PhD Thesis

Real Time Rendering of Parametric Surfaces on the GPU

Raquel Concheiro Figueroa

2013

Departamento de Electrónica y Sistemas
Universidade da Coruña, Spain

Departamento de Electrónica y Sistemas

Universidade da Coruña, Spain

PhD Thesis

Real Time Rendering of Parametric Surfaces on the GPU

Raquel Concheiro Figueroa

March 2013

PhD Advisors:
Margarita Amor López
Montserrat Bóo Cepeda

Dr. Margarita Amor López
Titular de Universidad
Dpto. de Electrónica y Sistemas
Universidade da Coruña

Dr. Montserrat Bóo Cepeda
Titular de Universidad
Dpto. de Electrónica y Computación
Universidade de Santiago de Compostela

CERTIFICAN

Que la memoria titulada "*Real Time Rendering of Parametric Surfaces on the GPU*" ha sido realizada por Dª. Raquel Concheiro Figueroa bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 27 de Febrero de 2013

Fdo.: Margarita Amor López
Directora de la Tesis Doctoral

Fdo.: Montserrat Bóo Cepeda
Directora de la Tesis Doctoral

Vº Bº: Juan Touriño Domínguez
Director del Dpto. de Electrónica y Sistemas

The Dissertation Committee for Raquel Concheiro Figueroa certifies that this is the approved version of the following dissertation:

Real Time Rendering of Parametric Surfaces on the GPU

Committee:

President,

Member,

Member,

Member,

Secretary,

Resumen

A pesar de que el primer circuito electrónico específico para acelerar la síntesis de la computación gráfica fue diseñado a principios de los años ochenta, hubo que esperar hasta 1999 para que la Nvidia Geforce 256 popularizara el término GPU (*Graphics Processing Unit*). Desde este primer chip gráfico que procesaba un mínimo de diez millones de polígonos por segundo, hasta las GPUs actuales que ofrecen una solución competitiva para la computación masivamente paralela, ha habido una continua investigación y un crecimiento ininterrumpido. En los últimos años, la computación gráfica se ha expandido a muchas y muy diversas áreas científicas y tecnológicas, lo que ha impulsado la mejora de la arquitectura de la GPU con el objetivo de proporcionar la síntesis de modelos complejos y realistas.

Tradicionalmente, la estructura de las GPUs está orientada a triángulos y ha sido específicamente diseñado para procesar y sintetizar mallas de triángulos. Sin embargo, se ha demostrado que este enfoque es una solución insuficiente para la síntesis de modelos complejos, ya que, entre otras cosas, el bus que conecta CPU y GPU puede llegar a ser un cuello de botella en las aplicaciones que suponen un intercambio masivo de información.

Con el objetivo de minimizar los inconvenientes detectados en las GPUs orientadas a triángulos, esta tesis propone una nueva estructura de GPU orientada a superficies paramétricas. Aunque en este momento hay implementaciones comerciales que proporcionan, en mayor o menor medida, cierta flexibilidad para la evaluación de superficies paramétricas, ninguna de ellas ha sido diseñada para soportar la síntesis directa de superficies paramétricas sin ninguna conversión a mallas de triángulos.

La elección de superficies paramétricas viene motivada porque éstas proporcionan una representación compacta, lo que minimiza los requerimientos de memoria y

permite obtener un modelo más continuo y suavizado que una malla de triángulos. Además de todo esto, las características matemáticas de estas superficies proporcionan propiedades muy interesantes en el campo de la computación gráfica ya que, por ejemplo, simplifican la animación y la colisión debido a que usando superficies paramétricas menos puntos deben ser controlados que con la síntesis de polígonos.

De todas formas, el aspecto más relevante para el uso de superficies paramétricas viene de la mano de los nuevos campos de aplicación en distintas áreas científicas y tecnológicas. En estas áreas, la exactitud del modelo representado es especialmente relevante lo que supone un gran impulso hacia el desarrollo de modelos complejos basados en superficies paramétricas, puesto que éstos pueden ser descritos de una forma más precisa mediante ecuaciones que por medio de una malla de triángulos.

El esquema tradicional para la síntesis de superficies paramétricas, se basa en la evaluación y subdivisión de las superficies paramétricas en la CPU, donde se genera una malla de triángulos que posteriormente será sintetizada en la GPU. Esta tesis va un paso más allá y propone diferentes esquemas para la evaluación y síntesis de estas superficies directamente en la GPU. Inicialmente, se proponen diferentes esquemas para la síntesis de las superficies Bézier, entre los que podemos encontrar una propuesta para una subdivisión no adaptativa con dos alternativas: Vertex Shader Tessellation (VST) y Geometry Shader Tessellation (GST); otra propuesta para una implementación completamente adaptativa (DABT) basada en la generación de una malla más adaptada a las condiciones del modelo y que permite renderizar modelos de alta calidad con un menor número de primitivas de salida; y por último una propuesta de una subdivisión semi adaptativa que sintetiza las mejores características de las dos propuestas anteriores con el objetivo de obtener una subdivisión adaptativa pero con una evaluación más eficiente. Por otro lado, y como actualmente los dispositivos móviles son los dispositivos más habituales con capacidad para síntesis gráfica, esta tesis incluye un diseño para la síntesis de modelos de superficies Bézier (VSTHD) en las GPUs implementadas en estos dispositivos.

En último lugar, se ha propuesto una nueva estructura de GPU para la síntesis de superficies NURBS en tiempo real. Un profundo análisis de las características hardware para la generación de nuevas primitivas ha detectado que las actuales GPUs no son adecuadas para la síntesis de superficies NURBS. Por ese motivo, Rendering Pipeline for NURBS Surface (RPNS) se propone como una novedosa solución para la síntesis de superficies NURBS en la GPU sin ningún pre-proceso ni ninguna subdivisión previa.

El desarrollo de esta tesis sigue las líneas de metodologías clásicas; incluyendo planificación, diseño, análisis, evaluación y viabilidad de las implementaciones propuestas. Su principal objetivo es contribuir con propuestas que hagan viable una estructura de la GPU orientada a la síntesis de superficies paramétricas en vez de mallas de triángulos. Además, se ha realizado un profundo análisis para resaltar las limitaciones de las GPUs orientadas a triángulos y las propuestas aquí recogidas se han adaptado para explotar las características hardware de este tipo de arquitecturas.

En primer lugar, se desarrolló un estudio cuidadoso del estado del arte en el campo de las GPU, específicamente de las capacidades de síntesis de modelos complejos. Como resultado de este análisis se detectaron las principales debilidades de este tipo de arquitecturas, y posteriormente se han propuesto diseños orientados a minimizar los problemas previamente detectados. Es por eso, que esta tesis se centra en la síntesis de modelos complejos con superficies paramétricas, ya que esta representación soluciona, entre otros problemas, la degradación del rendimiento debido al cuello de botella del bus CPU-GPU, permite hacer un uso más eficiente de la memoria de la GPU gracias a que tienen una representación más compacta, y además, proporcionan una representación más exacta ya que se describe el modelo mediante ecuaciones matemáticas. Hay que destacar que esta tesis se ha enfocado tanto a superficies Béziers como a superficies NURBS. Las primeras porque debido a la rigidez y simplicidad de su representación matemática son adecuadas para ser evaluadas en la arquitectura de las GPUs actuales enfocadas a triángulos, y las segundas porque sus características matemáticas las hacen especialmente indicadas para la representación de modelos complejos, siendo en estos momentos un estándar *de facto* para el software de diseño y modelado asistido por ordenador (CAD/CAM), lo que hace su evaluación directa en la GPU deseable.

Finalmente, resaltar que todas estos diseños han contribuido a enfatizar las ventajas de la evaluación de superficies paramétricas en la GPU, proponiendo estrategias, que si bien han sido diseñadas con el objetivo de lograr una implementación hardware, han logrado la síntesis de modelos complejos en tiempo real con una implementación software adaptada a la arquitectura subyacente, superando en algunos casos el rendimiento obtenido por propuestas implementadas en hardware.

Esta tesis ha demostrado que una estructura de la GPU orientada a superficies paramétrica encaja perfectamente en la síntesis de modelos complejos, debido a las propiedades matemáticas de estas superficies las hacen especialmente adecuadas para la computación gráfica. A continuación se detallan las principales características y las conclusiones más relevantes extraídas de los diseños incluidos en esta tesis. En primer lugar se tratan los diseños enfocados a las superficies Bézier tanto en GPUs de escritorio como para dispositivos móviles. Posteriormente, se analiza el diseño enfocado a superficies NURBS.

En primer lugar, esta tesis detalla una propuesta no adaptativa para la subdivisión de superficies Bézier en la GPU, la cual está basada en la explotación de la coherencia espacial de la información. Cada superficie Bézier se considera como una primitiva de entrada a la GPU y se representa por medio de sus puntos de control. Esta propuesta se caracteriza por incluir una subdivisión y evaluación eficiente de las superficies basada en un acceso coherente a la memoria. Se han implementado dos alternativas diferentes, considerando para ello diferentes arquitecturas de la GPU. Por un lado, VST se ha diseñado para GPUs que no tienen la capacidad de generar primitivas en tiempo de ejecución, mientras que GST ha sido diseñado para las GPUs que cuentan con un generador de primitivas.

VST se caracteriza por la eficiente utilización del mapa paramétrico de vértices virtuales que permite realizar un ajuste preciso del nivel de subdivisión deseado. Los buenos resultados obtenido por esta estrategia son debidos a una reducción de las sincronizaciones CPU-GPU, y a una eficaz evaluación de las superficies Bézier que permite una eficiente explotación de la localidad de los datos.

Por otro lado, GST se basa en las capacidades de la generación de primitivas en la GPU. En este caso el mapa paramétrico de vértices virtuales se generan en tiempo de ejecución, disminuyendo los requerimientos de almacenaje. GST realiza una eficiente evaluación de las Bézier previamente al proceso de muestreado, reutilizando la evaluación de la superficie Bézier para cada nuevo punto generado. El principal problema de esta estrategia viene dado por la arquitectura de la GPU que se caracteriza por restringir el número de primitivas que pueden ser generadas por cada primitiva de entrada. Para evitar esta limitación, GST incluye un particionado en zonas que permite obtener una mayor resolución que la que teóricamente proporciona la GPU. Debido a esto, la evaluación de la superficie Bézier pasa a reutilizarse únicamente dentro de cada una de las zonas de particionado del mapa paramétrico.

Ambas alternativas muestran el impacto en el rendimiento de un shader sin divergencia y resaltan los beneficios de la explotación de la coherencia espacial. Esta propuesta demuestra que una evaluación basada en superficies paramétricas puede producir una síntesis en tiempo real de modelos complejos donde el modelo es subdividido y sintetizado en la GPU con un nivel de detalle seleccionado en tiempo de ejecución.

Esta tesis también analiza las ventajas y desventajas de una subdivisión adaptativa, mediante Dynamic and Adaptive Bézier Tesselation (DABT). A diferencia de la propuesta no adaptativa que genera un gran número de triángulos que no van a contribuir a la calidad de la imagen resultante, DABT es un esquema de subdivisión adaptativo donde el número de triángulos procesados se reduce considerablemente sin que ello suponga ninguna pérdida de calidad en la imagen final.

DABT propone un procedimiento de subdivisión sin ninguna estructura recursiva y donde la posición de los vértices candidatos a ser insertados pueden ser fácilmente evaluadas a través de sus coordenadas baricéntricas. Además, el nivel de subdivisión puede seleccionarse en tiempo de ejecución. La metodología empleada se basa principalmente en tres puntos: el uso de un patrón de subdivisión fijo que se encarga de guiar el procedimiento, el uso de test locales para decidir si un vértice se inserta o no y un procedimiento eficiente para la reconstrucción de la malla resultante. DABT sigue una propuesta adaptativa con tres niveles de resolución por cada triángulo procesado. En concreto, se selecciona un nivel de resolución por cada uno de los lados de los triángulos, y cada uno de ellos se proyecta hacia el centro baricéntrico del triángulo, ocupando una tercera parte del triángulo.

DABT se caracteriza por una subdivisión adaptativa con el objetivo de sintetizar los menos triángulos posibles sin perder calidad en la imagen final. Sin embargo, a pesar de sintetizar un número bastante menor de primitivas que la propuesta no adaptativa, el rendimiento obtenido es peor, debido a la divergencia introducida para permitir la adaptabilidad. Esta tesis resalta el impacto e inconvenientes de la divergencia en la ejecución en la GPU. Con el objetivo de mantener las características más relevantes de las propuestas completamente adaptativa y de la no adaptativa, se ha desarrolla una propuesta semi adaptativa.

La propuesta semi adaptativa es una solución intermedia entre la propuesta no adaptativa y la completamente adaptativa. En este caso, se intenta mantener un cierto grado de divergencia del flujo de los algoritmos, pero sin sacrificar por ello el rendimiento obtenido cuando no se introduce divergencia. Este esquema reduce la divergencia con el objetivo de conseguir una utilización óptima de los recursos computacionales de la GPU, aunque sigue manteniendo cierto grado de adaptabilidad. Con todo esto, se ha diseñado una propuesta que aunque consigue sintetizar muchos menos triángulos que la propuesta no adaptativa, consigue también reducir considerablemente la divergencia introducida en la propuesta completamente adaptativa.

El esquema semi adaptativo es una versión simplificada de la estrategia completamente adaptativa y está basada únicamente en un único nivel de resolución por triángulo. Las condiciones que determinan la inserción o no de nuevos vértices, sólo se aplican en las posiciones candidatas localizadas en los lados originales del triángulo grueso. Mientras tanto, la inserción de nuevos vértices en el interior del triángulo se basa en los vértices insertados en los lados del mismo.

Aprovechando nuestras propuestas para las GPUs de escritorio hemos proyectado una aproximación a las GPUs de los dispositivos móviles, ya que éstos son actualmente los dispositivos con capacidad de síntesis gráficas más habituales. Una nueva generación de GPUs ha sido específicamente diseñada para encajar en estos dispositivos porque los consumidores demandan continuamente mejoras en las capacidades de síntesis. Estas GPUs implementan únicamente un subconjunto de las características disponibles en una GPU de escritorio, ya que sus características hardware como su reducido tamaño o que están alimentados por baterías, no permite implementar una GPU de escritorio actual. En concreto, estas GPUs han sido diseñadas para ofrecer una alto rendimiento gráfico, pero mientras garantizan una reducción el consumo de potencia. Esta tesis presenta una propuesta para la subdivisión de superficies Bézier en esos dispositivos y se propone un esquema para la síntesis de superficies Béziers en tiempo real, Vertex Shader Tessellation in HandHeld Devices (VSTHD), especialmente enfocado a las características de estos dispositivos, que además de identificar los factores clave en las limitación del rendimiento y permite identificar aquellos aspectos mejorables.

Como las GPUs implementadas en dispositivos móviles no permiten la generación de primitivas en tiempo de ejecución, VSTHD se basa en una malla de vértices virtuales al igual que VST. Sin embargo, y debido al tamaño de la memoria implementada actualmente en estos dispositivos, se han desarrollado dos variantes: Uniform VSTHD que almacena en la memoria de variables uniformes y Texture VSTHD que utiliza la memoria de texturas.

Por último, contemplamos también el modelado con superficies NURBS, ya que son un estándar de facto en el software CAD/CAM. Como las superficies NURBS son considerablemente más complejas que las Béziers, habitualmente se convierten en superficies Béziers en la CPU y estas últimas son enviadas a la GPU para ser sintetizadas. Sin embargo, esta tesis va un paso más allá y propone el diseño de una nueva estructrua de GPU para una evaluación y subdivisión eficiente de superficies NURBS en la GPU, llamado Rendering Pipeline for NURBS Surfaces (RPNS).

Este diseño se caracteriza por proponer una GPU completamente orientada a superficies paramétricas, y consta de tres módulos: *geometry*, *sampler* and *rasterizer*. Inicialmente, los KSQuads se procesan en la etapa del *geometry*, posteriormente en la etapa de *sampler* se muestrean los KSQuads en KSDices, lo que permite un ajuste detallado en la densidad de primitivas para evitar huecos o *cracks* en la imagen final, y finalmente los KSDices son sintetizados para formar la imagen resultante.

RPNS propone dos nuevas primitivas: KSQuad, una primitiva de entrada y KSDice una primitiva de síntesis. KSQuad es una primitiva de entrada a la GPU que mantiene las propiedades geométricas de la superficie NURBS original y proporciona un manejo regular y flexible lo que reduce la divergencia en la evaluación. Por su parte, KSDice es la primitiva de síntesis propuesta en RPNS. Como es una estructura orientada a superficies paramétricas, en lugar de sintetizar triángulos se sintetizaran KSDice, donde cada uno de ellos es la proyección de un trozo muestreado de la NURBS original.

Por otro lado, una de las principales aportaciones de RPNS es que incluye una fórmula eficiente para calcular las funciones base de las superficies NURBS, lo que era uno de los principales inconvenientes para la evaluación de superficies NURBS en la GPU hasta este momento. Este método, llamado *Stair Strategy*, se basa en el analisis de las características matemáticas de las superficies NURBS y se caracteriza por evitar una evaluación recursiva de las funciones base.

Además, técnicas de *culling* como el *backpatch* y el *backface culling* se han contemplado en esta propuesta. En concreto se ha implementado una técnica de *backface culling* que hemos llamado *Dice Culling* que se sitúa al final de la etapa de *sampler* ya que se evalúa la orientación de los KSDices. Por otro lado, se han introcido dos novedosas técnicas de backpach culling, *Light Quad Culling* y *Strong Quad Culling.* Estas técnicas de *culling* se caracterizan por eliminar lo antes posible aquellas regiones de la superficie que no van a ser visibles. Por ese motivo, se sitúan al principio de la etapa del *geometry* y se dedican a comprobar si los distintos KSQuad están orientados hacia la cámara o no.

En resumen, esta memoria presenta diseños con diferentes grados de adaptabilidad para la subdivisión de las superficies Bézier, contemplando también el caso de las GPUs implementadas en los nuevos dispositivos móviles, y por último, se propone una estructura para la síntesis de superficies NURBS. Estas propuestas demuestran que incluso en las GPUs actuales orientadas a triángulos es posible sintetizar modelos complejos de superficies Bézier y NURBS en tiempo real.

En resumen, las principales contribuciones de esta Tesis son:

1. Una nueva propuesta no adaptativa que consigue tiempo real en la síntesis de superficies Bézier incluso en arquitecturas previas de la GPU, las cuales no disponen de un generador de primitivas. Esta propuesta se basa en un uso eficiente de la memoria, en una explotación de la localidad de los datos y, en aquellas arquitecturas con generador de primitivas, también en una reutilización de la evaluación de las superficies Bézier, ya que en este caso el proceso de muestreo es posterior a la evaluación de la superficie lo que permite reutilizar estos cálculos.

2. El diseño de una técnica de particionado en zonas de la superficie paramétrica, que permite soslayar la limitación hardware del número de primitivas que pueden generarse. En este caso, esta restricción limitaría la máxima resolución que se puede aplicar a cada superficie Bézier, limitando por tanto la calidad de la imagen resultante. Sin embargo, la técnica del particionado en zonas, a pesar de complicar levemente el cálculo de la Bézier, permite obtener cualquier nivel de resolución deseado.

3. El diseño de una técnica completamente adaptativa para la evaluación de superficies Bézier (DABT) basada en el uso de un patrón fijo de subdivisión y en un procedimiento eficiente de reconstrucción de la malla a sintetizar.

4. Un proceso de subdivisión de triángulos sin ninguna estructura recursiva y que permite aplicar un nivel de subdivisión por cada uno de los lados del triángulo. Como el nivel subdivisión se calcula en base a información local al lado, este proceso permite una gran adaptabilidad sin introducir ningún hueco entre triángulos vecinos, ya que los lados que se solapan evalúan las mismas condiciones de localidad.

5. Una propuesta semi adaptativa que permite realizar una evaluación adaptativa mientras se minimiza la divergencia del flujo del algoritmo asociada a este proceso adaptativo. Este técnica se caracteriza por aunar las ventajas de una subdivisión adaptativa, en la cual se reduce el número de triángulos a sintetizar, y las mejores condiciones de una subdivisión no adaptativa, que se caracteriza por permitir una evaluación sin divergencia.

6. Un esquema VSTHD, para la evaluación de superficies Bézier en los dispositivos móviles actuales. VSTHD está adaptado a las fuertes resticciones hardware de las GPUs de estos dispositivos, ya que actualmente no implementan ningún generador de primitivas, el tamaño de la memoria que poseen es bastante limitado y tienen un número reducido de núcleos.

7. Una propuesta, RPNS, específicamente diseñada para la síntesis de superficies NURBS en la GPU. En concreto RPNS es una estructura orientada a superficies paramétricas donde no resulta necesario hacer una conversión a mallas de triángulos. El diseño de RPNS consta de tres módulos diferenciados: *geometry*, *sampler* y *rasterizer*. En la etapa del *geometry* se procesa la primitiva de entrada; en la etapa de *sampler* se realiza el muestreo correspondiente, y por último en la etapa de *rasterizer* se sintetiza la primitiva de salida.

8. Una nueva primitiva de entrada a la GPU, llamada KSQuad que se caracteriza por mantener las propiedades geométricas de la superficie NURBS original mientras permite evaluar en paralelo e independientemente distintas zonas de la misma superficie NURBS, que puede tener un nivel de detalle muy diferente.

9. Una primitiva de síntesis llamada KSDice que permite una síntesis directa de superficies paramétricas sin ninguna conversión a malla de triángulos. Cada KSDice se genera en la etapa de *sampler*, donde se realiza un muestreado de la zona de la superficie NURBS representada por cada KSQuad, siendo cada KSDice la proyección de un trozo muestreado de la superficie NURBS original.

10. Una formulación eficiente y no recursiva del cálculo de las funciones base de las superficies NURBS, llamada *Stair Strategy*. Este método ha sido diseñado pensando en las características de una evaluación en una GPU, y por ello permite evitar la recursividad de la NURBS, principal razón que dificultaba el cálculo de estas funciones en la GPU. En concreto *Stair Strategy* reduce el cálculo de las funciones base a una suma de productos que pueden ser evaluados eficientemente en GPU.

11. Diseño e implementación de diferentes técnicas de culling que han sido situadas en diferentes etapas de RPNS. Mientras, *Dice Culling* está situada al final de la etapa de *sampler* y se basa en una técnica clásica de *backface culling*, se han introcido dos novedosas técnicas de backpach culling, *Light Quad Culling* y *Strong Quad Culling*.

Todas estas contribuciones se pueden resumir diciendo que el objetivo principal de esta tesis es proponer diseños que apoyen la implementación de una GPU orientada a superficies paramétricas en lugar de orientada a triángulos. Es decir, esta tesis demuestra la versatilidad de las superficies paramétricas, así como lo adecuadas que son para la síntesis de modelos complejos. Por último, destacar que a su vez, esta tesis contribuye con varias propuestas para la evaluación eficiente y en tiempo real de los modelos complejos representados con superficies paramétricas en GPUs.

A padrino,
A mi familia.

Acknowledgments

It would not have been possible to write this thesis without the help and support of the people around me, to only some of whom it is possible to give particular mention here.

Firstly, I owe my thanks to my thesis advisors, Marga and Montse, for the confidence they placed in me; I especially owe my deepest gratitude to Marga for her guidance and support, every single step of the way. I would also like to acknowledge to the GAC group, especially Emilio and the past and present members of the *Lab 0.2* for their kindness and friendship, which made my experience easier over these years.

I would also like to acknowledge Michael Doggett and the guys from the LUGG group for their warm welcome in Sweden. Also my acknowledgements to Marisa Gil and Xabier Martorell for being my hosts during my visit to the UPC.

Next, and foremost, there are not enough words to thank my loving family, who have been there for me every single day of my life. Thanks to my parents, Ruth and madrina, for giving me solid roots from which I could grow, for encouraging me and for their unswerving love and support. I would also like to thank Dyer for being always positive, for making me laugh and for his friendship, affection and unequivocal support, for which my mere expression of thanks does not suffice. Also my acknowledgements to my grandma, my aunt Loli and my cousins Rosalía and Xurxo for their outstanding support and for being a constant and active presence in my life. Finally, I owe my thanks to my friends, Suso, Ana, Yoli, Lourdes, Marta and to those for whom there is not enough space to give a particular mention here, for their invaluable support, patience and friendship. Many thanks to all of you, who have been helping me at different occasions and who, in one way or another, have influenced this thesis.

Finally, I am thankful to the following institutions for funding this work: Computer Architecture Group and the Department of Electronic and Systems at the University of A Coruña for the human and material support, the Ministry of Education and Science and the former Ministry of Science and Innovation of Spain for the projects TIN2007-67537-C03-02, TIN2010-16735 and the FPI Grant BES-2008-004533, Xunta de Galicia under the Program for Consolidation of Competitive Research Groups ref. 2010/06, 08TIC001206P and INCITE08PXIB105161PR, partially supported by FEDER funds and finally to the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).

For any errors or inadequacies that may remain in this work, of course, the responsibility is entirely my own.

Raquel.

Abstract

Although the first electronic circuit specifically designed to accelerate rendering was developed in the early 1980s, the term GPU (*Graphics Processing Unit*) was popularized by the Nvidia Geforce 256 in 1999. From this first single-chip processor, which processes a minimum of ten million polygons per second, to current GPUs, which offer a competitive solution to massive parallel computation, there has been continuous research and uninterrupted growth.

In recent years, the demand for computer graphics has expanded across many scientific and engineering areas. Hence, the interactive rendering of complex and realistic models has become a hot topic in computer graphics, supported by unstoppable development in the pipeline of the GPU.

Current GPU pipelines are triangle oriented and have been designed to process and render a large amount of triangles. Nevertheless, as the CPU-GPU bus is a habitual bottleneck, a triangle-oriented pipeline has proved to be a limited solution. As complex models can be more precisely described by equations than by a triangle mesh, parametric surfaces have gained ground as a new paradigm as they introduce relevant characteristics into the representation along with the rendering of complex models in real time. The compact representation provided by these surfaces reduces memory consumption and, moreover, its representation provides smoother, more continuous models than a set of triangles. Parametric surfaces can also select the level of detail on the fly and they are invariant under an affine transformation, thus they can be easily scalable. In addition to their mathematical characteristics, parametric surfaces provide interesting properties in computer graphics as animation and collision detection become simpler and faster than a set of polygons, owing to the fact that a much smaller number of points need to be processed. Nonetheless, parametric surfaces are usually tessellated as set of triangles in the CPU and finally

these triangles are sent down the GPU pipeline to be rendered. This dissertation goes a step further and proposes the evaluation and tessellation of parametric surfaces on the GPU.

This dissertation includes a deep analysis of the rendering of parametric surfaces on the GPU focused on the mathematical characteristics of parametric surfaces with the aim of providing an efficient strategy for rendering complex models in real time. Two different parametric surfaces have been analyzed: Bézier and NURBS surfaces. Bézier surfaces have been considered as an input primitive owing to their simple and regular representation. However, as the NURBS descriptions are more suitable for complex models, the direct rendering of NURBS models has also been analyzed in this thesis. In conclusion, this thesis elaborates on different strategies for the real time rendering of complex models represented as parametric surfaces.

In this thesis a set of schemes for the tessellation of Bézier surfaces on the GPU are designed: a non-adaptive approach, a fully adaptive proposal and a semi-adaptive approach with an intermediate degree of flexibility. The non-adaptive proposal is based on the on-the-fly generation of the parametric grid according to the level of resolution of each object and the camera position that determines the refinement degree of the surface. This proposal considers each Bézier surface as the input primitive to the pipeline, thus Bézier surfaces are tessellated and evaluated on the GPU and the computational power of current GPUs is exploited with a computational complex shader and an optimized memory access is designed. Although a single version of the proposal is possible, generating two different variants allows many specific details to be tweaked for optimal performance, depending on the specific GPU architecture. Therefore, a Vertex Shader Tessellation (VST) variant is designed for GPU which could only operate on existing data, such as pipelines based on DirectX9 Meanwhile a Geometry Shader Tessellation (GST) variant is designed for GPUs which allows the generation and destruction of geometric primitives, such as those based on DirectX10 or DirectX11.

With respect to the fully adaptive, Dyanamic and Adaptive Bézier Tessellation (DABT), and semi-adaptive proposals, the aim is to reduce the number of triangles in the final mesh while maintaining the quality of the resulting image. Surface tessellation must be sufficiently fine to capture geometric and appearance details. Nevertheless, overtessellating results in an increasing surface evaluation and rasterization workload. Both schemes are based on a 3-stage pipeline: first, a fixed tessellation pattern is computed to guide the adaptive procedure for the patch; next, the new vertices obtained from the first step are conditionally inserted by applying a set of heuristics consisting of tests local to the patch; finally, a specific scheme is employed to represent the inserted vertices and the reconstruction methodology based on the preprocessing of this information. The quality of the final triangle mesh is determined by both the inserted vertices and the reconstruction method employed to generate the resulting mesh. These proposals allow all triangles generated by them to be processed independently without introducing T-junctions or mesh cracks.

Unlike the DABT proposal, which permits multiple levels of resolution inside a patch, the semi adaptive proposal is characterized by a lower degree of divergence, reducing the adaptive degree of flexibility. This latter scheme is a tradeoff between a non-adaptive tessellation scheme and a fully adaptive proposal. The objective is to reduce the irregularity of the algorithm and the associated divergence of the DABT in order to optimize the graphics hardware utilization.

