CE318: Games Console Programming

Lecture 8: Effects, HLSL and Billboarding

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Outline

Today's Lecture

In today's lecture we will add some trees and clouds to our 3D world using a technique called *billboarding*. Billboarding uses 2-dimensional images that are rotated to constantly face the camera and hence approximate the appearance of 3D models.

Q Why not use 3D models instead?

To create the billboards, we need to make use of our own effects. For this, we need to look at the High Level Shader Language (HLSL) which is used to specify the shader. We will also use this opportunity to cover a bit more about XNA effects in general. You can use either the built-in effects or the custom ones to visually enhance your game (assignment).

Next lecture, we will extend the notion of billboards to create fire and smoke using particles.

Outline

Recap: What is an Effect?

During rendering, the graphics pipeline maps 3D geometries to a 2D surface. XNA effects initialise the graphics pipeline for performing transforms, lighting, applying textures, and adding per-pixel visual effects. An effect implements at least one shader for processing vertices and at least one shader for processing pixels.

There are two types of effects in XNA:

- **Configurable Effects**: A configurable effect is an optimised rendering effect using a built-in object with options for user configuration.

- **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.

High Level Shader Language (HLSL)

HLSL is used to communicate with the graphics card.

Configurable Effects

In essence, all effects use HLSL to communicate with the graphics card. To simplify matters, XNA provides 5 built-in effects that do this for us:

- `BasicEffect`: Basic Lighting and Fog
- `SkinnedEffect`: Character Animation
- `DualTextureEffect`: More Sophisticated Lighting with a Light Map
- `AlphaTestEffect`: Billboards and Imposters
- `EnvironmentMapEffect`: Lighting Highlights Using an Environment Map

We can use built-in functions to create a limited set of effects, such as directional lighting or fog.

If the built-in effects are insufficient, we can make use of programmable effects: combining a vertex shader and a pixel shader in a programmable effect gives you tremendous flexibility over the way the pipeline processes your data.

Programmable Effects

These steps are required to create a programmable effect:

- Design the shaders using HLSL.
  - An effect usually contains one vertex and one pixel shader.
  - Specify a technique that invokes the shaders.
  - The technique and the shaders are stored in an effect (.fx) file.
- Create an effect object in your game using the Effect class.
  - Load the effect file using the Content Pipeline.
- Initialize the effect parameters (global variables in the effect file).
- Apply the effect to the graphics device and render the scene.

The custom effect can be applied in two ways:

- Load the model and replace output data with own effect at runtime.
- Use custom processor to attach custom effect when the model is built.

We will consider the first approach only.

Outline

High Level Shader Language (HLSL)

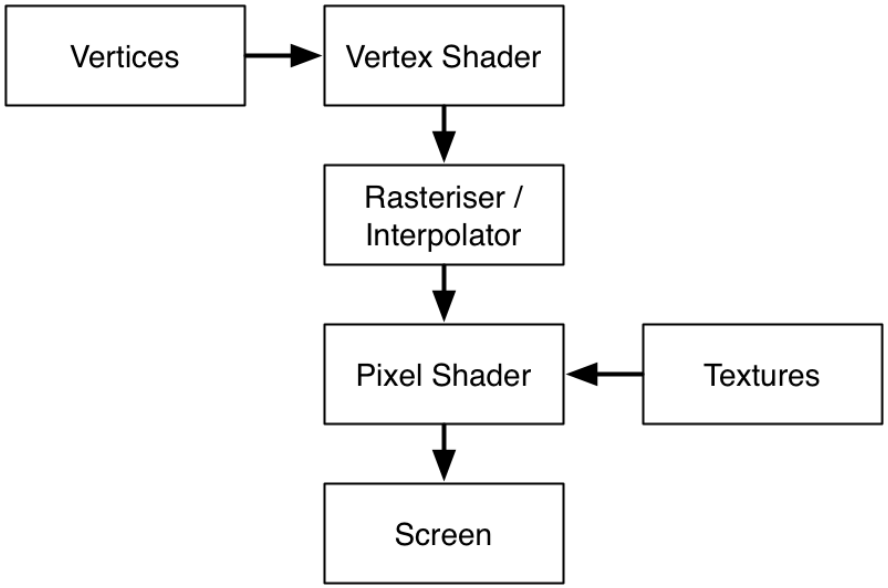
- We need to send instructions to graphics device
  - Initially done using assembly, we now use HLSL
- HLSL started off as joint project between Microsoft and NVIDIA
  - Uses a syntax similar to C
  - Translates instructions to assembly using built-in constructs and functions

