II. Direct3D Foundations
12. The Geometry Shader

Game Graphic Programming
Kyung Hee University
Software Convergence
Prof. Daeho Lee

Geometry Shader

- Assuming we are not using the tessellation stages, the geometry shader stage is an optional stage that sits between the vertex and pixel shader stages.
  - While the vertex shader inputs vertices, the geometry shader inputs entire primitives.
  - The three vertices of each triangle are input into the geometry shader, and the geometry shader outputs a list of primitives.
  - Unlike vertex shaders which cannot destroy or create vertices, the main advantage of the geometry shader is that it can create or destroy geometry.

Geometry Shader (1)

- General form of GS

```
[maxvertexcount(N)]
void ShaderName (
 PrimitiveType InputVertexType InputName [NumElements],
 inout StreamOutputObject<OutputVertexType> OutputName)
{
    // Geometry shader body…
    // Geometry shader is invoked per primitive.
}
```

VS → GS → PS

  - **InputVertexType**: Output type of VS
  - **OutputVertexType**: Input type of PS

- **PrimitiveType**: **point, line, triangle, lineadj, triangleadj**
- **maxvertexcount(N)**: **N** is the maximum number of vertices the geometry shader will output for a single invocation.

Geometry Shader (2)

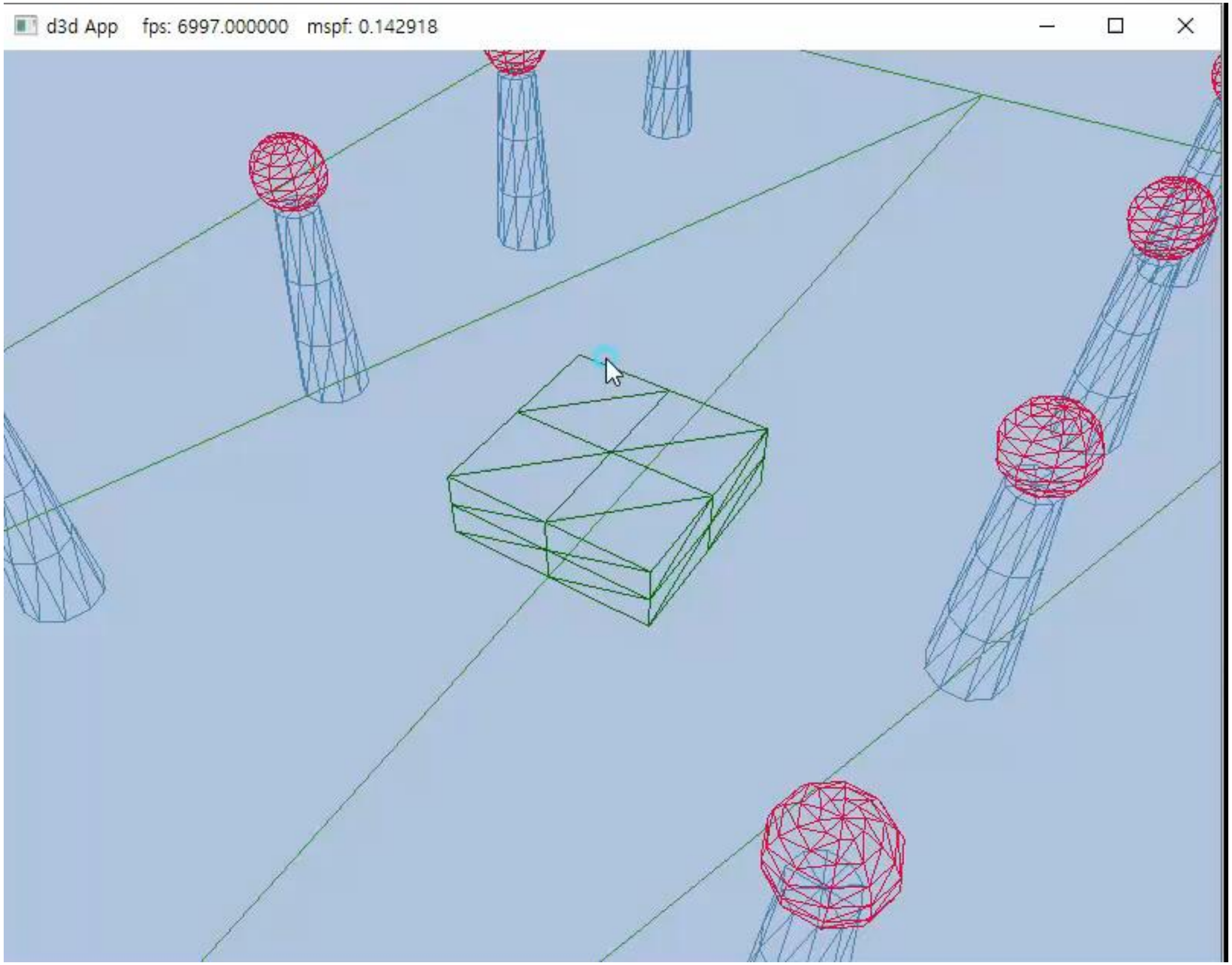
- Stream type
  - **`PointStream<OutputVertexType>`**: A list of vertices defining a point list.
  - **`LineStream<OutputVertexType>`**: A list of vertices defining a line strip.
  - **`TriangleStream<OutputVertexType>`**: A list of vertices defining a triangle strip.
  - For lines and triangles, the output primitive is always a strip. Line and triangle lists, however, can be simulated by using the intrinsic **`RestartStrip`** method:
    - **`void StreamOutputObject<OutputVertexType>::RestartStrip();`**

