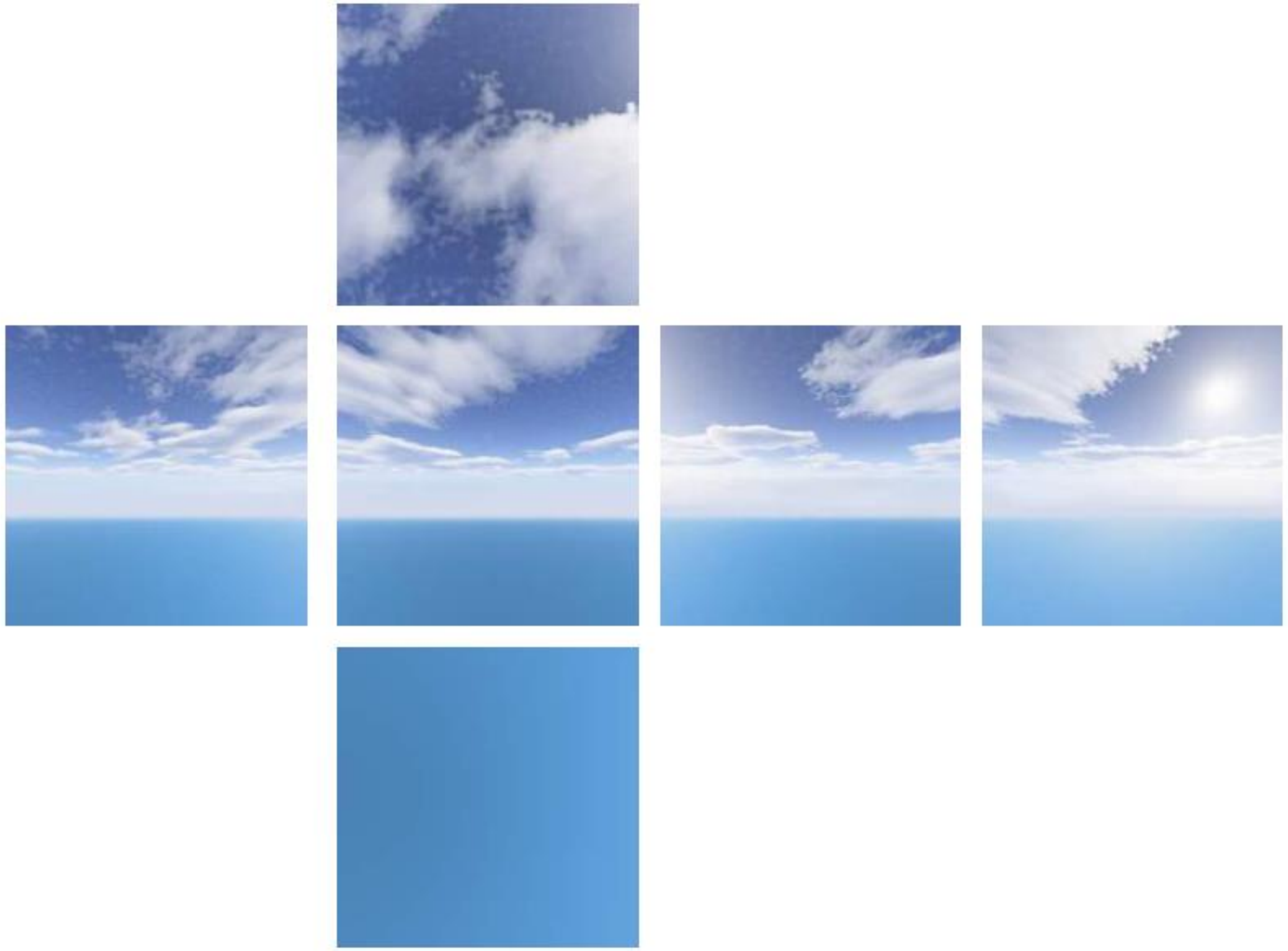
Sky Domes, Fog, and Getting the Height of Terrain

Wayne Rippin (UoD)
Dr Panagiotis Perakis (MC)

