Name

AMD_vertex_shader_tessellator

Name Strings

GL_AMD_vertex_shader_tessellator

Contact

Bill Licea-Kane, AMD (Bill.Licea-Kane 'at' amd.com)

Status

Complete

Version

Last Modified Date: 2009-03-06
Author Revision: 8

Number

363

Dependencies

OpenGL 2.0 is required.

EXT_gpu_shader4 affects the definition of this extension.

EXT_geometry_shader4 affects the definition of this extension.

This extension interracts with AMDX_vertex_shader_tesselator.

This extension is written against the OpenGL Shading Language 1.20 Specification.

The extension is written against the OpenGL 2.1 Specification.

Overview

The vertex shader tessellator gives new flexibility to the shader author to shade at a tessellated vertex, rather than just at a provided vertex.

In unextended vertex shading, the built-in attributes such as gl_Vertex, gl_Normal, and gl_MultiTexcoord0, together with the user defined attributes, are system provided values which are initialized prior to vertex shader invocation.

With vertex shading tessellation, additional vertex shader special values are available:

```
    ivec3 gl_VertexTriangleIndex; // indices of the three control
                                  // points for the vertex
    vec3 gl_BarycentricCoord;     // barycentric coordinates
                                  // of the vertex
```

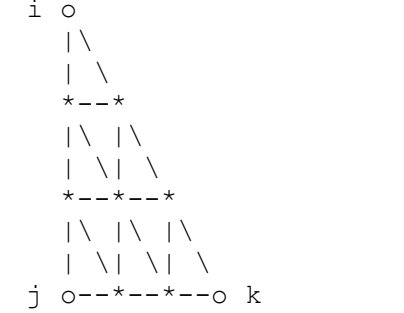

Figure 1 A Tessellated Triangle
o = control point (and tessellated vertex)
* = tessellated vertex

```
    ivec4 gl_VertexQuadIndex;   // indices for the four control
                                // points for the vertex
    vec2 gl_UVCoord;            // UV coordinates of the vertex
```

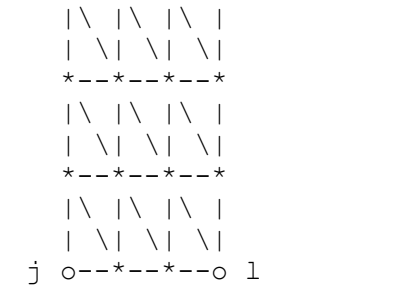

Figure 2 A Tessellated Quad
o = control point (and tessellated vertex)
* = tessellated vertex

When this extension is enabled, conventional built-in attributes and user defined attributes are uninitialized. The shader writer is responsible for explicitly fetching all other vertex data either from textures, uniform buffers, or vertex buffers.

The shader writer is further responsible for interpolating the vertex data at the given barycentric coordinates or uv coordinates of the vertex.

IP Status

No known claims.

New Procedures and Functions

```
    void TessellationFactorAMD( float factor );
    void TessellationModeAMD( enum mode );

New Types

    (None.)

New Tokens

    Returned by the <type> parameter of GetActiveUniform:

      SAMPLER_BUFFER_AMD                              0x9001
      INT_SAMPLER_BUFFER_AMD                          0x9002
      UNSIGNED_INT_SAMPLER_BUFFER_AMD                 0x9003

    Accepted by TessellationModeAMD

      DISCRETE_AMD                                    0x9006
      CONTINUOUS_AMD                                  0x9007

    Accepted by GetIntegerv

      TESSELLATION_MODE_AMD                           0x9004

    Accepted by GetFloatv

      TESSELLATION_FACTOR_AMD                         0x9005

Additions to Chapter 2 of the OpenGL 2.1 Specification
(OpenGL Operation)

   Modify section 2.15.3, "Shader Variables", page 75

   Add the following new return types to the description of
   GetActiveUniform on p. 81.

      SAMPLER_BUFFER_AMD,
      INT_SAMPLER_BUFFER_AMD,
      UNSIGNED_INT_SAMPLER_BUFFER_AMD.

   Replace section "Samplers" p. 83 with:

    Samplers

    Samplers are special uniforms used in the OpenGL Shading Language
    to identify the texture object or vertex buffer object used for
    each lookup.

    Samplers and Texture objects

    If the sampler is one of the texture types, the value of a sampler
    indicates the texture image unit being accessed.  Setting a
    sampler's value to i selects texture image unit number i. The values
```

of i range from zero to the implementation dependent maximum supported number of texture image units.

