Adaptive Scalable Texture Compression
Supplemental Material

J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, T. Olson

ASTC uses computational methods to generate the partition patterns that map the texels in a block to color spaces. This makes it possible to support a much larger repertoire of patterns than is practical with table-driven patterns. This note presents pseudocode for the partition function, with brief comments to identify the major steps of the computation. See the full paper for a more detailed explanation. Operators ^, <<, and & denote bit operations as in C and C++.

The code relies on the following hash function to convert a twelve-bit key into a highly random 32-bit integer:

```
unsigned int32 hash52( unsigned int32 p )
  p := p – (p << 17)
  p := p + (p <<  7)
  p := p + (p <<  4)
  p := p ^ (p >>  5)
  p := p + (p << 16)
  p := p ^ (p >>  7)
  p := p ^ (p >>  3)
  p := p ^ (p <<  6)
  p := p ^ (p >> 17)
  return p
```

This function was created by evaluating random sequences of the bit operations shown using the randomness tests described in "Some difficult-to-pass tests of randomness" by Marsaglia and Tsang, J. Statistical Software v. 7 no. 3, 2002, available at http://www.jstatoft.org/v07/i03. This function passes all of their tests, as well as the default configuration of the DieHarder test (http://code.googl.com/p/dieharder/).

The following uninteresting utility function splits a 32-bit argument into 4-bit fields, squares them, and stores them in an 8-element byte array.

```
unsigned int8 out[8] split4(unsigned int32 in)
  for i from 0 to 7 do
    tmp    := (in >> (i<<2)) & 0xF
    out[i] := tmp * tmp
  return
```

The partition function follows. Arguments are:

- ID: the ten-bit partition pattern identifier
- pc: the desired partition count: 2, 3, or 4
- x, y, z: the (integer) position of the texel within the block

Note that for 2D patterns, z should be set to zero. Also, for small blocks (defined as those with 30 texels or less), the coordinates should be multiplied by two, to scale the patterns to the smaller block size.

```
unsigned int32 partition(int32 ID, int32 pc,
                         int32 x, int32 y, int32 z)
  ID = ID + ((pc-1) << 10);
  unsigned int32 R := hash52(ID);
  unsigned int8  A[8], B[8];
```

Generate 12 random 8-bit values by squaring uniformly distributed 4-bit values. Squaring them biases the distribution toward smaller values:

```
  unsigned int8 A := split4(R)
  unsigned int8 B := split4((R >> 18) | (R << 14))
```

These scaling operations bias 50% of the patterns toward horizontal or vertical orientations:

```
  int sh1 := (pc == 3 ? 6 : 5)
  int sh2 := (ID & 2 ? 4 : 5)
  if (ID & 1) swap(sh1, sh2)
  int sh3 := (ID & 0x10) ? sh1 : sh2;

  A[0] := A[0] >> sh1
  A[1] := A[1] >> sh2
  A[2] := A[2] >> sh1
  A[3] := A[3] >> sh2
  A[4] := A[4] >> sh1
  A[5] := A[5] >> sh2
  A[6] := A[6] >> sh1
  A[7] := A[7] >> sh2
  B[0] := B[0] >> sh3
  B[1] := B[1] >> sh3
  B[2] := B[2] >> sh3
  B[3] := B[3] >> sh3
```

Use the above random values as coefficients in four plane equations. Evaluate the equations at the sample point, and then extract the bottom six bits. This has the effect of converting the planes into sawtooth functions.

```
int a := A[0]*x + A[1]*y + B[2]*z + (R >> 14)
int b := A[2]*x + A[3]*y + B[3]*z + (R >> 10)
int c := A[4]*x + A[5]*y + B[0]*z + (R >> 6)
int d := A[6]*x + A[7]*y + B[1]*z + (R >> 2)
a := a & 0x3F
b := b & 0x3F
c := c & 0x3F
d := d & 0x3F
```

Zero out samples corresponding to partitions we are not interested in:

```
if( pc < 4 ) d := 0
if( pc < 3 ) c := 0
```

Return the index of the plane that is on top at (x,y,z)

```
if      ( a is max(a,b,c,d) ) return 0
else if ( b is max(a,b,c,d) ) return 1
else if ( c is max(a,b,c,d) ) return 2
else return 3;
```