Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

Euro-Inf Bachelor awarded by EQANIE

Culling Techniques for Deformable NURBS Surfaces

Estudante: David Lema Núñez

Dirección: Margarita Amor López

Raquel Concheiro Figueroa

A Coruña, February de 2023.

Dedicado á constante busca do coñecemento

Acknowledgements

Gustaríame agradecer á miña familia polo seu apoio durante esta carreira, así como aos amigos e compañeiros que me axudaron durante a súa realización.
Tamén me gustaría agradecer o tempo e esforzo prestado polas miñas titoras para o desenvolvemento deste traballo.

Abstract

NURBS surfaces are a standard representation of models typically used in computer-assisted design (CAD), naval engineering, car engineering, or medical-processed images. Furthermore, NURBS surfaces have many advantages, a small modification on some control points result in highly complex models. However there is no guarantee of obtaining an high enough level of tessellation in order to avoid coarse models, which is why in the following project it will be described the use of different culling techniques that might help to reduce the unnecessary computation of no visible surfaces and in consequence the achievement of a better performance with fewer artefacts.

Resumo

As superficies NURBS son unha representación estándar de modelos que se usan normalmente en deseño asistido por ordenador (CAD), enxeñaría naval, enxeñaría de automóbiles ou imaxes procesadas para uso médico. Adicionalmente estas ofrecen moitas vantaxes, debido a que con pequenas modificacións nalgúns puntos de control resultan en modelos moi complexos. No entanto, non hai garantía de obter un nivel suficientemente alto de teselación para evitar modelos con redes toscas, polo que no seguinte proxecto describiranse o uso de diferentes técnicas de culling, que axudarán a reducir o cómputo innecesario de superficies non visibles e en consecuencia a obter un mellor rendemento con menos artefactos.

Keywords:

- Nurbs
- Culling
- Tessellation
- DirectX
- KSQuad
- Meshing Shading

Palabras chave:

- Nurbs
- Culling
- Teselación
- DirectX
- KSQuad
- Meshing Shading

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 1

1.3 Memory structure 2

2 Tessellation of parametric surfaces **3**

2.1 NURBS surfaces 3

2.1.1 NURBS curves 4

2.1.2 NURBS surfaces 5

2.2 GPU Tessellation 7

2.2.1 DirectX11 and the traditional pipeline 7

2.2.2 DirectX12 and Meshing Shading 9

2.3 Related Work 9

3 Methodology and Planification **11**

3.1 Methodology 11

3.2 Planification 11

3.2.1 Increment 1 12

3.2.2 Increment 2 12

3.2.3 Increment 3 12

3.2.4 Increment 4 12

3.2.5 Increment 5 12

3.2.6 Increment 6 13

3.2.7 Increment 7 13

3.2.8 Gantt Diagram 13

3.3 Project Costs 15

4 Culling Techniques for NURBS surfaces using Meshing Shading **17**
4.1 Rendering of NURBS surfaces using KSQuad 17
4.2 Culling Techniques on Rendering Pipeline for NURBS Surfaces 22
4.3 Culling Techniques Implementation in DirectX12 using Meshing Shading . . . 22
4.3.1 KSQuad Level Culling 23
4.3.2 Surface Level Culling 26

5 Experimental Results **28**
5.1 Test Environment 28
5.2 Performance results 29

6 Conclusions and Future Works **32**
6.1 Future works 32

A Code Access **35**

Bibliography **37**

List of Figures

2.1 Cubic curve represented as (a) a group of *Bézier* or (b) a NURBS curve [1, 2] . 4
2.2 Bi-quadratic NURBS surface example [1] 6
2.3 DirectX 11 pipeline stages [1] 7
2.4 Tessellation structure on DirectX11 pipeline [1] 8
2.5 DirectX12 pipeline using *Meshing Shading* 9

3.1 Gantt diagram 13
3.2 Detail of each task in the Gantt digram 14

4.1 Levels of tessellation 18
4.2 Threading group schema example of any DirectX shader with $4\times3\times2$ *SV_Groups* and a number of $10 \times 3 \times 2$ threads. 19
4.3 KSQuad position of surface depending of the number of knots at U and V . . . 19
4.4 KSQuad primitive defined by a knot span [1] 22
4.5 *Back-face* culling example with two surfaces, Surface a facing the camera and Surface b not facing the camera 23

5.1 Test model (*Head*) 29
5.2 Speed up of culling techniques in relation to No Culling Approach (a) *GS* platform and (b) *PL* platform 30
5.3 Comparison of the *Head* model (a) using [3] proposal and our (b) proposal . . 30

List of Tables

3.1 Human costs . . . 15
3.2 Cost and time details of the resources used . . . 15
3.3 Total costs of the project . . . 16

5.1 Description of the test platforms . . . 28
5.2 Performance comparison . . . 29

Chapter 1

Introduction

The capability of this NURBS (Non-Uniform Rational B-Spline) surfaces to render highly detailed models has led to its use in areas such as computer graphics, CAD (Computer-Aided Design) or medicine. However, such level of realism has originated performance issues, specially on particularly complex models. Thus to overcome this issues the use of culling techniques would be a way of regaining performance while preserving the advantages of NURBS surfaces.

1.1 Motivation

The DirectX11's pipeline is able to render NURBS surfaces, even though it has limitations to directly render them, furthermore as article [4] shows culling as a way to reduce the overhead is harder to archive at the start of this pipeline. Thus, as DirectX12 introduced the concept of *Meshing Shading* with the objective to overcome said limitations and as at [3] project many aspects of tessellation were implemented at DirectX12 new pipeline, this project continues the exploration for a more optimal way of rendering the widely used NURBS surfaces.

1.2 Objectives

The objective of this project is the analysis and comparison of different culling techniques using DirectX12 API, which allows an interactive and realistic representation of big deformable models. This analysis is based in the surface rendering and culling of NURBS surfaces on RPNS (Rendering Pipeline for NURBS surfaces) [4] in DirectX11. The main subgoals are:

- Understanding of NURBS surfaces and their properties, so the proposed culling techniques are able to use the required properties to properly identify the regions on the same plane in the surfaces.

- Analysis of the different culling techniques to determine the measures needed to adapt them to this project, considering the additional complexity from acting over a plane rather than over specific triangles.
- Study of the main function of DirectX12 API [5], the interface used to program the GPU (Graphical Processing Unit) and develop the propositions.
- Code analysis of RPNS (Rendering Pipeline For NURBS Surfaces) [4], developed by the Computer Architecture Group (GAC).
- Reimplementation of the *Mesh Shader* to archive KSQuad [4] level rendering, archiving a higher level of tessellation an quality of image.
- Analysis, design and implementation of the different culling techniques via the *Amplification Shader*.
- Performance analysis of the introduced changes via FPS (Frames Per Second) and speed up comparison on different architectures per each culling technique.

1.3 Memory structure

This memory is organized with an introduction chapter 1 exposing the project theme, its motivation and its objectives. Chapter 2 is an explanation about the NURBS surfaces characteristics, a standard on CAD (Computer Aided Design) as they are flexible and compact. Moreover, GPU tessellation and different DirectX pipelines are also detailed.

Chapter 3 details the followed methodology with the cost estimations that might have occurred to determine the viability of our proposals. These proposals are exposed in chapter 4, detailing the modifications applied to increase the tessellation factor in the used test models (see 5.1) and detailing the *Back-face* based culling techniques implemented.

In chapter 5 the results from testing our propositions on different models and architectures are exposed and contrasted with previous propositions. In addition, our conclusions about the data are detailed and in chapter 6 a project conclusion is exposed with possible continuations for this line of investigation.

Chapter 2

Tessellation of parametric surfaces

The need of real-time rendering of complex models in different areas has led to a widespread use of parametric surfaces based objects [1] in contrast to the traditional approach oriented to triangle rendering. DirectX11 (released in 2009) was specifically design to render parametric surfaces on real time, but as NURBS surfaces rendering is not straightforward different proposals were develop to overcome the issue, such as NURBS surface decompositions into *Bézier* surfaces [1] and surface subdivision techniques [6, 7]. Nevertheless, the attain quality is not enough with the complexity demand, as Bézier surfaces have limitations for models like spheres [8].

The remaining chapter is organized as follows: Section 2.1 presents the basic characteristics of NURBS surfaces, section 2.2 explains the evolution of the tessellation process via the DirectX traditional and new pipeline.

2.1 NURBS surfaces

On this section, there is a brief explanation about NURBS surfaces and their properties, it is important to note that the following statements are a summary of references [1] and [9].

To start with, NURBS surfaces have become the *the facto* standard on CAD/CAM (*Computer-Aided Design*/*Computer-Aided Manufacturing*) due to their ability to represent high complex curves, in combination with their design flexibility and their easy modifiable shape by changing their control points and weights [1].

In Figure 2.1 is easy to appreciate the superiority of NURBS surfaces to represent a curve. Figure 2.1a requires more control points and subdivisions into groups to render the same curve of 2.1b, which with a reduced number of control points is able to produce the same result.

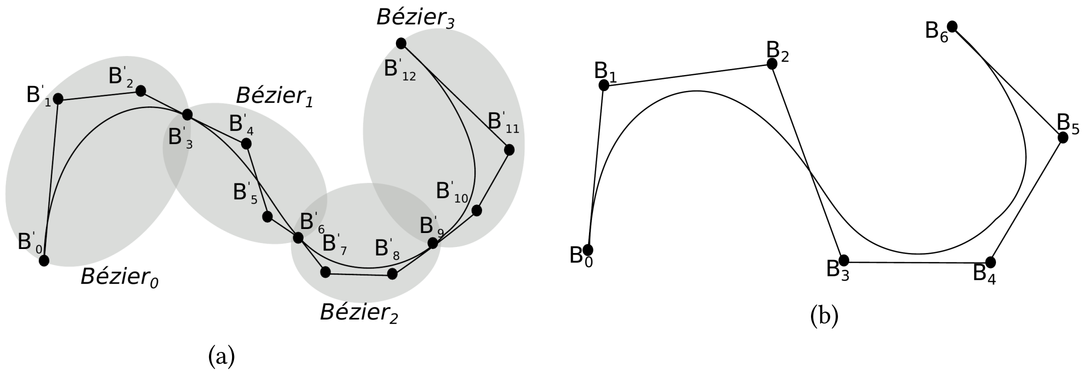


Figure 2.1: Cubic curve represented as (a) a group of *Bézier* or (b) a NURBS curve [1, 2]

The explanation of how NURBS are able to attain this result is divided into two subsections, which explain the properties at representing the curves at subsection 2.1.1 and surfaces at subsection 2.1.2.

