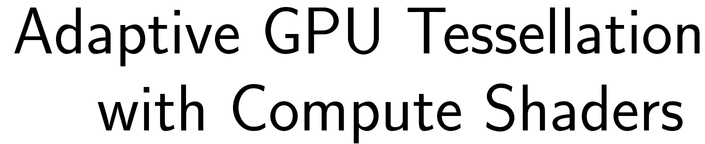


Adaptive GPU Tessellation with Compute Shaders

Jad Khoury, Jonathan Dupuy, and Christophe Riccio

1.1 Introduction

GPU rasterizers are most efficient when primitives project into more than a few pixels. Below this limit, the Z-buffer starts aliasing, and shading rate decreases dramatically [Riccio 12]; this makes the rendering of geometrically-complex scenes challenging, as any moderately distant polygon will project to sub-pixel size. In order to minimize such sub-pixel projections, a simple solution consists in procedurally refining coarse meshes as they get closer to the camera. In this chapter, we are interested in deriving such a procedural refinement technique for arbitrary polygon meshes.

Traditionally, mesh refinement has been computed on the CPU via recursive algorithms such as quadtrees [Duchaineau et al. 97, Strugar 09] or subdivision surfaces [Stam 98, Cashman 12]. Unfortunately, CPU-based refinement is now fundamentally bottlenecked by the massive CPU-GPU streaming of geometric data it requires for high resolution rendering. In order to avoid these data transfers, extensive work has been dedicated to implement and/or emulate these recursive algorithms directly on the GPU by leveraging tessellation shaders (see, e.g., [Niessner et al. 12, Cashman 12, Mistal 13]). While tessellation shaders provide a flexible, hardware-accelerated mechanism for mesh refinement, they remain limited in two respects. First, they only allow up to $\log_2(64) = 6$ levels of subdivision. Second, their performance drops along with subdivision depth [AMD 13].

In the following sections, we introduce a GPU-based refinement scheme that is free from the limitations incurred by tessellation shaders. Specifically, our scheme allows arbitrary subdivision levels at constant memory costs. We achieve this by manipulating an implicit (triangle-based) subdivision scheme for each polygon of the scene in a dedicated compute shader that reads from and writes to a compact, double-buffered array. First, we show how we manage our implicit subdivision scheme in Section 1.2. Then, we provide implementation details for rendering programs we wrote that leverage our subdivision scheme in Section 1.3.

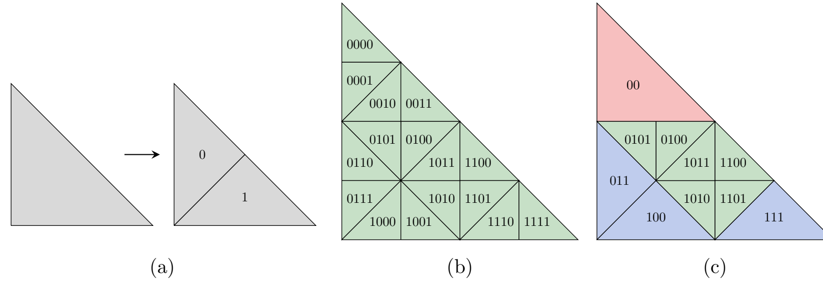


Figure 1.1. The (a) subdivision rule we apply on a triangle (b) uniformily and (c) adaptively. The subdivision levels for the red, blue, and green nodes are respectively 2, 3, and 4.

1.2 Implicit Triangle Subdivision

1.2.1 Subdivision Rule

Polygon refinement algorithms build upon a subdivision rule. The subdivision rule describes how an input polygon splits into sub-polygons. Here, we rely on a binary triangle subdivision rule, which is illustrated in Figure 1.1 (a). The rule splits a triangle into two similar sub-triangles 0 and 1, whose barycentric-space transformation matrices are respectively

$$M_0 = \begin{pmatrix} -1/2 & -1/2 & 1/2 \\ -1/2 & 1/2 & 1/2 \\ 0 & 0 & 1 \end{pmatrix}, \tag{1.1}$$

and

$$M_1 = \begin{pmatrix} 1/2 & -1/2 & 1/2 \\ -1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{pmatrix}. \tag{1.2}$$