- The geometry shader can take a special unsigned integer parameter with semantic **`SV_PrimitiveID`**.
  - Per-primitive identifier automatically generated by the runtime.
  - **`SV_PrimitiveID`** can be written to by the geometry or pixel shaders, and read by the geometry, pixel, hull or domain shaders.

Geometry Shader (3)

- `[unroll]` attribute
  - `for` attribute
    - An optional parameter that controls how the statement is compiled.
  - This unrolls the loop until it stops executing.
  - This can optionally specify the maximum number of times the loop is to execute.
    - `[unroll(x)]`
  - `[loop]` attribute
    - This generates the codes that use flow control to execute each iteration of the loop.
  - The `[unroll]` and `[loop]` attributes are mutually exclusive and will generate compiler errors when both are specified.

Tessellation Using Geometry Shader (1)



Tessellation Using Geometry Shader (2)

```
// hlsl
struct VertexIn
{
   float3 PosL  : POSITION;
    float4 Color : COLOR;
};
/*struct VertexOut
{
   float4 PosH  : SV_POSITION;
    float4 Color : COLOR;
};*/
struct VertexOut
{
    float3 PosL : POSITION;
    float4 Color : COLOR;
};

struct GeoOut
{
    float4 PosH : SV_POSITION;
    float4 Color : COLOR;
};
```

Tessellation Using Geometry Shader (3)

```
// hlsl
void Subdivide(VertexOut inVerts[3], out VertexOut outVerts[6]) {
    VertexOut m[3];
    m[0].PosL = 0.5f * (inVerts[0].PosL + inVerts[1].PosL);
    m[1].PosL = 0.5f * (inVerts[1].PosL + inVerts[2].PosL);
    m[2].PosL = 0.5f * (inVerts[2].PosL + inVerts[0].PosL);

    m[0].Color = 0.5f * (inVerts[0].Color + inVerts[1].Color);
    m[1].Color = 0.5f * (inVerts[1].Color + inVerts[2].Color);
    m[2].Color = 0.5f * (inVerts[2].Color + inVerts[0].Color);

    outVerts[0] = inVerts[0];
    outVerts[1] = m[0];
    outVerts[2] = m[2];
    outVerts[3] = m[1];
    outVerts[4] = inVerts[2];
    outVerts[5] = inVerts[1];
};
```

```
//       1
//       *
//      / \
//     /   \
// m0*------*m1
//   / \   / \
//  /   \ /   \
// *-----*-----*
// 0     m2     2
```

Tessellation Using Geometry Shader (4)

```
// hlsl
void OutputSubdivision(VertexOut v[6], inout TriangleStream<GeoOut> triStream)
{
    GeoOut gout[6];

    [unroll]
    for (int i = 0; i < 6; ++i) {
        float4 PosW = mul(float4(v[i].PosL, 1.0f), gWorld);
        gout[i].PosH = mul(PosW, gViewProj);

        gout[i].Color = v[i].Color;
    }
   // We can draw the subdivision in two strips:
   //    Strip 1: bottom three triangles
   //    Strip 2: top triangle
    [unroll]
    for (int j = 0; j < 5; ++j) {
        triStream.Append(gout[j]);
    }
    triStream.RestartStrip();
    triStream.Append(gout[1]);
    triStream.Append(gout[5]);
    triStream.Append(gout[3]);
}
```

```
//       5
//       *
//      / \
//     /   \
//   1*-----*3
//   / \   / \
//  /   \ /   \
// *-----*-----*
// 0     2     4
```