2.1.1 NURBS curves

First of all the mathematical definition of a NURBS curve is the following expression:

$$C(u) = \frac{\sum_{i=0}^{n} N_{i,p}(u)\ w_i B_i}{\sum_{i=0}^{n} N_{i,p}(u)\ w_i},\ a \leq u \leq b \tag{2.1}$$

where $n+1$ is the number of control points, B_i are the control points, w_i are the weights and $N_{i,p}$ are the *pth-degree* B-spline basis function defined on the knot vector.

A knot vector is a sequence of real number coordinates representing the parametric domain of a knot span and stored in an increasing order.

The mathematical expression of it is:

$$U = \{\underbrace{0, ..., 0}_{p+1}, x_{p+i}, ..., x_{m-p-1}, \underbrace{1, ..., 1}_{p+1}\} \tag{2.2}$$

where $m = n + p + 1$, $w_i > 0$ for all i and a and b are usually defined as 0 and 1 respectively, but depending on the desired properties this values might vary.

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

$$N_{i,p}(u) = \frac{u - x_i}{x_{i+p} - x_i} N_{i,p-1}(u) + \frac{x_{i+p+1} - u}{x_{i+p+1} - x_{i+1}} N_{i+1,p-1}(u) \tag{2.3}$$

$$N_{i,0}(u) = \begin{cases} 1 & \textbf{if } x_i \leq u \leq x_{i+1} \\ 0 & \textbf{otherwise} \end{cases} \tag{2.4}$$

NURBS curves have many properties advantageous for a high quality render, being the ones of more interest for this project:

1. Strong convex hull property: if $u \in [x_i, x_{i+1})$, then $C(u)$ lies within the convex hull of the control points B_{i-p}, ..., B_i.
2. Local approximation: if the control point B_i is moved, or the weight w_i is changed, it is constrained to the specific region defined by the interval $[x_i, x_{i+p+1})$. This property provide the NURBS curve with a great flexibility, due to its easy modifiable and localized shape with a small change on a control point or the weight of it.

It is important to note that the previously described properties are also inherited by the NURBS surfaces.

2.1.2 NURBS surfaces

A NURBS surface (see Figure 2.2) is obtained by a tensor product of two NURBS curves, and it is defined by its degrees, a set of weighted control points, and two knot vectors. It also has two independent parameters (u, v) of degrees (p, q):

The formula that define a surface is:

$$S(u,v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,p}(u)\ N_{j,q}(v) w_{i,j} B_{i,j}}{\sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,p}(u)\ N_{j,q}(v) w_{i,j}},\ 0 \leq u, v\ \leq 1 \tag{2.5}$$

where B_i 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 axes, and $N_{i,p}$ and $N_{j,q}$ are the non-rational B-spline basis function defined on two knot vectors of $p + n + 1$ and $q + m + 1$.

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

$$N_{i,p}(u) = \frac{u\ -\ x_i}{x_{i+p}\ -\ x_i} N_{i,p} + \frac{x_{i+p+1}\ -\ u}{x_{i+p+1}\ -\ x_{i+1}} N_{i+1,p-1} \tag{2.6}$$

with

$$N_{i,0}(u) = \begin{cases} 1 & \textbf{if } x_i \le u \le x_{i+1} \\ 0 & \textbf{otherwise} \end{cases} \tag{2.7}$$

In the same way, than in Equation 2.2 for NURBS curves, knot vectors in NURBS surfaces are defined in the same way with the same characteristics:

$$U = \{\underbrace{0, ..., 0}_{p+1}, x_{p+1}, ..., x_{r-p-1}, \underbrace{1, ..., 1}_{p+1}\} \tag{2.8}$$

$$V = \{\underbrace{0, ..., 0}_{q+1}, y_{q+1}, ..., y_{s-q-1}, \underbrace{1, ..., 1}_{q+1}\} \tag{2.9}$$

where $r = n + p + 1$ and $s = m + q + 1$
Hence, the most important parameters to determine the shape of a surface are:

1. $B_{i,j}$: The control points
2. $w_{i,j}$: The weights of said points
3. U and V knot vectors, which determine the knot span of the NURBS
4. p and q degrees of the surface

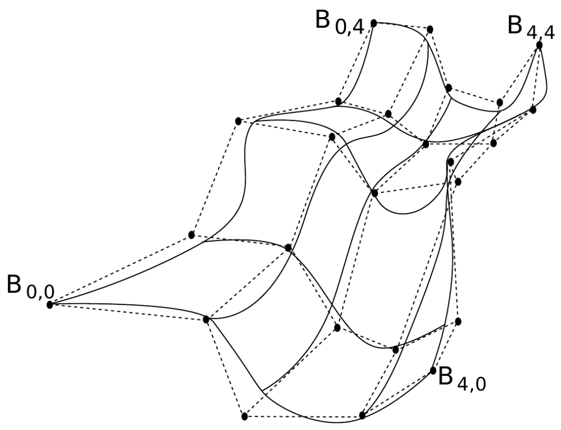


Figure 2.2: Bi-quadratic NURBS surface example [1]

In Figure 2.2 it is possible to see a NURBS surface example with the following control points $B = \{B_{0,0}, ..., B_{0,4}, ..., B_{4,0}, ..., B_{4,4}\}$, $U = V = \{0, 0, 0, \frac{1}{3}, \frac{2}{3}, 1, 1, 1\}$ and weights equal to 1.

2.2 GPU Tessellation

This section is dedicated to the explanation of tessellation and its execution via the different DirectX APIs (*Application Programming Interface*) versions, in which DirectX11 was the basis for the RPNS code developed by the Computer Architecture Group and used as the start point for this project.

Tessellation is defined as the process of completely filling with the same type of polygon a surface [10], usually triangles. As the GPUs lack the ability to directly render NURBS surfaces, this is a necessary step in order to generate them and according to the way this process is performed the resulted quality may differ.

2.2.1 DirectX11 and the traditional pipeline

Although DirectX12 was released on 2015, DirectX11 is still widely used. On DirectX11 many important features where introduced, such as: the update to Shader Model to version five, an improved texture compression, or the inclusion of multithreading support. Nevertheless the most relevant change for this project is the inclusion of a configurable tessellation phase in its pipeline, which is performed by the new programmable shaders, Hull Shader and Domain Shader, which use the HLSL (High Level Shader Language) programming language.

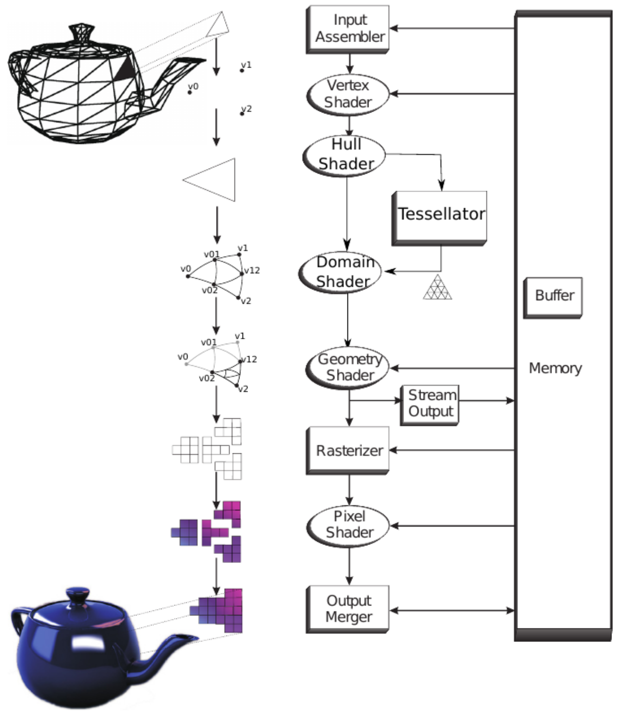


Figure 2.3: DirectX 11 pipeline stages [1]

Continuing with DirectX11 Pipeline which can be seen in Figure 2.3, a brief explanation is included bellow [10, 11]:

At the start of the pipeline, points is introduced to the *Input Assembler* to be arranged into primitives data types, as for example line or triangles [12]. Said primitives serve as the input for the *Vertex Shader*, which applies a user specified function to each vertex of the primitives regardless of their type. From this point on the tessellator process, which will be explained in detail bellow, is executed to fill the surfaces adding more detail to the model and sending the result to the *Geometry Shader*, which performs the last modifications to the model topology knowing, in contrast with the *Vertex Shader*, the primitives data type and the adjacent geometry to said primitives at execution.

At the end of the pipeline, the *Geometry Shader* output data is sent to the *Rasterizer* to be translated into pixels on the screen, performing a transformation from a tridimensional representation to a bidimensional one and sending the result to the *Pixel Shader*. At this stage colour is added to each pixel and its lightning is calculated to then be applied by the *Output Merger*, which also uses said data to determine the pixel visibility on the screen.

