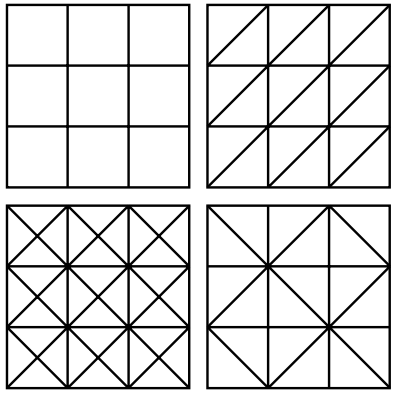
# Terrain rendering

- Terrain is an approximation of the surface shape of an outdoor environment
- Usually consider the world as being flat, not curved
- Terrain defined as a heightfield
  - Stores the height at numerous sample points over a 2D plane
  - Height at any spot found by interpolating between nearby sample points
- Challenge is to render a detailed terrain fast
  - For example, have a heightfield with 1 sample point per square metre, want to have terrain 10km by 10km = 100 million sample points. Can we keep them all in memory? If we render 1 polygon per sample point, that means 100 million polygons to render per frame!



# Terrain rendering (cont.)

- A basic terrain:
  - Terrain is a regular grid
  - Store a height at every grid point
  - Create polygons by connecting the grid points of a grid square
    - Two triangles or one quad per square
    - Possibly use four triangles, or alternate diagonal direction to remove direction bias





# Terrain rendering (cont.)

- Render a basic terrain by going through all grid squares and rendering its polygons

```
float height[size][size];

glBegin(GL_TRIANGLES);
for(z = 0; z < size-1; z++)
  for(x = 0; x < size-1; x++)
  {
    glVertex3f(x, height[z][x], z);
    glVertex3f(x+1, height[z][x+1], z);
    glVertex3f(x+1, height[z+1][x+1], z+1);

    glVertex3f(x, height[z][x], z);
    glVertex3f(x+1, height[z+1][x+1], z+1);
    glVertex3f(x, height[z+1][x], z+1);
  }
glEnd();
```



# Terrain rendering (cont.)

- Each vertex can be shared with 4, 6, or even 8 polygons
- Re-use the vertices, e.g. triangle strips:

```
for(z = 0; z < size-1; z++)
{
  glBegin(GL_TRIANGLE_STRIP);
  for(x = 0; x < size-1; x++)
  {
    glVertex3f(x, height[z][x], z);
    glVertex3f(x, height[z+1][x], z+1);
  }
  glEnd();
}
```

  - Could also use vertex arrays, but beware of maximum number of vertices
  - Terrain is usually static, so can also put in a display list for further optimisation

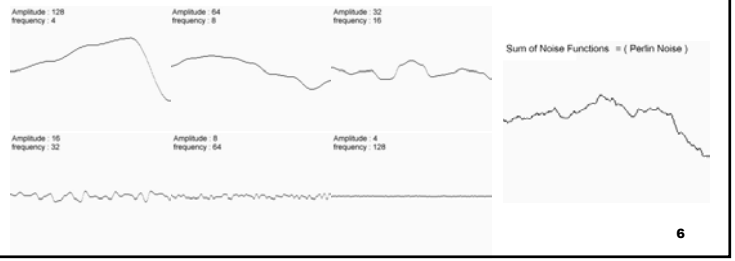
# Terrain rendering (cont.)

- How to create the heights?
  - From real-world data, e.g. DEM (Digital Elevation Model) files
  - Modeled by hand in 3D
  - Modeled by hand by painting a 2D image
    - Each pixel corresponds with a grid point
    - Greyscale value of pixel indicates height
    - 8 bits barely enough, may need to use more bits (use colour instead of greyscale, use format with more than 8 bits per channel such as PNG)
  - Generate procedurally (algorithmically)
    - Psuedo-random, Perlin noise
    - Sum sine functions with randomized phase and amplitude
    - Bezier patches (curved surfaces) with random displacements



# Terrain rendering (cont.)

- Perlin Noise: the basic idea:
  - Sum a bunch of random number sequences
  - Each sequence has half the amplitude and double the frequency of the previous sequence
  - Amplitude is the range of random numbers in sequence
  - Frequency can be chosen by sampling the random number sequence at some interval and interpolating