Tessellation Using Geometry Shader (5)

```
// hlsl
[maxvertexcount(3)]
void GS1(triangle VertexOut gin[3],
   inout TriangleStream<GeoOut> triStream) {
    GeoOut gout[3];
    [unroll]
    for (int i = 0; i < 3; ++i)
    {
        float4 PosW = mul(float4(gin[i].PosL, 1.0f), gWorld);
        gout[i].PosH = mul(PosW, gViewProj);

        gout[i].Color = gin[i].Color;
    }
    [unroll]
    for (int j = 0; j < 3; ++j)
    {
        triStream.Append(gout[j]);
    }

}
```

Tessellation Using Geometry Shader (6)

```
// hlsl
[maxvertexcount(8)]   //  triStream.Append(…) × 8
void GS2(triangle VertexOut gin[3],
    inout TriangleStream<GeoOut> triStream) {
     VertexOut v[6];
     Subdivide(gin, v);
     OutputSubdivision(v, triStream);
}
VertexOut VS(VertexIn vin) {
    VertexOut vout;

     vout.PosL = vin.PosL;
     vout.Color = vin.Color;

     return vout;
}
float4 PS(GeoOut pin) : SV_Target
{
     return pin.Color;
}
```

Tessellation Using Geometry Shader (7)

```
void ShapesApp::BuildShadersAndInputLayout() {
   mShaders["standardVS"]
      = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr,
       "VS", "vs_5_1");
    mShaders["standardGS"]
      = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr,
       "GS1", "gs_5_0");
    mShaders["standardGS2"]
      = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr,
       "GS2", "gs_5_0");
   mShaders["opaquePS"]
      = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr,
       "PS", "ps_5_1");
// …
}
```

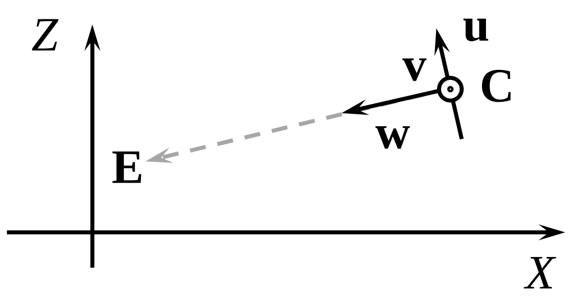
Tessellation Using Geometry Shader (8)

```
void ShapesApp::BuildPSOs() {
    D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc;
// …
    opaquePsoDesc.VS =   {
        reinterpret_cast<BYTE*>(mShaders["standardVS"]->GetBufferPointer()),
        mShaders["standardVS"]->GetBufferSize() };
     opaquePsoDesc.GS = {
      reinterpret_cast<BYTE*>(mShaders["standardGS"]->GetBufferPointer()),
      mShaders["standardGS"]->GetBufferSize() };
    opaquePsoDesc.PS = {
        reinterpret_cast<BYTE*>(mShaders["opaquePS"]->GetBufferPointer()),
        mShaders["opaquePS"]->GetBufferSize() };
// …
     D3D12_GRAPHICS_PIPELINE_STATE_DESC opaqueWireframePsoDesc =
opaquePsoDesc;
     opaqueWireframePsoDesc.GS = {
      reinterpret_cast<BYTE*>(mShaders["standardGS2"]->GetBufferPointer()),
      mShaders["standardGS2"]->GetBufferSize() };
// …
}
```

Overview (1)

Overview (2)

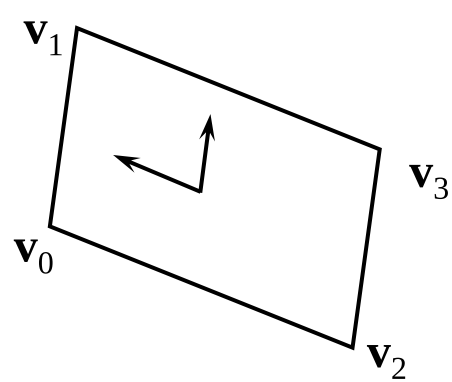
- The billboard technique is a method of rendering objects that always appear to face the camera.
  - When trees are far away, a billboard technique is used for efficiency.
    - Instead of rendering the geometry for a fully 3D tree, a quad with a picture of a 3D tree is painted on it.



