Interactive Rendering of NURBS Surfaces

Raquel Concheiro[a], Margarita Amor[a], Emilio J. Padrón[a,*], Michael Doggett[b]

[a] *Computer Architecture Group, Universidade da Coruña. Facultade de Informática, Campus Elviña S/N 15008 A Coruña, Spain*
[b] *Lund University Graphics Group, Lund University. Box 118, 221 00 Lund, Sweden*

Abstract

NURBS (*Non-uniform rational B-splines*) surfaces are one of the most useful primitives employed for high quality modeling in CAD/CAM tools and graphics software. Since direct evaluation of NURBS surfaces on the GPU is a highly complex task, the usual approach for rendering NURBS is to perform the conversion into Bézier surfaces on the CPU, and then evaluate and tessellate them on the GPU. In this paper we present a new proposal for rendering NURBS surfaces directly on the GPU in order to achieve interactive and real-time rendering. Our proposal, Rendering Pipeline for NURBS Surfaces (RPNS), is based on a new primitive KSQuad that uses a regular and flexible processing of NURBS surfaces, while maintaining their main geometric properties to achieve real-time rendering. RPNS performs an efficient adaptive discretization to fine tune the density of primitives needed to avoid cracks and holes in the final image, applying an efficient non-recursive evaluation of the basis function on the GPU. An implementation of RPNS using current GPUs is presented, achieving real-time rendering rates of complex parametric models. Our experimental tests show a performance several orders of magnitude higher than traditional approximations based on NURBS to Bézier conversion.

Keywords: NURBS Surface, Rendering pipeline, Interactive rendering

---

1. Introduction

NURBS (*Non-uniform rational B-splines*) surfaces [1] have been widely employed in CAD/CAM tools and graphics applications due to their capabilities for modeling complex geometries. In addition to the high quality of NURBS models, another advantage of the NURBS representations is the compactness of the description and, in consequence, the low storage and transmission requirements. Furthermore, graphic designers can produce models and animations in

*Corresponding author: +34 981167000 x1205
Email addresses: `rconcheiro@udc.es` (Raquel Concheiro), `margamor@udc.es` (Margarita Amor), `emilioj@udc.es` (Emilio J. Padrón), `mike@cs.lth.se` (Michael Doggett)

a simpler and faster way because they have to control fewer points than for triangle meshes. On the other hand, NURBS are easily scalable representations, 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).

In order to render NURBS surfaces on the current GPUs, these surfaces are commonly decomposed into a series of Bézier patches by the well-known technique called knot refinement [1], since they can not be directly rendered by the GPU, as we will prove. Such an approach using Bézier surfaces suffers from long preprocessing time as well as the potential introduction of artifacts, especially at the surfaces boundaries [2]. Besides, since the complete decomposition into a set of Bézier patches needs to be executed every frame for each NURBS surface with this approach, interactive rendering rates are not possible when a NURBS surface is continuously deforming or transforming [3, 4]. Therefore, Bézier surfaces are not a good replacement for NURBS surfaces in fields such as modeling, virtual reality or animation.

In the literature, recent proposals achieve real-time rendering by tessellating parametric surfaces directly on the GPU [5, 6, 7, 8]. In these proposals the rendering process is performed per patch [5, 7] or per set of patches according to the required level of detail [6]. In this approach the computational cost increases with the number of patches due to the amount of synchronous calls between CPU and GPU. [8] renders pixel-accurate Bézier surfaces using the tessellation unit of modern GPUs. Since the tessellation units added to the current GPUs do not provide a high enough level of tessellation to generate continuous surfaces with no holes from a NURBS surface. Another tessellation approach is presented in [9] where the tessellation of bi-cubic Bézier surfaces is performed following a GPGPU strategy (General-Purpose Computation on GPU) using CUDA.

A different approach to tessellate parametric surfaces is the dicing of these surfaces on micropolygons, small quadrilaterals each less than one pixel in size. The starting point of this approach is the Reyes rendering system [10], based on the development of a new and different pipeline. Although, Reyes rendering performance is far from meeting real-time requirements, different characteristics of this pipeline have been ported to GPUs [11]. Other proposals based on the modification of the GPU pipeline to implement micropolygon rendering are found in [12].

All these proposals above can not handle deforming NURBS surfaces interactively, due to the high computational cost of the transformation of a NURBS surface to Bézier patches has to be performed repeatedly in every frame as the surface is deforming.

In [13] non-uniform B-splines surfaces are tessellated on the CPU and then evaluated on the GPU to obtain positions and normal vectors. This alternative needs to use multiple fragment programs for the different surface degrees. In [14, 4] a rendering of NURBS surfaces was proposed for GPUs which only operate on existing data, since this approach is prior to the introduction of the geometry shader [15], the first shader that allowed the generation and destruction of geometry data on the GPU. In that approach, the CPU creates a set of grids that indicates the evaluation points for the different levels of detail, and these data are sent to the GPU and stored as textures. A fragment program computes the basis functions with a "ping-pong" technique which alternates between two textures. That is, $p \times q$ passes are needed for a NURBS surface of degree (p, q); for example, a bi-cubic surface point is evaluated in 16 passes. A second step multiplies these basis function values by the control points to get the surface point evaluation. Then, the resulting data are reduced in order to calculate the positions using a different fragment program. Finally, the connectivity of the points is generated on the CPU using the grid computed according to a selected level of detail. Therefore, these approaches perform a previous tessellation on the CPU.

In this paper we present RPNS, Rendering pipeline for NURBS Surface, a novel solution for the direct evaluation of NURBS surfaces on the GPU without any previous decomposition to Bézier surfaces. The objective is the efficient and interactive rendering of each surface so that the final image has no cracks or holes, neither inside each surface nor between neighbor surfaces, making it possible to exploit the parallelism of the GPU to perform common operations such as sketching on surfaces or interactive rendering of deforming surfaces [16, 17]. A new primitive, *KSQuad*, based on the regions defined by the projection on the parametric cell delimited by the different knot spans is proposed. This primitive does not need a preprocessing stage and intrinsically maintains the main geometric properties of NURBS surfaces, such as local support and strong convex hull.

In short, by means of the KSQuad primitive RPNS exploits the main features of NURBS surfaces to accomplish its real time evaluation and direct display, rivaling in quality and performance with approaches based on the REYES pipeline. Moreover, whereas other similar approaches really compute the basis functions previously on the CPU [4], RPNS goes one step forward and evaluates in real-time the whole NURBS equation in the GPU without any precomputation on the CPU. To test our proposal, and although this paper mainly focuses on algorithmic improvements to the rendering pipeline, rather than an optimized implementation, we have implemented it to measure its performance on current GPUs, achieving interactive and real-time rendering.

This rest of the paper is organized as follows: Section 2 presents KSQuad, the primitive our pipeline is based on, Section 3 introduces RPNS and Section 4 describes its implementation in DirectX11. Finally, Section 5 shows the experimental results obtained in our tests and the main conclusions are highlighted in Section 6.

2. KSQuad Primitive

In this section, we present a new primitive called KSQuad that allows a regular, flexible, efficient and interactive rendering of NURBS surfaces.

A NURBS surface is obtained as the tensor product of two NURBS curves, parametric curves that are a generalization of Bézier curves and are defined by its degree, a set of weighted control points, and a knot vector. Thus, using two independent parameters u and v, the NURBS surface of degree (p, q), respectively in both parametric directions, is given by the equation:

$$S(u,v) = \frac{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)\ N_{j,q}(v)\ w_{i,j} B_{i,j}}{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)\ N_{j,q}(v)\ w_{i,j}}, \quad 0 \leq u, v \leq 1$$

where $B_{i,j}$ are the control points, $w_{i,j}$ are the weights, $n+1$ and $m+1$ are the number of control points in u and v parametric directions, respectively, and $N_{i,p}$ and $N_{j,q}$ are the nonrational B-spline basis function defined on two knot vectors of $r = p+n+1$ and $s = q+m+1$ elements, respectively:

$$U = \left[\underbrace{0, \cdots, 0}_{p+1}, x_{p+1}, \cdots, x_{r-p-1}, \underbrace{1, \cdots 1}_{p+1}\right]$$

$$V = \left[\underbrace{0, \cdots, 0}_{q+1}, y_{q+1}, \cdots, y_{s-q-1}, \underbrace{1, \cdots 1}_{q+1}\right]$$

The basis function $N_{i,p}$ of degree p is defined for the parametric direction u as

$$N_{i,p}(u) = \frac{u - x_i}{x_{i+p} - x_i} N_{i,p-1}(u) + \frac{x_{i+p+1} - u}{x_{i+p+1} - x_{i+1}} N_{i+1,p-1}(u) \tag{1}$$

with

$$N_{i,0}(u) = \begin{cases} 1 & \text{if} \quad x_i \leq u < x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Analogously, the basis function $N_{j,p}$ of degree q is defined for the parametric direction v.

