Rendering Subdivision Surfaces Efficiently on the GPU

Gy. Antal, L. Szirmay-Kalos and L. A. Jeni

Department of Algorithms and their Applications, Faculty of Informatics, Eötvös Loránd Science University, Budapest
E-mail: gyantal@gmail.com

Abstract

Subdivision surfaces have become popular recently in computer graphics, because subdivision algorithms can easily provide smoother versions of an initially coarse base mesh of arbitrary topology. In this paper we present a GPU-based implementation of the Catmull-Clark subdivision surface scheme. This solution computes subdivisions in the geometry shader of Direct3D 10 compatible GPUs. This is not only much faster than doing it on the CPU, but it makes also possible for the CPU to work on other important tasks.

Categories and Subject Descriptors (according to ACM CCS) I.3.5 [Computer Graphics]: Curve, surface, solid, and object representations

1. Introduction

Subdivision surfaces are rapidly becoming popular in computer graphics since subdivision algorithms can easily provide coarser or smoother versions of a base mesh of arbitrary topology. It means that only the simple base mesh should be stored in the model and passed to the graphics API, but the rendered surface will still be smooth. However, GPU based subdivision refinement has not appeared in games so far, because of performance issues and the difficult integration of the subdivision tessellation into the incremental polygon rendering pipeline.

Our proposed solution computes subdivisions in the geometry shader unit of Direct3D 10 compatible GPUs. The recursions of subdivisions are unfolded into one single geometry shader run, so this is a single pass algorithm. It operates without loops and without auxiliary texture memory write, so the method is inherently fast. Each quad is sent as a point primitive down the pipeline and information on vertex locations and connected quads is read from prepared buffers by the geometry shader. The geometry shader subdivides each quad of the base mesh and directly generates an output vertex array encoding the triangle strips of the refined mesh. Since the current Direct3D 10 specification limits the number of vertices emitted per run, our algorithm cannot subdivide infinitely. Practically two subdivision levels are easily achievable, which provides high enough quality for games.

Our program uses the Catmull-Clark subdivision scheme [Catmull78], but other schemes, such as Loop, Butterfly, and Kobbelt can also be implemented similarly. The program also has features needed by games, like texture mapping and normal mapping (bumps). The method does not require pre-processing. We just have to upload buffers describing vertex locations, vertex indices of quads, and vertex indices of adjacent quads. These buffers can be prepared during the creation of the model.

2. Catmull-Clark Subdivision

Subdivision schemes take a mesh of large polygons, subdivide the polygons, and modify the position of the vertices to make the resulting mesh have a smoother appearance (Figure 1). The Catmull-Clark scheme is a particularly popular method, which produces quads in each level of subdivision. The base mesh can have arbitrary topology that is converted to a quad mesh in the first step.

Figure 1: *Original mesh and its level 1 and level 2 subdivisions applying the smoothing step once and twice, respectively.*

In this article we assume that the base mesh is also built of quads. This is not a serious limitation since should not the base mesh be a quad mesh, the first step is implemented separately, then the discussed algorithm can execute the further refinement steps.

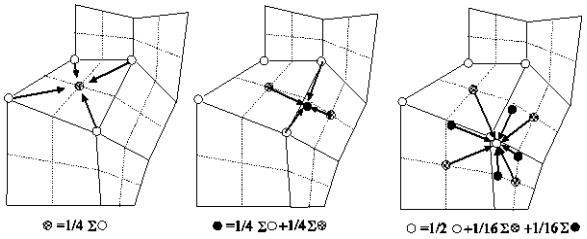

Figure 2: *One smoothing step of the Catmull-Clark subdivision scheme supposing valence 4 vertices. First the face points are obtained, then the edge midpoints are moved, and finally the original vertices are refined according to the weighted sum of its neighboring edge and face points.*

Let us consider a three-dimensional quadrilateral mesh (Figure 2). In the first step the Catmull-Clark scheme obtains face points as the average of the vertices of each face polygon. Then edge points are computed in the vicinity of the middle of the original edges by averaging the vertices at the ends of the edge and face points of polygons meeting at this edge. Finally, the original vertices are moved to the weighted average of the face points of those faces that share this vertex, and of edge points of those edges that are connected to this vertex. Connecting the edge points with the face points, we create a refined surface that has four times more quadrilaterals.