# Terrain rendering (cont.)

- Terrain generation by subdivision (midpoint displacement):
  - Start off by assigning random heights to the vertices of a square. Then:
    1. Set midpoint of each square edge by averaging heights at edge vertices
    2. Set midpoint of each square by averaging heights of square vertices, plus some random value
    3. Create 4 sub-squares
    4. Recurse for each sub-square
  - Set amplitude of random value based on the size of the square
  - Lots of variations possible on the algorithm, e.g. randomise edge midpoint heights (but note that edges are shared), include a smoothing pass, randomise the position of the square midpoint, change to a breadth-first subdivision instead of depth-first

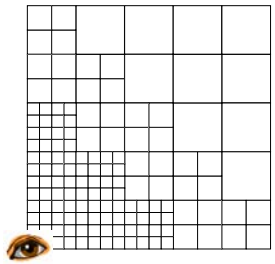


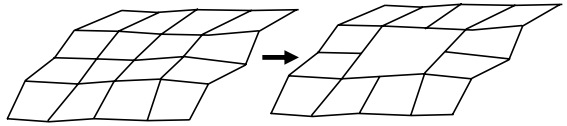
# Terrain rendering (cont.)

- To render large terrains, need to be able to do visibility culling, mesh simplification, and level of detail control
  - Visibility culling: for example, 60 degree FOV means typically only 1/6th of the terrain is visible, rest can be culled
  - Simplification: many polygons that form a large flat area could be replaced with a few polygons which approximate the terrain close enough
  - Level of detail (LOD): Far-away polygons look much smaller than nearby polygons. At some distance polygons may be smaller than a few pixels. Replacing many small polygons by fewer larger polygons should cause little visual change
- Simplification is a pre-process, independent of camera position and orientation
- Visibility depends on camera position and orientation
- LOD depends on camera position and orientation, and usually the shape of the terrain. It is a dynamic version of simplification

# Terrain rendering (cont.)

- For the moment, assume we make the polygons dynamically instead of using a vertex array or display list
- Visibility culling done by only processing squares which intersect the view frustum
  - Computational shortcuts using the fact that the terrain is a regular grid
- LOD: if a grid square becomes too small when projected, consider 2x2 grid squares as a single square when creating polygons
- Simplification: if merging 2x2 grid squares gives a result similar enough to unmerged grid squares, merge the squares





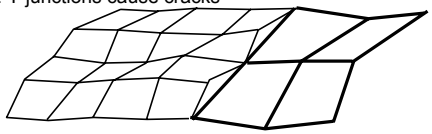


# Terrain rendering (cont.)

- Looks like spatial subdivision!
  - Keep the terrain as a quadtree
  - To render terrain:

```
RenderTerrain(node)
    If node intersects view frustum
        If node is a leaf
            Render the node
        If projected size of node < size threshold
            Render the node
        Else if render difference between node and child-nodes < error threshold
            Render the node
        Else
            For all child nodes
                RenderTerrain(child node)
```

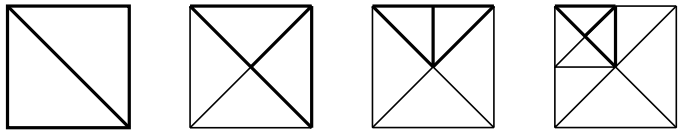
- Problem: T-junctions cause cracks





# Terrain rendering (cont.)

- An alternative to quad trees: binary triangle trees
  - Start with two triangles
  - Split each triangle into two
  - T-junctions can be avoided
    - If we decide to split a triangle, also split the triangle on the other side of the hypotenuse
    - May need to split other triangle multiple times
    - Easier but more triangles: ensure never more than one LOD difference between neighbouring triangles

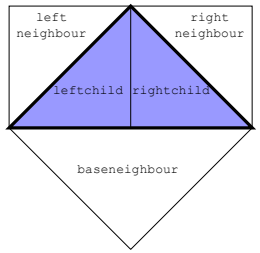




# Terrain rendering (cont.)

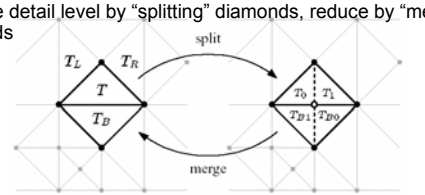
- Triangle bintree node
  - Need to keep track of neighbours in case they need to be split to avoid cracks

```
struct TriTreeNode
{
        TriTreeNode* leftchild;
        TriTreeNode* rightchild;
        TriTreeNode* baseneighbour;
        TriTreeNode* leftneighbour;
        TriTreeNode* rightneighbour;
};
```



