Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering

Steven P. Callahan, Milan Ikits, João L. D. Comba, and Cláudio T. Silva

APPENDIX I
VERTEX PROGRAM FOR HAVS

```
!!ARBvp1.0
# ---------------------------------------------------------------------------
# Vertex program for Transactions on Visualization and Computer Graphics:
# "Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering"
# (C) 2005 Steven Callahan, Milan Ikits, Joao Comba, Claudio Silva
# ---------------------------------------------------------------------------

ATTRIB iPos = vertex.position;
ATTRIB iTex0 = vertex.texcoord[0];
PARAM mvp[4] = { state.matrix.mvp };
PARAM mv[4] = { state.matrix.modelview };
OUTPUT oPos = result.position;
OUTPUT oTex0 = result.texcoord[0];
OUTPUT oTex1 = result.texcoord[1];

# ---------------------------------------------------------------------------
# transform vertex to clip coordinates
DP4 oPos.x, mvp[0], iPos;
DP4 oPos.y, mvp[1], iPos;
DP4 oPos.z, mvp[2], iPos;
DP4 oPos.w, mvp[3], iPos;

# ---------------------------------------------------------------------------
# transform vertex to eye coordinates
DP4 oTex1.x, mv[0], iPos;
DP4 oTex1.y, mv[1], iPos;
DP4 oTex1.z, mv[2], iPos;

# ---------------------------------------------------------------------------
# texcoord 0 contains the scalar data value
MOV oTex0, iTex0;

END
```

APPENDIX II
FRAGMENT PROGRAM FOR HAVS

```
!!ARBfp1.0
# ---------------------------------------------------------------------------
# Fragment program for Transactions on Visualization and Computer Graphics:
# "Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering"
# (C) 2005 Steven Callahan, Milan Ikits, Joao Comba, Claudio Silva
# ---------------------------------------------------------------------------
#
# The program consists of the following steps:
#
# 1. Find the first and second entries in the fixed size k-buffer list sorted
#    by d (6+1 entries)
# 2. Perform a 3D pre-integrated transfer function lookup using front and back
#    scalar data values + the segment length computed from the distance values
#    of the first and second entries from the k-buffer.
# 3. Composite the color and opacity from the transfer funcion with the
#    color and opacity from the framebuffer. Discard winning k-buffer entry,
#    write the remaining k-buffer entries.
#
# The following textures are used:
#
#   Tex 0: framebuffer (pbuffer, 2D RGBA 16/32 bpp float)
#   Tex 1: k-buffer entry 1 and 2(same)
#   Tex 2: k-buffer entry 3 and 4(same)
#   Tex 3: k-buffer entry 5 and 6(same)
#   Tex 4: transfer function (regular, 3D RGBA 8/16 bpp int)
#

# ---------------------------------------------------------------------------
# use the ATI_draw_buffers extension
OPTION ATI_draw_buffers;
# this does not matter now, but will matter on future hardware
OPTION ARB_precision_hint_nicest;

# ---------------------------------------------------------------------------
# input and temporaries
ATTRIB p = fragment.position; # fragment position in screen space
ATTRIB v = fragment.texcoord[0]; # v.x = scalar value
ATTRIB e = fragment.texcoord[1]; # fragment position in eye space
PARAM sz = program.local[0]; # scale and bias parameters
                             # {1/pw, 1/ph, (1-1/z_size)/max_len, 1/(2*z_size)}
TEMP a0, a1, a2, a3, a4, a5, a6; # k-buffer entries
TEMP r0, r1, r2, r3, r4, r5, r6, r7; # sorted results
TEMP c, c0; # color and opacity
TEMP t; # temporary (boolean flag for min/max, dependent texture coordinate,
        # pbuffer texture coordinate, fragment to eye distance)
```