$$\mathbf{w} = \frac{\left(E_x - C_x, 0, E_z - C_z\right)}{\left\|\left(E_x - C_x, 0, E_z - C_z\right)\right\|}$$

$$\mathbf{v} = (0,1,0)$$

$$\mathbf{u} = \mathbf{v} \times \mathbf{w}$$



$$\mathbf{v}_0 = \mathbf{C} + \frac{W}{2}\mathbf{u} - \frac{H}{2}\mathbf{v}$$

$$\mathbf{v}_1 = \mathbf{C} + \frac{W}{2}\mathbf{u} + \frac{H}{2}\mathbf{v}$$

$$\mathbf{v}_2 = \mathbf{C} - \frac{W}{2}\mathbf{u} - \frac{H}{2}\mathbf{v}$$

$$\mathbf{v}_3 = \mathbf{C} - \frac{W}{2}\mathbf{u} + \frac{H}{2}\mathbf{v}$$

HLSL (1)

```
struct VertexIn
{
   float3 PosW  : POSITION;
   float2 SizeW : SIZE;
};

struct VertexOut
{
   float3 CenterW : POSITION;
   float2 SizeW   : SIZE;
};

struct GeoOut
{
    float4 PosH    : SV_POSITION;
    float3 PosW    : POSITION;
    float3 NormalW : NORMAL;
    float2 TexC    : TEXCOORD;
    uint   PrimID  : SV_PrimitiveID;
};
```

HLSL (2)

```
VertexOut VS(VertexIn vin)
{
   VertexOut vout;

   // Just pass data over to geometry shader.
   vout.CenterW = vin.PosW;
   vout.SizeW   = vin.SizeW;

   return vout;
}
```

HLSL (3)

```
[maxvertexcount(4)]
void GS(point VertexOut gin[1], uint primID : SV_PrimitiveID,
   inout TriangleStream<GeoOut> triStream) {
   float3 up = float3(0.0f, 1.0f, 0.0f);
   float3 look = gEyePosW - gin[0].CenterW;
   look.y = 0.0f;
   look = normalize(look);
   float3 right = cross(up, look);

   float halfWidth  = 0.5f*gin[0].SizeW.x;
   float halfHeight = 0.5f*gin[0].SizeW.y;

   float4 v[4];
   v[0] = float4(gin[0].CenterW + halfWidth*right - halfHeight*up, 1.0f);
   v[1] = float4(gin[0].CenterW + halfWidth*right + halfHeight*up, 1.0f);
   v[2] = float4(gin[0].CenterW - halfWidth*right - halfHeight*up, 1.0f);
   v[3] = float4(gin[0].CenterW - halfWidth*right + halfHeight*up, 1.0f);

}
```