The knot vectors are non-decreasing sequences of real numbers that make a partition on the parametric domain. This partition defines the relation between different ranges of the parametric coordinates, known as knot spans or knot intervals, with the control points. Since basis functions are non-zero only in part of the domain, the functions $N_{i,p-1}$ and $N_{i+1,p-1}$, used for the computation of $N_{i,p}$, are non-zero for p knot spans, overlapping for $p-1$ knot spans.

A NURBS surface can be seen as a grid of cells in parametric space delimited by the different knot spans, with each cell containing a part of the surface computed with the non-zero basis functions in that interval. Thus, we have focused on that idea to propose a suitable input primitive that does not require a previous transformation as the seed of RPNS. This new primitive, Knot Span Quad (KSQuad), represents a half-open interval of the parametric domain, $[x_i, x_{i+1}) \times [y_j, y_{j+1})$, with non-zero length, and maintains the information of $q \times p$ neighboring knot spans, allowing an efficient evaluation of the NURBS

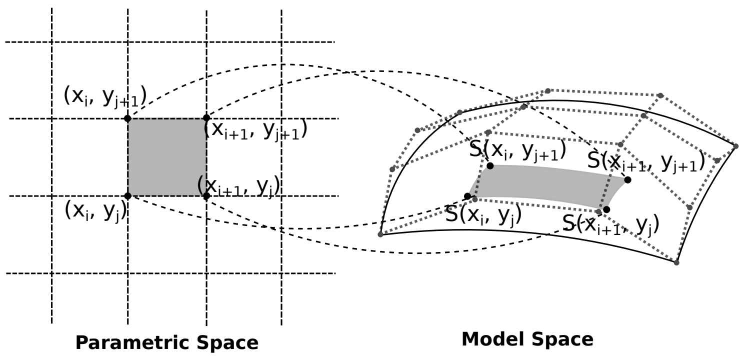


Figure 1: KSQuad primitive defined by a knot interval.

surface in this interval without any recursive computation, which makes it suitable for GPU implementation. It is important to emphasize that our proposal does not decompose the NURBS surface in any moment. Each position of the surface is directly evaluated on the NURBS surface.

A KSQuad$_{i,j}$ of degree q and p is defined like

$$\text{KSQuad}_{i,j} = \{\ \overbrace{x_i, x_{i+1}, y_j, y_{j+1}}^{\text{knot span}},\ \underbrace{B_{i-p,j-q}, \cdots, B_{i,j}}_{\text{control points}},\ \overbrace{w_{i-p,j-q}, \cdots, w_{i,j}}^{\text{weights}}\ \}$$

being $x_i \neq x_{i+1}$ and $y_j \neq y_{j+1}$.

Each KSQuad$_{i,j}$ controls a subset of the parametric domain, defined by the rectangle parametric sub-domain with corners $(x_k, y_l), k \in \{i, i+1\}, l \in \{j, j+1\}$, as illustrated in Figure 1.

A KSQuad preserves the many desirable geometric properties presented in NURBS curves and surfaces (properties the can be found in [1]), such as:

- Strong convex hull: a NURBS surface is contained in the convex hull of its control points. Moreover, if (u, v) is in the parametric rectangle defined by the knot spans $[x_i, x_{i+1}) \times [y_j, y_{j+1})$, then $S(u, v)$ is in the convex hull defined by the control points $B_{i-p,j-q}, \ldots, B_{i,j}$. This property assumes that all the weights in the NURBS surfaces are positive values, and it allows us to propose efficient culling methods on RPNS.

- Local support: $N_{i,p}(u) \cdot N_{j,q}(v) = 0$ if (u, v) is outside the rectangle $[x_i, x_{i+p+1}) \times [y_j, y_{j+q+1})$. Therefore, the influence of an individual control point over the surface is delimited to this parametric interval for each parametric direction. This feature is really interesting in our context, since it makes it possible both to reduce the computational cost of basis functions and to improve data locality. The latter is achieved as only $(p+1) \times (q+1)$ control points are used to evaluate every point in a given KSQuad, avoiding unnecessary accesses to the whole NURBS surface. Furthermore, the exploitation of the spatial coherence makes it possible that the data calculated for a given point into a cell may be reused for the rest of the points in the same cell. This saves both memory accesses and computations.

The number of KSQuad primitives generated for each surface is variable, but limited to $(r-2p)\times(s-2q)$:

$$U = \left[0, \cdots 0, \underbrace{0, \overbrace{x_{p+1}}^{}}_{KS_p}\!\!\overbrace{, x_{p+2}}^{KS_{p+1}}, \cdots \underbrace{x_{r-p-2}, \overbrace{x_{r-p-1}, 1}^{KS_{r-p-1}}}_{KS_{r-p-2}}, \cdots 1\right]$$

$$V = \left[0, \cdots 0, \underbrace{0, \overbrace{y_{q+1}}^{}}_{KS_q}\!\!\overbrace{, y_{q+2}}^{KS_{q+1}}, \cdots \underbrace{y_{s-q-2}, \overbrace{y_{s-q-1}, 1}^{KS_{s-q-1}}}_{KS_{s-q-2}}, \cdots 1\right]$$

where r and s are the number of knots for each parametric direction, respectively.

2.1. KSQuad vs. Bézier surface

The main advantages of our KSQuad-based proposals compared with the traditional decomposition of NURBS into rational Bézier patches are highlighted in this section.

NURBS surfaces are commonly decomposed into a series of rational Bézier patches by the well-known technique called knot refinement [1], since a NURBS surface can be divided into sections each one corresponding with a knot span in the knot vector. Each section can be mathematically represented as a rational Bézier surface maintaining the original shape. Each knot with multiplicity lower than the degree in each parametric direction has to be replicated in the knot vector until it appears p-times. The knot insertion algorithm inserts one into, then adds and adjust control points to yield a new description for the same curve or surface. The insertion of each new point depends on the value of the new knot added. Then, the algorithm moves the other control points near the new one to preserve the shape of the surface. Therefore, such an approach using Bézier surfaces suffers from long preprocessing times as well as the introduction of artifacts, especially at the surface boundaries. The resulting rational Bézier surfaces are defined by

$$S(u,v) = \frac{\sum_{i=0}^{p}\sum_{j=0}^{q} J_{i,p}(u)J_{j,q}(v)w_{i,j}B_{i,j}}{\sum_{i=0}^{p}\sum_{j=0}^{q} J_{i,p}(u)J_{j,q}(v)w_{i,j}} \tag{2}$$

being $B_{i,j}$ the control points, $w_{i,j}$ scalar weights and $J_{i,p}$ the p-th degree Bernstein functions.

Figure 2 depicts a cubic NURBS curve (Figure 2a) and the decomposition into piecewise Bézier segments (Figure 2b). The Bézier control points of the

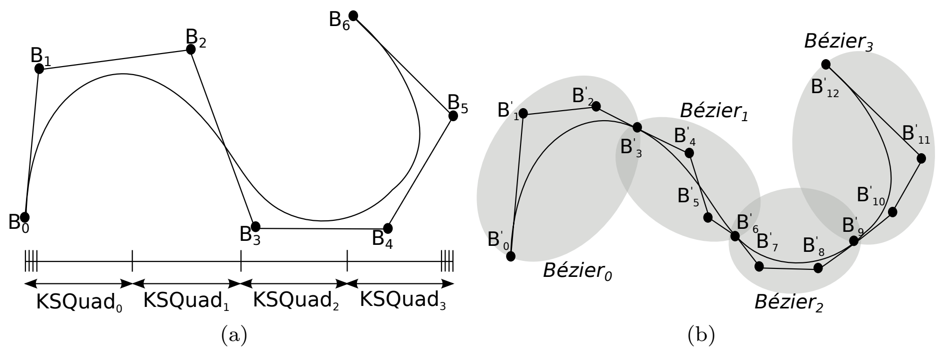


Figure 2: A cubic curve defined over $[0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4]$ (a) Four KSQuads defined over the interval knots (b) Decomposition of the curve into four Bézier curves via knot refinement.

