CE318: Games Console Programming

Lecture 9: Particles & Terrain

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Today's Lecture

In today's lecture, we will look at two techniques that allow us to create a significiant visual impact in our game at a moderate cost.

The first technique is **particles** which uses billboards that have movement and a life-span. Particles may be used to create convincingly looking effects such as fire and smoke.

We will use particles to place some rusty barrels in our world that have flames and smoke coming out of them. You can also use partciles to indicate damage in your opponent, etc.

The second technique we will look at are **heightmaps** which we will used to generate terrain (i.e., mountains). A heightmap is a 2D image that determines the height of the terrain using gray scale.

We will use mountain ranges to create the boundaries of our world.

For both techniques, we will follow the samples by S. James (chapters 6 & 7).

Outline

Outline

Billboarding Reminder

Billboarding is a technique where 2D sprites are drawn onto quads that are rotated to consistently face the camera. This creates an illusion of depth. There are different types of billboarding:

- Spherical
- Cylindrical
- Non-rotating

Q What are the differences between these types of billboards?

Remember:

- Billboarding is computationally cheaper than using 3D models
- We can perform the rotations on the GPU
- Billboarding can be used in *level of detail* systems (i.e., with models)

Another common use of billboards are particles where billboards are animated.

Introduction to Particles

Particles are billboards that

- are rotated towards the camera
- have a starting position, direction and speed
- have a lifetime
- are typically faded in and out for smoother transitions
- attributes like wind may be used to create better animations

To achieve these things, we will use our previous billboard implementation with added functionality and an extended effect file to specify the billboards' behaviour over time.

We will create the following

- A custom vertex type
- A class called `ParticleSystem`
- An effect called `ParticleEffect`
- We will use these to create fire and smoke coming out of rusty barrels

Custom Vertex Types

Since our billboards have numerous attributes such as position, direction and speed, and our effect requires these, we need to store this information in the vertices themselves. But XNA only offers the following 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

However, all of these inherit from the interface `IVertexType`. We can use this interface to define our own vertex type. All we need to do is:

- Create a `struct` that inherits `IVertexType`
- Create all the fields / properties / class variables required
- Define a **vertex declaration** to tell the graphics card what is what

We can then use our custom vertex type like any other.

Vertex Declarations

A vertex declaration is a way to tell the graphics card what data the vertex contains. Let's assume we define a vertex with a *position* and a *color*:

```
1  public readonly static VertexDeclaration VertexDeclaration =
2    new VertexDeclaration (
3          new VertexElement(
4                0,
5                VertexElementFormat.Vector3,
6                VertexElementUsage.Position,
7                0),
8        new VertexElement(
9                sizeof(float) * 3,
10               VertexElementFormat.Color,
11               VertexElementUsage.Color,
12               0));
```

- The first argument specifies the offset in the vertex stream in bytes
- The second argument specifies the actual format of the data
- The third argument specifies the usage of the data
  - For XNA to link data from vertex stream to variables in the vertex shader
  - Think semantics
  - Note: `Vector3` $\neq$ `Position`; `Color` $=$ `Color`
- The fourth argument is the index of the information passed through

ParticleVertex I

We use the following custom vertices for the particles:

```
1  struct ParticleVertex : IVertexType
2  {
3    Vector3 startPosition;
4    Vector3 direction;
5    Vector2 uv;
6    float speed;
7    float startTime;
8
9    public Vector3 StartPosition
10   {
11     get { return startPosition; } set { startPosition = value; }
12   }
13
14   public Vector2 UV
15   {
16     get { return uv; }     set { uv = value; }
17   }
18
19   public Vector3 Direction
20   {
21     get { return direction; } set { direction = value; }
22   }
23
24   public float Speed
25   {
26     get { return speed; }    set { speed = value; }
27   }
28
```

