Semi-uniform Adaptive Patch Tessellation

C. Dyken[1,2], M. Reimers[1] and J. Seland[1,2]
[1]Centre of Mathematics for Applications, University of Oslo, Norway
[2]SINTEF ICT, Applied Mathematics, Norway

EARLY DRAFT
Final version to appear in Computer Graphics Forum

Abstract

We present an adaptive tessellation scheme for surfaces consisting of parametric patches. The resulting tessellations are topologically uniform, yet consistent and watertight across boundaries of patches with different tessellation levels. Our scheme is simple to implement, requires little memory, and is well suited for instancing, a feature available on current GPUs that allows a substantial performance increase. We describe how the scheme can be implemented efficiently and give performance benchmarks comparing it to some other approaches.

1 Introduction

Higher-order primitives like Bézier patches can model complex and smooth shapes parameterized by relatively few control points, and are easier to manipulate than triangular meshes with corresponding fidelity. Since current graphics hardware (GPUs) usually can not handle higher order primitives directly, these must be tessellated into triangle meshes before rendering. A standard approach is to tessellate the parameter domain of the patch, map the vertices of this triangulation to the patch surface, and render the resulting triangulation. A uniform tessellation results if each patch is tessellated in the same way. Otherwise, one can use an adaptive approach with varying tessellation levels. A complication with adaptive methods is that the tessellations of adjoining patches must match edge-by-edge to guarantee watertight rasterization [SAF*06].

The performance of GPUs rely heavily on massive parallelism, and thus to achieve good GPU utilization, geometry must be grouped into large batches. *Instancing* is a feature of DirectX 10-class GPUs that drastically increases batch sizes by letting multiple instances of an object be rendered using a single draw-call. A patch tessellation approach, as described above, is well suited for instancing if the number of different tessellations is low and patches with identical tessellations are easily grouped.

We present an adaptive patch tessellation scheme based on dyadic tessellations of the triangular parameter domain. To make adjacent patches meet edge-by-edge, we use a *snap function* that in one step moves patch boundary vertices and collapses triangles in a consistent manner. The effect of the snap function can be be interpreted as edge-collapsing [HDD*93]. In this way we can efficiently produce adaptive and topologically consistent tessellations from a limited set of source tessellation patterns. As an example, a maximum refinement level of $M = 5$ requires $(M+1)^3 = 216$ tessellation patterns to cover all combinations in our setting. We produce all of these on the fly using just 6 uniform tessellation patterns and a simple vertex shader snap routine. We also show how the patches can be grouped into render queues according to tessellation level on the GPU. Combined, this provides a pure GPU rendering pipeline for adaptive tessellation with efficient use of instancing.

After a brief account of previous work in the next section, we discuss uniform tessellations and our new semi-uniform adaptive approach in Section 3. In the subsequent section we elaborate on implementation and describe a pure GPU pipeline suitable for current graphics hardware. Section 5 is devoted to performance benchmarks, comparing a number of alternative implementations with and without instancing. We conclude the paper with a few final remarks.

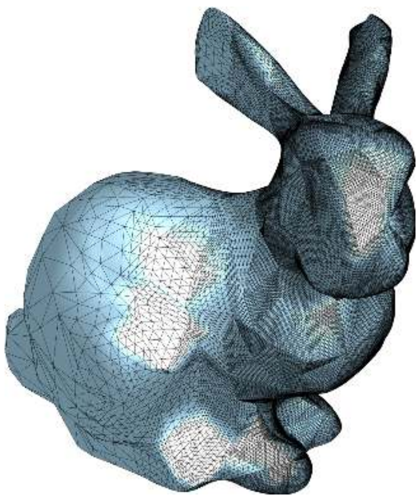
Figure 1: Semi-uniform adaptive tessellation of the Stanford bunny, with refinement level increasing from left to right.

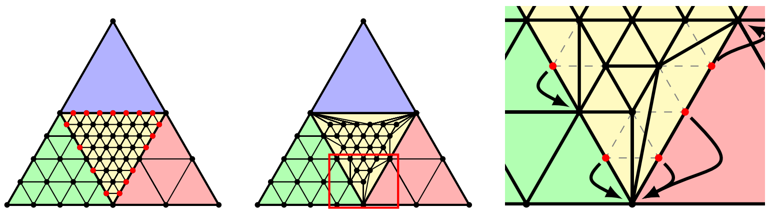
Figure 2: The left figure shows the case where patches with different tessellation levels meet, resulting in numerous hanging nodes (red) along the boundaries. The center triangulation is the result of our method, which amounts to moving hanging nodes to the nearest vertex common to both tessellations. The close-up to the right illustrates how the red vertices are moved by the snap function ϕ as indicated by the arrows. The dashed edges are either collapsed or moved as a result.

2 Related work

Boubekeur and Schlick [BS05] propose to store a parameter space tessellation in a vertex buffer object (VBO) on the GPU, letting the vertex shader evaluate the patch at the vertices. Their approach was extended by adaptive refinement patterns [BS08] that stores an atlas of tessellations covering all the different combinations of edge refinement levels in GPU memory, which allows patches with different refinement levels to meet edge-by-edge. However, the number of tessellations grows cubically with maximum tessellation level.