Sky Domes

- One thing that is missing in our scene is realistic sky
- We can add that by creating a node that handles a sky box
- This makes use of a TextureCube
- A TextureCube is a textured cube where the texture is on the inside of each face of the cube

A Sky Cube



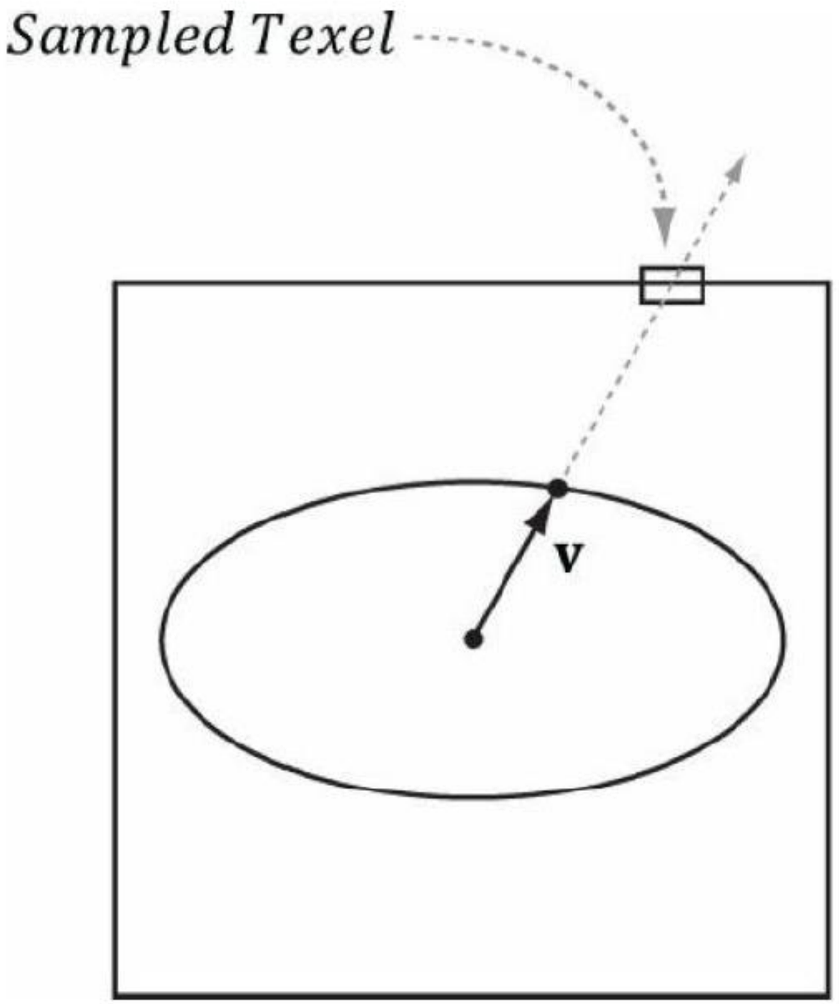
Creating a Sky Cube

- If you have the six images that make up a sky cube, you can use the texassembletool that comes with the DirectX Toolkit to create the sky cube dds file.
- https://github.com/Microsoft/DirectXTex/wiki/Texassemble

Sky Domes

- To use a sky cube, we need to:
  - Create a model of a sphere that surrounds the entire scene
  - We assume that the surface of the sphere is infinitely far away so that we never actually reach it
  - To do this, we centre the sphere in world space at the position of the camera and update its position as the camera moves
  - Then we use the vector from the origin to the point on the surface of the sphere as the coordinates into the cube map to give us the pixel value we want