# Terrain rendering (cont.)

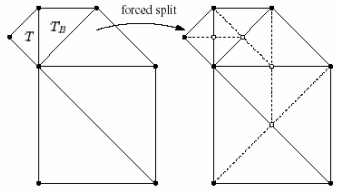
- ROAM
  - "ROAMing Terrain: Real-time Optimally Adapting Meshes", Duchaineau et al. (1997)
  - Represents terrain as two triangle bintrees, one for each half of the terrain
    - Neighbour pointers can point into other tree
    - Left/right neighbours at same level or level+1 as triangle
    - Base neighbour at same level or level-1 as triangle
    - If base neighbour at same level as triangle, call the pair a "diamond"
  - Increase detail level by "splitting" diamonds, reduce by "merging" diamonds






# Terrain rendering (cont.)

- Any triangle bintree can be changed into any other triangulation by a sequence of splits and merges
- Splitting introduces a new vertex
  - Get the height of the new vertex from a heightmap, or procedurally
  - Can introduce vertex gradually by animating its height from midpoint to new height over a number of frames
- Merging removes a vertex
  - Can animate like with splitting, only in reverse (animate from vertex height to edge midpoint, then remove vertex by merging)
- To split a triangle with a base neighbour at a lower detail level, must first split the base neighbour to same level
  - … which in turn may require splitting of further base neighbours
  - Such splits are called "forced splits"






# Terrain rendering (cont.)

- When to split or merge?
  - Compute error values ("error metric")
  - Split to reduce error until maximum allowable error is reached
    - Ensures that terrain is rendered with desired accuracy
  - Merge to increase error until maximum allowable error is reached
    - Ensures that the smallest number of polygons is used to render the terrain with desired accuracy
  - Errors for child triangles must be no larger than parent triangle
  - Errors define a priority for each triangle
  - Keep two priority queues: a split queue and a merge queue



# Terrain rendering (cont.)

- Split queue
  - Start off with the coarsest triangulation, then start splitting the triangles with largest error

```
For all triangles in base triangulation
        Insert triangle in split queue

While triangulation is not good enough
        Take highest priority triangle from split queue
        Split triangle with forcing if needed
        Remove triangles that were (force-)split from queue
        Add newly created triangles to queue
```

# Terrain rendering (cont.)

- Merge queue
  - Instead of starting from scratch and using the splitting algorithm every frame, re-use the triangulation from previous frame and modify it
  - Split queue algorithm can still be used to do extra splits
  - The merge queue is used to merge diamonds which no longer should be split
  - Merge queue contains mergeable diamonds
  - Priority of diamond is given by maximum error of its two triangles



# Terrain rendering (cont.)

- ROAM algorithm with split and merge queues:

```
If first frame
        Let T be the base triangulation
        Clear split and merge queues Qs Qm
        Compute priorities for triangles and diamonds in T
        Insert triangles in Qs
        Insert diamonds in Qm
Else
        Let T be the current triangulation
        Update priorities for elements in Qs, Qm

While T is not optimal
        If T is too large or accurate
                Take lowest priority diamond (t, tb) from Qm
                Merge (t, tb)
                Remove all merged children from Qs
                Add merged parents t, tb to Qs
                Remove (t, tb) from Qm
                Add all newly-mergeable diamonds to Qm
        Else
                Take highest priority triangle t from Qs
                Force-split t
                Remove t and other split triangles from Qs
                Add any new triangles in T to Qs
                Remove diamonds whose children were split from Qm
                Add all newly-mergeable diamonds to Qm
```



# Terrain rendering (cont.)

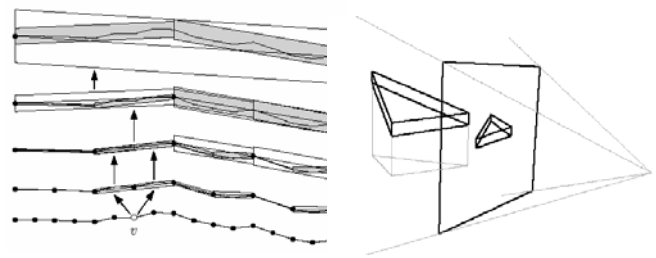
- Many implementations of ROAM only use the split queue (split-only ROAM)
  - Always starts at the base triangulation
  - Extra time needed for all that splitting per frame is regained by avoiding the more complex merge queue handling
- Can terminate the “while” loop at any time
  - Have a time budget for doing LOD
  - When out of time, stop doing ROAM
  - Terrain is always valid



