Rendering of Bézier Surfaces on Handheld Devices

Raquel Concheiro Margarita Amor Emilio J. Padrón
Universidade da Coruña
Facultade de Informática
Campus Elviña, S/N
15071, A Coruña, Spain
rconcheiro | margamor | emilioj @udc.es

Marisa Gil Xavier Martorell
Universitat Politècnica de Catalunya
Campus Nord, Mòdul D6
Jordi Girona, 1–3
08034 BARCELONA, Spain
marisa | xavim @ac.upc.edu

ABSTRACT

Bézier surfaces have been widely employed in the designing of complex scenes with high-quality results. Nevertheless, parametric surfaces cannot be directly rendered in the current GPUs of modern handheld devices. This work proposes a non-adaptive method for tessellating Bézier surfaces on a GPU without primitive generator, such as the GPUs implemented in handled devices. Our technique is based on the utilization of a parametric map of virtual vertices, and its operation can be adapted to the hardware resources available in the GPU by tuning a series of parameters. Additionally, an analysis of the most relevant hardware constraints in the graphics hardware of the current handheld devices has been carried out. As those constraints prevent interactive high-quality results from being achieved, even with our proposal, we present an algorithmic approach focused on the real-time rendering on future handheld devices.

Keywords

Bézier surfaces; GPUs; Handheld Devices; Tuning rendering

1 INTRODUCTION

The market of handheld devices, such as smartphones, consoles or tablets, is nowadays one of the fastest growing technology markets. Graphics processing has become a significant factor on these devices, as consumers' expectations have increased, demanding high quality visual contents and complex render capabilities. Consequently, a new GPU generation has been specifically designed to fit in the constraints of handheld devices: size and power-consumption. Hence, GPUs of these devices implement only a subset of the features available in commodity desktop GPUs. Furthermore, a stripped-down version of the well-known graphics API OpenGL has been developed for these devices: OpenGL ES [Khron10].

Although offline rendering is an important area in computer graphics, especially as far as photorealism is concern, real-time rendering is probably the traditional mainstay of computer graphics. Since an efficiency-quality trade-off is needed in this kind of rendering to maintain interactive rendering rates, the design of an efficient graphics pipeline arises as a key performance factor. This is an issue especially in handheld devices.

Moreover, these rendering pipelines and their supporting graphics hardware are usually designed to work with triangles and vertices. However, these geometric primitives are not always the best option from a modeling point of view. Thus, the use of parametric surfaces to design complex and detailed models has widely spread in fields such as CAD/CAM, virtual reality, animation and visualization. Specifically, the Bézier representation has been widely employed in the designing of high quality complex models [Roger01, Piegl97]. The excellent mathematical and algorithmic properties, combined with successful industrial applications, have contributed to the popularity of this representation.

Bézier surfaces have two significant features from the point of view of a rendering pipeline: compactness, which means low storage and transmission requirements of the resulting models; and scalability, so a surface can be converted into a triangle mesh with few triangles or with many triangles according to the required level of detail (LOD). There are two main approaches for rendering parametric surfaces: tessellation on the CPU or on the GPU. In the first approach, the Bézier surfaces are tessellated into triangles on the CPU, so the resulting triangle mesh is sent down to the GPU to be displayed. This strategy presents some disadvantages that could affect system performance: the amount of information to be transferred from CPU to GPU and the increment in the storage requirements in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

the GPU associated with the triangle mesh. These issues are fixed by performing the tessellation directly on the GPU [Guthe05, Dyken09, Conch10, Conch11].

In the case of handheld devices, there are still few proposals dealing with tessellation on the GPU. First works were oriented toward graphics hardware with low programmability, so they were implemented in additional and specific hardware units [Chung09, Chung08]. In [Kim12] a hardware unit for an efficient tessellation in handheld devices was also proposed, but this proposal describes a tessellation procedures for subdivision surfaces.

In this work we present a novel approach to the tessellation of Bézier surfaces on the GPU of handheld devices. Our proposal tessellates parametric surfaces into high-quality triangle meshes that accurately represent complex surfaces and do not contain artifacts such as T-junctions or cracks. It is based on the utilization of a parametric maps of virtual vertices [Boube05, Conch10, Guthe05], what makes it possible to work on GPUs with no primitive generator. More specifically, the guidelines proposed in [Conch10] have had to be adapted to fit the constraints of the graphics hardware in mobile devices, leading us to a completely different implementation, as described in Section 4. Our design allows the efficient exploitation of the information stored in the GPU and the minimization of the CPU-GPU communications. Three main parameters are exposed to allow a fine tuning of the method to the hardware resources available: maximum resolution level, number of surfaces to be rendered per draw call and number of draw calls per frame.

In order to test our approach, we have made an OpenGL ES implementation of the method for Android systems [Goo] and we have designed a full set of experiments to analyze the reasons why Bézier surfaces can not be real-time rendered with good quality by current handheld GPUs. The tests were focused on locating the main performance bottlenecks and identifying possible enhancements and tuning opportunities. Thus, the results obtained could be a useful tool to introduce architecture improvements. Let us emphasize that nowadays, complex triangle meshes can not be rendered in real-time in these devices either [Sarmi12].

This rest of the paper is organized as follows: Section 2 briefly goes over the basics of Bézier surfaces, Section 3 presents our approach to tessellate Bézier surfaces on handheld devices, Section 4 describes the implementation on Android smartphones with OpenGL ES and Section 5 presents the experimental results obtained in our tests. Finally, in Section 6 the main conclusions are highlighted.

2 BÉZIER SURFACES

In this section a brief introduction to the Bézier parametric representation is presented. For reasons of clarity, Bézier curves are first introduced and, after this, the description is extended to Bézier surfaces. An in-depth description can be found in [Piegl97, Roger01].

