()`
- `Matrix.CreateRotationY()`
- `Matrix.CreateRotationZ()`

Another very useful matrix is `CreateFromYawPitchRoll` which can rotate an object around multiple axes simultaneously.



Combining Matrix Transforms

Matrices may be used to transform vectors and may also be combined to yield composite transformations. We have created numerous geometries. It is now time to do something with them. We will scale, rotate and translate them. First, we do all of these in isolation, then combine them.

Remember: The 4×4 structure allows one to combine multiple transformations in a single matrix.

Add matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

Multiply matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

Scalar multiplication:

$$a \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae & af \\ ag & ah \end{bmatrix}$$

Vector multiplication:

$$\begin{bmatrix} a & b \end{bmatrix} \times \begin{bmatrix} c & d \\ e & f \end{bmatrix} = \begin{bmatrix} ac+be & ad+bf \end{bmatrix}$$

World Transforms

We will use a **model** for this. Models will be covered in the next lexture. We add our model (*box.fbx*) to the content pipeline and load it as follows:

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

The second line invokes the basic effect - don't worry about this for now.

We also need a view and projection as before:

```
1 view = Matrix.CreateLookAt(new Vector3(2, 3, 10), Vector3.Zero,
       Vector3.Up);
2 proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
       GraphicsDevice.Viewport.AspectRatio, 1.0f, 1000.0f);
```

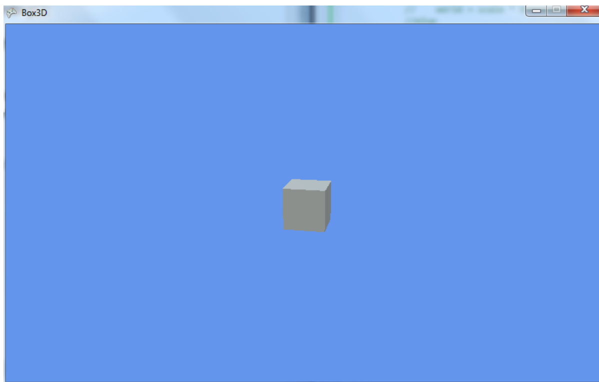
A model in XNA has a simple built-in draw method that takes the three required matrices as arguments:

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

For now, we use

```
1 Matrix world = Matrix.Identity;
```

World Transforms



Translation

To translate an object from the origin, we create a new Matrix:

```
Matrix translation = Matrix.Identity;
```

We enable translation via keyboard input:

```
if (Keyboard.GetState().IsKeyDown(Keys.D))
  translation *= Matrix.CreateTranslation(0.1f * Vector3.Right);
if (Keyboard.GetState().IsKeyDown(Keys.A))
  translation *= Matrix.CreateTranslation(0.1f * Vector3.Left);
if (Keyboard.GetState().IsKeyDown(Keys.W))
  translation *= Matrix.CreateTranslation(0.1f * Vector3.Up);
if (Keyboard.GetState().IsKeyDown(Keys.S))
  translation *= Matrix.CreateTranslation(0.1f * Vector3.Down);
```

We change the call to `model.Draw()` as follows:

```
model.Draw(translation, view, proj);
```

That's all: the user can now translate the cube along the x and y axes.

Rotation

Rotation may be achieved in a similar fashiont to translation:

```
1 Matrix rotation = Matrix.Identity;
```

Get user input:

```
1 if (Keyboard.GetState().IsKeyDown(Keys.Left))
2   rotation *= Matrix.CreateRotationY(0.1f);
3 if (Keyboard.GetState().IsKeyDown(Keys.Right))
4   rotation *= Matrix.CreateRotationY(-0.1f);
5 if (Keyboard.GetState().IsKeyDown(Keys.Up))
6   rotation *= Matrix.CreateRotationX(0.1f);
7 if (Keyboard.GetState().IsKeyDown(Keys.Down))
8   rotation *= Matrix.CreateRotationX(-0.1f);
```

Draw the model:

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

Scaling

Finally, to scale an object:

```
1 Matrix scale = Matrix.Identity;
```

Here we only have two options:

```
1 if (Keyboard.GetState().IsKeyDown(Keys.PageDown))
2   scale *= Matrix.CreateScale(0.99f);
3 if (Keyboard.GetState().IsKeyDown(Keys.PageUp))
4   scale *= Matrix.CreateScale(1.01f);
```

And once again we update the call to `model.Draw()`:

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

World Transforms

Now, we would like to carry out multiple transformations at once. It is important to note that the order in which individual transformations are applied are important. Rotations, for instance, are always rotations around the origin.

Q: What if we want to rotate an object not at the origin?

To apply all transforms simultaneously, we can combine them as follows:

```
1 model.Draw(scale * rotation * translation, view, proj);
```

Now, what would happen in the following cases?

```
1 model.Draw(scale * translation * rotation, view, proj);
```

```
1 model.Draw(rotation * translation * scale, view, proj);
```

In general, you will want to apply the transformations in the following order:

Scale $\longrightarrow$ **Rotate** $\longrightarrow$ **Translate**

Outline

1 Introduction to 3D

2 Drawing Simple Geometries

3 Drawing Cubes, Pyramids & Cones

4 Simple World Transformations

5 Summary

Summary

In this lecture you have been introduced to 3D graphics. 3D graphics are based on triangles that may be combined into complex shapes that are subsequently rendered onto the screen using effects and XNA's graphic pipeline.

We covered

- Rendering 3D
- Graphics pipeline
- World, View and Projection
- Effects and `BasicEffect`
- Vectors and matrices
- Primitive types
- Vertex types
- Drawing primitives
- Backface culling
- Lines, triangles, squares
- Textures, cubes and pyramids
- World transformations

Lab Preview

In the lab you will

- In the first part of the lab, you will draw a number of simple primitives in different ways, making use of `TriangleList` and `TriangleStrip` with and without indices and buffers. These shapes will include closed cubes and pyramids.
- In the second part of the lab, you will create a simple world full of the objects you created earlier. The objects will be scattered around randomly and you will animate all objects using simple transformations. You will be given a basic world and a simple camera implementation so you can explore the world you created.

Next lecture: Simple 3D models (including DCC packages), compound 3D models, more on transformations and different camera types.