# Terrain rendering (cont.)

- Error metrics
  - Many metrics possible
  - Some are more expensive, but give better results
  - Different error metrics are good for dealing with different cases
  - Combine several error metrics
  - Effects are often not exactly predictable. Try and tweak
  - Can have error metrics for entire terrain, each triangle, or each vertex
- ROAM’s error metric:
  - Uses a “wedgie” around a triangle
    - A thick triangle which bounds the wedgies of the child triangles
    - Wedgie has zero thickness for leaves, i.e. equal to leaf triangle
    - Wedgies are pre-computed
    - Error for triangle at a given LOD is the maximum projected thickness of the wedgie at that level

# Terrain rendering (cont.)





# Terrain rendering (cont.)

- Modify priority for important triangles, or triangles along important lines of sight, to subdivide them in preference to others (e.g. ridgelines, mountain tops)
- Error for entire visible terrain is the maximum pointwise distortion
  - For each point of the terrain at max LOD level
    - Project its actual position onto the screen
    - Project its position as given by triangulation onto screen
    - Compute distance between the two projections
  - Terrain error is the maximum distance found
  - Slow, so in practice an upper bound is used by taking the maximum triangle error as found using the wedgies



# Terrain rendering (cont.)

- ROAM breaks down at high LOD (Jonathan Blow, GDC 2000)
  - ROAM error metrics computes a 1D value from 3D data
  - Hence aliasing is inevitable
    - Different situations give same error, even if they should be different
    - ROAM doesn't distinguish between wedges coming closer and wedges moving away
    - Nearby wedges are approximately the same size as the distance moved by the viewer in a frame step
    - So nearby wedges are constantly being re-evaluated
    - But most wedges are nearby!
  - Solution: use a sphere instead of a wedge for a triangle
    - Sphere defines region where the camera has to be in order for the triangle to be split. When camera moves into sphere, split. When camera moves out of sphere, merge
    - Spheres are stored in a tree, like a bounding sphere hierarchy.



# Terrain rendering (cont.)

- Methods such as ROAM are "continuous" level of detail
  - LOD changes across the terrain anywhere
- Good: LOD at any point is (near) optimal, giving the smallest number of polygons needed to render the terrain within a given error bounds
- Bad: terrain mesh changes every frame, so need to send it to gfx card every frame
- Ideal when cost of spending CPU time to minimise the number of polygons and sending them to the gfx card every frame is outweighed by the reduction in rendering time
- True at the time that ROAM was created, but is it still true today?
- Answer: No!
  - Gfx cards like geometry in constant unchanging chunks which are re-used over numerous frames

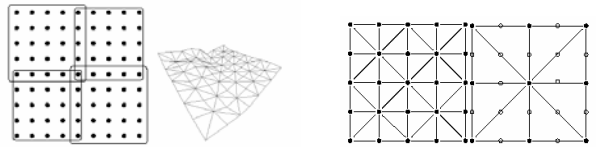
# Terrain rendering (cont.)

- Chunked level of detail
  - LOD the same over a large block of the terrain
  - LOD of a block changes only occasionally
  - Blocks are pre-defined
  - So geometry of each block for each LOD can be pre-computed and optimised as vertex arrays or display lists
  - Gfx card keeps blocks that are currently in use, have recently been used, or are likely to soon be used
  - Only need to upload blocks which have newly come into view, or blocks whose LOD has changed, which are not stored on the gfx card already (very similar to caching algorithms)



# Terrain rendering (cont.)

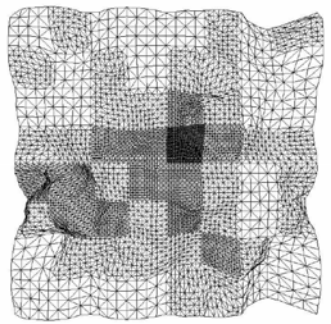
- Larsen & Christensen: “Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail” (2003)
  - Divide a regular grid into tiles
    - Number of vertices in a tile given by level of detail
    - The world size of a tile is the same for all levels of detail
  - Tiles are geometrically independent from each other
  - Tiles duplicate one row/column of vertices with neighbour tiles
  - Put tiles into the leaves of a quadtree for fast culling
  - Tiles are triangulated using alternating diagonals