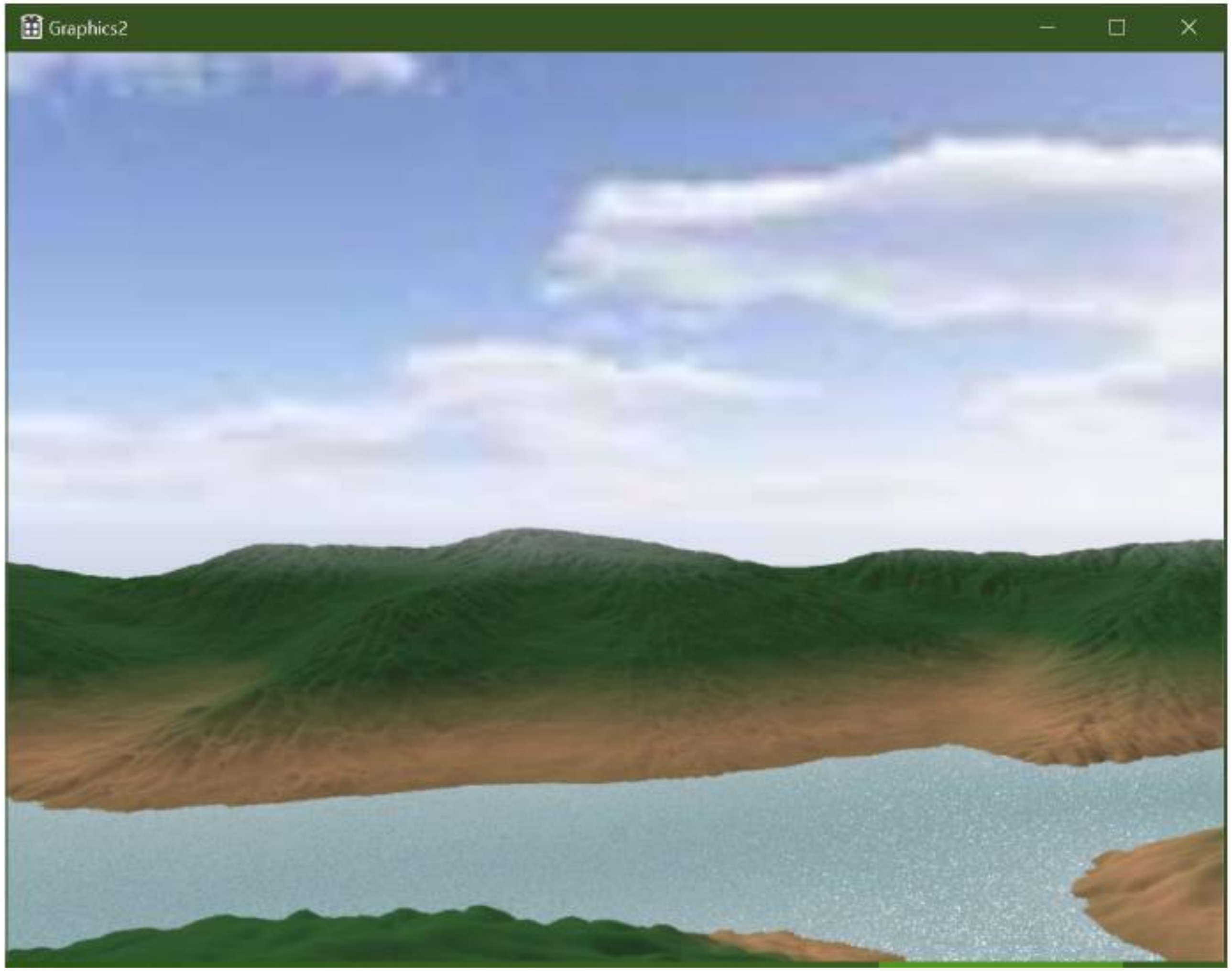
Sky Boxes



Sky Boxes

- When rendering the sky box, there are a few key points we need to remember:
  - We need to use the camera position to create the world transformation matrix for the sky box each time it is rendered
    - We then transpose the complete world-view-projection matrix as we want the inverse of the complete transformation
  - We need to turn off back face culling
    - This is because the camera is inside the sky box, so we would not normally see the texture. Turning off back-face culling fixes this.
  - We need to change the function for the depth buffer calculations to be LESS_EQUAL rather than the default value of LESS.
    - If this is so the sky is always rendered at the farthest point and everything else is rendered in front of it.

Rendering the Sky Box



Tutorial Materials

- The first part of the tutorial this week includes the steps you need to go through to create a sky node.   This includes the additional code you will need, including the shader.