Everything 3D in XNA uses HLSL

- Xbox supports an extended set; Windows Phone 7 doesn't support it at all

Graphics cards support two types of shaders:

- **vertex shader**: run once for every vertex in the field of view
- **pixel shader**: run on every pixel in all visible objects in the scene



High Level Language (HLSL)

The vertex shader sets the positions of the vertices based on the world and camera settings. The rasterisation process subsequently transforms the triangles into pixels to be rendered on the screen. Finally, the pixel shader assigns the correct colour to each (visible; determined by depth check) pixel.

To create a new effect file, we simply use the template provided (effects are added to the content pipeline). Effects have an *.fx* extension. The syntax used is similar to C and uses keywords such as:

- `bool`, `int`, `if`, `else`, `true`, `false`
- `sampler`, `technique`, `pixelshader`

as well as numerous *intrinsic* functions, including:

- `dot(a, b)`, `clip(x)`, `abs(x)`, `all(x)`, `any(x)`
- `clamp(x, min, max)`, `cross(x, y)`, `log(x)`

Compilation errors will be reported but code completion / highlighting is unavailable.

Effect Template File

```
1  float4x4 World;
2  float4x4 View;
3  float4x4 Projection;
4
5  struct VertexShaderInput {
6      float4 Position : POSITION0;
7  };
8
9  struct VertexShaderOutput {
10     float4 Position : POSITION0;
11 };
12
13 VertexShaderOutput VertexShaderFunction(VertexShaderInput input) {
14     VertexShaderOutput output;
15
16     float4 worldPosition = mul(input.Position, World);
17     float4 viewPosition = mul(worldPosition, View);
18     output.Position = mul(viewPosition, Projection);
19
20     return output;
21 }
22
23 float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0 {
24     return float4(1, 0, 0, 1);
25 }
26
27 technique Technique1 {
28     pass Pass1 {
29         VertexShader = compile vs_2_0 VertexShaderFunction();
30         PixelShader = compile ps_2_0 PixelShaderFunction();
31     }
32 }
```

Examining the Effect Template File

First there are some global variables:

```
1 float4x4 World;
2 float4x4 View;
3 float4x4 Projection;
```

These are 4×4 matrices. Vectors may be specified using `float4`. Such global variables can be set from XNA; they are also known as **effect parameters**.

The next two declarations define the input and output of the vertex shader:

```
1 struct VertexShaderInput
2 {
3     float4 Position : POSITION0;
4 };
```

```
1 struct VertexShaderOutput
2 {
3     float4 Position : POSITION0;
4 };
```

In this case, the input and output is simply the position of the vertex before and after the transformation by the world, view and projection. The output from the vertex shader is then interpolated and sent to the pixel shader.

Semantics

The term following the variable declaration (i.e., `POSITION0`) is known as a (HLSL) semantic.

> *A semantic is a string attached to a shader input or output that conveys information about the intended use of a parameter.*
>
> *Microsoft*

Semantics are essentially a way to connect HLSL variables with data XNA has access to. In this case, since we execute the vertex shader for every single vertex in the scene, the value will simply be the position of that vertex.

Note: semantics mean different things depending on whether the attached variable is used for vertex or pixel shaders and for input or output. You should consult MSDN to get a full overview of all semantics available.

Vertex Shader

The next element in the effect file is the actual vertex shader:

```
1  VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
2  {
3    VertexShaderOutput output;
4
5    float4 worldPosition = mul(input.Position, World);
6    float4 viewPosition = mul(worldPosition, View);
7    output.Position = mul(viewPosition, Projection);
8
9    return output;
10 }
```

This takes as argument the type of input defined earlier and produces an instance of the output which was also defined earlier. It also make use of the effect's parameters.

`mul(a, b)` is an intrinsic (built-in) function for matrix multiplication.

Q Can you tell what the code does?

Pixel Shader and Technique

The second function we need to provide is the pixel shader:

```
1 float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
2 {
3   return float4(1, 0, 0, 1);
4 }
```

This takes as input the output from the vertex shader. This does not always need to be the case and, in fact, the data is not used by the vertex shader at all.