GPU based adaptive tessellation has been demonstrated for subdivision surfaces [SJP05, Bun05] and for grid based terrain models using Geometry Clipmaps [LH04]. These methods either yield tessellations with hanging nodes, or watertight tessellations by inserting triangles.

We have earlier proposed an adaptive tessellation scheme based on dyadic uniform tessellations with geometric continuity [DRS08]. The present scheme is to some extent similar but guarantees that tessellations are topologically consistent.

Moreton [Mor01] discusses adaptive tessellations and crack removal strategies, and proposes a forward differencing scheme for Bézier patches, aimed at hardware implementation. The GPU of the Xbox 360 [AB06] game console contains a hardware tessellation unit, and it is likely that standard graphics hardware will incorporate such units in the future. The *geometry shader* of DirectX 10-class hardware can in principle be used for tessellation. For reference, we have benchmarked a geometry shader-based implementation of our scheme.

3 Semi-uniform adaptive patch tessellation

A parametric triangular patch $\mathcal{F} : \mathcal{P}^0 \to \mathbb{R}^3$ can be rendered by evaluating $\mathcal{F}$ at the vertices of a tessellation $\mathcal{P}$ of its triangular parameter domain $\mathcal{P}^0 \subset \mathbb{R}^2$. If the resulting tessellation $\mathcal{T}$ in $\mathbb{R}^3$ is fine enough it can be rendered as a good approximation of the patch itself. A *dyadic* tessellation of $\mathcal{P}^0$ is a natural choice due to its uniformity and that it is trivial to produce on the fly, as demonstrated by our geometry shader-implementation. A dyadic refinement $\mathcal{P}^1$ of $\mathcal{P}^0$ can be obtained by inserting a vertex at the midpoint of each edge of $\mathcal{P}^0$ and replacing the triangle with four new triangles. Continuing the refinement procedure, we get at the m'th step a triangulation $\mathcal{P}^m$ of the m-dyadic barycentric points

$$\mathbb{I}^m = \left\{ \frac{1}{2^m}(i,j,k) \ : \quad i,j,k \in \mathbb{N}, \quad i+j+k = 2^m \right\}.$$

Note that dyadic tessellations of $\mathcal{P}^0$ are nested in the sense that $\mathbb{I}^m \subset \mathbb{I}^{m+1}$, i.e. a vertex of a coarse tessellation is also a vertex of its refinements. A dyadic tessellation $\mathcal{P}^m$ of the patch parameter domain yields a corresponding tessellation $\mathcal{T}^m$ of the patch itself, consisting of the triangles $[\mathcal{F}(\mathbf{u}_i), \mathcal{F}(\mathbf{u}_j), \mathcal{F}(\mathbf{u}_k)]$ for triangles $[\mathbf{u}_i, \mathbf{u}_j, \mathbf{u}_k]$ of $\mathcal{P}^m$. This approach lends itself naturally to the use of VBOs and vertex shader programs for patch evaluations.

We are interested in the case where we have a number of triangular patches that meet with geometric continuity, i.e. with common boundaries. We wish to construct one tessellation for each patch such that the individual tessellations are sufficiently fine. Allowing this form of adaptivity faces us with the problem of making the tessellations of neighboring patches compatible. One approach is to add triangles on the patch boundaries in order to fill holes in the tessellations, resulting in a slightly more complex mesh than in the case of uniform tessellations. Another approach is to let two tessellations meet with geometric continuity [DRS08, LH04]. However, this results in *hanging nodes* as illustrated in Figure 2 (left), which can result in artifacts such as drop-out pixels. To guarantee a watertight tessellation one must ensure that the patch tessellations are *topologically* consistent, i.e. that adjacent triangles share the end-points of their common edge.

Consider dyadic tessellations as the ones in Figure 2 (left), with neighboring patches of different tessellation levels and thus a number of hanging nodes. Our approach is to move each hanging node to the nearest dyadic barycentric point shared by the neighboring tessellations, as illustrated in Figure 2 (center). This results in a new tessellation that is uniform in the interior and topologically consistent with neighboring tessellations, although with degenerate triangles. Since the resulting mesh is in fact still topologically uniform it can be rendered using a VBO corresponding to a dyadic tessellation $\mathcal{P}^m$. Degenerate triangles pose no problems as they are quickly discarded in the rendering process. We next discuss the details of our approach.