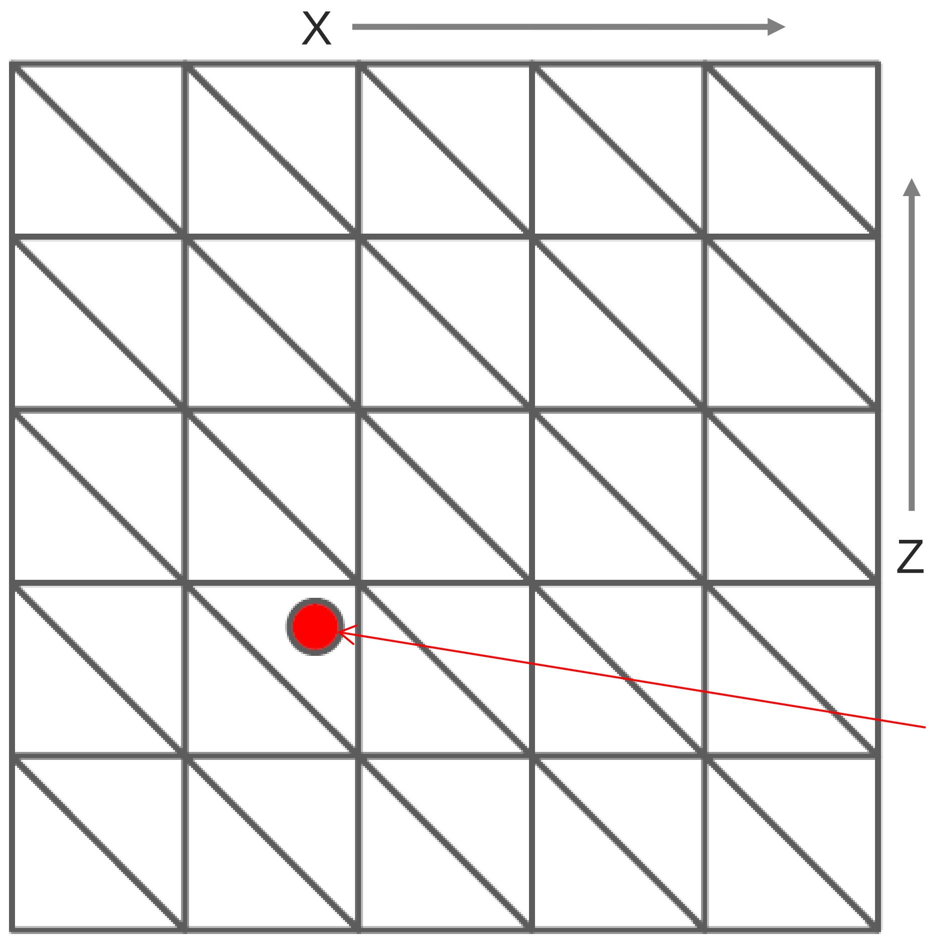
Avoiding Collision with and Placing Items on Terrain

- In order to place an object on terrain or to detect collision with terrain, we need to be able to determine the height at that point, i.e. given a particular coordinate (x, z), you need to be able to determine the y value at that coordinate.

Placing Items on Terrain

- Remember that our terrain is made up of squares that have the same width and height
- Each square is divided up into two triangles
- Dealing with terrain that is made up of irregularly sized squares is beyond the scope of this module

Placing Items on Terrain



What's the y value at this point?