Q Can you identify what the code does?

Finally, the last function we need to provide is how the rendering is performed:

```
1 technique Technique1
2 {
3   pass Pass1
4   {
5     VertexShader = compile vs_2_0 VertexShaderFunction();
6     PixelShader = compile ps_2_0 PixelShaderFunction();
7   }
8 }
```

Each HLSL file will have one or more techniques, each can have one or more passes. Remember: `BasicEffect` has a single pass.

Shaders and Parameters

The types `VertexShader` and `PixelShader` are HSLS objects. We need to specify how they are compiled. In this case, we use vertex and pixel shaders version 2.

Q Do you remember the differences between Reach and HiDef?).

The last thing required is to set the effect's parameters (global variables) from within XNA. This is very easy:

- Access all parameters via `effect.Parameters`
- Use name to quantify parameter: `effect.Parameters["myParameter"]`
- Use `SetValue` to supply value: `effect.Parameters["world"].setValue(m)`

So, we have two forms of exchanging data between XNA and HLSL:

- Set global variables directly from XNA
- Use semantics to attach a meaning to input and output variables

Custom Effects Examples

That is it. It is now time to see custom effects in action. Note: we have only covered the very basics of HLSL, enough to understand the effects used for billboarding. If you want more information, you should pick up one of many books dedicated to HLSL.

We will illustrate custom effects using (textured) quads which we also used in lecture 3. We will look at the following effects:

- Template effect (red quad)
- Colour effect (quad with interpolated colours)
- Replica effect (accurately display texture on quad)
- Invert effect (display negative texture)
- Gray Scale effect (display texture in gray scale)
- Blur effect (blur the texture)

We will also look at tiling.

HLSL Summary

- Global variables are the effect's parameters
  - They aren't given semantics; their values are supplied via XNA
- The name of the technique needs to be set from XNA
- An input parameter with a semantic will automatically receive its data
  - Data corresponds to meaning of the semantic when the effect is executed
  - An output variable with a semantic flag "sets" the appropriate data
- A vertex shader has different semantics for inputs and outputs
- A vertex shader must at least specify a `POSITION[n]` semantic for its output

Adopted from pp 275-276 A. Reed.

Outline

Billboarding

Billboarding is a technique where 2D textures are drawn onto 3D rectangles (quads) placed in the scene. Usually, the quads are rotated towards the camera such that they will always face the viewer. This allows one to create the illusion of a model albeit at a much reduced cost. A common usage is trees in the background or clouds in the sky. The types of billboard we consider are:

- Cylindrical
- Spherical
- Non-rotating (interleaved)

It is common to use billboards in the distance and replace them with models once the gamer gets sufficiently close. This is a common approach in many "level of detail" systems. Next lecture, we will use billboards to create fire and smoke (particles).

The implementation of billboards is based on chapter 6 in "3D Graphics with XNA Game Studio 4.0" by S. James but has been simplified.

Billboarding

First we need to be able to draw textured quads. We will use the code from the effects demo and use it to build a `BillboardSystem` class. We also require a set of positions of where to draw the quads and an effect that

- Rotates the billboards towards the camera
- Deals with transparency
- Is able to ensure proper occlusion

Q What information do we need to perform the rotation?
Q Why would we want the effect to perform the rotation?

The class `BillboardSystem` contains the following attributes:

- A quad (using index and vertex buffers)
- Number of billboards
- Size of each billboard
- Position of each billboard
- Texture to be drawn (e.g., tree, cloud)
- An effect (`BillboardEffect.fx`)

Billboarding

First we specify the quad using XNA primitives. We are actually drawing all vertices in the exact same position. Only the texture is placed correctly. We will adjust this inside the effect later.

```
1  // For each billboard...
2  for (int i = 0; i < nBillboards * 4; i += 4)
3  {
4    Vector3 pos = particlePositions[i / 4];
5
6    // Add 4 vertices at the billboard's position
7    particles[i + 0] = new VertexPositionTexture(pos, new Vector2(0,0));
8    particles[i + 1] = new VertexPositionTexture(pos, new Vector2(0,1));
9    particles[i + 2] = new VertexPositionTexture(pos, new Vector2(1,1));
10   particles[i + 3] = new VertexPositionTexture(pos, new Vector2(1,0));
11
12   // Add 6 indices to form two triangles
13   indices[x++] = i + 0;
14   indices[x++] = i + 3;
15   indices[x++] = i + 2;
16   indices[x++] = i + 2;
17   indices[x++] = i + 1;
18   indices[x++] = i + 0;
19 }
```

