Implementation Details of Automatic Shader Simplification Using Surface Signal Approximation

Rui Wang* Xianjin Yang* Yazhen Yuan* Wei Chen* Kavita Bala† Hujun Bao*

*State Key Lab of CAD&CG, Zhejiang University †Cornell University

In this document, we provide more details on our automatic shader simplification approach. Several illustrative code examples and implementation details are given to show the simplification of shaders while applying our simplification rules. More code examples can be found in supplementary files with this paper.

1 Slicing Shaders

Once we have the AST and PDG of the fragment shader, we use dependence graph based approaches [Ottenstein and Ottenstein 1984; Jackson and Rollins 1994] to construct *shader slices*. In Code 1, we show an example shader program and its *shader slice* w.r.t the criterion $(n = 2, 16, x)$, where the input uniform parameter n is 2, the number 16 denotes the statement at line 16, and x is the slicing variable. As can be seen in the code, all computations that are not relevant to the final value of variable x are "sliced away".

2 Transforming Code across Shaders

The simplification rule of code transformation supports three types of code transformations: from the fragment shader to the vertex shader, to the geometry shader, and to the domain shader (the tessellation evaluation shader). In Code 2, we show an example that transforms a part of loop from the fragment shader to the vertex shader. The original vertex and fragment shader are shown in Code 2(a). Given the uniform parameter $n = 32$, the loop in the original fragment shader is first un-looped to 32 iterations. For each iteration, we compute its slice program, and then transform them to the vertex shader. In Code 2(b), we show a vertex and fragment shader example with 8 transformed iterations of computing variable c. In this example, changes of code on original shaders are highlighted in red. As can be seen in the code, after such a code transformation, 8 iterations in computing c is taken in the vertex shader and the resultant value is sent back to the rendering pipeline. In the following rasterization stage, these values from vertices are interpolated to each pixel. Then, in the fragment shader, the interpolated c value is continuously taken part in the loop until the remaining 24 iterations are all computed.

*e-mail:rwang, bao@cad.zju.edu.cn

3 Approximating Shader Slices using Bezier Triangles

Once we compute the control points of Bezier triangles and store them in an additional buffer, we use these Bezier triangle approximations to replace the original *shader slices* in shaders. Such a replacement mainly takes three steps. First, in the geometry shader, a function that creates barycentric coordinates of vertices is added [Bæentzen et al. 2008]. Then, the statement of the variable of the original *shader slice* is replaced by the interpolation from control points of Bezier triangle using the barycentric coordinates. Finally, a global code simplification is taken to eliminate all redundant statements w.r.t. the replaced *shader slices*.

In Code 3, we show an illustrative example applied with such a simplification rule. In Code 3(a), the fragment shader takes a simple calculation on the input variable c. We approximate the *shader slice*, (37, c), which is the statement of variable c at line 37, by Bezier functions. All control points are stored in an extra buffer array g_cp. At runtime, in the geometry shader, barycentric coordinates are created as vertex attributes for each triangle (Code 3(b) line 14 to 16), and emitted to the rendering pipeline (Code 3(b) line 17 to 22). In this way, while one triangle is rasterized in the rasterization stage, these barycentric coordinates at vertices are interpolated for pixels and output to the fragment shader. In the fragment shader, control points are first fetched from the buffer array, (Code 3(b) line 36) and then combined with the barycentric coordinate of this pixel to compute the approximation of value c from a reconstruction function, B, using Bernstein polynomials. In Code 3, all new code applying with the simplification rule is highlighted in red.

4 Subdividing Surfaces

In the runtime surface subdivision, we mainly take two steps. The first step is to add he subdivision code in the tessellation shader or in the geometry shader. In Figure 1, we give the subdivision patterns used in our method. In Code 4 and Code 5, we give two example templates used in our method to perform the runtime surface subdivision. Both templates given here perform two level subdivisions. In the tessellation hull shader, the $SV_TessFactor$ is used to control the tessellation on edges. In the geometry shader, we manually compute new vertices and emit them to form new triangles. In Code 5, a triangle is subdivided into 6 sub-triangles, (Figure 1(b)) with three triangle strips, each of which has four vertices.

The second step is to apply two simplification rules, the code transformation and the Bezier triangle approximation, on subdivided triangles. For the code transformation rule, transformed *shader slices* are inserted in certain positions in the template code, i.e., Code 4 line 28 or Code 5 line 17. For the Bezier triangle approximation, a unique sub-triangle ID is necessary for indexing correct control points for pixels in the fragment shader. We use the geometry shader to compute it. The example code is given in Code 6. The basic idea is to use the barycenter of each sub-triangle to compute the local index in the original triangle. Such local indices of sub-triangles are shown in numbers on each sub-triangle in Figure 1.

