Terrain

305890
Spring 2010
6/4/2010
Kyoung Shin Park

Overview

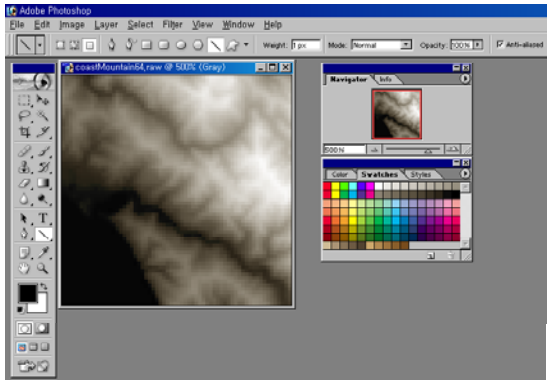
- To learn how to *generate height info* for a terrain that results in smooth transitions between mountains and valleys
- To find out how to *light and texture* the terrain
- To learn how to implement a *camera* whose interface provides control suited for first-person perspective games
- To discover a way to keep the camera planted on the terrain so that walking or running on the terrain is simulated

Heightmap

- We use a heightmap to describe the hills and valleys of our terrain
  - A heightmap is a matrix in which each element specifies the **height** of a particular vertex in the *terrain grid*.
  - The *ij*th heightmap entry provides the height for the *ij*th vertex.
  - Height can range from 0 to 255, and it can be scaled with this range value.
  - Gray-scale map (0: smallest height, 255: largest height)

Creating a Heightmap

- Using an image editor (such as, Adobe Photoshop)
  - Need to save it as a gray-scale image, 8-bit RAW file (with no header)



Heightmap Class

```
class Heightmap {
public:
    Heightmap();
    Heightmap(int m, int n);
    Heightmap(int m, int n, const string& filename, float heightScale, float heightOffset);

    void recreate(int m, int n);
    void loadRAW(int m, int n, const string& filename, float heightScale, float heightOffset);

    int numRows() const;
    int numCols() const;

    float& operator() (int i, int j);          // returns the ijth heightmap element
    const float& operator() (int i, int j) const;

private:
    bool inBounds(int i, int j);
    float sampleHeight3x3(int i, int j);
    void filter3x3();

    string mHeightMapFilename;
    Table<float> mHeightMap;
    float mHeightScale;
    float mHeightOffset;
}
```

Loading RAW Heightmap File

- RAW file is nothing more than a contiguous block of bytes (where each byte is a heightmap entry)
  - We read in the block of memory with std::ifstream::read call
  - We simply creates the vertices for terrain mesh by # of bytes

Loading RAW Heightmap File

```
void Heightmap::loadRAW(int m, int n, const string& filename, float heightScale, float height
    Offset) {
    mHeightMapFilename = filename;
    mHeightScale       = heightScale;
    mHeightOffset      = heightOffset;
    std::vector<unsigned char> in( m * n ); // A height for each vertex
    // Open the file.
    std::ifstream inFile;
    inFile.open(filename.c_str(), ios_base::binary);
    if(!inFile) HR(E_FAIL);
    // Read the RAW bytes.
    inFile.read((char*)&in[0], (streamsize)in.size());
    // Done with file.
    inFile.close();
    // Copy the array data into a float table format and scale the heights.
    mHeightMap.resize(m, n, 0);
    for(int i = 0; i < m; ++i) {
        for(int j = 0; j < n; ++j) {
            int k = i * n + j;
            mHeightMap(i, j) = (float)in[k] * heightScale + heightOffset;
        }
    }
    // Filter the table to smooth it out.  We do this because 256 height steps is rather course.
    // And now that we copied the data into a float-table, we have more precision.
    // So we can smooth things out a bit by filtering the heights.
    filter3x3();
}
```

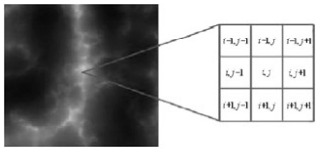
Filtering

- Problem of using an 8-bit heightmap: it means we can only represent ***256 discrete height steps***.
  - Terrain may be rough because there is a large step size from one height level to the next.
  - These step sizes are magnified if the terrain is scaled.
- Solution: we load the heightmap into memory and use ***floats*** to represent each height element.
  - Then, we ***apply filter*** to the heightmap, which *smoothes out the heightmap,* making the difference in heights between adjacent elements less drastic.