# Terrain rendering (cont.)

- Each LOD of each tile can be pre-computed and turned into a display list or vertex array
- Terrain is rendered by choosing the LOD for each tile, then calling the display list for that tile at that LOD
- LOD chosen based on an screen-space error metric
- Problems to solve:
  - Cracks between tiles with different LOD
  - Smooth transition when a tile changes LOD



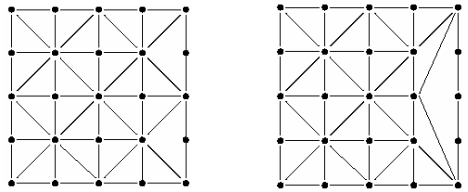


# Terrain rendering (cont.)

- Error metric
  - Uses method of de Boer, “Fast Terrain Rendering Using Geometrical MipMapping” (2000)
  - Largest change in projected height at vertices of tile as a result of changing detail level
  - Changes up to about 4 pixels are acceptable; reduce level of detail until the change becomes more than that
  - Use an approximation instead of exact for speed
    - Assume camera is always viewing horizontally
    - Can pre-compute max change in height
    - Tends to use too high LOD when looking down, but CPU time saved may be more than extra render time needed
  - Also pre-computes a minimum distance camera has to be from tile for a particular LOD to be used

# Terrain rendering (cont.)

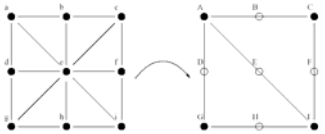
- Avoiding cracks
  - Make neighbouring tiles with different LOD join up
  - Modify the geometry of the tile with the lower LOD
    - Need to keep pointers to neighbour tiles in quadtree





# Terrain rendering (cont.)

- Smooth transition between LOD for a tile
  - Uses a morphing method
  - To change to a higher LOD, interpolate new vertices from midpoint to new height
  - But vertices at tile boundary are duplicated in neighbours
    - Boundary vertices of the neighbours may need to be changed as well so that the tiles still join without cracks
  - Morphing is done by a vertex program on the graphics card, so tile does not need to be updated by the CPU



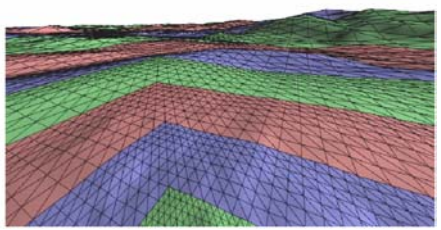
Morph calculations

$$A = a$$
$$B = v\frac{(a+c)}{2} + (1-v)b$$
$$C = c$$
$$D = v\frac{(a+g)}{2} + (1-v)d$$
$$E = v\frac{(a+i)}{2} + (1-v)e$$
$$F = v\frac{(c+i)}{2} + (1-v)f$$
$$G = g$$
$$H = v\frac{(g+i)}{2} + (1-v)h$$
$$I = i$$



# Terrain rendering (cont.)

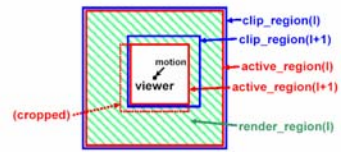
- Losasso & Hoppe “Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids” (2004)
  - Concentrate on distance-dependent LOD only
  - Have one tile per LOD
  - Tiles have the same number of vertices (n*n)
  - Tile at lower level of detail is twice the world size
  - Tiles are stacked on top of each other centred on the viewer
  - Control the number of triangles to render by changing how much of each tile to render





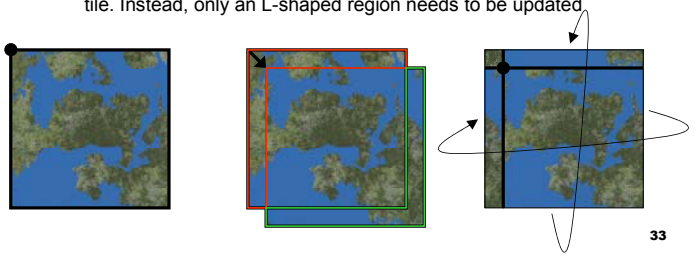
# Terrain rendering (cont.)