Billboarding

We have to set all the parameters for the effect:

```
1  void setEffectParameters(Matrix View, Matrix Projection, Vector3 Up,
       Vector3 Right)
2  {
3    effect.Parameters["ParticleTexture"].SetValue(texture);
4    effect.Parameters["View"].SetValue(View);
5    effect.Parameters["Projection"].SetValue(Projection);
6    effect.Parameters["Size"].SetValue(billboardSize / 2f);
7    effect.Parameters["Up"].SetValue(Up);
8    effect.Parameters["Side"].SetValue(Right);
9
10   effect.CurrentTechnique.Passes[0].Apply();
11 }
```

We need to supply the up and right vectors of the camera to make sure we can rotate the billboard properly.

The draw method then simply:

- Sets the buffers
- Sets the effect parameters (call to method `setEffectParameters()`)
- Draws the billboard (using built-in draw method)

Billboarding Effect I

```
1  float4x4 View;
2  float4x4 Projection;
3  float2 Size;
4  float3 Up;
5  float3 Side;
6  texture ParticleTexture;
7
8  sampler2D texSampler = sampler_state
9  {
10   texture = <ParticleTexture>;
11 };
12
13 struct VertexShaderInput
14 {
15   float4 Position : POSITION0;
16   float2 UV : TEXCOORD0;
17 };
18
19 struct VertexShaderOutput
20 {
21   float4 Position : POSITION0;
22   float2 UV : TEXCOORD0;
23 };
24
25 VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
26 {
27   VertexShaderOutput output;
28
29   float3 position = input.Position;
30
```

Billboarding Effect II

```
31    // Determine which corner of the rectangle this vertex represents
32    float2 offset = float2((input.UV.x - 0.5f) * 2.0f, -(input.UV.y -
          0.5f) * 2.0f);
33
34    // Move the vertex along the camera's 'plane' to its corner
35    position += offset.x * Size.x * Side + offset.y * Size.y * Up;
36
37    // Transform the position by view and projection
38    output.Position = mul(float4(position, 1), mul(View, Projection));
39    output.UV = input.UV;
40
41    return output;
42  }
43
44  float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
45  {
46    float4 color = tex2D(texSampler, input.UV);
47
48    return color;
49  }
50
51  technique Technique1
52  {
53    pass Pass1
54    {
55      VertexShader = compile vs_2_0 VertexShaderFunction();
56      PixelShader = compile ps_2_0 PixelShaderFunction();
57    }
58  }
```

Billboarding

We have seen most parts of this effect before. The most interesting part is the vertex shader.

```
1  VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
2  {
3    VertexShaderOutput output;
4    float3 position = input.Position;
5
6    float2 offset = float2((input.UV.x - 0.5f) * 2.0f, -(input.UV.y -
         0.5f) * 2.0f);
7    position += offset.x * Size.x * Side + offset.y * Size.y * Up;
8
9    output.Position = mul(float4(position, 1), mul(View, Projection));
10   output.UV = input.UV;
11
12   return output;
13 }
```

First we get the position of the vertex (POSITION0). We then calculate the offset (remember, all vertices are at the same location when the quad is created), including the camera's local coordinate system. Finally, once we have the position, we transform it via the view and projection.

Billboarding

This is pretty much it. We now have the code to draw a textured quad that rotates according to the cameras orientation. We now need to instantiate the billboards. For this we create positions randomly and then instantiate `BillboardSystem` with the desired arguments.

Demo.

We can see that the empty regions of the billboard are not transparent. This can be fixed easily by enabling *alpha blending*:

```
1 graphicsDevice.BlendState = BlendState.AlphaBlend;
2
3 graphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
      4 * nBillboards, 0, nBillboards * 2);
4
5 graphicsDevice.BlendState = BlendState.Opaque;
```

Q Will this solve our problems?

Demo.

Billboarding

We now have billboards with transparency but the transparent pixels mask the billboards that are behind it.