Filtering

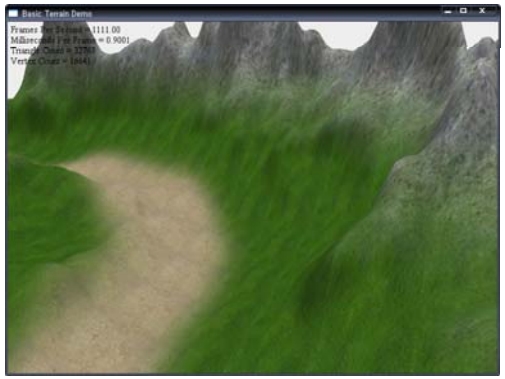
```
void Heightmap::filter3x3() {
    Table<float> temp(mHeightMap.numRows(), mHeightMap.numCols());

    for(int i = 0; i < mHeightMap.numRows(); ++i)
        for(int j = 0; j < mHeightMap.numCols(); ++j)
            temp(i,j) = sampleHeight3x3(i,j);

    mHeightMap = temp;
}

float Heightmap::sampleHeight3x3(int i, int j) {
    float avg = 0.0f;
    float num = 0.0f;

    // averages itself with its eight neighbor pixels.
    for(int m = i-1; m <= i+1; ++m) {
        for(int n = j-1; n <= j+1; ++n) {
            // note that if a pixel is missing neighbor, we don't include it in averages
            if( inBounds(m,n) ) {
                avg += mHeightMap(m,n);
                num += 1.0f;
            }
        }
    }
    return avg / num;
}
```

$$\tilde{h}_{i,j} = \frac{h_{i-1,j-1} + h_{i-1,j} + h_{i-1,j+1} + h_{i,j-1} + h_{i,j} + h_{i,j+1} + h_{i+1,j-1} + h_{i+1,j} + h_{i+1,j+1}}{9}$$



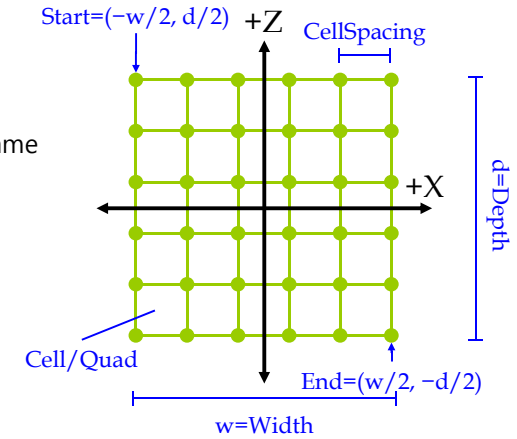
Basic Terrain Demo

- Heightmaps
- Building the Terrain Geometry
- Lighting & Texturing the Terrain



Building the Terrain Geometry

- Terrain geometry
  - # of cells per row
  - # of cells per column
  - Cell spacing
  - Heightmap data filename
  - Height scale



Building the Terrain Geometry

```
void BasicTerrainDemo::buildGridGeometry()
{
    std::vector<D3DXVECTOR3> verts;
    std::vector<DWORD> indices;
    int vertRows = 129;
    int vertCols = 129;
    float dx = 1.0f;
    float dz = 1.0f;
    GenTriGrid(vertRows, vertCols, dx, dz, D3DXVECTOR3(0.0f, 0.0f, 0.0f), verts, indices);

    // Create the mesh.
    int numVerts = vertRows*vertCols;
    int numTris  = (vertRows-1)*(vertCols-1)*2;
    D3DVERTEXELEMENT9 elems[MAX_FVF_DECL_SIZE];
    UINT numElems = 0;
    HR(VertexPNT::Decl->GetDeclaration(elems, &numElems));
    HR(D3DXCreateMesh(numTris, numVerts,
            D3DXMESH_MANAGED, elems, gd3dDevice, &mTerrainMesh));
```