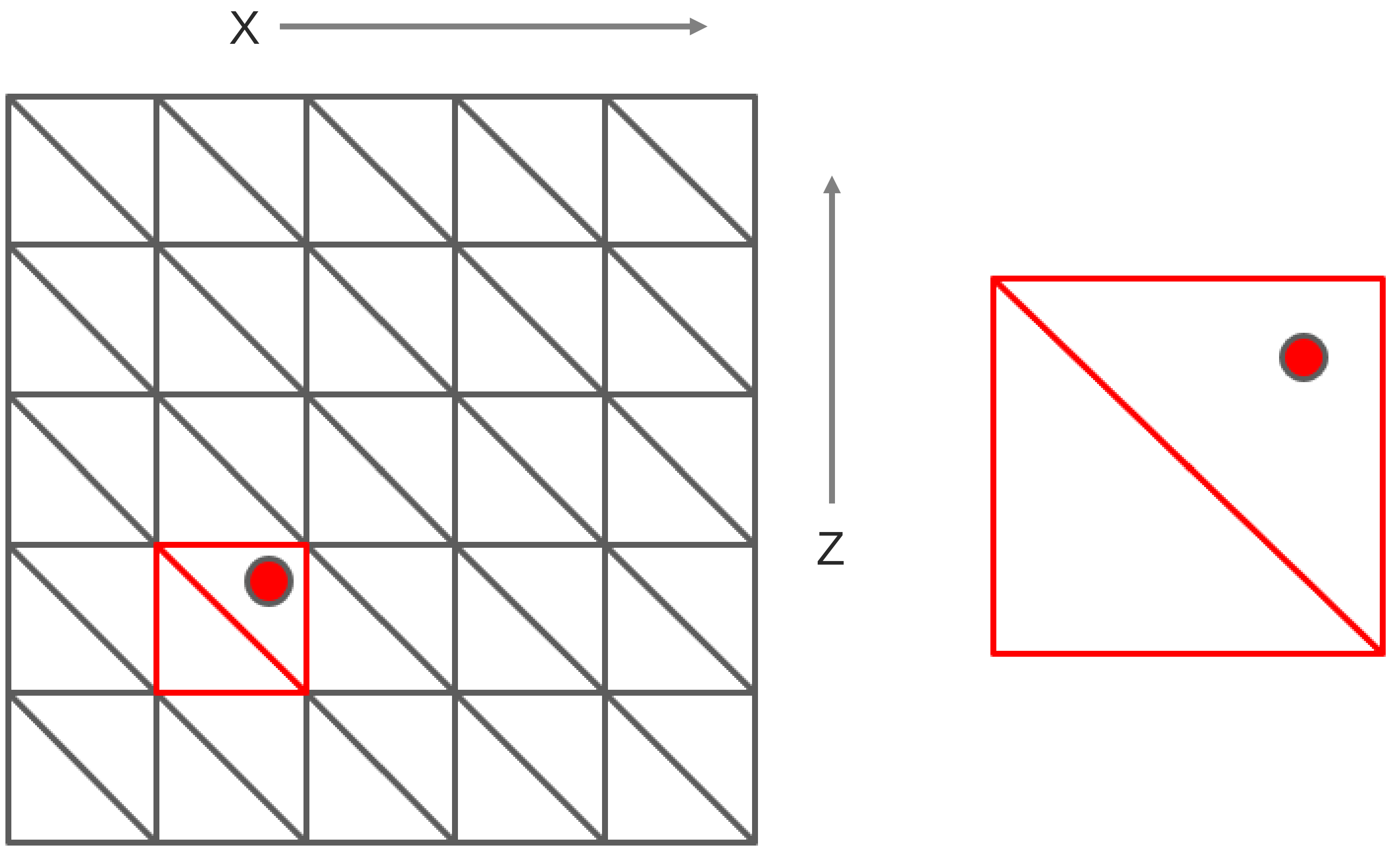
Placing Items on Terrain

- The first thing we need to calculate is which cell in the grid contains the x, z value we are interested in
- If we maintain the start x and z values of the terrain and the width/height of a cell, then:

```
intcellX= (int)((x -_terrainStartX) / _spacing);
intcellZ= (int)((_terrainStartZ-z) / _spacing);
```

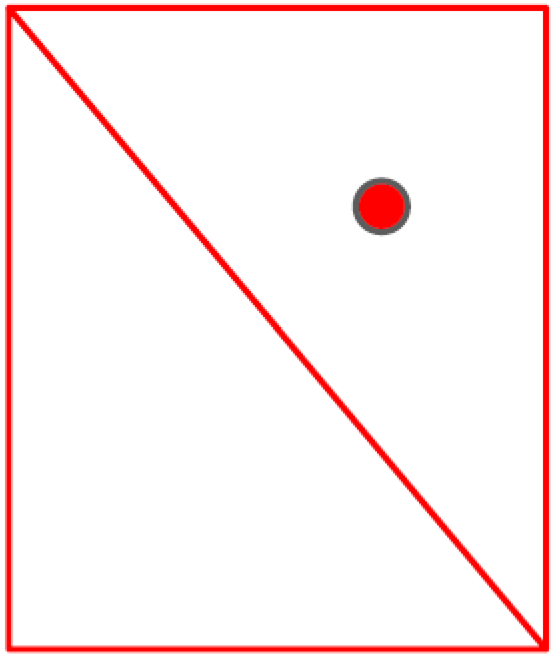
Different order because the z value starts positive and decreases to negative values