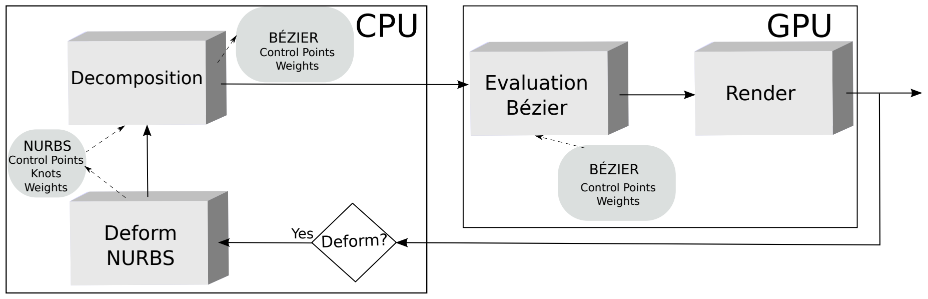
Table 1: Data of the experimental models.

Model	#NS	#KS	NURBS$_{\#CP}$	NURBS$_{\#M}$	Bezier$_{\#CP}$	Bezier$_{\#M}$	$\frac{\text{Bezier}_{\#M}}{\text{NURBS}_{\#M}}$
Killeroo	89	11532	17181	279.63	184512	2883	10.31
Head	601	15025	38464	657.34	240400	3756.25	5.71
Hinge	427	34891	61663	1016.69	558256	8722.75	8.58
Car	1364	63000	130380	2183.53	1008000	15750	7.21

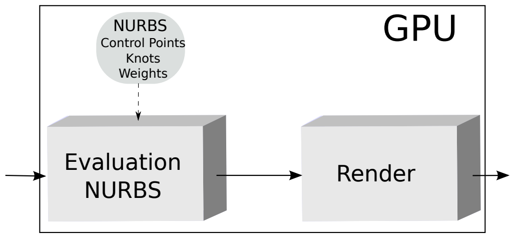
segments are obtained by insetting each interior knot until it has multiplicity p. In this example $B_0 = B'_0$ and $B_6 = B'_{12}$, the rest of control points of four Bézier surfaces are inserted or modified in the procedure of conversion.

Next, we analyze the two main advantages of the KSQuad primitive over the use of Bézier surfaces: first, the smaller memory requirements, and second, the possibility of achieving interactive deformable NURBS. Table 1 shows the main data of the models used in our experimental tests, such as the number of NURBS surfaces and KSQuads of each model, #NS, #KS (these data were obtained according to [1]). Additionally, this table also makes clear the low memory footprint of our proposal. Thus, being NURBS$_{\#CP}$ the number of control point of all the NURBS surfaces in the model, Bezier$_{\#CP}$ the number of control points of all the Bézier surfaces once applied a decomposition, and NURBS$_{\#M}$ and Bezier$_{\#M}$ the memory requirements for each case in KBytes, respectively; the last column shows how using Bézier decomposition requires significantly more control points and more memory. In the worst case, ten times more memory is used to render the model *Killeroo* with Bézier surfaces than by using the KSQuad primitive. In short, the memory requirements of our approach are significantly lower than the memory consumed by the naive approach of decomposing the NURBS surfaces into Bézier surfaces.

Regarding the possibility of obtaining interactive deformable surfaces, let us remember that NURBS properties are not projected in the new Bézier surfaces



(a)



(b)

Figure 3: Rendering of NURBS surfaces (a) Common approach based on decomposition into Bézier surfaces (b) RPNS proposal.

resulting from a NURBS-to-Bézier decomposition. For example, when the control points are moved, the level of continuity at the knots can change (increase or decrease). That is, any change in one Bézier point can reduce the continuity from C^1 to C^0 on the edge between surfaces. Moreover, in a Bézier-based modeling process the user is restricted to sketching on tangent planes instead of directly dealing with the NURBS surface, resulting in a lack of flexibility and good response feedback.

Our approach achieves interactive deformable NURBS surfaces, maintaining the local support property. Modifications in the shape due to changes of $B_{i,j}$ or $w_{i,j}$ means that the KSQuads affected, $\{KSQuad_{i,j}, \ldots, KSQuad_{i+p+1,j+q+1}\}$, access to these new values. For example, let us consider the example of Figure 2b, the modification of B'_4 implies the modification of B'_2 in order to achieve the same continuity in B'_3. However, using KSQuads we get a more compact representation of the surface and the modification of a particular control point B_2 in Figure 2b only affect the surface region $S(u)$ with $u \in [x_2, x_{2+3+1})$.

The traditional approach to the interactive rendering of deforming NURBS is outlined in Figure 3a. The transformation of NURBS surfaces into Bézier surfaces is carried out in the CPU, and these are eventually rendered by the GPU. No additional CPU-GPU data transfer is needed if the surfaces are not modified. However, if interactive deforming operations were applied, new conversions and transfers are required. Our approach (see Figure 3b) needs a unique CPU-GPU communication during a preprocessing stage.

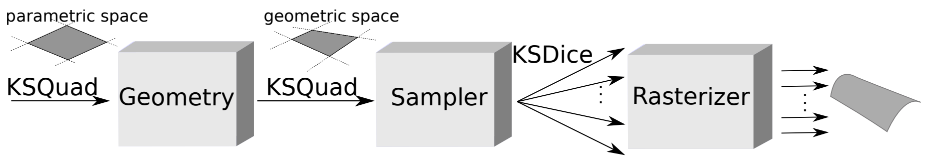


Figure 4: Generic structure of the rendering pipeline for NURBS surfaces based on KSQuad.

3. Rendering Pipeline for NURBS Surfaces

The architecture of the rendering pipeline can usually be divided into three conceptual stages: application, geometry and rasterizer. In the application stage the geometry to be rendered is generated by a software application. This results in a stream of primitives that are processed by the geometry stage, that computes what, how and where the things are drawn. Finally, the rasterizer stage renders an image; that is, sets the color for the pixels covered by the different objects in the scene.

In this section, our Rendering Pipeline for NURBS Surfaces (RPNS) is described. As it was commented in the previous section, RPNS adds a new primitive, KSQuad, to the input stream of the geometry stage. Furthermore, an intermediate stage, sampler, between the geometry and the rasterizer stages is added, as depicted in Figure 4. In this stage an adaptive sampling of the KSQuad primitives is performed according to the point of view, the geometric characteristics of the surface and the boundary edges between surfaces. Precisely, the geometry stage precedes the sampling stage in RPNS to use all this data to guide the sampling process, that is carried out by evaluating the KSQuad primitives with no approximation at all. This sampling results in a set of sampled points or dice that we have named *KSDice* and that make it possible to render the surface without cracks or holes.

The number of KSDice depends on the evaluation of each KSQuad in the geometry stage. Each KSDie consists of a sampled point and additional information such as the parametric size of the die and the degree of the corresponding surface, and it does not save any explicit connectivity information, analogously to the concept of surfels [18] in the field of point rendering. KSDie is a primitive that can be finally projected to a unique pixel or to a set of pixels.

Finally, a new explicit and non-recursive method for the evaluation of the basis function has been developed with the aim of obtaining an efficient GPU implementation. Basis function evaluation is usually one of the main bottlenecks of NURBS evaluation, since these functions have a recursive formulation and they are re-evaluated for each parametric position. Our evaluation strategy is deeply analyzed in Section 3.2.