Building the Terrain Geometry

```
// Write the vertices.
VertexPNT* v = 0;
HR(mTerrainMesh->LockVertexBuffer(0,(void**)&v));

// width/depth
float w = (vertCols-1) * dx;
float d = (vertRows-1) * dz;
for(int i = 0; i < vertRows; ++i){
        for(int j = 0; j < vertCols; ++j) {
                DWORD index = i * vertCols + j;
                v[index].pos     = verts[index];
                v[index].pos.y   = mHeightmap(i, j);
                v[index].normal = D3DXVECTOR3(0.0f, 1.0f, 0.0f);
                v[index].tex0.x = (v[index].pos.x + (0.5f*w)) / w;
                v[index].tex0.y = (v[index].pos.z - (0.5f*d)) / -d;
        }
}
```

Building the Terrain Geometry

```
// Write the indices and attribute buffer.
WORD* k = 0;
HR(mTerrainMesh->LockIndexBuffer(0, (void**)&k));
DWORD* attBuffer = 0;
HR(mTerrainMesh->LockAttributeBuffer(0, &attBuffer));

// Compute the indices for each triangle.
for(int i = 0; i < numTris; ++i) {
        k[i*3+0] = (WORD)indices[i*3+0];
        k[i*3+1] = (WORD)indices[i*3+1];
        k[i*3+2] = (WORD)indices[i*3+2];
        attBuffer[i] = 0; // Always subset 0
}

HR(mTerrainMesh->UnlockIndexBuffer());
HR(mTerrainMesh->UnlockAttributeBuffer());
```

Building the Terrain Geometry

```
// Generate normals and then optimize the mesh.
HR(D3DXComputeNormals(mTerrainMesh, 0));

DWORD* adj = new DWORD[mTerrainMesh->GetNumFaces()*3];
HR(mTerrainMesh->GenerateAdjacency(EPSILON, adj));
HR(mTerrainMesh->OptimizeInplace(D3DXMESHOPT_VERTEXCACHE|
                                 D3DXMESHOPT_ATTRSORT, adj, 0, 0, 0));
delete[] adj;
}
```

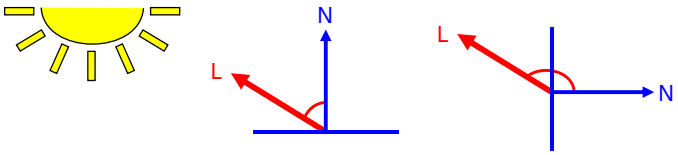
Terrain Texturing

- Load a texture from files

```
BasicTerrainDemo::BasicTerrainDemo(// 중간 생략....) {
    // 중간 생략..
    mHeightmap.loadRAW(129, 129, "heightmap17_129.raw", 0.25f, 0.0f);

    // Load textures from file.
    HR(D3DXCreateTextureFromFile(gd3dDevice, "grass.dds", &mTex0));
    HR(D3DXCreateTextureFromFile(gd3dDevice, "dirt.dds", &mTex1));
    HR(D3DXCreateTextureFromFile(gd3dDevice, "stone.dds", &mTex2));
    HR(D3DXCreateTextureFromFile(gd3dDevice, "blend_hm17.dds",
    &mBlendMap));

    buildGridGeometry();

    // 중간 생략..
}
```

Lighting

- diffuse surface lighting
  - Directional light
    - Light vector L
  - We compute the shade factor in the vertex shader
    - outVS.shade = saturate(max(0.0f, dot(normalW, gDirToSunW)) + 0.3f);
    - Diffuse light: max(0.0f, dot(normalW, gDirToSunW))
    - Ambient light: 0.3f
    - saturate HLSL function simply clamps the result to [0, 1]