We consider a single patch to be tessellated at level m and denote by p_0, p_1, p_2 the tessellation levels of its neighboring patches. In order to remove hanging nodes we define a *dyadic snap function* $\phi : \mathbb{I}^m \to \mathbb{I}^m$ which maps $\mathbf{u} = (u_0, u_1, u_2) \in \mathbb{I}^m$ to the nearest dyadic barycentric point consistent with neighboring tessellations, illustrated in Figure 2 (right). More precisely,

$$\phi(\mathbf{u}) = \begin{cases} (u_0, u_1, u_2), & \text{if } u_0, u_1, u_2 \neq 0; \\ (\sigma_{p_i}(u_0), \sigma_{p_i}(u_1), \sigma_{p_i}(u_2)), & \text{if } u_i = 0, \end{cases}$$

where σ_p maps a real value to the nearest p-dyadic number,

$$\sigma_p(t) = \frac{1}{2^p} \begin{cases} \lceil 2^p t - 1/2 \rceil & \text{if } t < 1/2; \\ \lfloor 2^p t + 1/2 \rfloor & \text{otherwise,} \end{cases} \tag{1}$$

breaking ties towards 0 if $t < 1/2$ and towards 1 otherwise. Thus, for a tie with $\mathbf{u}$ being equally distant from two p_i-dyadic points, ϕ snaps towards the closest corner vertex of $\mathcal{P}^0$, yielding some degree of symmetry.

Let us verify that ϕ is well defined and works as required. If $u_0, u_1, u_2 \neq 0$, then $\phi(\mathbf{u}) = \mathbf{u}$, i.e. interior vertices of $\mathcal{P}^m$ are preserved. Since $\sigma_p(i) = i$ for any integers i and p, the corners of $\mathcal{P}^0$ are left unchanged by ϕ. This also implies that ϕ is well defined even if two of the coordinates (u_0, u_1, u_2) are zero. Suppose now that $\mathbf{u} \in \mathbb{I}^m$ is a boundary vertex, i.e. with some $u_i = 0$. Since $\sigma_p(t) + \sigma_p(1-t) = 1$ for any integer p and real value t, then $\sigma_{p_i}(u_0) + \sigma_{p_i}(u_1) + \sigma_{p_i}(u_2) = 1$ and hence $\phi(\mathbf{u}) \in \mathbb{I}^{p_i}$. Note that this holds for all integers p_i. Therefore $\phi(\mathbf{u}) \in \mathbb{I}^m$ and in the case $\mathbf{u}$ is on the i'th boundary edge, $\phi(\mathbf{u}) \in \mathbb{I}^{p_i}$. In conclusion, ϕ preserves interior vertices and snaps a hanging boundary vertex to the closest dyadic barycentric coordinate at the required tessellation level.

Applying ϕ to all the vertices of $\mathcal{P}^m$ we obtain a corresponding planar tessellation of $\mathcal{P}^0$,

$$\mathcal{P}^m_{p_0 p_1 p_2} = \{[\phi(\mathbf{u}_i), \phi(\mathbf{u}_j), \phi(\mathbf{u}_k)] : [\mathbf{u}_i, \mathbf{u}_j, \mathbf{u}_k] \in \mathcal{P}^m\},$$

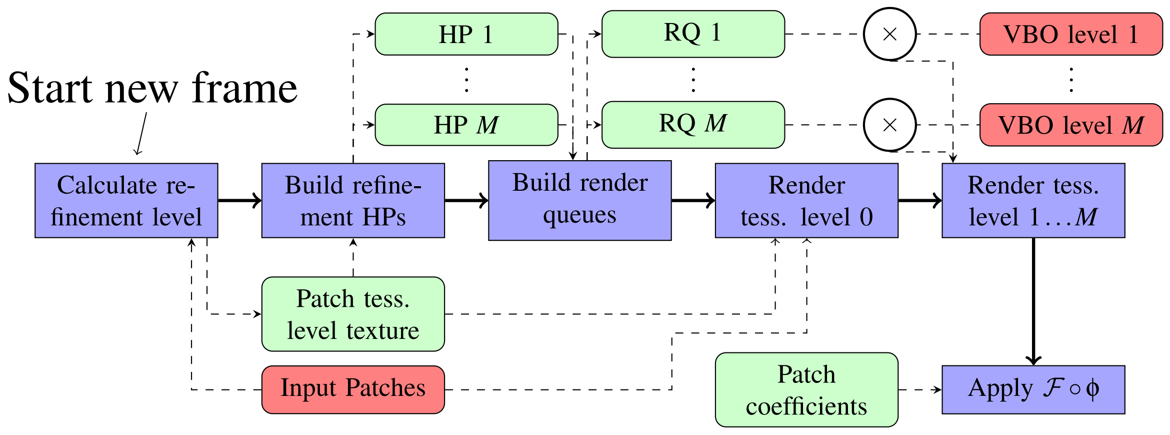

Figure 3: A schematic view of our implementation. The thick arrows designate control flow, while the data flow is drawn in green using dashed arrows. Static data is drawn in red. Instancing is indicated by the $\otimes$-symbol.

```
vec4 p = gl_Vertex;
vec4 m = equal(floor(p.wwww), vec4(0,1,2,3));
float l = dot(levels, m);
if(m.w == 0) {
  float s = exp2(ceil(lod));
  float t = (1/s)*floor(
    s*dot(p.xyz, m.xyz) + fract(p.w));
  p.xyz = mix(mask.zxy, mask.xyz, t);
}
```

Listing 1: GLSL-implementation of ϕ.

with vertices $\mathbb{I}^m_{p_0p_1p_2} = \{\phi(\mathbf{u}) : \mathbf{u} \in \mathbb{I}^m\} \subseteq \mathbb{I}^m$. A triangle in $\mathcal{P}^m_{p_0p_1p_2}$ is degenerate if two of its vertices are identical. The remaining triangles cannot fold over and since ϕ preserves the order of the boundary vertices, $\mathcal{P}^m_{p_0p_1p_2}$ is a valid tessellation of $\mathcal{P}^0$ if we ignore the degenerate triangles.