The input program

```
1  struct in{
2      int a;
3      int b;
4  }
5  void example(in param, uniform n){
6      param.b = param.b+1;
7      int i=0;
8      int x=0;
9      while (i<n){
10         sum = param.a+i;
11         x = param.a*i;
12     }
13     x = x+1;
14 }
```

Slice of x at line 16 with 2 iterations

```
1  struct in {
2      int a;
3      //int b;
4  }
5  void example(in param){
6      //param.b = param.b+1;
7      int i=0;
8      int x=0;
9      while (i<2){
10         //sum = param.a+i;
11         x = param.a*i;
12     }
13     x = x+1;
14 }
```

Code 1: *A shader slice example.*

The original vertex and fragment shaders

```
1  // Vertex Shader
2  struct VS_INPUT{
3      float c : c ;
4  };
5  struct VS_OUTPUT{
6      float c : c ;
7  };
8  VS_OUTPUT main (VS_INPUT vin){
9      VS_OUTPUT vout;
10
11
12
13
14     vout.c = vin.c;
15     return vout;
16 }
17 // Fragment Shader
18 struct VS_OUTPUT{
19     float c : c ;
20 };
21 struct PS_OUTPUT{
22     float4 color : SV_TARGET ;
23 };
24 PS_OUTPUT main (VS_OUTPUT pin, uint n){
25     PS_OUTPUT pout ;
26     float c = pin.c;
27     for (int i = 0; i < 32; i = i + 1){
28         c += pow(i,2);
29     }
30     pout.color = c;
31     return pout;
32 }
```

(a)

New vertex and fragment shaders with transformed loops

```
1  // Vertex Shader
2  struct VS_INPUT{
3      float c : c ;
4  };
5  struct VS_OUTPUT{
6      float c : c ;
7  };
8  VS_OUTPUT main (VS_INPUT vin){
9      VS_OUTPUT vout;
10     float c = vin.c;
11     for (int i = 0; i < 8; i = i + 1){
12         c += pow(i,2);
13     }
14     vout.c = c;
15     return vout;
16 }
17 // Fragment Shader
18 struct VS_OUTPUT{
19     float4 c : c ;
20 };
21 struct PS_OUTPUT{
22     float4 color : SV_TARGET ;
23 };
24 PS_OUTPUT main (VS_OUTPUT pin, uint n){
25     PS_OUTPUT pout ;
26     float c = pin.c;
27     for (int i = 8; i < 32; i = i + 1){
28         c += pow(i,2);
29     }
30     pout.color = c;
31     return pout;
32 }
```

(b)

Code 2: *An example of code transformation.*

The original fragment shader

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24 // Fragment Shader
25 struct VS_OUTPUT{
26     float c : c ;
27 };
28
29
30 struct PS_OUTPUT{
31     float4 color : SV_TARGET ;
32 };
33
34 PS_OUTPUT main(VS_OUTPUT pin){
35     PS_OUTPUT pout ;
36     float c = pin.c;
37     c += pow(c,2);
38     pout.color = c;
39     return pout;
40 }
```

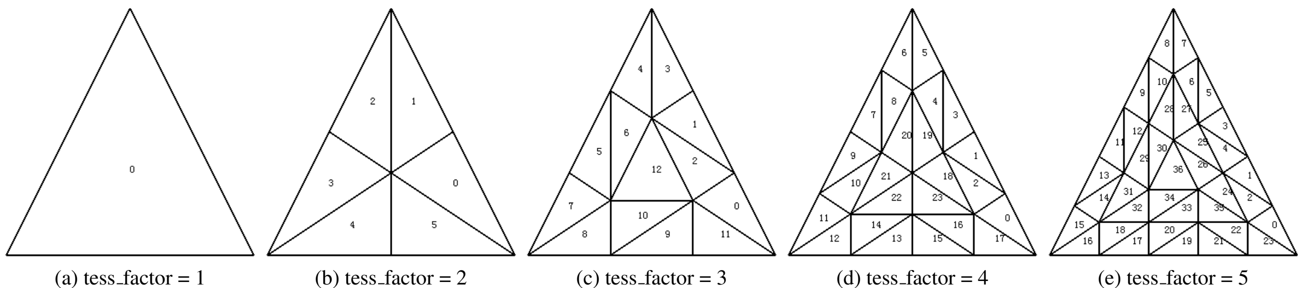
(a)

New shaders using Bezier triangle approximation

```
1  // Geometry Shader
2  struct GS_INPUT{
3     float4 pos :  SV_POSITION;
4  };
5  struct GS_OUTPUT{
6     float4 pos :  SV_POSITION ;
7     float2 uv :  BARYCENTRIC;
8     uint id :  PRIMITIVE_ID;
9  };
10 void main(inout TriangleStream<GS_OUTPUT> out,
11          triangle GS_INPUT in[3],
12          uint id:SV_PrimitiveID){
13    GS_OUTPUT p[3];
14    p[0].uv = float2(0.0, 0.0);
15    p[1].uv = float2(0.0, 1.0);
16    p[2].uv = float2(1.0, 0.0);
17    for(int i=0;i<3;++i){
18       p[i].pos = in[i].pos;
19       p[i].id = id;
20       out.Append(p[i]);
21    }
22    out.RestartStrip();
23 }
24 // Fragment Shader
25 struct GS_OUTPUT{
26    float4 pos :  SV_POSITION ;
27    float2 uv :  BARYCENTRIC;
28    uint id :  PRIMITIVE_ID;
29 };
30 struct PS_OUTPUT{
31     float4 color : SV_TARGET ;
32 };
33 AppendStructuredBuffer<ControlPoint> g_cp;
34 PS_OUTPUT main (GS_OUTPUT pin, ){
35     PS_OUTPUT pout ;
36     ControlPoint cp = g_cp[pin.id];
37     float c = B(cp,pin.uv);
38     pout.color = c;
39     return pout;
40 }
```

(b)

Code 3: *An illustrative example of Bezier triangle approximation.*



(a) tess_factor = 1 (b) tess_factor = 2 (c) tess_factor = 3 (d) tess_factor = 4 (e) tess_factor = 5

Figure 1: *Tessellation patterns used in our surface subdivision. The number on each sub-triangle is the local index used to compute the unique sub-triangle ID.*