Solution: we just draw 'solid' pixels

- Discard all pixels with an alpha value less than x
- We check for this in our pixel shader

```
1  bool AlphaTest = true;
2  float AlphaTestValue = 0.5f;
3
4  float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
5  {
6    float4 color = tex2D(texSampler, input.UV);
7
8    if (AlphaTest)
9      clip((color.a - AlphaTestValue));;
10
11   return color;
12 }
```

Note: we are discarding all pixels with an alpha value less than 0.5. This might not be desirable. Instead, we can draw the billboards in two passes: first, the solid pixels with depth buffering, then the transparent ones without depth buffering.

Billboarding

Now our trees display correctly: demo.

But, what if we are on top of the trees (e.g., in a plane?). We are using **spherical** billboarding for the trees. It would be better to use **cylindrical** billboarding: rotate around one axis only (up in this case).

We thus add the option to switch between the two:

```
1 public enum BillboardMode { Cylindrical, Spherical };
2 public BillboardMode Mode = BillboardMode.Spherical;
```

We then replace:

```
1 effect.Parameters["Up"].SetValue(Up);
```

with

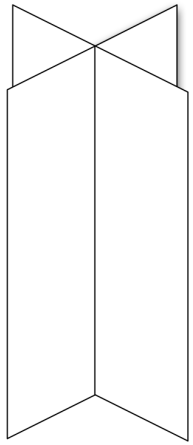
```
1 effect.Parameters["Up"].SetValue(Mode == BillboardMode.Spherical ? Up
      : Vector3.Up);
```

Demo.

There is one more type of billboarding: non-rotating billboards.

Non-Rotating Billboarding

We now need to draw to rectangles at 90 degrees to one another. This time we specify the exact positions in 3D space since we require no further changes to that.



We can use much of the code we developed earlier and change:

- We need to draw two quads instead of one
- There is no choice between spherical and cylindrical rotations
- We thus don't need the up and right / side vectors
- The effect can be simplified (no rotation required)
- We should turn off culling to make sure the trees are visible from all sides

Demo.

Some Notes about Billboarding

Some textures work much better for billboarding that others:

- With trees, for instance, the trunk should be round as this prevents the gamer from noticing the rotation (compare the trees we saw earlier).

- The drawn tree did not look good at all with non-rotating billboards because the two planes are too obvious. A more complex and realistic tree overcomes this issue.

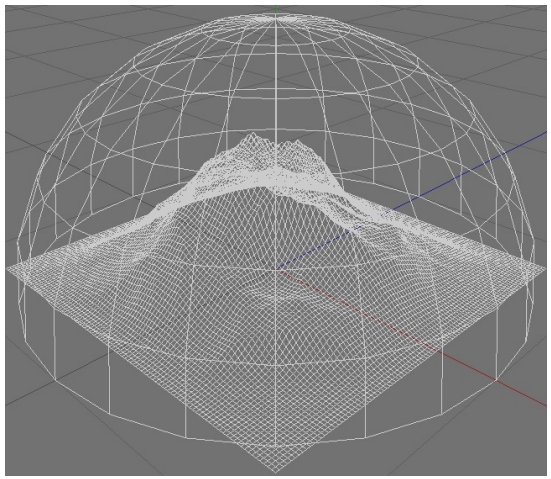
With billboarding, one can draw a lot more objects than would otherwise be possible. In your game you can try to combine billboarding with models to ensure the gameplay experience is consistent.

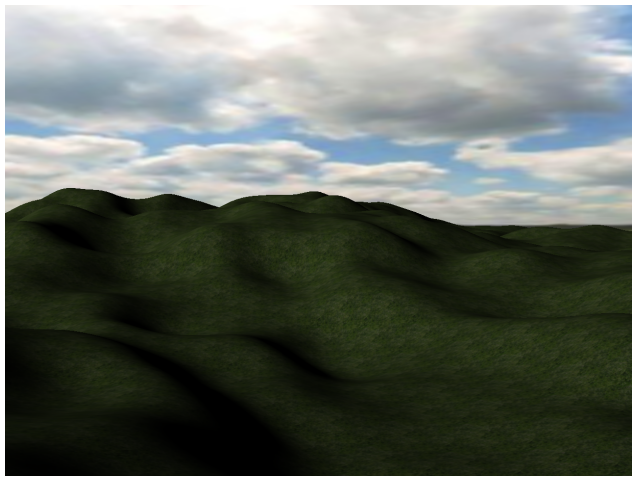
The way we drew the textures, we require transparent backgrounds. If you use an image from somewhere without a transparent background, use Paint.NET to remove the background (use magic wand and bucket tools).