ParticleVertex II

```
29    public float StartTime
30    {
31      get { return startTime; } set { startTime = value; }
32    }
33
34    public ParticleVertex(Vector3 StartPosition, Vector2 UV, Vector3
          Direction, float Speed, float StartTime)
35    {
36      this.startPosition = StartPosition; this.direction = Direction;
37      this.uv = UV; this.speed = Speed; this.startTime = StartTime;
38    }
39
40    public readonly static VertexDeclaration VertexDeclaration =
41      new VertexDeclaration(
42        new VertexElement(0, VertexElementFormat.Vector3,
              VertexElementUsage.Position, 0),
43        new VertexElement(12, VertexElementFormat.Vector2,
              VertexElementUsage.TextureCoordinate, 0),
44        new VertexElement(20, VertexElementFormat.Vector3,
              VertexElementUsage.TextureCoordinate, 1),
45        new VertexElement(32, VertexElementFormat.Single,
              VertexElementUsage.TextureCoordinate, 2),
46        new VertexElement(36, VertexElementFormat.Single,
              VertexElementUsage.TextureCoordinate, 3));
47
48    VertexDeclaration IVertexType.VertexDeclaration
49    {
50      get { return VertexDeclaration; }
51    }}
```

Creating ParticleSystem

Next we create a class called `ParticleSystem` with the following properties:

```
1  class ParticleSystem
2  {
3    GraphicsDevice graphicsDevice;
4    Effect effect;
5
6    VertexBuffer verts;
7    IndexBuffer ints;
8    ParticleVertex[] particles;
9    int[] indices;
10
11   int nParticles;
12   float lifespan = 1;
13   float fadeInTime;
14   Vector2 particleSize;
15   Vector3 wind;
16   Texture2D texture;
17
18   int activeStart = 0, nActive = 0;
19   DateTime start;
20 }
```

We then add:

- Constructor to initialise the class variables
- 3 public methods: `AddParticle()`, `Update()`, `Draw()`
- 2 helper methods: `generateParticles()`, `offsetIndex()`

Creating ParticleSystem

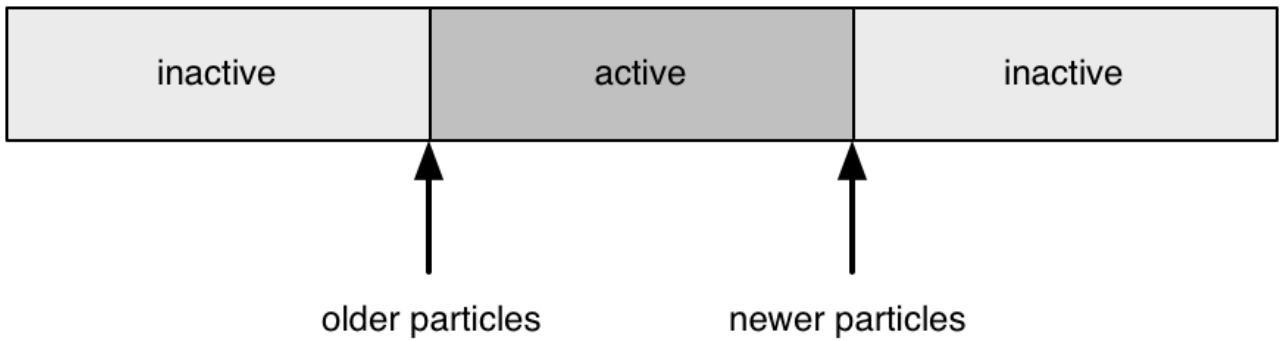
The constructor `ParticleSystem`:

- Intialises all class variables
- Creates vertex and index buffers
- Loads the effect
- Calls `generateParticles()`

The method `generateParticles()` creates a quad using `ParticleVertex`:

- Creates `nParticles` $\times$ 4 vertices (for `nParticles` particles)
- All values (e.g., position, speed) are initialised to 0

The next method to look at is `AddParticle()`. We constantly need to keep track of our particles and update them accordingly. For this we use a **circular queue**.



AddParticle()

```
1  public void AddParticle(Vector3 Position, Vector3 Direction, float
       Speed) {
2    if (nActive + 4 == nParticles * 4)
3      return;
4
5    int index = offsetIndex(activeStart, nActive);
6
7    nActive += 4;
8
9    float startTime = (float)(DateTime.Now - start).TotalSeconds;
10
11   for (int i = 0; i < 4; i++)
12   {
13     particles[index + i].StartPosition = Position;
14     particles[index + i].Direction = Direction;
15     particles[index + i].Speed = Speed;
16     particles[index + i].StartTime = startTime;
17   }
18 }
```

```
1  int offsetIndex(int start, int count) {
2    for (int i = 0; i < count; i++)
3    {
4      start++;
5
6      if (start == particles.Length)
7        start = 0;
8    }
9
10   return start;
11 }
```

Update()