Placing Items on Terrain

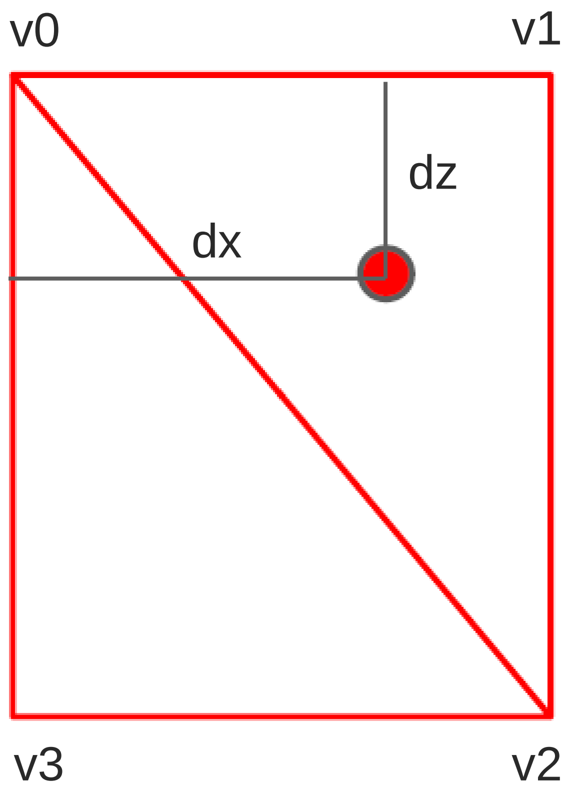


Placing Items on Terrain

- Once we have identified the cell the point x, z is in, we now need to identify which of the triangles we are interested in



Placing Items on Terrain



If dz is the difference in the z value between the point and the edge of the grid cell and dx is the difference in the x value between the point and the edge of the grid cell, then:

If dx > dz

  We are in triangle with vertices v0, v1, v2

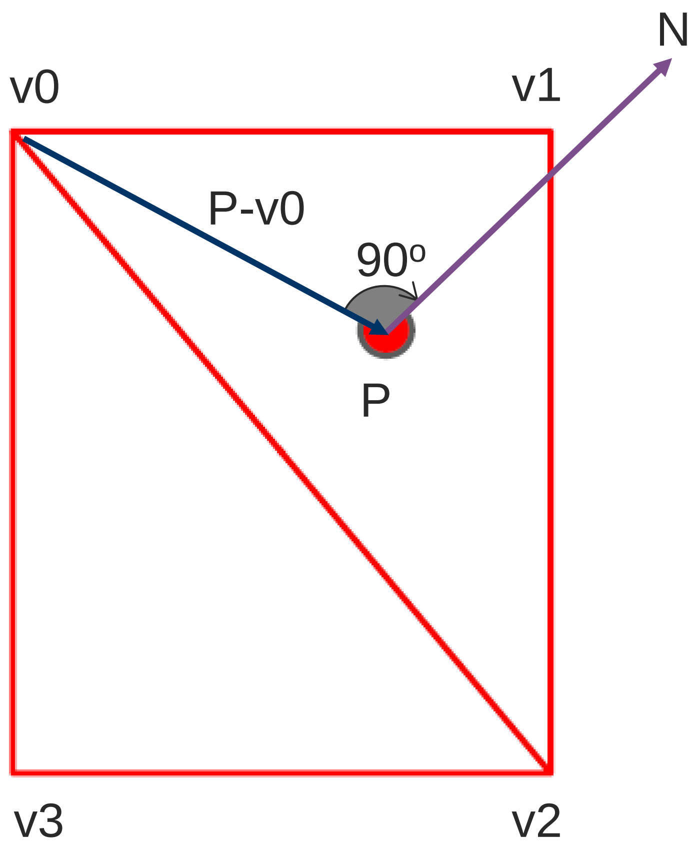
Else

  We are in triangle with vertices v0, v2, v3

Placing Items on Terrain

- Once we have identified which triangle contains the point x, z, we now need to identify the y value at that point
- To do this, we make use of the fact that the dot product of a vector on a plane and the planes' normal vector gives a value of 0 (i.e. they are perpendicular)