The location of the new face points, edge points, and the modified original vertices can also be directly expressed from the vertices of the original polygon and its neighboring polygons. The weights used to compute these points are shown in Figure 3. Note that the weights in the formula of the modified original vertices depend on how many edges meet in this particular vertex. The number of meeting edges is called the valence. Figure 3 shows the weights for valence values 3, 4, and 5, respectively. Vertices of valence greater than 5 are very rare in models. On the other hand, it is possible to convert these models to have vertices of at most valence 5. Thus we assume that valence 5 is the maximum and it is the modeller's responsibility to create the mesh accordingly. Actually, in theory, handling valence 5 vertices is not necessary either, because it can be proved that valence 5 vertices can be substituted by valence 3 and valance 4 vertices in every case. If we would have written our geometry shader handling only valence 3 and valence 4 cases, the shader would be much faster. However, we put this requirement for the convenience of the modeller, because valence 5 vertices generated quite often in the process of modelling.

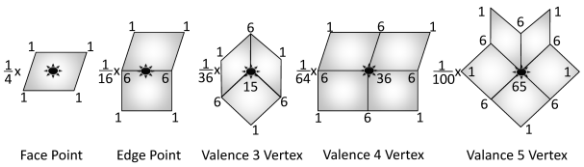

Figure 3: *Weighting of the original vertices for face points, edge points, and for modified original vertices, respectively. The formula of the positions of the modified original vertices depends on the valence of the vertex.*

Each subdivision step multiplies the number of quadrilaterals in the mesh by four. Thus, if the subdivision is executed on the CPU, then the data of the subdivided geometry uploaded on the GPU increases dramatically. To attack this problem, not only the rendering, but also the subdivision operation should be executed on the GPU. In the following section such a solution is presented.

3. Subdivision on the GPU

Subdivision algorithms take the base mesh and replace each polygon by a series of polygons. Unfortunately, this operation does not fit into earlier GPU architectures (Direct3D 9 or Shader Model 3 and earlier), because they were not able to modify the topology of the uploaded geometry, neither could they introduce new primitives in the rendering pipeline. Although Direct3D 9 or earlier GPUs cannot render subdivision surfaces in a single pass, they can execute subdivision algorithms in multiple passes [Bunnel05]. These implementations store the vertices in textures and recognize that subdivision schemes are very similar to filtering these geometry images. They refine the geometry stored in textures by the fragment shader and transform the refined vertices in the next pass by the vertex shader. Every refinement level needs a new pass. These methods require a lot of texture fetches and cumbersome offline preparation of the input data before sending the geometry to the pipeline.

A straightforward single pass implementation of subdivision algorithms has become possible with the introduction of the geometry shader in Direct3D 10 GPUs.

4. The Geometry Shader Approach for Subdivision Surfaces

In Direct3D 10 compatible GPUs, the geometry goes through the vertex shader, then the geometry shader processes primitives one by one and outputs an array of triangles that are rasterized later in the pipeline [Blythe06]. When a triangle is processed by the geometry shader, the information on adjacent triangles is also available. However, the Catmull-Clark scheme works with quads and not with triangles, and subdivision rules need information about adjacent quads. Direct3D does not support quads and neither does it provide the geometry shader with their required adjacency. To solve this problem, in our solution quads encoded as points are sent down the pipeline. The geometry shader assembles the subdivided mesh corre-

sponding to a single base mesh quad, and outputs the mesh as a triangle strip.