The update method goes throigh all particles, checking which ones are active and keeps the list of particles up to date:

```
1  public void Update()
2  {
3    float now = (float)(DateTime.Now - start).TotalSeconds;
4    int startIndex = activeStart;
5    int end = nActive;
6
7    for (int i = 0; i < end; i++)
8    {
9      if (particles[activeStart].StartTime < now - lifespan)
10     {
11       activeStart++;
12       nActive--;
13
14       if (activeStart == particles.Length)
15         activeStart = 0;
16     }
17   }
18
19   verts.SetData<ParticleVertex>(particles);
20   ints.SetData<int>(indices);
21 }
```

Next is the draw method which assignes the buffers, sets all effect parameters, changes the render states and draws the particles.

Draw()

```
1  public void Draw(Matrix View, Matrix Projection, Vector3 Up, Vector3
       Right)
2  {
3    graphicsDevice.SetVertexBuffer(verts);
4    graphicsDevice.Indices = ints;
5
6    effect.Parameters["ParticleTexture"].SetValue(texture);
7    effect.Parameters["View"].SetValue(View);
8    effect.Parameters["Projection"].SetValue(Projection);
9    effect.Parameters["Time"].SetValue((float)(DateTime.Now - start).
         TotalSeconds);
10   effect.Parameters["Lifespan"].SetValue(lifespan);
11   effect.Parameters["Wind"].SetValue(wind);
12   effect.Parameters["Size"].SetValue(particleSize / 2f);
13   effect.Parameters["Up"].SetValue(Up);
14   effect.Parameters["Side"].SetValue(Right);
15   effect.Parameters["FadeInTime"].SetValue(fadeInTime);
16
17   graphicsDevice.BlendState = BlendState.AlphaBlend;
18   graphicsDevice.DepthStencilState = DepthStencilState.DepthRead;
19
20   effect.CurrentTechnique.Passes[0].Apply();
21
22   graphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0,
         0, nParticles * 4, 0, nParticles * 2);
23
24   graphicsDevice.SetVertexBuffer(null);
25   graphicsDevice.Indices = null;
26
27   graphicsDevice.BlendState = BlendState.Opaque;
28   graphicsDevice.DepthStencilState = DepthStencilState.Default;
29 }
```

ParticleEffect I

```
1  float4x4 View, Projection;
2  float Time, Lifespan, FadeInTime;
3  float2 Size;
4  float3 Wind, Up, Side;
5  texture ParticleTexture;
6
7  sampler2D texSampler = sampler_state {
8    texture = <ParticleTexture>; };
9
10 struct VertexShaderInput {
11   float4 Position : POSITION0;
12   float2 UV : TEXCOORD0;
13   float3 Direction : TEXCOORD1;
14   float Speed : TEXCOORD2;
15   float StartTime : TEXCOORD3; };
16
17 struct VertexShaderOutput {
18   float4 Position : POSITION0;
19   float2 UV : TEXCOORD0;
20   float2 RelativeTime : TEXCOORD1; };
21
22 VertexShaderOutput VertexShaderFunction(VertexShaderInput input) {
23   VertexShaderOutput output;
24   float3 position = input.Position;
25
26   // Move to billboard corner
27   float2 offset = Size * float2((input.UV.x - 0.5f) * 2.0f,-(input.UV.
         y - 0.5f) * 2.0f);
28   position += offset.x * Side + offset.y * Up;
29
```

ParticleEffect II

```
30    // Determine how long this particle has been alive
31    float relativeTime = (Time - input.StartTime);
32    output.RelativeTime = relativeTime;
33
34    // Move vertex along movement and wind direction
35    position += (input.Direction * input.Speed + Wind) * relativeTime;
36
37    // Transform the final position by the view and projection matrices
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    // Ignore particles that aren't active
46    clip(input.RelativeTime);
47
48    // Sample texture
49    float4 color = tex2D(texSampler, input.UV);
50
51    // Fade out towards end of life
52    float d = clamp(1.0f-pow((input.RelativeTime / Lifespan),10),0,1);
53
54    // Fade in at beginning of life
55    d *= clamp((input.RelativeTime / FadeInTime), 0, 1);
56
57    // Return color * fade amount
58    return float4(color * d);
59 }
```

ParticleEffect III

```
60
61 technique Technique1
62 {
63   pass Pass1
64   {
65     VertexShader = compile vs_2_0 VertexShaderFunction();
66     PixelShader = compile ps_2_0 PixelShaderFunction();
67   }
68 }
```

The effect is quite long but mostly standard. The interesting bits are, of course, the vertex and pixel shaders. They are extensions of the shaders we implemented in the last lecture.