Listing 1.1 shows the GLSL code we use to procedurally compute either M_0 or M_1 based on a binary value. It is clear that at subdivision level $N \geq 0$, the rule produces 2^N triangles; Figure 1.1 (b) shows the refinement produced at subdivision level $N = 4$, which consists of $2^4 = 16$ triangles.

```
mat3 bitToXform(in uint bit)
{
    float s = float(bit) - 0.5;
    vec3 c1 = vec3(    s, -0.5, 0);
    vec3 c2 = vec3(-0.5,    -s, 0);
    vec3 c3 = vec3(+0.5, +0.5, 1);

    return mat3(c1, c2, c3);
}
```

Listing 1.1. Computing the subdivision matrix M_0 or M_1 from a binary value.

1.2.2 Implicit Representation

By construction, our subdivision rule produces unique sub-triangles at each step. Therefore, any sub-triangle can be represented implicitly via concatenations of binary words, which we call a key. In this key representation, each binary word corresponds to the partition (either 0 or 1) chosen at a specific subdivision level; Figure 1.1 (b, c) shows the keys associated to each triangle node in the context of (b) uniform and (c) adaptive subdivision. We retrieve the subdivision matrix associated to each key through successive matrix multiplications in a sequence determined by the binary concatenations. For example, letting M_{0100} denote the transformation matrix associated to the key 0100, we have

$$M_{0100} = M_0 \times M_1 \times M_0 \times M_0. \tag{1.3}$$

In our implementation, we store each key produced by our subdivision rule as a 32-bit unsigned integer. Below is the bit representation of a 32-bit word, encoding the key 0100. Bits irrelevant to the code are denoted by the '_' character.

```
MSB                                     LSB
____ ____ ____ ____ ____ ____ ___1 0100
```

Note that we always prepend the key's binary sequence with a binary value of 1 so we can track the subdivision level associated to the key easily. Listing 1.2 provides the GLSL code we use to extract the transformation matrix associated to an arbitrary key.

```
mat3 keyToXform(in uint key)
{
    mat3 xf = mat3(1);

    while (key > 1u) {
        xf = bitToXform(key & 1u) * xf;
        key = key >> 1u;
    }

    return xf;
}
```

Listing 1.2. Key to transformation matrix decoding routine.

Since we use 32-bit integers, we can store up to a $32 - 1 = 31$ levels of subdivision, which includes the root node. Naturally, more levels require longer words. Because longer integers are currently unavailable on many GPUs, we emulate them using integer vectors, where each component represents a 32-bit wide portion of the entire key. For more details, see our implementation, where we provide a 63-level subdivision algorithm using the GLSL `uvec2` datatype.

1.2.3 Iterative Construction

Subdivision is recursive by nature. Since GPU execution units lack stacks, implementing GPU recursion is difficult. In order to circumvent this difficulty, we store the triangles produced by our subdivision as keys inside a buffer that we update iteratively in a ping-pong fashion; we refer to this double-buffer as the *subdivision buffer*. Because our keys consists of integers, our subdivision buffer is very compact. At each iteration, we process the keys independently in a compute shader, which is set to write in the second buffer. We allow three possible outcomes for each key: it can be subdivided to the next level, downgraded to the previous subdivision level, or conserved as is. Such operations are very straightforward to implement thanks to our key representation. The following bit representations match the parent of the key given in our previous example along with its two children:

```
         MSB                                   LSB
parent:  ____ ____ ____ ____ ____ ____ ____ 1010
key:     ____ ____ ____ ____ ____ ____ ___1 0100
child1:  ____ ____ ____ ____ ____ ____ __10 1000
child2:  ____ ____ ____ ____ ____ ____ __10 1001
```

Note that compared to the key representation, the other keys are either 1-bit expansions or contractions. The GLSL code to compute these representations is shown in Listing 1.3; it simply consists of bitshifts and logical operations, and is thus very cheap.

```
uint parentKey(in uint key)
{
    return (key >> 1u);
}

void childrenKeys(in uint key, out uint children[2])
{
    children[0] = (key << 1u) | 0u;
    children[1] = (key << 1u) | 1u;
}
```

Listing 1.3. Implicit subdivision procedures in GLSL.