$$\mathbf{w} = \frac{(E_x - C_x, 0, E_z - C_z)}{\|(E_x - C_x, 0, E_z - C_z)\|}$$

$$\mathbf{v} = (0,1,0)$$

$$\mathbf{u} = \mathbf{v} \times \mathbf{w}$$

$$\mathbf{v}_0 = \mathbf{C} + \frac{W}{2}\mathbf{u} - \frac{H}{2}\mathbf{v}$$

$$\mathbf{v}_1 = \mathbf{C} + \frac{W}{2}\mathbf{u} + \frac{H}{2}\mathbf{v}$$

$$\mathbf{v}_2 = \mathbf{C} - \frac{W}{2}\mathbf{u} - \frac{H}{2}\mathbf{v}$$

$$\mathbf{v}_3 = \mathbf{C} - \frac{W}{2}\mathbf{u} + \frac{H}{2}\mathbf{v}$$

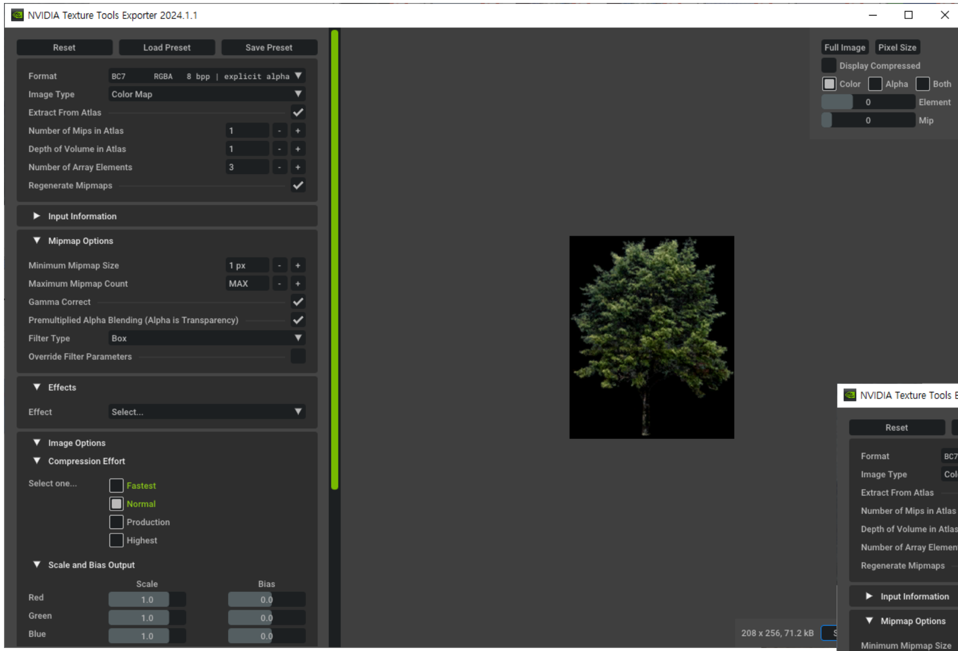
HLSL (4)

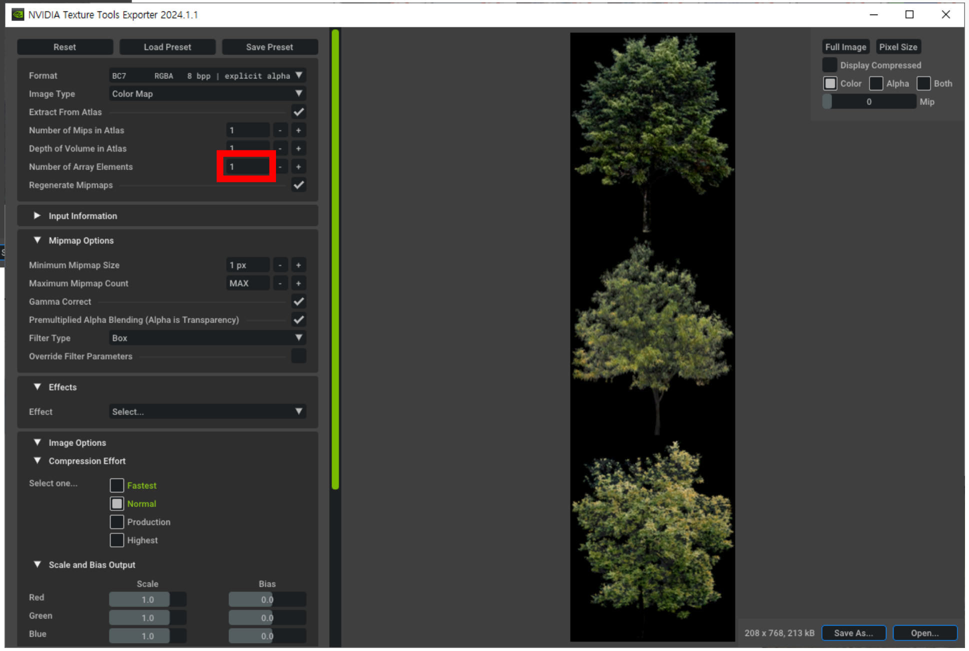
```
    float2 texC[4] = {
        float2(0.0f, 1.0f),                float2(0.0f, 0.0f),
        float2(1.0f, 1.0f),                float2(1.0f, 0.0f)
    };

    GeoOut gout;
    [unroll]
    for(int i = 0; i < 4; ++i) {
        gout.PosH     = mul(v[i], gViewProj);
        gout.PosW     = v[i].xyz;
        gout.NormalW  = look;
        gout.TexC     = texC[i];
        gout.PrimID   = primID;

        triStream.Append(gout);
    }
}
```