Vertex shader:

- Unforld / Rotate billboard
- Move billboard to its position

Pixel shader:

- Ignore inactive particles
- Fade billboard in and out

Also note the relationship between the vertex declaration and the semantics used in the effect.

Particles for Fire and Smoke

It is now time to demonstrate the code. We will place some rusty barrels in our 3D world for the flames and smoke to come out of:

- Create hollow cylinder in SketchUp
- Apply a rusty texture inside and out
- Make sure to turn off culling when drawing the barells

The textures we will use are:

Particles

```
ParticleSystem fire, smoke;
Random rnd = new Random(0);
```

```
float fireHeight = 0.8f;
float smokeHeight = 0.8f;

fire = new ParticleSystem(device, content, content.Load<Texture2D>("
    textures/lightFire"), 20, new Vector2(fireHeight), 1, Vector3.
    Zero, 0.5f);
smoke = new ParticleSystem(device, content, content.Load<Texture2D>("
    textures/lightSmoke"), 20, new Vector2(smokeHeight), 1, new
    Vector3(1, 0.3f, 0), 0.2f);
```

```
private void UpdateParticlesFire(GameTime gameTime)
{
  Vector3 offset = new Vector3(MathHelper.ToRadians(20.0f));
  Vector3 randAngle = Vector3.Up + randVec3(-offset, offset);
  Vector3 randPosition = Vector3.Zero;

  randPosition.X = location.column + 0.5f;
  randPosition.Y = 0.3f;
  randPosition.Z = location.row + 0.5f;

  float randSpeed = (float)rnd.NextDouble() + 0.2f;
  fire.AddParticle(randPosition, randAngle, randSpeed);
  fire.Update();
}
```

`randVec3()` is a helper method and `UpdateParticlesSmoke()` is very similar.

Notes about Particles

Similar to standard billboards, choosing the right textures is essential:

- Realism of particles depends on textures
- For fire and smoke, "light" textures tend to look better

Equally important are the parameters chosen for speed, direction, "wind" etc. These usually need to be established using trial and error. In our case, we chose fixed starting positions and varried the speed only. Choosing random positions can create a more realistic effect.

There are other tricks to make your effects more impressive and immersive:

- Using appropriate sounds adds an additional layer of immersion
- Mixing different types of billboards / particles (e.g., fire and smoke)

Outline

Terrain

Almost all video games make use of terrains. This could be terrain that can be actively explored (e.g., first person shooter) or terrain in the background to create the illusion of open-ended worlds (e.g., racing game).

Terrain can be generated automatically on the fly to create truly open-ended worlds. Alternatively, another common usage of mountain ranges is to limit the world available to the gamer. This is what we will be doing.

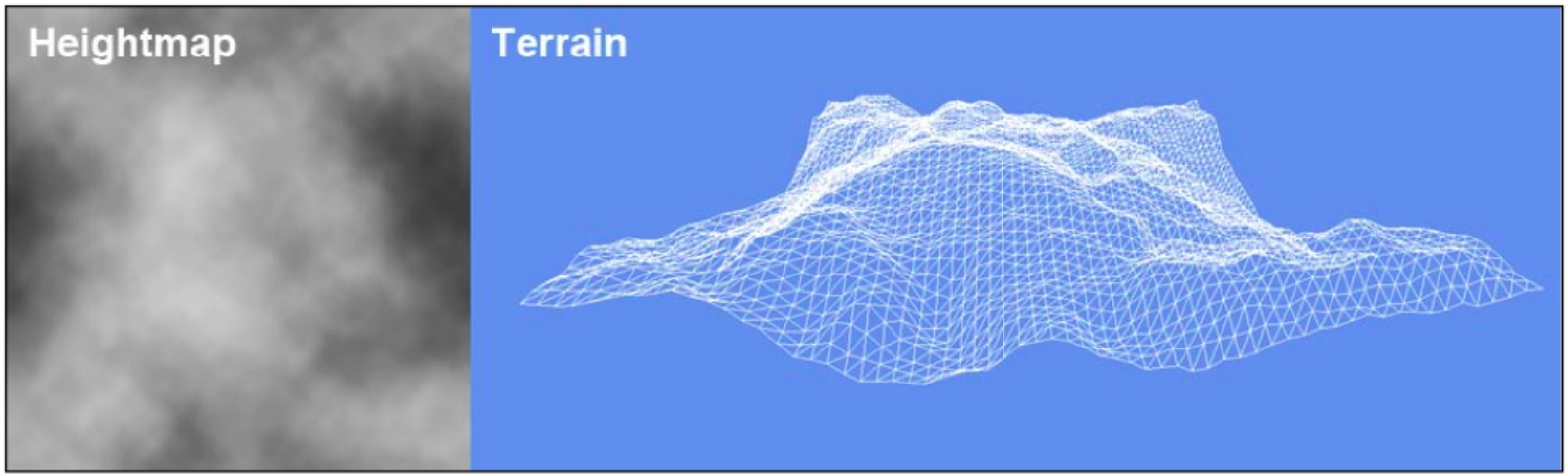
Note: remember that distant elements will sudently come into view given the far plane of the viewing furstum. One can use elements like fog to ensure smooth transitions without the need to raw too many items in the distance.

