Catmull Clark Subdivision Demo White paper

Demo Overview

This demo shows how to use the AMD hardware tessellation unit to tessellate Bicubic Bezier patches. The demo is written in OpenGL and it implements the algorithm described in the paper "Approximating Catmull-Clark Subdivision Surfaces with Bicubic patches", by Charles Loop and Scott Schaefer. In this paper we will refer to this method ACC.

The demo program reads a .bez file that contains Bezier patches and their control points. The .bez file is generated through a tool called SubDToBezier that converts quad meshes in OBJ format into equivalent Bezier patches following the method described in the Charles Loop paper.

The tessellation program reads these Bezier patches from the .bez file and saves them in internal data structures for rendering. These patches are then tessellated on the hardware through a vertex shader program. The vertex shader implements the Bezier patch evaluation code.

ACC Implementation

The tool SubDToBezier implements the ACC algorithm, converting each OBJ quad into a Bicubic Bezier patch. In order to do this, the program finds the 1-ring neighborhood vertices for every OBJ quad. From this 1-ring neighbor and the masks mentioned in the ACC paper, the Bezier control points are computed for every quad. The 4 corner control points from this list become the 4 vertex positions of the patch to be rendered. These vertex positions, control points, and other per vertex data such as color, normal, texture coordinates, etc. are written to a binary file in the .bez format. Please refer to the Microsoft SubD10 sample document for an excellent discussion of the conversion process.

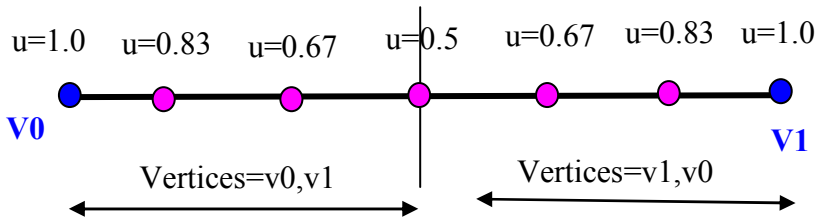
Hardware Tessellator

The AMD GPU is equipped with a hardware programmable tessellation unit. This allows a developer to subdivide low resolution polygon meshes based on any curved surface evaluation function such as Bezier surfaces, B-splines, NURBS, etc. Various surface techniques such as Displacement Mapping, Lighting, etc. can then be applied to the tessellated mesh.

Given a tessellation factor, the hardware tessellator tessellates the input primitive (quads in our case) into finer polygons by generating parametric coordinates: u, v, and w. These coordinates are accessed from a vertex shader and used for the Bezier patch evaluation.

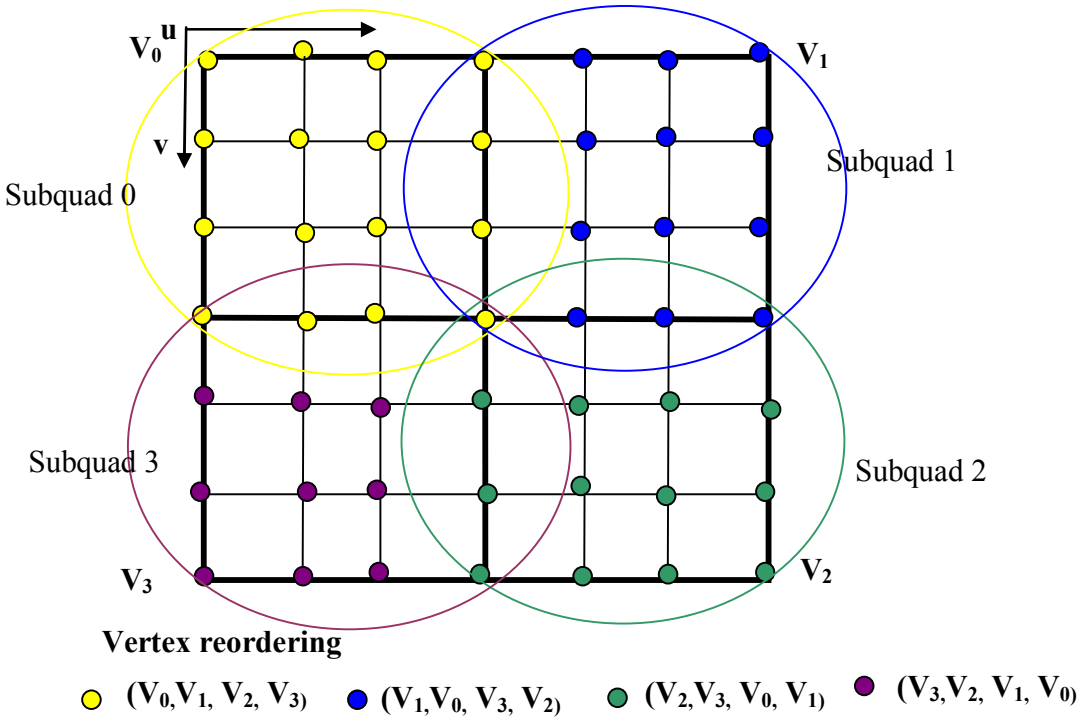
Watertightness

The hardware tessellator guarantees watertightness along polygon edges. In order to do this, the hardware reorders the incoming vertices whenever the generated (u, v) go past half way between the starting and the ending vertices. This ensures that the vertex evaluation for shared edges between neighboring primitives results in the same calculation. For example, when primitive type is line as shown below, vertices are ordered as below.