Listing 1.4 provides the pseudocode we typically use for updating the subdivision buffer in a GLSL compute shader. In practice, if a key needs to be split, it emits two new words, and the original key is deleted. Conversely, when two sibling keys must merge, they are replaced by their parent's key. In order to avoid generating two copies of the same key in memory, we only emit the key once from the 0-child, identified using the test provided in Listing 1.5. We also provide some unit tests we perform on the keys to avoid producing invalid keys in Listing 1.6. For the keys that do not require any modification, they are simply re-emitted, unchanged.

```
buffer keyBufferOut { uvec2 u_SubdBufferOut[]; };
uniform atomic_uint u_SubdBufferCounter;

// write a key to the subdivision buffer
void writeKey(uint key)
{
    uint idx = atomicCounterIncrement(u_SubdBufferCounter);
    u_SubdBufferOut[idx] = key;
}

// general routine to update the subdivision buffer
void updateSubdBuffer(uint key, int targetLod)
{
    // extract subdivision level associated to the key
    int keyLod = findMSB(key);

    // update the key accordingly
    if (/* subdivide ? */ keyLod < targetLod && !isLeafKey(key)) {
        uint children[2]; childrenKeys(key, children);

        writeKey(children[0]);
        writeKey(children[1]);
    } else if (/* keep ? */ keyLod == targetLod) {
        writeKey(key);
    } else /* merge ? */ {
        if (/* is root ? */isRootKey(key)) {
            writeKey(key);
        } else if (/* is zero child ? */isChildZeroKey(key)) {
            writeKey(parentKey(key));
        }
    }
}
```

Listing 1.4. Updating the subdivision buffer on the GPU.

```
bool isChildZeroKey(in uint key)  { return (key & 1u == 0u); }
```

Listing 1.5. Determining if the key represents the 0-child of its parent.

```
bool isRootKey(in uint key) { return (key == 1u); }
bool isLeafKey(in uint key) { return findMSB(key) == 31; }
```

Listing 1.6. Determining whether a key is a root key or a leaf key.

It should be clear that our approach maps very well to the GPU. This allows us to compute adaptive subdivisions such as the one shown in Figure 1.1 (c). Note that an iteration only permits a single refinement or coarsening operation per key. Thus when more are needed, multiple buffer iterations should be performed. In our rendering implementations, we perform a single buffer iteration at the beginning of each frame.

1.2.4 Conversion to Explicit Geometry

For the sake of completeness, we provide here some additional details on how we convert our implicit subdivision keys into actual geometry. We achieve this easily with GPU instancing. Specifically, we instantiate a triangle for each subdivision key located in our subdivision buffer. For each instance, we determine the location of the triangle vertices using the routines of Listing 1.7. Note that these routines focus on computing the coordinates of the vertices of the subdivided triangles; extending them to handle other attributes such as normals or texture coordinates is straightforward.

```
// barycentric interpolation
vec3 berp(in vec3 v[3], in vec2 u)
{
    return v[0] + u.x * (v[1] - v[0]) + u.y * (v[2] - v[0]);
}

// subdivision routine (vertex position only)
void subd(in uint key, in vec3 v_in[3], out vec3 v_out[3])
{
    mat3 xf = keyToXform(key);
    vec2 u1 = (xf * vec3(0, 0, 1)).xy;
    vec2 u2 = (xf * vec3(1, 0, 1)).xy;
    vec2 u3 = (xf * vec3(0, 1, 1)).xy;

    v_out[0] = berp(v_in, u1);
    v_out[1] = berp(v_in, u2);
    v_out[2] = berp(v_in, u3);
}
```

Listing 1.7. Compute the vertices v_out of the sub-triangle associated to a subdivision key generated from a triangle defined by vertices v_in.

1.3 Adaptive Subdivision on the GPU

1.3.1 Overview

In this section, we describe a tessellation technique for polygonal geometry that leverages our implicit subdivision scheme. Our technique computes an adaptive subdivision for each polygon in the scene, so as to control their extent in screen-space and hence minimize sub-pixel projections; we describe how we compute such subdivisions using a distance-based LOD criterion in Section 1.3.2. Since adaptive subdivisions usually lead to T-junction polygons, we also discuss how we avoid them entirely; we discuss the issue of T-junctions in Section 1.3.3.