Terrain

A common way to generate terrain is to built a mesh (using triangles) from a **heightmap** and to apply a texture to that mesh.

A heightmap is a 2D image in gray scale:

- Each pixel corresponds to a 'tile'
- The colour value of the pixel determines its height

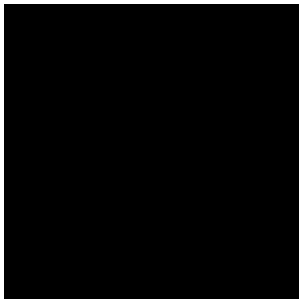


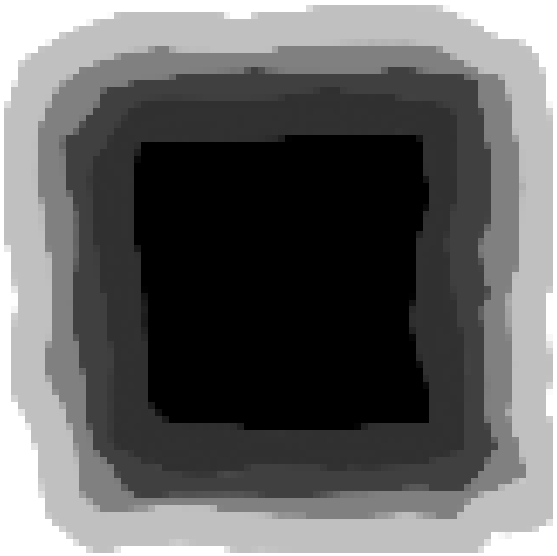
A heightmap may be generated using special tools to simply using an image editor such as Photoshop to paint.NET.

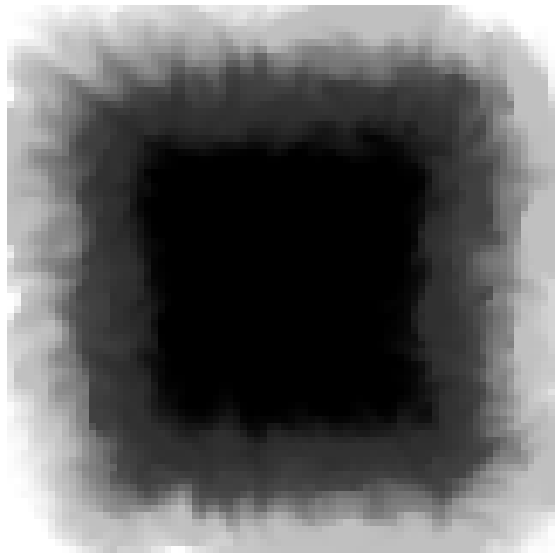
Creating the Heightmap

We will create the heightmap from scratch, using Paint.NET:

1. Create a new image, 80×80
2. Create a filled black square at the centre (20 pixels from the edge)
3. Using the brush, draw bands around center using ligher shades of gray
4. Use the smudge tool (plugin required) to soften the transitions