HLSL (5)

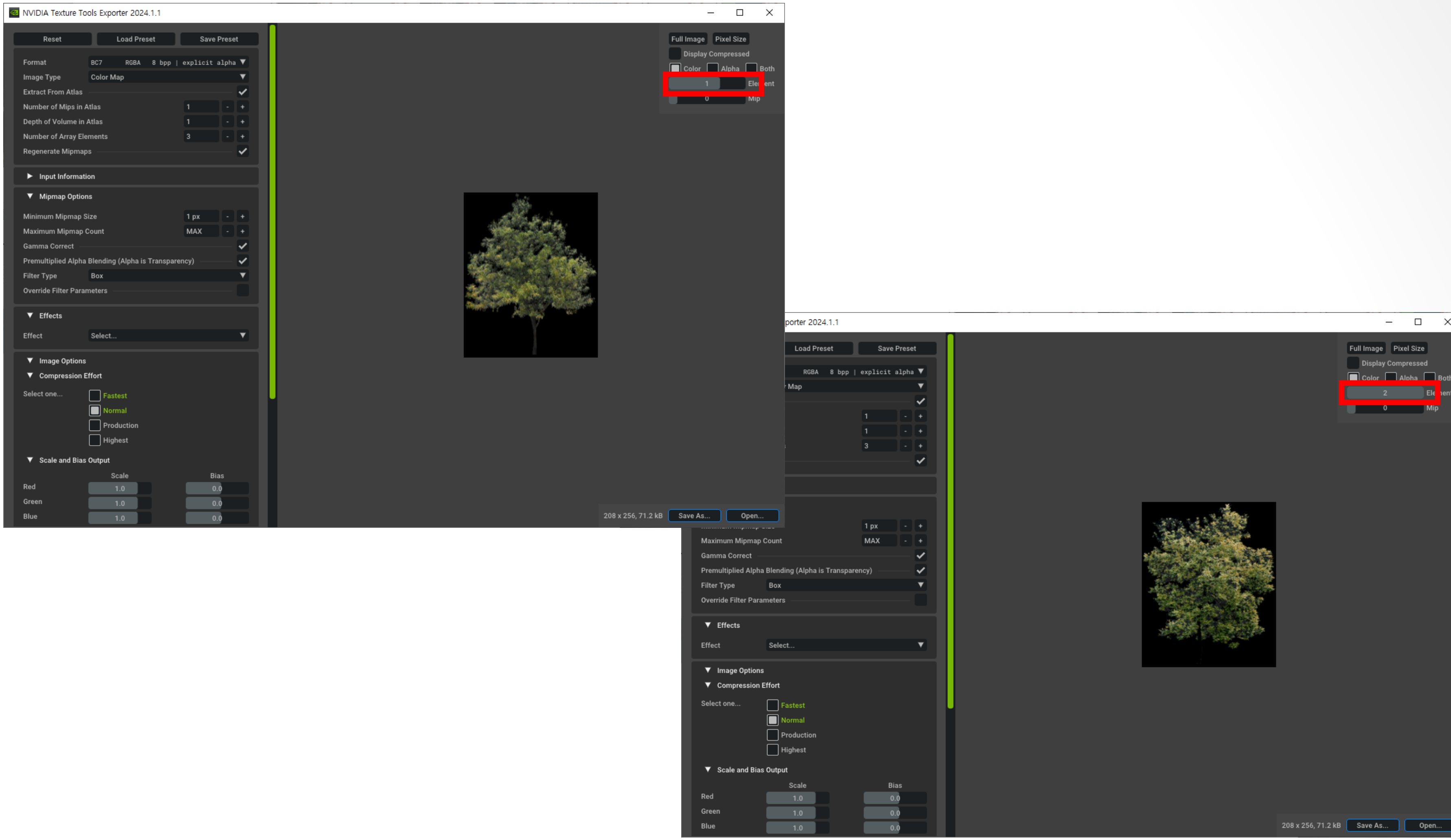
```
float4 PS(GeoOut pin) : SV_Target
{
    float3 uvw = float3(pin.TexC, pin.PrimID % 3);
                                  // texture array
    float4 diffuseAlbedo
      = gTreeMapArray.Sample(gsamAnisotropicWrap, uvw)
          * gDiffuseAlbedo;

// …
}
```

Texture Array (1)





Texture Array (2)



Kyung Hee University
nize@khu.ac.kr