Our choice of snap function minimizes the problem of long thin triangles for a given set of vertices, since ϕ always maps a boundary point to the *closest* boundary point in $\mathbb{I}^m_{p_0p_1p_2}$. It can be shown that if $\mathcal{P}$ is an equilateral triangle, then $\mathcal{P}^m_{p_0p_1p_2}$ is a Delaunay triangulation of $\mathbb{I}^m_{p_0p_1p_2}$.

4 Implementation

In this section we describe a pure GPU implementation of the tessellation scheme on DirectX 10-class hardware. The performance benchmark in the next section compares this approach to CPU-assisted alternatives on older hardware.

Conceptually, an implementation of the scheme is straightforward, and the snap function can be realized in a few lines of shader code, see Listing 1. For simplicity we assume the input patches are static, but this is not a requirement. Given an input mesh of patches and a maximum refinement level M, we use some predicate to associate a refinement level $0 \leq l_e \leq M$ with every edge, e.g. silhouetteness [DRS08], patch curvature, or the distance from the camera. The patch tessellation level m is the maximum of the integer edge tessellation levels $\lceil l_e \rceil$. We then issue the rendering of $\mathcal{P}^m$, applying the snap function ϕ and the patch evaluator $\mathcal{F}$ to calculate the final vertex positions. Any parametric patch evaluator can be used, such as subdivision patches [Sta98] or PN-triangles [VPBM01]. Similar to adaptive refinement patterns [BS08], the tessellations only handle integer refinement levels. However, continuous refinement levels can be accommodated by using a blending scheme like adaptively blended Bézier patches [DRS08].

The calculations are organized as in Figure 3. We first calculate the patch tessellation level for each input patch. Thereafter we group the patches according to tessellation level, using the HistoPyramid [ZTTS06] data compaction algorithm. For each level $m = 1, \dots, M$ we first extract a render queue consisting of all patches with tessellation level m. Then we render the $\mathcal{P}^m$-patches in the queue using a single draw call using instancing, applying $\mathcal{F} \circ \phi$ to the vertices using a vertex shader.

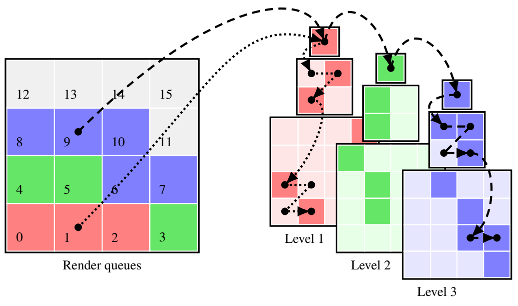

Figure 4: The render queues (left) are extracted in a single pass by traversing an array of HistoPyramids (right).

4.1 Building the render queues

For each frame, the first step is to determine the tessellation level of each patch. This is done in a GPGPU-pass, resulting in a patch tessellation level texture. Then, for each $m = 1, \dots, M$ we build a HistoPyramid, setting the base level of the pyramid to 1 if the patch is of level m and 0 otherwise. The upper levels in the HistoPyramid are built by a sequence of GPGPU-passes. Finally, we trigger an asynchronous readback of the top-elements of the HistoPyramids to the CPU, a total of M integers, which give the number of patches in each render queue. These numbers are needed by the CPU to control subsequent passes, e.g the number of instances to produce of a given tessellation.

The render queues are laid out in a texture and extracted using a single GPGPU-pass, see Figure 4. The elements of the render queue texture are indexed by linear enumeration. The HistoPyramids are laid out in an array of mipmapped textures, and the accumulative sums of the top elements form ranges of indices occupied by each queue. Thus, by inspecting the top elements of the HistoPyramids, one can determine which render queue a render queue element belongs to, and then descend into the appropriate HistoPyramid and determine the corresponding patch. The patch index is stored in the render queue, along with the tessellation levels of the neighbors.

Our OpenGL bucket sort approach based on combining several HistoPyramids can be used to group geometry for other uses, e.g. sorting geometry into layers to control sequence of pixel overwrites in scenes with transparent geometries.

The HistoPyramid algorithms are quite efficient on sparse data [DZTSar], which is the case when we extract patches along the silhouette. In other settings, it may be beneficial to use a data parallel programming environment, like Nvidia's CUDA, instead of the graphics API to build the render queues. The use of CUDA would allow for a less restrictive choice of algorithms. Our instancing approach requires the number of patches with a given refinement level to be known, and thus a regular sorting algorithm alone does not suffice without some subsequent reduction passes. However, Prefix Sum Scan [Har07] can be used in the same manner as we use the HistoPyramid, using one stream reduction per bucket. Finally, the render queues must be passed to a graphics API in order to exploit instancing for the final rendering pass.