3.1. KSDice: Adaptive sampling of KSQuad primitives

The main purpose of RPNS is the adaptive sampling of KSQuad to obtain the appropriate number of KSDice (samples) that provides a quality render with no

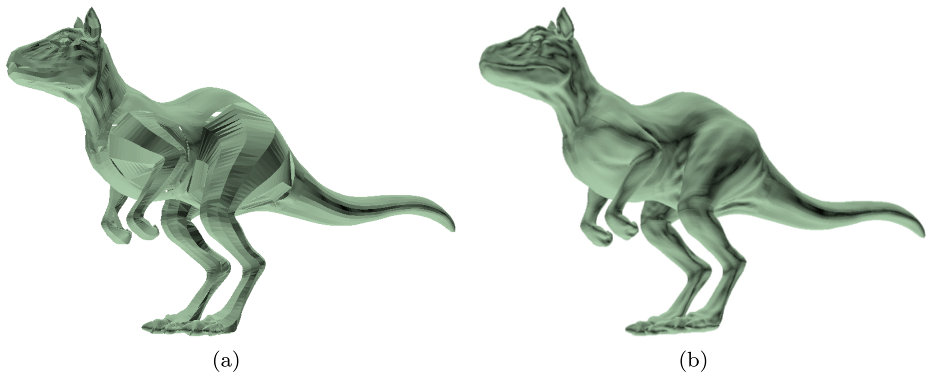


Figure 5: *Killeroo* model rendered using as input primitives: (a) NURBS surfaces (b) KSQuads.

holes, yet increasing the performance. As a matter of fact, the use of KSQuad as the input primitive to be sampled already favors this adaptive behavior, since it allows a better exploitation of the local geometric features of the surface. Therefore, it solves an important problem in current GPUs, that can not obtain enough samples for adequately covering the whole surface by directly using the NURBS surface as the input primitive of the pipeline. Thus, for example, supposing the tessellation of a NURBS surface in triangles, it is better to select a tessellation factor per KSQuad than compute a tessellation considering the whole NURBS surface. This is shown in Figure 5, where a model, *Killeroo*, is first rendered using surfaces as input primitives, after applying the maximum tessellation level implemented in current GPUs (Figure 5a), and then KSQuads (Figure 5b). As can be observed, the quality of the final image is much better with KSQuads, even though the number of triangles being used is significantly less: 12016.05 K vs. 385.56 K. Figure 5a contains numerous artifacts, such as cracks, holes and creases, since it does not have enough triangles to follow the curvature of the surface. Thus, thirty times more primitives are evaluated but a lower quality rendered model is obtained.

A KSDie obtained from a KSQuad is defined as

$$KSDie = \{(x_k, y_l), \delta_x, \delta_y, S^{p,q}, f\}$$

where (x_k, y_l) is the starting parametric coordinate of the range covered by the KSDie, so $(x_i, y_j) \leq \{(x_k, y_l), (x_k + \delta_x, y_l + \delta_y)\} < (x_{i+1}, y_{j+1})$, $S^{p,q}$ is a set of indices that provide access to the surface's data, and f is a 4-bit tag that indicates which edges of the KSDie are boundary edges with another surface.

The key to get good performance with RPNS is the number of KSDice that are sampled and rendered. An adaptive sampling that focuses on the geometric features of a surface and the number of pixels to be rendered can provide an important boost in performance with no reduction in image quality. We propose

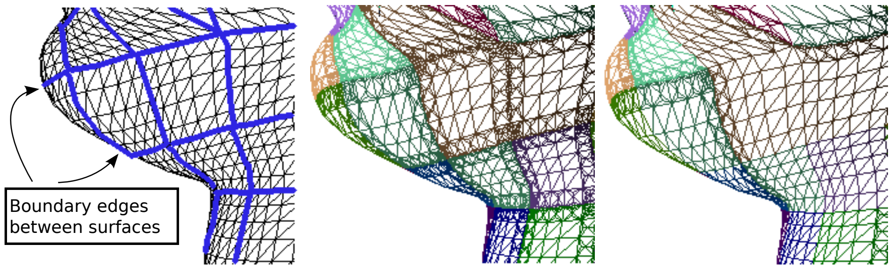


Figure 6: Boundary edges between surfaces (a) KSQuads with boundary edges (b) Oversampling all boundary edges (c) Oversampling only non G^1 boundary edges.

an adaptive sampling procedure that is based on this set of tests:

Local Area Average. The number of pixels in screen space that are covered by a KSQuad from a specific point of view is evaluated. Then, a sampling factor τ is chosen and applied in parametric space to get the appropriate number of KSDie primitives from the projected KSQuad, considering a maximum pixel-size for each KSDie of μ:

$$\begin{aligned} \mathrm{dist}(p(S(x_k, y_l)) - p(S(x_m, y_n)))/\tau < \mu, \\ \forall k, m \in \{i, i+1\} \textit{ and } k < m \\ \forall l, n \in \{j, j+1\} \textit{ and } l < n \end{aligned} \tag{3}$$

where $p()$ means a screen space projection, so the distance between the projected corners of the KSQuad is computed.

Linear Approximation. This test measures the difference between the evaluation of the NURBS surface at a point and the position where that point will be rendered if the KSDie is not further subdivided. Therefore, the maximum deviation between the KSDice to be rendered $S(u, v)$ and the ideal projection of the NURBS surface $S'(u, v)$ is computed, which provides a measure about the accuracy of the linear approximation applied, that is

$$\begin{aligned} max\ abs(p(S(u, v)) - p(S'(u, v))) < \epsilon, \\ \forall (u, v) \in [x_k, y_l) \times [x_k + \delta_x, y_l + \delta_y) \end{aligned} \tag{4}$$

Boundary Region. Since each KSQuad is independently processed in the pipeline, it is necessary to apply a higher sampling in the boundary edges between adjacent NURBS surfaces to prevent discontinuities, especially if there is not G^1 continuity in the boundary between the two surfaces. This is usually the case when different tessellation factors are applied in these surfaces. In this respect, our first approach was to introduce a test that evaluates whether a KSQuad is in a boundary edge with an adjacent NURBS surface and, if so, forces a higher sampling in the corresponding boundary regions. The information about the boundary edges of a KSQuad is coded in the field f during the creation of the KSQuad in the application stage of the pipeline. Then, to avoid cracks and holes, a boundary edge $\{S(x_k, y_l)), S(x_k + \delta_x, y_l + \delta_y)\}$ is oversampled according to:

$$max\ abs(p(S(x_k, y_l)) - p(S(x_k + \delta_x, y_l + \delta_y))) < \eta \tag{5}$$

An example is depicted in Figure 6b: thick lines mark the boundary edges between surfaces in Figure 6a, and Figure 6b shows how the boundary regions have a higher sampling.

A better solution to deal with possible discontinuities but avoiding over-tessellation is our second approach to this test, depicted in Figure 6c. This new implementation is based on the idea of *friend* surfaces, which means that two adjacent surfaces are *friends* if they are G^1 according to the necessary and sufficient conditions of G^1 continuity [19] between two adjacent NURBS surfaces with arbitrary degree and generally structured knots. In order to guarantee this *friend* condition, data defining the boundary surfaces (control points, weights and knot span) is analyzed in a preprocessing stage on the CPU but one time only, for subsequent rendering processing. This results in a *boundary region* test that only applies a higher sampling in the boundary edges of a KSQuad when it is really needed, as can be observed in Figure 6c.

These tests measure the improvement achieved in image quality with each new KSDie inserted by the sampler stage. As other similar proposals [12], our tests work with a series of thresholds that must be precomputed for each surface in order to reach the required quality level.

3.2. Explicit equations: Stair strategy

Another relevant contribution in this work is a novel approach to the computation of the basis functions of a NURBS surface based on a non-recursive strategy, called *stair strategy.*

Stair strategy provides a straightforward, efficient and general procedure with a simple control flow that makes it suitable to be implemented on current GPUs. Nowadays, NURBS surfaces evaluation on the GPU is usually based on the de Boor algorithm [4]. Thus, evaluating the p^{th}-degree B-splines basis function requires the evaluation of the B-spline basis function of order $p - 1$. In [4] the B-spline basis function of order $p - 1$ is stored as a texture on the GPU and this intermediate result is used as input for evaluating the p^{th}-degree B-spline basis function. In [2, 20] several approaches are presented to improve the performance on CPU of the computationally expensive de Boor recursion

Table 2: Nonzero basis functions on knot span $[x_i, x_{i+1})$ for $p = 5$.