S. Callahan, M. Ikits, and C. Silva are with the Scientific Computing and Imaging Institute, University of Utah.
J. Comba is with the Federal University of Rio Grande do Sul, Brazil.

```
# ---------------------------------------------------------------------------
# compute texture coordinates from window position so that it is not
# interpolated perspective correct.  Then look up the color and opacity from
# the framebuffer
MUL t, p, sz; # t.xy = p.xy * sz.xy, only x and y are used for texture lookup
TEX c0, t, texture[0], 2D; # framebuffer color

# ---------------------------------------------------------------------------
# Check opacity and kill fragment if it is greater than a constant tolerance
SUB t.w, 0.99, c0.w;
KIL t.w;

# ---------------------------------------------------------------------------
# set up the k-buffer entries a0, a1, a2, a3, a4, a5, a6
# each k-buffer entry contains the scalar data value in x or z
# and the distance value in y or w
TEX a1, t, texture[1], 2D; # k-buffer entry 1
TEX a3, t, texture[2], 2D; # k-buffer entry 3
TEX a5, t, texture[3], 2D; # k-buffer entry 5
MOV a2, a1.zwzw; # k-buffer entry 2
MOV a4, a3.zwzw; # k-buffer entry 4
MOV a6, a5.zwzw; # k-buffer entry 6

# ---------------------------------------------------------------------------
# compute fragment to eye distance
DP3 t, e, e;
RSQ t.y, t.y;
MUL a0.y, t.x, t.y; # fragment to eye distance
MOV a0.x, v.x; # scalar data value

# ---------------------------------------------------------------------------
# find fragment with minimum d (r0), save the rest to r1, r2, r3, r4, r5

# r0 = min_z(a0.y, a1.y); r1 = max_z(a0.y, a1.y);
SUB t.w, a0.y, a1.y; # t.w < 0 iff a0.y < a1.y
CMP r1, t.w, a1, a0; # r1 = (a0.y < a1.y ? a1 : a0)
CMP r0, t.w, a0, a1; # r0 = (a0.y < a1.y ? a0 : a1)

# r0 = min_z(r0.y, a2.y); r2 = max_z(r0.y, a2.y)
SUB t.w, r0.y, a2.y; # t.w < 0 iff r0.y < a2.y
CMP r2, t.w, a2, r0; # r2 = (r0.y < a2.y ? a2 : r0);
CMP r0, t.w, r0, a2; # r0 = (r0.y < a2.y ? r0 : a2);

# r0 = min_z(r0.y, a3.y); r3 = max_z(r0.y, a3.y)
SUB t.w, r0.y, a3.y; # t.w < 0 iff r0.y < a3.y
CMP r3, t.w, a3, r0; # r3 = (r0.y < a3.y ? a3 : r0)
CMP r0, t.w, r0, a3; # r0 = (r0.y < a3.y ? r0 : a3);

# r0 = min_z(r0.y, a4.y); r4 = max_z(r0.y, a4.y)
SUB t.w, r0.y, a4.y; # t.w < 0 iff r0.y < a4.y
CMP r4, t.w, a4, r0; # r4 = (r0.y < a4.y ? a4 : r0);
CMP r0, t.w, r0, a4; # r0 = (r0.y < a4.y ? r0 : a4);

# r0 = min_z(r0.y, a5.y); r5 = max_z(r0.y, a5.y)
SUB t.w, r0.y, a5.y; # t.w < 0 iff r0.y < a5.y
CMP r5, t.w, a5, r0; # r5 = (r0.y < a5.y ? a5 : r0);
CMP r0, t.w, r0, a5; # r0 = (r0.y < a5.y ? r0 : a5);

# r0 = min_z(r0.y, a6.y); r6 = max_z(r0.y, a6.y)
SUB t.w, r0.y, a6.y; # t.w < 0 iff r0.y < a6.y
CMP r6, t.w, a6, r0; # r6 = (r0.y < a6.y ? a6 : r0);
CMP r0, t.w, r0, a6; # r0 = (r0.y < a6.y ? r0 : a6);

# ---------------------------------------------------------------------------
# find fragment with minimum d (r7) from r1, r2

# r7 = min_z(r1.y, r2.y);
SUB t.w, r1.y, r2.y; # t.w < 0 iff r1.y < r2.y
CMP r7, t.w, r1, r2; # r7 = (r1.y < r2.y ? r1 : r2);

# r7 = min_z(r7.y, r3.y);
SUB t.w, r7.y, r3.y; # t.w < 0 iff r7.y < r3.y
CMP r7, t.w, r7, r3; # r7 = (r7.y < r3.y ? r7 : r3);

# r7 = min_z(r7.y, r4.y);
SUB t.w, r7.y, r4.y; # t.w < 0 iff r7.y < r4.y
CMP r7, t.w, r7, r4; # r7 = (r7.y < r4.y ? r7 : r4);

# r7 = min_z(r7.y, r5.y);
SUB t.w, r7.y, r5.y; # t.w < 0 iff r7.y < r5.y
CMP r7, t.w, r7, r5; # r7 = (r7.y < r5.y ? r7 : r5);

# r7 = min_z(r7.y, r6.y);
SUB t.w, r7.y, r6.y; # t.w < 0 iff r7.y < r6.y
CMP r7, t.w, r7, r6; # r7 = (r7.y < r6.y ? r7 : r6);

# ---------------------------------------------------------------------------
# set up texture coordinates for transfer function lookup

MOV t.x, r0.x; # front scalar
MOV t.y, r7.x; # back scalar
SUB t.z, r7.y, r0.y; # distance between front and back fragment
MAD t.z, t.z, sz.z, sz.w; # normalization scale and bias

# ---------------------------------------------------------------------------
# transfer function lookup
```

```
TEX c, t, texture[4], 3D; # look up pre-integrated color and opacity

# -----------------------------------------------------------------------------
# nullify winning entry if the scalar value < 0

CMP c, r0.x, 0.0, c;

# -----------------------------------------------------------------------------
# composite color with the color from the framebuffer !!!front to back!!!

SUB t.w, 1.0, c0.w;
MAD result.color[0], c, t.w, c0;

# -----------------------------------------------------------------------------
# write remaining k-buffer entries

MOV r1.zw, r2.xxxy;
MOV r3.zw, r4.xxxy;
MOV r5.zw, r6.xxxy;
MOV result.color[1], r1;
MOV result.color[2], r3;
MOV result.color[3], r5;

END
```