The final chapter in this dissertation goes a step further, and a new pipeline called Rendering Pipeline for NURBS Surfaces (RPNS) is presented. RPNS is a novel solution for the direct rendering of NURBS surfaces on the GPU with no previous tessellation procedure or preprocessing. A deep analysis of current GPU pipeline evinces that the current stages for primitive generation, such as geometry shader or tessellator, are not suitable for the direct rendering of NURBS surfaces on the GPU. Hence, a NURBS-oriented pipeline (RPNS) has been designed according to the geometric characteristics of NURBS surfaces. The aim is to efficiently render 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, interactive trimming or surface intersection. RPNS is based on a new primitive called KSQuad, which allows the direct rendering of NURBS surfaces. The design of RPNS relies on two mainstays to achieve sound performance and high-qualilty results: adaptive discretization of KSQuad and evaluation of NURBS surfaces with no approximation.

To test our proposals, and even though this thesis focuses principally on algorithmic improvements to the rendering pipeline rather than an optimized implementation, these proposals have been implemented to measure their performance on current GPUs, achieving real-time rendering rates.

Contents

List of Tables

List of Figures

Chapter 1

Tessellation of parametric surfaces

For the last thirty years, interactive graphic systems have become a hot topic widely developed by the community with unstoppable growth. GPU (*Graphics Processing Unit*) research has been supported by an increasing real-time rendering demand for complex and realistic models across many engineering and scientific areas, such as medicine, computer-aided design (CAD), virtual reality, video games and computational biology. Hence, the interactive rendering of complex and realistic images in GPUs has been a longstanding goal in computer graphics.

Real time rendering of scenes is the traditional mainstay of computer graphics. Even though an off-line renderer that takes minutes is acceptable, real-time graphics should synthesize frames in a few milliseconds. However, owing to the high cost of 3D graphic operations, it is critical for GPU implementations of the real-time graphics pipeline to be highly efficient systems. Hence, a scene is usually represented using a low-resolution mesh in interactive graphics, and accordingly designers usually omit complex objects from environments or approximate them coarsely using polygons that cover many pixels on the screen.

Nevertheless, graphic designers have been demanding powerful creative control over image generation and a realistic and detailed image synthesis. The most common complex models are designed with parametric and subdivision surfaces. Subdivision of surfaces [16, 32, 59] is a powerful tool for modelling smooth surfaces of arbitrary topology with complex details from coarse meshes with complex details, and it is used in a wide range of applications, such as 3D games, movie production and commercial modelers from coarse meshes. Evaluating subdivision surface involves a recursive linear interpolation to refining the coarse mesh. An approximate conversion from subdivision surfaces into parametric surfaces is viable [60, 93]. Hence, this thesis is focused in parametric surfaces. Specifically NURBS (*Non-uniform rational B-splines*) surfaces are one of the most useful primitives employed for high quality modelling, as they are *de facto* standard in CAD/CAM tools and graphic software. Despite the fact that computational capability continues to increase, current interactive graphics systems are still a long way off being able to interactively synthesize realistic and detailed scenes.

Traditionally, GPUs pipelines are triangle-oriented, hence they have been designed for the efficient processing of triangles and they do not work properly with parametric surfaces. Since direct evaluation of NURBS surfaces on the GPU is a highly complex task, they are usually converted into Bézier surfaces as a pre-processing step and then Bézier surfaces are usually tessellated in the CPU (*Central Processing Unit*) and the set of generated triangles is sent to the GPU. Obviously, for numerous reasons, it is not practical for an interactive application to compute in the CPU and send a high-resolution mesh representation of an entire scene to the GPU. For example, doing so would incur substantial storage and bandwidth costs; increase the cost of non-rendering operations, such as simulation and animation, which do not require high-resolution meshes; and make it challenging to provide the appropriate level of mesh detail for all possible views.

An alternative proposal for reducing triangle mesh costs is to compute Bézier surfaces on the GPU. In this process, the NURBS surfaces are converted into Bézier surfaces on the CPU, and then evaluated and tessellated on the GPU. As Bézier structures are simpler than NURBS, evaluating the former is easier and faster. Despite this mathematical conversion being a high-consumption process unsuitable for interactive graphics, current proposals and current GPU designs focus on this approach owing to the fact that NURBS surfaces are too complex to be processed on the GPU on the fly.

In the literature, there have principally been two different approaches to the synthesis of parametric surfaces on the GPU: tessellation and direct evaluation. The former performs the tessellation of the parametric models directly on the GPU. Recent tessellation proposals exploit the programmable capabilities of current graphics cards to achieve the real-time rendering of parametric models on the GPU [21, 33, 47, 93]. In these proposals the rendering process is performed per patch [21, 47] or per set of patches, depending on the level of detail required [33]. In this approach the computational cost increases with the number of patches, owing to the amount of synchronous calls between CPU and GPU. [93] 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, hole-free surfaces from a NURBS surface. Another tessellation approach is presented in [34, 35, 84], where the tessellation of bicubic Bézier surfaces is performed following a GPGPU strategy (General-Purpose Computation on GPU) using CUDA.

A different approach to tessellating parametric surfaces is the dicing of these surfaces into micropolygons, small quadrilaterals each less than one pixel in size. The starting point of this approach is the Reyes rendering system [29], based on the development of a new and different pipeline. Even though the rendering performance of the Reyes system is far from meeting real-time requirements, different characteristics of this pipeline have been ported to GPUs [77, 92, 95]. Other proposals based on the modification of the GPU pipeline for implementing micropolygon rendering are found in [39, 41, 93].

The second approach to the synthesis of parametric models is their direct evaluation on the GPU [50, 55]. However, this alternative either needs to use multiple fragment programs for the different surface degrees, [50] or presents the requirement of complex fragment shaders with several stages and with the parametric grid determined on the CPU [55].

One strategy which is close to the direct evaluation of parametric surfaces is the synthesis with *Ray tracing* algorithms [1, 65, 71]. These algorithms produce very good visual results but with high timing requirements.

This chapter is organized as follows. Firstly in Section 1.1 tessellation capabilities of current graphics card are summarized. In Section 1.2 an introduction to the parametric surface focused on Bézier and NURBS surfaces is given. Lastly, an overview of this thesis content is given in Section 1.3.

1.1. Tessellation options in current graphics cards

In this section we briefly summarize the structure of current GPUs and the available hardware options for tessellation. The objective is to provide a brief revision of the hardware possibilities and to summarize the working framework and the reasons behind our proposals. With the tessellation procedure in mind, we shall focus our analysis on the programmable stages and the possibilities for implementing a tessellation procedure on them.

Until the end of 2006, the programmable vertex and pixel shaders found in the GPUs could only operate on existing data [2, 42, 46, 88] (see Figure 1.1). The vertex shader has traditionally been employed for vertex transformations and per-vertex computations. No vertices can be generated or destroyed in this unit and each vertex has no access to the information associated with another vertex. The pixel shader is usually employed for the computation of each fragment's color. Neither of these two units could be employed for generating/destroying geometry in a direct way. The scene changed a few years ago with the introduction with DirectX 10 (see Figure 1.2) of a new programmable unit, the geometry shader [9, 72, 91]. The geometry shader allows the generation and destruction of geometry data on the graphics processor introducing, with this characteristic, an incredible range of new possibilities. In consequence, computer graphics algorithms that were traditionally condemned to have limitations in their GPU implementation can be reconsidered again for an efficient GPU implementation. Clear examples are mesh simplification [30] and mesh tessellation [62].

The geometry shader works with primitives (point, line segment, or triangle) and the output number of primitives may be higher or lower than the input number. Adjacent information is available so that for each triangle the information of the three neighbouring triangles can be accessed. However, the main drawback is the limitation on the number of output primitives per invocation, as currently only 1024 32-bit values can be output [9, 72]. The intermediate results processed by the vertex shader or the geometry shader can either be sent back to the pipeline through stream out, allowing iterative processing, or sent directly to the rasterization stage.

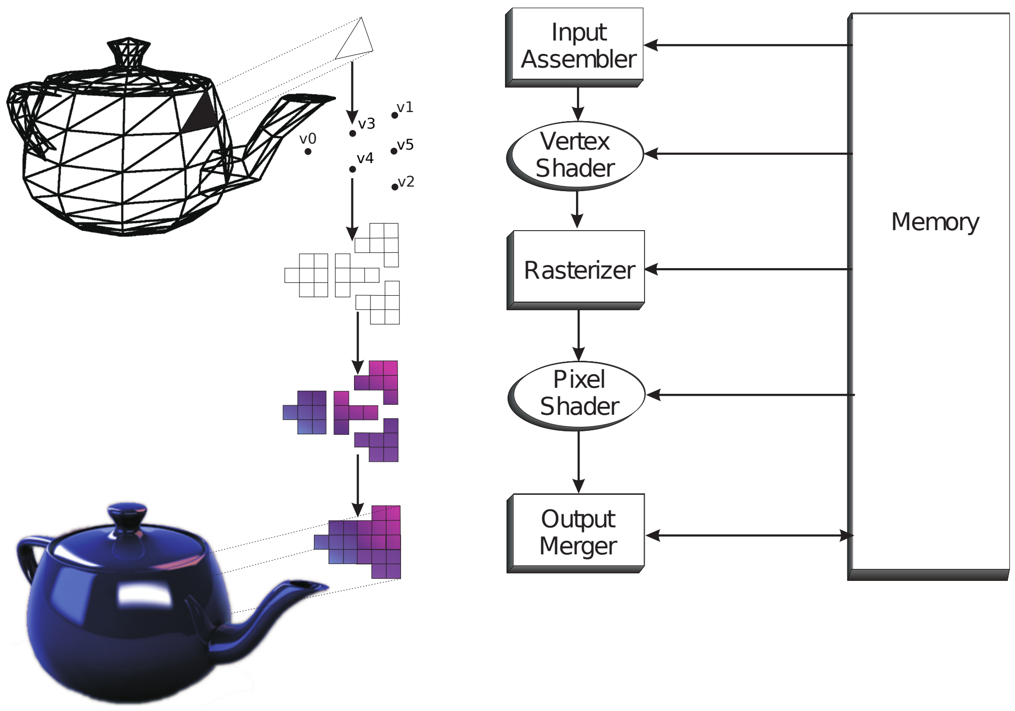


Figure 1.1: DirectX 9 pipeline

With DirectX11 three new stages (hull shader, tessellator unit and domain shader) were introduced to support programmable tessellation (see Figure 1.3) [15, 63, 90]. These new stages are inserted between the vertex and the geometry shader. The hull shader and the domain shader are programmable stages, whereas the unit where the real data expansion takes place, the tessellator, is a configurable stage. As is shown in Figure 1.4, the hull shader is invocated once for each input primitive; in the figure is invocated once for each Bézier patch. It is the first stage of the tessellation procedure and it configures tessellator and domain shader execution. Hence, the hull shader generates two different outputs to guide the tessellation procedure: one is sent to the domain shader while the other is sent to the tessellator. Both outputs include the tessellation factors which are generated on the fly in the hull shader. More specifically, from the hull shader the DirectX11 tessellator receives six independent tessellation factors, one for each domain edge and two for the internal axes of the patch. This new tessellation unit [90] offers a high performance solution, but with reduced flexibility in its current implementation, as it applies a fixed or

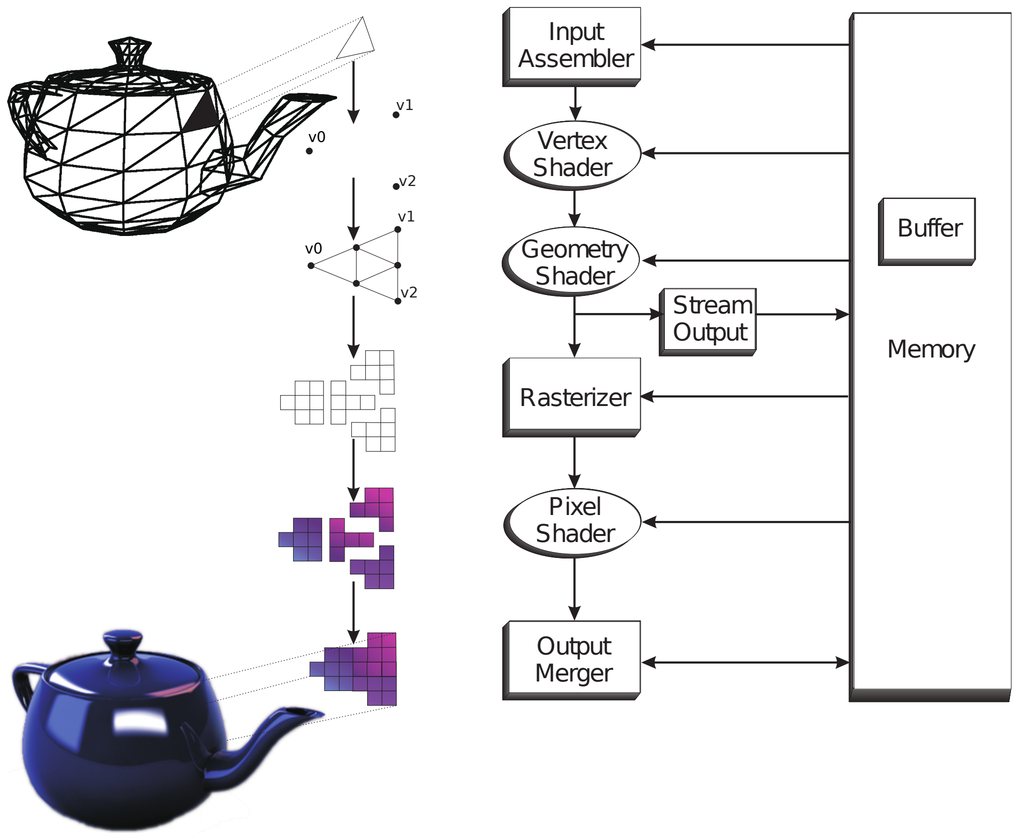


Figure 1.2: DirectX 10 pipeline

a semi-regular tessellation pattern (see Figure 1.4). Once these factors are set, the edges and the inside of the patch are uniformly tessellated in the parametric domain (see Figure 1.4). Finally, the domain shader receives the parametric coordinates from the tessellator as well as the input primitive and the tessellator factors from the hull shader. According to the recived data, these parametric coordinates are evaluated in the domain shader; i.e. they are invocated once for each parametric coordinate generated in the tessellator. In [93] an implementation of a tight estimator of the variance between the screen projection of the exact surface and its triangulation is proposed using the GPU tessellation engine. This tessellation unit also supports regular fractional tessellation, and some works, such as [6, 67], add a non-uniform, fractional tessellation to achieve a more uniform screen-space triangle area. Nevertheless, this scheme does not provide enough support for free adaptive tessellation, and the independent primitive process requires special care by

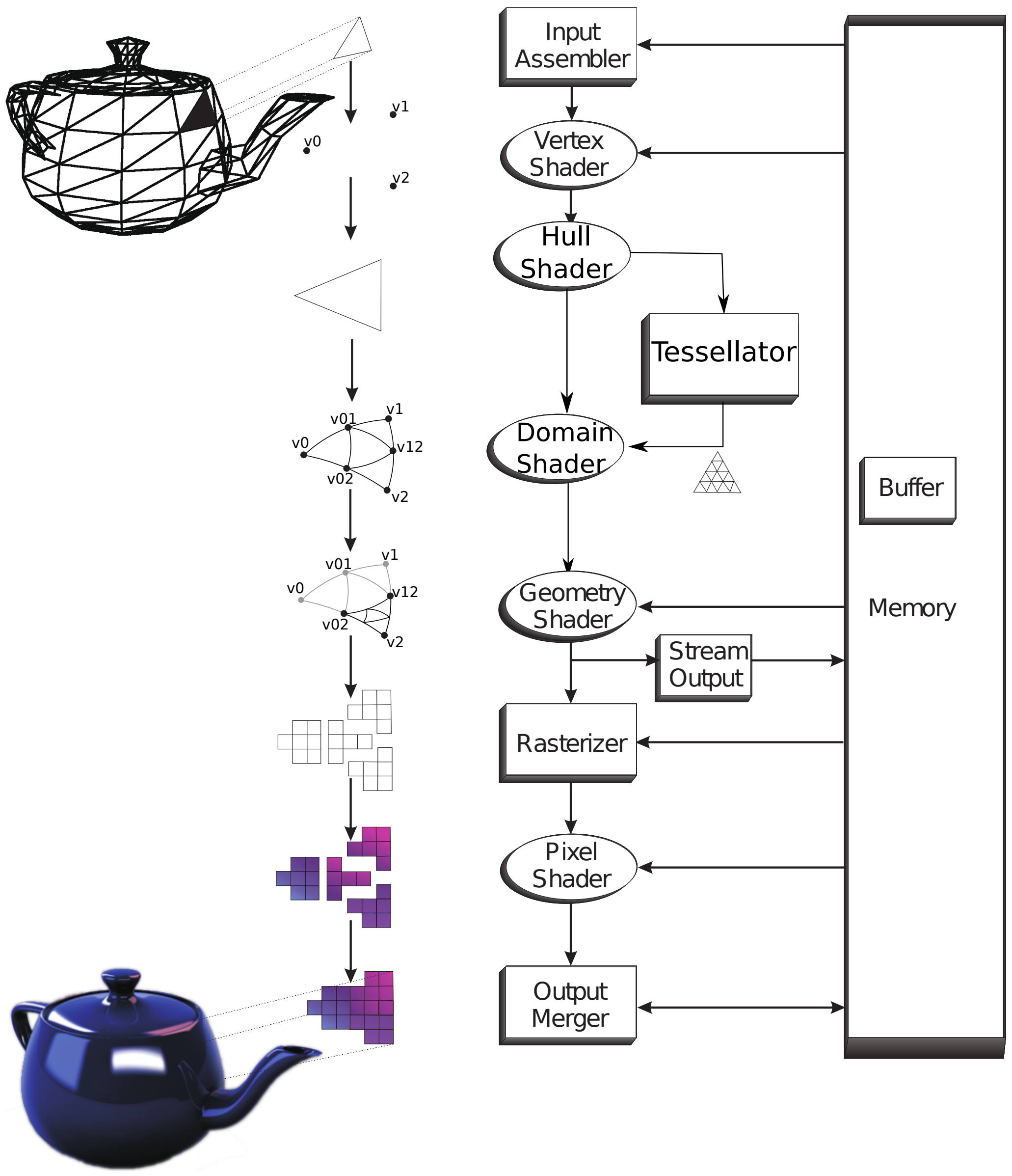


Figure 1.3: DirectX 11 pipeline

application developers to prevent cracks.

In DiagSplit proposal [40], a modification of the DirectX11 hardware structure is proposed to allow greater adaptability, though still keeping a uniform strategy per surface patch. The DiagSplit algorithm [40] generates view-dependent adaptive tessellation with a recursive approach, where sub-patches are created with the

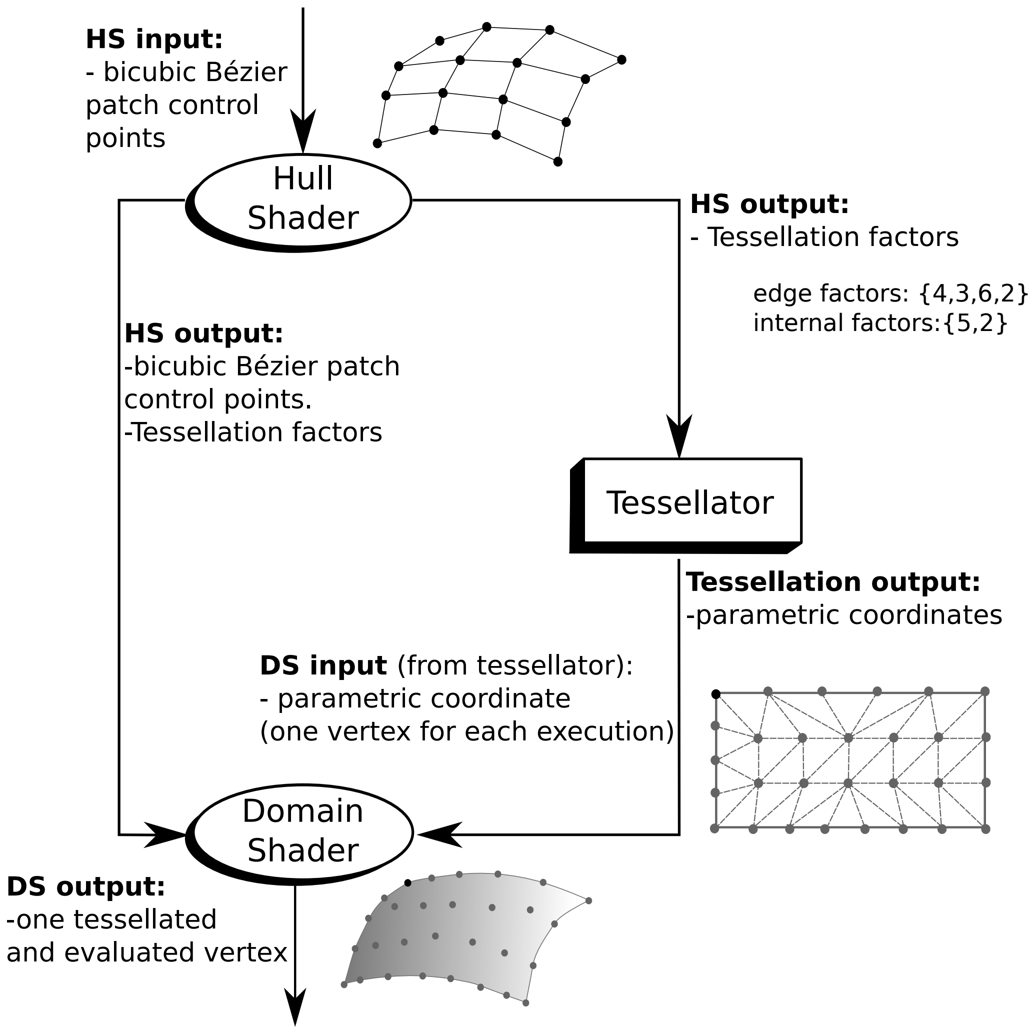
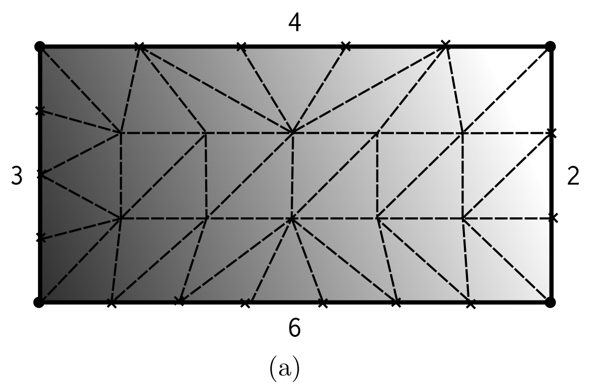


Figure 1.4: Tessellation structure in the DX11 pipeline

evaluation of edges if non-uniform tessellation is required. DiagSplit performs a non-uniform tessellation along an edge by applying a recursive process: first, the edge is partitioned at its parametric midpoint, and then seven factors are used, one for each edge of the two sub-patches. This proposal, however, is a long way off obtaining an adaptive tessellation inside the patch as the interior of each sub-patch or patch is uniformly diced according to the tessellation factor of their edges.

Figure 1.5 depicts how a patch is partitioned into triangles with the DirectX11 tessellation unit compared to the DiagSplit proposal. In this example it is assumed



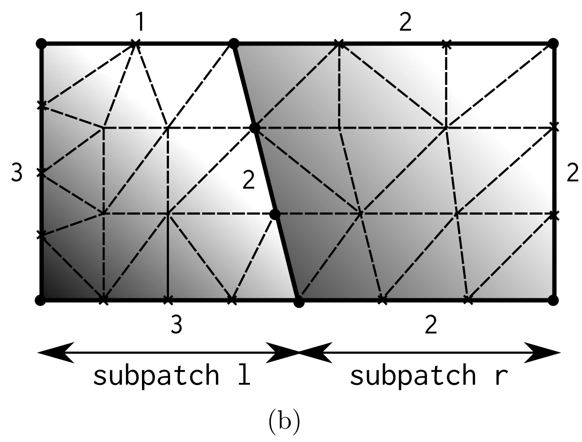


Figure 1.5: Comparison of two different tessellation methods (a) DirectX11 Tessellation Unit (b) DiagSplit [40]

that the left side of the patch has a greater complexity and needs a higher tessellation factor. Figure 1.5(a) shows the patch after being tessellated by DirectX11 using the factors {4, 3 ,6, 2} for the edges and {5, 2} inside the patch, resulting in a mesh of triangles with similar shape and size. In Figure 1.5(b), the partitioning proposed in DiagSplit is applied. In an intitial step the patch is partitioned into two sub-patches (*sub-patch l* and *sub-patch r*). Each sub-patch is subsequently split according to four tessellation factors, $\{1, 3, 3, 2\}$ and $\{2, 2, 2, 2\}$, respectively. As shown, DirectX11 tessellator provides a configurable tessellation factor where this tessellation process cannot be modified inside the patch; consequently it is not suitable for the tessellation of surface with a high degree of variability inside the surface, such as NURBS surfaces. DiagSplit provides a more configurable and recursive tessellation procedure, with a regular tessellation pattern inside each patch.

Broadly speaking, the tessellation of surfaces in the DirectX11 pipeline is known for the lack of flexibility of sampling schemes in the tessellation unit, as well as for the independent evaluation of each sample in the domain shader.

1.2. Parametric surfaces

Curves and surfaces are mathematically represented either explicitly, implicitly or parametrically [42]. As coefficients of many parametric functions introduce considerable geometric significance, the parametric form is more natural for designing and representing shape in a computer. Hence, the present thesis is concerned with parametric representation; more specifically, Béziers and NURBS surfaces are considered.

Parametric representation is extremely flexible as each of the coordinates on the curve or on the surface is represented separately as an explicit function, so they are axis independent. Although not strictly necessary, a function defined in the interval $[a, b]$ is usually normalized to $[0, 1]$.

Despite the advantageous characteristics of parametric surfaces, several typical operations, such as determining the intersection of two parametric curves or finding the distance from a point to a curve, are considerably more difficult in a parametric representation.

Although many surfaces can be analytically represented, there are also many surfaces for which an analytical description does not exist. In this case, surfaces are represented in a piecewise fashion, where each individual patch is joined together along the edges to create a complete surfaces. Within this context, a patch is a curve or surface which represent a piece of the model and it is joined together with another patches to represent the whole model.

As parametric curves and surfaces are created as a join of patches, continuity is a relevant factor to curve or surface smoothness. Specifically, two different kinds of continuity associated to parametric curves and surfaces can be defined: geometric, G, and parametric continuity, C. According to geometric continuity, a curve or surface has a G^0 continuity at the join if two curves or surface segments are joined together at their respective end point. Meanwhile, the resulting curve is G^1 if the slope of the tangent vectors at the join are geometrically equals;

Like geometric continuity, if two curves or surfaces are joined together, the resulting curve or surface is said to have C^0 continuity at the join. However, the curve or surface is said to be C^1 continuous if the tangent vectors at the join have the same direction and the same magnitude. Hence, parametric continuity is more restrictive than geometric continuity.

If the resulting curve or surface is C^1 at the join, there is a smooth transition from one curve or surface segment to the next. However, if the curve or surface is only G^1 at the join, there is a more abrupt transition.

In the next, the most usual parametric representation, Bézier and NURBS, are detailed.

1.2.1. Bézier Representation

In this section a brief introduction to the Bézier representations is presented [37, 38, 79, 82]. Bézier representations are a special case of NURBS and they are commonly used owing to their regular structure and simplicity. For reasons of clarity we start the presentation by introducing Bézier curves and we subsequently extend the description to Bézier surfaces.

Bézier curves

A Bézier curve is specified by giving a set of coordinate positions, called control points, which indicate the general shape of the curve, as shown in Figure 1.6. These control points are then fitted with piecewise continuous parametric polynomial functions. Mathematically, a parametric n-degree Bézier curve is defined by:

$$P(t) = \sum_{i=0}^{n} B_i J_{n,i}(t), \quad 0 \leq t \leq 1 \tag{1.1}$$

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

$$J_{n,i}(t) = \binom{n}{i}(1-t)^{(n-i)}t^i \tag{1.2}$$

where n is the degree of the Bézier basis function. These functions decide the extent to which a particular control point controls the surface at a particular parametric value t. Only $n+1$ control points and the n-degree Bernstein polynomials are required for the computation of each point of the curve.

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

$$P(t) = [T][N][G] \tag{1.3}$$

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

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

For example, for $n = 3$ the matrix form is:

$$P(t) = [T][N][G] = [t^3 \;\; t^2 \;\; t^1 \;\; 1]\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

First of all, Bernstein basis functions properties derived from Equation 1.2 are detailed:

1. The basis functions are real.

2. Non-negativity:

$$J_i(u) \;\geq\; 0 \;\forall\; i \;\mathit{and}\; 0 \;\leq\; u \;\leq\; 1 \tag{1.4}$$

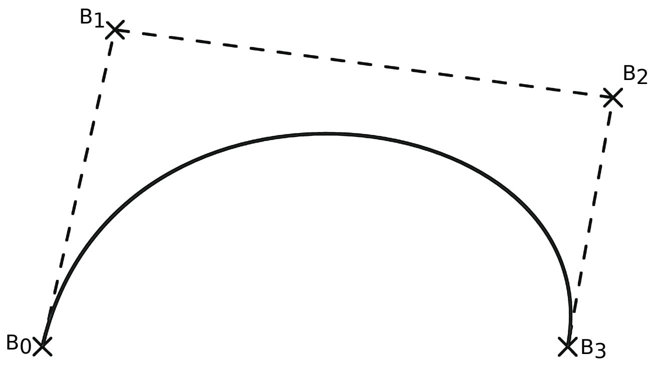


Figure 1.6: Cubic Bézier curve, $n = 3$

3. Partition of unity:
$$\sum_{i=0}^{n} J_i(u) = 1 \ \forall \ 0 \leq \ u \ \leq \ 1 \tag{1.5}$$

4. $J_0(u) = J_n(1) = 1$.