A Bézier curve is specified by giving a set of coordinate positions, called control points, which indicate the general shape of the curve. These control points are then fitted with piecewise continuous parametric polynomial functions. Mathematically, a parametric n degree Bézier curve is defined by:

$$P(t) = \sum_{i=0}^{n} B_i J_{n,i}(t), \quad 0 \leq t \leq 1 \tag{1}$$

where B_i are the control points and $J_{n,i}$ are the classical n-degree Bernstein polynomials defined by:

$$J_{n,i}(t) = \binom{n}{i} (1-t)^{(n-i)} t^i \tag{2}$$

where n is the degree of the Bézier basis functions. These functions decide the extent to which a particular control point controls the surface at a particular parametric value t. Only $n+1$ control points and the n-degree Bernstein polynomials are required for the computation of each point of the curve. Note that the first and last control points are coincident with the end points of the curve, that is, $P(0) = B_0$ and $P(1) = B_n$.

The equation for a Bézier curve can be also expressed in matrix form:

$$P(t) = [T][N][G] \tag{3}$$

where $[T] = [t^n \;\; t^{n-1} \ldots \; t^1 \;\; t^0]$, the geometry of the curve is represented as $[G]^T = [B_0 \;\; B_1 \ldots B_n]$, and the $[N]$ matrix is defined by:

$$\begin{bmatrix} \binom{n}{0}\binom{n}{n}(-1)^n & \binom{n}{1}\binom{n-1}{n-1}(-1)^{n-1} & \cdots & \binom{n}{n}\binom{n-n}{n-n}(-1)^0 \\ \cdots & \cdots & \cdots & \cdots \\ \binom{n}{0}\binom{n}{1}(-1)^1 & \binom{n}{1}\binom{n-1}{0}(-1)^0 & \cdots & 0 \\ \binom{n}{0}\binom{n}{0}(-1)^0 & 0 & \cdots & 0 \end{bmatrix}$$

Thus, the matrix form for a cubic Bézier ($n = 3$) is:

$$P(t) = [T][N][G] =$$

$$= [t^3 \;\; t^2 \;\; t^1 \;\; 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{4}$$

Likewise, the shape of a (n,m)-degree Bézier surface is controlled by a set of control points through the equation:

$$Q(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,j} J_{n,i}(u) K_{m,j}(v), \quad 0 \leq u,v \leq 1 \tag{5}$$

where $J_{n,i}(u)$ and $K_{m,j}(v)$ are the Bézier basis functions in the u and v parametric directions and $B_{i,j}$ are the vertices of a polygonal control net. Again the number of control points in the u and v directions are $n+1$ and $m+1$ respectively. In matrix form, a Bézier surface is given by:

$$Q(u,v) = [U][N][B][M]^T[V] \tag{6}$$

For the specific case of a bicubic Bézier surface, the matrix form is given by:

$$Q(u,v) = [u^3 \;\; u^2 \;\; u \;\; 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{7}$$

3 BÉZIER TESSELLATION BASED ON PARAMETRIC MAPS OF VIRTUAL VERTICES

The tessellation of a parametric surface involves the computation of a set of surface points that correspond to the vertices of the triangular mesh, and the identification of the connectivity among them. Since the GPU of current handheld devices do not generate any new geometry, the design of our tessellation proposal is based on virtual vertices [Boube05, Conch10, Guthe05]. This technique uses a parametric map as vertex shader input, with as many positions in the parametric domain as output vertices are needed for the desired resolution of the triangle mesh. Then, by accessing the control points on the Bézier surface to be tessellated, these virtual vertices are evaluated on the vertex shader, generating the resulting triangle mesh. Hence, the resolution of the triangle mesh is chosen by the parametric map being used. Since this approach was initially designed for commodity GPUs, we propose a tuning technique for the effective utilization of the scarce resources available in the GPUs of handheld devices.

Our approach subdivides the parametric domain into uniform squares, where the granularity is selected in function of the desired resolution. More specifically, it tessellates the surface in the parametric space (u,v) in $2^l \times 2^l$ squares of size $\frac{1}{2^l} \times \frac{1}{2^l}$, for a resolution level l that is previously selected by the application taking into account different factors, such as computational power, screen space error or model complexity. Therefore, the Bézier surface is evaluated for each one of the $2^{l+1} \times 2^{l+1}$ to obtain the corresponding Euclidean space points (see Equation 5). The resulting vertices are conveniently arranged to output a triangle strip.

Thus, the grid of parametric values P^l for a resolution level l would be:

$$P^l = \begin{bmatrix} (u_1,v_1) & (u_2,v_1) & \cdots & (u_{2^{l+1}},v_1) \\ (u_1,v_2) & (u_2,v_2) & \cdots & (u_{2^{l+1}},v_2) \\ \vdots & \vdots & \ddots & \vdots \\ (u_1,v_{2^{l+1}}) & (u_2,v_{2^{l+1}}) & \cdots & (u_{2^{l+1}},v_{2^{l+1}}) \end{bmatrix} \tag{8}$$

where

$$u_i, v_i = \frac{i-1}{2^{l+1}-1}, \quad i \in \{1,\cdots,2^{l+1}\}$$

The base case, $l=1$, directly projects the control points into the surface to obtain the vertices of the triangle strip.

Before starting, a set of L grids of parametric maps is precomputed on the CPU, where L is the highest resolution level needed: $\{P^1,P^2,\cdots,P^L\}$. These grids are stored in the GPU to be selected and employed as vertex shader input for the different surfaces of the model. The parametric grids are stored in a convenient pattern that implicitly contains connectivity information, preventing the need for any additional indices.