- Tiles are 2D arrays, accessed with wrap-around (toroidal, “donut”)
  - Do not need to shift all the data in a tile when viewer moves
- Define several square regions for each level:
  - Clip region: world extent of n*n grid
  - Active region: world extent of n*n grid we wish to render (may be offset from clip region), centred on the viewer
  - Render region: hollowed frame between active region of level and active region of next level
- When the viewer moves, shift clip region to match desired active region
- If there is not enough time to update the clip region, let the clip region fall behind the active region
  - Gap between clip and active region filled by previous LOD tile
  - Adjust render region of previous level to fill to the cropped active region



# Terrain rendering (cont.)

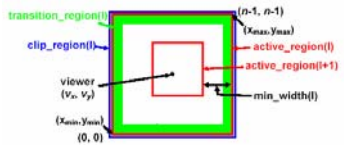
- Number of vertices in a tile chosen so that average projected size of triangles is some number of pixels (about 3)
  - n is fixed
  - If viewer is looking down from high up, don't render the highest LOD tiles
- Shifting the clip regions
  - As toroidal access is used, do not need to shift all the heights in the tile. Instead, only an L-shaped region needs to be updated

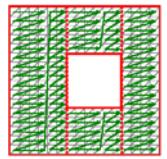




# Terrain rendering (cont.)

- The render region is rendered using triangle strips
- Transition between render regions of tiles by morphing over a transition region inside the active region
- May still get single pixel errors at boundaries due to numerical errors
  - Fill by stitching the vertices of the render regions together with zero-area triangles





# Terrain rendering (cont.)

- Terrain heightmap data is compressed
  - Create a pyramid of heightmaps by scaling by 50%
  - Can predict next LOD heightmap by interpolating previous heightmap
  - Difference between predicted and actual gives a residual map
  - Compress residual map with a lossy image compression method
  - Only decompress blocks which are needed
- For fine detail, use procedural algorithms (e.g. noise) to fake detail



# Terrain rendering (cont.)

- Texturing the terrain
  - Terrain texturing is used to give the appearance of detail at smaller scale than can be conveniently rendered with polygons
  - For example, use polygons down to a scale of 1 metre, then use texture to paint details at 1cm scale
  - The problem: terrain is too large to cover with unique textures, but the repeating nature of a tileable texture looks obvious when looking over a large area of the terrain
  - An answer: detail textures

# Terrain rendering (cont.)

- Use different textures at various levels of detail
- Blend them together by varying amounts to break up repetitiveness
- At low level of detail, texture shows large-scale details (land, water, wooded areas, snow)
  - Typically not tileable, but stretched over the entire terrain or part of the terrain
- At high level of detail, textures represent a patch of grass, pebbles, water ripples, small square of sand
  - Typically tileable, covering one or a few grid squares (e.g. 1x1 metre grid square, 256x256 texture, want about 1cm detail, means texture can be stretched over 2x2 or 3x3 grid squares)



# Terrain rendering (cont.)

- Multitexturing
  - Can apply more than one texture to a polygon at the same time
  - How many depends on the gfx card
    - Given by the number of texture units
    - Typically 2, 4, 8, 16
  - Each texture unit can have its own texture loaded with its own parameters
  - Can specify separate texture coordinates at a vertex for each texture unit
  - Texture units form a pipeline, where the result from one texture unit can be fed into the next texture unit where it is combined in some way
  - Introduced in OpenGL1.3, so probably need to get the functions as extensions



# Terrain rendering (cont.)

OpenGL2.0 spec. pp182-187, 190

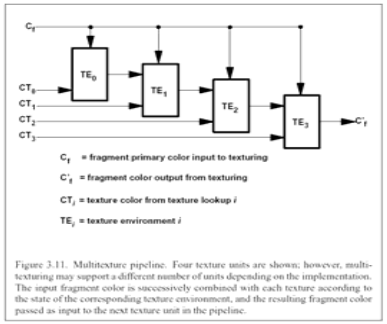



Figure 3.11. Multitexture pipeline. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. The input fragment color is successively combined with each texture according to the state of the corresponding texture environment, and the resulting fragment color passed as input to the next texture unit in the pipeline.