The same texturing technique can be applied without modification

Vertex and Pixel Shaders

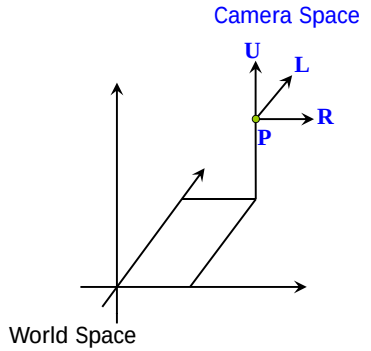
```
struct OutputVS {
    float4 posH         : POSITION0;
    float2 tiledTexC    : TEXCOORD0;
    float2 nonTiledTexC : TEXCOORD1;
    float  shade        : TEXCOORD2;
};
OutputVS TerrainVS(float3 posW : POSITION0, float3 normalW : NORMAL0,
    float2 tex0: TEXCOORD0) {
    OutputVS outVS = (OutputVS)0;
    // Just compute a grayscale diffuse and ambient lighting
    outVS.shade = saturate(max(0.0f, dot(normalW, gDirToSunW)) + 0.3f);
    // Transform to homogeneous clip space.
    outVS.posH = mul(float4(posW, 1.0f), gViewProj);
    // Pass on texture coordinates to be interpolated in rasterization.
    outVS.tiledTexC    = tex0 * gTexScale; // Scale tex-coord to tile.
    outVS.nonTiledTexC = tex0; // Blend map not tiled.
    return outVS;         }
```

Vertex and Pixel Shaders

```
float4 TerrainPS(float2 tiledTexC : TEXCOORD0,
        float2 nonTiledTexC : TEXCOORD1, float shade : TEXCOORD2) : COLOR {
    // Layer maps are tiled
    float3 c0 = tex2D(Tex0S, tiledTexC).rgb;
    float3 c1 = tex2D(Tex1S, tiledTexC).rgb;
    float3 c2 = tex2D(Tex2S, tiledTexC).rgb;
    // Blendmap is not tiled.
    float3 B = tex2D(BlendMapS, nonTiledTexC).rgb;
    // Find the inverse of all the blend weights – scale the total color to [0, 1].
    float totalInverse = 1.0f / (B.r + B.g + B.b);
    // Scale colors by each layer by its corresponding weight stored in blendmap
    c0 *= B.r * totalInverse;
    c1 *= B.g * totalInverse;
    c2 *= B.b * totalInverse;
    // Sum the colors and modulate with the shade to brighten/darken.
    float3 final = (c0 + c1 + c2) * shade;
    return float4(final, 1.0f);     }
```

Building a Flexible Camera

- In view space, the camera is positioned at the origin and looking down the positive z-axis, and the x-axis extends out from the right side of the camera, and the y-axis extends upward.
- Camera frame of reference
  - **(P, R, U, L)**
  - P = Position vector
  - R = Right vector
  - U = Up vector
  - L = Look vector



Camera Design

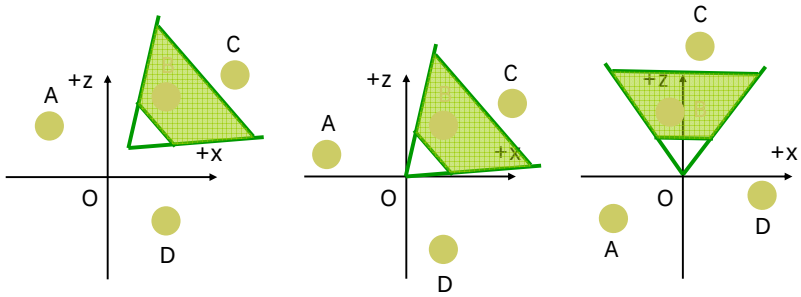
- Camera functionality – 6DOF
  - Pitch – rotate the camera around its right vector
  - Yaw – rotate the camera around its up vector
  - Roll – rotate the camera around its look vector
  - Strafe – move the camera along its right vector
  - Fly – move the camera along its up vector
  - Walk – move the camera long its look vector
- Camera type
  - Flight simulation mode – 6DOF
  - Terrain mode – the camera always modes tangent to the terrain surface

View Matrix

- View space transformation – view matrix **V**

p = (p_x, p_y, p_z) : position vector
r = (r_x, r_y, r_z) : right vector
u = (u_x, u_y, u_z) : up vector
l = (l_x, l_y, l_z) : look vector

pV** = (0, 0, 0) **: origin
rV** = (1, 0, 0) **: x-axis
uV** = (0, 1, 0) **: y-axis
lV** = (0, 0, 1) **: z-axis