Obviously, the other essential data that need to be accessed by the vertex shader during surface evaluation are the control points. Since memory is a scarce resource in this kind of GPU (the next section explains how and where the Bézier surfaces are stored in the GPU), the surface's data is transferred to the GPU in chunks of N_d Bézier surfaces (of the total N_S surfaces to be rendered for each frame). Therefore, each Draw Primitive call processes a chunk of N_d surfaces, resulting in a total of N_{DP} drawing call for each frame:

$$N_{DP} = \frac{N_S}{N_d}$$

with $1 \leq N_d \leq N_S$.

Thus, if N_d surfaces are processed in each draw call, a total of $N_{samples} = 2^{l+1} \times 2^{l+1} \times N_d$ samples could be concurrently evaluated (assuming a fixed resolution l for all the surfaces in the chunk). This means an input of $N_{samples}$ virtual vertices is needed in the vertex shader, which is provided by N_d copies of the P^l parametric map as vertex shader input (stored in the vertex buffer). Regarding the GPU memory needed for the storage of the N_d Bézier surfaces, the required amount of memory is

$$M = M_{[B^s]} \times N_d \tag{9}$$

where $M_{[B^s]}$ is the storage needed for the control points of each surface and $N_d \ll N_S$ in current handheld devices. Since in most of these devices GPU computation and CPU-GPU transfers do not overlap, each draw call implies a synchronization point, as new M data is sent down to the GPU. The worst case would be a sequential

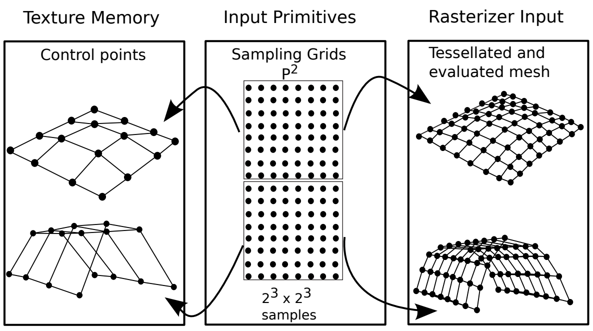


Figure 1: Example of parametric maps for $l = 2$

process of as many draw call as surfaces to render (N_S), with only one surface processed by draw call.

Figure 1 depicts an example of our approach for a resolution level of 2 ($l = 2$) and a couple of Bézier surfaces to be processed concurrently ($N_d = 2$). The parametric map for $l = 2$ is replicated and the resulting samples are the input primitives for the vertex shader (middle box in the figure). The control points of the two surfaces are transferred to the GPU (left box, texture memory is used in this example) and a draw call causes the evaluation of the samples that results in the meshes of the right box.

In summary, GPU performance depends on the right balance between: the number of simultaneous samples $N_{samples}$ that may be concurrently processed, which is a function of N_d and L; the amount of memory needed to storage the N_d Bézier surfaces of a chunk, M; and the number of synchronizations between CPU-GPU, N_{DP}. Therefore, an optimal balance can be expressed by three factors, $\{L, N_d, N_{DP}\}$. Even though two of these three factors are mutually dependent, the analysis is more clear considering the all three.

The number of samples to be processed in parallel may be restricted by the low computational power of the shaders in this kind of GPUs, the size of the vertex buffer or the storage capacity (this is dealt with in the next section). Regarding the influence of each draw call on the performance, it is important to bear in mind that they introduce a certain amount of processing overhead. For each draw call in a OpenGL ES compliant GPU, the graphics driver also collects all current OpenGL ES states, textures and vertex attribute data. The driver processes all this information to generate appropriate commands for the graphics hardware to perform the specified draw operation. This process can take a significant amount of time, and it is even more significant in the case of embedded systems. Finally, to evaluate the number of surfaces that can be processed per draw call, our proposal requires that the control points $[B^S]$ of N_d surfaces be stored in the GPU. Nevertheless, the scarce memory of current handheld devices makes it impossible to store large amount of surfaces.

4 IMPLEMENTATION DETAILS

In this section, we summarize the details of our implementation. The kernel implemented processes bicubic Bézier surfaces and exploits the capabilities of OpenGL ES 2.0. The structure of our algorithm is shown in Figure 2. In the preprocessing stage the grids of virtual vertices P^l, $1 \leq l \leq L$ are transferred from CPU to GPU. During the synthesis process the level of resolution per surface is selected and the control points of N_d surfaces are sent down from CPU to GPU.

As mentioned in the previous section, $N_{samples}$ samples or virtual vertices are sent down to GPU and stored in the *vertex buffer* to compose the input primitives for each vertex shader execution. The control points of each surface $[B^s]$ are stored in a 4×4 float3 arrays $[B^s_x, B^s_y, B^s_z]$.

All draw calls use the same parametric maps, while the resolution level is unchanged, reducing CPU-GPU transfers (i.e. synchronization points). Then, these virtual vertices are evaluated in the vertex shader of the GPU for all the N_d surfaces of a chunk.

Instead of applying the de Casteljau algorithm [Shirl03], in our kernel a direct evaluation strategy is used to compute the tessellation, as it results in a more efficient GPU implementation, avoiding recursion. Figure 3 shows the simple vertex shader pseudocode used for the bicubic surfaces evaluation. The input parameters of the vertex shader (line 1) are the grid parametric values P^l, which are employed in the evaluation of the N_d surfaces of a chunk. The (u, v) parametric values are stored in P^l coordinates x and y whereas the z coordinate stores a surface index $\{0, \cdots, N_d - 1\}$. Thus, each surface within a chunk can be directly indexed (line 8). To evaluate Equation 6 (line 12) $[U]$ and $[V]$ are calculated (lines 9 and 10), the control points of the surface are read from memory (line 11), and the basis functions coefficients (lines 3 to 6) are employed. As a result, the vertices of the final tessellated mesh are obtained.

As will be shown in the results section, the simplicity of this strategy and the efficient management of the data storage are key points, together with the CPU-GPU transfers, for the real time rendering of high quality models.