5. Symmetry: for any i, the set of polynomials $J_i(u)$ is symmetric with respect to $u = \frac{1}{2}$.

6. Recursive definition: $J_i(u) = (1 - u)J_i(u) + uJ_{i-1}(u)$; defining $J_i(u) \equiv 0$ if $i < 0$ or $i > n$.

Next, properties summarize the geometric characteristics of Bézier curves:

1. The degree of the polynomial defining the curve segment is one less than the number of control polygon points.

2. The curve generally follows the shape of the control polygon.

3. The first and last points on the curve are coincident with the first and last points of the control polygon; i.e., $P(0) = B_0$ and $P(1) = B_n$.

4. The tangent vectors at the ends of the curve have the same direction as the first and last polygon spans, respectively.

5. Convex hull property: the curve is contained within the convex hull of the control polygon; i.e., within the largest convex polygon defined by the control

polygon vertices. In Figure 1.6, the convex hull is shown by the dashed line and an imaginary straight line from the start point to the last point.

6. Variation diminishing property: As a Bézier curve follows its control polygon rather closer and does not wiggle more than its control polygon, no straight line intersects the Bézier curve more times than it intersects the control polygon.

Bézier surfaces

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

$$Q(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,j} J_{n,i}(u) J_{m,j}(v), \quad 0 \leq u, v \leq 1 \tag{1.6}$$

where $J_{n,i}(u)$ and $J_{m,j}(v)$ are the Bernstein basis functions in the u and v parametric directions and $B_{i,j}$ are the vertices of a polygonal control net. Again, the number of control points in the u and v directions are $n + 1$ and $m + 1$, respectively. As an example, Figure 1.7 shows a bicubic Bézier surface, $n = m = 3$.

In matrix form, a Bézier surface is given by:

$$Q(u, v) = [U][N][B][M]^T[V] \tag{1.7}$$

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

$$\begin{aligned} Q(u, v) = [u^3 \quad u^2 \quad u \quad 1] &\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} &\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \end{aligned} \tag{1.8}$$

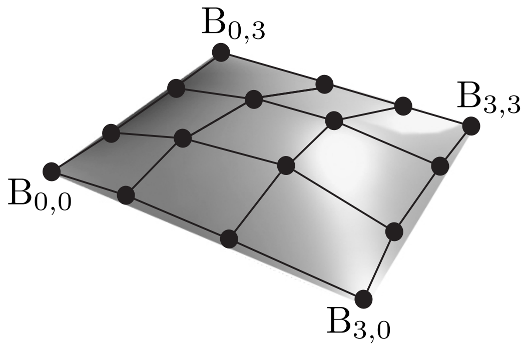


Figure 1.7: Bicubic Bézier surface, $n = m = 3$

As a Bernstein basis is used for surface blending functions, many properties of the Bézier surfaces are known. Below, several properties are summarized:

1. Non-negativity:
$$J_{i,n}(u) \ \geq\ 0 \ \forall\ i, n \ \mathit{and}\ 0 \ \leq\ u \ \leq\ 1 \tag{1.9}$$

2. Partition of unity:
$$\sum_{i=0}^{n} J_{i,n}(u) \ =\ 1 \ \forall\ 0 \ \leq\ u \ \leq\ 1 \tag{1.10}$$

3. $J_{i,n}(0) = J_{n,n}(1) = 1$

4. Symmetry: for any n, the set of polynomials $J_{i,n}(u)$ is symmetric with respect to $u = \frac{1}{2}$.

5. Recursive definition: $J_{i,n}(u) = (1-u)J_{i,n-1}(u) + uJ_{i-1,n-1}(u)$; we define $J_{i,n}(u) \equiv 0$ if $i < 0$ or $i > n$.

The following properties summarize the geometric characteristics of Bézier surfaces:

1. The degree of the surface in each parametric direction is one less than the number of control net vertices in that direction.

2. The continuity of the surface in each parametric direction is two less than the number of control net vertices in that direction.

3. The surface generally follows the shape of the control net.

4. Only the corner points of the control net and the resulting Bézier surface are coincident.

5. The surface is contained within the convex hull of the control net.

6. The surface is invariant under an affine transformation.

It is interesting to note that there is no known variation diminishing property for Bézier surfaces [80].

1.2.2. Non Uniform Rational B-Splines (NURBS)

In this section an introduction to NURBS surfaces is presented. A more detailed review can be found in [37, 38, 79, 82]. In order to clarify the explanation, we start by introducing NURBS curves and then go on to extend the description to NURBS surfaces.

NURBS are *de facto* standard to CAM/CAD applications. According to [79], the main reason for the widespread acceptance and popularity of NURBS representation is their common mathematical form for representing and designing both standard analytic shapes and free-form curves and surfaces. Moreover, they provide the flexibility to design a large variety of shapes by manipulating the control points and weights. NURBS evaluation is reasonably fast and computationally stable and they are invariant under scaling, rotation, translation and shear as well as parallel and perspective projection. NURBS are genuine generalizations of non-rational B-spline forms as well as rational and non-rational Bézier curves, and a powerful geometric tool kit, including knot insertion, refinement, removal, degree elevation and splitting, has been designed. Furthermore, NURBS have clear geometry interpretations, making them particularly useful for designers.

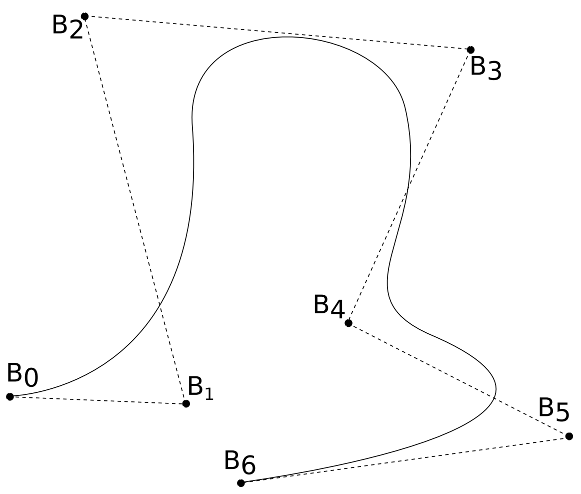


Figure 1.8: Cubic NURBS curve

NURBS Curves

A *pth*-degree NURBS curve (see Figure 1.8) is defined by:

$$C(u) = \frac{\sum_{i=0}^{n} N_{i,p}(u)\ w_i B_i}{\sum_{i=0}^{n} N_{i,p}(u) w_i}, \quad a \leq u \leq b \tag{1.11}$$

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

$$\mathrm{U} = \{\underbrace{0, \cdots, 0}_{\text{p+1}}, x_{p+1}, \cdots, x_{m-p-1}, \underbrace{1, \cdots, 1}_{\text{p+1}}\} \tag{1.12}$$

where $m = n + p + 1$. Unless otherwise stated, we assume that $a = 0, b = 1$ and $w_i > 0$ for all i.

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

$$N_{i,p}(u) = \frac{u - x_i}{x_{i+p} - x_i} N_{i,p-1}(u) + \frac{x_{i+p+1} - u}{x_{i+p+1} - x_{i+1}} N_{i+1,p-1}(u) \tag{1.13}$$

with

$$N_{i,0}(u) = \begin{cases} 1 & \textbf{if} \quad x_i \leq u < x_{i+1} \\ 0 & \textbf{otherwise} \end{cases} \tag{1.14}$$

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.

Figure 1.8 shows a cubic NURBS curve defined by its control point net $B = \{B_0, B_1, B_2, B_3, B_4, B_5, B_6\}$, its knot vector $U = \{0, 0, 0, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, 1, 1, 1\}$ and the curve weights are equal to 1.

First of all, basis function properties derived from Equation 1.21 are explained:

1. Step function: $N_{i,0}$ is a step function, equal to zero everywhere except on the half-open interval $u \in [x_i, x_{i+1})$.

2. For $p > 0$, $N_{i,p}$ is a linear combination of two $(p-1)$-degree basis functions.

3. Non-negativity: Basis functions are positive and real functions for each knot, degree and parametric position:

$$N_{i,p}(u) \geq 0 \;\; \forall i, p \; and \; u \in [0,1] \tag{1.15}$$

4. Partition of unity:

$$\sum_{i=0}^{n} N_{i,p}(u) = 1 \; \forall \; u \in [0,1] \tag{1.16}$$

5. Local support: $N_{i,p}(u) = 0$ for $u \notin [x_i, x_{i+p+1})$. Furthermore, in any given knot span, at most $p+1$ of the $N_{i,p}(u)$ are non-zero (in general $N_{i-p,p}(u), ... N_{i,p}(u)$ are non-zero in $[x_i, x_{i+1})$).

6. All derivatives of $N_{i,p}(u)$ exist in the interior of a knot span, where it is a rational function with non-zero denominator. At a knot, $N_{i,p}(u)$ is $p-k$ times continuously differentiable, where k is the multiplicity of the knot.

The following properties summarize the geometric characteristics of NURBS curves:

1. The control polygon represents a piecewise linear approximation to the curve; this approximation is improved by knot insertion or degree elevation. As a general rule, the lower the degree the closer a curve follows its control polygon.
2. A curve $C(u)$ with degree p, number of control points $n+1$ and number of knots $m+1$ are related by:

$$m = n + p + 1 \tag{1.17}$$

3. $C(0) = B_0$ and $C(1) = B_n$.
4. Affine invariance: an affine transformation is applied to the curve by applying it to the control points; NURBS curves are also invariant under perspective projections, a fact which is important in computer graphics.
5. Strong convex hull property: if $u \ \in [x_i, xi+1)$, then $C(u)$ lies within the convex hull of the control points $B_{i-p}, \cdots, B_i$.
6. $C(u)$ is infinitely differentiable on the interior of the knot spans and is $p-k$ times differentiable at a knot of multiplicity k.
7. Variation diminishing property: no plane has more intersections with the curve than with the control polygon.
8. A NURBS curve with no interior knots is a rational Bézier curve, since the $N_{i,p}(u)$ reduce to the $B_{i,n}(u)$. So, NURBS curves contain non-rational and rational Bézier curves as a special case, as shown in Figure 1.9. In this Figure, the same curve which is drawn as a set of Bézier curves in Figure 1.9(a) is also drawn as a NURBS Curve in Figure 1.9(b).

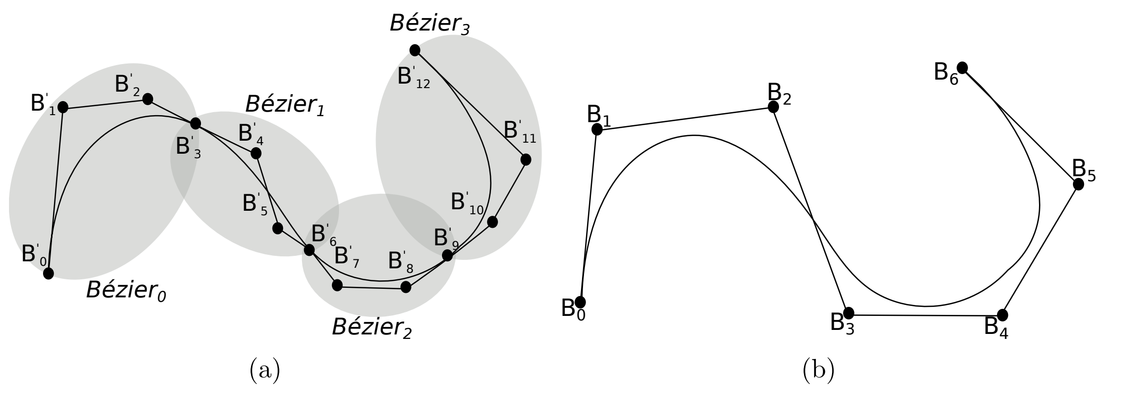


Figure 1.9: A cubic curve represented as (a) a group of Bézier curves or (b) a NURBS Curve

9. Local approximation: if the control point B_i is moved, or the weight w_i is changed, it affects only that portion of the curve on the interval $u \in [x_i, x_{i+p+1})$ This property is highly important for interactive shape design, as control point movement and weight modification can be used to attain local shape control.

NURBS Surface

A NURBS surface (see Figure 1.10) is obtained as the tensor product of two NURBS curves, and is 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:

$$S(u,v) = \frac{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)\ N_{j,q}(v)\ w_{i,j}B_{i,j}}{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)\ N_{j,q}(v)\ w_{i,j}}, \quad 0 \leq u,v \leq 1 \tag{1.18}$$

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 non-rational B-spline basis function defined on two knot vectors of

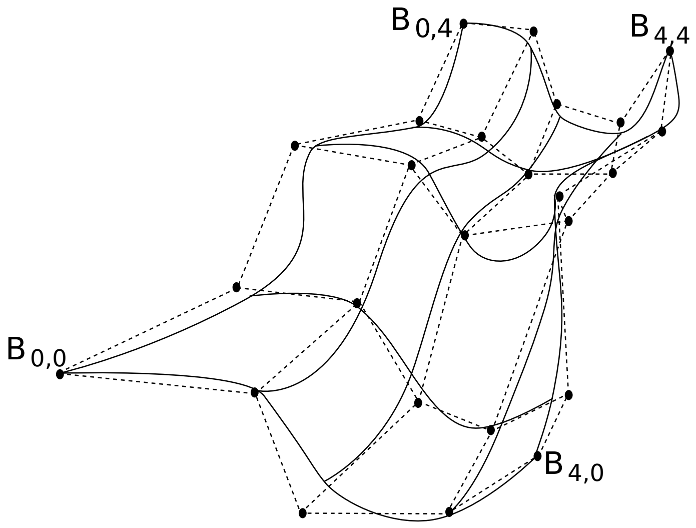


Figure 1.10: Bi-quadratic NURBS surface

$p + n + 1$ and $q + m + 1$ elements, respectively:

$$\mathrm{U} = \{\underbrace{0, \cdots, 0}_{\text{p+1}}, x_{p+1}, \cdots, x_{r-p-1}, \underbrace{1, \cdots, 1}_{\text{p+1}}\} \tag{1.19}$$

$$\mathrm{V} = \{\underbrace{0, \cdots, 0}_{\text{q+1}}, y_{q+1}, \cdots, y_{s-q-1}, \underbrace{1, \cdots, 1}_{\text{q+1}}\} \tag{1.20}$$

where $r = n + p + 1$ and $s = m + q + 1$

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

$$N_{i,p}(u) = \frac{u - x_i}{x_{i+p} - x_i} N_{i,p-1}(u) + \frac{x_{i+p+1} - u}{x_{i+p+1} - x_{i+1}} N_{i+1,p-1}(u) \tag{1.21}$$

with

$$N_{i,0}(u) = \begin{cases} 1 & \textbf{if} \quad x_i \leq u < x_{i+1} \\ 0 & \textbf{otherwise} \end{cases} \tag{1.22}$$

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

Figure 1.10 shows a bi-quadratic NURBS surface and its control net, where $B = \{B_{0,0}, \cdots, B_{0,4}, \cdots, B_{4,0}, \cdots, B_{4,4}\}$, $U = V = \{0, 0, 0, \frac{1}{3}, \frac{2}{3}, 1, 1, 1\}$ and weights equal to 1.

The important properties of the functions $N_{i,j}(u,v)$ are summarized as follows:

1. Non-negativity: $N_{i,j}(u,v) \geq 0 \; \forall \; i, \; j, \; u \; and \; v.$
2. Partition of unity:
$$\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,j}(u,v) = 1 \; \forall \; (u,v) \; \in [0,1] \times [0,1] \tag{1.23}$$
3. Local support: $N_{i,j}(u,v) = 0$ if (u,v) is outside the rectangle given by $[x_i, x_{i+p+1}) \times [y_j, y_{j+q+1})$.
4. In any given rectangle of the form $[x_{i_0}, x_{i_{0+1}}) \times [y_{j_0}, y_{j_{0+1}})$, at most $(p+1)(q+1)$ basis functions are non-zero, in particular the $N_{l,m}(u,v)$ for $i_0 - p \leq l \leq i_0$ and $j_0 - q \leq k \leq j_0$ are non-zero.
5. Extreme: if $p > 0$ and $q > 0$, then $N_{i,j}(u,v)$ attains exactly one maximum value.
6. Differentiability: interior to the rectangles formed by the u and v knot lines, all partial derivatives of $N_{i,j}(u,v)$ exist. At a u knot (v knot) it is $p-k$ $(q-k)$ times differentiable in th u (v) direction, where k is the multiplicity of the knot.

Following properties summarize the geometric characteristics of NURBS surfaces:

1. The control net forms a piecewise planar approximation to the surfaces; as is the case for curves, the lower the degree the better the approximation.
2. Curve degree in parametric direction u, v (p,q), number of control points in each direction $(n+1, m+1)$ and number of knots $(r+1, s+1)$ are related by:
$$r = n + p + 1 \qquad s = m + q + 1 \tag{1.24}$$
3. Corner point interpolation: $S(0,0) = B_{0,0}$, $S(1,0) = B_{n,0}$, $S(0,1) = B_{0,m}$ and $S(1,1) = B_{n,m}$.

4. Affine invariance: an affine transformation is applied to the surface by applying it to the control points.

5. Strong convex hull property: assume $w_{i,j} \geq 0 \ \forall \ i, j$. If $(u, v) \ \in \ [x_{i_0}, x_{i_0+1}) \times [y_{j_0}, y_{j_0+1})$, then $S(u, v)$ is in the convex hull of the control points $B_{i,j}$,$i_0 - p \leq i \leq i_0$ and $j_0 - q \leq j \leq j_0$.

6. Local modification: if $B_{i,j}$ is moved, or $w_{i,j}$ is changed, it affects the surface shape only in the rectangle $[x_i, x_{i+p+1}) \times [y_j, yj + q + 1)$.

7. Non-rational B-spline and Bézier and rational Bézier surfaces are special case of NURBS surfaces. The same example of cubic curves shown in Figure 1.9 can be extended to cubic surfaces.

8. Differentiability: $S(u, v)$ is $p - k$ $(q - k)$ times differentiable with respect to u (v) at a u knot (v knot) of multiplicity k.

It should be noted that there is no known variation diminishing property for NURBS surfaces [80].

Note that the computation of the p-degree basis functions detailed in Equation 1.21 is a recursion relation of a degree p, which depends on lower-order basis functions down to order 1. These basis functions generate a truncated triangle basis table as illustrated in Figure 1.11. This pattern is based on a list of important properties of the NURBS basis functions, which determine the many desirable geometric characteristics of NURBS curves and surfaces. Specifically, two different features of the NURBS basis functions should be considered in order to analyze the recursivity of this basis function: the dependence of a basis function (see Figure 1.12) and its influence (see Figure 1.13).

According to the local support property, $N_{i,j} = 0$ if u is outside the knot interval $[u_i, u_{i+p+1})$. Hence, a basis function $N_{i,p}$ of degree p is computed as a combination of $N_{i,p-1}$, $N_{i+1,p-1}$, two basis functions of $p-1$-degree, and subsequently, this recursion is repeated until the order 1 is reached. Therefore, $N_{i,p}$ is a combination of $p + 1$ basis functions of order 1, $\{N_{i,0}, \ N_{i+1,0}, \ \cdots \ N_{i+p,0}\}$ (see Figure 1.12).

On the other hand, in any given knot span $[u_i, u_{i+1})$ the only non-zero zeroth degree function is $N_{i,0}$. Consequently, in the same given knot span $[u_i, u_{i+1})$ at most

$p+1$ of the basis functions are non-zero, namely the functions $N_{i-p,p} \cdots N_{i,p}$, as illustrated in Figure 1.13.

1.3. Thesis structure

The real-time tessellation of parametric surfaces is the mainstay of this thesis. As parametric surfaces, and specifically NURBS surfaces, are fairly flexible and compact, they are considered a standard in graphic design for the rendering of complex models. In this dissertation, different proposals for tessellating parametric surfaces based on different hardware capabilities are presented. As parametric surfaces, specifically Bézier surfaces, are a new trend as tessellation primitives in computer graphics, the initial chapters of this thesis focused on them. Another parametric primitive, NURBS surface is also considered. NURBS curves and NURBS surfaces are genuine generalizations of non-rational B-spline forms as well as rational and non-rational Bézier curves and surfaces. As more complex models can be codified in a NURBS representation, they are the standard in computer aided design.

Bézier surfaces are the more common examples of parametric surfaces in graphics rendering. In recent years, they have become new trendy primitives owing to their flexibility and simplicity. Hence, current graphics pipeline has been specifically designed for tessellating Bézier surfaces. However, this dissertation analyzes the rendering of Bézier surfaces in simpler pipelines, even with high performance. Hence, different techniques for evaluating Bézier surfaces in real time, exploiting different hardware features, are described (see Chapters 2, 3, 4 and 5). The present thesis goes an step further and also suggests the design of adaptive hardware units for tessellating Bézier surfaces.

Chapter 2 introduces two non-adaptive tessellation proposals for Bézier surfaces: VST and GST. Both proposals synthesize the Bézier models on the GPU. VST tessellates Bézier surfaces in the vertex shader as well as minimizing the CPU-GPU transfers. VST tessellation is guided by a pre-computed parametric grid of virtual vertices and it optimizes the GPU memory accesses to increase data locality. However, GST generates the parametric grid in the geometry shader on the fly, requiring neither pre-computation nor storage of predefined grids. Furthermore, the

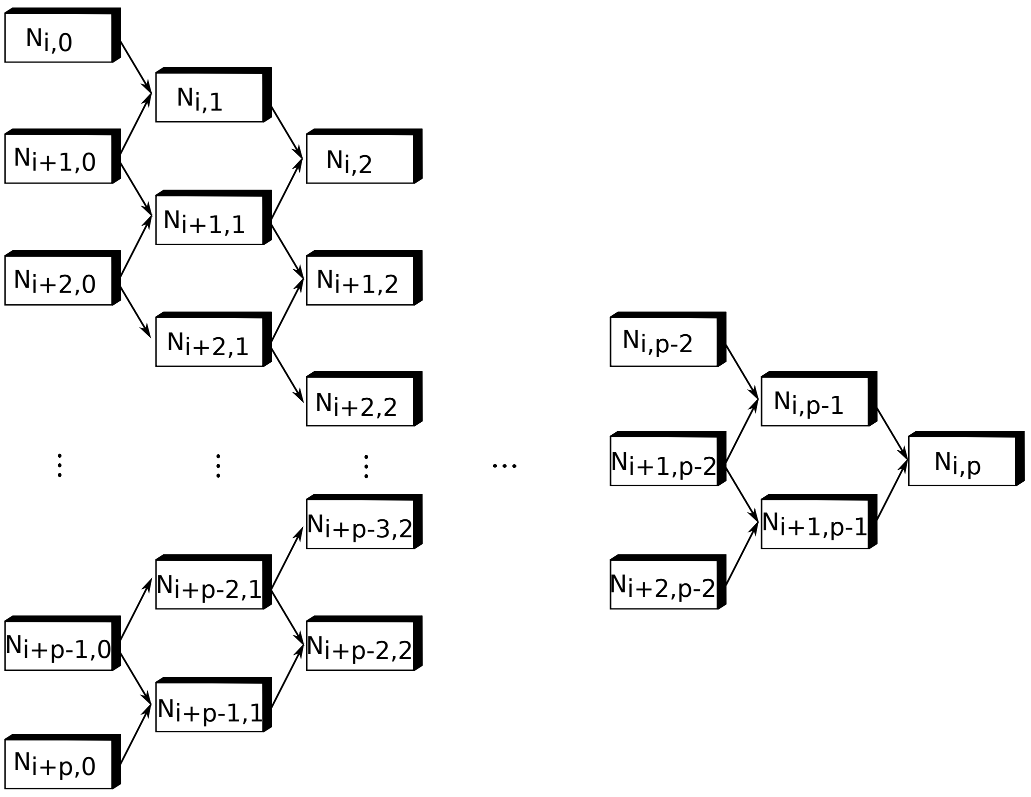


Figure 1.11: Pattern to compute $N_{i,p}$

GPU memory is not a limiting factor in model tessellation. This work has been published in [20], [21], [22] and [23].

Chapter 3 focuses on the adaptive tessellation of Bézier surfaces. Adaptive and Dynamic Mesh Refinement (ADMR) as well as Dynamic and Adaptive Bézier Tessellation (DABT) are summarized; there is an in-depth analysis of geometry tessellation and geometry adjacency capabilities are detailed. ADMR is based on three mainstays: firstly the tessellation is guided by a tessellation pattern according to a local test, with no pre-computed pattern; secondly, inserted vertices are organized in strips; and, finally, the triangle mesh is generated by linking inserted vertices on the fly. Nevertheless, DABT extends ADMR to the tessellation of parametric surface and it introduces a higher adaptively. DABT does not just have one fixed tessellation level across the whole patch, but three: one for each triangle side. Hence, DABT reduces the number of processed triangles without reducing the quality of the final rendering. Furthermore, three different tests have been considered to guide

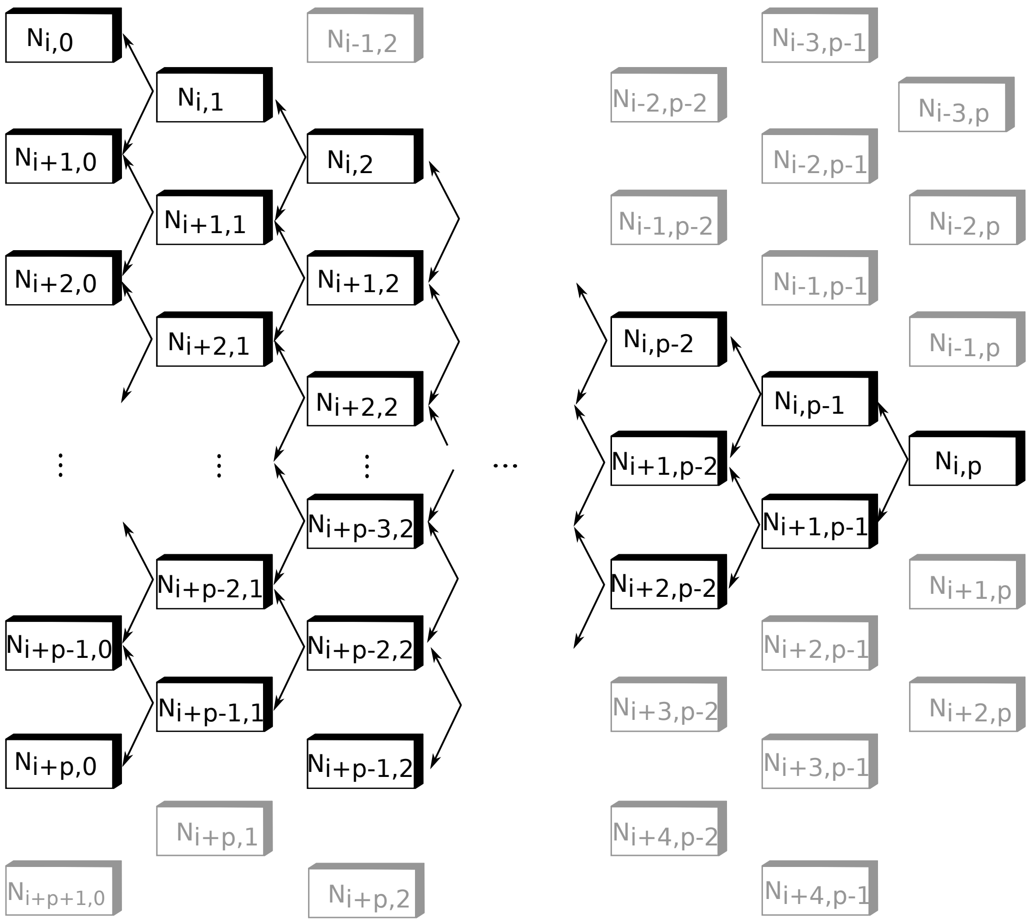


Figure 1.12: Dependence of $N_{i,p}$

this tessellation process: distance between mesh and ideal surface, flatness of the mesh and size of the triangles. This work has been published in [10], [24] and [25].

Chapter 4 details a semi adaptive proposal based on Chapter 2 and Chapter 3. Regarding adaptive tessellation, the semi adaptive approach is midway between the non-adaptive strategy and the fully adaptive strategy. The semi adaptive strategy is characterized by a regular grid pattern as well as a level of resolution per triangle. In this tessellation procedure, local tests evaluate the candidate positions located in the edges of the coarse triangle. Finally, a reconstruction of generated triangles links vertices in two consecutive rows. This work has been submitted to [28].