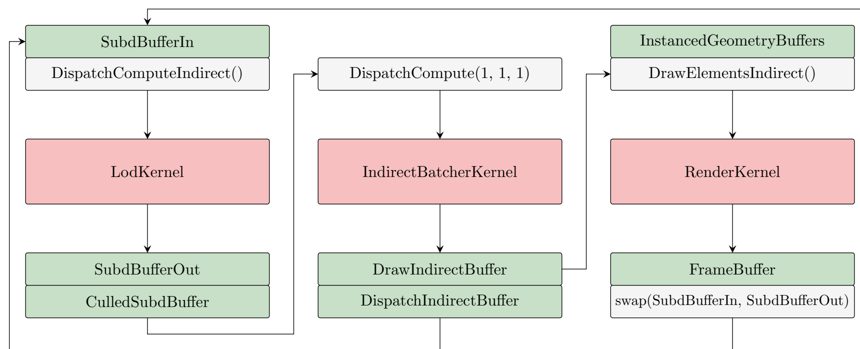


Figure 1.2. OpenGL pipeline of our compute-based tessellation shader. The green, red, and gray boxes respectively denote GPU memory buffers, GPU code execution, and CPU code execution.

In practice, our technique requires three GPU kernels with OpenGL 4.5; Figure 1.2 diagrams the OpenGL pipeline of our implementation. The first kernel (LodKernel) updates the subdivision buffer in a compute shader using the algorithms described in the previous section. In addition, we perform view-frustum culling for each key and write the visible ones to a buffer (CulledSubdBuffer) using an atomic counter. Next, we launch a second compute kernel (IndirectBatcherKernel) that prepares an indirect compute dispatch call for the next subdivision update (i.e., the next invocation of LodKernel), as well as an indirect draw call for the third and final kernel. The final kernel (RenderKernel) executes the indirect drawing commands to render the final geometry to the framebuffer (FrameBuffer). It instances a grid of triangles (InstancedGeometryBuffers) for each key located in the frustum-culled subdivision buffer (CulledSubdBuffer).

1.3.2 LOD Function

In order to guarantee that the transformed vertices produce rasterizer-friendly polygons, we rely on a distance-based criterion to determine how to update the subdivision buffer. Indeed, under perspective projection, the image plane size s at distance z from the camera scales according to the relation

$$s(z) = 2z \tan\left(\frac{\theta}{2}\right),$$

where $\theta \in (0, \pi]$ is the horizontal field of view. Based on this observation, we derive the following routine to determine the ideal subdivision level k that each key should target:

```
float distanceToLod(float z)
{
    float tmp = s(z) * targetPixelSize / screenResolution;
    return -log2(clamp(tmp, 0.0, 1.0));
}
```

Here, the parameter z denotes the distance from the camera to the subtriangle associated to the key being processed. Listing 1.8 provides the GLSL pseudocode we execute in LodKernel.

```
buffer VertexBuffer { vec3 u_VertexBuffer[];  };
buffer IndexBuffer  { uint u_IndexBuffer[];   };
buffer SubdBufferIn { uvec2 u_SubdBufferIn[]; };

void main()
{
    // get threadID (each key is associated to a thread)
    int threadID = gl_GlobalInvocationID.x;

    // get coarse triangle associated to the key
    uint primID = u_SubdBufferIn[threadID].y;
    vec3 v_in[3] = vec3[3](
        u_VertexBuffer[u_IndexBuffer[primID * 3    ]],
        u_VertexBuffer[u_IndexBuffer[primID * 3 + 1]],
        u_VertexBuffer[u_IndexBuffer[primID * 3 + 2]],
    );

    // compute distance-based LOD
    uint key = u_SubdBufferIn[threadID].x;
    vec3 v[3]; subd(key, v_in, v);
    float z = distance((v[1] + v[2]) / 2.0, camPos);
    int targetLod = int(distanceToLod(z));

    // write to u_SubdBufferOut
    updateSubdBuffer(key, targetLod);
}
```

Listing 1.8. Adaptive subdivision using a distance-based criterion.