The type of the sampler identifies the target on the texture image unit. The texture object bound to that texture image unit's target is then used for the texture lookup. For example, a variable of type sampler2D selects target TEXTURE 2D on its texture image unit. Binding of texture objects to targets is done as usual with BindTexture. Selecting the texture image unit to bind to is done as usual with ActiveTexture.

It is not allowed to have variables of different sampler types pointing to the same texture image unit within a program object. This situation can only be detected at the next rendering command issued, and an INVALID OPERATION error will then be generated.

Samplers and vertex buffer objects

If the sampler is one of the vertex types, the value of a sampler indicates the vertex array being accessed. Setting a sampler's value to i selects vertex array i. The values of i range from zero to the implementation dependent maximum supported max vertex attributes. Binding of vertex buffer objects to vertex arrays is done as usual with BindBuffer.

It is not allowed to have multiple variables of samplers to the same vertex array within a program object. This situation can only be detected at the next rendering command issued, and an INVALID OPERATION error will then be generated.

All samplers

The location of a sampler needs to be queried with GetUniformLocation, just like any uniform variable. Sampler values need to be set by calling Uniform1i{v}. Loading samplers with any of the other Uniform* entry points is not allowed and will result in an INVALID OPERATION error.

Active samplers are samplers actually being used in a program object. The LinkProgram command determines if a sampler is active or not. The LinkProgram command will attempt to determine if the active samplers in the shader(s) contained in the program object exceed the maximum allowable limits. If it determines that the count of active samplers exceeds the allowable limits, then the link fails (these limits can be different for different types of shaders). Each active sampler variable counts against the limit, even if multiple samplers refer to the same texture image unit. If this cannot be determined at link time, for example if the program object only contains a vertex shader, then it will be determined at the next rendering command issued, and an INVALID OPERATION error will then be generated.

Insert section prior to "Validation" on p. 87

Tessellation

If a vertex shader enables GL_AMD_vertex_shader_tessellation, then the shader writer is responsible for fetching and evaluating the vertex attributes at the barycentric coordinates of the vertex. (See the shading language specification.)

Only indexed triangles or indexed quads may be drawn with such a shader. Each triangle or quad will introduce generated vertices (including the original vertices of the triangle or quad) controlled by:

```
void TessellationFactorAMD( float factor );
```

where the factor is a value between 1.0 and 15.0 inclusive

The introduction of generated vertices is further controlled by:

```
void TessellationModeAMD( enum mode );
```

where mode is either DISCRETE_AMD or CONTINUOUS_AMD.

Add to the list of "begin errors":

* any two samplers of vertex type refer to the same vertex array.
* Any sampler bound to a vertex array has vertex buffer object 0 bound.
* A vertex shader enables GL_AMD_vertex_shader_tessellation, statically reads gl_VertexTriangleIndex or gl_BarycentricCoord and the Implicit Begin mode is NOT GL_TRIANGLES
* A vertex shader enables GL_AMD_vertex_shader_tessellation, statically reads gl_VertexQuadIndex or gl_UVCoord and the Implicit Begin mode is NOT GL_QUADS
* A vertex shader enables GL_AMD_vertex_shader_tessellation and the command is RasterPos.