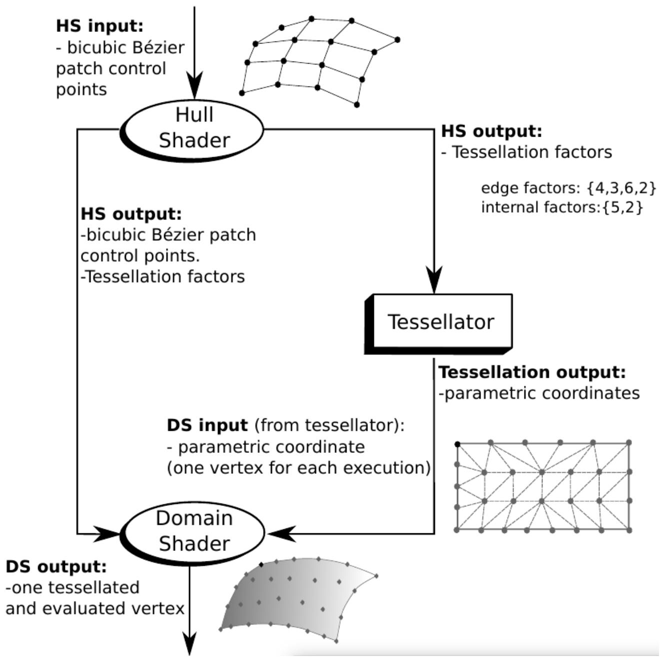


Figure 2.4: Tessellation structure on DirectX11 pipeline [1]

Going into detail in how tessellation is performed, these phases are composed of two shaders: the *Hull Shader* and the *Domain Shader* with the *Tessellator* operating between them. At the start of this process [10, 11], the vertices calculated in the *Vertex Shader* are received with the primitive data types determined at the *Input Assembler*. With this information proceeds to calculate the surface or patch control points and its tessellation factors, which serve as the input for the *Tessellator* to know the detail level to apply. From this point, the *Tessellator* starts to fill the surfaces in accordance with the data provided by the *Hull Shader*, by executing a user configured tessellation function, which returns the parametric coordinates. This coordinates in junction with the surface control points and the tessellation factors are used at the *Domain Shader* to compute the total number of vertices, definitely including the tessellation into the surface. Figure 2.4 exposes the explained tessellation process in a graphical manner.

It is important to acknowledge that in spite of being an optative phase, it may help to increase the model quality as it is exposed on [10].

2.2.2 DirectX12 and Meshing Shading

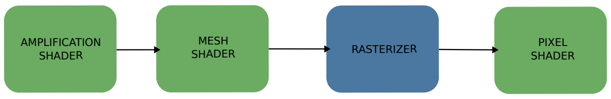


Figure 2.5: DirectX12 pipeline using *Meshing Shading*

DirectX12 is the latest version of the graphical API, even though some notable benefits were introduced, such as a reduction on CPU overhead and parallel GPU execution [13] among others, the most notable change for this project is the addition along side with the legacy pipeline (see Figure 2.3) of the DirectX12 *Ultimate* pipeline (see Figure 2.5) introducing with it the concept of the *Meshing Shading* and simplifying the pipeline.

The change on the pipeline brings with it the concept of *Meshlet*, being each one a segment of the geometry of the whole model [14]. As consequence of this partitioning at the *Mesh Shader* there is the possibility to easily modify the level of detail per model region and also performing a concurrent rendering process of the primitives and the triangle mesh (tessellation) [15].

In addition this pipeline also includes the *Amplification Shader*, also known as the *Task Shader*, which offers the possibility to easily perform culling over the *Meshlets* in a multithreading manner, thanks to its ability to determine the number of workgroups prior to the execution of the *Mesh Shader* [5, 14].

2.3 Related Work

In order to contextualize the evolution of NURBS rendering, this section provides a brief explanation of the different techniques used to archive this objective.

The first way is the NURBS subdivision [16] into different representations, usually *Bézier* patches as they are a specific case of NURBS [1]. If the results obtained from changing the representation end up in errors higher than the established as acceptable, it is possible to apply a recursive subdivision over a specific region to progressively reduce the error. Nevertheless [17] proposal exposes a way to avoid recursively subdividing patches by applying subpixel variant tessellation, increasing and decreasing the detail dynamically.

The second approach and the one used at this proposal is the direct evaluation of NURBS. [18] method takes advantage of the subdivisions applied by [16] and performs the evaluation by using different meshes with the control point storing this information as textures. The approach used at this project is based on the KSQuad primitive [2, 4], avoiding the previous subdivisions and storing of textures by using the Strong Convex Hull and Local support properties.

Chapter 3

Methodology and Planification

In this chapter the planification and methodology followed to develop this project is presented and tasks in each phase are described in detail.

3.1 Methodology

This project uses a scrum methodology [19], in which a set of product's requirement are determined by the *Product Owner*. This objectives are divided into smaller goals and the set of tasks needed for its accomplishment are called increments.

This methodology stipulates recurrent meeting (*Daily Scrum*) to study the increment evolution, identify obstacles or even reschedule tasks in order to achieve the incremental objectives. At the end of each increment, a meeting is arranged to review the obtained results and to analyse future requirements, called *Spring Review*.

3.2 Planification

This project follows a Scrum methodology, but some changes were introduced to adequate it to the directors necessities. The modifications were applied to the *Daily Scrum* to arrange the meetings approximately each two weeks in a mainly online format.

The workflow of the project started with the studio of the NURBS surfaces and the KSQuad primitive, followed by the analysis of the code developed at the project [3] and the RPNS [4]. The second step involved the analysis, design and implementation of NURBS surfaces rendering generalization and the incrementation of their tessellation level via the *Mesh Shader*. The next increments were oriented to develop KSQuad culling and surface culling approaches using the *Amplification Shader* and on the last one, performance tests were conducted on different architectures. In the next subsections, each increment is described in more detail.

3.2.1 Increment 1

The first increment starts on 14th of February 2022 and finish on 3rd of March 2022 with a duration of 14 days. The main objective was the study of the theoretical concepts needed to begin the project. As such, the first task was an introduction to NURBS, followed by the study of culling at the RPNS code and ending up with the DirectX API study. This is a necessary step due to the lack of knowledge at crucial aspects of the project when it started.

3.2.2 Increment 2

Once the theoretical bases of the project were acquired, the second increment started on 4th of March 2022 and ended on 28th of March 2022 during 17 days. The development of this increment was divided into the deployment of the *Mesh Shader* code from [3] and on the RPNS [4] code deployment. The increment was finished with the study of both implementations, in order to determine the needed modifications for this project.

3.2.3 Increment 3

This increment started on 29th of March 2022 and ended on 6th of July 2022 with an extent of 72 days. Over this period, the rendering of the NURBS surfaces was generalized to perform tests on different models and the level of tessellation was reduced to a smaller scale, via the *Mesh Shader*. These goals were achieved by dividing this increment into four tasks dedicated to render: an individual KSQuad, a complete surface, a specific model (see Figure 5.1) and all the test models.

3.2.4 Increment 4

This increment is related to the study and addition of the *Amplification Shader* to the pipeline in order to start culling, from 7th of July 2022 to 21th of September 2022 and taking 55 days. The first task was related to the identification of the properties of the *Amplification Shader* and the second one was the addition of the *Amplification Shader*.

3.2.5 Increment 5

From 22th of September 2022 to 9th of November 2022 during 35 days, the *Amplification Shader* is programmed to perform our first proposition, culling at KSQuad level. This increment is divided in two tasks: the study of its characteristics and the implementation of the culling technique.

3.2.6 Increment 6

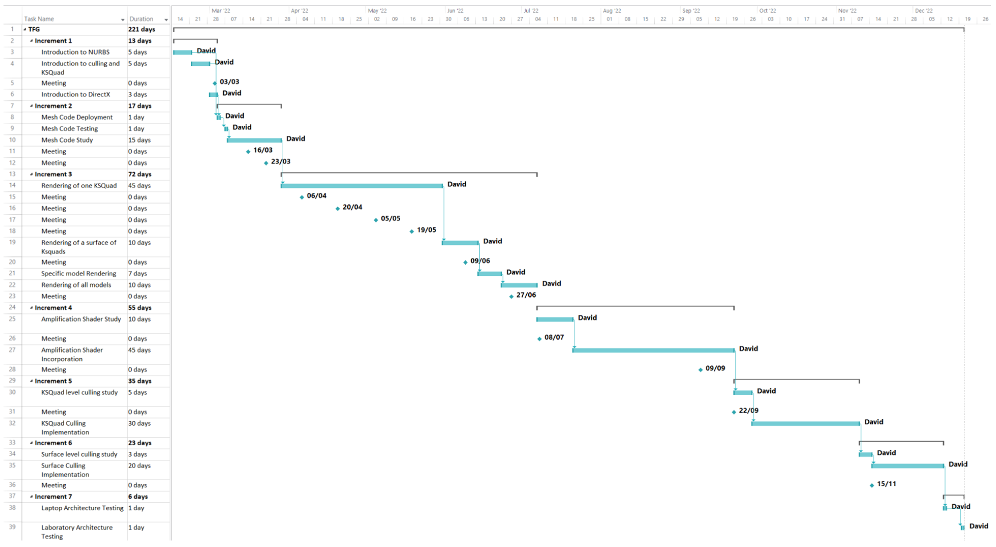
The increment number six was dedicated to add culling at surface level, taking place from 10th of November 2022 to 12th of December 2022 with an extent of 23 days. Although this and the previous increment are interchangeable, this order was applied so the knowledge acquired at the previous increment would reduce the time required at this stage. In the same way, the tasks are the same as those used in the previous increment.

3.2.7 Increment 7

The performance tests were performed on a laptop and on a high-end laboratory system provided by GAC (Computer Architecture Group) at the Universidade da Coruña. Two tasks compose this increment: the performance test in the personal laptop and then at the high end GAC System, starting on 13th of December 2022 and ending on 20th of December 2022, taking 6 days.