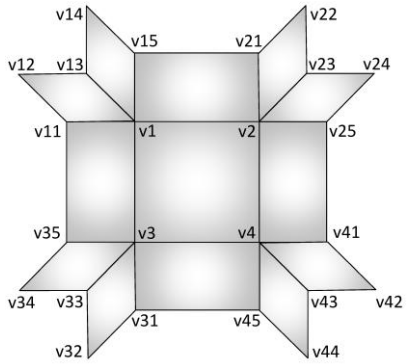

Figure 4: *Topology of the processed quad and its neighboring quads of the base mesh assuming valence 5 at each vertex.*

A single quad of vertices *v*1, *v*2, *v*3, *v*4 with adjacency information is shown in Figure 4. This figure corresponds to the case when all vertices have valence five, i.e. the most complicated case that we consider. If a vertex had valence four or three, then connected quads and their vertices would be missing from Figure 4. To describe topology, extra vertex indices of adjacent quads are also added to the quad information. The order of these extra vertex indices follow the order of quads connected only to *v*1, then to *v*2, etc. If vertex *v*1 has valence five, then two adjacent quads are connected only to *v*1, which are surrounded by five extra vertices *v*11, *v*12, *v*13, *v*14, and *v*15. These extra vertices are called the *ear* of *v*1. If the vertex had valence four or three, then the ear would have three or one vertices, respectively.

The CPU program sends a sequence of quad indices as a "vertex stream". The vertex shader simply passes these indices to the geometry shader. The geometry shader takes the quad index and addresses buffers to get the quad, vertex, and adjacency information. We use a `quadBuffer` that specifies the vertex indices of a given quad, a `vertPosBuffer` that stores vertex coordinates in modeling space, a `vertValenceBuffer` that gives the valence of a vertex, and finally an `adjacencyBuffer` that lists the vertex indices of the four ears of the given quad. The adjacency buffer always reserves space according to the valence five case, thus it allocates 20 items for each quad. These buffers and their organization are shown in Figure 5. The CPU is responsible for filling up these buffers and uploading them to the GPU memory.

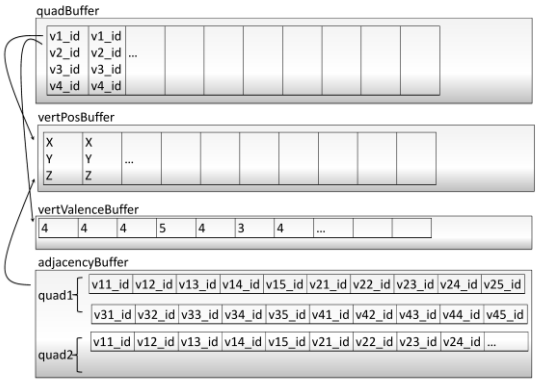

Figure 5: *Organization of quad, vertex, valence, and adjacency buffers.*

Using these buffers the geometry shader collects the vertices of the processed and its adjacent quads, applies subdivision rules, and outputs a vertex array encoding the triangle strips of the refined mesh. The algorithm executes one or more subdivisions directly. Note that neighboring quads in higher order subdivisions can be unambiguously defined by the vertices of the base mesh, thus the information input by the geometry shader is enough to produce arbitrary subdivisions.

In the following subsection we present the algorithm of zero and one subdivision step. Theoretically, multi-level subdivisions could be realized by the multiple execution of the single-level subdivision step. However, this requires loops and the storage of the temporary meshes. Thus we also present a direct solution that immediately provides a level 2 subdivision of the base mesh.