Additions to Chapter 3 of the OpenGL 2.1 Specification (Rasterization)

Additions to Chapter 4 of the OpenGL 2.1 Specification (Per-Fragment Operations and the Frame Buffer)

Additions to Chapter 5 of the OpenGL 2.1 Specification (Special Functions)

Additions to Chapter 6 of the OpenGL 2.1 Specification (State and State Requests)

Additions to Appendix A of the OpenGL 2.1 Specification (Invariance)

Modifications to The OpenGL Shading Language 1.20 Specification

Including the following line in a shader can be used to control the language features described in this extension:

#extension GL_AMD_vertex_shader_tessellator : <behavior>

where <behavior> is as specified in section 3.3.

A new preprocessor #define is added to the OpenGL Shading Language:

#define GL_AMD_vertex_shader_tessellator 1

Additions to Chapter 1 of the OpenGL Shading Language 1.20 Specification (Introduction)

Additions to Chapter 2 of the OpenGL Shading Language 1.20 Specification (Overview of OpenGL Shading)

2.1 Vertex Processor

Change 2nd paragraph to:

The vertex processor operates on one vertex at a time. It does not replace graphics operations that require knowledge of several vertices at a time. While a tessellated vertex however has LIMITED knowledge of the immediately adjacent control points (three for a triangle, four for a quad), the vertex processor is still operating on one tessellated vertex at a time. The vertex shaders running on the vertex processor must compute the homogeneous position of the incoming vertex.

Additions to Chapter 3 of the OpenGL Shading Language 1.20 Specification (Basics)

3.6 Keywords

Add the keywords

__samplerVertexAMD
__isamplerVertexAMD
__usamplerVertexAMD

Additions to Chapter 4 of the OpenGL Shading Language 1.20 Specification (Variables and Types)

4.3.4 Attribute, Change third sentence:

"Attribute values are read-only as far as the vertex shader is concerned, unless GL_AMD_vertex_shader_tessellator is enabled. If GL_AMD_vertex_shader is enabled, they are read-write with undefined initial values."

Additions to Chapter 5 of the OpenGL Shading Language 1.20 Specification (Operators and Expressions)

Additions to Chapter 6 of the OpenGL Shading Language 1.20 Specification (Statements and Structure)

Additions to Chapter 7 of the OpenGL Shading Language 1.20 Specification (Built-in Variables)

7.1 Vertex Shader Special Variables

Add the list of intrinsically declared with the following types:

```
        // if GL_AMD_vertex_shader_tessellator enabled

        ivec3 gl_VertexTriangleIndex; // may be read
                                      // indices of the three control
                                      // points for the vertex
        vec3 gl_BarycentricCoord;     // may be read
                                      // barycentric coordinates of the
                                      // vertex

        ivec4 gl_VertexQuadIndex;     // may be read
        vec2 gl_UVCoord;              // may be read
```

    If gl_VertexTriangleIndex and/or gl_BarycentricCoord is statically read by the shader, the shader is a Triangle Tessellator shader.

    If gl_VertexQuadIndex and/or gl_UVCoord is statically read by the shader, the shader is a Quad Tessellator shader.

    It is a link error if both a Triangle Tessellator shader and a Quad Tessellator shader are attached to a program.

7.3 Vertex Shader Built-In Attributes

Add the following paragraph.

    If GL_AMD_vertex_shader_tessellator is enabled, the values of the built-in Attributes are undefined.

Additions to Chapter 8 of the OpenGL Shading Language 1.20 Specification (Built-in Functions)

8.7 Texture Lookup Functions

Rename section to "Lookup Functions"

Add in front of first sentence:

    Vertex lookup functions are available to the vertex shader.

Add to the front of the table of functions:

    Syntax:
    vec4 vertexFetchAMD(__samplerVertexAMD sampler, int i);
    ivec4 vertexFetchAMD(__isamplerVertexAMD sampler, int i);
    uvec4 vertexFetchAMD(__usamplerVertexAMD sampler, int i);

    Description:

    If GL_AMD_vertex_shader_tessellator is enabled, fetch the "ith"
    element from the vertex buffer bound to the vertex array bound to
    the sampler.

