Assembling Multiple Tiles into Large Textures

General Idea

- We have created tiles that fit against themselves in any direction, left to right or top to bottom
- We have also seen a technique to create multiple instances of tiles that will fit against any tile of that instance type, giving us textures that will minimize repetition
- Now, the goal is to create textures that will combine tiles from different texture types and make sure that we have no visible seams as we move from one type of terrain to another
- This will be done using a library of texture building blocks

Texture Building Block Library

- Each texture in the library is of same [fixed] size, say 256×256 pixels and shows a given area in the same resolution ($\frac{1}{2} \times \frac{1}{2}$ mile)
- Textures in the library are created such that each edge of a texture shows only one type of terrain
- Textures will fit against each other seamlessly in top-down or left-right sides as long as the common edges represent the same terrain type
- Problem is in terms of recognizing the textures and what is on each edge
  - Solved by the following scheme
  - Each terrain type is numbered from 00 to 99 (two-digit ASCII numbers)
  - This gives us the ability to handle up to 100 terrain types
  - Use the numbers to create names for texture patterns
- Texture naming scheme
  - Textures need to be identified at the time we assemble the large texture
  - It will be nice to be able to tell the type of terrain contained in the texture tile without having to examine it, or establishing some correlation table
  - We have just assigned two-digit numbers to identify up to 100 different types of terrain
  - We can accommodate multiple instances of same texture type tiles by using an instance number
  - Now, our texture tiles can contain different types of tiles along each edge of the tile
  - A two-digit number along each side of the tile identifies the texture type along that edge
  - Each tile is now named $nneesswwsd$.rgb
    - Each of nn, ee, ss, and ww is replaced by a two-digit number to describe the terrain texture type along that direction
    - d gives the instance number of the texture tile containing the pattern, varying between 0 and 9
    - If urban area is denoted by 01, a tile containing just the urban area will be given by 01010101s0.rgb (instance 0 of the tile)

Rules for Tile Placement

- With the library in place, we need rules to place tiles by some specified method, to get a feel for the terrain
- Pseudocolor map

  - Used to describe the general placement of tiles
  - Describes the desired layout of the terrain
  - Describes the colors to correlate a terrain tile to an area on our desired texture
  - Accompanied by a color table to describe the tile types to be placed in the corresponding color area on the map
  - Can be generated from maps or satellite data for realism
  - Color described in terms of its RGB values, and correlated to terrain types through an index value
    - Orange in pseudocolor map corresponds to urban area
    - Orange color RGB values are placed in line 1 in the color table
- Relationship between pseudocolors and terrain types
  - Flat file with following structure

000	085	136	// 0: Water
255	221	100	// 1: City
239	074	036	// 2: Farm/Town
000	255	000	// 3: Forest
187	221	136	// 4: Farmland
242	255	000	// 5: Farm/Trees

Algorithm to place tiles

- Overview of algorithm
  - Recognize different colors in the pseudocolor map
  - Place tiles corresponding to differnt colors in their position
  - Place crossover tiles to provide blending from one terrain type to another
- Recognizing pseudocolors
  - Input pseudocolor template is a rectangular array of 24-bit RGB pixel values
  - Size of input template may not match the number of tiles
    - For example, a 1000×2000 pixel pseudocolor template may be used to build a terrain map of 300×400 tiles
    - In this case, each tile in the generated terrain corresponds to 3.33×5 pixels in the pseudocolor template
- Classifying pseudocolors
  - Identify colors corresponding to known terrains, classifying each pixel block (or partial pixel) into two categories:
    1. Forced colors
       - Indicate tiles required to be present in the indicated position
       - Indicated by a high degree of match with a color in pseudocolor table
       - High degree of match defined as a probability of match > 0.95
    2. Desired colors
       - These colors do not have a high degree of match with any color in the color table
       - Algorithm should strive to fill these blocks with desired texture tiles
  - Comparing colors
    - Find the average color of pixel block in the pseudocolor map corresponding to a tile
    - Average color can be found in RGB space, L*a*b* space, or HSV space
    - Compare the average color with all the colors in pseudocolor map table and compute the distance between average color and all pseudocolors