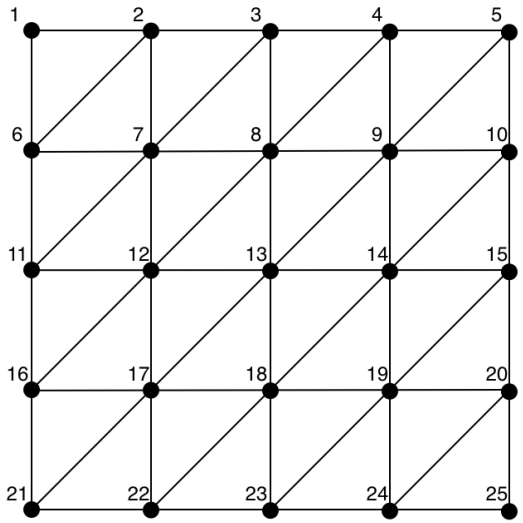






From Heightmap to Mesh

The heightmap contains the elevation information required to render the terrain. We need to make use of this information to create a mesh:



The mesh is the terrain we will be using (we will apply a texture to the mesh to make it look realistic). We can choose different cell sizes to balance computational complexity and looks. Likewise, we can tile the applied texture(s) to different degrees.

Terrain

We create a class `Terrain` with the following attributes:

```
1  class Terrain
2  {
3    VertexPositionNormalTexture[] vertices;
4    VertexBuffer vertexBuffer;
5    int[] indices;
6    IndexBuffer indexBuffer;
7
8    float[,] heights;
9    float height, cellSize, textureTiling;
10   int width, length, nVertices, nIndices;
11
12   Effect effect;
13   GraphicsDevice GraphicsDevice;
14   Texture2D heightMap, baseTexture;
15   Vector3 lightDirection;
16 }
```

- Constructor initialises / computes most of these variables
  - `nVertices = width * length;` (i.e., one vertex per pixel)
  - `nIndices = (width - 1)* (length - 1)* 6;` (**Q** why?)
- Constructor sets the index and vertex buffers
- The constructor also calls 4 helper methods:

  `getHeights()`, `createVertices()`, `createIndices()`, `genNormals()`

getHeights()

```
1  private void getHeights()
2  {
3    // Extract pixel data
4    Color[] heightMapData = new Color[width * length];
5    heightMap.GetData<Color>(heightMapData);
6
7    // Create heights[,] array
8    heights = new float[width, length];
9
10   // For each pixel
11   for (int y = 0; y < length; y++)
12     for (int x = 0; x < width; x++)
13     {
14       // Get color value (0 - 255)
15       float amt = heightMapData[y * width + x].R;
16
17       // Scale to (0 - 1)
18       amt /= 255.0f;
19
20       // Multiply by max height to get final height
21       heights[x, y] = amt * height;
22     }
23 }
```

`.R` returns the red component of the colour. **Q** Why do we just use that?

createVertices()

```
1  private void createVertices()
2  {
3    vertices = new VertexPositionNormalTexture[nVertices];
4
5    Vector3 offsetToCenter = -new Vector3(((float)width / 2.0f) *
         cellSize, 0, ((float)length / 2.0f) * cellSize);
6
7    // For each pixel in the image
8    for (int z = 0; z < length; z++)
9      for (int x = 0; x < width; x++)
10     {
11       // Find position based on grid coordinates and height in
             heightmap
12       Vector3 position = new Vector3(x * cellSize, heights[x, z], z *
             cellSize) + offsetToCenter;
13
14       // UV coordinates range from (0, 0) at grid location (0, 0) to
             (1, 1) at grid location (width, length)
15       Vector2 uv = new Vector2((float)x / width, (float)z / length);
16
17       // Create the vertex
18       vertices[z * width + x] = new VertexPositionNormalTexture(
             position, Vector3.Zero, uv);
19     }
20 }
```

Might need to use offsets to shift terrain along X and Z and possibly also Y.

Q Where would you add these offsets?

createIndices()

```
1  private void createIndices()
2  {
3    indices = new int[nIndices];
4    int i = 0;
5
6    // For each cell
7    for (int x = 0; x < width - 1; x++)
8      for (int z = 0; z < length - 1; z++)
9      {
10       // Find the indices of the corners
11       int upperLeft = z * width + x;
12       int upperRight = upperLeft + 1;
13       int lowerLeft = upperLeft + width;
14       int lowerRight = lowerLeft + 1;
15
16       // Specify upper triangle
17       indices[i++] = upperLeft;
18       indices[i++] = upperRight;
19       indices[i++] = lowerLeft;
20
21       // Specify lower triangle
22       indices[i++] = lowerLeft;
23       indices[i++] = upperRight;
24       indices[i++] = lowerRight;
25     }
26 }
```

We need to create each cell using two clock-wise triangles.