5	$N_{i-5,5}$	$N_{i-4,5}$	$N_{i-3,5}$	$N_{i-2,5}$	$N_{i-1,5}$	$N_{i,5}$	0
4	0	$N_{i-4,4}$	$N_{i-3,4}$	$N_{i-2,4}$	$N_{i-1,4}$	$N_{i,4}$	0
3	0	0	$N_{i-3,3}$	$N_{i-2,3}$	$N_{i-1,3}$	$N_{i,3}$	0
2	0	0	0	$N_{i-2,2}$	$N_{i-1,2}$	$N_{i,2}$	0
1	0	0	0	0	$N_{i-1,1}$	$N_{i,1}$	0
0	0	0	0	0	0	1	0
	$i-5$	$i-4$	$i-3$	$i-2$	$i-1$	i	$i+1$

algorithm by avoiding the recursion. Our stair strategy follows a similar strategy focused on the GPU.

The basis function of a NURBS can be calculated in each parametric point by applying the Cox-de Boor recursive expression shown in Equation 1. As it was commented in Section 2, the local support property of a NURBS surface is preserved in the KSQuad primitive, which means that at most $p+1$ of the $N_{i,p}$ functions are non-zero within a given knot span $[x_i, x_{i+1})$, namely the functions $\{N_{i-p,p}, \cdots, N_{i,p}\}$. Table 2 shows the non-zero basis functions in the i^{th} knot span for $p = 5$. Hence, the only non-zero p^{th}-degree functions on this knot span are $\{N_{i-5,5}, N_{i-4,5}, N_{i-3,5}, N_{i-2,5}, N_{i-1,5}, N_{i,5}\}$.

Our proposal is based on the non-recursive reformulation of the $N_{i-k,p}$ functions, replacing the recursion that can be represented by a truncated triangular table [1] with a simple expression of additions and multiplications. Thus, the triangular table can be represented like a rectangle table with size $(p-k) \times k$. Whereas Cox-de Boor is $O(N^2)$ in the basis functions evaluation, the proposed method is $O(N)$ due to the sums for each of the basis functions. Figure 7 shows the rectangle tables for $N_{i-2,5}$ (Figure 7a) and for $N_{i-3,5}$ (Figure 7b). The basis functions $N_{i-k,p}$ with $k = \{0, \cdots, p\}$ can only be formulated according to some of the $N_{i,c}$ and $N_{i-d,d}$ with $c = \{1, \cdots, p-k\}$ and $d = \{1, \cdots, k\}$, that is, a total of p basis functions. We designate the basis functions $N_{i,c}$ as *column* functions and $N_{i-d,d}$ as *diagonal* functions. Our strategy allows a simple and efficient computation of the column and diagonal basis functions by simply applying the following expressions:

$$N_{i,c} = \frac{(u - x_i)^c}{\prod_{l=1}^{c}(x_{i+l} - x_i)} \qquad N_{i-d,d} = \frac{(x_{i+1} - u)^d}{\prod_{l=1}^{d}(x_{i+1} - x_{i-l+1})}$$

Therefore, the basis functions $N_{i-k,p}$ with $k = \{1, \cdots, p-1\}$ are formulated according to $p-k$ column functions $N_{i,c}$ and k diagonal functions $N_{i-d,d}$, with

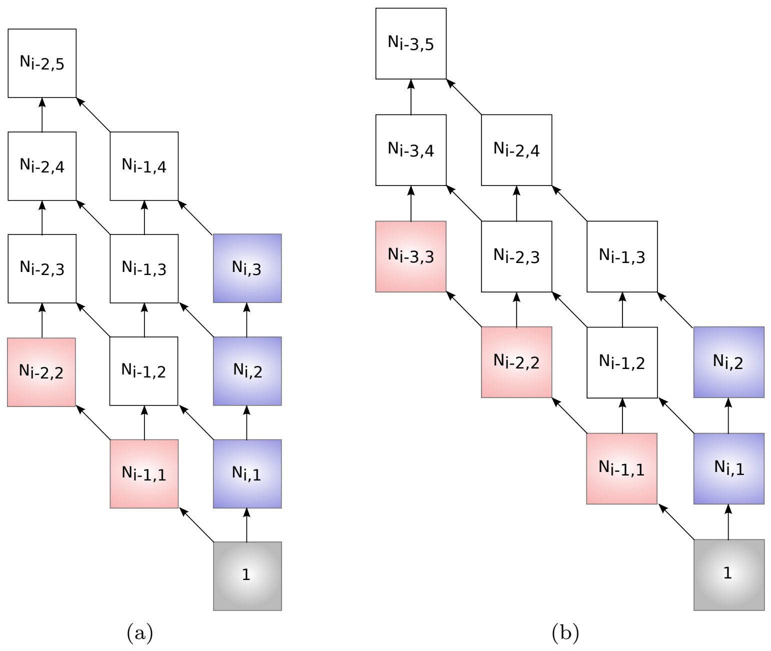


Figure 7: Stair strategy (a) $N_{i-2,5}$ (b) $N_{i-3,5}$.

$c = \{1, \cdots, k\}$ and $d = \{1, \cdots, p-k\}$:

$$N_{i-k,p} = \sum_{j=0}^{k-1} \frac{(u - x_{i-k+j})^{p-k}(x_{i+k-1+j} - u)^j}{\displaystyle\prod_{l=1}^{p-k+j} (x_{i+l} - x_{i-k+j})} N_{i-k+j,k-j}$$
$$+ \sum_{j=0}^{p-k-1} \frac{(x_{i+p-k+1-j} - u)^k (u - x_{i-k})^j}{\displaystyle\prod_{l=0}^{k-1+j} (x_{i+p-k+1-j} - x_{i-l})} N_{i,p-k-j}$$

The denominator terms are constant for each KSQuad and are computed only once for each shader invocation:

$$A_j = \frac{1}{\displaystyle\prod_{l=1}^{p-k+j} (x_{i+l} - x_{i-k})} \quad B_j = \frac{1}{\displaystyle\prod_{l=0}^{k-1+j} (x_{i+p-k+1-j} - x_{i-l})}$$

Finally, the ending non-recursive expression for a basis function is:

$$\begin{aligned} N_{i-k,p} = & \sum_{j=0}^{k-1} (u - x_{i-k+j})^{p-k} (x_{i+p-1+j} - u)^j A_j N_{i-k+j,k-j} \\ & + \sum_{j=0}^{p-k-1} (x_{i+p-k+1+j} - u)^k (u - x_{i-k})^j B_j N_{i,p-k-j} \end{aligned}$$

4. RPNS with DirectX11 on current GPUs

The Geometry Shader (GS) introduced with DirectX10 was the first stage in the graphics pipeline capable of generating new primitives on the GPU. Although this programmable stage exploits data locality and allows an efficient tessellation on the GPU, it is highly limited by the number of output primitives that can be created for each input primitive, since the maximum size of its output stream is 1024 bytes per invocation.

The main limitations of the GS were solved by the introduction of a new configurable stage in DirectX11: the Tessellator. This stage can create up to 64 samples per edge, but needs two additional programmable stages in the rendering pipeline: Hull Shader (HS) and Domain Shader (DS). The HS is called once for each input primitive in the pipeline, KSQuad in our implementation, and this is the stage in charge of configuring the Tessellator. Culling can also be performed in this stage.

The new primitives generated by the tessellator are sent to the DS, so this stage is called once for each KSDie in our implementation. The DS receives both the parametric positions created by the tessellator and the KSQuad data directly from the HS. The four corners of each KSDie, $S(x_k, y_l)$, $S(x_k + \delta_x, y_l)$, $S(x_k, y_l + \delta_y)$, and $S(x_k + \delta_x, y_l + \delta_y)$, are efficiently evaluated in the DS by taking advantage of access locality and avoiding redundant computations. Let us emphasize that like Reyes vertex shading, RPNS also allows the user to specify an arbitrary shading rate. In Reyes, shading rate is expressed in samples per pixel meanwhile we specify samples per KSDie in RPNS, with a value of 4.0 in our implementation. A more efficient RPNS implementation would adaptively choose the shading rate per KSDie, but that objective is beyond the scope of this paper.