4.1. Level 0 subdivision

Level 0 subdivision means no subdivision at all. The geometry shader just assembles a triangle strip of two triangles defining a single base mesh quad. We included this trivial case here to highlight the responsibilities of the geometry shader, such as fetching the indexed vertices from buffers, transforming them from modeling space to clipping space, and outputting the quad in the form of a triangle strip. A simplified version of the geometry shader, which gets the index of the current quad in input variable `p_quadIndex`, is:

```
void Subdiv0Function(uint p_quadIndex,/*index of the quad*/
        inout TriangleStream<PS_INPUT> p_stream)
{
  PS_INPUT output;
  // get vertex indices of the quad
  uint4 quadInd = quadBuffer.Load(p_quadIndex);

  // get vertices
  float3 v1 = vertPosBuffer.Load(quadInd.x);
  float3 v2 = vertPosBuffer.Load(quadInd.y);
  float3 v3 = vertPosBuffer.Load(quadInd.z);
  float3 v4 = vertPosBuffer.Load(quadInd.w);

  // transform vertices to clipping space
  // and assemble a strip of two triangles
  output.Pos = mul(float4(v1, 1), mWorldViewProj);
  p_stream.Append(output);
  output.Pos = mul(float4(v3, 1), mWorldViewProj);
```

```
  p_stream.Append(output);
  output.Pos = mul(float4(v2, 1), mWorldViewProj);
  p_stream.Append(output);
  output.Pos = mul(float4(v4, 1), mWorldViewProj);
  p_stream.Append(output);
  p_stream.RestartStrip();
}
```

It may seem strange that points are transformed to clipping space by the geometry shader and not by the vertex shader. The explanation of this choice is that we send quad indices down the pipeline, and not vertices. Thus, the vertex shader cannot access vertex information directly.

4.2. Level 1 subdivision

Level 1 subdivision takes the quad of Figure 4 and generates two triangle strips encoding four quads of vertices *a*1, *a*2, …, *a*9 as shown by Figure 6.

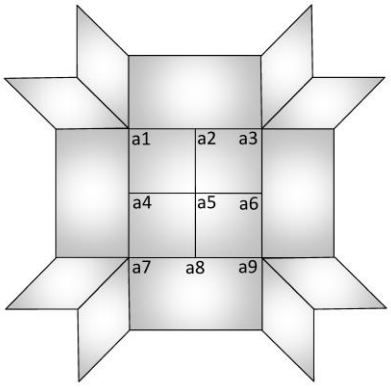

Figure 6: *Topology of the processed quad and its level 1 subdivision.*

The geometry shader calls function `GetEar()` that takes base index `iBase` of an ear and the valence of the vertex, and returns the vertices of the ear. In order to simplify the handling of different valence cases, the last vertex index of the ear is also stored in variable `vvLast`.

```
void GetEar(in uint iBase, in uint valence,
   out float3 vv1, out float3 vv2, out float3 vv3,
   out float3 vv4, out float3 vv5, out float3 vvLast)
{
  float3 vv1 =
     vertPosBuffer.Load(adjacencyBuffer.Load(iBase));
  float3 vv2 =
     vertPosBuffer.Load(adjacencyBuffer.Load(iBase + 1));
  float3 vv3 =
     vertPosBuffer.Load(adjacencyBuffer.Load(iBase + 2));
  float3 vv4 =
     vertPosBuffer.Load(adjacencyBuffer.Load(iBase + 3));
  float3 vv5 =
     vertPosBuffer.Load(adjacencyBuffer.Load(iBase + 4));
  float3 vvLast =
     vertPosBuffer.Load(adjacencyBuffer.Load(iBase
          +(valence - 3)*2));
}
```