Additions to Chapter 9 of the OpenGL Shading Language 1.20 Specification (Shading Language Grammar)

Additions to Chapter 10 of the OpenGL Shading Language 1.20 Specification (Issues)

Additions to the AGL/EGL/GLX/WGL Specifications

    None

Dependencies on ARB_vertex_shader

    ARB_vertex_shader is required.

Interactions with EXT_gpu_shader4

    If EXT_gpu_shader4 is not supported, remove all references to:

    __isamplerVertexAMD
    __usamplerVertexAMD
    ivec4 vertexFetchAMD
    uvec4 vertexFetchAMD

Interactions with EXT_geometry_shader4

    If EXT_geometry_shader4 is supported, change the last
    paragraph of Section 2.16, Geometry Shaders to:

    A program object that includes a geometry shader must also include
    a vertex shader; otherwise a link error will occur. If a program
    object that includes a geometry shader also includes a vertex shader
    with that has enabled GL_AMD_vertex_shader_tessellator, a link error
    will occur.

Interactions with AMDX_vertex_shader_tessellator

    This extension is symantically identical to the experimental
    AMDX_vertex_shader_tessellator. (It has been "promoted" to
    non-experimental status.)

Only the prefix AMDX has been changed to AMD.
Only the suffix AMDX has been changed to AMD.

We encourage applications and shader writers to migrate from AMDX to AMD. However, the AMDX entry points, enums, keywords and function names are not yet deprecated.

Errors

New State

Add to Table 6.5 Vertex Array Data

Get Value	Type	Get Command	Value	Description	Sec.	Attribute
TESSELLATION_FACTOR_AMD	R	GetFloatv	1.0	tessellation factor	2.8	vertex-array
TESSELLATION_MODE_AMD	Z_2	GetIntegerv	DISCRETE_AMD	tessellation mode	2.8	vertex-array

New Implementation Dependent State

None.

Sample Code

```
#extension GL_AMD_vertex_shader_tessellator : require

__samplerVertexAMD Vertex;
__samplerVertexAMD Normal;
__samplerVertexAMD Texcoord0;
__samplerVertexAMD Temperature;
__samplerVertexAMD Pressure;

attribute float myTemperature;

void main ( void )
{
    gl_Vertex = vec4( 0.0 );
    gl_Normal = vec4( 0.0 );
    gl_MultiTexCoord0 = vec4( 0.0 );
    myTemperature = 0.0;
    float myPressure = 0.0;  // Don't have to interpolate to attribute

    for ( int i=0; i<3; i++ )
    {
       float weight = gl_BarycentricCoord[i];

       gl_Vertex         += weight*vertexFetchAMD( Vertex,
gl_VertexTriangleIndex[i] );
       gl_Normal         += weight*vertexFetchAMD( Normal,
gl_VertexTriangleIndex[i] );
```

```
       gl_MultiTexCoord0 += weight*vertexFetchAMD( Texcoord0,
gl_VertexTriangleIndex[i] );
       myTemperature     += weight*vertexFetchAMD( Temperature,
gl_VertexTriangleIndex[i] ).x;
       myPressure        += weight*vertexFetchAMD( Pressure,
gl_VertexTriangleIndex[i] ).x;

    }
    // Rest of vertex shader goes here....
}
```

Issues

1) Does this belong conceptually in the pipe as subsuming geometry shader (after primitive combine) or vertex unpack.

   Vertex unpack. Even though there is "primitive information" it is limited to the immediate neighborhood.

2) Do we need a new stage?

   If we add a "tessellation" stage:

   Input to the tessellator is the unpacked vertex attributes, but each attribute is now an array of size 3, the "superprim" attributes, plus a barycentric coordinate.

   The output of the tessellator is the varying.

   The varying output of the tessellator then becomes the attributes input to the vertex shader.

   Alternatively, we can make the "unpack" part of the vertex shader responsibility.

   No. We'll just make the attributes undefined, and the "vertex unpack" stage naturally collapses into the vertex shader.

3) Why make attributes undefined but writable?

   This is the easiest way to have an unpack shader merged into existing shaders.