Placing Items on Terrain



$$N \cdot (P - v0) = 0$$

Where:

N is the normal vector for the triangle

P is the point we are interested in

v0 is one of the vertices of the triangle

Placing Items on Terrain

- Expanding out the dot product equation, we get:

$$N_x(P_x - v0_x) + N_y\left(P_y - v0_y\right) + N_z(P_z - v0_z) = 0$$

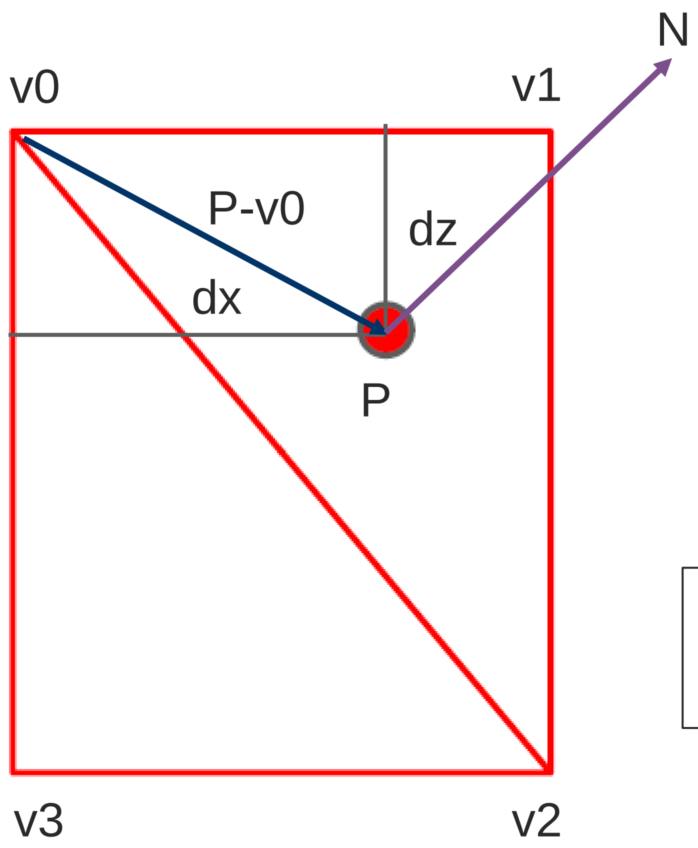
- We extract the y components by moving them to the other side and dividing by N_y

$$P_y - v0_y = [N_x(P_x - v0_x) + N_z(P_z - v0_z)]/{-N_y}$$

- We can now rearrange so that we have P_y on its own:

$$\boxed{P_y = v0_y + [N_x(P_x - v0_x) + N_z(P_z - v0_z)]/{-N_y}}$$

Placing Items on Terrain



In terms of our triangle,

$dx = P_x – v0_x$ and

$dz = P_z – v0_z$,

so, we can calculate the height of the point using the equation:

$$\boxed{P_y = v0_y + \frac{N_x dx + N_z dz}{-N_y}}$$

Placing Items on Terrain

- This will give you the height of the terrain at the x, z value you require so you can now position an item at that point –remember to adjust your position relative to where the model origin for your item is