```
// Hull Shader
struct hs_constant_data_output{
    float edges[3] : SV_TessFactor ;
    float inside : SV_InsideTessFactor ;
};
hs_constant_data_output constant_hs (
    InputPatch<VS_OUTPUT , 3> ip ,
    uint patch_id : SV_PrimitiveID){
    hs_constant_data_output output ;
    output.edges[0] = 2; output.edges[1] = 2;
    output.edges[2] = 2; output.inside = 2;
    return output;
}[patchconstantfunc("constant_hs")]
HS_OUTPUT main (InputPatch<VS_OUTPUT, 3> hs_in,
      uint i : SV_OutputControlPointID){
    HS_OUTPUT hs_out = hs_in;
    return hs_out;
}

// Domain Shader
struct hs_constant_data_output{
    float edges[3] : SV_TessFactor ;
    float inside : SV_InsideTessFactor ;
};
DS_OUTPUT main(hs_constant_data_output input ,
        float3 uvw : SV_DomainLocation ,
        const OutputPatch<HS_OUTPUT , 3> ds_in){
    //Insert transformed shader slice here
    ds_out = ds_in
    return ds_out;
}
```

Code 4: *A triangle subdivision template using tessellation shader.*

```
// Geometry Shader
[maxvertexcount(12)]
void main (triangle GS_INPUT gs_in[3] ,
    inout TriangleStream<GS_OUTPUT> out){
  // Declaration of new vertices
  GS_OUTPUT p0; GS_OUTPUT p1;   GS_OUTPUT p2;
  GS_OUTPUT p3; GS_OUTPUT p4; GS_OUTPUT p5;
  GS_OUTPUT p6;
  // Baycentric cooridantes of new vertices
  float3 p0_uvw = float3(0, 1, 0);
  float3 p1_uvw = float3(0, 0.5, 0.5);
  float3 p2_uvw = float3(0.333328, 0.333328, 0.333344);
  float3 p3_uvw = float3(0, 0, 1);
  float3 p4_uvw = float3(0.5, 0, 0.5);
  float3 p5_uvw = float3(1, 0, 0);
  float3 p6_uvw = float3(0.5, 0.5, 0);
  //Insert transformed shader slice here

  // Stream out new triangles
  out.Append(p0); out.Append(p1);
  out.Append(p2); out.Append(p3);
  out.RestartStrip();

  out.Append(p3); out.Append(p4);
  out.Append(p2); out.Append(p5);
  out.RestartStrip();

  out.Append(p5); out.Append(p6);
  out.Append(p2); out.Append(p0);
  out.RestartStrip();
}
```

Code 5: *A triangle subdivision template using geometry shader.*

After we have a local index, it is appended to a global offset ID of the original triangle to obtain the unique sub-triangle ID.

```
struct DS_OUTPUT{
  float3 uvw : BARYCENTRIC ;
};
struct GS_OUTPUT{
  uint id : PRIMITIVE ID;
};
void main(DS_OUTPUT gs_in,
    uint tri_id : SV_PrimitiveID){
  // Compute the barycenter of sub-triangle
  //  in the original triangle
  float3 p = compute_barycenter(gs_in.uvw);

  // Use the barycenter to compute the
  //  local and outer index
  int local_index = compute_local_index(p);
  int outer_index = compute_outer_index(p);

  // Use the local and outer index to compute
  //  the sub-triangle local ID in this triangle
  int tess_index = outer_triangles + local_index;

  // Append the local sub-triangle ID to the
  //  global ID of original triangle to get the
  //  unique sub-triangle ID.
  out.id = tri_id * num_subtris + tess_index;
}
```

Code 6: *The example code to compute a unique sub-triangle ID in the geometry shader.*

References

BÆENTZEN, J., NIELSEN, S., GJØL, M., AND LARSEN, B. 2008. Shader-based wireframe drawing. *Computer Graphics and Geometry 10*, 2, 66–79. Invited paper. Extended version of previous conference paper.

JACKSON, D., AND ROLLINS, E. J. 1994. A new model of program dependences for reverse engineering. *SIGSOFT Softw. Eng. Notes 19*, 5 (Dec.), 2–10.

OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. 1984. The program dependence graph in a software development environment. *SIGSOFT Softw. Eng. Notes 9*, 3 (Apr.), 177–184.