View Matrix Transformation: Translation

- Move **p** to the origin

$$\mathbf{x} - \mathbf{p} = 0 \qquad (x \quad y \quad z) - (p_x \quad p_y \quad p_z) = (0 \quad 0 \quad 0)$$

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & -p_y & -p_z & 1 \end{bmatrix}$$

$$\mathbf{xT} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & -p_y & -p_z & 1 \end{bmatrix} = \begin{bmatrix} x - p_x & y - p_y & z - p_z & 1 \end{bmatrix}$$

View Matrix Transformation: Rotation

- Rotate the camera vectors (**r**, **u**, **l**) aligned with the world coordinates

$$\mathbf{rR} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix} \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{uR} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix} \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{lR} = \begin{bmatrix} l_x & l_y & l_z \end{bmatrix} \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

View Matrix Transformation: Rotation

- Rotate the camera vectors (**r**, **u**, **l**) aligned with the world coordinates

$$\mathbf{rR} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix} \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{uR} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix} \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{lR} = \begin{bmatrix} l_x & l_y & l_z \end{bmatrix} \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

$$\begin{matrix} \mathbf{M} & & \mathbf{R} & & \mathbf{I} \end{matrix}$$

$$\begin{bmatrix} r_x & r_y & r_z \\ u_x & u_y & u_z \\ l_x & l_y & l_z \end{bmatrix} \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{MR} = \mathrm{I} \Rightarrow \mathrm{R} = \mathrm{M}^{-1}$$

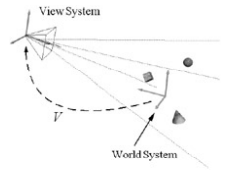
$$\mathrm{R} = \mathrm{M}^{-1} = \mathrm{M}^{\mathrm{T}} = \begin{bmatrix} r_x & u_x & l_x \\ r_y & u_y & l_y \\ r_z & u_z & l_z \end{bmatrix}$$

View Matrix Transformation: V

- View matrix **V**:

$$\mathbf{V} = \mathbf{TR} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & -p_y & -p_z & 1 \end{bmatrix} \begin{bmatrix} r_x & u_x & l_x & 0 \\ r_y & u_y & l_y & 0 \\ r_z & u_z & l_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} r_x & u_x & l_x & 0 \\ r_y & u_y & l_y & 0 \\ r_z & u_z & l_z & 0 \\ -\mathrm{p}\cdot\mathrm{r} & -\mathrm{p}\cdot\mathrm{u} & -\mathrm{p}\cdot\mathrm{l} & 1 \end{bmatrix}$$



Camera Class

```
class Camera {
public:
    Camera();
    const D3DXMATRIX& view() const;
    const D3DXMATRIX& proj() const;
    const D3DXMATRIX& viewProj() const;

    D3DXVECTOR3& pos();
    const D3DXVECTOR3& right() const;
    const D3DXVECTOR3& up() const;
    const D3DXVECTOR3& look() const;

    // our implementation of D3DXMatrixLookAtLH
    void lookAt(D3DXVECTOR3& pos, D3DXVECTOR3& target, D3DXVECTOR3& up);
    // Perspective projection parameters.
    void setLens(float fov, float aspect, float nearZ, float farZ);
```

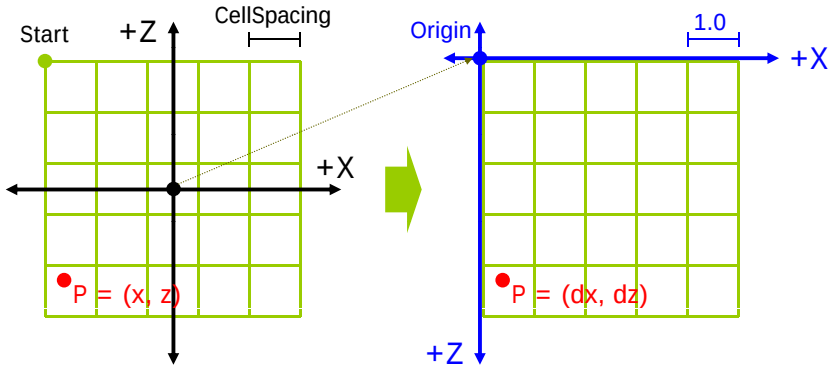
Camera Class

```
    // Sets the camera speed.
    void setSpeed(float s);
    // Updates the camera's basis vectors and origin, relative to the world space,
     void update(float dt);

protected:
    void buildView();        // build the view matrix
    D3DXMATRIX mView;
    D3DXMATRIX mProj;
    D3DXMATRIX mViewProj;
    D3DXVECTOR3 mPosW;
    D3DXVECTOR3 mRightW;
    D3DXVECTOR3 mUpW;
    D3DXVECTOR3 mLookW;
    float mSpeed;
};
```

