Memory-Hazard-Aware K-Buffer Algorithm for Order-Independent Transparency Rendering

Nan Zhang, Member, IEEE

APPENDIX I: IEEE 754 Floating Point Format

IEEE 754 standard represents floating-point numbers in binary. For a 32-bit single precision floating point number, it consists of three fields: a 1-bit sign, a 8-bit biased exponent, and a 23-bit fraction. For a binary floating number $x$, IEEE 754 first requires $x$ to be rewritten to a “scientific form” with a single 1 bit to the left of the "binary point". Then, the three fields for $x$ are:

- sign = 0 for positive, 1 for negative;
- biased exponent = $x$’s exponent + 127;
- fraction = $x$’s fraction;

Fig. 1 illustrates the three fields using a decimal number 0.1875$_{10}$, which can be represented in binary as 0.00112 = $1.12 \times 2^{-3}$.

IEEE 754 defines specific encodings for special numbers:

Denormal number: A denormal number is represented with a biased exponent of all 0 bits and a non-zero fraction. Note that denormal numbers are not supported in current graphics hardware.

Positive and negative infinity: sign = 0 for positive infinity, 1 for negative infinity; biased exponent = all 1 bits; fraction = all 0 bits.

Not a number (NaN): sign = either 0 or 1; biased exponent = all 1 bits; fraction = anything except all 0 bits.

APPENDIX II: SAMPLE GLSL FRAGMENT PROGRAM

We provide a sample implementation of the GLSL fragment program in the main geometry pass for the MHA k-buffer algorithm. The example is for the compact 2-MRT-buffer case. The k-buffer elements are stored in the first 5 positions, the ECC uses the 6th position, and the rest two are for the compressed RGBA color. The main sorting function is mha_kbuffer_frasort(), which has two parameters: the integer-valued array and the incoming depth value in integer. Note that it assumes that the depth value has been combined with the lighting dot value in the lower bits. The code shows the convenience of using a descending order of k-buffer array elements, where only GPU registers are used for the sorting, after its compilation.

```c
const uint TOTAL_BITS = 32;
const uint SEQ_BITS = 9;
const uint ZDEPTH_BITS = (TOTAL_BITS - SEQ_BITS);
const uint ZDEPTH_BITS_MASK = (1u<<(ZDEPTH_BITS)) - 1;

//encode float RGBA color into uvec2
uvec2 encode_color(const in vec4 fcolor);
//decode compressed uvec2 into float RGBA color
vec4 decode_color(const in uvec2 encoded_rgba);

void insertKBuffer(inout ivec4 mem[2], in int x){
    int t=x;
    for (int i=0; i<2; ++i){
        ivec4 e = mem[i];
        mem[i].x=max(e.x, t); t=min(e.x, t);
        mem[i].y=max(e.y, t); t=min(e.y, t);
        mem[i].z=max(e.z, t); t=min(e.z, t);
        mem[i].w=max(e.w, t); t=min(e.w, t);
    }
}

bool existInArray(in ivec4 mem[2], in int x){
    ivec4 inarray = equal(mem[0], ivec4(x));
    inarray += equal(mem[1], ivec4(x));
    inarray.xy += inarray.zw;
    inarray.x += inarray.y;
    return (inarray.x!=0);
}

void checkECCField(in ivec4 elements[2], in int ecc,
    in int ipos, out int missingZval){
    bool Valid = (ecc!=0) && (ecc!=ZDEPTH_BITS_MASK);
    int X = ZDEPTH_BITS_MASK - ecc;
    bool E = existInArray(elements, X);
    bool B = (X != ipos);
    bool C = (!E) && B;
    missingZval = (C && Valid) * X;
}

vec4 compositeSurfColor(in vec4 bgcolor, in int dist){
    const vec3 matcolor = vec3(255, 168, 245)/255.0;
    const float K = (1.0/255.0); //scaling factor
    const float a = 0.50; //alpha value
    const float Ka = K*a;
    float diffuse = float(dist & 0x0000007F)*Ka;
    float s = (1.0-bgcolor.w)*float(dist!=0);
    return bgcolor + vec4(matcolor*diffuse, a)*s;
}
```
void encodeEccField(in uint timestamp, in int ipos, out int ecc)
{  
timestamp = update_timestamp(timestamp);
  timestamp <<= ZDEPTHBITS;
  int encodedpos = ZDEPTH_BITS_MASK - ipos;
  ecc = encodedpos | timestamp;
}

void mha_kbuffer_fragsort(inout ivec4 reg[2], in int ipos)
{  
  //decode k-buffer element array
  uint timestamp = uint(reg[1].y) >> ZDEPTH_BITS;
  reg[0] &= ZDEPTH_BITS_MASK;
  reg[1].xy &= ZDEPTH_BITS_MASK;
  int ecc = reg[1].y;
  vec4 fcolor = decode_color(reg[1].zw);
  reg[1].yzw = ivec3(0);

  //insert the incoming z value into the array
  insertKBuffer(reg, ipos);

  //insert the missing z-value into the array
  int missingZval;
  checkECCField(reg, ecc, ipos, missingZval);
  insertKBuffer(reg, missingZval);

  //accumulate surface color
  fcolor = compositeSurfColor(fcolor, reg[1].z);
  fcolor = compositeSurfColor(fcolor, reg[1].y);

  //encode the ECC code with time sequence ID
  encodeEccField(timestamp, ipos, reg[1].y);

  //avoid denormalized numbers (biased exponent all 0)
  reg[0] |= 0x40000000;
  reg[1].x |= 0x40000000;
  reg[1].zw = encode_color(fcolor);
}