Fog

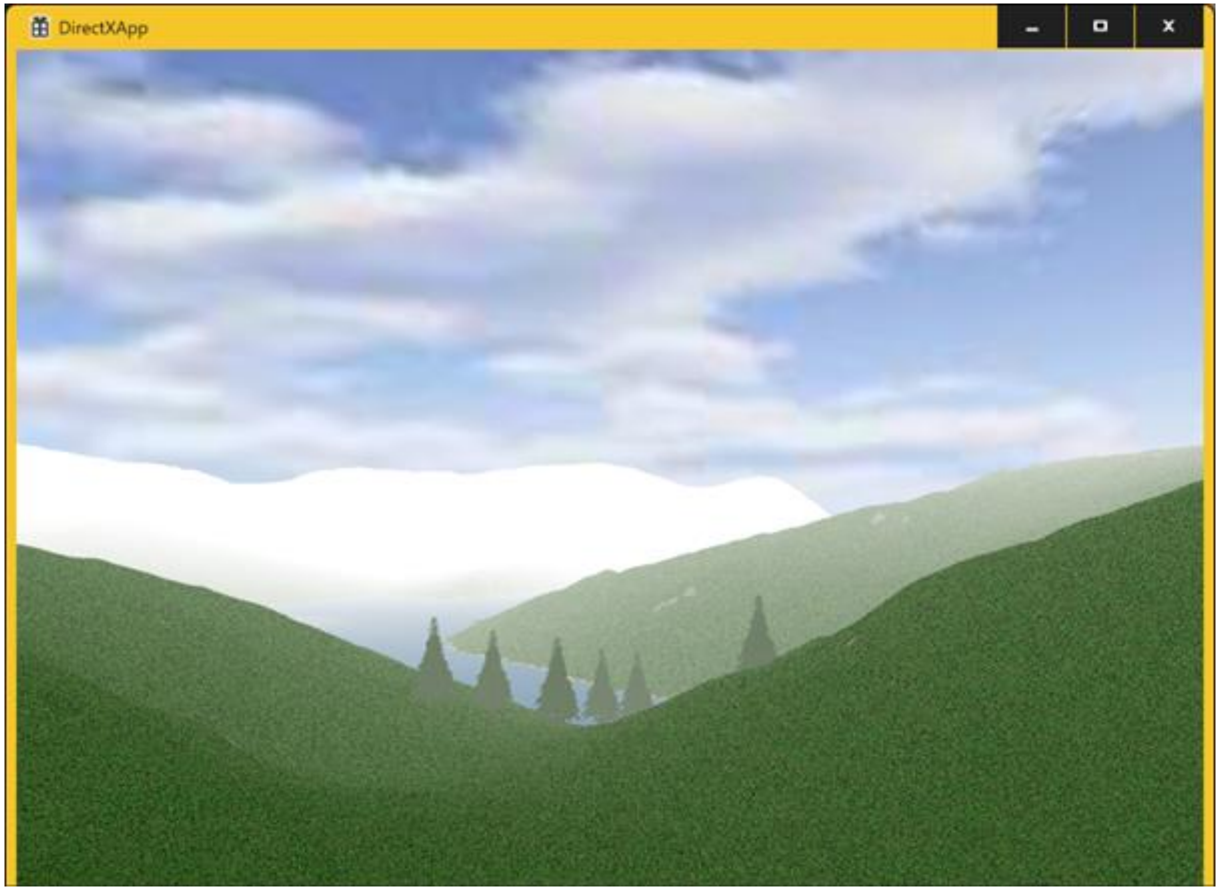
- Simulating a fog effect can be very useful beyond simulating certain weather:
- For example, it can mask distant rendering artifacts and prevent *popping*. Popping refers to when an object that was previously behind the far plane all of a sudden comes in front of the frustum, due to camera movement, and thus becomes visible; so, it seems to “pop” into the scene abruptly. By having a layer of fog in the distance, the popping is hidden.
- Even if your scene takes place on a clear day, you may wish to still include a subtle amount of fog at far distances, because, even on clear days, distant objects such as mountains appear hazier and lose contrast as a function of depth, and we can use fog to simulate this.

Fog

- To simulate fox, we need three values:
  - A fog colour
  - A fog start distance from the camera
  - A fog range – the distance from the start of the fog to the point where the fog completely hides any objects
- The colour of any pixel can be adjusted using:

```
float fogLerp = saturate((distanceToEye - fogStart) /
    fogRange);
colour = lerp(colour, fogColour, fogLerp);
```

Example



This Week

- Create a SkyNode to render a sky dome
- Add a method to your terrain node to calculate the height at a particular point
- Adding fog to the scene.

Next Week

- Collision detection between objects