The output from the DS is sent to the GS, where two triangles are generated for each KSDie due to the triangle-oriented graphics pipeline of current GPUs. Furthermore, to optimize the rendering of NURBS surfaces in RPNS we propose a backface and view frustum culling in the GS, called Exact Visible Set (EVS). This culling stage has a high impact on the overall performance, since it achieves an important reduction in the number of KSDice to get rasterized.

Although DirectX11 introduces a *patch* primitive to deal with parametric surfaces, this primitive is only suitable to work with simple and regular surfaces such as bi-cubic Bézier surfaces, where only the positions of the control points need to be stored and the number of control points can be deduced from the

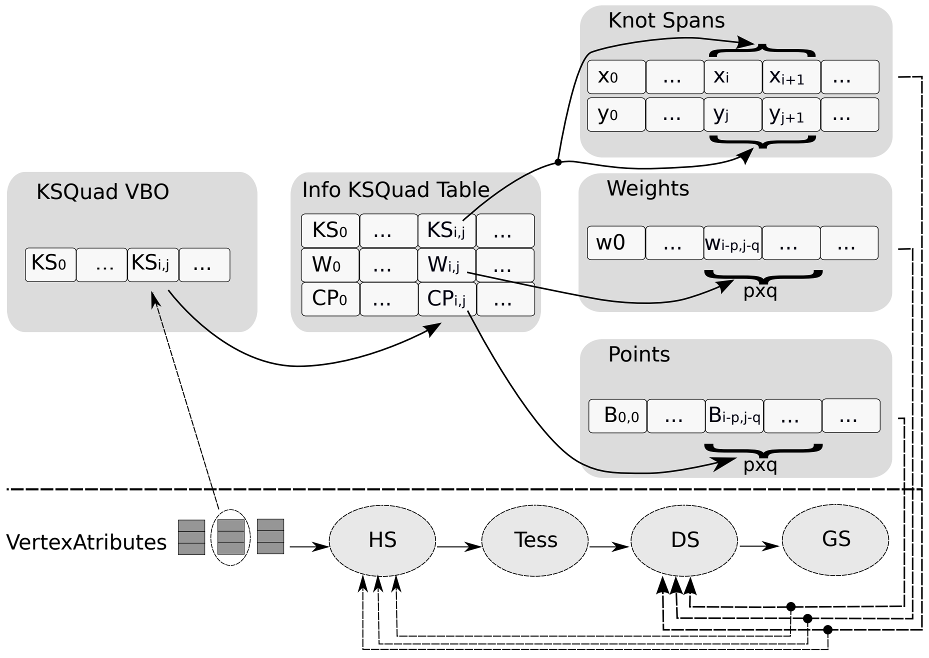


Figure 8: Memory layout of the data structures.

surface degree. Due to the inherent complexity of the NURBS surfaces, we need to define a storage layout in texture memory more complex than the one available through the patch primitive. As shown in Figure 8, our proposal uses two indirection levels to access all the data required to work with a KSQuad primitive: firstly, an access to the *Info KSQuad Table*, which contains indices to the NURBS surface data needed for each KSQuad, and then the access to the surface data using those indices. Both the Info KSQuad Table and the rest of the surface data (knot spans, weights and control points) are stored in texture memory. Although this arrangement needs a double memory access, it saves an important number of CPU-GPU transfers. Other alternatives could be easily implemented, such as sending all the required data for each KSQuad via the vertex buffer.

Every KSQuad primitive that enters in the pipeline has enough information to access to the Info KSQuad Table and fetch all the needed data from texture memory to completely evaluate the KSQuad. The access to all this information is needed in the HS and DS stages (see Figure 8), that correspond to the input of the Geometry and Sampler stages in RPNS, as shown in Figure 4.

We have implemented and tested three different alternatives to map RPNS on DirectX11, one exploiting the GS capability of generating new geometry, and the other two that use the HS, Tessellator and DS stages to overcome the GS limitations. One of these two proposals implements a non-uniform version of RPNS that focuses on minimizing the number of generated primitives, whereas the other one obtains a uniform result.

GS-based RPNS. Our first proposal for the implementation of RPNS on the DirectX pipeline focuses on the GS stage. Thus, even though KSQuads evaluation in the GS makes it possible to reuse part of the computations, the bounds in the maximum number of KSDice that can be generated for each input KSQuad make impossible to achieve high quality renders. This proposal is not shown in the section of results.

Uniform RPNS (RPNS-U). This implementation uses the HS, Tessellator and DS stages to map the stages in RPNS. The work in HS is done with a KSQuad granularity, fetching the necessary information from texture memory. This stage applies the *local average area* test and, optionally, a previous culling. This test is used to set the subdivision factor in the tessellator that guarantees a maximum size (in pixels) for the KSDice to be generated (Equation 3). All this work corresponds with the geometry stage of RPNS.

Once the KSDice are created by the tessellator, they are sent to the DS. Thus, the DS is called once for each (still empty) KSDie. In this stage, each KSDie is evaluated in the NURBS surface using the *stair strategy* described in Section 3.2. This stage together with the tessellator correspond with the sampler stage of RPNS.

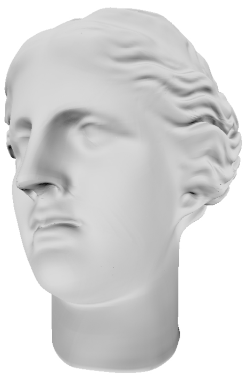
Non-uniform RPNS (RPNS-NU). This implementation of RPNS follows the same structure than the previous one, but with the addition of a GS stage that implements the adaptability in the sampler stage of RPNS [21]. In this case, the HS previously sets the tessellator to create a fewer number of KSDice. The output from the DS is sent to the GS, where the *linear approximation* and the *boundary region* test are used to guide the subdivision level applied to each KSDie. Thus, the linear approximation test (Equation 4) assures that a higher subdivision level is applied to non-flat regions. Besides, the boundary region test (Equation 5) detects the regions of KSQuads that are boundaries to other surfaces and applies the highest subdivision factor to prevent cracks between adjacent surfaces.

5. Experimental Results

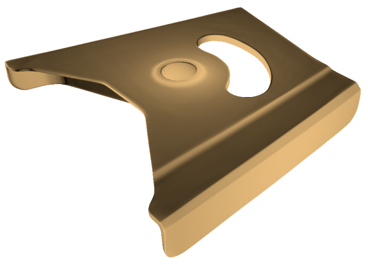
In this section we present the results obtained with different versions of our implementations of RPNS on GPU. Our test platform is an Intel Core 2 Duo $2.4GHz$ with $2GB$ of RAM and a nVidia Geforce 580 GTX with DirectX11 Microsoft's HLSL. The models used in our tests are shown in Figure 9. The final images were rendered with a screen resolution of 2048×1152 pixels.

Our first experiment is focused on measuring the effectiveness of the adaptive process that our KSQuad primitive allows us to introduce in RPNS; specifically we concentrate on the RPNS-U approach. The results obtained from the tests are shown in great detail in Table 3, Table 4 and Figure 10. The experiments

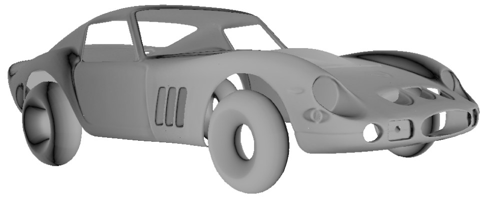
(a)



(b)



(c)



(d)

Figure 9: Test models: (a) *Killeroo* (b) *Head* (c) *Hinge* (d) *Car*.

Table 3: #KS (in k) with different μ values for the four test models

Killeroo	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	1564.79	485.76	178.47	83.16	51.20	45.05
EVS	847.16	261.89	95.72	44.18	26.98	23.95
	54.14 %	53.91 %	53.63 %	53.13 %	52.70 %	53.17 %
Head						
No Culling	2309.04	742.09	225	115.96	69.09	58.69
EVS	1223.83	368.87	131.85	59.78	34.96	29.74
	53.00 %	49.71 %	58.60 %	51.55 %	50.59 %	50.68 %
Hinge						
No culling	4245.46	1593.56	576.54	263.69	163.00	139.56
EVS	1578.48	716.35	252.02	83.96	66.78	55.57
	62.81 %	55.04 %	56.28 %	68.16 %	59.02 %	60.18 %
Car						
No Culling	4448.50	1493.51	607.11	332.07	257.90	246.09
EVS	2520.33	839.08	311.96	180.35	136.83	123.05
	56.66 %	56.18 %	51.38 %	54.31 %	53.06 %	50.00 %

were carried out for the four test models with different values of the threshold μ (maximum pixel-size for each KSDie, see Equation 3).