According to the vertex shader structure of OpenGL ES, this work proposes two different approaches to store Bézier's data in the GPU. The first option, Uniform method, is based on storing the control points of the surfaces in uniform variables, whereas the second one, Texture method, stores them in the texture memory. Both alternatives are described below.

4.1 Uniform method

Uniform variables memory is one type of variable modifiers in the OpenGL ES Shading Language (it has

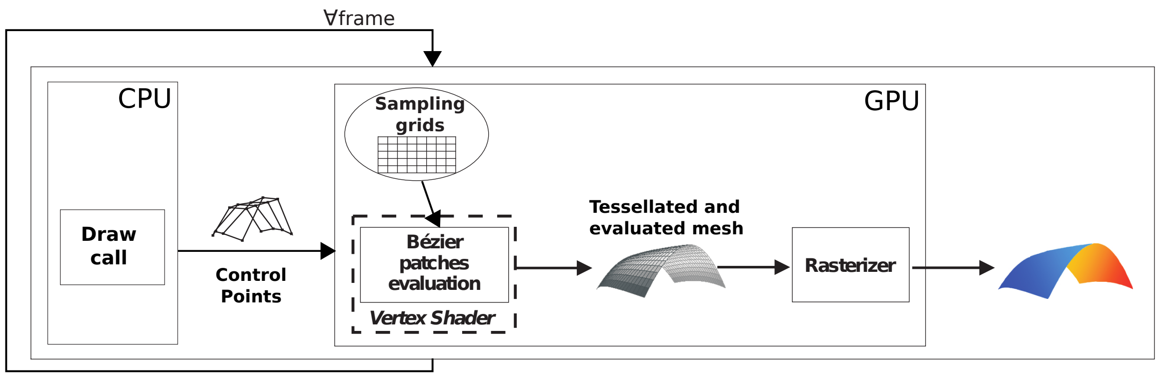


Figure 2: Structure of the method

```
VS_OUTPUT DefaultVS(VS_INPUT $P^l$)
{
  float4x4 [N]= { -1,  3, -3,  1,
                   3, -6,  3,  0,
                  -3,  3,  0,  0,
                   3,  0,  0,  0, }
  $u = P^l$.x; $v = P^l$.y;
  $s = P^l.z \times dp \times N_d$;
  float1x4 [U]=($u^3$, $u^2$, u, 1);
  float1x4 [V]=($v^3$, $v^2$, v, 1);
  float4x4 {[$B^s_x$],[$B^s_y$],[$B^s_z$]} = read from memory ($s$);
  float3  vertex = mul([U], [N],[$B^s$], [N], [V]);
  return vertex;
}
```

Figure 3: Vertex shader pseudocode

evolved in modern desktop GPUs into what is now known as constant memory). These uniform variables are useful for storing all kinds of constant data that shaders can need. Basically, any parameter provided to a shader that is constant across either all vertices or fragments, but that is known before executing the shader should be passed in as a uniform variable. This is the case of the control points of the Bézier surfaces to be tessellated.

From a performance point of view, and according to hardware manufacturers [Mali09, NVIDI11], any access to uniform variable memory is simple and fast and it has a low impact on execution speed. Moreover, this access overhead is similar to an arithmetic operation such as addition or subtraction, and considerably faster than other operation, such as division or square root.

Regarding the capacity of this constant storage, and according to the standard, any implementation of OpenGL ES 2.0 must provide at least uniform memory in the vertex shader, M_{uv}, to store 128 4-float vectors and uniform memory in the fragment shader, M_{uf}, to store 16 4-float vectors. Hence, the maximum number of surfaces per chunk would be

$$N_d = \frac{M_{uv}}{M_{[B^S]}} \qquad (10)$$

In the case of bicubic Bézier surfaces, 16 vectors of points are needed to store the control points of each surface, so $N_d = \frac{128}{16} = 8$ is the maximum number of surfaces that can be evaluated in the same draw call, assuming the minimum of uniform variables defined by OpenGL ES 2.0. There are commercial devices that provides a higher number of vertex uniform vectors; for instance Mali 400 provides 256 vertex uniform vectors, $N_d = \frac{256}{16} = 16$.

Clearly, the main drawback of this approach is the reduced number of surfaces that can be stored for each draw call (a low N_d), which means the bottleneck lies in the great number of draw calls needed (a high N_{DP}). This is especially a problem in devices that do not overlap GPU computation and transference.

4.2 Texture method

An alternative to the uniform variables is to store the control points in texture memory, M_T. As texture memory can store a higher number of surfaces than uniform memory, this alternative prevents an important number of draw calls per frame, N_{DP}, one of the main drawbacks of using uniform variables.

$$N_d = \frac{M_T}{M_{[B^S]}} \qquad (11)$$

where M_T is considerably larger than M_{uv} and subsequently more primitives can be stored in texture memory than in the uniform variables memory simultaneously. Specifically, for one of our test devices, Mali 400, $M_T = 16MB$, that is four times larger than M_{uv}.

Although Bézier control points can be stored in the texture memory, this storage space has not been designed for store floats. In OpenGL ES 2.0 texture memory formats have been implemented to store color information

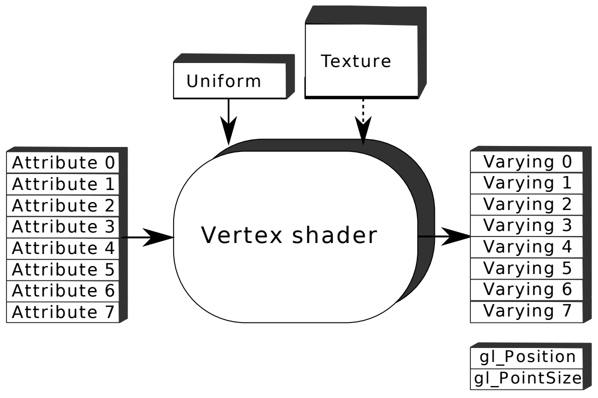


Figure 4: OpenGL ES 2.0 vertex shader

as a 4-byte vector. There are different formats in texture storage, but typically each color is 32 bit data and they are split up into 4 groups of 8 bits: rgba [Foley90] red, green and blue colors and the alpha channel. The texture method defines a codification process to store and recover float values from texture memory. This encoding process to pack a float into a rgba texture is a simple process based on multiplications and divisions by the largest number that can appear.