1.3.3 T-Junction Removal

As for any other adaptive polygon-refinement scheme, our technique can produce T-junction triangles whenever two neighbouring keys differ in subdivision level. For instance, Figure 1.1 (c) shows a T-junction between the neighboring triangles associated to the keys 00, 0101 and 0100. T-junctions are problematic for rendering because they lead to visible cracks whenever the vertices are displaced by a smoothing function or a displacement map. Fortunately, our subdivision scheme has the property that it does not produce T-junctions as long as two neighboring keys differ by no more than one subdivision level; this is noticeable for the green and blue keys of Figure 1.1 (c). In order to guarantee such key configurations, we apply our distance-based criteria to the centroid of the hypotenuse of each sub-triangle; see Listing 1.8. We observed that this approach guarantees crack-free renderings for any target edge length lower than 16 pixels (we noticed some T-junctions above this value when the instanced grid is highly tessellated). Therefore, we chose to rely on such a system as it avoids the need for a sophisticated T-junction removal system; Listing 1.9 shows the code we use in the vertex shader of our RenderKernel.

```
buffer VertexBuffer { vec3 u_VertexBuffer[];  };
buffer IndexBuffer  { uint u_IndexBuffer[];   };
in vec2 i_InstancedVertex;
in uvec2 i_PerInstanceKey;

void main() {
    // get coarse triangle associated to the key
    uint primID = i_PerInstanceKey.y;
    vec3 v_in[3] = vec3[3](
        u_VertexBuffer[u_IndexBuffer[primID * 3    ]],
        u_VertexBuffer[u_IndexBuffer[primID * 3 + 1]],
        u_VertexBuffer[u_IndexBuffer[primID * 3 + 2]],
    );

    // compute vertex location
    uint key = i_PerInstanceKey.x;
    vec3 v[3]; subd(key, v_in, v);
    vec3 finalVertex = berp(v, i_InstancedVertex);

    // displace, deform, project, etc.
}
```

Listing 1.9. Adaptive subdivision using a distance-based criterion.

1.3.4 Results

To demonstrate the effectiveness of our method, we wrote a renderer for displacement-mapped terrains, and another one for meshes; our source code is available on github at https://github.com/jadkhoury/TessellationDemo, and a terrain rendering result is shown in Figure 1.3. In Table 1.1, we

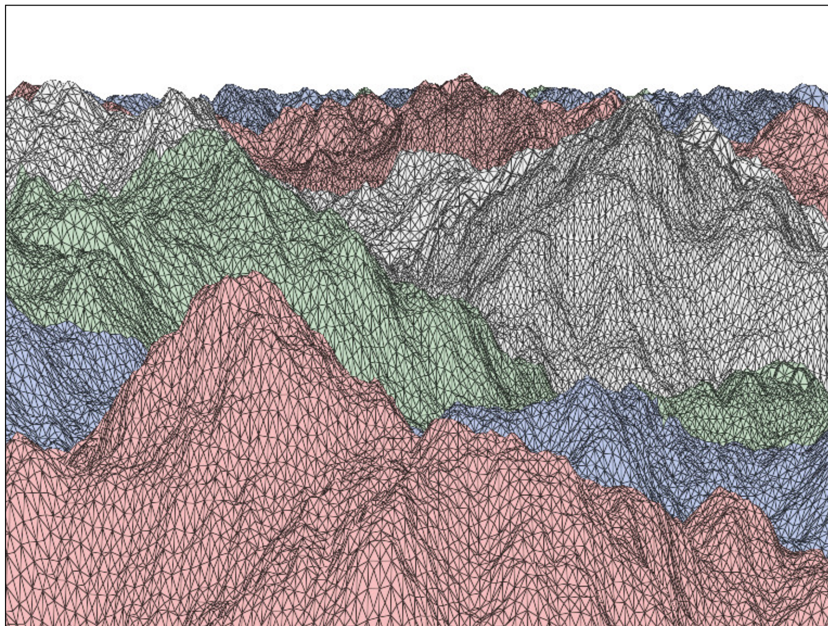
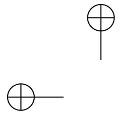
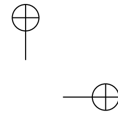