The evaluation shader should be written as pair of operations such as (A+B)+C instead of A + B + C. This guarantees that the evaluation shader results in same evaluation for shared vertices.

Similarly, when primitive type is quad as shown below, vertices are ordered as below.

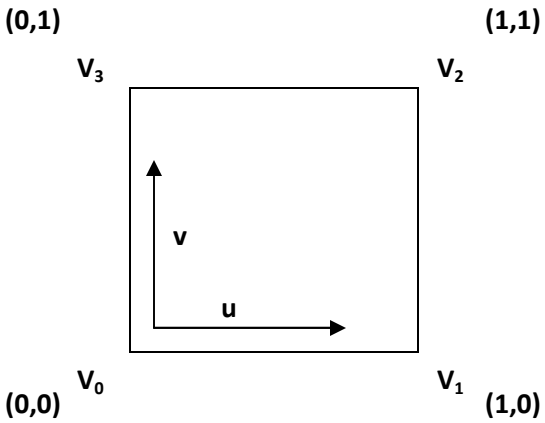


The direction of tensor coordinates (u,v) are as shown in the figure. The vertices of original primitive are reordered for each sub quad (shown in the figure above) in order to compute tessellated vertices. This is done so that shared vertices on common edge for neighboring primitive compute vertices exactly in the same way.

Vertex Shader

The vertex shader evaluates the Bezier patches. Since the hardware reorders the vertices, the (u,v) coordinates obtained in the shader are between 0.0 and 0.5. The vertex indices are also reordered. These parametric coordinates and the reordered indices can't be used for Bezier patch evaluation directly. Bezier patch evaluation requires the (u, v) coordinates to be between 0.0 and 1.0, and the vertices to remain in the same order for all surface points computation for that patch. Hence, the vertex shader takes care of these by mapping the (u,v) coordinates onto [0.0-1.0] range and undoing the vertex reordering.

The mapping of (u, v) coordinates onto the range [0.0-1.0] is done through bilinear interpolation of the (u, v) coordinates fetched in the vertex shader as shown below.



The above diagram shows a Bezier patch with vertices V0, V1, V2, V3, whose (u, v) coordinates should range as shown, for the purposes of patch evaluation. We define a constant table containing (u, v) values : [(0,0), (1,0), (1,1), (0,1)]. Then, for every Bezier surface point evaluation, this table is indexed four times with the current four vertex indices fetched from the hardware, to obtain four (u,v) values that form the end points of the bilinear interpolation. We then bilinearly interpolate the fetched (u, v) coordinates using these (u,v) values to obtain (u, v) coordinates in the range [0.0-1.0].

To undo the vertex reordering done by the hardware, we compare the fetched array of indices to see if they match any of the reordered indices (0,1,2,3), (1,0,3,2), (2,3,0,1), or (3,2,1,0). For any match found, we fetch the vertices using these indices, but always assign them to V0, V1, V2, and V3, in this order. This ensures that irrespective of the current reordered vertices, we are evaluating the surface points of the patch whose corners are always the same four vertices.

Each Bezier patch has 16 control points that need to be passed into the vertex shader for patch evaluation. We do this by creating a floating point texture of size 16 x N, where N is the number of patches in the entire scene. This texture is then passed into the vertex shader as a sampler2D, where it is looked up to obtain the row of control points corresponding to that patch. The ‘t’ coordinate is nothing but the row number that is found by dividing the current patch number by N to get a value between 0.0 and 1.0. Since N can be very large, the ‘t’ coordinate value obtained through this division can cause precision issues during texture lookup, especially on the border. To avoid this problem, we add an offset of 0.5/N to every ‘t’ so that we are sampling the texel at its center and not at its lower left corner (which is the default in OpenGL).

This demo program shows subdivision on Bicubic Bezier patches. But, due to the generality of the tessellation unit, any kind of patch evaluation can be performed inside the vertex shader. The hardware provides the (u,v) coordinates inside the vertex shader and it is left to the developer to use it the way he/she wants.

Performance Considerations

The advantages of using the AMD hardware tessellation unit are evident with huge significant performance gains in the area of GPU memory usage, GPU speed, and memory bandwidth. Since the hardware manages the newly tessellated points, the application is free from having to manage and store these new points. As a result, we use less memory bandwidth. The only time bandwidth is used due to tessellation is when the application wishes to get back the tessellated points. This can be achieved through the Transform Feedback mechanism of OpenGL 3.0.

The hardware tessellates the geometry every frame. For deforming geometry whose topology changes very frequently during runtime, using the hardware tessellator will also provide large improvements in speed. Again, this is because the hardware manages all of the newly tessellated points and there is no data being sent back and forth between the application and the tessellation pipeline.

Further Reading

1. "Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches", by Charles Loop and Scott Schaefer
2. "Bezier Patches", by Michael Skinner
3. Microsoft DirextX SubD10 Sample

DISCLAIMER AND ATTRIBUTION

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

Trademark Attribution