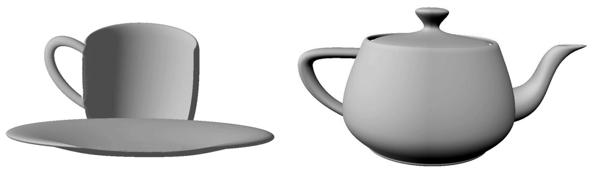
On the other hand, the texture method solves the main disadvantages of the uniform approach for handheld devices, however it has yet to be implemented on Android platforms. As it is shown in Figure 4 with a dashed line, access to texture memory from vertex shader is not implemented in any commercial OpenGL 2.0 device at this moment. First devices implementing this feature are expected in lately 2013.

5 EXPERIMENTAL RESULTS

In this section, the results of the evaluation of our proposal on different GPU architectures is analyzed. In particular, the platforms we have used are a Samsung Galaxy S2 (*Mali*), a Samsung Galaxy ACE (*Adreno*) and a Asus Transformer TF 300 (*Tegra 3*).

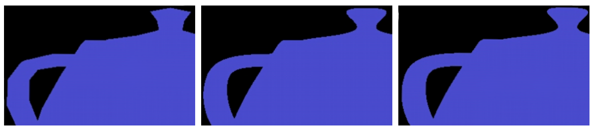
Samsung Galaxy S2 has a 1.2 GHz dual core ARM Cortex-A9 processor and uses ARM's Mali-400 MP GPU with a vertex shader and 4 fragment shaders. Samsung Galaxy ACE features an 800 MHz Qualcomm MSM7227 processor with the Adreno 200 GPU. Adreno 200 GPU implements a unified architecture where a core can dynamically allocate vertex or fragment processing. Finally, Asus Transformer TF300 implements a Nvidia Tegra 3 Quad-core at 1.2GHz and a ULP Geforce 12-core: 4 vertex shaders and 8 fragment shaders.

Different scenes, composed of replicas of a set of models, have been used in our tests. The models (*Teacup* and *Teapot*) are depicted in Figure 5. The number of primitives generated for the different resolution levels is shown in Table 1. Column N_s presents the number



(a) Teacup (b) Teapot

Figure 5: Models employed in the test scenes



(a) L=1 (b) L=3 (c) L=5

Figure 6: Screenshots of the *teacup* model with different levels of resolution

of Bézier surfaces whereas the rest of columns include the number of triangles generated for the corresponding level of detail with a uniform tessellation, i.e. all surfaces are tessellated with the same level of detail. Figure 6 depicts a screenshot of the *teacup* model rendered with $L = 1$, $L = 3$ and $L = 5$ in *Tegra 3*. Obviously, a higher level of detail results in a smoother render.

Our analysis is mainly focused on obtaining the optimal tuning factors for the three parameters used to characterize the behavior of our method: $\{L, N_d, N_{DP}\}$. As mention in the previous sections, we consider these three factors in our analysis for clarity reasons, even though two of them are mutually dependent. As the different graphs depict, the results obtained clearly show that our proposal obtains a better performance than the best CPU results: up to 3 fps in *Mali* and 5 fps in *Tegra* for the scene S_{5pots} with $L = 1$. In this CPU implementation each sample is evaluated in the CPU and the whole vertex buffer is sent down to the GPU for each frame.

The first factor we have analyzed in our method is the resolution level, L. Specifically, each of the $2^{l+1} \times 2^{l+1} \times N_d$ samples is evaluated in each GPU vertex shader. The worst configuration was chosen for the other two parameters: $N_d = 1$, so $N_{DP} = N_S$ and there are so many draw calls as surfaces. This configuration is the closest to the CPU tessellation behavior, so the pure computational power determines the difference in performance.

The graphs in Figures 7 and 8 show the frame rate on two distinct platforms for 4 of the test scenes with the different resolution levels. As can be observed, the performance achieved is far better than the described for the CPU, even in this unfavorable configuration. In both devices, *Mali* and *Tegra*, the frame rate drop when the resolution level increases, but some differences between the two platforms can be observed. *Mali* obtains better results when the tessellation level is low (less

Scene	N_s	$L=1$	$L=2$	$L=3$	$L=4$	$L=5$
S_{5cups}	130	2.29	12.44	57.13	244.00	1007.75
S_{5pots}	160	2.81	15.31	70.31	300.31	1240.31
S_{10ups}	260	4.57	24.88	114.26	488.01	2015.51
S_{10pots}	320	5.63	30.63	140.63	600.63	2480.63
S_{15cups}	390	6.86	37.32	171.39	732.01	3023.26
S_{15pots}	480	8.44	45.94	210.94	900.94	3720.94
S_{20cups}	520	9.14	49,77	228.52	976.02	4031.02
S_{20pots}	640	11.25	61.25	281.25	1201.25	4961.25

Table 1: Number of surfaces and triangles generated (in K) for each scene

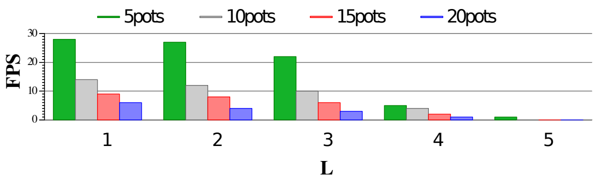


Figure 7: FPS of our implementation in *Mali* with different levels of resolution