Figure 1.3. Crack-free, multiresolution terrain rendered entirely on the GPU using compute-based subdivision and displacement mapping. The alternating colors show the different subdivision levels.

give the CPU and GPU timings of a zoom-in/zoom-out sequence in the terrain at 1080p. The camera's orientation was fixed, looking downwards, so that the terrain would occupy the whole framebuffer, thus maintaining constant rasterizeration activity. We configured the renderer to target an average triangle edge length of 10 pixels; Figure 1.3 shows the wireframe of such a target. The testing platform is an Intel i7-8700k CPU, running at 3.70 GHz, and an NVidia GTX1080 GPU with 8GiB of memory. Note that the CPU activity only consists of OpenGL uniform variables and driver management. On current implementations, such tasks run asynchronously to the GPU.

Kernel	CPU (ms)	GPU (ms)	CPU stdev	GPU stdev
LOD	0.038	0.042	0.160	0.031
Batch	0.028	0.003	0.011	0.001
Render	0.035	0.184	0.018	0.013

Table 1.1. CPU and GPU timings and their respective standard deviation over a zoom-in sequence of 5000 frames.

As demonstrated by the reported numbers, the performance of our implementation is both fast and stable. Naturally, the average GPU rendering time depends on how the terrain is shaded. In our experiment, we use a constant color so that the reported performances correspond exactly to the overhead caused by vertex processing of our subdivision technique.

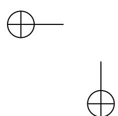
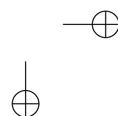
1.4 Discussion

We introduced a novel compute-based subdivision algorithm that runs entirely on the GPU thanks to an implicit representation. In future work, we would like to explore the feasibility of this representation for more complex subdivision schemes such as Catmull-Clark. In the meantime, we provide next a few additional considerations that we think can be relevant in the context of our work.

How much memory should be allocated for the buffers containing the subdivision keys? This depends on the target polygon density in screen space. The buffers should be able to store at least $3 \times \text{max_level} + 1$ nodes, and do not need to exceed a capacity of $4^{\text{max_level}}$ nodes. The lower bound corresponds to a perfectly restricted subdivision, where each neighboring triangle differ by one level of subdivision at most. The higher bound gives the number of cells at the finest level in case of uniform subdivision.

Is our subdivision technique prone to Floating-point precision issues ? There are no issues regarding the implicit subdivision itself, as each key is represented with bit sequences only. However, problems may occur when computing the transformation matrices in Listing 1.1. Our 31-level subdivision implementation does not have this issue, but higher levels will, eventually. A simple solution to delay the problem on OpenGL4+ hardware is to use double precision, which should provide sufficient comfort for most applications.

How about combining this technique with tessellation shaders to overcome the subdivision limits of the hardware ? We have actually implemented such an approach. Our open-source implementation is available on github at https://github.com/jdupuy/opengl-framework (see the *demo-isubd-terrain* demo). With both approached at hand, we leave it up to the developper to decide which approach is best given his software and hardware constraints.

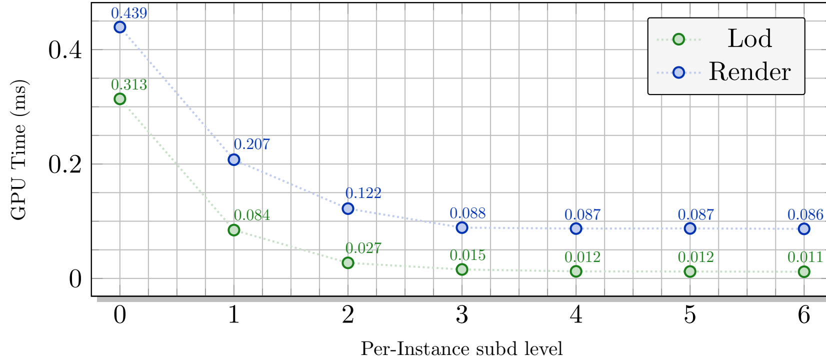


Figure 1.4. Performance evolution with respect to the level of subdivision of the instanced triangle grid on an NVidia GTX1080.