Chapter 5 describes a new method for rendering Bézier surfaces on the GPU

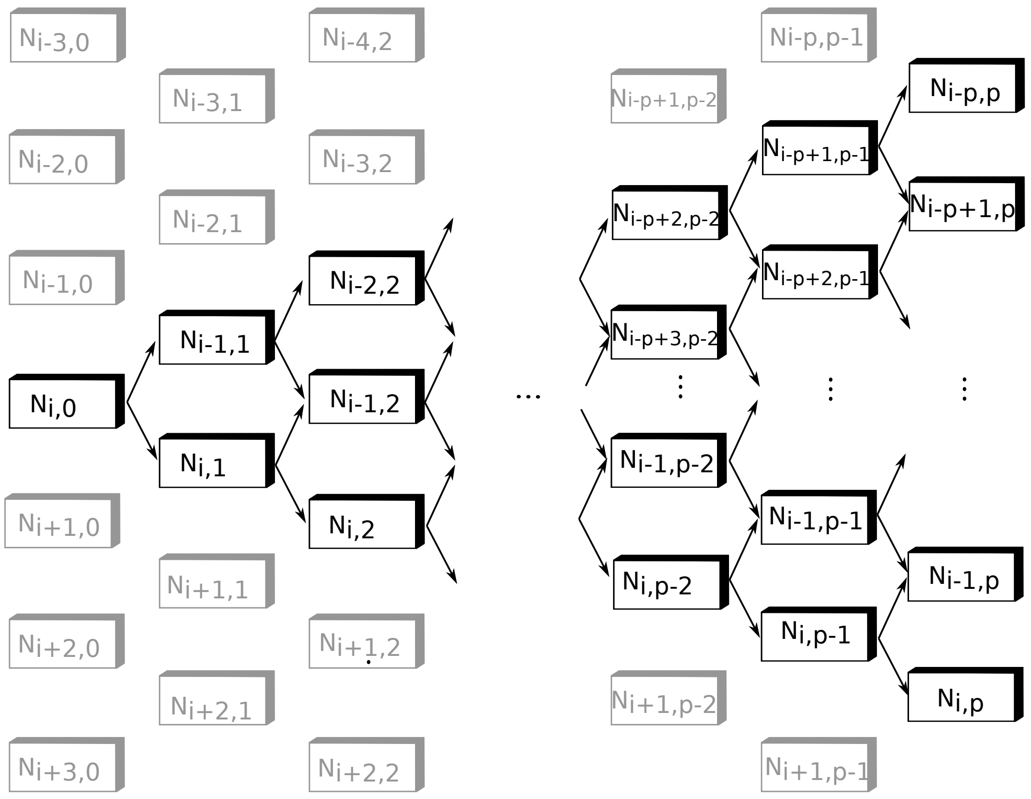


Figure 1.13: Influence of $N_{i,0}$

of handheld devices. This proposal is based on VST detailed in Chapter 2. No other Bézier proposals detailed in this thesis can be selected as the baseline, since they are all implemented in GPU pipelines that are more complex than the one implemented in modern handheld devices. As this proposal has been designed for handheld devices, hardware constraints were considered. In fact, Bézier evaluation and rendering have been designed as a benchmark for testing hardware features. This work has been published in [83] and submitted to [27].

Finally, another type of parametric surfaces, NURBS surfaces, are considered as they are the standard in CAD/CAM applications. NURBS are more complex than Bézier surfaces, thus NURBS surfaces are habitually converted into Bézier surfaces before the rendering process. As parametric surfaces are the vogue, they have been considered in the design of the latest generation of GPUs. Therefore, current GPUs allow the tessellation and direct evaluation of bicubic Bézier surface on the fly, however, they are not flexible enough for the tessellation and evaluation of NURBS.

Chapter 6 introduces a new proposal for tessellating and evaluating NURBS surfaces on the GPU with no previous computation. The rendering Pipeline for NURBS Surfaces (RPNS) is based on the geometric characteristics of NURBS surfaces and it also details the entire process through the GPU pipeline in order to obtain real-time rendering on the GPU. RPNS proposes a new primitive, KSQuad which maintains the main geometric properties of NURBS surfaces. The adaptive discretization of KSQuad and the stair strategy to explicit evaluation are the key points of RPNS. Culling and sampling processes have also been considered. This work has been submitted to [26].

Chapter 2

Non-Adaptive Tessellation

Bézier representations have been widely employed as a standard way of designing complex scenes with high-quality results. In many applications involving CAD/-CAM, virtual reality, animation and visualization, object models are described in terms of Bézier surfaces. The excellent mathematical and algorithmic properties [79, 82], combined with successful industrial applications, have contributed to the popularity of this representation. One of the main advantages of this representation is its compactness and, as a consequence, the low storage and transmission requirements of the resulting models. Additionally, graphics designers can produce animations in a simpler and faster way as fewer points need to be controlled than for triangle meshes. On the other hand, these representations are easily scalable so a surface can be converted into a triangle mesh with few or many triangles, depending on the required *level of detail* (LOD).

There are currently two main approaches to the synthesis of parametric surfaces: tessellation on the CPU or on the GPU. In the former, these representations are tessellated into triangles in the CPU before being sent to GPU to be displayed. This strategy presents a number of disadvantages that could reduce the system performance: the amount of information to be sent from CPU to GPU and the increment in the storage requirements in the GPU associated with the triangle mesh. The second approach performs the tessellation of the parametric models directly on the GPU [21, 33, 47]. In these proposals the tessellation level is selected per patch [21, 47] or per set of patches [33].

This chapter describes a non-adaptive proposal for tessellating Bézier surfaces into high-quality meshes which accurately represent complex surfaces and contain no artefacts, such as T-junctions or cracks. Furthermore, in order to achieve real-time rendering, the tessellation is performed on the GPU and two alternatives have been designed to tweak many specific details for optimal performance, depending on the GPU architecture. The VST (Vertex Shader Tessellation) alternative has been designed for a GPU without primitive generator, such as GPUs implemented in handheld devices. VST consists in using a parametric map of virtual vertices [11, 47] with an efficient exploitation of the information stored on the GPU. More specifically, a technique that permits the optimization of the memory usage of the GPU to increase the data locality exploitation is proposed. This strategy allows the minimization of draw calls and the CPU-GPU communications. Nevertheless, our second alternative, GST (Geometry Shader Tessellation) has been designed based on the capabilities of a primitive generator on the GPU. Therefore, it is based on the on-the-fly generation of the parametric grid; thus, as the tessellation can be executed on-the-fly, it avoids the pre-computation and storage of predefined grids in the local memory. Therefore, the GPU memory does not limit the level of resolution per surface. Both designs have being tested under different GPU platforms and good results in terms of quality and timing requirements have been obtained for both. As result of our analysis, we conclude that the adequate exploitation of the GPU capabilities is close to permitting the real-time rendering of parametric models, even for very complex scenes.

This work has been published in [20], [21], [22] and [23].

2.1. Structure of the Non-Adaptive tessellation proposal

In this chapter we present our proposal for the evaluation of Bézier surfaces on the GPU based on the exploitation of spatial coherence of data within each surface. This proposal considers each Bézier surface as the input primitive to the pipeline; thus, Bézier surfaces are tessellated and evaluated on the GPU instead of the evaluation of independent samples. Furthermore, we provide a computationally complex but efficient shader which exploits the computational power of current GPUs as well as optimized memory access.

The representation of a Bézier surface $Q(u,v),\ 0 \leq u,v \leq 1$ (see Equation 1.6) is based on the utilization of two parametric values defined in a normalized interval $[0,1]$. In our non-adaptive proposal, the tessellation is performed on the GPU and this implies the evaluation of the surface equation $Q(u,v)$ for different parametric values (u,v) (see Figure 2.1). The resulting points are vertices that are connected in order to build the triangles of the final mesh. For the sake of clarity we work with a simple algorithm that performs a uniform subdivision of the parametric space in the two dimensions. More specifically and for a tessellation level l, 2^{l+1} parametric values in each dimension are considered. The grid of parametric values P^l to be evaluated are:

$$P^l = \begin{bmatrix} (u_1, v_1) & \cdots & (u_1, v_{2^{l+1}}) \\ (u_2, v_1) & \cdots & (u_2, v_{2^{l+1}}) \\ \vdots & \ddots & \vdots \\ (u_{2^{l+1}}, v_1) & \cdots & (u_{2^{l+1}}, v_{2^{l+1}}) \end{bmatrix} \tag{2.1}$$

where

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

with $i \in \{1, \cdots, 2^{l+1}\}$.

For a resolution level l, the grid of parametric values to be evaluated P^l is made up of $2^{l+1} \times 2^{l+1}$ samples (see Figure 2.2). The resolution level to be applied to each Bézier surface is selected by the application taking into account different factors such as screen space error, model complexity or computational requirements. Taking this into account, a system of L grids of parametric values for the different resolution levels $\{P^1, P^2, \cdots, P^L\}$ can be computed a priori, L being the highest resolution level.

Here we should stress that our proposal takes advantage of the constant result of $[N][B][M]^T$ for every point in the surface and that every control point $[B]$ is only accessed once, transforming the Equation 1.7 into

$$Q(u,v) = [U][A][V] \quad 0 \leq u,v \leq 1 \tag{2.2}$$

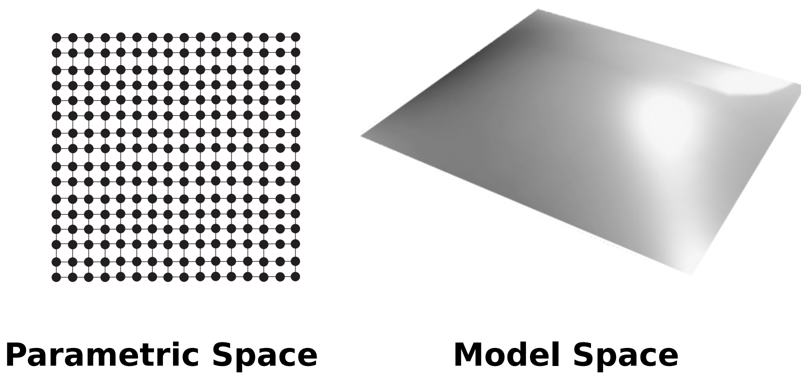


Figure 2.1: Parametric space and model space for a Bézier surface

with $[A] = [N][B][M]^T$. Note that while in our proposal the [A] matrix is computed only once per surface, the most advanced tessellation units recently included with DirectX11 would evaluate the [B] matrix once per sample. Features and performance of this GPU pipeline will be analyzed in the following chapters.

Our proposal exploits two different alternatives for the tessellation. The first is based on the exploitation of the vertex shader (VST, Vertex Shader Tessellation see Section 2.2). In this case, and due to the impossibility of generating geometries, the utilization of techniques based on virtual vertices [11, 47] is the key for a multi-resolution application. This idea is based on the pre-computation of a set of parametric maps on the CPU. The selection and evaluation of the final vertices, according to the resolution level selected, is performed on the vertex shader. This alternative allows the information stored in the GPU to be used and exploited efficiently. The geometry shader is not required in this strategy.

The second alternative is based on the generation of primitives on the GPU (GST, Geometry Shader Tessellation see Section 2.3). In this case, the surface tessellation is performed on the geometry shader. The resolution level can be selected on-the-fly and the generated geometry can be fed back to the standard pipeline through the stream-out unit. This alternative reduces storage requirements and geometry capabilities for primitive generation are exploited. Despite the limitations and drawbacks of the geometry shader unit, promising results have been obtained with our application and tests performed.

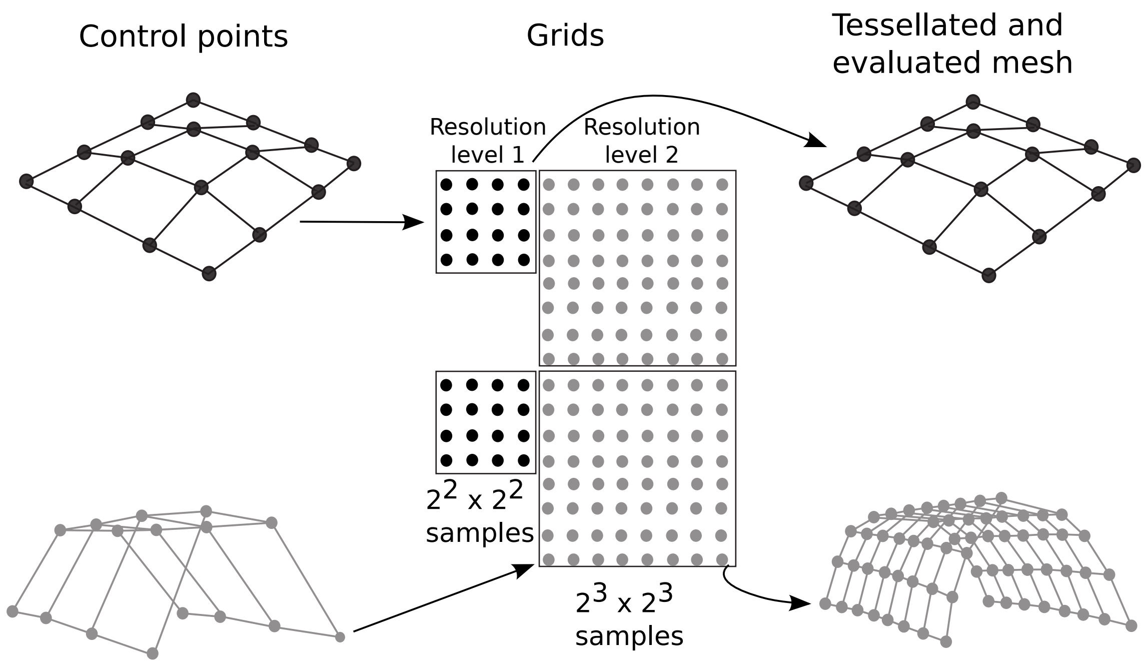


Figure 2.2: Samples evaluated to resolution level 1 and 2

2.2. Vertex Shader Tessellation (VST)

VST is a tuning alternative that optimizes the utilization of GPU resources and a series of performance features. In this alternative, L-grids $\{P^1, P^2, \cdots, P^L\}$ are computed and stored in the GPU to be selected and employed for the different surfaces of the model. Taking into account that this alternative is based on the storage of pre-computed information, an analysis of the storage requirements and the consequent implications should be performed. Specifically, the memory requirements for the application are:

$$M = \sum_{l=1}^{L} M_{P^l} + M_{[B^s]} \times N_S$$

where M_{P^l} is the memory requirements for the grid, P^l, of resolution level l and N_S is the number of surfaces in the scene. $M_{[B^s]}$ includes the amount of memory used for the control points of each surface. For a (n, n)-degree surface this amount is:

$$M_{[B^s]} = 3 \times (n+1) \times (n+1)$$

However, the utilization of a single system of grids limits the speed of the application. If a unique system of grids stored in memory is accessed by all surfaces in the scene, a sequential procedure is forced. This means that for each frame there are as many *Draw Primitive* calls as surfaces N_S, so the performance decreases owing to the amount of calls. Therefore, the amount of synchronizations, N_{DP}, per frame is $N_{DP} = N_S$. As only one surface is computed per *Draw Primitive* call, GPU parallelism is not exploited. Additionally, a large amount of synchronous calls adversely affects performance as a *Draw Primitive* is a slow operation.

Therefore, VST uses several copies of the system of grids of parametric values to process more surfaces per draw call; i.e. several copies of $\{P^1, P^2, \cdots, P^L\}$ are used. By way of an example, Figure 2.2 shows the evaluation of two input Bézier surfaces that are tessellated with a different resolution level. The utilization of different copies of the grid systems permits the simultaneous evaluation of the two surfaces, with the consequent increment in the processing speed.

To evaluate the number of surfaces that can be processed per *Draw Primitive* call the storage requirements of the application have to be evaluated. In our application, and due to the global memory latency, the control points $[B^s]$ of the surfaces are stored in the texture memory. This memory is cached, so if there is a cache miss, the information is obtained from global memory, with the consequent delay. The desirable framework is storing all surface control points in the texture memory and performing one single draw call. But when the storage requirements exceed the texture cache capacity, the performance decreases due to the latency of the global memory in each cache miss. Taking this into account, we have developed a technique whereby the number of draw calls is selected as a trade-off between two objectives: minimizing the number of draw calls and assuring that the storage requirements per draw call do not exceed the texture memory capabilities. As a result, VST performs N_{DP} draw calls, processing and rendering N_d surfaces per call:

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

with $1 \leq N_d \leq N_S$. Thus, the required amount of memory is

$$M = \sum_{l=1}^{L} M_{P^l} \times N_d + M_{[B^s]} \times N_S$$

In detail, and to obtain an optimum application in terms of speed, the following transmission and storage requirements have to be verified:

1. The data transfer between CPU and GPU has to be minimized. In VST the information required (parametric grids and control points of the surfaces) is sent once to the GPU. The information is efficiently stored and re-employed for optimum performance.

2. The storage requirements associated with the grids of parametric values should not exceed the global memory capabilities. Taking into account that the global memory is used for more purposes, not all the space is available for the grids storage, only a fraction thereof. Specifically, in our application the grids are stored in a vertex buffer, but exceeding the recommended capabilities would result in limitations for other utilizations and could affect resource swapping. As result the following condition has to be verified:

$$\sum_{l=1}^{L} M_{P^l} \times N_d < per \cdot M_{GPU}$$

   M_{GPU} being the GPU global memory size and *per* a percentage value that depends on each GPU.

3. The storage of the control points associated to the Bézier surfaces to be processed per draw call should not exceed the capabilities of the texture memory; i.e., $M_{[B^s]} \times N_d < M_T$ being M_T the texture memory size. Therefore, the cache properties of the texture memory are efficiently exploited and the data loaded to this memory are fully processed for all surfaces to be rendered in a draw call before being replaced.

4. The number of draw calls (N_{DP}) should be minimized owing to their fixed-cost overhead [2]. The basic idea of our *batching* strategy is to combine many small transfers into one large one to optimize the data communication procedure.

The analysis of the storage requirements, along with the recommended number of draw calls according to our tests, is included in the results section.

2.2.1. Implementation Details

In this subsection, we summarize the details of our VST kernel implementation. The kernel processes bicubic Bézier surfaces and exploits the capabilities of the DirectX 10 Microsoft's HLSL [66].

The VST algorithm is shown in Figure 5.10 and it consists of two stages: *Preprocessing Stage* and *Synthesis Stage*. In the *Preprocessing Stage* two tasks are carried out: the control points (Task 1) and the grids of virtual vertices P^l, $1 \leq l \leq L$ (Task 2) are sent from CPU to GPU. In the *Synthesis Stage* the LOD per surface is selected (Task 3) and the Bézier surfaces is tessellated (Task 4) and rendered on the GPU. While the *Pre-processing Stage* is processed once per scene, the *Synthesis Stage* can be performed multiple times per data set for successive frames f (see feedback in the figure). The tasks to be carried out and their optimizations are explained below.

In Task 1, the Bézier surfaces control points $[B^s]$, $1 \leq s \leq N_S$ are sent from CPU to GPU. Unlike previous proposals [11, 47], in our application the information is sent only once from CPU to GPU and stored in the texture memory. This improvement, a key point in our proposal, is possible thanks to the improved storage capabilities of current graphics cards. As a consequence, VST reduces CPU-GPU communication. The control points of the surface $[B^s]$ are stored in three float 4×4 arrays $[B^s_x, B^s_y, B^s_z]$, one per coordinate. As texture memory provides better performance that constant or global memory, these data are stored in a texture buffer (*tbuffer*) [66]. Therefore, the texture memory has less restrictive access patterns and it hides memory latency accesses. Furthermore, *tbuffer* allows the simultaneous access to different variables packed in the same buffer, thus improving performance.

$\sum_{l=1}^{L} M_{P^l} \times N_d$ virtual vertices grids are sent to GPU and stored in the *Vertex Buffer* [66] in Task 2; thus, the vertex buffer accelerates the synthesis of geometries. As geometries are stored in the vertex shader following a specific pattern which contains connectivity information, no *Index Buffer* is required.

During the synthesis stage and for each frame N_{DP} draw calls are performed.

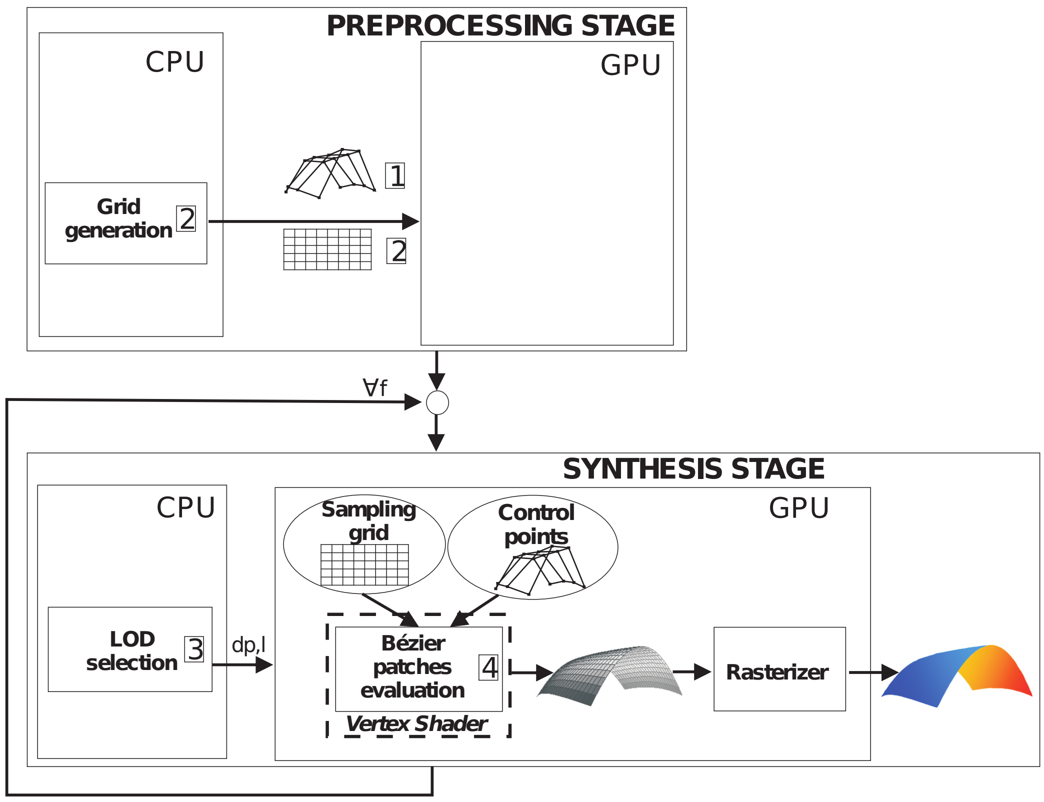


Figure 2.3: Structure of the VST algorithm

In each call N_d surfaces are selected to be tessellated and rendered. Following a similar strategy to previous proposals [33, 47], N_d surfaces grouped in the same draw call are represented with the same level of detail l. As a consequence, the mesh connectivity can be directly extracted from the systems of grids and no *Index Buffer* is required, with the consequent advantages in terms of time and storage requirements. Note that the classification into groups of N_d surfaces could imply small adjustments of the real levels of details in some surfaces. However, small variations of the levels of detail are not noticeable, while a considerable reduction in the timing requirements is achieved with this simplification. In summary and according to the surfaces selection strategy in terms of their level of detail, Task 3 sends the level of detail l and an index DP which identify the set of surfaces to be rendered in that draw call.

Finally, Task 4 is executed in the vertex shader of the GPU and Equation 2.2

```
1  VS_OUTPUT DefaultVS(VS_INPUT $P^l$)
2  {
3    float4x4 [N]= { -1, 3, -3, 1,
4                     3, -6, 3, 0,
5                    -3,  3, 0, 0,
6                     3,  0, 0, 0, }
7    $u = P^l$.x; $v = P^l$.y;
8    $s = P^l.z \times dp \times N_d$;
9    float1x4 [U]=($u^3$, $u^2$, u, 1);
10   float1x4 [V]=($v^3$, $v^2$, v, 1);
11   float4x4 {[$B^s_x$],[$B^s_y$],[$B^s_z$]} = read from texture ($s$);
12   float3 vertex = mul([U], [N],[$B^s$], [N], [V]);
13   return vertex;
14 }
```

Figure 2.4: vertex shader pseudocode for the VST method

is evaluated. Due to its efficiency in a GPU implementation, VST employs a direct evaluation strategy instead of the de Casteljau algorithm [88].

Figure 2.4 shows the simple vertex shader pseudocode of VST for the bicubic surface evaluation. The input parameters of the vertex shader (line 1) are the grid parametric values P^l, which will be employed in the evaluation. Note that the N_d surfaces have the same level of detail l and the same group identification indexed by DP. The (u, v) parametric values are stored in P^l coordinates x and y while the z coordinate stores a surface index $\{0, \cdots, N_d - 1\}$. In consequence, the identification of each surface can be performed directly (line 8). To evaluate Equation 1.7 (line 12) $[U]$ and $[V]$ are calculated (lines 9 and 10), the control points of the surface are read from the texture memory (line 11), and the basis function coefficients (lines 3 to 6) are employed. As a result, the vertices of the final tessellated mesh are obtained (line 12). As will be shown in the results section, the simplicity of the strategy and the efficient management of the information storage are key points for the real-time rendering of high-quality models.

2.3. Geometry Shader Tessellation (GST)

In this section, we include our second alternative. This is based on the exploitation of the geometry shader for the Bézier surface tessellation (GST). The objective is to exploit the geometry shader capabilities for geometry generation. This, in contrast to the VST, permits the generation of geometry without the need to use a virtual vertex strategy. As a result, the storage requirements are reduced and the new capabilities of current graphics cards are exploited.

The key idea of GST is the on-the-fly computation of the P^l values for each input surface. As a consequence, no pre-computed grids are employed and the storage requirements are reduced since only the control points of the surfaces are required. Although the possibilities of the geometry shader as a geometry generator are promising, current implementations still have strong limitations [2]. Specifically, current versions permit the generation of 1024 32-bit elements per input primitive. In our implementation, this limits the number of triangles to be generated per Bézier surface and, in consequence, the maximum resolution level to be generated. Specifically, the maximum resolution level allowed is $l = 3$; i.e., $2^4 \times 2^4$ triangles can be generated. In future graphics cards we expect this limitation to be reduced or eliminated, with the consequent benefits for our technique.

The method currently employed to obtain a higher level of detail is an iterative execution of the geometry shader for each surface. GST is based on the geometry shader output which can be stored in output stream and feedback as input for the rendering pipeline (see Figure 2.5). However, the inherent timing costs of iterative procedures render the reduction in the number of iterations relevant. The mainstay of GST is to reduce this number of iterations using an efficient method to increase the highest level of detail that can be managed per iteration.

The key idea of GST for increasing the resolution level is partitioning the parametric map into zones and the parallel evaluation of these zones on the geometry shader. That is, the P^l grid (see Equation 2.1) with $2^{l+1} \times 2^{l+1}$ parametric values is partitioned and the corresponding parametric values groups processed in parallel in the geometry shader. Considering groups of $m \times m$ parametric values the P^l matrix of values can be rewritten as a system of sub-matrices:

$$P^l = \begin{bmatrix} P^l_{[1,1]} & \cdots & P^l_{[1,Nz^v]} \\ \vdots & \ddots & \vdots \\ P^l_{[Nz^u,1]} & \cdots & P^l_{[Nz^u,Nz^v]} \end{bmatrix} \tag{2.3}$$

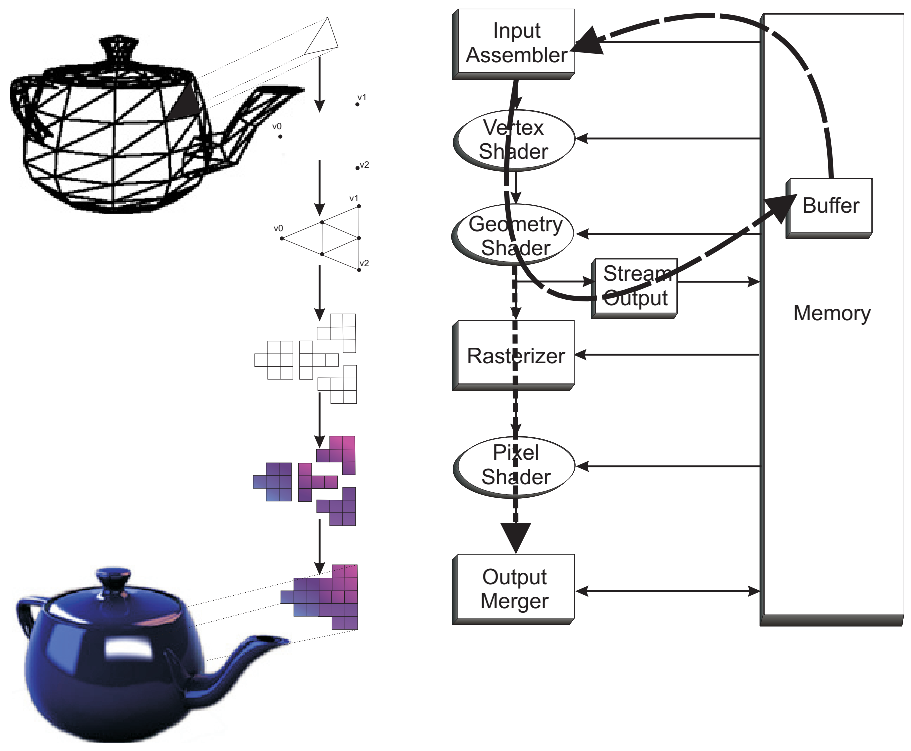


Figure 2.5: DirectX10 pipeline where the iterative procedure through the Stream Output is remarked.

Nz^u and Nz^v are the number of zones in u and v directions, respectively:

$$Nz^u = \frac{2^{l+1}}{m}; \quad Nz^v = \frac{2^{l+1}}{m}$$

The structure of GST is schematically depicted in Figure 2.6. Two geometry shader kernels are devoted to two tasks: zone identification and tessellation per zone.