Building the View Matrix

```
void Camera::buildViewMatrix()
{
    // Keep camera's axes orthogonal to each other
    D3DXVec3Normalize(&mLookW, &mLookW);

    D3DXVec3Cross(&mUpW, &mLookW, &mRightW);
    D3DXVec3Normalize(&mUpW, &mUpW);

    D3DXVec3Cross(&mRightW, &mUpW, &mLookW);
    D3DXVec3Normalize(&mRightW, &mRightW);

    // Build the view matrix:
    float pr = -D3DXVec3Dot(&mRightW, &mPosW);
    float pu = -D3DXVec3Dot(&mUpW, &mPosW);
    float pl = -D3DXVec3Dot(&mLookW, &mPosW);

    mView(0, 0) = mRightW.x;    mView(1, 0) = mRightW.y;
    mView(2, 0) = mRightW.z;    mView(3, 0) = pr;
    mView(0, 1) = mUpW.x;       mView(1, 1) = mUpW.y;
    mView(2, 1) = mUpW.z;       mView(3, 1) = pu;
    mView(0, 2) = mLookW.x;     mView(1, 2) = mLookW.y;
    mView(2, 2) = mLookW.z;     mView(3, 2) = pl;
    mView(0, 3) = 0.0f;         mView(1, 3) = 0.0f;
    mView(2, 3) = 0.0f;         mView(3, 3) = 1.0f;
    }
```

Walking on the Terrain



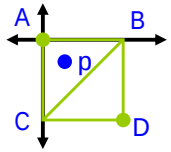
- start → origin
- cell spacing → 1.0
- -z → +z

floor(*t*) = 가장 큰 정수 ≤ *t*
셀을 구성하는 네 버텍스의 높이를 얻어 낼 수 있음

Walking on the Terrain

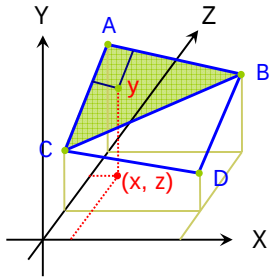
- Given the x- and z-coordinates of the camera position, we want to know how to adjust the y-coordinate of the camera on top of the terrain.

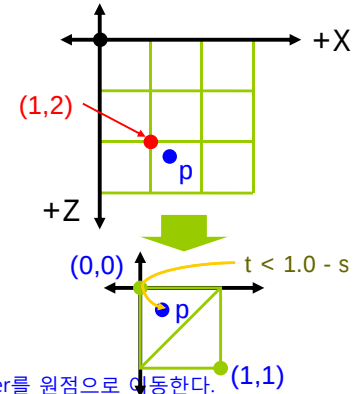
```
float Terrain::getHeight(float x, float z) {
    // transform from 'terrain local space' to 'cell space'
    float c = (x + 0.5f*mWidth) / mDX;
    float d = (z – 0.5f*mDepth) / -mDZ;

    // scaling cellspacing to 1 (1 / cellspacing)
    c /= (float)mCellSpacing;
    d /= (float)mCellSpacing;

    // get the row and color we are in
    int col = (int) floorf(c);
    int row = (int) floorf(d);

    // get the heights of the cell we are in
    float A = mHeightmap(row,   col);
    float B = mHeightmap(row,   col+1);
    float C = mHeightmap(row+1, col);
    float D = mHeightmap(row+1, col+1);
```



Walking on the Terrain

- Need to know in which triangle of the cell we are.