4) What variants of vertexFetch do we need.

   1D is probably all we need, and probably all we will ever need.
   The return types should be vec4, ivec4 and uvec4.
   So, we need:

```
   vec4  vertexFetchAMD( __samplerVertexAMD  sampler, int i );
   ivec4 vertexFetchAMD( __isamplerVertexAMD sampler, int i );
   uvec4 vertexFetchAMD( __usamplerVertexAMD sampler, int i );
```

5) How does __samplerVertexAMD and vertexFetchAMD interact with vertex arrays?

The __samplerVertexAMD becomes an active uniform. As existing samplers are bound to texture units, the samplerVertex is bound to a VertexAttrib array, and similarly, the "enable" of the VertexAttribArray is ignored. vertexFetchAMD will use the size, type, normalized and stride to fetch the "ith" element from the array as the following pseudocode:

```
if (generic vertex attribute j array normalization flag is set, and
    type is not FLOAT or DOUBLE)
  VertexAttrib[size]N[type]v
    (j, generic vertex attribute j array element i);
else
  VertexAttrib[size][type]v
    (j, generic vertex attribute j array element i);
```

6) What happens if a buffer object is not bound to an array?

There is no reason why it shouldn't work, but there's also no good reason to make it work. Undefined.

7) What about "conventional" OpenGL array state (Vertex, Color, Normal, TexCoord, etc....)?

By binding the buffer objects to the appropriate vertexAttrib array, and setting appropriate size, type, normalized and stride, the application programmer can access all "conventional" OpenGL array state?

8) Are attributes declared or used in the shader "active?"

For the purposes of GetActiveAttrib, GetAttribLocation and BindAttribLocation, no.

9) What about geometry shaders and tessellation?

Future hardware may relax this restriction, but you can not successfully link a program that includes a vertex shader that has enabled GL_AMD_vertex_shader_tessellator and a geometry shader.

10) What draw calls do we support?

To the shader writer, everything looks like indexed triangles or indexed quads, with discrete and continuous tessellation. These indexed triangles result from a polygon Begin/End object, a triangle resulting from a triangle strip, triangle fan, or series of separate triangles, or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or a Rect command.

Points, Lines and pixel rectangles and bitmaps are unsupported by a tessellation shader.

11) Do we need additional enables?

    Lets first see how "implicit" enable of vertex arrays and
    tessellation draw calls works. The first follows precedent (samplers
    override texture enable hierarchy.) The second seems to follow.

11) What about begin errors?

    They are evil, but I don't see how they can be avoided.
    Clearly sampler validation needs to follow precedent.

12) What about quads?

    Quads are necessary for subdivision surfaces such as Catmull-Clark.
    We have received several significant requests to support subdivision
    surfaces.

Revision History

    Revision 1, 2007-06-26 wwlk
    Preliminary review document

    Revision 2, 2007-08-16 wwlk
    Review document

    Correct spelling of "tessellate" throughout. Blush.
    Rename special variables.
    Add additional sampler types.
    Remove "1D" from sampler types and vertex fetches.
    Add core OpenGL api spec changes.
    Add interactions with EXT_gpu_shader4.
    Add many issues.
    Expanded example shader.

    Revision 3, 2007-08-17 wwlk
    Correct edit headers
    (OpenGL 1.5 -> OpenGL 2.0)
    (Shading Language 1.10 -> Shading Language 1.20)

    Revision 4, 2007-09-21 wwlk
    Fix typo in reserved keywords (remove "1D")
    Added support for all polygon calls, explicitly disallowing points
    lines and RasterPos,
    List additional BEGIN errors - yes they are evil.

    Revision 5, 2008-05-22 wwlk
    Add quad support

    Revision 6, 2009-03-05 wwlk
    General cleanup to ready for posting to repository

    Revision 7, 2009-03-05 wwlk
    Promote from AMDX to AMD.

Revision 8, 2009-03-06 wwlk
Minor update to enums section.
Cleaned up typos and <cr><lf>.