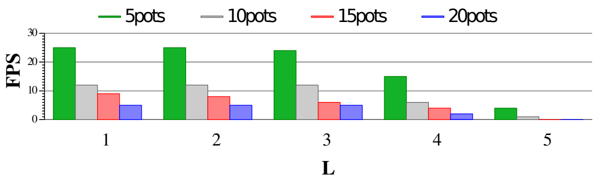


Figure 8: FPS of our implementation in *Tegra* with different levels of resolution

synchronization penalty) whereas *Tegra* performs better when the resolution level increases (4 vertex shaders vs. 1 vertex shader in *Mali*). In any case, frame rate dramatically drops for $L=5$ in both cases, since a vertex buffer size greater than 16 MB is needed. This information is provided by Table 2, that shows the KBytes stored in the vertex buffer for different scenes and different resolution levels.

In short, the main problems of current GPUs of handheld devices are the computing power and the low number of vertex shaders. This implies a limit on the resolution that can be achieved and the complexity of scenes that can be rendered.

With respect to the rest of factors $\{N_d, N_{DP}\}$ where $N_{DP} = N_S/N_d$, four different scenes have been considered and their performance is depicted with different resolution levels in Figure 9 on the *Mali*. This graph analyzes how the number of surfaces that can be tessellated by a single draw call affects the GPU performance. Similar behavior is observed on *Tegra* and *Adreno* and other scenes. Table3 presents the number of N_{DP} for these scenes with different number of draw calls. As can be observed, values lower than 40 reach maximum performance on *Mali*, that is 62 fps.

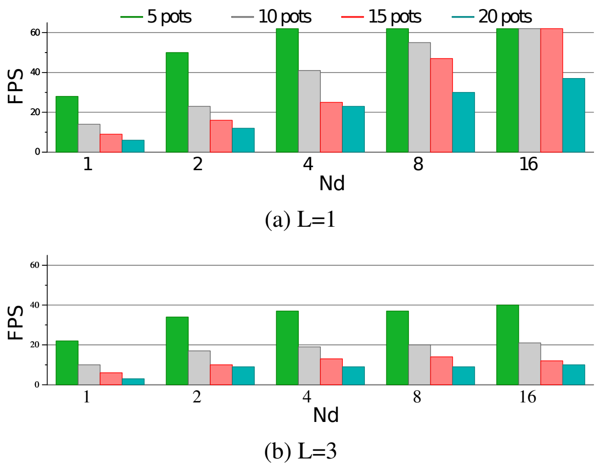


Figure 9: *Mali* with different levels of resolution

Broadly speaking, if the complexity of the model increases (more surfaces, Ns), maintaining a high performance usually implies to try to reduce N_{DP}. For example, S_{5pots} with $N_S = 160$ for $\{L=1, N_d=4, N_{DP}=40\}$ achieves 60 fps, and S_{15pots} with $N_S = 480$ also achieves 60 fps for a configuration $\{L=1, N_d=16, N_{DP}=30\}$. Thus, a trade off between the number of draw calls and the primitives processed in parallel is needed to increase the performance as much as possible.

Figure 10 and Figure 11 present a final comparison of our best results in different platforms: *Adreno*, *Mali* and *Tegra*. Different models such as S_{5pots}, and S_{20pots} have been depicted because a wide range of rendered primitives and vertex buffer size are considered. As has previously been explained, the uniform method stores Bézier control points in the vector uniform variables. Hence, as there are only 128 vector uniform variables in the Adreno 200 architecture, only 8 surfaces can be stored for each draw call, $N_d = 8$, meanwhile up to 16 surfaces can be stored in Mali 400 or in a Tegra 3 in 256 vector uniform variables, $N_d = 16$.

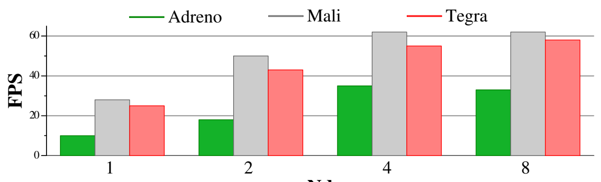
For a low level of resolution, $L=1$ or $L=2$, *Mali* and *Adreno* offer the best performance for $N_d = 8$, that is 62 and 35 fps, respectively. *Tegra* also has the best performance, 58 fps, but scales perfectly as the level of resolution increases, up to 55 fps for $L=3$. Nonethe-

Scene	$L=1$	$L=2$	$L=3$	$L=4$	$L=5$
S_{5cups}	60.94	255.94	1035.94	4155.93	16635.94
S_{5pots}	75.00	315.00	1275.00	5115.00	20475.00
S_{10cups}	121.88	511.88	2071.88	8311.88	33271.88
S_{10pots}	150.00	630.00	2550.00	10230.00	40950.00
S_{15cups}	182.81	767.81	3107.81	12467.81	49907.81
S_{15pots}	225.00	945.00	3825.00	15345.00	61425.00
S_{20cups}	243.75	1023.75	4143.75	16623.75	66543.75
S_{20pots}	300.00	1260.00	5100.00	20460.00	81900.00

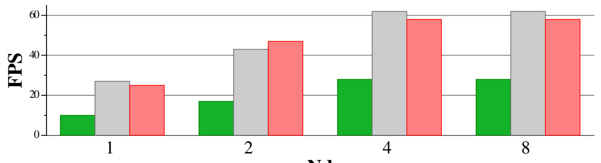
Table 2: Vertex Buffer size (in *KB*) for each scene