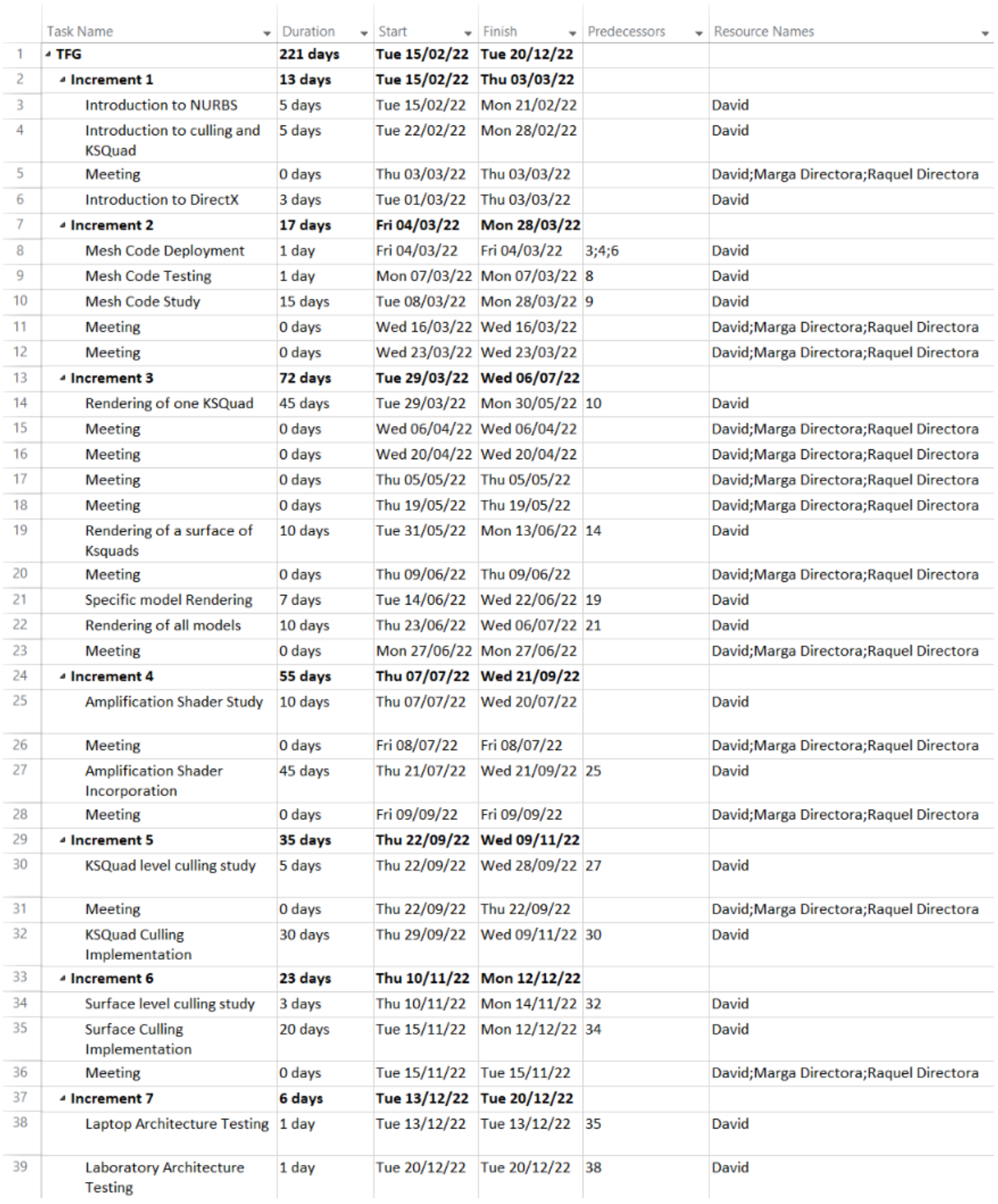
3.2.8 Gantt Diagram

The Microsoft Project tool was used to determine the planification as the accounts at the Universidade da Coruña are granted access. It also provides features to ease the task planning and visually represent their evolution. Figure 3.1 is the Gantt diagram with the time required by each task and the relations between them and Figure 3.2 corresponds to the task details. From left to right at Figure 3.2: the task name, the duration of each task, start and end date, dependencies between them and resources assigned are displayed with summaries to determine the corresponding increment.



	Task Name	Duration
1	**TFG**	**221 days**
2	**Increment 1**	**13 days**
3	Introduction to NURBS	5 days
4	Introduction to culling and KSQuad	5 days
5	Meeting	0 days
6	Introduction to DirectX	3 days
7	**Increment 2**	**17 days**
8	Mesh Code Deployment	1 day
9	Mesh Code Testing	1 day
10	Mesh Code Study	15 days
11	Meeting	0 days
12	Meeting	0 days
13	**Increment 3**	**72 days**
14	Rendering of one KSQuad	45 days
15	Meeting	0 days
16	Meeting	0 days
17	Meeting	0 days
18	Meeting	0 days
19	Rendering of a surface of Ksquads	10 days
20	Meeting	0 days
21	Specific model Rendering	7 days
22	Rendering of all models	10 days
23	Meeting	0 days
24	**Increment 4**	**55 days**
25	Amplification Shader Study	10 days
26	Meeting	0 days
27	Amplification Shader Incorporation	45 days
28	Meeting	0 days
29	**Increment 5**	**35 days**
30	KSQuad level culling study	5 days
31	Meeting	0 days
32	KSQuad Culling Implementation	30 days
33	**Increment 6**	**23 days**
34	Surface level culling study	3 days
35	Surface Culling Implementation	20 days
36	Meeting	0 days
37	**Increment 7**	**6 days**
38	Laptop Architecture Testing	1 day
39	Laboratory Architecture Testing	1 day

Figure 3.1: Gantt diagram



	Task Name	Duration	Start	Finish	Predecessors	Resource Names
1	**TFG**	**221 days**	**Tue 15/02/22**	**Tue 20/12/22**		
2	**Increment 1**	**13 days**	**Tue 15/02/22**	**Thu 03/03/22**		
3	Introduction to NURBS	5 days	Tue 15/02/22	Mon 21/02/22		David
4	Introduction to culling and KSQuad	5 days	Tue 22/02/22	Mon 28/02/22		David
5	Meeting	0 days	Thu 03/03/22	Thu 03/03/22		David;Marga Directora;Raquel Directora
6	Introduction to DirectX	3 days	Tue 01/03/22	Thu 03/03/22		David
7	**Increment 2**	**17 days**	**Fri 04/03/22**	**Mon 28/03/22**		
8	Mesh Code Deployment	1 day	Fri 04/03/22	Fri 04/03/22	3;4;6	David
9	Mesh Code Testing	1 day	Mon 07/03/22	Mon 07/03/22	8	David
10	Mesh Code Study	15 days	Tue 08/03/22	Mon 28/03/22	9	David
11	Meeting	0 days	Wed 16/03/22	Wed 16/03/22		David;Marga Directora;Raquel Directora
12	Meeting	0 days	Wed 23/03/22	Wed 23/03/22		David;Marga Directora;Raquel Directora
13	**Increment 3**	**72 days**	**Tue 29/03/22**	**Wed 06/07/22**		
14	Rendering of one KSQuad	45 days	Tue 29/03/22	Mon 30/05/22	10	David
15	Meeting	0 days	Wed 06/04/22	Wed 06/04/22		David;Marga Directora;Raquel Directora
16	Meeting	0 days	Wed 20/04/22	Wed 20/04/22		David;Marga Directora;Raquel Directora
17	Meeting	0 days	Thu 05/05/22	Thu 05/05/22		David;Marga Directora;Raquel Directora
18	Meeting	0 days	Thu 19/05/22	Thu 19/05/22		David;Marga Directora;Raquel Directora
19	Rendering of a surface of Ksquads	10 days	Tue 31/05/22	Mon 13/06/22	14	David
20	Meeting	0 days	Thu 09/06/22	Thu 09/06/22		David;Marga Directora;Raquel Directora
21	Specific model Rendering	7 days	Tue 14/06/22	Wed 22/06/22	19	David
22	Rendering of all models	10 days	Thu 23/06/22	Wed 06/07/22	21	David
23	Meeting	0 days	Mon 27/06/22	Mon 27/06/22		David;Marga Directora;Raquel Directora
24	**Increment 4**	**55 days**	**Thu 07/07/22**	**Wed 21/09/22**		
25	Amplification Shader Study	10 days	Thu 07/07/22	Wed 20/07/22		David
26	Meeting	0 days	Fri 08/07/22	Fri 08/07/22		David;Marga Directora;Raquel Directora
27	Amplification Shader Incorporation	45 days	Thu 21/07/22	Wed 21/09/22	25	David
28	Meeting	0 days	Fri 09/09/22	Fri 09/09/22		David;Marga Directora;Raquel Directora
29	**Increment 5**	**35 days**	**Thu 22/09/22**	**Wed 09/11/22**		
30	KSQuad level culling study	5 days	Thu 22/09/22	Wed 28/09/22	27	David
31	Meeting	0 days	Thu 22/09/22	Thu 22/09/22		David;Marga Directora;Raquel Directora
32	KSQuad Culling Implementation	30 days	Thu 29/09/22	Wed 09/11/22	30	David
33	**Increment 6**	**23 days**	**Thu 10/11/22**	**Mon 12/12/22**		
34	Surface level culling study	3 days	Thu 10/11/22	Mon 14/11/22	32	David
35	Surface Culling Implementation	20 days	Tue 15/11/22	Mon 12/12/22	34	David
36	Meeting	0 days	Tue 15/11/22	Tue 15/11/22		David;Marga Directora;Raquel Directora
37	**Increment 7**	**6 days**	**Tue 13/12/22**	**Tue 20/12/22**		
38	Laptop Architecture Testing	1 day	Tue 13/12/22	Tue 13/12/22	35	David
39	Laboratory Architecture Testing	1 day	Tue 20/12/22	Tue 20/12/22	38	David

Figure 3.2: Detail of each task in the Gantt digram

3.3 Project Costs

This section presents the estimation of the project cost derived from the used resources and time dedicated to it. These can be classified into two categories: human costs and indirect equipment costs. To provide a cost estimation it would be assumed that the project was done by a junior programmer and two directors, being the time and salaries of each one detailed at the table 3.1, representing the first one in hours and the second one in euros (€) per hour.