4.2 Rendering

The geometry is rendered in two stages, one for unrefined patches and one for the refined patches. The input patches are stored in a static VBO, with the patch index encoded. We render this VBO using a vertex shader that degenerates all patches tagged for refinement in the patch tessellation level texture.

We then render the patches in the render queues. For each tessellation level m we have a parameter space tessellation $\mathcal{P}^m$ stored in a static VBO [BS05]. The xyz-coordinates of `gl_Vertex` contain the barycentric coordinates, and the integer part of w specifies if the vertex is in the interior or on an edge. Since the first case of (1) can be exchanged with $\lfloor 2^p t + \frac{1}{2} - \varepsilon \rfloor$ for a sufficiently small $\varepsilon > 0$, we let the fraction part of w contain $\frac{1}{2}$ or $\frac{1}{2} - \varepsilon$ and thus avoiding run-time conditionals in σ_p. Listing 1 shows our GLSL-implementation of ϕ. We bind the render queue as a texture and trigger the given number of instances of $\mathcal{P}^m$. The vertex shader fetches the neighboring tessellation levels and patch index from the render queue using the instance ID, and then use the patch index to fetch the per-patch data. We get the final vertex position by applying ϕ and $\mathcal{F}$, producing the evaluated patch at the primitive assembly stage. Thus, any regular fragment shader can be used without modification.

		Number of triangles											
		Base	Uniform	Adaptive	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
GeForce 7800 GT	max level 3	0.2 k	13 k	11 k	1988 (25)	1146 (15)	672 (7.1)	648 (6.8)					
		0.8 k	51 k	25 k	1396 (71)	440 (23)	387 (9.8)	368 (9.4)					
		1.0 k	64 k	42 k	1187 (76)	363 (23)	291 (12)	276 (12)					
		1.5 k	98 k	31 k	981 (96)	252 (25)	329 (10)	323 (10)					
		4.0 k	256 k	80 k	500 (128)	103 (26)	182 (15)	172 (14)					
		6.1 k	393 k	71 k	351 (138)	68 (27)	197 (14)	184 (13)					
		23 k	1486 k	325 k	120 (178)	18 (27)	59 (19)	54 (18)					
		97 k	6206 k	914 k	3.6 (22)	4.4 (27)	22 (20)	20 (18)					
	max level 5	0.2 k	205 k	155 k	609 (125)	214 (44)	232 (36)	203 (32)					
		0.8 k	819 k	347 k	199 (163)	57 (47)	113 (39)	98 (34)					
		1.0 k	1024 k	595 k	158 (162)	46 (47)	71 (42)	62 (37)					
		1.5 k	1573 k	412 k	113 (178)	30 (47)	97 (40)	86 (35)					
		4.0 k	4096 k	1055 k	8.8 (36)	12 (49)	41 (43)	36 (38)					
		6.1 k	6291 k	876 k	5.7 (36)	7.6 (48)	49 (43)	42 (37)					
		23 k	23773 k	4151 k	—	2.0 (48)	11 (46)	9.3 (39)					
		97 k	99293 k	11945 k	—	0.5 (50)	4.0 (48)	3.4 (41)					
GeForce 8800 GT	max level 3	0.2 k	13 k	11 k	3593 (46)	1667 (21)	745 (7.9)	736 (7.8)	1014 (11)	1000 (11)	931 (9.8)	798 (8.4)	250 (2.6)
		0.8 k	51 k	25 k	2513 (129)	614 (31)	482 (12)	482 (12)	850 (22)	832 (21)	862 (22)	841 (21)	81 (2.1)
		1.0 k	64 k	42 k	3098 (198)	511 (33)	369 (15)	384 (16)	813 (34)	693 (29)	824 (35)	822 (35)	60 (2.5)
		1.5 k	98 k	31 k	2341 (230)	293 (29)	428 (13)	419 (13)	821 (25)	762 (24)	813 (25)	814 (25)	65 (2.0)
		4.0 k	256 k	80 k	1082 (277)	117 (30)	218 (17)	219 (18)	590 (47)	500 (40)	685 (55)	684 (55)	25 (2.0)
		6.1 k	393 k	71 k	746 (293)	93 (37)	239 (17)	234 (17)	559 (40)	488 (35)	686 (49)	685 (49)	26 (1.8)
		23 k	1486 k	325 k	216 (321)	24 (36)	66 (21)	65 (21)	193 (63)	195 (63)	278 (90)	273 (89)	6.0 (2.0)
		97 k	6206 k	914 k	52 (323)	5.9 (37)	23 (21)	23 (21)	67 (61)	68 (62)	111 (101)	108 (99)	1.2 (1.1)
	max level 5	0.2 k	205 k	155 k	1417 (290)	1349 (276)	732 (114)	731 (113)	680 (106)	846 (131)	693 (108)	693 (108)	—
		0.8 k	819 k	347 k	532 (436)	475 (389)	484 (168)	485 (168)	420 (146)	497 (173)	435 (151)	509 (177)	—
		1.0 k	1024 k	595 k	425 (435)	381 (390)	382 (227)	377 (225)	294 (175)	362 (216)	336 (200)	397 (236)	—
		1.5 k	1573 k	412 k	296 (466)	258 (406)	424 (175)	427 (176)	383 (158)	530 (218)	403 (166)	461 (190)	—
		4.0 k	4096 k	1055 k	115 (471)	101 (414)	222 (234)	220 (232)	171 (180)	262 (276)	210 (222)	235 (248)	—
		6.1 k	6291 k	876 k	76 (478)	66 (415)	239 (209)	240 (210)	191 (167)	277 (243)	240 (210)	290 (254)	—
		23 k	23773 k	4151 k	20 (475)	18 (428)	64 (266)	64 (266)	45 (187)	78 (324)	59 (245)	74 (307)	—
		97 k	99293 k	11945 k	—	4.2 (417)	24 (287)	23 (275)	16 (191)	30 (358)	22 (263)	29 (346)	—