Assuming only valence four vertices the geometry shader program of level 1 subdivision is as follows:

```
void Subdiv1Function(uint p_quadIndex, inout
         TriangleStream<PS_INPUT> p_stream)
{
  PS_INPUT output;
  // get vertex indices of the quad
  uint4 quadInd = quadBuffer.Load(p_quadIndex);
  // get the vertices
  float3 v1 = vertPosBuffer.Load(quadInd.x);
  float3 v2 = vertPosBuffer.Load(quadInd.y);
  float3 v3 = vertPosBuffer.Load(quadInd.z);
  float3 v4 = vertPosBuffer.Load(quadInd.w);
  // get valence values of the vertices
  uint val1 = vertValenceBuffer.Load(quadInd.x);
  uint val2 = vertValenceBuffer.Load(quadInd.y);
  uint val3 = vertValenceBuffer.Load(quadInd.z);
  uint val4 = vertValenceBuffer.Load(quadInd.w);
  // the base index of the vertices of a quad
  // in the adjacency buffer
  int quadBase = 20 * p_quadIndex;
  // get four ears one by one
  float3 v11, v12, v13, v14, v15, v1Last;
  GetEar(quadBase, val1, v11, v12, v13, v14, v15, v1Last);
  float3 v21, v22, v23, v24, v25, v2Last;
  GetEar(quadBase+5,val2, v21, v22, v23, v24, v25, v2Last);
  float3 v31, v32, v33, v34, v35, v3Last;
  GetEar(quadBase+10,val3,v31, v32, v33, v34, v35, v3Last);
  float3 v41, v42, v43, v44, v45, v4Last;
  GetEar(quadBase+15,val4,v41, v42, v43, v44, v45, v4Last);
  // new vertices: 1 face point and 4 edge points
  float3 a5 = (v1 + v2 + v3 + v4) / 4;
  float3 a2 = (6 * (v1 + v2) + v1Last + v21 + v3 + v4)/16;
  float3 a6 = (6 * (v2 + v4) + v2Last + v41 + v1 + v3)/16;
  float3 a8 = (6 * (v4 + v3) + v4Last + v31 + v1 + v2)/16;
  float3 a4 = (6 * (v3 + v1) + v3Last + v11 + v2 + v4)/16;
  // modification of the original vertices if valence is 4
  a1 = (36 * v1 + 6 * (v11 + v13 + v2 + v3) +
          v3Last + v12 + v21 + v4) / 64;
  a3 = (36 * v2 + 6 * (v21 + v23 + v1 + v4) +
          v1Last + v22 + v41 + v3) / 64;
  a7 = (36 * v3 + 6 * (v31 + v33 + v1 + v4) +
          v4Last + v32 + v11 + v2) / 64;
  a9 = (36 * v4 + 6 * (v41 + v43 + v2 + v3) +
          v2Last + v42 + v31 + v1) / 64;
  // optionally caculate texture,
  // normal, bi-normal data here . . .
  // transform vertices and emit 2 triangle strips
  output.Pos = mul( float4(a1, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a4, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a2, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a5, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a3, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a6, 1), mWorldViewProj );
  p_stream.Append(output);
  p_stream.RestartStrip();
  output.Pos = mul( float4(a4, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a7, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a5, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a8, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a6, 1), mWorldViewProj );
  p_stream.Append(output);
  output.Pos = mul( float4(a9, 1), mWorldViewProj );
  p_stream.Append(output);
  p_stream.RestartStrip();
}
```

If the program needs to be prepared not only for valence four but also for other valence values, the part entitled “modification of the original vertices” should be changed introducing several branches according to the actual valence. For example, the computation of vertex position a1 would look like this:

```
// modification of the original vertices
if (val1 == 3)
  a1 = (15*v1 + 6*(v11 + v2 + v3) +
          v3Last + v21 + v4)/36;
else if (val1 == 4)
  a1 = (36*v1 + 6*(v11 + v13 + v2 + v3) +
          v3Last + v12 + v21 + v4)/64;
else if (val1 == 5)
  a1 = (65*v1 + 6*(v11 + v13 + v15 + v2 + v3) +
          v3Last + v12 + v14 + v21 + v4)/100;
```

In addition to the position of the new vertices, their texture coordinate, normal vector, tangent, and bi-tangent should also be computed if we wish to present a textured

model with bump or displacement mapping. The linear interpolation of per-vertex data of the base mesh is the most efficient solution for generating this information for the new vertices.

4.3. Level 2 subdivision

Level 2 subdivision generates four triangle strips that correspond to 16 quads and 25 vertices, *b*1, *b*2, …, *b*25 (Figure 7). These vertices can be calculated from the original vertices *v*1,…, *v*4 and from the ears, similarly to the level 1 subdivision. However, the code is much longer; therefore we do not include it here. Our solution first calculates the vertices of level 1 subdivision (*a1*, *a2*, …, *a9*), that serves as an aid to determine the vertices of level 2 subdivision.