Sky Dome

There is a simple alternative to billboard clouds that can often look nicer: sky boxes or sky domes. The former is more difficult to implement and use as one needs to get the texturing right. The latter, however, is extremely easy.

A sky dome is essentially just a textured model: a dome that can be placed in the world such that it encompasses the entire viewing frustum. We can then use a texture to display a sky with clouds etc.





The example I am showing was actually taken from the Google 3D warehouse.

Outline

Assignment

Some more details regarding your assignment (also see lecture 5):

- You should have a clear idea of what game you intend to produce
  - Identify a target audience
  - The game should have simple and clearly defined rules
  - The game must have a clear objective / goal
  - The game should have a good degree of variety (e.g., different levels)
  - All these design decision should go into your report
- By now you should already have written the foundation of your game
  - You should aim to complete a basic version of your game first
  - Once the game works as expected, you should try to refine it
    - Improve controls (using gamepad)
    - Visuals: better graphics, visual effects and sounds
- Graphics, sounds and effect:
  - They can be simple but need to work together to create a good atmosphere
- There should be an opponent that is fun to play against
  - Use FSMs or BTs, even rule-based systems

Assignment

In general, you can expect higher marks if your game is fun to play. A less complex game that is implemented very well will score higher than a complex game that has gameplay issues. However, the game should not be too simple either (use the game we developed in the course as guidance).

Remember: game has to be in 3D. However, 2D games can often serve as inspiration. Some useful examples from the app hub:

- Fuel cell game
- Space ship combat game
- Spacewar (2D)

The app hub also has numerous code sample for a more immersive gameplay experience:

- Chase camera
- Camera shake

Lab

Last week you were asked to create the pathfinding network (graph) required for the NPC-AI. In case you did not finish this, the answer is as follows:

```
1  // pass 1: generate all the nodes
2  for (int row = 0; row < map.Rows; row++) {
3    for (int col = 0; col < map.Columns; col++) {
4      if (!Map.solids.Contains(map.Get(row, col)))
5        nodes.Add(new N(row, col));
6
7  //pass 2: assign references to neighbouring nodes
8  for (int i = 0; i < nodes.Count(); i++) {
9    int[,] coods = { { nodes[i].row, nodes[i].column - 1 } ,
10   { nodes[i].row, nodes[i].column + 1 } ,
11   { nodes[i].row - 1, nodes[i].column } ,
12   { nodes[i].row + 1, nodes[i].column } ,        //4-way end
13   { nodes[i].row - 1, nodes[i].column - 1 } ,
14   { nodes[i].row - 1, nodes[i].column + 1 } ,
15   { nodes[i].row + 1, nodes[i].column - 1 } ,
16   { nodes[i].row + 1, nodes[i].column + 1 } };  //8-way end
17
18   for (int j = 0; j < connect; j++)
19   {
20     if (coods[j, 0] >= 0 && coods[j, 1] >= 0 && coods[j, 0] < map.Rows
           && coods[j, 1] < map.Columns && !Map.solids.Contains(map.Get
          (coods[j, 0], coods[j, 1]))) {
21       if(j<4)
22         nodes[i].adj.Add(new E(Find(coods[j, 0], coods[j, 1]),
               straightCost));
23       else
24         nodes[i].adj.Add(new E(Find(coods[j, 0], coods[j, 1]),
               diagCost));
25   }}}
```

Outline

Summary

In this lecture we covered XNA effects (built-in and custom) and looked in detail at billboarding to create a large number of objects in our game efficiently.

We covered

- Built-in (configurable) effects
- Custom (programmable) effects
- HLSL
- Skydomes
- Cylindrical billboarding
- Spherical billboarding
- Non-rotating billboarding
- Assignment

In the lab you will implement some advanced AI behaviour using behaviour trees. The game will be a simple capture-the-flag scenario. There will be two goal-oriented behaviours:

- Capture the flag
- Kill the enemy