	Junior Programmer	Director A	Director B	Total
Salary (€/hour)	10	40	40	-
Time (hours)	484	12	12	508
Labour cost (€)	4840	480	480	5800

Table 3.1: Human costs

Considering the indirect costs, these would be mainly linked to the price of the used computers, a personal laptop dedicated to the code development and a desktop computer from GAC-UDC to perform the tests. If an average life span of five years is assumed for both architectures, the cost of using them is the purchased price divided by their life span and multiplied by the used time, as in Equation 3.1 can be appreciated. These three metrics are exposed with price and cost expressed in euros (€) and time in months (see table 3.2).

$$Used\,Time * \frac{Price}{Life\,Span} \tag{3.1}$$

System	Personal Laptop	GAC System	Total
Price (€)	1150	4500	5650
Used time (months)	7	1	8
Life Span (months)	60	60	-
Cost of use (€)	134.17	75	209.17

Table 3.2: Cost and time details of the resources used

As a result of the previous calculations and assumptions, a project of this characteristics would have resulted in 6009.17 euros (see table 3.3).

Expense Name	Cost (€)
Labour Costs	5800
Indirect costs	209.17
Total	6009.17

Table 3.3: Total costs of the project

Chapter 4

Culling Techniques for NURBS surfaces using Meshing Shading

This chapter is the explanation of the different culling techniques developed with DirectX12 *Ultimate* pipeline using *Meshing Shading*. A brief explanation of the KSQuad primitive [4] is also presented, as it is highly related to our proposal.

One of the objectives of the project was changing the way models were rendered. The previous approach [3] tessellated at surface level and our proposal is at KSQuad level. This transition would allow a higher level of tessellation and parallelism, as the used scale is smaller and it is also rendered in a multithreading way. Moreover, this modifications are also oriented to generalize the rendering process of models with different properties. In particular, those with different number of knots on their surfaces.

4.1 Rendering of NURBS surfaces using KSQuad

KSQuad (*Knot Span Quad*) is a primitive specifically designed to ease the NURBS surface rendering in the GPU [1]. As the data describes a surface's section, the application of localized modifications over the NURBS surface becomes easier. A small change on the control points or weights stored in the KSQuad is enough.

A KSQuad$_{i,j}$ of a surface of p and q degrees is:

$$KSQuad_{i,j} = \{\overbrace{x_i, x_{i+1}, y_j, y_{j+1}}^{Knot\ Span}, \overbrace{B_{i-p,j-q}, ..., Bi, j}^{Control\ Points}, \overbrace{w_{i-p,j-q}, ..., w_{i,j}}^{Weights}\} \tag{4.1}$$

where $x_i \neq x_{i+1}$ and $y_j \neq y_{j+1}$. This primitive maintains the Convex Hull and Local Support properties. The first property dictates that a NURBS surface is constrained to the convex hull defined by its control points and the second one limits de influence of a specific control point [2].

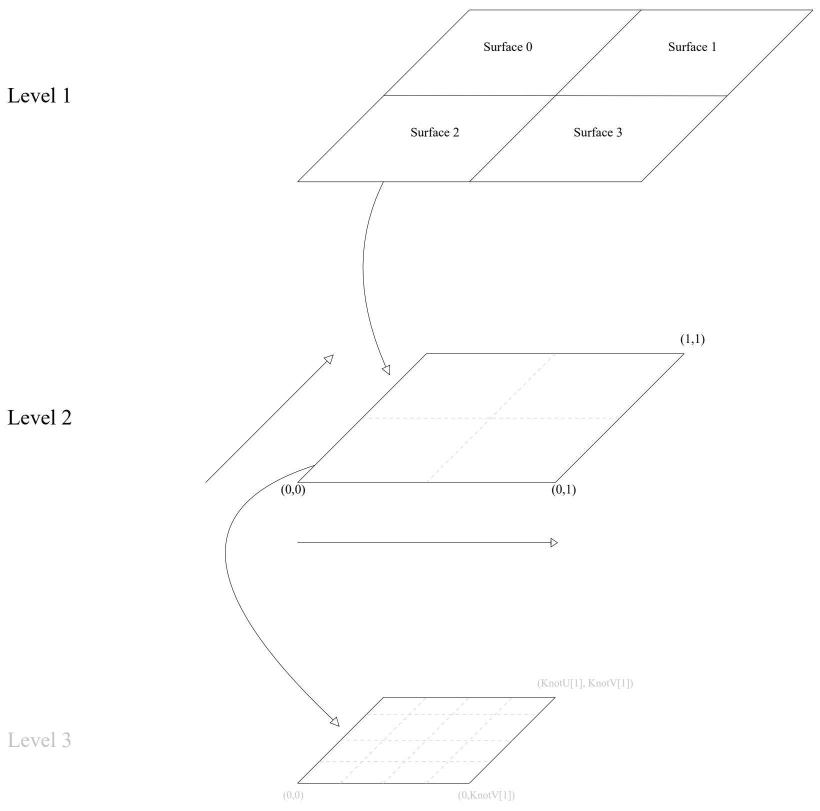


Figure 4.1: Levels of tessellation

In contrast to the previous proposal [3], our proposal presents three levels of tessellation in order to increase the quality of the image with respect to previous proposal. Previous proposal only presents two levels of tessellation.

The first level corresponds to the set of NURBS surfaces that define the model, while the second level is dedicated to the different KSQuads. In each one of those KSQuads tessellation is applied into a 9×9 mesh of triangles. Accomplishing a higher level of tessellation compared with the previous proposal [3]. Figure 4.1 shows a representation of the three levels of the proposal.

In Section 2.2.2 *Meshlets* were introduced as a segmentation of the model, which represent a set of primitives. In our proposal, each *Meshlet* represents specifically one KSQuad primitive. These are rendered in a parallel manner by the *Mesh Shader.* This process is performed by assigning to each thread group (SV_Group) a *Meshlet.* From this point, the KSQuad is tessellated into 9×9 points by 128 threads. The result of this process is the triangle mesh of the tessellated KSQuad.

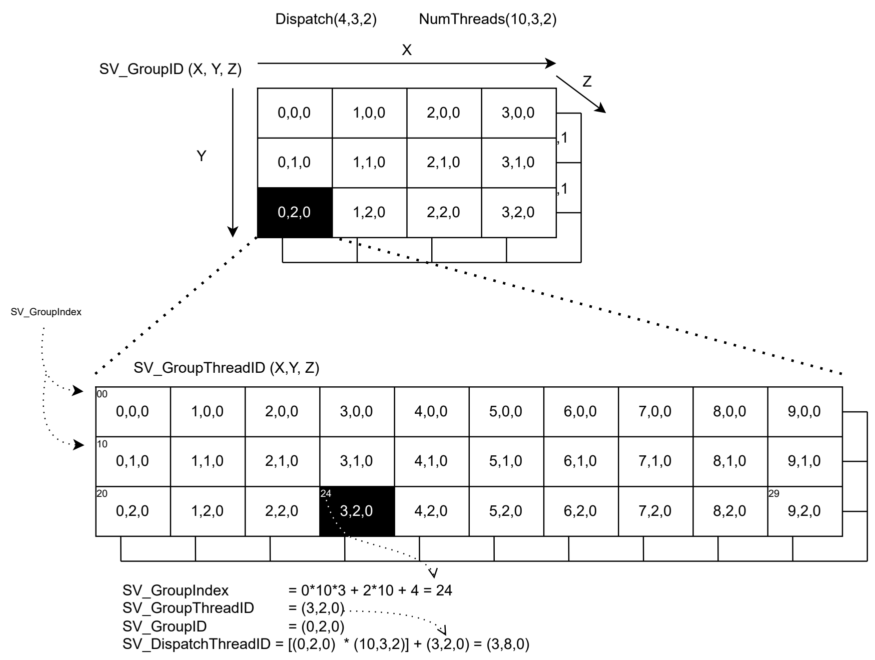


Figure 4.2: Threading group schema example of any DirectX shader with $4 \times 3 \times 2$ *SV_Groups* and a number of $10 \times 3 \times 2$ threads.

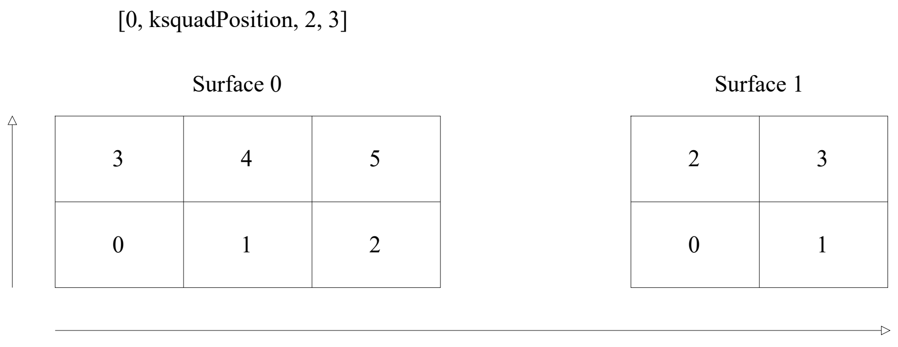


Figure 4.3: KSQuad position of surface depending of the number of knots at U and V

Figure 4.2 exposes an example of 24 ($4 \times 3 \times 2$) thread groups (*SV_Group*), defined by the *Dispatch* function. Each one of them has 60 ($10 \times 3 \times 2$) threads, concurrently operating per thread group. The built in variables of DirectX that permit the management of the threads are:

- *SV_GroupID* identifies a *SV_Group* in a shader.
- *SV_GroupThreadID* identifies a thread in a *SV_Group* using a 3-component vector.