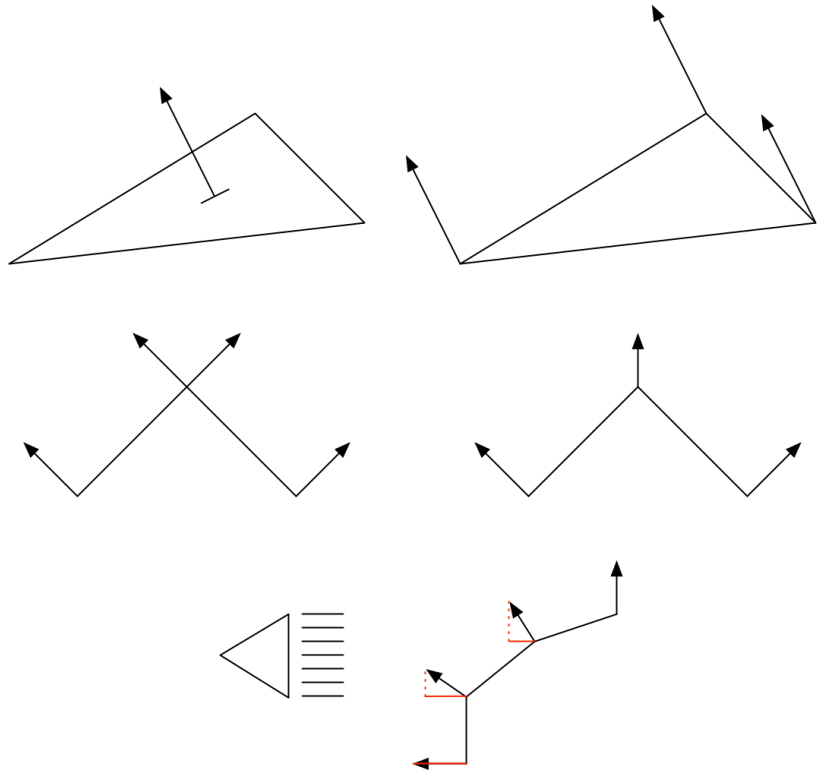
Normals

To calculate the impact of the light we are using, we need to know the normals of the terrain. Normals are essential for almost all lighting calculations.

There are two types of normals:

- Surface normal
- Vertex normal

In real-time 3D graphics, vertex normals are used. Note: normals indicate direction only, so their length should be 1.



genNormals()

```
1  private void genNormals()
2  {
3    // For each triangle
4    for (int i = 0; i < nIndices; i += 3)
5    {
6      // Find the position of each corner of the triangle
7      Vector3 v1 = vertices[indices[i]].Position;
8      Vector3 v2 = vertices[indices[i + 1]].Position;
9      Vector3 v3 = vertices[indices[i + 2]].Position;
10
11     // Cross the vectors between the corners to get the normal
12     Vector3 normal = Vector3.Cross(v1 - v2, v1 - v3);
13     normal.Normalize();
14
15     // Add the influence of the normal to each vertex in the triangle
16     vertices[indices[i]].Normal += normal;
17     vertices[indices[i + 1]].Normal += normal;
18     vertices[indices[i + 2]].Normal += normal;
19   }
20
21   // Average the influences of the triangles touching each vertex
22   for (int i = 0; i < nVertices; i++)
23     vertices[i].Normal.Normalize();
24 }
```

We calculate the normal for each triangle of the terrain and add the normals to their respective vertices. We then normalise the normals to average the different influences (i.e., gradients).