    * If there is a pseudocolor whose distance is 0.05 times the maximum possible distance between colors in the selected colorspace, that color is considered to be forced color
    * RGB and L*a*b* color space distance is simplest to compute, given by Euclidean distance; in HSV, you must account for the angle
    * Another way to compute colors in RGB space is based on psychophysical measurements
      · Human eye assigns different weights to each component in the RGB color
      · Use the following expression to assign maximum weight to the green component

$$\sqrt{0.299 \times (r_1 - r_2)^2 + 0.587 \times (g_1 - g_2)^2 + 0.114 \times (b_1 - b_2)^2}$$

      · This provides a better comparison in RGB color space directly

- Modeling algorithm
  - In theoretical terms, can be described as a bin-packing algorithm
  - Characterize the input pseudocolor map as a symbolic template, with forced and desired tiles based on color recognition
  - Determine clusters of like colors/tiles in the symbolic template
  - Determine neighboring clusters
  - Grow the regions between neighboring clusters, using desired colors as a guide for placement of tiles
  - Fill in the rest of the template
- Results of modeling algorithm
  - We'll have all the textures filled in properly due to forced tile, with multiple instances to avoid repetition of patterns
  - We'll have an indication of textures to go in the unassigned pixel groups in terms of color distribution
  - We'll know the texture tiles to go in the regions from the name of forced tiles
    * South of water will be water
    * West of urban area will be urban area
- Symbolic template

		w	w												
	w	w	w	w											
	w	w	w	w					u	u	u	u	u		
	w	w	w	w	w	?	?	u	u	u	u	u	u	u	
	w	w	w	w				u	u	u	u	u	u	u	
	w	w	w	w					u	u	u	u	u		
		w	w	w											

- Clusters of forced tiles
  - Identify clusters of forced textures by looking at the 4-neighbors of each forced tile
  - 4-neighbors N of pixels P are given by

	N	
N	P	N
	N	

- Cluster identification
  - Initialize a cluster template where each element corresponds to a texture tile and is initialized to 0 (unassigned)

  - Start at top-left tile, and go through each tile position in the template in row-major order – left to right and top to bottom
  - If the texture is a forced tile, it is part of a cluster; possibly a singleton
  - If the forced tile in symbolic template is the same as the one above or to the left (4-neighbor), copy the cluster number of corresponding matching tile
  - If the forced tile does not match the one above or to the left, assign it as a new cluster
- Cluster properties
  - Determine the boundaries of each cluster as well as its centroid
  - Boundaries are described by the maximum and minimum x and y coordinates for the cluster in the symbolic template, given in terms of column number and row number
  - Centroid is determined by the area of cluster, and could conceivably be outside the cluster, but within the cluster boundary
- Growing the clusters
  - Determine the distance of each cluster centroid from every other cluster centroid and store it in an array with cluster index poiinters
  - Number of clusters is given by the highest valued cluster index number
  - Sort the array in increasing order by distance between clusters
  - Join the clusters by using city block traversal from left to right and top to bottom (or bottom to top if cluster to the left is lower than the one to the right)
  - Joining clusters implies filling in the ? symbols that were left in the symbolic template earlier
  - These symbols depict the unknown terrain types, and we need to fill them with the desired texture tiles based on the pseudocolor vector that we saved corresponding to each tile in the symbolic template
  - At this point, we'll have a symbolic template with all tiles assigned but there is a slight problem
    - We assigned the desired tiles without any regard to what goes on in the neighborhood
    - We may have inconsistencies in terms of mismatch with neighboring tiles
      - Left side of the right tile could be urban area while the right side of left tile depicts water
    - These inconsistencies give well-defined straight line boundaries between tiles that stand out from the rest of the textures
- Fixing the symbolic template problems
  - Start at the bottom right of the template and look at each tile, comparing it to the ones above and to the left
  - If the edges do not match the corresponding edges above and to the left, change the edge designation of tiles above and to the left to match the current tile's edges
  - We may have to modify the current tile in some cases but in no case, we should modify the tiles to the bottom and to the right
- Overlaying tiles
  - We now have a symbolic template that shows the terrain type for each tile in terms of edges
    - We could have water towards the bottom edge, urban area towards the left edge, and so on
  - We simply have to pick up tiles from the library and place them as dictated by the symbolic template
  - We can use random instances of tiles that were generated earlier to reduce repetition and achieve more variation in terrain
  - The naming scheme for tiles has all the required information corresponding to our pseudocolor table – $nneesswwsd$.rgb