The first task partitions the parametric grid into zones. As indicated in Equation 2.3 the P^l matrix is generated with a set of sub-matrices $P^l[i,j]$, where $i = 1, \cdots, Nz^u$ and $j = 1, \cdots, Nz^v$. In GST the first shader calculates the parametric map through the identification of the first element of each sub-matrix, $(u_{(i \cdot m)+1},\ v_{(j \cdot m)+1})$. Once this value is calculated, the remaining parametric values can be generated with simple incremental operations. As indicated in the figure, the possibility of an it-

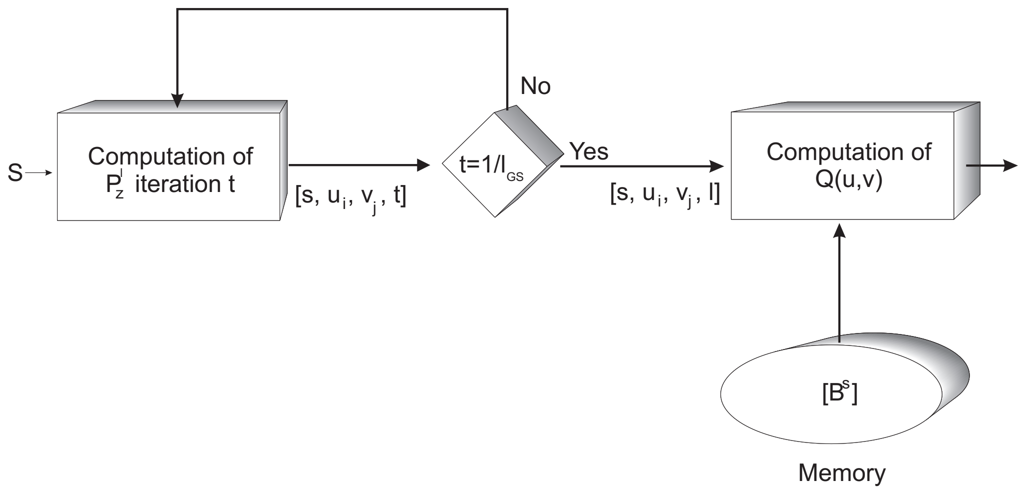


Figure 2.6: Structure of GST proposal

erative process is considered. As a result, the first shader generates four values per zone $[s, u_{(i\cdot m)+1}, v_{(j\cdot m)+1}, t]$, where s is the surface index and t indicates the iteration number. Due to the limitations of the geometry shader (only 1024 32-bit data can be generated per input primitive) only up to $1024/4 = 256$ zones can be processed in each step of the iterative algorithm. In consequence, the resolution level that can be obtained with GST per iteration is $2^l_{GS} \cdot m \times 2^l_{GS} \cdot m$, with $m = 4$, $l_{GS} = 4$.

The second shader performs the surface evaluation (Equation 2.2) for the points assigned to each zone. The zones will be managed by the geometry shader as isolated input primitives, thus the vertices located in the border among zones are evaluated more than once. This means that cracks between contiguous zones can be avoided. Consequently, the matrices are of size $(m+1) \times (m+1)$ with an overlap of elements between matrices with consecutive indices:

$$P^l[i,j] = \begin{bmatrix} (u_{(i\cdot m)+1}, v_{(j\cdot m)+1}) & \cdots & (u_{(i\cdot m)+1}, v_{(j\cdot m)+(m+1)}) \\ \vdots & \ddots & \vdots \\ (u_{(i\cdot m)+(m+1)}, v_{(j\cdot m)+1}) & \cdots & (u_{(i\cdot m)+1}, v_{(j\cdot m)+(m+1)}) \end{bmatrix} \qquad (2.4)$$

An example of the parametric map generation is depicted in Figure 2.7 where

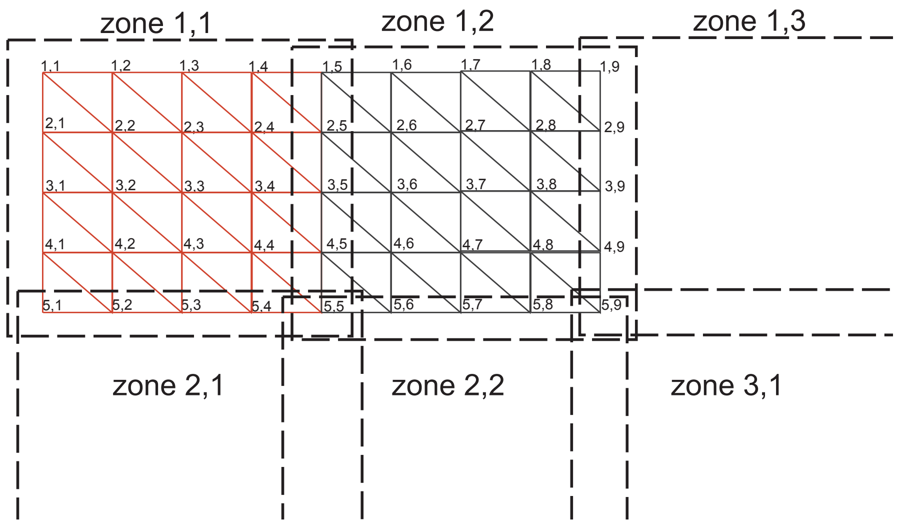


Figure 2.7: Parametric map partitioning in zones

$m = 4$. In this case, each zone has 5×5 elements with an overlap of points between neighbouring zones. For example, zones labeled as (1,1) and (1,2) in the figure share the following points: $\{(u_1, v_5),\ (u_2, v_5),\ (u_3, v_5),\ (u_4, v_5),\ (u_5, v_1),\ (u_5, v_2),\ (u_5, v_3),\ (u_5, v_4)\}$.

Here we should stress that GST uses a regular grid in the (u, v) parametric directions and could exploit the GPU vector computation capabilities by computing sixteen points of the Bézier surface simultaneously, $Q(u_{c+k}, v_{j+k})$ with $0 \leq k < 4$.

2.3.1. Implementation Details

In this subsection, the details of our GST kernel implementation are summarized. The kernel processes bicubic Bézier surfaces and exploits the capabilities of the DirectX 10 Microsoft's HLSL [66]. The structure of the implementation is schematically depicted in Figure 2.8. Similarly to VST, the algorithm has two stages: *Preprocessing Stage* and *Synthesis Stage*. In the *Preprocessing Stage* two tasks are performed: the control points (Task 1) and the surface indices (Task 2) are sent from CPU to GPU. In the *Synthesis Stage*, the LOD per surface is selected (Task 3), the parametric map partitioned into zones (Task 4), and the corresponding tessellated surface sections are evaluated (Task 5) and finally rendered on the GPU. As for

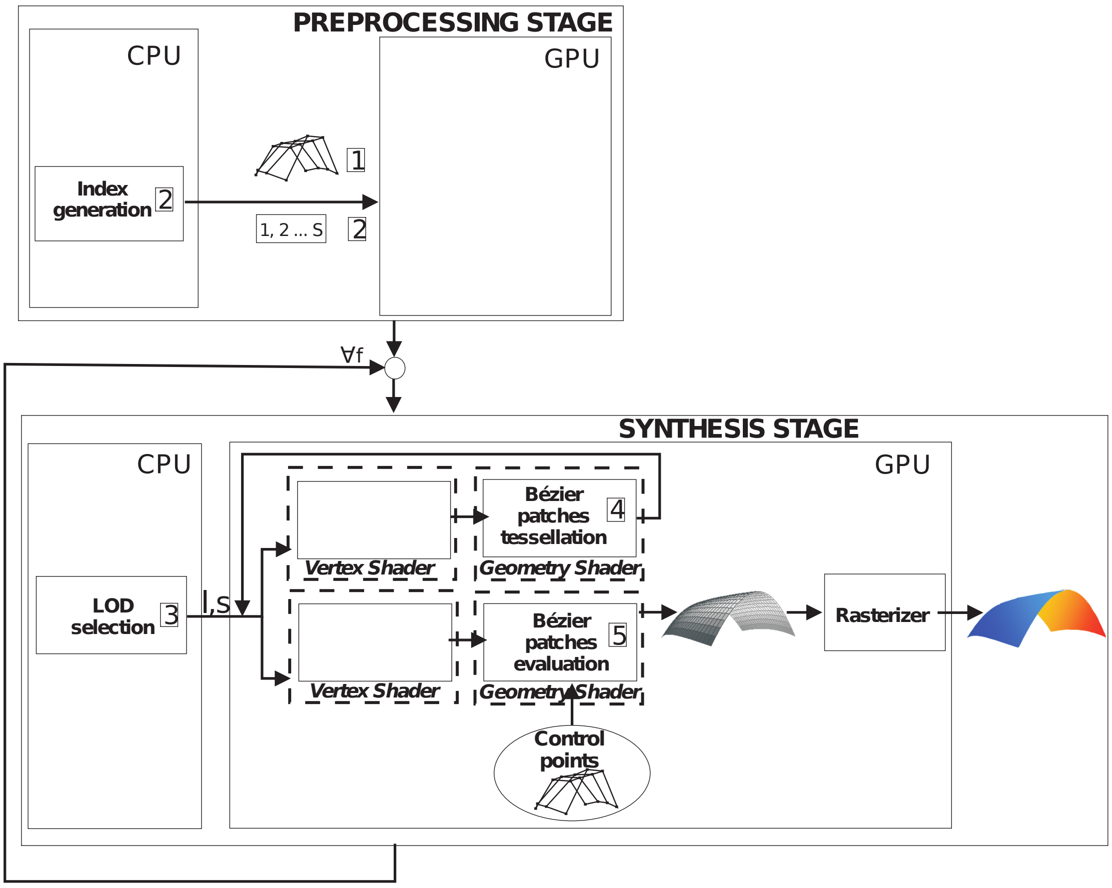


Figure 2.8: Structure of the GST algorithm

the VST algorithm, while the *Preprocessing Stage* is performed once per scene, the *Synthesis Stage* can be performed multiple times per data set for successive frames f. We now to on to explain these tasks and the optimizations thereof.

In Task 1, the control points of the Bézier surfaces $[B^s]$, $1 \leq s \leq N_S$ are sent from CPU to GPU. These control points are stored in the GPU's texture buffer (*tbuffer*) as three arrays $[B^s_x, B^s_y, B^s_z]$.

Task 2 is required to generate the vertex buffer. This simple vertex buffer comprises a surface index which is required by the geometry shader for the surfaces identification. The geometry shader receives the surface indices as input, which is then used to recover the corresponding control points from the texture memory.

During the synthesis stage, each Bézier surface is selected and processed on the geometry shader to generate a triangle mesh according to the desired level of detail specified by the application. Surface index and level of detail are sent from CPU to GPU in Task 3. Hence, this proposal minimizes transmission requirements between CPU and GPU.

The partitioning of the parametric map into zones is associated with Task 4. This simple kernel, to be executed on the geometry shader, generates as output a tuple $[s, u_{(i \cdot m)+1}, v_{(j \cdot m)+1}, t]$; i.e., the surface identifier s, the zone identifier (first parametric coordinate of each submatrix), and an iteration index t. As has previously been indicated, up to 256 tuples can be generated in each iteration of the algorithm. For a higher resolution, an iterative process can be executed (see feedback in the figure). Finally, Task 5 includes the evaluation of the Bézier surfaces for values associated with each zone of the parametric map. As the new vertices are generated using the gometry shader, the memory requirements are minimized. Note that two buffers are required for the passes of the geometry shader to hold the stream-out results [70]. In this manner, the shader can ping-pong between them using an initial buffer for input and a second one for output on odd-numbered passes, and vice versa on even-numbered passes.

As will be shown in next section, the utilization of the geometry shader for the direct tessellation of the Bézier surfaces on the GPU is a promising technique.

2.4. Experimental Results

In this section we present the results of the evaluation of VST and GST. We have implemented both algorithms by exploiting the capabilities of the DirectX 10 Microsoft's HLSL [66]. Comparisons in terms of performance with the algorithm presented in [47], employed as a benchmark test in recent publications, are also included. We have run our implementations on a Intel Core 2 2.4 GHz with 2 GB of RAM and on two different GPUS: GeForce 9800 GTX and ATI Radeon 3870 X2.

VST and GST are evaluated in different scenes that are composed of replicated versions of a small set of models. The models (*Teacup*, *Teapot* and *Elephant*) employed are depicted in Figure 2.9 at different resolution levels. The final images have

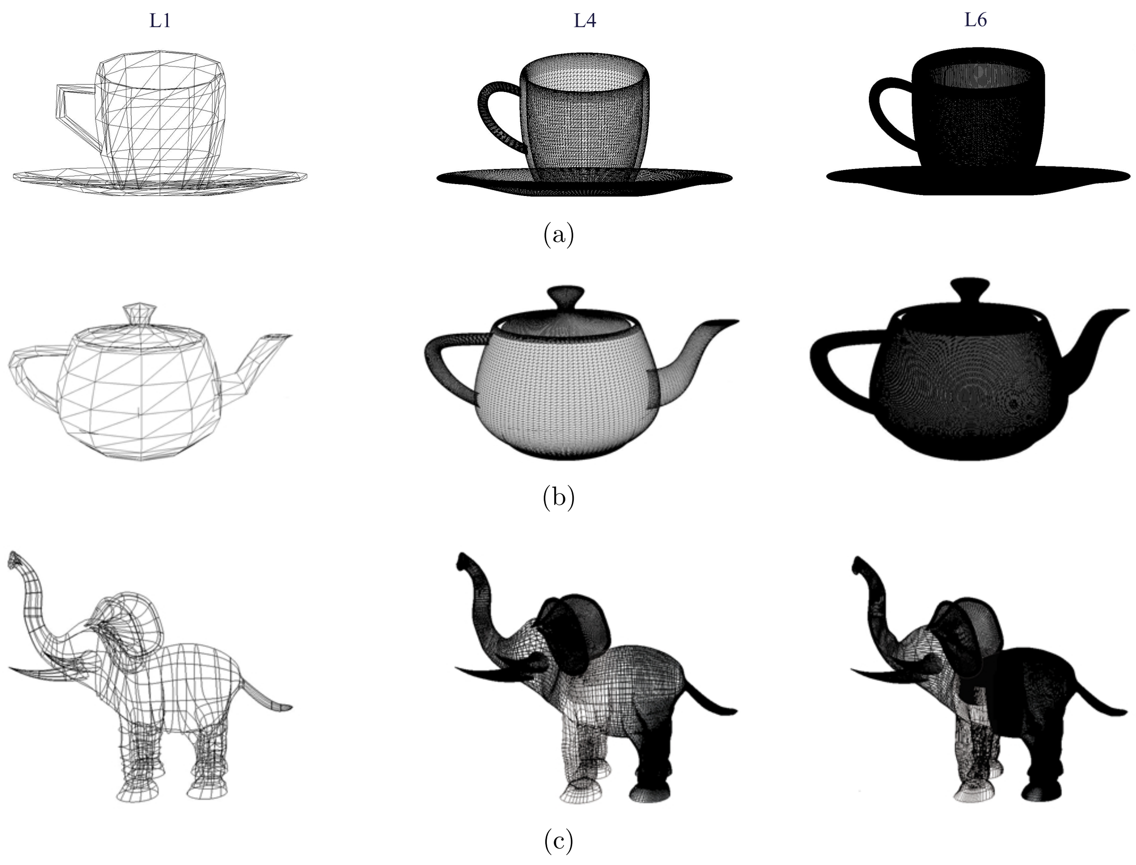


Figure 2.9: Models employed in the test scenes (a) *Teacup* (b) *Teapot* (c) *Elephant*

a screen resolution of 1280×1024 pixels. In Table 2.1 we include the results obtained for 16 of those scenes, denoted as S_i, with $i = 1, \cdots, 16$. Column N_s shows the number of Bézier surfaces while column N_T^1 indcates the number of triangles generated for the coarsest level of detail; i.e., $L = 1$. Columns N_T^4 and N_T^6 show the number of triangles generated with $L = 4$ and $L = 6$ for a non-adaptive tessellation; i.e., all surfaces were tessellated with the same level of detail. Columns N_T^4 Adpt. and N_T^6 Adpt. show the mean number of triangles generated for an adaptive tessellation proposal with $L = 4$ and $L = 6$; i.e., when the resolution level of each surface is up to 4 or 6 respectively. In the adaptive approach, the same LOD is applied to the whole model and it is modified on the fly according to the model position respect to the viewpoint. In this case, the resolution of each surface is selected on the basis of its position in the scene with a varied set of viewpoints. Note that complex scenes

Scene	N_s	N_T^1	N_T^4	N_T^4 Adpt.	N_T^6	N_T^6 Adpt.
S_1	26	0.46	48.80	25.49	819.05	432.66
S_2	32	0.56	60.06	34.39	1008.06	554.98
S_3	260	4.57	488.01	254.86	8190.51	3815.56
S_4	320	5.63	600.62	343.95	10080.63	5206.89
S_5	520	9.14	976.02	509.72	16381.02	6935.26
S_6	640	11.25	1201.25	687.90	20161.25	9454.31
S_7	780	13.71	1465.02	764.59	24571.52	9259.56
S_8	811	14.26	1522.21	1241.23	25548.08	14923.50
S_9	960	16.88	1801.88	1031.84	30241.88	12856.10
S_{10}	1040	18.28	1952.03	1019.45	32762.03	10801.40
S_{11}	1280	22.50	2402.50	1375.79	40322.50	15142.10
S_{12}	1300	22.85	2440.04	1274.31	40952.54	11495.80
S_{13}	1600	28.13	3003.12	1719.74	50403.13	16525.70
S_{14}	2600	45.70	4880.08	2548.62	81905.08	38865.80
S_{15}	3200	56.25	6006.25	3439.48	100806.25	30826.25
S_{16}	8110	142.56	15222.09	12421.30	255480.84	55340.50

Table 2.1: Number of triangles generated (in K) for each scene

with a high number of surfaces were used.

First, and for the VST alternative, the number of draw calls N_{DP} were analyzed. As an example of our analysis, Figure 2.10 shows the frames per second in scene S_5 for different N_{DP} and L values, considering GeForce 9800 GTX. Similar behaviour was obtained for all the scenes tested. As can be observed in the figure, the number N_{DP} has a strong influence on the algorithm's performance. For example, the obtained speedup is 1.42 with $L = 5$ for $N_{DP} = 4$, and up to 1.31 with $L = 6$ for $N_{DP} = 8$. Initially, and for a small number of N_{DP}, the frames per second are increased. The good performance in terms of frames per second is due to the reduction in global memory accesses and the efficient utilization of the texture memory. The good results are associated with the exploitation of data locality and the scheduling strategy employed. For larger N_{DP} values, this trend changes owing to the overhead of each draw call. With respect to the dependence, with the value L, for larger L values the best frames-per-second values are obtained for larger N_{DP} values. According to Wloka's rule [2], this is due to the larger number of polygons per surface and the rasterization costs that become the standard GPU pipeline in

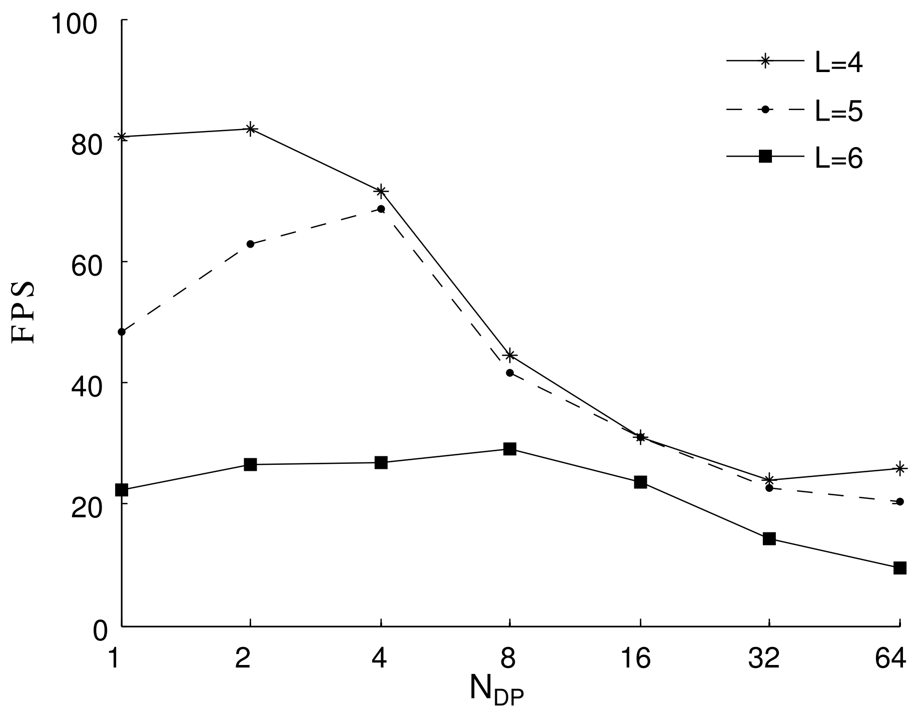


Figure 2.10: VST proposal for S_5 with $L = 4$, $L = 5$ and $L = 6$ in an Nvidia GeForce 9800 GTX

the bottleneck of the application.

With respect to GST, Figure 2.11 shows the influence of the number of iterations in the performance, due to GST partitioning the parametric maps into zones. The influence of the feedback process is analyzed. With $L = 1$ only one geometry shader stage is computed, while $L = 2$ computes two passes through the geometry shader. As can be seen in the figure, the feedback process through the GPU pipeline reduces the performance dramatically. In both, $L = 1$ and $L = 2$, the computational power of the GPU core is underused; however, the performance with $L = 2$ is considerable reduced. This gap in the performance is due to the fact that $L = 2$ iterates two passes through the GPU pipeline and a geometry shader stage is computed twice for each Bézier surface (Task 4 and Task 5 in Figure 2.8). The same trend can be observed for $L = 4$ (see Figures 2.11 and 2.12). However, as both, $L = 2$ and $L = 4$, pass twice through the GPU pipeline (geometry shader is executed twice) there is a smaller gap when comparing $L = 2$ and $L = 4$ than with $L = 1$ and $L = 2$. In this

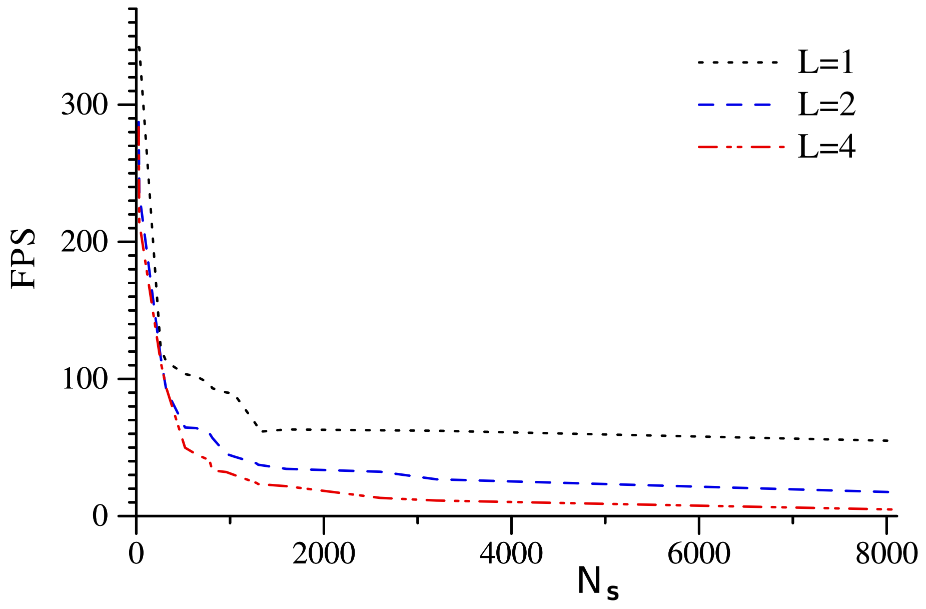
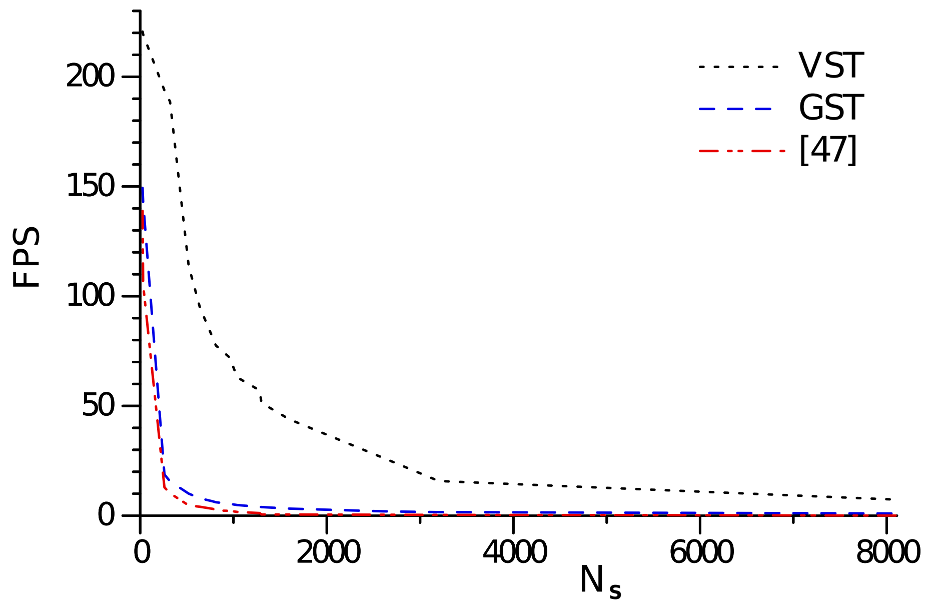


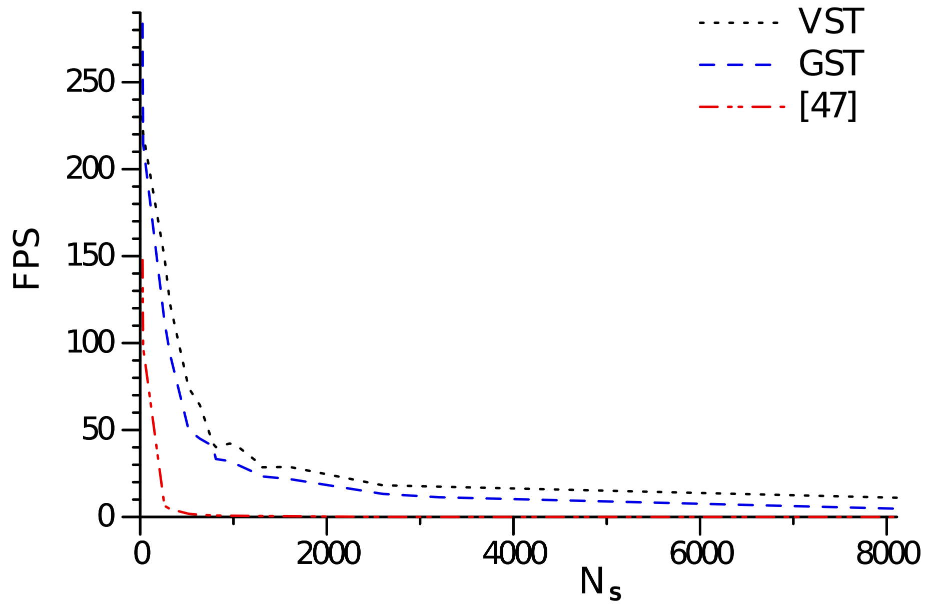
Figure 2.11: Performance of the GST alternative in an ATI 3870x2

case, as $L = 4$ evaluates and renders a greater amount of primitives than $L = 2$, because $L = 2$ renders about a 5% of the primitives of $L = 4$, and consequently lower performance is obtained in $L = 4$.

Figure 2.12 depicts the performance obtained with $L = 4$ in two different GPUs architectures considering the same number of triangles. As a first step in our analysis, we compare our designs with the proposal presented in [47] - our baseline, as it is one of the most classical algorithms in surface tessellation. Our proposal clearly achieves better performance in all cases and in all architectures.VST and GST obtain good performance in terms of FPS, allowing real time adaptive tessellation, including a large tessellation with a high number of triangles. For example, as shown in Table 2.1 and in Figure 2.12 for 9.14 K input triangles (scene S_5), 976.02 output triangles are generated in a non-adaptive tessellation, where more than a hundred output triangles are generated for each input triangle. Considering the performance 113.8 fps for VST and 10.04 for GST are obtained with GeForce 9800GTX while 74.163 and 49.886 fps are obtained with VST and GST, respectively in ATI Radeon 3870 X2. In this case, as indicated in Table 2.1, the number of triangles generated with an adaptive approach is 509.72 K. On the other hand, GST performs worse than VST, although in the latter case a greater amount of memory is used. As in all



(a)



(b)

Figure 2.12: Comparative for $L = 4$ (a) Nvidia GeForce 9800 GTX and (b) ATI Radeon 3870 X2

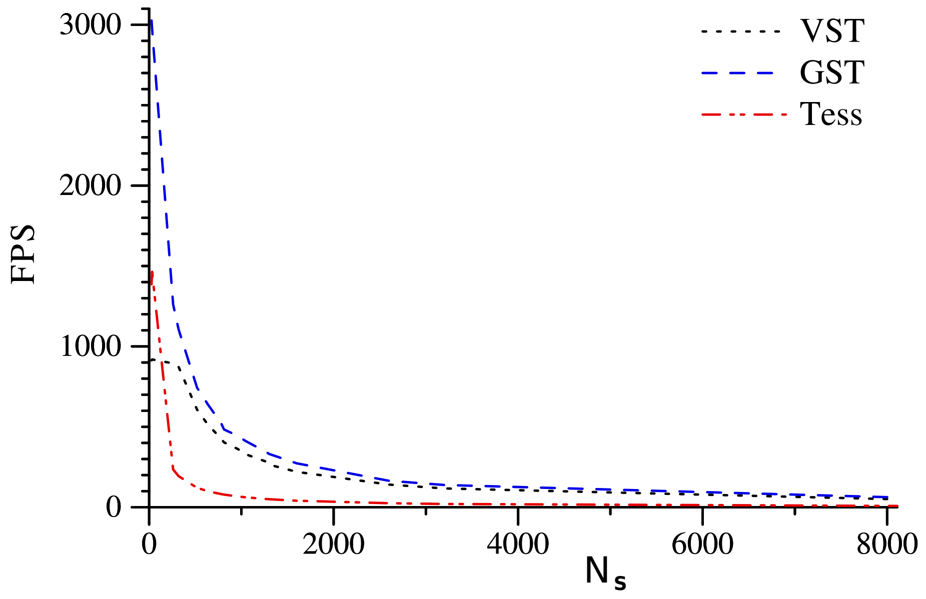


Figure 2.13: Comparative for $L = 4$ in an ATI 5870

cases we compare equivalent systems with the same resolution level, the differences in performance seems to stem from the limitation of the geometry shader. More specifically, for $L = 4$ the computation of two geometry shaders (two stages) is required to obtain the desired resolution level.