Table 3 indicates the thousands of KSDice that are rendered, #KS, whereas Table 4 details the PSNR [1] (Peak Signal-to-Noise Ratio in dB) values obtained for each case. Both tables present the results either applying EVS-based culling or not: the row labeled as *EVS* shows the results when the backface and view frustum culling are enabled in the GS (see Section 4). The third row in Table 3 is the percentage of KSDice that are eliminated after the culling and, therefore, are not rendered. The columns $\mu = 1$ to $\mu = 16$ represent how results vary with different thresholds in the number of pixels in screen space covered by a KSQuad (see Equation 3), with the last column showing the results when the KSQuad is not sampled and directly generates a KSDie. As can be observed, similar results have been obtained in all cases for the four test models.

An adaptive sampling of KSQuad allows the utilization of larger KSDice where the geometry is less detailed, concentrating smaller KSDice at silhouette edges and curves. Regarding the performance of our implementation of RPNS in the pipeline of current GPUs, the graphs of Figure 10 show the good results

[1] Peak Signal-to-Noise Ratio is the distortion between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. In this case, the distortion is measured with respect to the model rendered with the maximum tessellation factor PSNR $= 20 \cdot \log_{10}(\text{MAX}/\sqrt{\text{MSE}})$.

Table 4: PSNR with different μ values for the four test models

Killeroo	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	44.51	42.95	40.75	39.28	39.30	36.74
EVS	44.50	42.95	40.75	39.28	39.30	36.72
Head						
No Culling	42.37	43.93	42.97	40.87	40.53	36.26
EVS	42.36	43.86	42.57	40.87	40.51	36.26
Hinge						
No Culling	41.89	41.39	39.77	40.94	40.30	38.77
EVS	41.02	41.65	39.94	40.18	39.92	37.92
Car						
No Culling	38.64	38.35	40.47	38.47	37.58	30.45
EVS	38.53	38.34	40.45	38.44	37.57	30.17

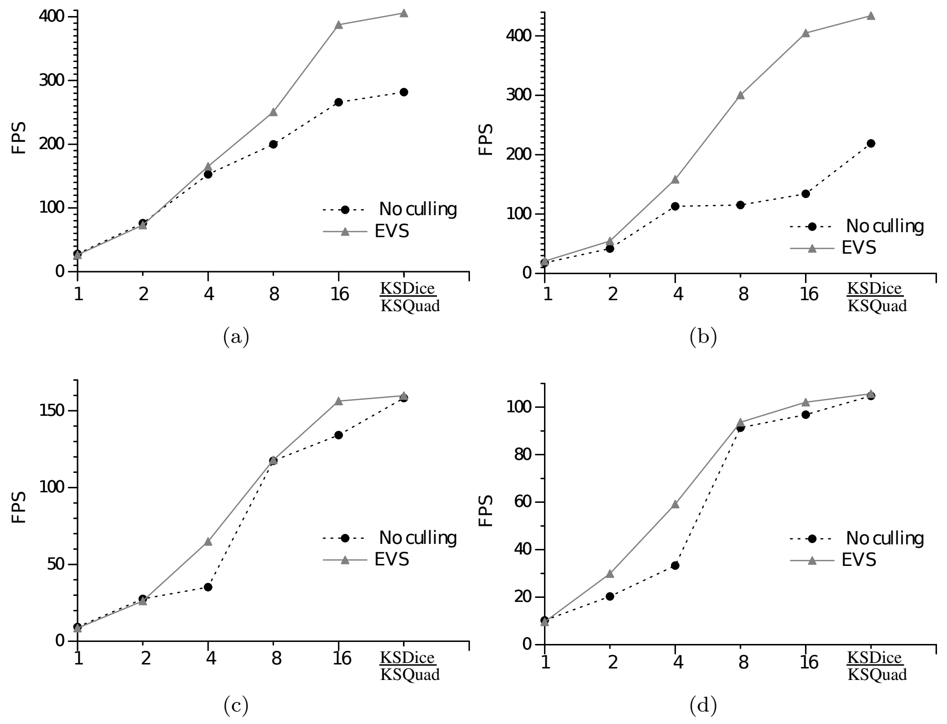


Figure 10: Frame rate with the different culling approaches for the four test models: (a) *Killeroo* (b) *Head* (c) *Hinge* (d) *Car*.

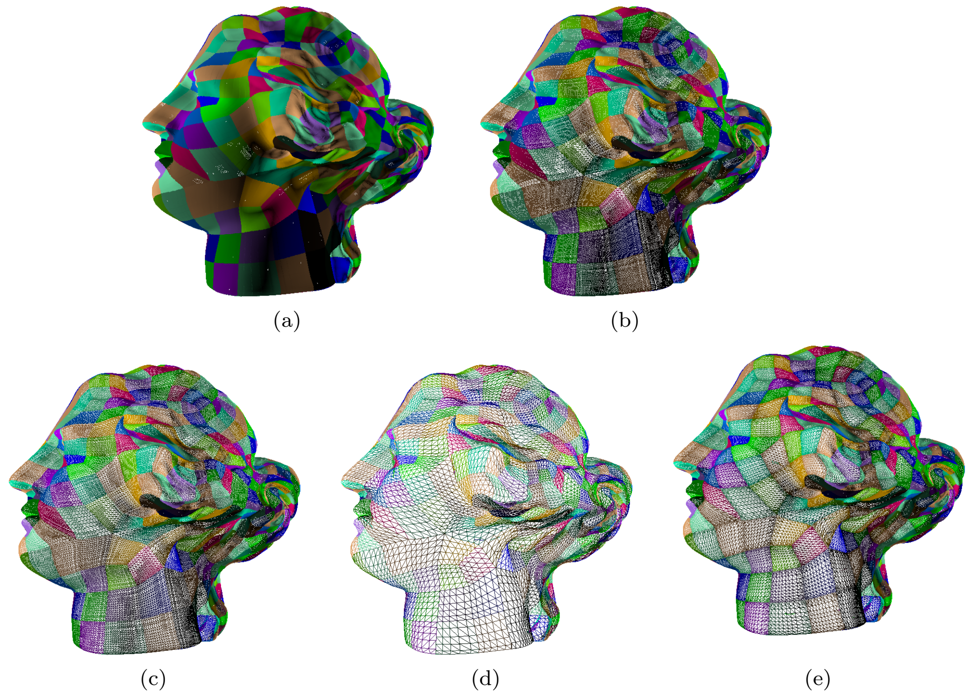


Figure 11: *Head* model rendered with different screen areas (a) $\mu = 1$, (b) $\mu = 8$ and (c) $\mu = 16$ (d) 1 KSDie per KSQuad (e) RPNS-NU

in terms of frame rate obtained by RPNS with $\mu > 1$ for the four test models. Thus, for example, 115.96 K KSDice are generated with no culling and $\mu = 8$ for the *Head* model, achieving a frame rate of 134.128 fps that gets increased to 404.93 fps with the EVS solution. This results in speedups of 7.49 x and 22.62 x, respectively, while maintaining rendering quality, as shown in Figure 11. Both strategies (No culling and EVS) achieve good quality results, always over 30 dB and nearby 40 dB for $\mu \leq 16$, with no significant quality loss as shown in the video attached.

On the other hand, the table and the graphs show that the introduction of EVS culling dramatically improves the performance of the pipeline, with speedups of more than 3x in the frame rate in some cases, and without decreasing the quality in the render, since the PSNR values are mostly identical. The removal of non-visible KSDice, backface or outside the current view frustum, with very simple computations in the GS is worth the additional computation in the rasterization stage.

The performance of several RPNS implementations is analyzed in Table 5 in terms of frame rate. This table shows the frames per second obtained by rendering the four test models using five different implementations of RPNS on DirectX11, with the speedup values in parentheses. The first method, *de Boor*,

Table 5: Frame rate (FPS) of the different RPNS implementations.