A	Uniform, dyadic tess, static VBO
B	Uniform, dyadic tess, per patch VBO [BS05]
C	Adaptive, dyadic tess, per patch VBO [DRS08]
D	Adaptive, SU tess, per patch VBO
E	Adaptive, SU tess, CPU RQ, texture coeffs
F	Adaptive, SU tess, CPU RQ, bindable uniforms
G	Adaptive, SU tess, GPU RQ, texture coeffs
H	Adaptive, SU tess, GPU RQ, bindable uniforms
I	Adaptive, SU tess, geometry shader

Table 1: Performance in frames per second for a variety of algorithms, including both dyadic and semi-uniform (SU) tessellation. We indicate how many millions of triangles that are processed per second in parentheses. The first column gives the number of triangles unrefined, with uniform refinement, and with adaptive refinement. Algorithms E–H use instancing, and algorithm I rely on the geometry shader, features only available on DirectX 10-class GPUs.

4.3 Optimizations

The description above gives a simplified view of the implementation. We reduce the number of GPGPU-passes by storing the set of HistoPyramids in a single mipmapped 2D texture array. In this way, all HistoPyramids can be bound to a single texture sampler and built in parallel using multiple render targets.

Per-patch data, such as shading normals and coefficients, can be uploaded through bindable uniforms instead of buffer textures. The rationale behind this is that it is faster to have data that is constant for numerous vertices stored in constant memory instead of (cached) textures. We store the extra data in addition to the edge refinement levels in the render queue, increasing the storage requirement to 40 floats per patch. The render queues are built using transform feedback. A restriction of bindable uniforms is that the amount of constant memory is limited. This limits the batch size to roughly 400 patches, resulting in more draw calls. The performance effect of this approach is described in the following section.

5 Performance analysis

We have performed a series of benchmarks on the implementation described in the last section, as well as a few variations of it, on two GPUs: Nvidia GeForce 7800 GT and 8800 GT. The benchmarks consists of measuring the average framerate and throughput (number of triangles per second) for a series of differently shaped and sized meshes with a randomly varied viewpoint. The results are presented in Table 1 and some of the meshes are depicted in Figure 5. For the benchmarks, we have used the *silhouetteness* criterion [DRS08] to determine the patch tessellation level and PN-Triangles [VPBM01] as the patch evaluator.

Method A is a brute force approach that, as a preprocess, tessellates all patches to level M, and stores this in a static VBO which is rendered for every frame. Method B is the uniform refinement method [BS05], using a single parameter space tessellation of level M that is invoked for every patch. Method C is our earlier method which results in hanging nodes [DRS08]. In method D this method is augmented with the proposed snap function ϕ, and Method E also uses instancing to draw the geometry instead of per patch triggering. We build the render queues on the host CPU using memory-mapped buffer objects. Then, each queue is bound to a buffer texture, and rendered in one batch using instancing. Method F is identical to Method D except that the render queues are exposed to the vertex shader through bindable uniforms as outlined in Section 4.3. Methods G and H are the pure GPU implementation described in Section 4. Method G builds a single texture with all render queues in a GPGPU-pass. Method H exposes the per-patch data through bindable uniforms. Method I is semi-uniform tessellation implemented as a single geometry shader program. The triangle adjacency primitive is used to stream the base triangles along with direct neighbors into the geometry shader, which initially calculate the silhouetteness of the edges. If none of the edges dictate a refinement, the triangle is passed directly through. Otherwise, an uniform dyadic tessellation is produced on the fly and $\mathcal{F} \circ \phi$ is applied to every emitted vertex.

We now discuss the results of our performance benchmarks. As method A renders the refined mesh directly, it is the fastest uniform approach, as long as the VBO fits in GPU memory. Since methods A and B yield the same set of triangles, their relative performance gives some information on the cost associated with triggering one VBO per patch and evaluating f in the vertex shader. It is clear that a large pre-evaluated VBO is much faster than evaluating the vertices for every frame, but it can be prohibitively expensive with respect to memory usage. We note, however, that for deeper refinement levels, the methods are roughly equal in performance on the 8800 card but not on the 7800. We believe that this is due to the unified shader architecture of the 8800 which allows for better load balancing when the vertex processing dominates.