Furthermore, both GPUs, Nvidia and ATI, are from the first generation of graphics cards with a geometry shader stage, where a non-optimal implementation of the geometry shader is included, whereas the next generation of GPU has been provided with an improved geometry shader implementation, as shown in Figure 2.13. This ATI 5870 GPU offers a better geometry shader implementation than previous versions, thus the GST alternative obtains a better performance than the VST one. Moreover, in this case both alternatives demonstrate better performance than the tessellator stage included with DX11. A different computation scheme is considered in DX11 implementation, as the Bézier surface has to be evaluated for each new sample, instead of reusing the computation, as this non-adaptive proposal does. Furthermore, this ATI card includes two hardware tessellators(one for high tessellation factors and the other for low tessellator factors), and as rendered models are quite compact, they select a similar tessellator factor for their surfaces. Consequently, the selected hardware tessellator is the bottleneck of this rendering.

In summary, the results demonstrate that both proposals obtains a good performance on current GPU.

2.5. Conclusions

In this chapter we have presented a non-adaptive proposal for the tessellation of Bézier surfaces on the GPU based on the exploitation of spatial coherence, with two different alternatives considering different GPU features. The first alternative, VST, performs the tessellation by exploiting the vertex shader as a vertex coordinate evaluator, while the second one, GST, exploits the capabilities of the geometry shader as a primitive generator.

VST is based on the utilization of virtual vertex strategy and a system of multiresolution parametric maps. The utilization of this system of maps to evaluate the final coordinates of the virtual vertices allows multiple surfaces to be processed in parallel. Additionally, and in order to exploit the data locality and reduce the number of global memory accesses, an analysis of the optimum number of surfaces to be processed in parallel was performed.

With respect to the second alternative, GST, it is based on the exploitation of the geometry shader as a primitive generator. Due to the current limitations of the shader, in terms of number of primitives generated per input primitive, GST is based on the utilization of a smaller primitive: a parametric map section. This strategy leads to an increment in the output resolution while keeping all the advantages of a direct implementation.

We have obtained highly satisfactory results in terms of timing requirements for complex scenes, with VST being slightly superior to GST.

Chapter 3

Dynamic and Adaptive Bézier Tessellation

Non-adaptive tessellation has a number of different drawbacks. These include producing an excessive amount of triangles without increasing the quality of the final mesh, and not generating sufficient triangles in highly complex areas. The best solution is a high-quality adaptive tessellation that can be run on the fly and characterized with a high degree of freedom in the tessellation pattern generation. A desirable adaptive subdivision strategy adapts the tessellation pattern depending on a certain measure, such as view-dependent parameters or flatness [5, 31].

There are two main approaches to performing the adaptive tessellation on the GPU [12, 34, 35, 62, 76, 85] with DirectX10. The first solutions [12, 62] are based on the computation and storage of a set of tessellation patterns on the GPU and the selection of the correct pattern for each triangle of the incoming mesh at run time. The disadvantage of the proposal is that the number of available patterns is limited and must be pre-computed, reducing the adaptability of potential tessellations patterns. The second group proposals [34, 35, 76, 85] perform a per primitive view-dependent adaptive tessellation with a fixed pattern where the same tessellation pattern is applied to everything.

Previous proposals in the bibliography work with more flexible adaptive subdivision strategies [5, 31]. These proposals were oriented towards early GPUs with a lower degree of programmability. For this reason the adaptive tessellation was designed not to be programmed but to be implemented in an additional specific hardware unit. The adaptive tessellation algorithms employed are based on recursive tessellation strategies, where each edge of the triangle is conditionally subdivided into two in each iteration. Owing to this recursive structure, the mesh has to be reconstructed after each iteration, resulting in irregular memory access patterns and complex control flow or hardware implementations.

In this chapter we present a new method for adaptively tessellating Bézier surfaces on the GPU. Our Dynamic and Adaptive Bézier Tessellation (DABT) is based on the Adaptive and Dynamic Mesh Refinement (ADMR) [10] tessellation scheme for triangle meshes. Tessellation is performed according to a local test to generate primitives dynamically. The refinement procedure does not require the pre-computation of any refinement pattern. The resulting adaptive procedure is efficient, simple and generates the tessellation pattern of each triangle dynamically. The non-recursive strategy simplifies mesh reconstruction, avoids irregular memory access patterns, and uses simple control flow to make it a good candidate for guiding the evolution of tessellation algorithms on future graphics cards.

However, DABT has been designed to tessellate parametric surface meshes with a higher level of adaptively. So, DABT extends the capabilities of the proposal presented in [10] as different levels of resolution inside the patch are allowed. Furthermore, DABT permits the minimization of the number of triangles to be processed without reducing the quality of the final images. In our method, the tessellation level is not fixed for the full patch and can be locally changed as a function of different speed/quality parameters. The adaptive tessellation procedure is guided by local tests that avoid cracks between adjacent patches. More specifically, in this chapter we work with three tests that analyze different properties as guidance for the tessellation: distance between mesh and ideal surface, flatness of the mesh and size of the triangles.

This chapter is organized as follows: firstly, in Section 3.1 the Adaptive and Dynamic Mesh Refinement scheme is detailed. In Section 3.2 our Dynamic and Adaptive Tessellation of Bézier surfaces procedure is presented. In Section 3.3 the implementation we performed to test our proposal is detailed. The results obtained are included in Section 3.4. Finally, in Section 3.5 the main conclusions are highlighted.

Experimental results and methodology have been published in [10], [24] and [25].

3.1. Adaptive and Dynamic Mesh Refinement

In this section we describe the proposal, ADMR [10], for performing adaptive tessellation on GPUs. This proposal was originally developed for the adaptive tessellation of triangle meshes as the result of the collaboration of different research groups: GAC at UDC, GAC at USC and Graphics Group at Lund University. We have subsequently developed the DABT algorithm by extending, improving and adapting the ADMR algorithm to the of case of Bézier surfaces. The ADMR strategy is to tessellate the mesh by computing the tessellation pattern on the fly for each incoming triangle without employing any pre-computed pattern. The objective is a freely adaptive tessellation of triangle meshes where the resolution and number of triangles generated per incoming triangle can be selected as a trade-off between quality and computational requirements.

The ADMR method performs the tessellation on an independent triangle basis, so the procedure can be applied to any triangle mesh. The coarse mesh is transmitted from CPU to GPU and the tessellation is performed completely on the GPU without the supervision of the CPU. The proposal can be implemented in current GPUs by exploiting their geometry shader capabilities. However, the simplicity and good results obtained make it a good candidate for integration into future graphics cards as a tessellation unit.

The tessellation pattern is computed on the fly and any local test can be employed to guide the tessellation. The tests employed for the tessellation can be freely selected; for example, tests based on camera position, normal analysis, displacement map analysis, curvature analysis, etc. could be implemented [5, 31]. Note that a combination of tests can be applied before inserting each candidate position. This increases the flexibility of the tessellation technique, as a wide range of different criteria can be simultaneously employed in each area of the mesh. Additionally, the adaptive tessellation procedure can be guided by any tests local to the edges or to the full triangle, or could be extended to the neighbor triangles. For example, for a geometry shader implementation the subdivision quality can be enriched by exploiting the data accessibility available [9, 25]. The candidate vertices to be inserted are checked and conditionally inserted.

The tessellation is guided by a fixed tessellation pattern, where each vertex location is tested and conditionally inserted. Once the inserted vertices are calculated, the next step is to connect the vertices to construct a new mesh. The tessellation should generate triangles covering the surface with no holes or cracks.

ADMR is based on the representation and management of these tessellation decisions. More specifically, the method we employ is based on the classification and organization of the new vertices into strips. The representation and utilization of these strips of vertices permits the construction of an adaptive tessellation in a simple, efficient way without requiring a recursive procedure.

In summary, the ADMR proposal is based on three key ideas: the tessellation pattern employed, the representation of inserted vertices and an efficient tessellation procedure based on this representation. We now go in to present each key point of our proposal.

3.1.1. Tessellation Pattern

Tessellation algorithms with a recursive nature have a number of disadvantages. With these strategies each triangle is recursively subdivided and the mesh has to be reconstructed and stored after each iteration. This results in complex meshing algorithms with irregular memory access patterns that are not adequate for either a direct GPU implementation on the geometry shader or for a specific hardware implementation. Hence, the ADMR method employs a non-adaptive tessellation pattern as a basis for the adaptive case. The resolution level of this pattern can be selected, for example, by using a view point criteria and is applied to the entire mesh. Once the tessellation level of the pattern is selected only the positions in this tessellation pattern are evaluated for conditional insertion at each position.

Figure 3.1 shows the tessellation patterns for four different levels of resolution $L = \{0, 1, 2, 3\}$. The original coarse triangle is depicted with bold lines and the sampling points corresponding to the candidate vertices with a cross. The parametric coordinates (u_B, v_B) of the sampling point associated with a candidate vertex

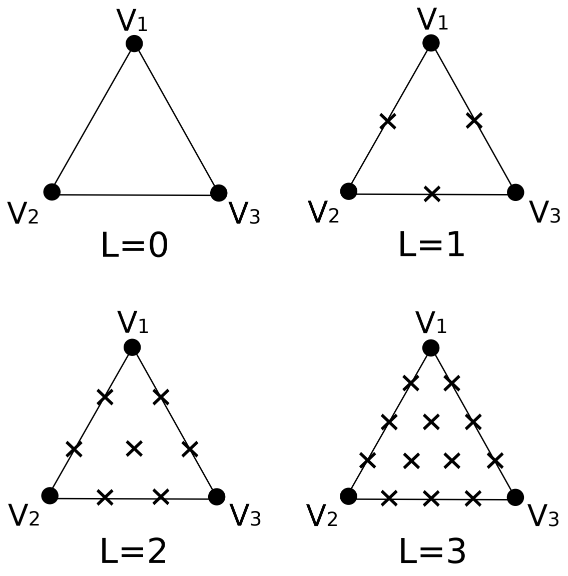


Figure 3.1: Tessellation patterns for $L = \{0, 1, 2, 3\}$

V_B that lies on the Bézier surface are computed through its barycentric coordinates with:

$$(u_B, v_B)(w_i, w_j, w_k) = w_i \times (u_1, v_1) + w_j \times (u_2, v_2) + w_k \times (u_3, v_3)$$

where (u_1, v_1), (u_2, v_2) and (u_3, v_3) are the parametric coordinates of the vertices of the initial coarse triangle and (w_i, w_j, w_k) are the barycentric coordinates of the candidate vertex. The barycentric values are in the interval $[0, 1]$ and verify $w_i = i \times \delta w$, $w_j = j \times \delta w$ and $w_k = k \times \delta w$ with $i, j, k = \{0, \cdots, L+1\}$ and $\delta w = \frac{1}{L+1}$.The tessellation is performed in the parametric domain, so to obtain the Euclidean space coordinates of the vertex V_B Equation 1.8 is evaluated.

The positions associated with this non-adaptive pattern are classified into strips of candidate vertices. Figure 3.2 shows the tessellation pattern we employ for five strips of vertices. The original coarse triangle is depicted with a bold line and the triangles generated for the non-adaptive case with dashed lines. In our ADMR proposal, the vertices (indicated with labels in the figure) are evaluated and are conditionally inserted depending on the result of a specific test.

This tessellation methodology is generic and can be applied to any number of strips, where this number is selected according to the resolution desired. The advantages of this simple tessellation methodology are multiple. First, the tessellation has a non-recursive structure and for each triangle, once its tessellation level is selected, the locations of the candidate vertices are determined. On the other hand, the positions of the final vertices to be inserted can easily be evaluated through their barycentric coordinates. In the case of a hardware implementation, the barycentric coefficients could be stored to be applied in run time. Moreover, the tessellation procedure is not limited to specific pre-stored patterns, but can be determined on the fly, depending on the tessellation level selected and the tessellation test employed. The lack of recursiveness and the simplicity of the algorithm make it suitable for including a full adaptive tessellation unit based on our tessellation strategy on a GPU.

3.1.2. Strips of Vertices Representation

In our ADMR proposal the adaptive tessellation is generated once the vertices are conditionally inserted. Our tessellation algorithm, explained below, is based on the efficient management of the tessellation pattern representation. This representation, dealt with in this section, is based on the classification of the inserted vertices in strips. This simple representation, together with the efficient management thereof, leads to a simple tessellation algorithm, as will be shown in the remainder of this chapter.

Let us represent the strips of vertices as, Sv, a tuple of s lists $Sv = (Sv_1, \cdots, Sv_s)$ corresponding to the s strips. Each list includes the vertices inserted in each strip; i.e., vertices that comply with the tessellation criterion. Non-inserted vertices are not included in the list and their positions are empty. We define a *limit vertex* as one on the edge of the original triangle with a vertex on the left edge of the triangle being an *opening limit vertex* and on the right edge of the triangle a *closing limit vertex*. We also define an *extreme vertex* as the first/last vertex in a strip when there is no opening/closing limit vertex. By way of example, with a non-adaptive tessellation with five strips (see Figure 3.2) the system of lists is:

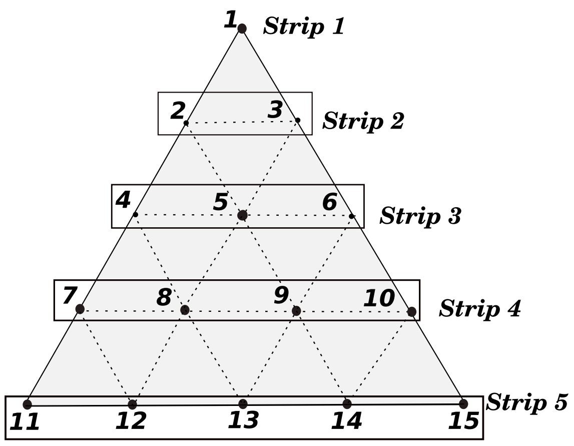


Figure 3.2: Tessellation Pattern employed

$$
\begin{aligned}
Sv_1 &= \{\mathbf{1}\} \\
Sv_2 &= \{\mathbf{2}, \mathbf{3}\} \\
Sv_3 &= \{\mathbf{4}, 5, \mathbf{6}\} \\
Sv_4 &= \{\mathbf{7}, 8, 9, \mathbf{10}\} \\
Sv_5 &= \{\mathbf{11}, 12, 13, 14, \mathbf{15}\}
\end{aligned}
$$

where each list Sv_i includes the vertices j in the $i-th$ strip, with $j = 1, \cdots, n$ being $n <= i$. The limit vertices in each strip are indicated in bold typeface.

An example with an adaptive tessellation is shown in Figure 3.3 where vertices 5, 6, 7, 9 and 10 are inserted. In this case the strips of vertices are represented according to the following lists:

$$
\begin{aligned}
Sv_1 &= \{\mathbf{1}\} \\
Sv_2 &= \{\} \\
Sv_3 &= \{5, \mathbf{6}\} \\
Sv_4 &= \{\mathbf{7}, 9, \mathbf{10}\} \\
Sv_5 &= \{\mathbf{11}, \mathbf{15}\}
\end{aligned}
$$

The limit vertices for this example, indicated in bold typeface, are: 1 in the first strip, none in the second strip, 6 in the third strip, 7 and 10 in the fourth strip, and 11 and 15 in the last strip. On the other hand, vertex 5 is an extreme vertex. As will be shown below, the identification of limit and extreme vertices will be the key

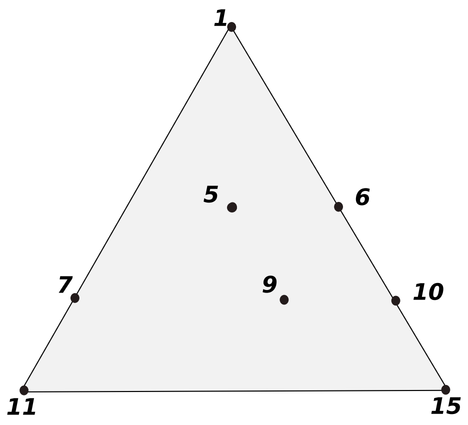


Figure 3.3: Adaptive tessellation according to the pattern of strips of vertices

to our reconstruction algorithm.

To avoid generating overlapping triangles, when the tessellation procedure is applied two modifications are introduced into the representation. These updatings are expounded in the following two subsections.

Reuse of limit vertices

The first modification, reuse of limit vertices from previous strips, is required when a limit vertex is missing in the current strip. For each non-empty list, when no opening/closing limit vertex is included, the limit vertex in the previous strip is copied. Note that only opening limit vertices can be employed for opening positions and closing limit vertices for closing positions with the unique exception of vertex 1 that can be employed for both purposes. By way of example, the extended lists for the tessellation pattern of Figure 3.3 becomes (limit vertices reused are indicated with a hat):

$$\begin{aligned} Sv_1 &= \{\mathbf{1}\} \\ Sv_2 &= \{\} \\ Sv_3 &= \{\hat{\mathbf{1}}, 5, \mathbf{6}\} \\ Sv_4 &= \{\mathbf{7}, 9, \mathbf{10}\} \\ Sv_5 &= \{\mathbf{11}, \mathbf{15}\} \end{aligned} \tag{3.1}$$

That is, the third strip is extended with the reuse of vertex 1 as an opening limit vertex.

Incorporation of extreme vertices

For the second modification we add an additional notation. We enrich the vertex classification with the definition of two kinds of diagonals in the tessellation pattern: opening diagonals D^o and closing diagonals D^c. For the example of Figure 3.4 with five strips, there are three diagonals of each type:

$$\begin{array}{ll} D^o_1 = \{-,3,5,8,12\} & D^c_1 = \{-,2,5,9,14\} \\ D^o_2 = \{-,-,6,9,13\} & D^c_2 = \{-,-,4,8,13\} \\ D^o_3 = \{-,-,-,10,14\} & D^c_3 = \{-,-,-,7,12\} \end{array}$$

where $D_j[i]$ is the vertex of Strip i in diagonal D_j, with $i = 1, 2, \cdots, s$. This classification of the diagonals is used in the ADMR tessellation procedure.

The second modification is the incorporation of new vertices in non-empty strips with no opening or closing vertex in Sv_i and Sv_{i-1} with $i > 2$. For these strips an additional vertex is conditionally incorporated. For the situation with no opening vertices the procedure is as follows: first the opening diagonal D^o_j that includes the extreme vertex in Sv_{i-1}, that is, $Sv_{i-1}[1] \in D^o_j$, has to be identified. Additionally, the opening diagonal strip, that is, $Sv_i[1] \in D^o_k$, also has to be identified. If D^o_j is on the left of D^o_k, i.e. $j < k$, a vertex has to be incorporated into the current Sv_i in the position indicated by the diagonal associated to the extreme vertex above $D^o_j[i]$.

Similarly, for closing vertices, if the closing diagonal of the extreme vertex of Sv_{i-1}, D^c_j, is on the right of the closing diagonal of the extreme vertex of current strip, D^c_k, that is $j < k$, a vertex has to be incorporated into the current strip in the position indicated by the diagonal of the extreme vertex above $D^c_j[i]$.

As an example let us consider the tessellation pattern indicated in Figure 3.5(a). In this example we find one example of an extreme vertex incorporation when the third and fourth strips are analyzed. In this case the original list of vertices are: $Sv_3 = \{5, 6\}$ and $Sv_4 = \{9\}$. The upper strip has a closing limit vertex, vertex 6, and no opening limit vertex. The lower strip has no opening or closing limit vertex.

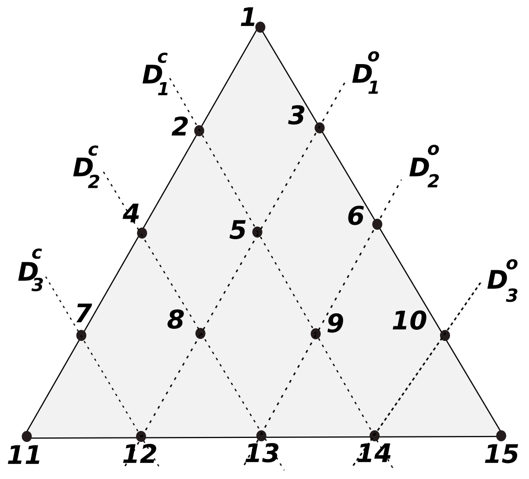


Figure 3.4: Diagonal organization

As both strips have no opening limit vertices the left extreme vertices have to be analyzed. The diagonal of the left most extreme vertex above, vertex 5, is D_1^o and it is on the left of the D_2^o associated to the left most extreme vertex below, vertex 9. This indicates that an extreme vertex has to be incorporated into Sv_4 in the position associated to $D_1^o[4]$, i.e. position 8. This insertion is depicted in Figure 3.5(b) where the new vertex is indicated with a circle. Taking into account this new configuration the resulting lists of vertices are:

$$\begin{aligned}
Sv_1 &= \{\mathbf{1}\} \\
Sv_2 &= \{\hat{\mathbf{1}}, 3\} \\
Sv_3 &= \{\hat{\mathbf{1}}, 5, \mathbf{6}\} \\
Sv_4 &= \{\hat{\mathbf{1}}, 8, 9, \hat{\mathbf{6}}\} \\
Sv_5 &= \{\mathbf{11}, 12, 13, 14, \mathbf{15}\}
\end{aligned} \tag{3.2}$$

Note that there is no mark to distinguish an incorporated extreme vertex from an original vertex.

3.1.3. Adaptive Tessellation Procedure

In this section we present the adaptive tessellation algorithm. The tessellation procedure works by processing pairs of consecutive strips of vertices. The method is

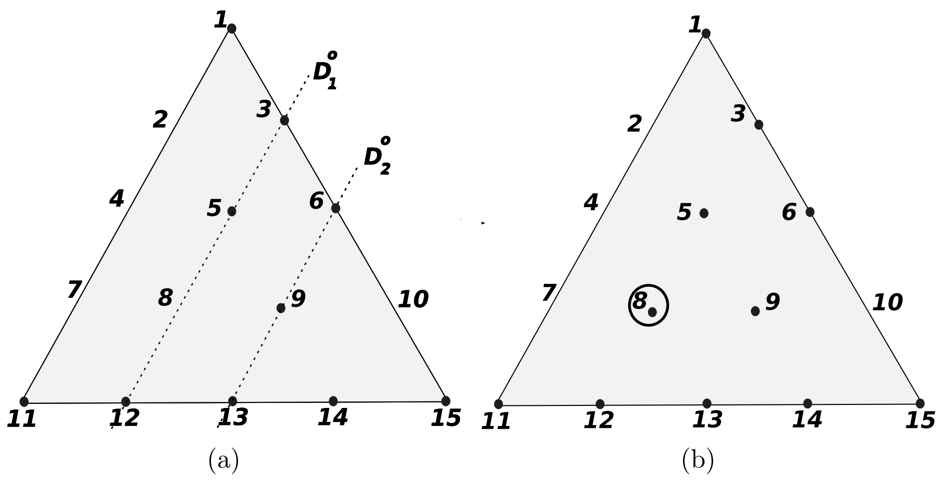


Figure 3.5: Example of extreme vertex incorporation

based on the utilization of the lists of vertices presented so far. Using these lists of vertices a simple and efficient tessellation procedure is obtained. In the following we present our meshing algorithm that permits the encoding/reconstruction of triangles.

In the method we propose the triangles are generated by joining the vertices between consecutive strips (parent-children relation) taking into account the following rules:

- Two consecutive vertices in the same strip are connected (sibling relation).
- Two identical vertices are not considered for connection.
- A reused opening/closing limit vertex has a limited connection with the following non empty strip. More specifically, it can only be connected with a non-reused opening/closing limit vertex.
- Each non-reused vertex of each strip, considered as parent, is connected with consecutive children in the following strip. The following parent is connected with another group of consecutive children. There is an overlap of one common child between two consecutive parents.

The application of these rules to each pair of extended vertex lists generates the final tessellation. For the sake of clarity, let us once again consider the example of

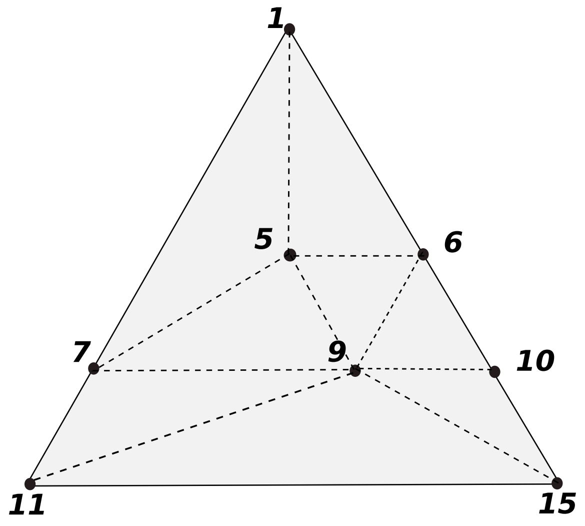


Figure 3.6: Example of tessellation for a list with reused limit vertices

Figure 3.3 and its extended list of vertices indicated in Equation 3.1. The resulting tessellation generated by our proposal is shown in Figure 3.6. The procedure for generating this tessellation is as follows. For the analysis of the first two strips, $\{\mathbf{1}\}$ and $\{\hat{\mathbf{1}}, 5, \mathbf{6}\}$, vertex 1 is connected with vertices 5 and 6. The following set of two strips are $\{\hat{\mathbf{1}}, 5, \mathbf{6}\}$ and $\{\mathbf{7}, 9, \mathbf{10}\}$. As vertex 1 is a reused vertex, it is only connected with vertex 7. After this, vertex 5 is the first non-reused parent vertex, so it is connected with a set of consecutive children {7, 9}. After this, vertex 6 is the last parent and is connected with the remaining children with an overlap of one vertex with the previous set of children; i.e., with {9, 10}. In the last set of strips, $\{\mathbf{7}, 9, \mathbf{10}\}$ and $\{\mathbf{11}, \mathbf{15}\}$, vertex 7 is connected with child 11 in the following strip, vertex 9 with children 11 and 15 and vertex 10 with the last child. The distribution of children among parents was selected to produce good results in terms of quality.

For the example of Figure 3.5 with the list of strips of vertices indicated in Equation 3.2 the resulting tessellation is indicated in Figure 3.7. The incorporated extreme vertex 8 is equivalent to any other internal vertex. In consequence, the tessellation procedure can be applied directly. If vertex 8 were not inserted, an undesirable triangle (1,5,9) would be generated. Note that the reuse of limit vertices allows the two strips tessellation philosophy to be preserved for triangles with larger extensions. Note also that two identical vertices do not imply any real connection

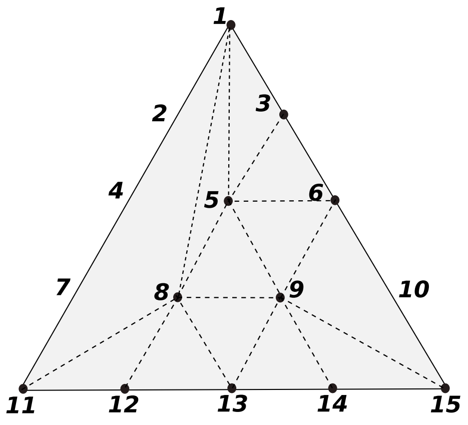


Figure 3.7: Example of tessellation for a list with reused limit vertices and incorporated extreme vertices

as the reused vertices play the same role as the original vertex.

3.2. Structure of Dynamic and Adaptive Bézier Tessellation

In this section we present our Dynamic and Adaptive Bézier Tessellation (DABT) proposal. Our strategy performs the adaptive tessellation of the Bézier surface by computing the tessellation pattern on the fly without employing a set of precomputed patterns. The objective is a freely adaptive tessellation inside each patch where the resolution and number of triangles generated can be selected as a trade-off between quality and computational requirements.

Non-adaptive subdivision strategies can generate meshes with a high number of triangles that do not always contribute to the increment of the quality of the final image. The objective of our adaptive subdivision proposal is to reduce the number of triangles with no discernible loss in the quality of the final surface. The DABT follows a similar strategy to the ADMR algorithm, but increasing its flexibility in terms of the adaptivity of the tessellation. More specifically, the DABT does not apply a fixed resolution for the whole surface, thus enriching the possibilities of its predecessor.