N_d	S_{5pots}	S_{10pots}	S_{15pots}	S_{20pots}
1	160	320	480	640
2	80	160	240	320
4	40	80	120	160
8	20	40	60	80
16	10	20	30	40

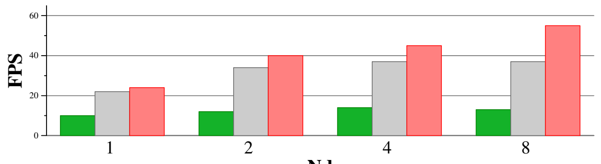
Table 3: N_{DP} for each scene



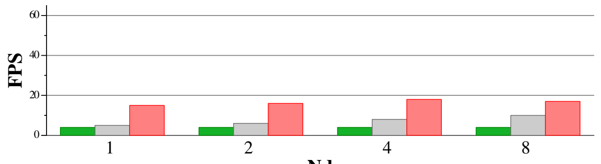
(a) L=1



(b) L=2

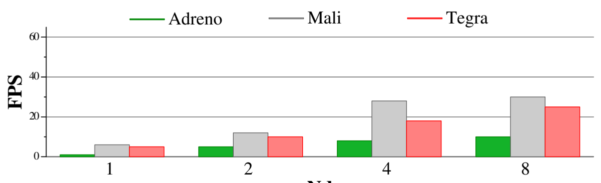


(c) L=3

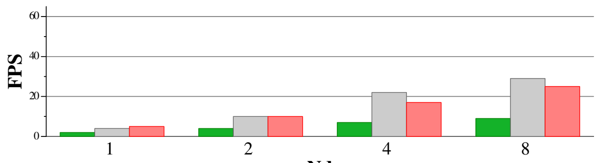


(d) L=4

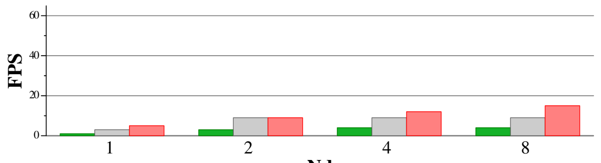
Figure 10: Frame rate comparative in *Adreno*, *Mali* and *Tegra* with S_{5pots} and different resolution levels



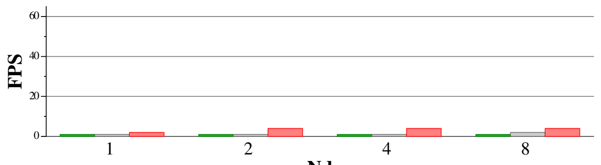
(a) L=1



(b) L=2



(c) L=3



(d) L=4

Figure 11: Frame rate comparative in *Adreno, Mali and Tegra* with S_{20pots} and different resolution levels

less, in *Mali* and *Adreno* the performance drop for $L \geq 3$ (37 and 13 fps are achieved, respectively, for $L = 3$ and $N_d = 8$) confirms that the number of computational cores (i.e. the computational power) becomes a limiting factor and is noticeable in the performance. More specifically, Mali 400 MP GPU has a vertex shader meanwhile Tegra 3 has four. For larger levels of resolution, $L = 4$, the performance is below 20 fps for all GPUs due to the low number of vertex shaders.

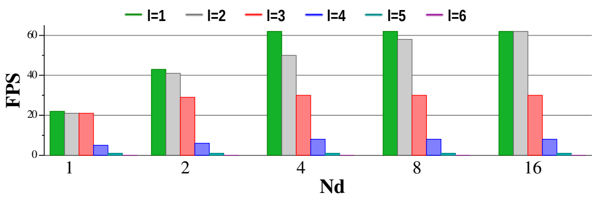
In conclusion, and taking into consideration, for example, the scene S_{20pots} with a setup of $\{L = 1, N_d = 16, N_{DP} = 40\}$ on *Mali*, we achieve only 37 fps, i.e.

a 60% of the maximum frame rate. This performance could be improved with a higher N_d, but that means a greater memory consumption. Besides, with the same scene and platform, but a configuration of $\{L = 1, N_d = 16, N_{DP} = 40\}$, we obtain a poor result of 3 fps due to the lack of computational power. None of these cases allows the use of a realistic illumination algorithm due to the limit in the number of instructions to be executed in the vertex shader [Munsh08]. The maximum number of instructions of the vertex shader across all OpenGL ES 2.0 implementations does not allow the computation of the normals, since most of the instructions are needed to evaluate the surface. On the other hand, as it is commented in [Munsh08], there is no way to query the maximum number of instructions supported by a specific vertex shader.

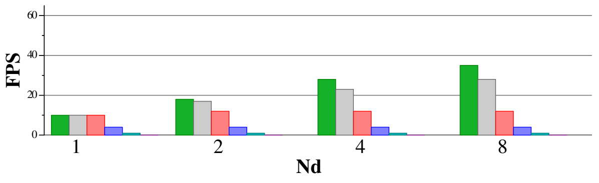
Nowadays, the Texture method cannot be tested on any market devices, as the access to texture memory from the vertex shader has yet to be implemented in Android platform. This approach would solve the main problem of the uniform method, since texture memory provides a larger storage space than uniform variables. To test how would this approach would be, we have designed a group of tests to measure the texture access latency. Although the evaluation of a Bézier surface cannot be carried out in the fragment shader, the texture memory accesses are processed in this stage to analyze the access latency. Figure 12 shows the performance of a texture memory access from the fragment shader in *Mali* and *Adreno* architectures. A simple model comprising 160 surfaces (S_{5pots}) has been chosen for this test to reduce the impact of computational power and CPU-GPU communication as much as possible. According to the results, the overhead associated with the texture access is about the 20% of the final performance in Mali architecture and under the 10% in an Adreno devices, as a unified architecture as implemented in Adreno devices, dynamically configuring its GPU cores to allocate vertex or fragment processing.