For the semi-uniform schemes we observe that there is a marginal difference between C and D on the same hardware, indicating that applying ϕ has a negligible overhead. The adaptive methods (C–H) outperform the uniform method B as the number of patches grow above 1000–1500, probably due to the reduced number of vertices processed.

The algorithms based on instancing (E–H) give a considerable performance increase, and for larger meshes even outperforms using a static VBO (A). We observe that patches at intermediate refinement levels yield a relatively low throughput when rendered per patch, while using instancing with a small number of tessellation types reduce the triggering cost significantly. For deeper refinement levels we observe that our approach performs nicely, with the relative cost of the snapping function diminishing.

We believe that adaptive refinement patterns [BS08] would have performance comparable to Method C. Instancing could be used, however, we believe that the benefit would be somewhat limited since the number of tessellation patterns grows like $\mathcal{O}(M^3)$, which in turn results in smaller batches and more elaborate sorting.

For meshes with more than about 1000 patches, render queue generation appears to be faster on the GPU (G and H) than on the CPU (E and F). When it comes to the different methods for passing the patch coefficients, the results are less clear. For $M = 3$ it appears that texture buffers is the fastest approach while for $M = 5$ it is faster to use bindable uniforms. We believe this is because the patch data is more heavily accessed, so the fast constant memory outweighs the smaller batch sizes.

The geometry shader implementation (I) is the simplest adaptive method to implement, however, it has consistently the worst performance. We believe the reason is that the hardware does not know in advance the number of output primitives per patch, and this makes load balancing difficult. Also, the geometry upscale capabilities are insufficient for refinement levels greater than three, which limits the applicability of this approach.

6 Conclusion

We have presented a framework for semi-uniform adaptive tessellation of triangular patches. Our approach is independent of the patch type and tessellation level criterion, and can easily be modified to e.g. quadrilateral patches and subdivision surfaces and other refinement predicates.

The individual patch tessellations are dyadic refinements, which are topologically uniform and trivial to generate. A simple snapping operation with marginal overhead ensures that tessellations are consistent across patch boundaries. The snap function is well suited for implementation in a vertex shader, requiring only a few lines of code, and can be incorporated in a rendering framework using one single VBO for each tessellation level. Since the number of VBOs is low, the memory requirements are minimized and allows for deep refinement levels. Moreover, it makes our approach well suited for instancing, which improves the performance significantly as shown by our performance analysis. Our experiments also confirm that performing tessellation using the geometry shader is not competitive on current GPUs.

The advent of fast GPUs built on wide parallelism has favored large regular meshes over small irregular meshes, and uniform refinement methods over more general refinement. From our experiments, one might draw the conclusion that adaptivity can be competitive as long as enough uniformity is present in the vertex stream to use instancing. This allows for all GPU pipelines to be used efficiently, while the unified shader architecture adaptively balances the workload. Our semi-uniform approach appears to find a good balance between adaptivity and uniformity, and seems particularly well suited for massively parallel modern GPUs.

7 Acknowledgments

We thank Denis Kovacs and Ignacio Castaño for helpful comments in this work. Furthermore, we would like to thank Nvidia for hardware donations.

References

[AB06] ANDREWS J., BAKER N.: Xbox 360 system architecture. *IEEE Micro 26*, 2 (2006), 25–37.

[BS05] BOUBEKEUR T., SCHLICK C.: Generic mesh refinement on GPU. In *Graphics Hardware 2005* (July 2005), pp. 99–104.

[BS08] BOUBEKEUR T., SCHLICK C.: A flexible kernel for adaptive mesh refinement on GPU. *Computer Graphics Forum 27*, 1 (2008), 102–114.

[Bun05] BUNNELL M.: Adaptive tessellation of subdivision surfaces with displacement mapping. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 109–122.

[DRS08] DYKEN C., REIMERS M., SELAND J.: Real-time GPU silhouette refinement using adaptively blended Bézier patches. *Computer Graphics Forum 27*, 1 (2008), 1–12.

[DZTSar] DYKEN C., ZIEGLER G., THEOBALT C., SEIDEL H.-P.: High-speed marching cubes using histogram pyramids. *Computer Graphics Forum* (To appear).

[Har07] HARRIS M.: Parallel prefix sum (scan) with CUDA. NVIDIA CUDA SDK 1.0, 2007.

[HDD*93] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Mesh optimization. In *Proceedings of SIGGRAPH 93* (Aug. 1993), Computer Graphics Proceedings, Annual Conference Series, pp. 19–26.

[LH04] LOSASSO F., HOPPE H.: Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 769–776.

[Mor01] MORETON H.: Watertight tessellation using forward differencing. In *Graphics Hardware 2001* (2001), pp. 25–32.