# Terrain rendering (cont.)

| COMBINE_RGB | Texture Function |
|---|---|
| REPLACE | $Arg0$ |
| MODULATE | $Arg0 * Arg1$ |
| ADD | $Arg0 + Arg1$ |
| ADD_SIGNED | $Arg0 + Arg1 - 0.5$ |
| INTERPOLATE | $Arg0 * Arg2 + Arg1 * (1 - Arg2)$ |
| SUBTRACT | $Arg0 - Arg1$ |
| DOT3_RGB | $4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$ |
| DOT3_RGBA | $4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$ |

| COMBINE_ALPHA | Texture Function |
|---|---|
| REPLACE | $Arg0$ |
| MODULATE | $Arg0 * Arg1$ |
| ADD | $Arg0 + Arg1$ |
| ADD_SIGNED | $Arg0 + Arg1 - 0.5$ |
| INTERPOLATE | $Arg0 * Arg2 + Arg1 * (1 - Arg2)$ |
| SUBTRACT | $Arg0 - Arg1$ |

Table 3.24: COMBINE texture functions. The scalar expression computed for the DOT3_RGB and DOT3_RGBA functions is placed into each of the 3 (RGB) or 4 (RGBA) components of the output. The result generated from COMBINE_ALPHA is ignored for DOT3_RGBA.

| SRC0_RGB | OPERAND0_RGB | Argument |
|---|---|---|
| TEXTURE | SRC_COLOR | $C_s$ |
| | ONE_MINUS_SRC_COLOR | $1 - C_s$ |
| | SRC_ALPHA | $A_s$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_s$ |
| TEXTURE*n* | SRC_COLOR | $C_s^n$ |
| | ONE_MINUS_SRC_COLOR | $1 - C_s^n$ |
| | SRC_ALPHA | $A_s^n$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_s^n$ |
| CONSTANT | SRC_COLOR | $C_c$ |
| | ONE_MINUS_SRC_COLOR | $1 - C_c$ |
| | SRC_ALPHA | $A_c$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_c$ |
| PRIMARY_COLOR | SRC_COLOR | $C_f$ |
| | ONE_MINUS_SRC_COLOR | $1 - C_f$ |
| | SRC_ALPHA | $A_f$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_f$ |
| PREVIOUS | SRC_COLOR | $C_p$ |
| | ONE_MINUS_SRC_COLOR | $1 - C_p$ |
| | SRC_ALPHA | $A_p$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_p$ |

Table 3.25: Arguments for COMBINE_RGB functions.

| SRC0_ALPHA | OPERAND0_ALPHA | Argument |
|---|---|---|
| TEXTURE | SRC_ALPHA | $A_s$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_s$ |
| TEXTURE*n* | SRC_ALPHA | $A_s^n$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_s^n$ |
| CONSTANT | SRC_ALPHA | $A_c$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_c$ |
| PRIMARY_COLOR | SRC_ALPHA | $A_f$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_f$ |
| PREVIOUS | SRC_ALPHA | $A_p$ |
| | ONE_MINUS_SRC_ALPHA | $1 - A_p$ |

Table 3.26: Arguments for COMBINE_ALPHA functions.

# Terrain rendering (cont.)

```
// Work on texture unit 2. Units are counted from 0
glActiveTexture(GL_TEXTURE2);

// Enable the use of the current texture unit
glEnable(GL_TEXTURE_2D);

// Use texture combining for the current texture unit
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
// Bind some texture to the current texture unit
glBindTexture(GL_TEXUTRE_2D, texture);

// Set the combine method for RGB colours
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_INTERPOLATE);
// GL_INTERPOLATE mixes two sources together based on a third source
// result = source 0 * source 2 + source 1 * (1 - source 2)

// Set source 0 to RGB of texture in unit 0
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE0);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);

// Set source 1 to RGB result of previous unit
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB, GL_PREVIOUS);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_RGB, GL_SRC_COLOR);

// Set source 2 to alpha of texture in current unit
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE2_RGB, GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND2_RGB, GL_SRC_ALPHA);
```