As a result, we can conclude that the proposed texture method could obtain a better performance, as the number of total draw calls could be significantly reduced.

In summary, our analysis shows the main problems of current GPUs in handheld devices, in order to achieve a rendering of Bézier surfaces. These drawbacks are the computing power, the low storage capacity and the low number of vertex shaders and their length. All this implies an important limit on the complexity of the scenes that can be managed, as well as on the realism of the rendering. Thus, simply increasing the device memory would solve the N_d/N_{DP} bottleneck. These constraints should be improved in the future designs of GPUs for these devices. A possible evolution would be a geometry shader-based pipeline, more similar to a DX10/OpenGL 3.2 architecture than to a



(a) *Mali*



(b) *Adreno*

Figure 12: Performance of scene S_{5pots} with texture access

DX11/OpenGL 4.0. This would allow the exploitation of locality in the generation of new primitives and a greater versatility, in addition to a lower power consumption.

6 CONCLUSIONS

In this paper we have presented a proposal for the rendering of Bézier surfaces on the GPU of handheld devices. Parametric surfaces cannot be directly rendered in the current GPUs of modern handheld devices, thus our first contribution is to achieve a rendering of Bézier surface. Another related contribution is to describe some of the tuning techniques employed.

Our proposal is based on parametric maps of virtual vertices, and its operation can be adapted to the hardware resources available in the GPU by tuning a series of parameters.

Additionally, an analysis of the most relevant capabilities and constraints in the graphics hardware of the current handheld devices has been carried out by tuning the main parameters of our method. This tuning permits the optimization of the memory usage of the GPU and the minimization of draw calls, that is, the CPU-GPU communication and synchronization barriers.

As a result of our analysis, we can conclude that the current graphics capabilities of these devices are far from allowing the real-time tessellation of complex parametric models. We present our proposal on an algorithmic approach, we do so with an eye toward real-time rendering on future GPUs in handheld devices. As future work, an implementation in a suitable GPU would be worthwhile. Additionally, an adaptive proposal that uses small triangles only where they are needed would better exploit our proposal on GPUs in handheld devices.

ACKNOWLEDGMENTS

This work was carried out through a HIPEAC's Collaboration grant, and all the research has been economically supported by Ministry of Education and Science of Spain under the contracts MEC TIN 2010-16735 and also by the Galician Government under the contracts 'Consolidation of Competitive Research Groups, Xunta de Galicia ref. 2010/6', and CN2012/211, partially supported by FEDER funds.

7 REFERENCES

[Boube05] T. Boubekeur and C. Schlick. Generic Mesh Refinement on GPU. In *Proc. HWWS'05: ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 99–104, New York, NY, USA, 2005. ACM.

[Chung08] K. Chung, C. Yu, D. Kim, and L. Kim. Tessellation-enabled shader for a bandwidth-limited 3D graphics engine. In *Proc. CICC'08: Custom Integrated Circuits Conference*, pages 367 –370, sept. 2008.

[Chung09] K. Chung, C. Yu, D. Kim, and L. Kim. Shader-based tessellation to save memory bandwidth in a mobile multimedia processor. *Computer and Graphics*, 33(5):625–637, 2009.

[Conch10] R. Concheiro, M. Amor, and M. Bóo. Synthesis of Bézier Surfaces on the GPU. In Paul Richard, José; Braz, and Adrian Hilton, editors, *Proc. GRAPP'10: International Conference on Computer Graphics Theory and Applications*, pages 110–115. INSTICC Press, 2010.

[Conch11] R. Concheiro, M. Amor, M. Bóo, and D. Doggett. Dynamic and Adaptive Tessellation of BÃ©zier Surfaces. In *Proc. GRAPP'11: International Conference on Computer Graphics Theory and Applications*, pages 100–105, 2011.

[Dyken09] C. Dyken, M. Reimers, and J. Seland. Semi-uniform Adaptive Patch Tessellation. *Computer Graphics Forum*, 28(8):2255–2263, 2009.

[Foley90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics: principles and practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 1990.

[Goo] Google. *Android documentation*.

[Guthe05] M. Guthe, A. Balázs, and R. Klein. GPU-Based Trimming and Tessellation of NURBS and T-Spline Surfaces. *ACM Transations on Graphics*, 24(3):1016–1023, 2005.

[Khron10] Khronos group. OpenGL ES. Technical report, 2010.

[Kim12] S. Kim, S. Yoon, S. Chung, Y. Kim, H. Kim, K. Chung, and L. Kim. A mobile 3-D display processor with a bandwidth-saving subdivider. *IEEE Trans. VLSI Syst.*, 20(6):1082–1093, 2012.

[Mali09] Mali. Mali GPU OpenGL ES. Application Development Guide. Technical report, 2009.

[Munsh08] A. Munshi, D. Ginsburg, and D. Shreiner. *OpenGL(R) ES 2.0 Programming Guide*. Addison-Wesley Professional, 1 edition, 2008.

[NVIDI11] NVIDIA. Technical Brief. Bringing High-End Graphics to Handheld Devices. Technical report, 2011.

[Piegl97] L. Piegl and W. Tiller. *The NURBS Book*. Springer, 1997.

[Roger01] D. F. Rogers. *An Introduction to NURBS with Historical Perspective*. Morgan Kaufmann, 2001.

[Sarmi12] Andrés L. Sarmiento, Margarita Amor, Emilio J. Padrón, Carlos V. Regueiro, Raquel Concheiro, and Pablo Quintía. Evaluating performance of android systems as a platform for augmented reality applications. *International Journal on Advances in Software*, 5(3&4):335–344, 2012.

[Shirl03] P. Shirley. *Fundamentals of Computer Graphics*. Addison-Wesley, 2003.