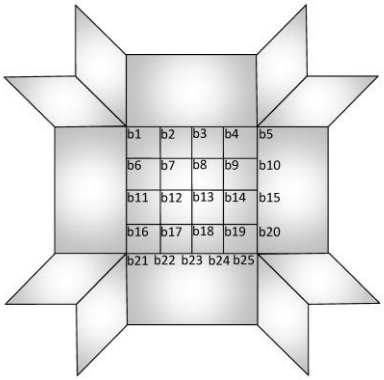

Figure 7: *Topology of the processed quad and its level 2 subdivision.*

4.4. Level 3 and more subdivisions

The strategy used for level 2 can be followed to deeper levels since higher order subdivisions can be unambiguously defined by the vertices of the neighboring quads that are available for the geometry shader. However, extending the subdivision level challenges the current output limit of the Direct3D 10 geometry shader. Currently maximum 1024 float (dword) values can be emitted in a geometry shader run. Level 3 subdivision generates 64 quads, i.e. 144 triangle strip vertices, which permits maximum 7 float values per vertex. However, a textured and normal mapped model would require a float4 position, a float2 texture coordinate pair, a float3 normal, and a float3 bi-tangent, i.e. 12 float values per vertex. So the practical limit of the subdivision is level 2, which provides enough quality for games. On the other hand, a level 3 subdivision would produce 64 times more quads (triangles) than the original base mesh, which is prohibiting in real-time applications anyway.

5. Implementation

The implementation is based on the August 2007 DirectX SDK using the latest NVIDIA driver 162.22. Texture mapping, normal mapping, and per-pixel lighting were used in the pixel shader to enhance the visual quality.

Figure 8 shows a lizard like creature rendered by the algorithm. The original base mesh has 1300 quads. Our measurements show that on an NVIDIA 8800GTX video card the level 0, level 1, and level 2 subdivisions are processed with 1073, 306, and 18.8 FPS, respectively. In comparison, we also implemented a CPU version of the same algorithm and run it on an Intel Core2 Duo/2.66 GHz computer. When the CPU executed level 1 subdivision, we obtained 7.7 FPS, which means that the GPU implementation is more than 39 times faster.

There are some implementation choices which can greatly affect the rendering speed. An important one is the AVOID_FLOW_CONTROL flag that can be used at the compilation of the shader. By default the HLSL flow control instructions (if, loop) are compiled to their assembly language counterparts. If the AVOID_FLOW_CONTROL flag is set, the flow control assembly instructions will be missing and both the "if" and the "else" branch will be executed sequentially on every GPU unit and they don't have to wait to each other for finishing the other branch of the "if" instruction. The 1073, 306, and 18.8 FPS rendering speed is achieved when the flow control assembly instructions were avoided. By default, when they are generated our measurement showed 1073, 122, and 22.1 FPS. This large difference suggests that there are some driver inefficiencies and it is expected that the speed of the GPU subdivision will get higher as Nvidia releases better drivers.

6. Conclusion

This article presents a geometry shader program to render Catmull-Clark subdivision surfaces assuming valence three, four, and five vertices. To solve the problem of accessing the adjacency information of quads, only the quad indices are sent down the pipeline and the geometry shader builds up the subdivided meshes from buffers. The algorithm renders level 1 and level 2 subdivisions efficiently.

Acknowledgments

The lizard creature model is the courtesy of Bay Raitt.

References

[Blythe06] David Blythe, "The Direct3D 10 System," in SIGGRAPH 2006 Proceedings, 2006, pp 724-734.

[Bunnel05] Bunnel, Michael, "Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping," in GPU Gems 2 (Edited by Matt Pharr), Addison Wesley, 2005, pp 109-122.

[Catmull78] Catmull, E. and Clark, J., "Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes," Computer Aided Design, Volume 10, 1978, pp 350-355.