- *SV_GroupIndex* provides the identifier of a thread using a 1-component vector.
- *SV_DispatchThreadID* identifies a thread in relation to all the *SV_Groups* in a shader.

```
// Access to surfaceId
int surfaceId = ks[4 * SV_GroupID];

// Access to the KSQuad position
int ksquadPosition = ks[4 * SV_GroupID + 1];

// Number of knots in the surface
int knotIntervalU = ks[4 * SV_GroupID + 2];
int knotIntervalV = ks[4 * SV_GroupID + 3];
```

Listing 4.1: Access to KS data structure

On the other hand, in order to include the KSQuad primitive, a new structure called KS is defined. The purpose of this structure is to provide to each thread the required data to identify the parametric position of a KSQuad. It stores: the id of the surface, the number of knots at U and V directions and the corresponding position of each KSQuad in a surface. Figure 4.3 displays an example with two surfaces, one with two knots at U and three knots at V and other with two knots at both parametric directions.

The first step performed by the *Mesh Shader* is the utilization of the *SV_GroupID* to load the KS. The relation between this variables is established by the fact that each *SV_Group* is responsible of one *Meshlet* and each *Meshlet* represent a specific KSQuad primitive. Algorithm 4.1 exposes the way this data is accessed by the *Mesh Shader*.

The second step determines the interval of parametric space to tessellate. This process is displayed by Algorithm 4.2. The variables *knotsU* and *knotsV* contain all the knots of all the KSQuads in the whole model, at U and V directions. In order to locate the values of the tessellated KSQuad, the variable *tablaKnots* returns from each surface id the index of the first surface's KSQuad knots.

In lines 2 and 3 of Algorithm 4.2, the KS data is used to locate the KSQuad position on the surface. In the following lines, this value is used to return the interval of parametric space, by changing the index of *knotsU* and *knotsV* variables.

The third step calculates the parametric coordinates of the tessellation points inside each KSQuad. Algorithm 4.3 starts by assigning to each thread a point depending on their id, in lines 2 and 3. In the remaining lines, the distance of each point to the knots 0 at U and V directions in the KSQuad, is used to calculate the parametric coordinates of the tessellation point.

Each thread generates the cartesian coordinates of the point by using the function *NurbsEval2*, where it is evaluated according to Equation 2.5.

```
// ksquad intervals at U and V axis per specific surface
int ksquadCoordU = floor(ksquadPosition / knotIntervalV);
int ksquadCoordV = ksquadPosition % knotIntervalV;

// Final knots position of tessellated KSQuad
int knotIndexU = tablaKnots[surfaceId].x + ksquadCoordU + 1;
int knotIndexV = tablaKnots[surfaceId].y + ksquadCoordV + 1;

// Final points of knot interval
float knotFinalPointU = knotsU[knotIndexU];
float knotFinalPointV = knotsV[knotIndexV];

// Start knots position of tessellated KSQuad
int knotIndexU = tablaKnots[surfaceId].x + ksquadCoordU;
int knotIndexV = tablaKnots[surfaceId].y + ksquadCoordV;

// Start points of knot interval
float knotInitialPointU = knotsU[knotIndexU];
float knotInitialPointV = knotsV[knotIndexV];
```

Listing 4.2: Parametric border coordinates of the KSQuad

```
// Triangle's point assignation per thread
int trianglePointU = floor(SV_GroupThreadID / 9);
int trianglePointV = SV_GroupThreadID % 9;

// Distance between triangle's points inside KSQuad interval
float tesLengthU = (knotFinalPointU - knotInitialPointU) / 8
float tesLengthV = (knotFinalPointV - knotInitialPointV) / 8

// Triangle position in relation to the KSQuad first knots
float stepU = trianglePointU * tesLengthU;
float stepV = trianglePointV * tesLengthV;
```

Listing 4.3: Assignation of a point of the mesh of triagnles per thread id (SV_GroupThreadID)

4.2 Culling Techniques on Rendering Pipeline for NURBS Surfaces

Culling is the process of removing those segments of a model that do not play a part in the final rendering of the scene. The purpose is the reduction of computational load in the rest of the stages of the pipeline. From the wide variety of ways to perform culling, our proposal is focused on *Back-face* culling techniques [20], which consists on removing portions of the model not facing the camera 2.4.

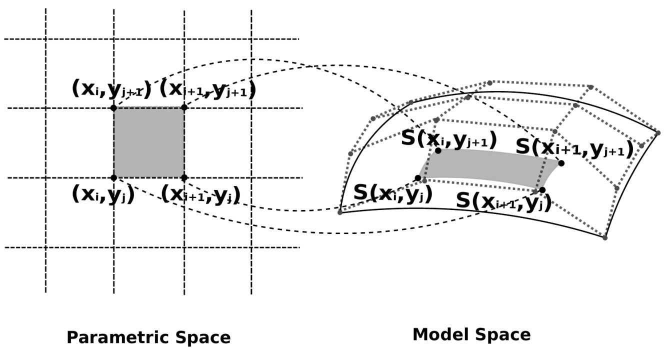


Figure 4.4: KSQuad primitive defined by a knot span [1]

In [4], different interactive culling strategies for NURBS surfaces are presented based on *Back-face*, the one of more interest for our proposal is LBC (*Light Backpatch Culling*).

In LBC, the normal vector is computed in relation to the plane defined by the skeleton of each KSQuad. Another characteristic of this approach is that it requires less computational power. Taking Figure 4.4 as reference, the operation used to cull a KSQuad$_{i,j}$ [4] is:

$$\square_{i,j} = \{S(x_i, y_j),\ S(x_{i+1}, y_j),\ S(x_i, y_{j+1}),\ S(x_{i+1}, y_{j+1})\} \tag{4.2}$$

4.3 Culling Techniques Implementation in DirectX12 using Meshing Shading

In this section, we present the culling techniques proposals based on LBC of RPNS [4]. These techniques use the dot product between the camera vector and the normal of a plane to determine the visibility. Figure 4.5 exposes two planes with their normal vectors inside the cone of vision of a camera. It is possible to appreciate how visibility is conditioned to the facing of each plane to the viewer. Surface a is visible as it faces the camera and b is culled as does not faces the viewer.

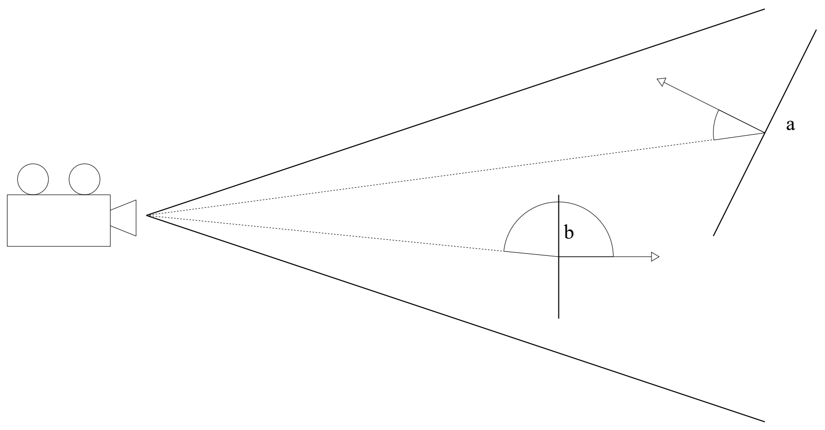


Figure 4.5: *Back-face* culling example with two surfaces, Surface a facing the camera and Surface b not facing the camera

Our proposal presents two culling techniques, the first technique is *KSQuad Level Culling*, which operates at KSQuad scale by computing a normal vector per each individual KSQuad. The second technique is *Surface Level Culling*, which calculates the normal vector in relation to a KSQuad skeleton defined by their NURBS surfaces.

4.3.1 KSQuad Level Culling

In this section, the implementation details of the *KSQuad Level Culling* are exposed.

Culling requires the *Amplification Shader*, which must not change the received input of the *Mesh Shader*. This situation would involve two ways of rendering and it would make harder the technique evaluation. The second requirement is that the *Amplification Shader* must take advantage of its parallel capabilities to perform an efficient culling.

The desired situation is one in which each thread concurrently determines the visibility of one KSQuad in the *Amplification* stage and the *Mesh Shader* tessellates with multiple threads on a specific KSQuad.

In order to achieve one thread per KSQuad, the number of thread groups (*SV_Group*) at the *Amplification Shader* is configured. The amount is the total number of KSQuads in the model, divided by the number of threads to use at the *Amplification* stage. Hence, each thread group is responsible of applying culling to 32 KSQuads.

With respect to the *Amplification Shader* implementation. It must be able to uniquely identify each KSQuad inside the group, in order to determine the normal vector of the KSQuad and transfer its information to the *Mesh Shader*, if the KSQuad is not culled. The *SV_DispatchThreadID* provides a unique identification of each thread in the *Amplification Shader* and by extension a unique identification of each KSQuad of the model. The justification comes from the total number of threads being equal to the number of KSQuads.

```
1  ...
2  int dtid: SV_DispatchThreadID
3  )
4  {
5    bool visible = false;
6
7    if (dtid < totalNumberOfKSQuads) {
8      visible = isVisible(dtid, observerPosition);
9    }
10   ...
11 }
```

Listing 4.4: First instructions of the *Amplification Shader*

In Algorithm 4.4 the first steps of the *Amplification Shader* are displayed. All KSQuads are considered like no visible (line 5), unless proved the opposite at the function *isVisible* (line 8). This function receives as parameters the *SV_DispatchThreadID* of the thread assigned to a KSQuad and the position of the camera. So, the dot product can be calculated and applied to our variant of *Back-face* culling. The condition of line 7 is located before the invocation of the function, to avoid calculations over threads which might not have KSQuads assigned at the last thread group (*SV_Group*).

```
#define AS_GROUP_SIZE 32
struct Payload {
  uint MeshletIndices[AS_GROUP_SIZE]
};
```

Listing 4.5: *Payload* data structure used at our proposal

```
groupshared Payload s_Payload;
...
{
  // Id of visible elements are stored for renderization
  if (visible) {
    uint index = WavePrefixCountBits(visible);

    s_Payload.MeshletIndices[index] = dtid;
  }

  int visibleCount = WaveActiveCountBits(visible);
  DispatchMesh(visibleCount, 1, 1, s_Payload);
}
```

Listing 4.6: Preparation and transfer of visible KSQuads information to the *Mesh Shader*

Once the visibility of the concurrently computed KSQuads have been determined, preparations to communicate the KSQuads to the *Mesh Shader*. The data structure used to send this information to the next shader is the *Payload*, which can be adapted to the user necessities. Algorithm 4.5 exposes the *Payload* used at this project, being for our needs a 32 length array. This array stores the *SV_DispatchThreadID* of the threads with visible KSQuads in the thread group (*SV_GroupID*).