TerrainEffect I

```
1  float4x4 View;
2  float4x4 Projection;
3  float3 LightDirection = float3(1, -1, 0);
4  float TextureTiling = 1;
5  texture2D BaseTexture;
6
7  sampler2D BaseTextureSampler = sampler_state
8  {
9    Texture = <BaseTexture>;
10   AddressU = Wrap;
11   AddressV = Wrap;
12   MinFilter = Anisotropic;
13   MagFilter = Anisotropic;
14 };
15
16 struct VertexShaderInput
17 {
18   float4 Position : POSITION0;
19   float2 UV : TEXCOORD0;
20   float3 Normal : NORMAL0;
21 };
22
23 struct VertexShaderOutput
24 {
25   float4 Position : POSITION0;
26   float2 UV : TEXCOORD0;
27   float3 Normal : TEXCOORD1;
28 };
29
30
```

TerrainEffect II

```
31 VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
32 {
33   VertexShaderOutput output;
34
35   output.Position = mul(input.Position, mul(View, Projection));
36   output.Normal = input.Normal;
37   output.UV = input.UV;
38
39   return output;
40 }
41
42 float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
43 {
44   float light = dot(normalize(input.Normal),
45                     normalize(LightDirection));
46   light = clamp(light + 0.4f, 0, 1); // Simple ambient lighting
47   float3 tex = tex2D(BaseTextureSampler, input.UV * TextureTiling);
48
49   return float4(tex * light, 1);
50 }
51
52 technique Technique1
53 {
54   pass Pass1
55   {
56     VertexShader = compile vs_2_0 VertexShaderFunction();
57     PixelShader = compile ps_2_0 PixelShaderFunction();
58   }
59 }
```

Drawing the Terrain

That is it, we can now implement the draw method and display the terrain.

Demo.

There are numerous improvements that can be made to create more realistic looking terrains:

- Multitexturing: use an additional map with red, green and blue channels to indicate different textures in the terrain (e.g., snow, sand)
- Detail textures: use noise textures that fad in when the camera is close to fake high resolution textures
- Place plants (especially grass) on the terrain at random positions etc.

You should consult chapter 7 of S. James for more information.

To place plants on the terrain and also to perform collision detection, we need to know the height of the terrain given some X and Z coordinates.

Collision Detection

```
1  public float GetHeightAtPosition(float X, float Z)
2  {
3    // Reset the offset applied earlier (map from 0, 0 to Width, Length)
4    X += (width / 2f) * cellSize;
5    Z += (length / 2f) * cellSize;
6
7    // Map to cell coordinates
8    X /= cellSize;
9    Z /= cellSize;
10
11   // Truncate coordinates to get coordinates of top left cell vertex
12   int x1 = (int)X;
13   int z1 = (int)Z;
14
15   // Try to get coordinates of bottom right cell vertex
16   int x2 = x1 + 1 == width ? x1 : x1 + 1;
17   int z2 = z1 + 1 == length ? z1 : z1 + 1;
18
19   // Get the heights at the two corners of the cell
20   float h1 = heights[x1, z1];
21   float h2 = heights[x2, z2];
22
23   // Find the average loss due to truncation above
24   float leftOver = ((X - x1) + (Z - z1)) / 2f;
25
26   // Interpolate between the corner vertices' heights
27   return MathHelper.Lerp(h1, h2, leftOver);
28 }
```

If we used offsets for the terrain earlier, we need to apply them here also.

Outline

Review to the Game

We have now finished our game. We have used the following techniques:

- Level generation from text file using custom importers and processors
- Use of custom models for game entities
- Collision detection using bounding spheres and bounding boxes
- Behaviour trees for NPC AI
- Pathfinding algorithms: Dijstra and A*
- Interactivity: shooting and capture the flag
- Billboarding:
  - Spherical billboarding for clouds
  - Non-rotating billboarding for trees
  - Cylindrical billboarding for particles (fire and smoke)
- Terrain generation from heightmap
- Skydome
- Background music and sound effects

Improvements to the Game

Although the 3D world looks fairly good, there are many ways to improve it.

These include:

- Improved collision detection with walls (player should *glide* along walls)
- Efficiency of computations: implement collision detection more efficiently
- User control via game pad
- Better looking models and textures
- Crosshair for aiming
- Lighting
- Explosions and richer interactivity
- Multiple different enemies
- Splash screen
- Graphics for score, health, etc.

You should attempt to consider some of these extensions for your assignment.

Outline

Summary & Lab Preview

In this lecture we covered particles (billboards) and terrains (heighmaps) to further visually enhance our 3D world.

We covered

- Particles (fire and smoke)
- Terrains from heightmaps

The latest code (including billboards, terrains, particles, etc.) can now be downloaded from the course web site. You should spend some time looking through the different implementations.

The last two labs of the course you should be working on your assignments. This is the last chance you will have to get help so it is vital you attend the labs to write the code for the foundation of your game.