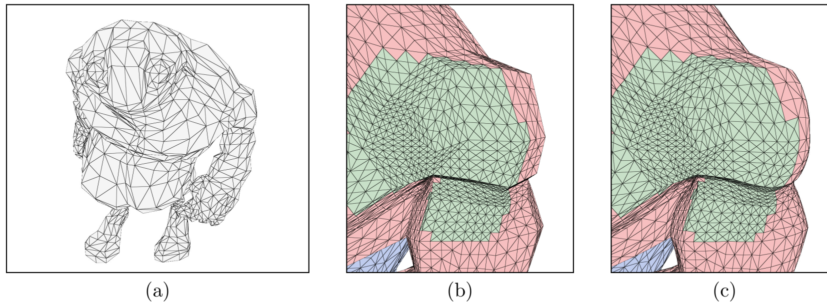


Figure 1.5. Our subdivision technique applied on (a) a triangle mesh using (b) bilinear interpolation and (c) Phong tessellation [Boubekeur and Alexa 08].

There are two ways to control polygon density. Either use the implicit subdivision, or refine the instanced triangle grid. Which approach is best? This will naturally depend on the platform. Our code provides tools to modify the tessellation of the instanced triangle grid, so that its impact can be thoroughly measured; Figure 1.4 plots the performance evolution that we measured on our platform.

Can our implicit subdivision scheme smooth input meshes? Our implicit subdivision scheme offers the same functionality as tessellation shaders. Therefore, any smoothing technique that runs with tessellation shaders run with our subdivision shaders. For instance, the mesh renderer we provide implements PN-triangles [Vlachos et al. 01] and Phong Tessellation [Boubekeur and Alexa 08] to smooth the surface of the coarse meshes we refine; Figure 1.5 shows our mesh renderer applying either bilinear interpolation or Phong Tessellation to a coarse triangle mesh.

1.5 Acknowledgments

This chapter is the result of Jad Khoury's master thesis, which was supervised by Jonathan Dupuy. All authors conducted this work at Unity Technologies.

Bibliography

[AMD 13] AMD. "GCN Performance Tweets.", 2013. List of all GCN performance tweets that were released during the first few months of 2013. Available online (http://developer.amd.com/wordpress/media/2013/05/GCNPerformanceTweets.pdf).

[Boubekeur and Alexa 08] Tamy Boubekeur and Marc Alexa. "Phong Tessellation." *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2008)* 27:5.

[Cashman 12] Thomas J. Cashman. "Beyond Catmull Clark? A Survey of Advances in Subdivision Surface Methods." *Comput. Graph. Forum* 31:1 (2012), 42–61. Available online (https://doi.org/10.1111/j.1467-8659.2011.02083.x).

[Duchaineau et al. 97] Mark Duchaineau, Murray Wolinsky, David E Sigeti, Mark C Miller, Charles Aldrich, and Mark B Mineev-Weinstein. "ROAMing terrain: real-time optimally adapting meshes." In *Proceedings of the 8th Conference on Visualization'97*, pp. 81–88. IEEE Computer Society Press, 1997.

[Mistal 13] Benjamin Mistal. "Gpu terrain subdivision and tesselation." *GPU Pro* 4 (2013), 3–20.

[Niessner et al. 12] Matthias Niessner, Charles Loop, Mark Meyer, and Tony Derose. "Feature-adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces." *ACM Trans. Graph.* 31:1 (2012), 6:1–6:11.

[Riccio 12] Christophe Riccio. "Southern Islands in deep dive.", 2012. SIGGRAPH Tech Talk. Available online (https://www.g-truc.net/doc/Siggraph2012%20Tech%20talk.pptx).

[Stam 98] Jos Stam. "Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values." In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '98*, pp. 395–404. New York, NY, USA: ACM, 1998. Available online (http://doi.acm.org/10.1145/280814.280945).

[Strugar 09] Filip Strugar. “Continuous distance-dependent level of detail for rendering heightmaps.” *Journal of graphics, GPU, and game tools* 14:4 (2009), 57–74.

[Vlachos et al. 01] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. “Curved PN Triangles.” In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, *I3D ’01*, pp. 159–166. New York, NY, USA: ACM, 2001. Available online (http://doi.acm.org/10.1145/364338.364387).