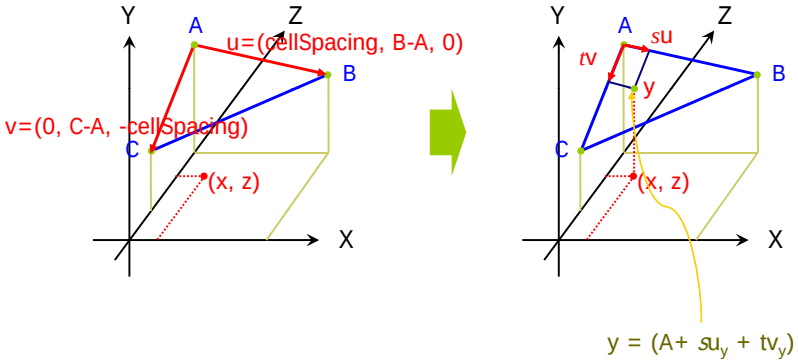





```
// 현재 있는 곳의 triangle을 찾는다.
// 현재 있는 곳의 cell의 upper left corner를 원점으로 이동한다.
// 따라서 cellspacing이 1로 되어 있으므로, +x/+z가 오른쪽/아래이며
// 원점이 정사각형이 있게 된다.
// +x -> 'right' and +z -> 'down' system.
float s = c – (float)col;
float t = d – (float)row;
```

Walking on the Terrain

- linear interpolation



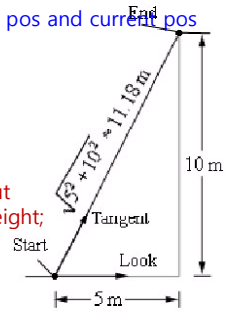
Walking on the Terrain

```
    // calculate u, v
    float height = 0.0f;
    if(dz < 1.0f - dx)  // upper triangle ABC
    {
        float uy = B - A; // A->B
        float vy = C - A; // A->C

    // Linearly interpolate on each vector.  The height is the vertex
    // height the vectors u and v originate from {A}, plus the heights
    // found by interpolating on each vector u and v.
        height = A + s*uy + t*vy;
    }
    else // lower triangle DCB
    {
        float uy = C - D; // D->C
        float vy = B - D; // D->B

    // Linearly interpolate on each vector.  The height is the vertex
    // height the vectors u and v originate from {D}, plus the heights
    // found by interpolating on each vector u and v.
        height = D + (1.0f – s)*uy + (1.0f - t)*vy;
    }
    return height;
}
```

Moving Tangent to the Terrain

- We also want to move tangent to the terrain surface.

```
void Camera::update(float dt, Terrain * terrain, float offsetHeight) {
    if (terrain != 0) {
    // move at mSpeed along net direction
    D3DXVec3Normalize(&dir, &dir);
    D3DXVECTOR3 newPos = mPosW + dir*mSpeed*dt;

    // approximate tangent vector using the difference of the new pos and current pos
    D3DXVECTOR3 tangent = newPos - mPosW;
    D3DXVec3Normalize(&tangent, &tangent);

    // now move camera along tangent vector
    mPosW += tangent*mSpeed*dt;

    // force camera to correct height, offset by the specified amout
    mPosW.y = terrain->getHeight(mPosW.x, mPosW.z) + offsetHeight;
    }
    else {
    mPosW = newPos;
    }
```



Moving Tangent to the Terrain

```
    // rotate at a fixed speed
    float pitch = gDInput->mouseDY() / 150.0f;
    float yaw = gDInput->mouseDX() / 150.0f;

    // rotate camera's look and up vectors around the camera's right vector
    D3DXMATRIX R;
    D3DXMatrixRotationAxis(&R, &mRightW, pitch);
    D3DXVec3TransformCoord(&mLookW, &mLookW, &R);
    D3DXVec3TransformCoord(&mUpW, &mUpW, &R);

    // Rotate camera axes about the world's y-axis.
    D3DXMatrixRotationY(&R, yaw);
    D3DXVec3TransformCoord(&mRightW, &mRightW, &R);
    D3DXVec3TransformCoord(&mUpW, &mUpW, &R);
    D3DXVec3TransformCoord(&mLookW, &mLookW, &R);

    // Rebuild the view matrix to reflect changes.
    buildView();
    mViewProj = mView * mProj;
}
```