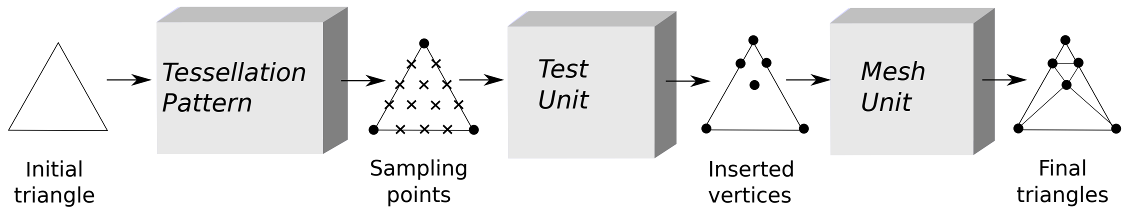


Figure 3.8: Scheme of the DABT algorithm

Specifically, like ADMR, our DABT method is based on three different key proposals: the utilization of a fixed tessellation pattern that guides the adaptive tessellation, the application of local tests, and an efficient meshing procedure to reconstruct the resulting mesh once the tests are applied. Figure 3.8 schematically shows these three key cores of the algorithm. As can be observed in the figure, a non-adaptive tessellation pattern is employed to guide the procedure and each candidate vertex can be conditionally inserted in the positions specified by this pattern. The vertices to be finally inserted are determined by local tests. And once the decisions are performed, the new vertices have to be organized to reconstruct a high quality triangle mesh. This tessellation procedure is based on the classification of the inserted vertices into strips and the efficient management of the resulting list of vertices. The objective in mind is to generate the triangle structure directly from the irregular pattern obtained with the evaluation of the subdivision tests.

In the following each key point of our strategy is presented.

3.2.1. Utilization of a Fixed Pattern to Guide the Adaptive Tessellation Procedure

The objective of the proposal is to exploit the large number of cores available in current GPUs. With this objective in mind, the model's patches are initially tessellated and the coarse triangles employed as input primitives for the application. Once the initial coarse mesh is obtained (see Figure 3.9 by way of example), the DABT method computes the real-time adaptive tessellation of the resulting triangles.

For the ADMR proposal, once the surface resolution level is determined, the

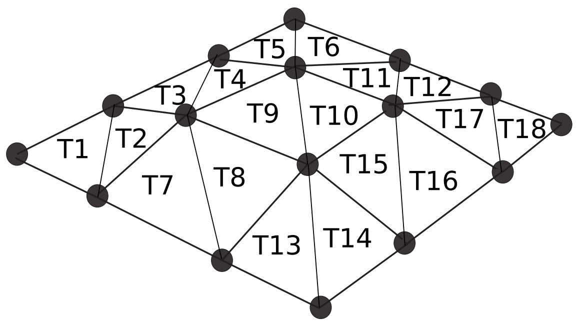


Figure 3.9: Initial coarse tessellation of the Bézier surface

candidate vertex locations can be evaluated directly. Those positions associated with the original coarse edges have to be coherently processed by the neighbor triangles sharing each edge. This can be assured with the assignation of the same resolution level to neighbor triangles and the application of local tests to guide the adaptive tessellation. With this uniform approach, once the tessellator level is selected for a given surface L, triangles are subdivided according to the pattern for that resolution level.

In manner similar to the ADMR proposal, the DABT algorithm employs fixed tessellation patterns from ADMR (see Figure 3.1) to guide the adaptive tessellation procedure. To enrich the adaptative capabilities of the algorithm, the DABT includes a method for assigning different resolution levels to neighbor triangles. With this non-uniform approach, the resolution level can be selected dynamically and modified along the patch. As neighbor triangles could have different resolution levels, the direct application of the fixed tessellation pattern methodology would result in the generation of cracks. For this reason, the DABT algorithm permits the application of multiple resolution levels inside a triangle. More specifically, a resolution level is selected for each triangle edge so that one triangle may have three resolution levels. In order to apply three levels per triangle, each one would be applied to one third of the triangle, as indicated in Figure 3.10. By way of example, the resolution level associated with edge $\{V_1, V_2\}$ is applied to the area labeled with $E_{1,2}$.

Specifically and in order to follow a row computation strategy, an unified resolution is selected and employed. For a system with resolution levels $L = \{0, \cdots, L_{max}\}$, the unified resolution level corresponds with the least common multiple of $\{0, \cdots,$

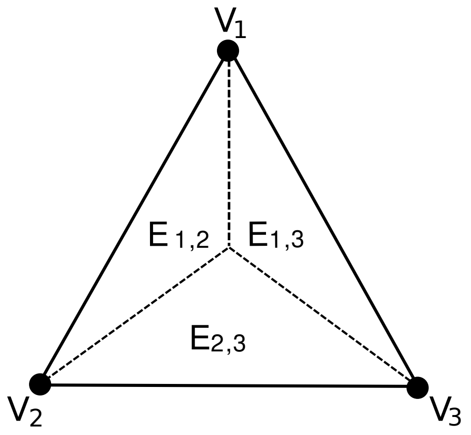


Figure 3.10: Resolution areas inside a triangle

$L_{max} + 1\}$ minus 1. As an example, let us consider a system with resolutions $L = \{0, 1, 2, 3\}$ as indicated in Figure 3.1; in this case the unified resolution level is $L_{unified} = 11$. The barycentric weights employed for the candidate vertices for each resolution are:

$$w^1 = \left\{\tfrac{1}{2}\right\}$$
$$w^2 = \left\{\tfrac{1}{3}, \tfrac{2}{3}\right\}$$
$$w^3 = \left\{\tfrac{1}{4}, \tfrac{1}{2}, \tfrac{3}{4}\right\}$$

being w^L the weights employed for level L. The unified system of weights is:

$$w^{11} = \left\{\frac{1}{12}, \frac{1}{6}, \frac{1}{4}, \frac{1}{3}, \frac{5}{12}, \frac{1}{2}, \frac{7}{12}, \frac{2}{3}, \frac{3}{4}, \frac{5}{6}, \frac{11}{12}\right\}$$

where $w^1_0 = w^3_1 = w^{12}_5$, $w^2_0 = w^{12}_3$, $w^2_1 = w^{12}_7$, $w^3_0 = w^{12}_2$ and $w^3_2 = w^{12}_8$. As a result, a unified system of weights and rows can be employed for any resolution level. Figure 3.11 shows the tessellation pattern for $L_{unified} = 11$. In fact, the test unit processes only those points associated with the resolution level selected. In the figure the points corresponding to L=1 (lines s7 and s13), L=2 (lines s5, s9 and s13) and L=3 (lines s4, s7, s10 and s13) are indicated with circles.

Figure 3.12 shows a triangle where three different resolution levels coexist. In this example the area $E_{1,2}$ follows a tessellation pattern $L = 3$ that means that three

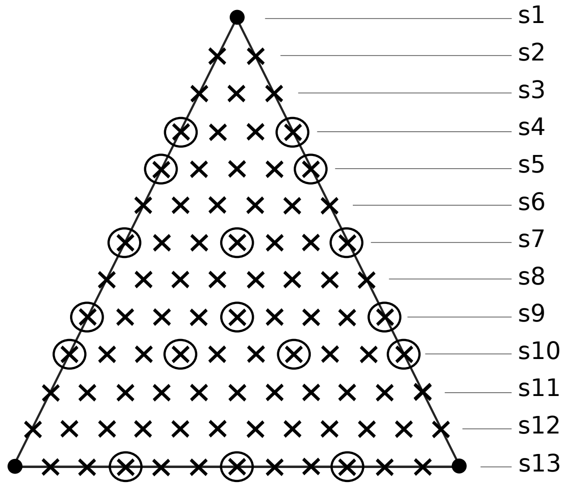


Figure 3.11: Unified resolution $L_{unified}$=11 for a system with $L = \{0, 1, 2, 3\}$

rows of candidate vertices have to be analyzed. Area $E_{1,3}$ is subdivided according to a tessellation pattern $L = 2$; i.e., two rows of candidate vertices have to be considered. Finally, $E_{2,3}$ is tessellated with a resolution level $L = 1$. In this case and according to the tessellation pattern corresponding to this area (see Figure 3.1) only one candidate vertex has to be evaluated. All the candidate vertices have to be evaluated and conditionally inserted on the basis of results from specific tessellation tests.

The advantages of the tessellation methodology employed are multiple. First, the tessellation has a non-recursive structure. On the other hand, the positions of the candidate vertices can easily be evaluated through their barycentric coordinates. The tessellation procedure is not limited to specific pre-computed patterns, but can be determined on the fly according to the tessellation level selected. And finally, the resolution level can be dynamically changed so that no fixed resolution level per surface is required.

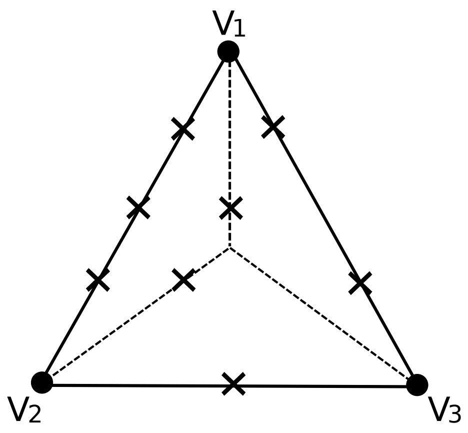


Figure 3.12: Example of triangle with three different resolution areas

3.2.2. Selection of Tests Employed to Guide the Adaptive Tessellation

Once the resolution levels per triangle are determined and the sampling points identified, the candidate vertices are conditionally inserted according to the result of specific tessellation tests (see Figure 3.8). The objective of the adaptive tessellation is to generate detailed structures only on those areas where a high resolution is required, while keeping the coarse structures in those areas where a higher tessellation does not imply any increment in the quality of the final image.

We have evaluated different tests to guide the tessellation procedure. Through the utilization of different quality thresholds, these tests measure the increase in quality of the mesh if a given vertex is inserted. From among the tests evaluated we have selected three tests according to their sound results in terms of simplicity and quality obtained. The first test is based on a distance mesh-surface analysis, the second on the study of the curvature of the surface, while the third is based on length evaluations. The three tests compute the analysis of the surface properties and can be applied in combination with other tests. We now go on to explain the three tests individually.

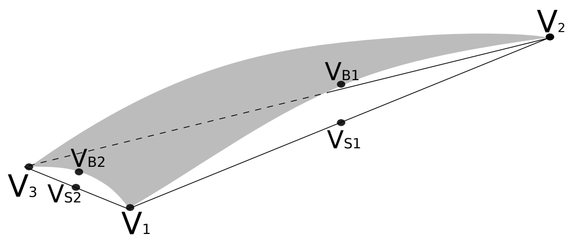


Figure 3.13: Example of candidate vertices under test

Distance test

This test analyzes the distance between the triangle mesh and the Bézier surface. More specifically the distance between a sampling point on the triangle mesh and the corresponding point on the Bézier surface is analyzed. If the distance is small enough, the triangle mesh is considered a good approximation of the surface, so no vertex is inserted. On the contrary, if the distance is large, the vertex is introduced as this will increase the quality of the final image.

To apply this test, the barycentric coordinates of the candidate vertex are obtained. These coordinates are employed to compute the parametric coordinates of the vertex and, by using Equation 1.8, the coordinates of the candidate vertex V_B on the Bézier surface. Additionally the barycentric coordinates are also employed for computing the coordinates of the corresponding sampling point V_S on the coarse triangle by interpolating the position of the original vertices of that triangle. Once both sets of coordinates have been computed, the test is given by:

$$distance = [|V_S - V_B| > t_{distance}]$$

where $t_{distance}$ is a quality threshold that is selected according to the quality/timing requirements of the application. Figure 3.13 shows an example of application. In this example, the analysis of the distance between the candidate vertices V_{B1} and V_{B2} and the corresponding points on the coarse mesh, V_{S1} and V_{S2} could lead to different insertion decisions. In this case, vertex V_{B1} would be inserted as the distance between V_{B1} and V_{S1} is larger than the threshold and its inclusion would contribute to the quality of the final image. However, the insertion of vertex V_{B2} would not provide enough additional detail to the final image, so it would not be inserted.

As will be shown in the results section, this test produces good results in terms of quality of the final mesh and, in this sense, is adequate for guiding the adaptive tessellation. As a drawback, the test is based on the analysis of the candidate vertices (V_B). This means that the coordinates of all candidate vertices have to be obtained for the testing procedure. Consequently, the computational cost associated with the vertices not finally inserted is an important drawback of the test.

Vector deviation flat test

The objective of this test is to employ the curvature of the surface as a parameter for guiding the tessellation procedure. With this test candidate vertices on flat areas are not considered for insertion as the quality of the surface would not be incremented. To check the curvature of the surface a simple vector deviation flat test [36] can be employed.

In order to reduce the costly computation of all candidate vertices, our flatness test follows a per edge philosophy. In our proposal each candidate position associated with the coarse edges is analyzed and the decision performed is applied to all candidate positions of the same row and resolution area (see Figure 3.10). Thus, with this technique the curvature of the surface is estimated only for the sampling points on the edges of the coarse triangle and the results of the tests are employed in the interior of the triangle

For this test the curvature of the surface on the positions corresponding to the evaluation of the triangle edges is analyzed. For an edge with extreme points V_1 and V_2 and for a position V_B under test, the curvature in this position is tested. With this objective, the test computes the normalized vector $|V_1 - V_2|$ and the dot product of this vector with $|V_B - V_1|$ and $|V_B - V_2|$. The deviation between the vectors that point to the new vertex and the edge vector are analyzed as an estimation of the curvature in this point. More specifically, the test consists of the following steps:

1. Calculation and normalization of vectors $A = |V_1 - V_2|$, $B = |V_B - V_1|$ and $C = |V_B - V_2|$.

2. Computation of the unsigned dot products $|BA|$ and $|CA|$

3. Comparison between the dot products and a threshold, t_{flat}. If one of them is smaller than the threshold, the new vertex is inserted. The test can be represented by the equation:

$$flat = (|BA| < t_{flat}) \ OR \ (|CA| < t_{flat})$$

This test has the same drawback as the distance test analyzed above. The coordinates of each candidate vertex V_B are calculated for testing purposes. For those vertices not finally inserted, this information is not longer useful. However, and even though the computational costs associated with each point under test are higher, the global costs are lower as only the candidate positions located on the triangle edges are really tested.

Length test

In geometric design applications, rather than using a very high degree surface to approximate a very complex surface, it is more common to break the surface up into several simple surfaces. The usual way to make larger and complex surfaces is to connect up a set of low degree Bézier surfaces. The test we have employed exploits this typical low degree/curvature of the Bézier surfaces. More specifically, the test is based on the utilization of the length of the coarse triangle edges as a measure of the curvature of the Bézier surface in the corresponding area.

Due to the low degree of each Bézier surface, and as the vertices of the coarse triangle mesh lies on the surface, if the coarse triangle is small then it can be considered a good approximation to the surface. More specifically, if the two vertices of one edge are close enough, the inclusion of additional vertices on that edge will not increase the quality of the final mesh.

Similarly to the previous proposal, this test works on the edge basis as it is only applied for the sampling points on the edges of the coarse triangle. In the case of a vertex corresponding to the edge being inserted, the vertices on the same row are directly inserted (only for the resolution area associated with that edge, as indicated in Figure 3.10). Note that the test is based on the analysis of the original vertices of the triangle and does not require the computation of the candidate vertices.

Consequently, the computational requirements of this new test are very low.

More specifically, to test whether a candidate vertex V_B has to be inserted in the edge with vertices V_1 and V_2 the following analysis is performed:

$$length = (|V_1 - V_S| > t_{length})\ AND\ (|V_2 - V_S| > t_{length})$$

this means that the point V_B is inserted only when the distance of the corresponding sampling point on the triangle V_S to both extreme vertices V_1 and V_2 is larger than a threshold t_{length}.

By way of example, let us consider Figure 3.13. To test if V_{B2} is inserted the distance of its projection V_{S2} to V_1 and V_2 is evaluated. It should be noted that the test does not require the costly evaluation of the V_{B2} coordinates.

This test has a number of different advantages in terms of computational requirements. First, the test is applied at the edge level, so only the candidate positions on the triangle edges are tested. Additionally, the test does not imply the computation of the vertex coordinates, so computations associated with vertices not finally inserted are avoided. As result, the computational requirements are very low.

3.3. Implementation of the DABT Algorithm on the Geometry Shader

Our implementation of the DABT was performed to test the quality of the tessellation technique as the speed results are conditioned by the limited hardware tessellation resources available in current graphics cards. The new tessellation units included with DirectX 11 [90] offer a high performance solution for the tessellation proposals. However, current versions of these units have reduced flexibility with respect to the adaptivity allowed. In this sense, the geometry shader introduced with DirectX 10 [9] is the only unit that provides support for a fully adaptive tessellation. However, the geometry shaders have different disadvantages in terms of the number of primitives generated and reduced processing speed.

Our implementation processes bi-cubic Bézier surfaces and exploits the capabilities of the geometry shaders. As will be shown in the results section, even with the strong limitations associated with these units, a very good performance in terms of quality and timing requirements is obtained. We hope that the good results obtained will lead to the integration of more flexible tessellation units in future graphics cards.

Figure 3.14 shows the scheme of our DABT implementation. The algorithm consist of two stages: *Preprocessing Stage* and *Synthesis Stage*. In the *Preprocessing Stage* two step are performed: the control points of the Bézier surfaces (Step 1) and the virtual vertices (Step 2) are sent from CPU to GPU. In the *Synthesis stage*, Bézier surfaces are tessellated in the geometry shader (Step 3) and rendered on the GPU. While the *Preprocessing Stage* is performed only once per scene, the *Synthesis Stage* can be performed multiple times per data set for successive frames f (see feedback in the figure).

The objective of our implementation is to exploit the large number of cores available in current GPUs. With this objective in mind, the patches of the model are initially tessellated and the resulting coarse triangles are employed as input primitives for the application. To do this, the parametric domain (u, v) is partitioned into $N_u \times N_v$ cells of size $\frac{1}{N_u} \times \frac{1}{N_v}$, where two adjoining triangles are generated per cell. More specifically, our implementation employs $N_u = N_v = 3$ cells, so that eighteen coarse triangles are generated per Bézier patch (see as an example Figure 3.9). The reason for this number of triangles is directly associated with the degree of the Bézier surfaces employed. As our implementation processes bi-cubic surfaces (defined by 4×4 control points) the vertices of the coarse mesh can be directly computed by evaluating the parametric coordinates of the control points with E quation 2.2.

An additional advantage of using the control points as a basis for the initial vertices computation is the implicit simplicity of the coarse mesh reconstruction. The control points are sent once from CPU to GPU (Step 1 in Figure 3.14) and efficiently stored. More specifically, the surface control points $[B^s]$ are stored in three float 4×4 arrays $[B^s_x, B^s_y, B^s_z]$: one per coordinate. The storage is performed in a texture buffer (*tbuffer*) [66]. The reason for this selection is the superior performance of the texture memory with respect to the constant or global memory. The texture memory has less restrictive access patterns and hides memory latency accesses. Furthermore, the *tbuffer* has better performance as it allows the simul-

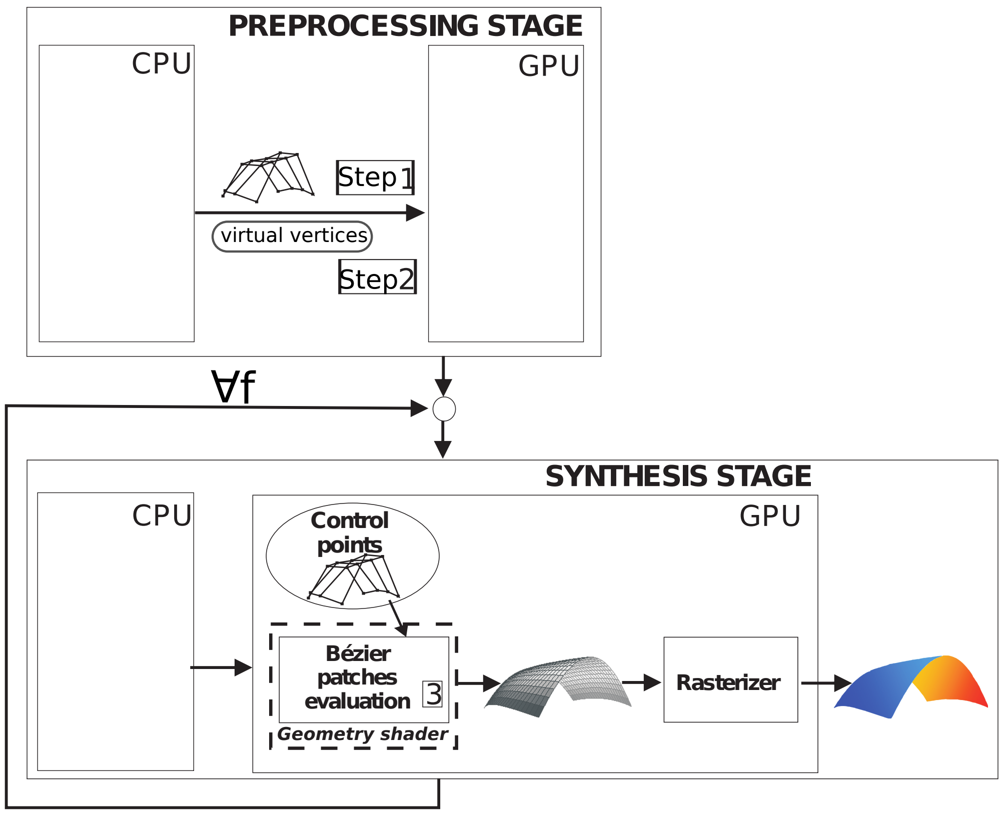


Figure 3.14: DABT structure using the Geometry Shader

taneous obtaining of different variables packed in the same buffer, thus increasing the bandwidth. As a result of the organized storage scheme employed, no *Index Buffer* is required. During the tessellation in the geometry shader, to generate each coarse triangle the control points coordinates are read from the texture memory as indicated in Figure 3.15. In this case a surface identification is stored on the vertex buffer instead of the whole surface. Thus, control points of the surfaces are stored in the texture memory in advance, which minimizes surface accesses due to the fact that the whole surface is accessed when it is computed. This process is schematically depicted in Figure 3.15, where the vertex buffer stores the surface ID which is used to access to the control points of the surface stored in the texture memory. Therefore, the control points of each surface are read once for each triangle in the surface and the projection of those points in the Bézier surface are generated according to Equation 1.8. Our implementation employs the direct evaluation strategy instead

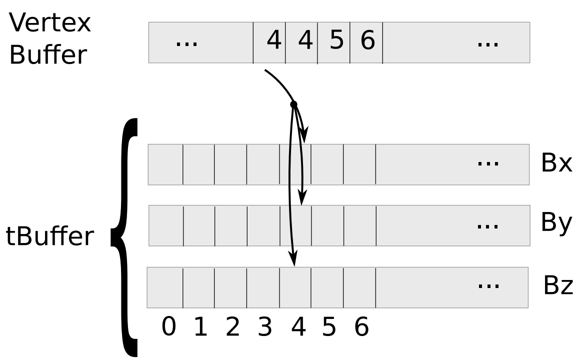


Figure 3.15: Access to tbuffer to recover control points

of the de Casteljau algorithm for this evaluation, owing to its greater efficiency in a GPU implementation [88].

With respect to the implementation of the adaptive tessellation and to increase the performance of the application a set of optimization strategies were used. The performed optimizations include the minimization of conditional branches and the utilization of vector operations and intrinsic functions. Flow control instructions are necessary for a fully adaptive tessellation performed on the basis of the evaluation of tests. Nonetheless, these instructions can significantly affect the throughput by causing threads assigned to different processing element of the same compute unit to diverge; i.e., to follow different execution paths. If this happens, the different execution paths are serialized, decreasing the performance. To increase the performance, conditional branches have been replaced with predication when possible. With branch predication, each instruction is associated with a predicate that is set to true or false according to the controlling condition. Only the instructions with a true predicate are actually executed and these are not serialized.

On the other hand, the Bézier surface evaluation is also a costly operation; hence, this evaluation has been minimized by employing Equation 3.3, which is shown again here for the sake of clarity:

$$Q(u,v) = [u^3 \quad u^2 \quad u \quad 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$
$$\begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{3.3}$$

This equation is characterized by a matrix evaluation so vector operations are used. This is more efficient than a loop evaluation (see Equation 1.6). Moreover, intrinsic functions have been also used owing to their superior performance. More specifically, Bézier and tests evaluations have been implemented using intrinsic functions.

3.4. Results

In this section we present the results of the evaluation of our proposal in terms of the quality of the final image and frames per second (fps). We have evaluated our proposal on an Intel Core 2 2.4 GHz with 2 GB of RAM and on two different GPUs: Nvidia GeForce 295 GTX (*Nvidia*) with DirectX 10 Microsoft's HLSL and ATI Radeon 5870 (*ATI*) with DirectX 11 Microsoft's HLSL.

The models employed in the tests presented in this section are shown in Figure 3.16: *Teacup* in Figure 3.16 (a), *Teapot* in Figure 3.16 (b) and *Elephant* in Figure 3.16 (c). The scenes we have employed for our tests contains replicated and scaled versions of these models.

Table 3.1 summarizes the results obtained in terms of number of primitives. For the tests summarized in this table, $L_{max} = 3$ and three quality set of thresholds were employed: High, Medium and Low. To obtain this data and for each quality level, the thresholds for the three tests were selected to ensure a similar number

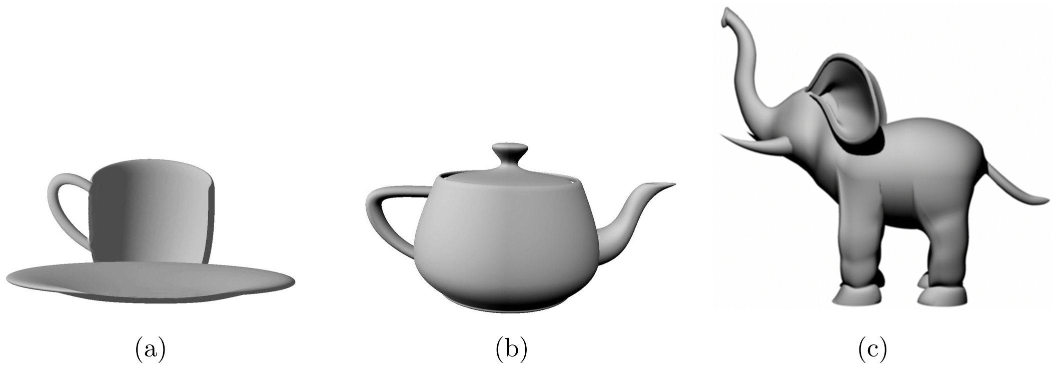


Figure 3.16: Models employed: (a) *Teacup*, (b) *Teapot* and (c) *Elephant*

of triangles for the tessellated models. The second column includes the number of Bézier surfaces per scene (each scene is composed by replicated and scaled versions of the model specified). The third, fourth and fifth columns include the average number of triangles generated with each test presented in Section 3.2.2. More specifically, the test based on the distance between mesh and surface is labeled as *Distance test*, the Vector deviation flat test as *Flat test*, while the test based on the length of the triangle coarse edges is labeled as *Length test*.

The tests we have performed indicate that the tessellation method produces high quality meshes with no visual artifacts. Additionally, the multi-resolution method employed and the use of multiple resolution areas per triangle have produced the expected good results. In Figure 3.17, a zoom shows the different tessellations obtained for the teacup model for the three tests: Distance, Flat and Length. To further evaluate and compare the results in terms of quality, the numerical error of the resulting meshes was also analyzed. We have computed the mesh error as an estimation of the distance between the real surface (approximated with a high resolution non-adaptive tessellation of the mesh) and the resulting mesh obtained through the adaptive technique. Figure 3.18 shows the mesh errors for a resulting scene (*teacup*), with a resolution level $L_{max} = 3$ for three different quality levels. As can be observed, and as expected, the best results are obtained with the Distance test. This is due to the fact that this test is performed per candidate position instead of per edge. Similar results are obtained with the Length test, while poorer results are obtained with the Flat test. Both tests are performed per edge and the results

		High Quality		
$Scene \backslash Test$	# surfaces	Distance test	Flat test	Length test
Crazy teacups	260	72.96 k	70.94 k	71.22 k
Crazy teapots	960	267.63 k	217.24 k	266.28 k
Elephant herd	8110	2034.082 k	1918.933 k	2266.391 k
		Medium Quality		
$Scene \backslash Test$	# surfaces	Distance test	Flat test	Length test
Crazy teacups	260	57.49 k	57.69 k	43.15 k
Crazy teapots	960	257.9 k	195.34 k	208.10 k
Elephant herd	8110	1986.02 k	1635.84 k	1781.87 k
		Low Quality		
$Scene \backslash Test$	# surfaces	Distance test	Flat test	Length test
Crazy teacups	260	15.1 k	14.49 k	15.44 k
Crazy teapots	960	135.79 k	126.40 k	142.83 k
Elephant herd	8110	742.64 k	497.44 k	559.94 k

Table 3.1: Number of triangles generated with the different tests presented and $L_{max} = 3$

indicate that for low degree surfaces the Length test gives a good estimation of the surface curvature.