The process to inform the *Mesh Shader* of the KSQuads to tessellate involves two steps: the *SV_DispatchThreadIDs* storage into the *Payload* array and the calculation of the total number of KSQuads to tessellate of each thread group. Algorithm 4.6 shows these steps. The *WavePrefixCountBits* function (line 6) provides a unique index number to store the *SV_DispatchThreadID* of each task with visible KSQuads. Then, the total number of visible KSQuads (line 11) is returned and the *Dispatch* function executes the *Mesh Shader* (line 12). Its first parameter is the number of visible KSQuads in the thread group and its last parameter is the *Payload*. This structure has the *SV_DispatchThreadIDs* of each visible KSQuads' threads.

```
1   ...
2   int gtid : SV_GroupThreadID,
3   int gid : SV_GroupID,
4   in payload Payload payload,
5   ...
6   )
7   {
8     // Extraction of DispatchThreadID from Amplification shader
    stage
9     int meshletIndex = payload.MeshletIndices[gid];
10
11    // No renderization of a ksquad number higher than the total
    number of them
12    if (meshletIndex >= totalNumberOfKSQuads) {
13      return;
14    }
15    ...
16    verts[gtid] = GetVertexAttributes(meshletIndex, gtid);
17  }
```

Listing 4.7: Extraction of transfered information at *Mesh Shader* from the *Amplification Shader*

The operations performed at the *Amplification Shader* manage to provided a similar output to the one which would have been received, if the pipeline directly started with the *Mesh Shader*. In Algorithm 4.6, the *visibleCount* variable specifies a thread group (SV_Group) per visible KSQuad at the *Mesh Shader*. Nevertheless, to determine the specific KSQuad to render the data from the *Payload* must be accessed. Algorithm 4.7 shows the *Mesh Shader* access of the *SV_DispatchThreadID* used at the *Amplification Shader* (line 9). With this information the

Mesh Shader can uniquely identify the corresponding KSQuad and proceed to tessellate it. This process is performed in a parallel way by using the *SV_DispatchThreadID* as a parameter for function *GetVertexAttributes* (line 16).

In relation to the chosen number of threads, the number 32 was selected as a result of the recommendations in [21]. The *Payload* should reduce its size the maximum possible and at the same time, the number of invocations of the *Mesh Shader* should be the minimum possible. In our project, the first recommendation can be satisfied by reducing the number threads, however this would also increase the number of invocations of the *Mesh Shader*, as less threads would apply culling per group. As a result of this situation, 32 was selected as the middle ground between this two requirements.

4.3.2 Surface Level Culling

In this section, we present the second culling technique, which uses the normal defined by the computed skeleton of each surface, instead of calculating the normal of each KSQuad. Most of the implementation followed by the *Amplification Shader* is maintained in relation with the Section 4.3.1.

```
// ks data structure
int surfaceId = ks[4 * SV_DispatchThreadID];
int knotIntervalU = ks[4 * SV_DispatchThreadID + 2];
int knotIntervalV = ks[4 * SV_DispatchThreadID + 3];

// Final knots position of surface
int knotFinalIndexU = tablaKnots[surfaceId].x + knotsIntervalU;
int knotFinalIndexV = tablaKnots[surfaceId].y + knotsIntervalV;

// Final point of the surface
float surfaceFinalPointU = knotsU[knotFinalIndexU];
float surfaceFinalPointV = knotsV[knotFinalIndexV];

// Start knots position of surface
int knotInitialIndexU = tablaKnots[surfaceId].x;
int knotInitialIndexV = tablaKnots[surfaceId].y;

// Start point of the surface
float surfaceInitialPointU = knotsU[knotInitialIndexU];
float surfaceInitialPointV = knotsV[knotInitialIndexV];
```

Listing 4.8: Calculation of surface's normal vector points

The *Amplification Shader* organizes the number of thread groups (*SV_Group*) as the total number of KSQuads in the model divided by 32. This number corresponds to the amount of threads of the thread group, making their *SV_DispatchThreadIDs* a useful variable to properly identify each KSQuad. With the KSQuad properly identified, the KS data structure is able to determine the KSQuad's surface and performing culling over it, if it is required.

The most notable change in relation with *KSQuad Culling* technique occurs in the points used to determine the visibility. All the KS is accessed, with the exception of the KSQuad position, as it is not required. By following a process similar to the one used for the tessellation points at the *Mesh Shader*, the parametric coordinates of the surface limits are calculated. Algorithm 4.8 displays this process. The surface limits are calculated by adding the total number of knots in the parametric directions (lines 7 and 8).

Once the points of the surface borders were determined, the dot product between the border points of the surface and the camera position is performed, to determine the KSQuad visibility. All the KSQuads of the same surface operate with the same normal vector and all the KSQuads must be visible or culled depending on the surface's points. If a KSQuad is visible, the *SV_DispatchThreadID* of its thread is stored and send to the *Mesh Shader* at the end of the *Amplification Shader*.

Chapter 5

Experimental Results

In this chapter, the results of our proposals for the different NVIDIA GPUs are presented with our conclusions.

5.1 Test Environment

In this section, the results of our proposal on two different architectures is analysed and exposed. The test platforms used in our experiment were described in table 5.1.

The second column presents the characteristics of a **personal laptop** (*PL*) using a GPU with Turing architecture, used to develop this project and perform the first tests on the models. The third column presents the characteristics of a **desktop computer** at the GAC UDC (*GS*) Laboratory with a Turing architecture GPU of high gamma.

	Personal Laptop (*PL*)	GAC System (*GS*)
CPU	AMD Ryzen 7-4800H 2.90GHz	Intel Core i7-4790H 3.6 GHz
RAM Memory	16GB DDR4 3200 MHz	32 GB DDR4 3200 MHz
OS	Microsoft Windows 10 Pro	Microsoft Windows 10 Enterprise LTSC
GPU	GeForce RTX 2060 Mobile	GeForce RTX 2080
Driver	527.56, SDK 10.0	255.255, SDK 10.0

Table 5.1: Description of the test platforms

The model used for the tests is displayed in Figure 5.1 and it is called *Head*[1]. It is composed of 601 surfaces (*#NS*) and 15025 KSQuads (*#KS*). Each surface has 3 degrees (p and q) and 4 knots (n and m), at both parametric directions (U and V).

[1] Model provided by Direct Dimensions Inc. https://www.dirdim.com/portfolio/

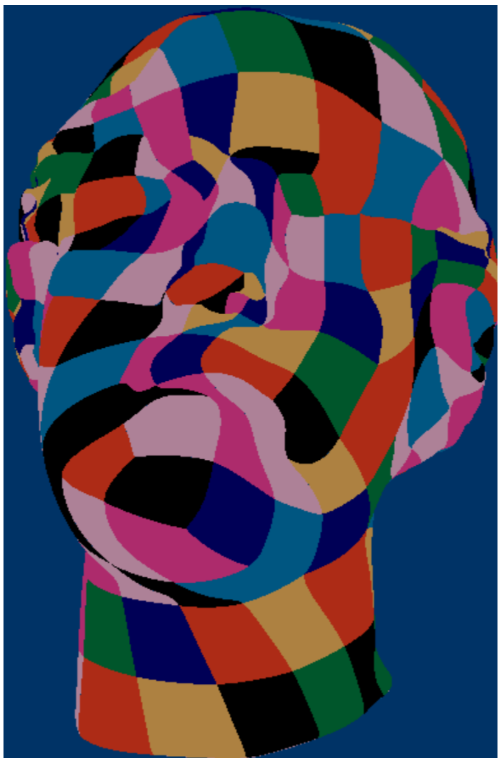


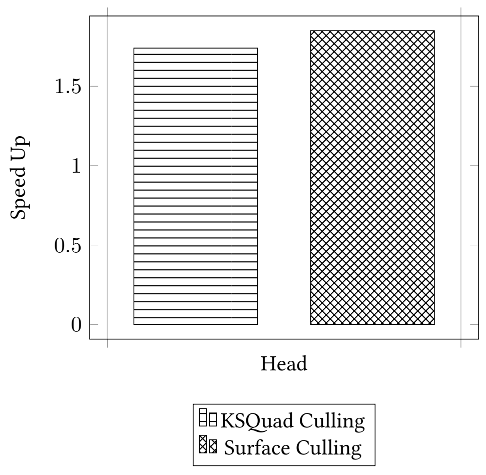
Figure 5.1: Test model (*Head*)

5.2 Performance results

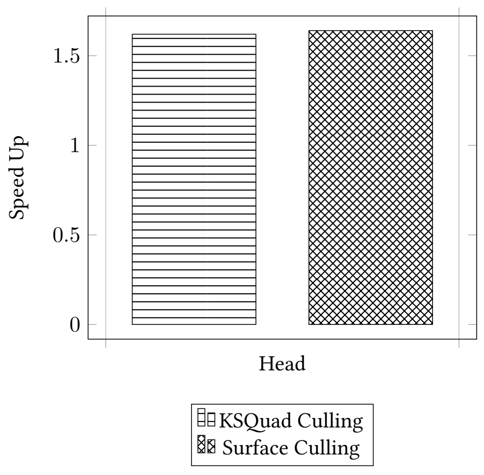
This section presents the results of our proposal, exposing at Table 5.2 the performance at FPS (Frames Per Seconds) and Figure 5.2 exposes the speed up of each system per test model and culling technique used.