[SAF*06] SEGAL M., AKELEY K., FRAZIER C., LEECH J., BROWN P.: *The OpenGL Graphics system: A Specification, version 2.1.* Silicon Graphics, Inc., 2006.

[SJP05] SHIUE L.-J., JONES I., PETERS J.: A realtime GPU subdivision kernel. *ACM Transactions on Graphics 24*, 3 (Aug. 2005), 1010–1015.

[Sta98] STAM J.: Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of SIGGRAPH 98* (July 1998), Computer Graphics Proceedings, Annual Conference Series, pp. 395–404.

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J. L.: Curved PN triangles. In *2001 ACM Symposium on Interactive 3D Graphics* (Mar. 2001), pp. 159–166.

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.-P.: *GPU Point List Generation through Histogram Pyramids*. Tech. Rep. MPI-I-2006-4-002, Max-Planck-Institut für Informatik, 2006.

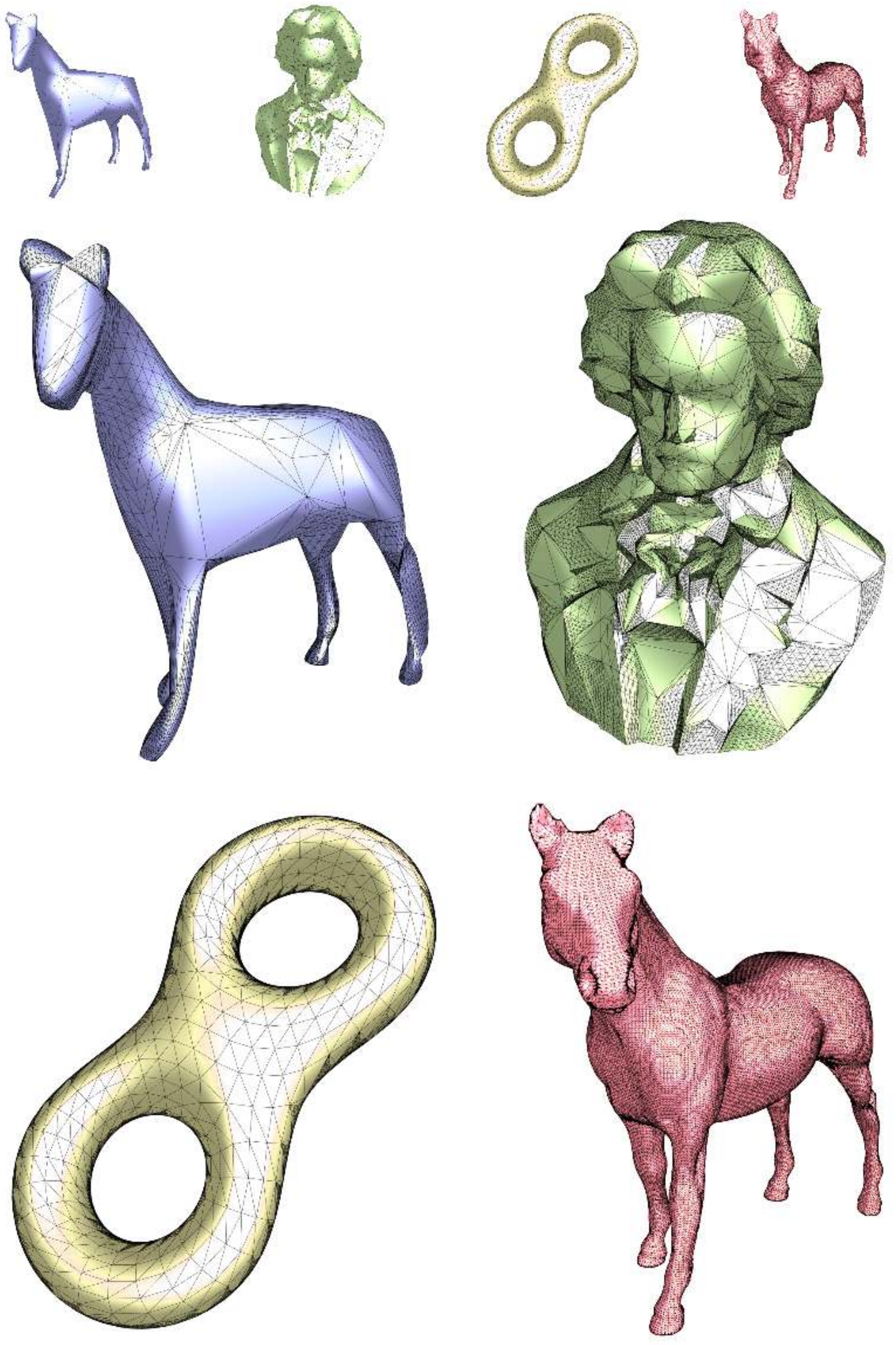

Figure 5: Some of the models used in the performance analysis. The top row shows the base meshes of four models with 200, 1000, 1546, and 96966 triangles respectively. The middle and bottom rows show these base meshes adaptively refined using semi-uniform tessellations guided by the silhouetteness-predicate, with a maximum refinement level of three.