Model	de Boor	Stair Strategy	RPNS-U	(GS) RPNS-NU	(HS) RPNS-NU
Killeroo	118.34	**(1.22)** 144.39	**(4.12)** 487.54	31.99	**(1.57)** 186.25
Head	120.47	**(1.11)** 134.13	**(3.36)** 404.93	23.66	**(2.37)** 285.86
Hinge	82.74	**(1.62)** 134.16	**(1.88)** 156.30	9.75	**(2.00)** 165.78
Car	61.71	**(1.60)** 96.84	**(1.65)** 102.09	5.45	**(1.90)** 116.96

is the result of applying EVS culling on the KSQuads in the GS, evaluating the NURBS surfaces in a traditional way with $\mu = 16$. Only by changing the surface evaluation to our *stair strategy*, second column in the table, we achieve a significant improvement in performance. The next two columns, *RPNS-U* and *RPNS-NU (GS)*, correspond to the implementations proposed in Section 4. As can be observed, in general the uniform approach obtains the best results, since it allows to exploit the HS, tessellator and DS by using regular operations. However, the non-uniform proposal, based on some conditional branches, introduces a greater divergence in the control flow of the GS stage, in addition to call a supplementary kernel, what results in an important dropping of performance. Finally, the last column (*RPNS-NU (HS)*) is a less adaptive implementation of our non-uniform approach, now implemented in the HS by adapting the behavior of the tessellator to improve the number of KSQuads in the regions with important variations of the linear approximation test.

In order to perform a comparison between our approach and the common approximation based on performing the decomposition of a NURBS surface into a series of Bézier patches in the CPU, the test models were converted to Bézier surfaces. For example, the *Killeroo* model was decomposed into 10935 bi-cubic Bézier surfaces by means of the “Rebuild Surfaces” function in Autodesk Maya. Our RPNS approach needs only the 10.93% of the total amount of memory the decomposition solution uses, which also means a higher performance due to a better exploitation of the memory hierarchy. However, it is in the real bottleneck of this kind of solutions, the CPU-GPU transfer when the surfaces are interactively modified, where RPNS performance is especially favorable: the performance drops up to 0.18 FPS, even though we are not considering the NURBS-Bézier conversion time, whereas RPNS obtains up to 487.54. Thus, there is no significant performance degradation in our proposal when the surface models are interactively deformed.

As long as we know, there is no other proposal to render NURBS on the GPU without any previous preprocessing step. In [4] a previous tessellation is generated on the CPU, whereas the basis functions are calculated on the GPU with several fragment programs and passes through the fragment shader. In order to compare this proposal and RPNS, we have tested the 16 passes with the ping-pong technique detailed in [4], but without any computation or texture access. Let us emphasize that performing only such technique on the *Killeroo* model results in 246.98 FPS, whereas our proposal obtains 487.54 FPS (see

Table 5) with a complete rendering.

In short, even though our main objective is to evaluate the robustness of all the algorithmic proposals behind RPNS, we have proved that high quality results can be obtained using our novel primitive KSQuad and RPNS, doing a test implementation on current GPUs and achieving real-time rendering for complex models. Let us emphasize that a hardware implementation of RPNS would attain a higher performance than the one obtained with our software implementation in this work.

6. Conclusions

In this work a proposal of a new pipeline for the efficient rendering of NURBS surfaces, RPNS, is presented. Our pipeline is based on a new primitive, KSQuad, that provides a regular and flexible management of NURBS surfaces but maintaining their main geometric properties. This primitive allows an efficient rendering of the NURBS surface in all its parametric knot spans, what is especially suitable to achieve high performance GPU implementations.

RPNS performs an efficient adaptive discretization of KSQuads into KSDice, what allow us to fine tune the density of primitives needed to avoid cracks and holes in the final image, in addition it applies an efficient non-recursive evaluation of the basis function of a NURBS surface on the GPU. Different GPU implementations of RPNS are proposed on DirectX11, exploiting the programmable and configurable stages of current graphics hardware to achieve interactive and real-time rendering rates of complex parametric models. Our experimental tests show that RPNS is several orders of magnitude faster than traditional approximations based on NURBS to Bézier conversion when considering the interactive handling of rendered models.

To optimize the rendering of NURBS surfaces in RPNS as well as a proof of the versatility of the KSQuad primitive, we will include backface and view frustum culling in future work. Finally, let us emphasize results show that our method is more efficient than existing methods.

References

[1] L. A. Piegl, W. Tiller, The NURBS Book, Springer, 1997.

[2] O. Abert, M. Geimer, S. Müller, Direct and fast ray tracing of NURBS surfaces, in: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 161–168.

[3] F. W. B. Li, R. W. H. Lau, M. Green, Interactive rendering of deforming NURBS surfaces, Computer Graphics Forum 16 (3) (1997) 47–56.

[4] A. Krishnamurthy, R. Khardekar, S. McMains, Optimized GPU evaluation of arbitrary degree NURBS curves and surfaces, Comput. Aided Des. 41 (12) (2009) 971–980.

[5] R. Concheiro, M. Amor, M. Bóo, Synthesis of bézier surfaces, in: International Conference on Computer Graphics Theory and Applications (GRAPP2010), 2010, pp. 110–115.

[6] C. Dyken, M. Reimers, J. Seland, Semi-uniform adaptive patch tessellation, Computer Graphics Forum 28 (8) (2009) 2255–2263.

[7] M. Guthe, A. Balázs, R. Klein, GPU-based trimming and tessellation of nurbs and t-spline surfaces, ACM Trans. Graph. 24 (3) (2005) 1016–1023.

[8] Y. I. Yeo, L. Bin, J. Peters, Efficient pixel-accurate rendering of curved surfaces, in: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D'12), 2012, pp. 1–8.

[9] C. Eisenacher, Q. Meyer, C. Loop, Real-time view-dependent rendering of parametric surfaces, in: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games(I3D09), 2009, pp. 137–143.

[10] R. L. Cook, L. Carpenter, E. Catmull, The reyes image rendering architecture, SIGGRAPH Comput. Graph. 21 (1987) 95–102.

[11] K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, B. Guo, Renderants: Interactive reyes rendering on GPUs, ACM Trans. Graph. 28 (2009) 155:1–155:11.

[12] M. Fisher, K. Fatahalian, S. Boulos, K. Akeley, W. R. Mark, P. Hanrahan, DiagSplit: Parallel, crack-free, adaptive tessellation for micropolygon rendering, ACM Trans. Graph. 28 (2009) 150:1–150:10.

[13] T. Kanai, Fragment-based evaluation of non-uniform b-spline surfaces on GPUs, Computer-Aidded Design and Applications 4 (1-4) (2007) 287–294.

[14] A. Krishnamurthy, R. Khardekar, S. McMains, Direct evaluation of NURBS curves and surfaces on the GPU, in: Proceedings of The 2007 ACM Symposium on Solid and Physical Modeling (SPM'07), 2007, pp. 329–334.

[15] D. Blythe, The Direct3D 10 system, ACM SIGGRAPH 2006 25 (2006) 724–734.

[16] A. Krishnamurthy, S. McMains, K. Haller, GPU-accelerated minimum distance and clearance queries, IEEE Transactions on Visualization and Computer Graphics 17 (2011) 729–742.

[17] A. Krishnamurthy, S. McMains, I. Hanniel, GPU-accelerated hausdorff distance computation between dynamic deformable NURBS surfaces, Computer-Aided Design 43 (11) (2011) 1370–1379.

[18] H. Pfister, M. Zwicker, J. van Baar, M. Gross, Surfels: surface elements as rendering primitives, in: Proceedings of the 27th annual conference on Computer graphics and interactive techniques (SIGGRAPH'00), 2000, pp. 335–342.

[19] X. Chea, X. Liangb, Q. Lib, G^1 continuity conditions of adjacent nurbs surfaces, Computer Aided Geometric Design 22 (4) (2005) 285–298.

[20] K. Jankauskas, Time-efficient NURBS curve evaluation algorithms, in: Proc. of the 16th International Conference on Information and Software Technologies (IT 2010), 2010, pp. 60–69.

[21] M. Bóo, M. Amor, R. Concheiro, M. Doggett, Efficient adaptive and dynamic mesh refinement based on a non recursive strategy, The Computer Journal 56 (7) (2013) 843–851.