	No culling (FPS)	KSQuad Level Culling (FPS)	Surface Level Culling (FPS)
GS	481	840	890
PL	660	1073	1082

Table 5.2: Performance comparison



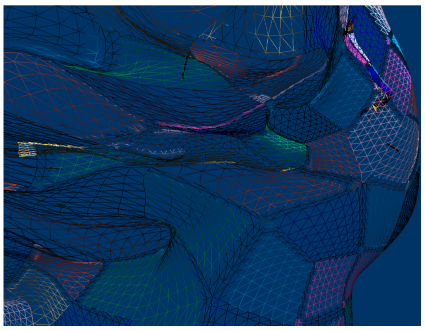
(a)



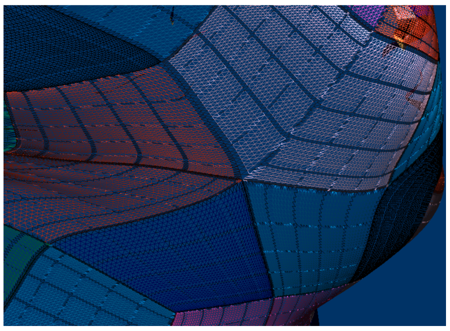
(b)

Figure 5.2: Speed up of culling techniques in relation to No Culling Approach (a) *GS* platform and (b) *PL* platform

Analysing the obtained results at table 5.2 the first impressions are good. All the test models, in all architectures have a bigger number of FPS in all cases than without of culling. In fact, KSQuad Culling provides a dramatic performance improvement, easily notable at the Figure 5.2, attaining speed-ups of more than 1.5x over all the test models regardless of the architecture. Moreover, the results of our proposals expose that interactive deformation is feasible with a KSQuad level of detail.



(a)



(b)

Figure 5.3: Comparison of the *Head* model (a) using [3] proposal and our (b) proposal

Continuing with the comparison between [3] proposal and ours. The result from [3] at the *GS* system attain 1846 FPS for the *Head* model. A FPS decrease is observed regardless of the culling technique used, occurring the higher reduction of 3.84x without culling. If the data from project [3] is compared with both culling techniques, the decrease is reduced to 2.19x for KSQuad Culling and 2.07x Surface Culling.

In relation to the models' quality, Figure 5.3 displays the levels of tessellation of the project [3] and our proposal. In Figure 5.3a the tessellation factor is smaller, while in Figure 5.3b the level of detail is higher. Although the FPS results are bellow of the project [3], our level of detail is higher. So, the FPS reduction can be justified by the increased quality of the model.

In relation to the *Surface Level Culling*, it is important to note the technique limitations. During its testing, it had been observed that some surfaces facing the camera were not rendered, which becomes worse as the surfaces size is increased. The observed situation can be explained by the difficulty of determining the optimal normal to the plane's skeleton.

In conclusion, respect project [3] a reduction in performance is experienced, however in our proposal the tessellation factor is higher and it is able to provide a better quality of rendering. If the proposed culling techniques are applied, the attained performance in relation to KSQuad rendering, experience speed ups of more than 1.5x in all models. Moreover, interactive deformation can be easily performed at a high level of detail. Nevertheless, *Surface Level Culling* is probably not the best option, despite attaining the best performance. It can cull surfaces facing the camera.

Chapter 6

Conclusions and Future Works

At the beginning, this project had three main objectives. The first one was using *Meshing Shading* to achieve tessellation at a KSQuad scale, the second one was an expansion of the proposal [4] for models with different m and n surface values and the last objective was the analysis and implementation of the different culling techniques. In order to achieve these objectives, the RPNS proposal [4], [3], DirectX12 API and *Meshing Shading* were studied.

This proposal has been able to expand the scope of the previous proposal [3], as tessellation at KSQuad level has been implemented and this process was even generalized to operate with different models. The image quality resulted of our proposal is also superior to previous [3], due to the higher level of tessellation.

Culling techniques were implemented and tested, taking as inspiration the *Light Quad Culling* (LQC) technique in [4]. The objective of these techniques was to increase the performance of model's deformation. The obtained results achieve speed ups beyond 1.5x for all the tested models. Nevertheless, the techniques culled KSQuads that should be visible, specially in *Surface Level Culling* technique.

In conclusion, the tessellation process was generalized for models with different n and m in their surfaces and their level of detail increased. Tessellation at such level also comes at the expense of a reduction of performance compared with [3]. Nevertheless, this situation can be improved with the proposed culling techniques.

6.1 Future works

In this Section, the possible continuations derived from this project are detailed.

The first continuation would be an improvement on the rendering process. The objective was the analysis of the culling techniques, but there are still possible optimizations. For example, the *Mesh Shader* can be optimized by using a more efficient structure to implement the KSQuad primitive.

Another area of improvement, it is on the implemented culling techniques. Both techniques are based on *Light Quad Culling* (LQC) from [4], but techniques based on *Strong Quad Culling* (SQC) had not been implemented with *Meshing Shading* pipeline. SQC technique provides a better quality of image in proposal [4], by calculating the normal vector in relation to the convex hull polygon, defined by the adjacent control points.

Appendices

Appendix A

Code Access

In this appendix, the instructions required to obtain the executed code are detailed along side a brief explanation of the main files.

First of all, the code is saved in a public Github repository https://github.com/DavLN/TFGCulling. The repository has three branches, one per code implementation previously treated:

- master: Implementation without culling and with the *Mesh Shader* modified to render at KSQuad level
- feature/surfaceCulling: Implementation with *Surface Level Culling*
- feature/ksquadCulling: Implementation with *KSQuad Level Culling*

The main files of this code are:

- MeshletCull/MeshletMS.hlsl: Mesh shader, where tessellation takes place
- MeshletCull/MeshletAS.hlsl: Amplification shader, where culling is performed
- MeshletCull/Model.cpp: Model data extraction and preparation for the GPU
- MeshletCull/D3D12MeshletCull.cpp: Base program and middleman between CPU and GPU communications. It contains information about the camera and location of the shader files.

Bibliography

[1] R. Concheiro, *Real Time Rendering of Parametric Surfaces on the GPU.* Ph D. da Universidade Da Coruña, 2013.

[2] R. Concheiro, M. Amor, E. J. Padron, and M. Doggett, "Interactive rendering of nurbs surfaces," *Computer-Aided Design*, vol. 56, pp. 34–44, 2014.

[3] B. Añón Lema, *Representación Interactiva de modelos masivos usando Mesh Shading.* Traballo de Fin de Grao da Facultade de Infromática, Universidade Da Coruña, 2021.

[4] R. Concheiro, M. Amor, E. J. Padrón, and M. Doggett, "Efficient Culling Techniques for Interactive Deformable NURBS Surfaces on GPU," in *11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2016)-Volume 1: GRAPP.* Rome, Italy: SciTePress, 2016, pp. 17–27.

[5] S. Jobalia, "Coming to Directx 12— Mesh Shaders and Amplification Shaders: Reinventing the Geometry Pipeline," 2019.

[6] E. Catmull and J. Clark, "Recursively generated B-spline surfaces on arbitrary topological meshes," *Computer-aided design*, vol. 10, no. 6, pp. 350–355, 1978.

[7] R. E. Barnhill and S. N. Kersey, "A marching method for parametric surface/surface intersection," *Computer aided geometric design*, vol. 7, no. 1-4, pp. 257–280, 1990.

[8] J. E. Cobb, "Tiling the sphere with rational bézier patches," in *TR UUCS-88-009.* University of Utah USA, 1988.

[9] L. Piegl and W. Tiller, *The NURBS Book*, 2nd ed. Springer Science & Business Media, 1996.

[10] J. Zink, M. Pettineo, and J. Hoxley, *Practical Rendering & Computation with Direct3D 11.* AK Peters, Ltd. / CRCPress, 2011.

[11] Microsoft, “Direct3d 11 graphics pipeline,” 2018, last accessed 2023-02-14. [Online]. Available on: https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline

[12] ——, “Primitive topologies,” 2020, last accessed 2023-02-14. [Online]. Available on: https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-primitive-topologies

[13] B. Langley, “Windows 10 and DirectX 12 released!” 2015, last accessed 2023-02-14. [Online]. Available on: https://devblogs.microsoft.com/directx/windows-10-and-directx-12-released/

[14] C. Kubisch, “Introduction to Turing Mesh Shaders,” 2018, last accessed 2023-02-14. [Online]. Available on: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/

[15] NVIDIA, *NVIDIA Turing Architecture Whitepaper.* NVIDIA, 2018.

[16] M. Guthe, A. Balázs, and R. Klein, “GPU-based trimming and tessellation of NURBS and T-Spline surfaces,” *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, pp. 1016–1023, 2005.

[17] Y. I. Yeo, L. Bin, and J. Peters, “Efficient pixel-accurate rendering of curved surfaces,” in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2012, pp. 165–174.

[18] A. Krishnamurthy, R. Khardekar, and S. McMains, “Direct evaluation of nurbs curves and surfaces on the gpu,” in *Proceedings of the 2007 ACM symposium on Solid and physical modeling*, 2007, pp. 329–334.

[19] K. Schwaber and J. Sutherland, “Scrum Methodology,” 2020, last accessed 2023-02-14. [Online]. Available on: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf

[20] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-time rendering*, 3rd ed. AK Peters/crc Press, 2008.

[21] A. Mihuț, C. Kubisch, and M. Kraemer, “Advanced API Perfomance: Mesh Shaders,” 2021, last accessed 2023-02-14. [Online]. Available on: https://developer.nvidia.com/blog/advanced-api-performance-mesh-shaders/