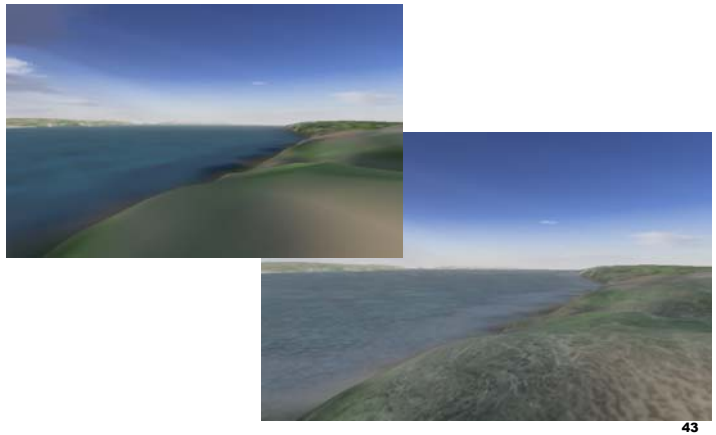


# Terrain rendering (cont.)

- For example:
  - A texture giving the large scale overall colouring and shading of the entire terrain
  - 4 tileable greyscale detail textures for small scale
  - A texture map defined over the entire terrain whose 4 channels specify how much of each detail texture to mix together at any point
- Detail textures can be combined into one RGBA texture, one detail texture in each channel
- Assign the mixing values to the vertex colours (“primary” in multitexturing) instead of using a texture
- Use a constant to balance the strength between the detail texturing and the colour texture
- Can now stuff it all into two texture units:
  1. Detail texture. (R,G,B,A,)_1 = (R,G,B,A)_detail . (R,G,B,A)_mix
     ```
     RGB1 = DOT3(TEXTURE:SRC_COLOUR, PRIMARY_COLOR:SRC_COLOUR)
     ALPHA1 = TEXTURE:SRC_ALPHA * PRIMARY_COLOR:SRC_ALPHA
     ```
  2. Colour texture. (R,G,B,A)_2 = (1-(R,G,B,A)_1) * c + (R,G,B,A)_colour * (1-c)
     ```
     RGB2 = INTERPOLATE(PREVIOUS:ONE_MINUSSRC_COLOUR, TEXTURE;SRC_COLOR,
                        CONSTANT:SRC_COLOR)
     ALPHA2 = PREVIOUS:SRC_ALPHA * CONSTANT:SRC_ALPHA
     ```
- Render with glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_ZERO) to blend in detail texture in alpha channel with the other three detail textures



# Terrain rendering (cont.)





# Terrain rendering (cont.)

- Texture coordinates
  - Texture coordinates at a vertex of the terrain are the (x,z) coordinates of the vertex scaled and translated
  - Several ways of assigning texture coordinates
    1. Compute for each vertex based on its (x,z) coordinates, and assign with `glMultiTexCoord()` which is like `glTexCoord()` except that you also specify the texture unit to set coords for
    2. Let OpenGL generate the texture coordinates automatically from the vertex coordinates
    3. Set texture coordinates for all vertices, then use the texture matrix to apply scaling and translation to the texture coordinates for each texture unit

# Terrain rendering (cont.)

```
const float scale0 = 1.0f / (size-1);
const float scale1 = 0.5f;

glBegin(...);
...
glMultiTexCoord2f(0, x*scale0, z*scale0);
glMultiTexCoord2f(1, x*scale1 + 0.25f, z*scale1 - 0.75f);
glVertex3f(x,y,z);
...
glEnd();
```



# Terrain rendering (cont.)

```
float plane0[2][4] = { { 1.0f / (size-1), 0, 0, 0 },
                       { 0, 1.0f / (size-1), 0, 0 } };
float plane1[2][4] = { { 0.5f, 0, 0,  0.25f },
                       { 0, 0.5f, 0, -0.75f } };

glActiveTexture(GL_TEXTURE0);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGenf(GL_S, GL_OBJECT_PLANE, plane0[0]);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGenf(GL_T, GL_OBJECT_PLANE, plane0[1]);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);

glActiveTexture(GL_TEXTURE1);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGenf(GL_S, GL_OBJECT_PLANE, plane1[0]);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGenf(GL_T, GL_OBJECT_PLANE, plane1[1]);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);

glBegin(...);
...
glVertex3f(x,y,z);
...
glEnd();
```



# Terrain rendering (cont.)

```
glMatrixMode(GL_TEXTURE);
glActiveTexture(GL_TEXTURE0);
glLoadIdentity();
glScalef(1.0f/(size-1), 1.0f/(size-1), 0);

glActiveTexture(GL_TEXTURE1);
glLoadIdentity();
glTranslatef(0.25, -0.75, 0);
glScalef(0.5f, 0.5f, 0);

glMatrixMode(GL_MODELVIEW);

glBegin(...);
...
glMultiTexCoord2f(0, x,z); glMultiTexCoord2f(1, x,z);
glVertex3f(x,y,z);
...
glEnd();
```



# Terrain rendering (cont.)

- What if the terrain is too large to fit a texture over it?
  - For example, have gigabytes of satellite photos for the entire country at 10 metre resolution
- Use the same sort of LOD algorithms used for the heightfields, but then for the texture data