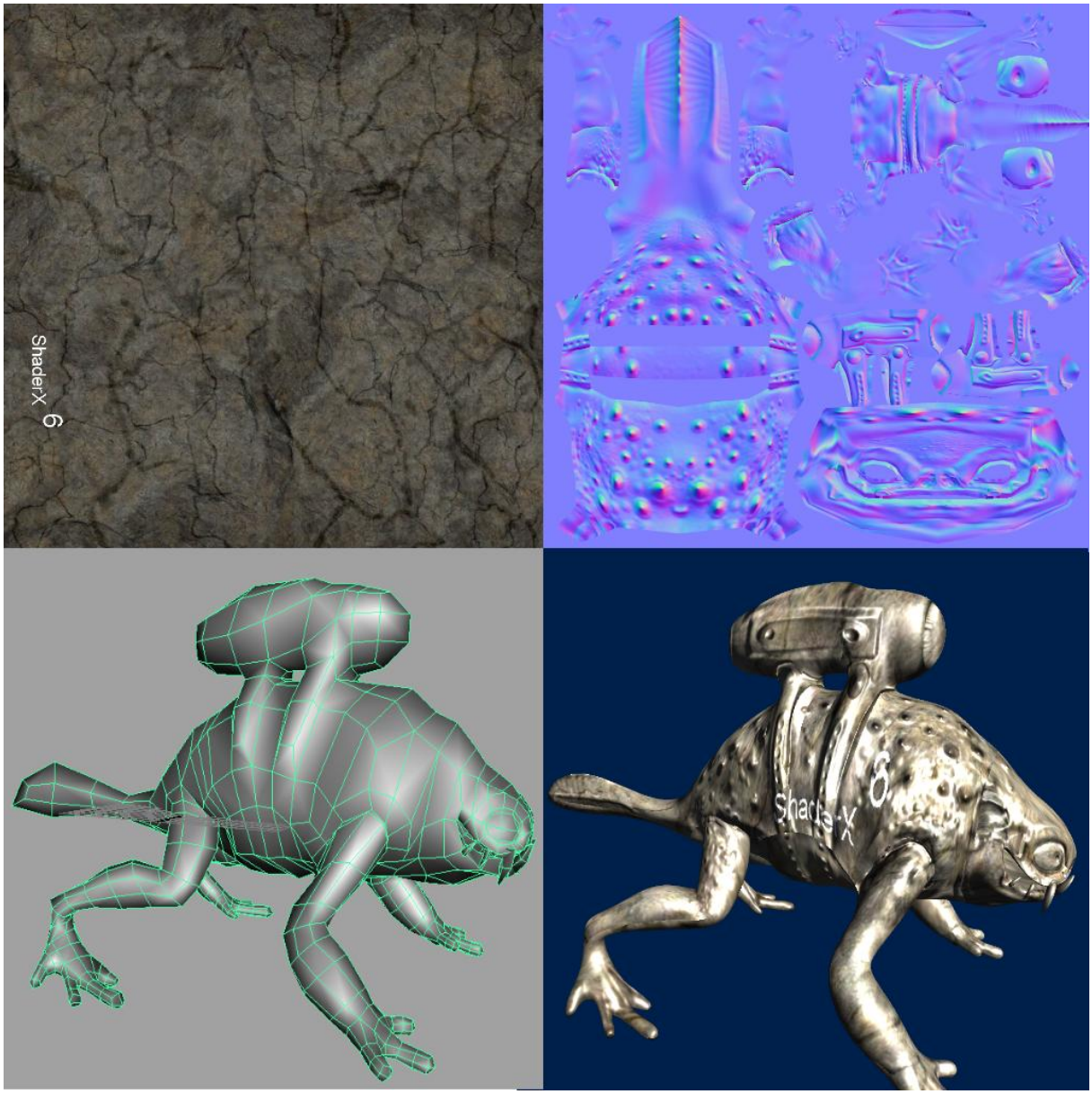

Figure 8: *The diffuse map, the normal map, the base mesh in the 3D editor, and the subdivided mesh rendered by the discussed algorithm.*