The main objective of the proposal is the real-time rendering of complex models without reducing the quality of the final image. With the aim of testing the timing requirements of the application, we have analyzed a walk-through animation with the same movement of the camera for all tests. The final images have a screen resolution of 1280×1024 pixels. Figure 3.19 shows the results in fps for different tests with the two GPUs employed using a resolution level $L_{max} = 3$ for a high quality threshold. The results indicate highly satisfactory performances in terms of fps in both architectures, allowing real-time adaptive tessellation, even for a high number of triangles. For example, for the Length Test, 284.94 K triangles were rendered at 43.32 fps in the *Nvidia* and 148.97 fps in the *ATI*. Slightly better results were obtained for the Length and Flat tests with the *Nvidia* card due to the lower computational requirements as these tests are performed per edge. Note that in the *ATI* card, and for a large number of triangles, the Distance test has similar timing requirements to the other proposals. This is due to the exploitation of VLIW with the use of short vector data types (like $float4$) and vector computations.

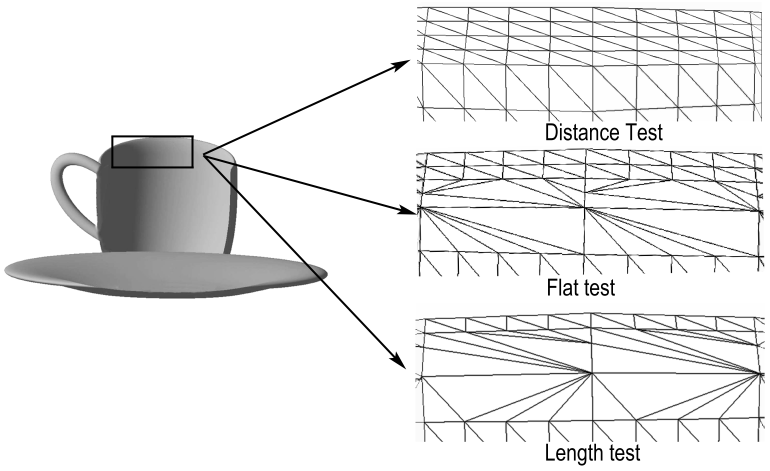


Figure 3.17: Zoom of the tessellated *teacup* model obtained with the three tests: Distance, Flat and Length

No comparisons with previous proposals are included as, to the best of our knowledge, this is the first analysis on fully adaptive tessellation of Bézier surfaces on current GPUs. There are a few recent proposals in the field but they offer reduced freedom in the adaptivity permitted. By way of example, in [34, 84] two proposals for the adaptive subdivision are presented; however, adaptivity is selected at the patch level. This means that once the level is selected, the patch is uniformly subdivided, with the exception of the border areas, where the tessellation has to be modified to avoid cracks between neighbor patches. In the field of triangle meshes there are proposals for adaptive tessellation based on the use of previous pre-computed patterns [62] so comparisons are not applicable.

The purpose of our implementation was to test the adaptive tessellation method employed, the evaluation of the tests proposed and evaluating the capabilities of current GPUs to process Bézier surfaces in real time. Owing to the good results obtained, the flexibility of the adaptive proposal and the simplicity of the computations involved, the proposal is a good candidate for integration as a specific tessellation unit in future graphics cards. As nowadays the specific tessellation units included in the graphics cards do not offer these adaptive characteristics, the algorithm was tested by exploiting current geometry shader capabilities. We hope that the results obtained will encourage the inclusion of more flexible tessellation units in future

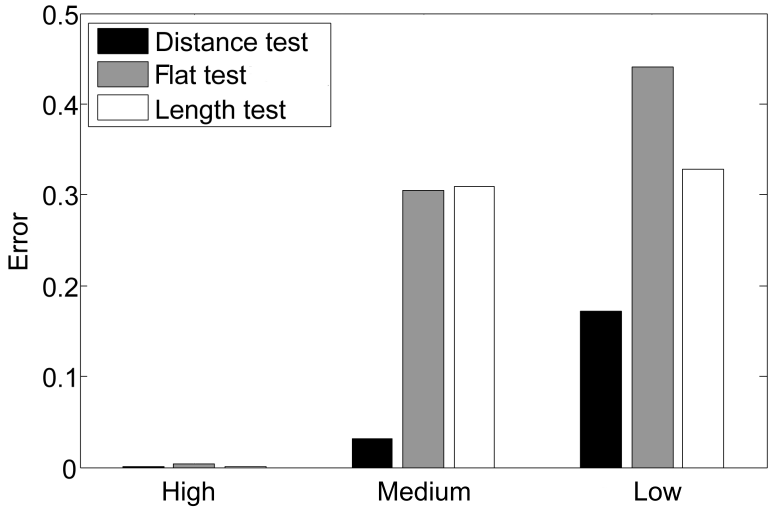


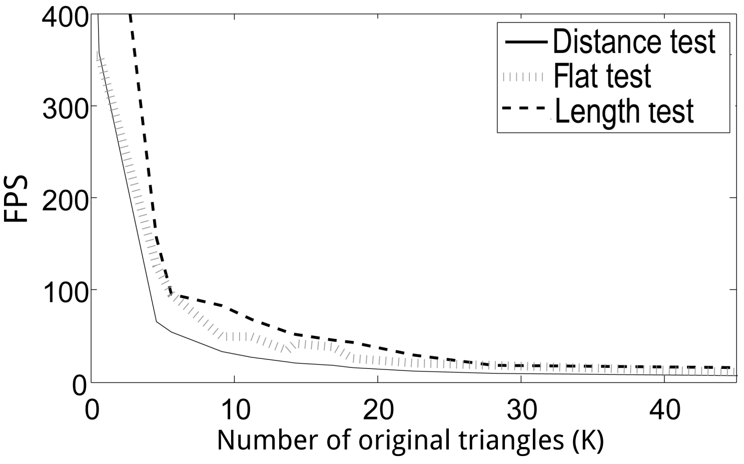
Figure 3.18: Error obtained with the *teacup* model for $L_{max} = 3$ and three different quality levels

graphics cards.

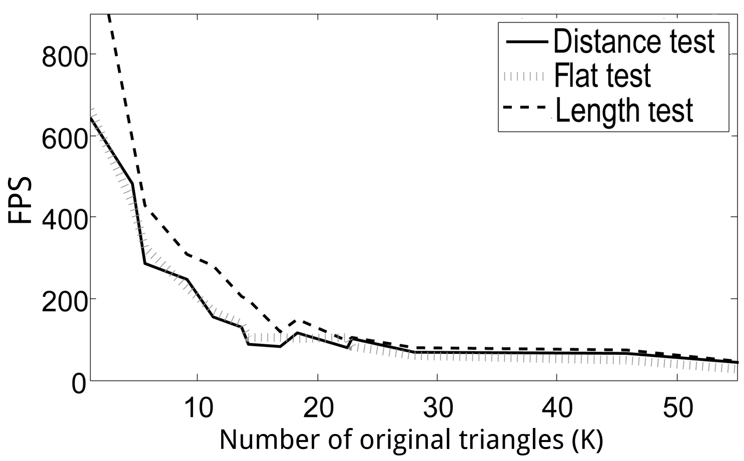
3.5. Conclusions

In this chapter we present a new method, Dynamic and Adaptive Bézier Tessellation (DABT), for the real-time adaptive tessellation of Bézier surfaces on the GPU. The method is based on the generation of a initial coarse triangle mesh that approximates the Bézier surface and the adaptive tessellation of each resulting triangle in the GPU. The methodology employed allows multiple resolutions to be applied to the same Bézier surface. This means that neighbor triangles can be processed with different resolutions and no visual artifacts are assured.

Our proposal does not employ any pre-computed pattern stored in memory as in previous proposals. Inserted vertices are calculated on the fly according to a non-adaptive tessellation pattern that guides the tessellation. The resulting inserted vertices are organized and classified into strips. The management of this simple and efficient representation permits the generation of the triangles associated to the adaptive tessellation in a direct way.



(a)



(b)

Figure 3.19: FPS with tessellation level $L_{max} = 3$ for a high quality threshold (a) Nvidia Geforce 295 GTX (b) ATI Radeon 5870

The proposal is based on three main strategies: the use of a fixed tessellation pattern to guide the procedure, the use of local tests for the adaptive tessellation decisions and an efficient meshing procedure to reconstruct the resulting meshes. With respect to the tests employed, in this work we have included three tests that analyze different surface features to guide the tessellation. More specifically, we include one test studying the distance mesh-surface, another based on the surface curvature analysis and a third one based on the edge length.

For algorithm testing purposes and to evaluate the capabilities of current GPUs, we have implemented our DABT algorithm by exploiting the geometry shader unit. On the other hand, and as expected, the capabilities of the technique are reduced owing to the limitations of the geometry shader. We consider that the introduction of the geometry shader with DirectX 10 was an important step forward and we hope that new generations will improve this unit and, with this, will encourage the use of adaptive tessellation techniques. Increasing interest in tessellation algorithms is acknowledged with the inclusion of specific tessellation units in DirectX 11. Taking this increased interest in tessellation units into account, we hope our proposal helps to guide the implementation of future tessellation units. The good results obtained in terms of quality and frames per second makes our proposal an interesting candidate for its real hardware implementation on future GPUs.

Chapter 4

Semi Adaptive Tessellation Proposal

This chapter presents an adaptive tessellation scheme based on a halfway approach between a non-adaptive tessellation scheme, GST, detailed in Chapter 2, and a fully adaptive proposal, DABT, presented in Chapter 3. Hence, this adaptive proposal is based on the use of different sampling strategies (adaptive and non-adaptive) and on the exploitation of the spatial coherence of data within each surface.

The non-adaptive method detailed in Chapter 2 is characterized by the simplicity of the associated shader program, with the ultimate aim of achieving the optimum use of the shader by reducing the divergence and irregularity of the program. The fully adaptive tessellation (see Chapter 3) proposal allows the number of triangles to be processed to be minimized to a given quality. The new semi adaptive approach presented in this chapter aims to better exploit the GPU by decreasing the degree of flexibility and adaptability. The objective is to reduce the irregularity and the associated divergence of the fully adaptive proposal to optimize the graphics hardware utilization, while generating the minimum number of triangles to achieve the required quality.

The main purpose of this work is to analyze novel solutions for improving the tessellation capabilities of the current rendering pipeline, rather than on developing an optimized implementation. Furthermore, this semi adaptive proposal introduces a variable degree of flexibility which entails a deep study of the tessellation and computation features of the geometry shader stage. The evaluation and comparison of a semi adaptive technique with respect to those techniques presented in Chapter 2 and 3 is performed. In any case, as shown in the results section, good results in term of quality and performance have being achieved with these methods. Additionally, these three methods are compared with the tessellation performed by the DirectX11 tessellation unit.

This chapter is organized as follows: in Section 4.1 a brief description of the main characteristics of the tessellation methods under analysis is presented. Section 4.2 describes the semi adaptive strategy. Section 4.3 presents the experimental results and the comparison between algorithms. Finally, the main conclusions are highlighted in Section 6.8.

This work has been submitted to [28].

4.1. Structure of the tessellation proposals

This section describes the structure of the tessellation methods under analysis. Figure 4.1 schematically depicts the structure of the proposals: a non-adaptive solution, GST, detailed in Chapter 2; a fully adaptive proposal, DABT, described in Chapter 3; and a novel semi adaptive approach with an intermediate degree of flexibility. For reasons of clarity, the figure shows the tessellation performed at triangle level.

One key of our approaches is the exploitation of the spatial coherence of data within each surface patch to take advantage of the constant result of $[N][B][M]^T$ for every point on the surface and that the different control points $[B]$ are accessed only once, whereas the current DirectX11 tessellation proposal accesses $[B]$ and computes $[N][B][M]^T$ for each sample on the surface.

The non-adaptive proposal is represented in the right part of Figure 4.1. The level of resolution of each object depends on the camera position and determines the refinement degree of the surface. The tessellation is performed by evaluating the parametric equation for uniformly distributed parametric values. The proposal

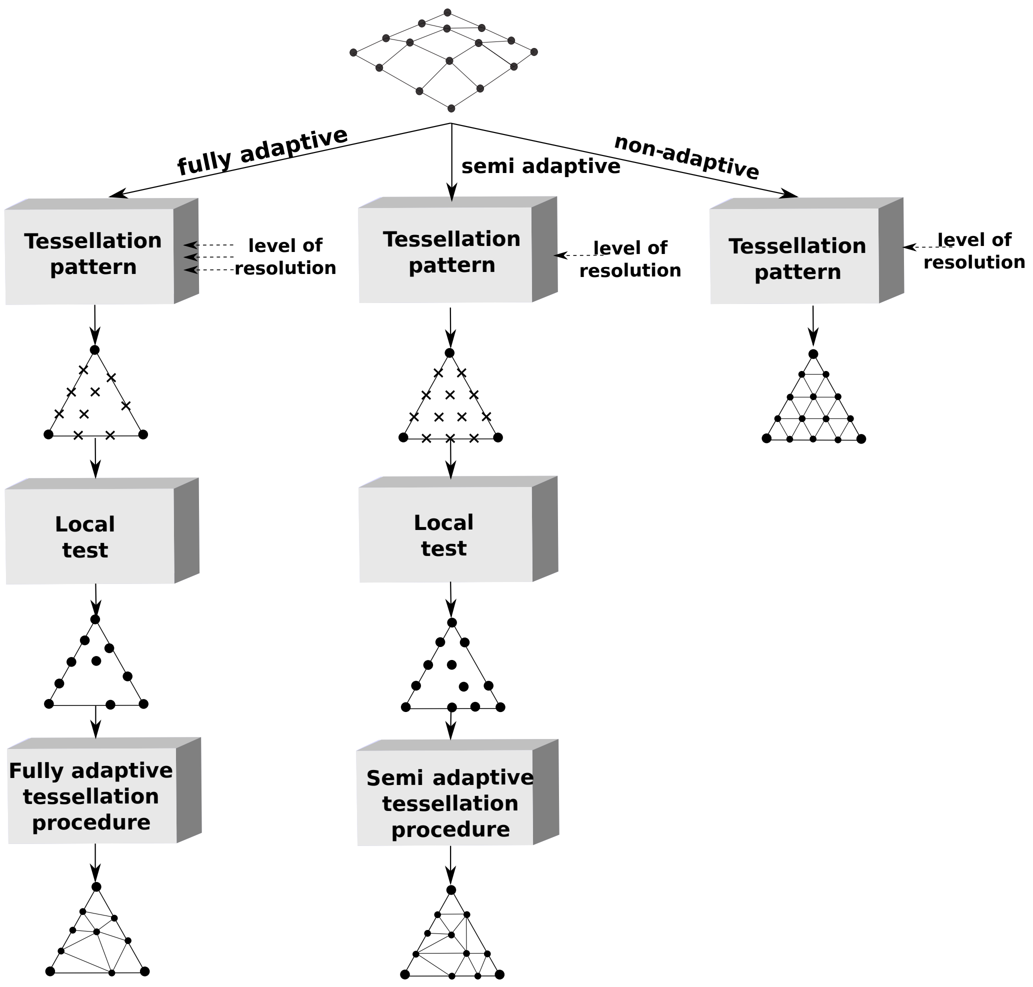


Figure 4.1: Structure of the tessellation algorithms.

under analysis is based on a strategy for increasing the number of output primitives per patch, which makes it possible to achieve high processing speed, as will be shown in the experimental analysis.

With respect to the fully adaptive and semi adaptive proposals (left part of Figure 4.1), the objective is to reduce the number of triangles of the final mesh while keeping the quality of the resulting image. Both algorithms are based on a 3-stage programmable pipeline: first, a fixed tessellation pattern is computed to guide the adaptive procedure for the patch; next, the new vertices obtained from the first step are conditionally inserted by applying a set of heuristics consisting of tests local to the triangle; finally, a specific scheme is employed to represent the inserted vertices and the reconstruction methodology based on the processing of this information.

The first step carries out a non recursive procedure based on the utilization of a fixed tessellation pattern to guide the tessellation. The tessellation patterns are employed to guide the adaptive tessellation in such a way that the new vertices can only be inserted into the candidate positions specified by the patterns. Once the level of resolution is selected only the positions of the uniform tessellation pattern are evaluated for their conditional insertion. While for the fully adaptive proposal different levels of resolution in the patch are permitted, for the semi adaptive proposal only one is employed to reduce complexity of the algorithm.

The second step decides which candidate vertices are really inserted as a result of a tessellation test. A local test is evaluated in order to guarantee that the same result will be obtained in several recomputations. Different local test has been considered (see Section 3.2.2); however, length test has been chosen for this analysis due to its simplicity and its low computational cost (see Section 3.4).

The last step of the adaptive tessellation algorithms is the reconstruction of the mesh from the set of vertices finally inserted; i.e., once the new vertices have been conditionally added, all the vertices (old and new) have to be organized and connected to reconstruct the final mesh, ensuring that there are no cracks or holes in the result (*Tessellation Procedure* in Figure 4.1). In the fully adaptive proposal, the new inserted vertices are organized into rows and represented as a set of lists. This representation is the key for the mesh reconstruction, as the final mesh can be extracted from this information easily and directly. For the semi adaptive proposal the meshing scheme is much simpler and a triangle strip is directly generated. The quality of the final triangle mesh is determined by both the inserted vertices and the reconstruction method employed to generate the resulting mesh.

4.2. Semi adaptive Tessellation Strategy

The semi adaptive strategy is a new tessellation proposal that represents an intermediate degree of flexibility between the non-adaptive strategy and the fully adaptive strategy. The objective of the proposal is to increase the processing speed of the fully adaptive algorithm by reducing its flexibility. The strategy aims to simplify the shader program as irregular shader programs (i.e., with an irregular control flow and branches) have an associated performance degradation [2].

The semi adaptive algorithm we propose is a simplified version of the fully adaptive strategy. The semi adaptive strategy is characterized by a unique level of resolution per triangle and its use of a regular grid pattern in the (u, v) parametric directions. Similarly to the non-adaptive proposal, this allows the GPU vector computation capabilities to be exploited. Furthermore, while for the fully adaptive algorithm the tests can be performed per candidate vertex, in this semi adaptive algorithm the tests are only applied in the candidate positions located in the original edges of the coarse triangle. Finally, if a vertex is inserted in the edge, the insertion is also performed along the row in all the candidate positions inside the triangle.

An example of the vertex insertion procedure is depicted in Figure 4.2. Figures 4.2(a) and 4.2(b) show examples of tessellation patterns for the fully adaptive and semi adaptive proposals, respectively. In the first case three resolution areas are employed: $L = 2$, $L = 3$ and $L = 1$ for the areas of edges a, b and c respectively. In the second case a single level of resolution $L = 3$ was selected. Figures 4.2(c) and Figures 4.2(d) depict the result of tessellation once the insertion decisions have been performed for the fully adaptive and semi adaptive algorithms, respectively. For comparative reasons, a similar set of decisions were assumed in both examples. While in the first sub-figure the test was applied in all the candidate positions, in the second case it was only applied on the original triangle edges. As will be detailed in this section, this different test application procedure reduces the complexity not only of the test phase but also of the meshing scheme employed to reconstruct the final triangle mesh.

Once the insertion decisions have been taken a meshing procedure to connect the vertices is performed. The semi adaptive algorithm we propose also follows a row strategy in which triangles are generated by connecting vertices in two consecutive rows of vertices. Similarly to the fully adaptive proposal, larger triangles connecting vertices in non-consecutive rows have to be generated in those locations where vertices are not inserted.

The resulting meshing scheme has a low complexity in comparison with the fully

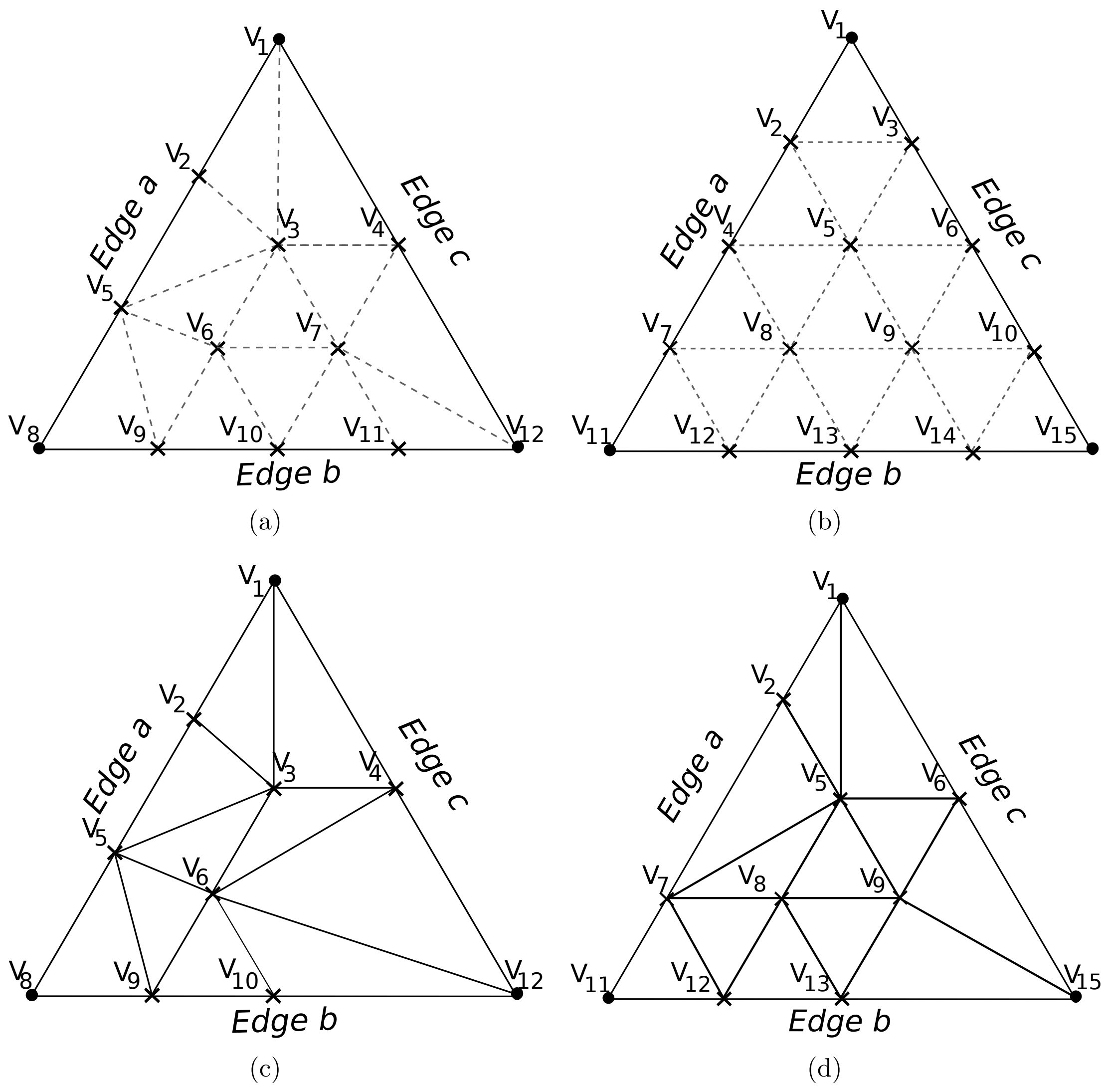


Figure 4.2: Adaptive tessellation (a) Fully adaptive pattern (b) Semi-adaptive pattern (c) Fully adaptive tessellation and (d) Semi-adaptive tessellation.

adaptive proposal. Similarly to the fully adaptive algorithm, once the subdivision tests are performed, the resulting inserted vertices are organized into a set of lists. The efficient management of this information permits the reconstruction of the final mesh in a direct way. While in the fully adaptive proposal the representation is based on rows of vertices, in the semi adaptive proposal the representation can be directly performed as Triangles Strips (TS). A triangle strip is defined by a sorted list of vertices:

$$TS = \{v_1, v_2, \cdots, v_{Nt}\}$$

where each triangle is defined by three sequential vertices, with an overlap of two vertices between two consecutive triangles. As an example the $i - th$ triangle is defined by:

$$\triangle v_i v_{i+1} v_{i+2}$$

while the $(i + 1) - th$ triangle is defined by:

$$\triangle v_{i+1} v_{i+2} v_{i+3}$$

The basic idea of the representation and reconstruction algorithm is the organization of the resulting the triangles in rows and their representation as triangles strips. The regular triangle strip structure is broken only in those positions where no vertices are inserted. This happens either in a position in the edge of the coarse triangle or in a full row of vertices if both vertices in the extreme positions are missing. For the sake of clarity, we shall start by analyzing the case when only one of the two extreme vertices of the row are missing.

The methodology we propose is based on the utilization of the TS structure that would be generated with a non-adaptive tessellation as basis for the representation. This TS list is updated with the utilization of a *Virtual Vertex* (VV) for the substitution of missing vertices. More specifically, each non-inserted vertex on the edge of the triangle is substituted, in the TS representation by the closest vertex located in the same edge. A simple set of rules for updating the TS structure when missing vertices appear are applied:

1. Missing vertex. If there is a non-inserted vertex in a triangle edge, the VV is the vertex located on the same edge and in the following row of vertices.

2. Group of missing vertices. If there is a group of adjacent non-inserted vertices on a triangle edge, each non-inserted vertex is replaced in the TS representation by the nearest vertex on the edge. In case of equidistant vertices, the vertex in the lower row is selected.

3. Replicated vertices in the TS list. Once the missing vertices are replaced by

virtual vertices, replicated vertices can appear in the TS list. These replicated vertices have to be eliminated; i.e., $v_i\ v_i$ is substituted by v_i.

4. Vertices from alternating rows. In the non-adaptive partitioning, consecutive vertices in the TS list belong to two rows of vertices. For the semi adaptive proposal, when this regular structure is broken a modification has to be performed. Let us analyze a TS list with three consecutive vertices, $v_i\ v_j\ v_k$, where v_j and v_k come from the same row of vertices. In this case the TS list is updated by including a replicated version of v_i in between. As result the list of vertices becomes $v_i\ v_j\ v_i\ v_k$.

An example of application is depicted in Figure 4.3(a). A non-adaptive tessellation would generate three rows of triangles, each one to be represented with at TS list. The TS lists for the non-adaptive tessellation are:

$$\begin{aligned} TS_1 &= \{v_2\ v_1\ v_3\} \\ TS_2 &= \{v_4\ v_2\ v_5\ v_3\ v_6\} \\ TS_3 &= \{v_7\ v_4\ v_8\ v_5\ v_9\ v_6\ v_{10}\} \end{aligned} \tag{4.1}$$

As result of the semi adaptive tessellation, the vertex v_6 is not inserted. The TS lists are updated by the following steps:

- The missing vertex v_6 is replaced by a virtual vertex $VV = v_{10}$. This is the closest vertex located in the same triangle edge and in the following row of vertices. As a result the TS_2 and TS_3 lists are updated.
- The utilization of v_{10} as virtual vertex in the TS_3 list generates a replicated vertex. According to rule 3, the list $TS_3 = \{v_7\ v_4\ v_8\ v_5\ v_9\ v_{10}\ v_{10}\}$ becomes $TS_3 = \{v_7\ v_4\ v_8\ v_5\ v_9\ v_{10}\}$.
- With respect to the alternating rows property and according to rule 4, the list $TS_3 = \{v_7\ v_4\ v_8\ v_5\ v_9\ v_{10}\}$ becomes $TS_3 = \{v_7\ v_4\ v_8\ v_5\ v_9\ v_5\ v_{10}\}$.

Finally the triangle strips for this example are:

$$\begin{aligned} TS_1 &= \{v_2\ v_1\ v_3\} \\ TS_2 &= \{v_4\ v_2\ v_5\ v_3\ v_{10}\} \\ TS_3 &= \{v_7\ v_4\ v_8\ v_5\ v_9\ v_5\ v_{10}\} \end{aligned}$$

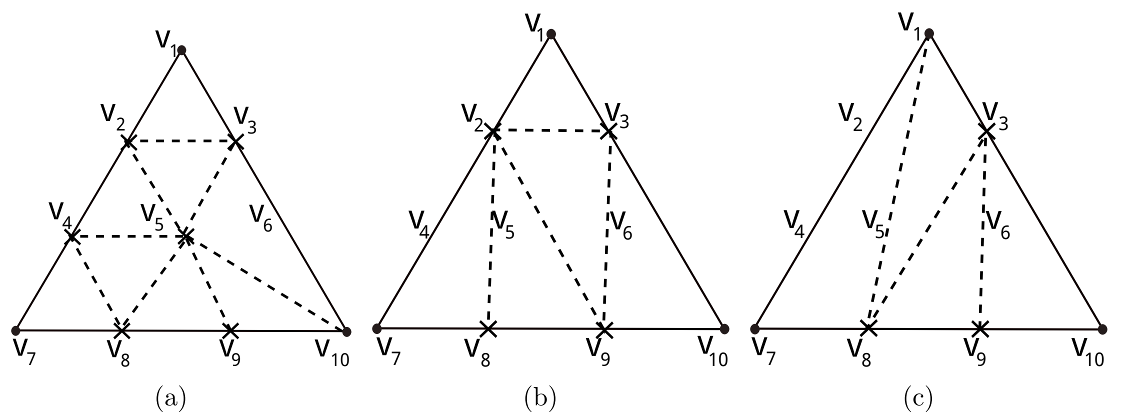


Figure 4.3: Examples of semi adaptive tessellations (a) No empty rows (b) Empty row, upper row with no missing vertex (c) Empty row, upper row with a missing vertex.

and the following triangles are generated:

$$
\begin{aligned}
&TS_1 \rightarrow \triangle v_2 v_1 v_3 \\
&TS_2 \rightarrow \triangle v_4 v_2 v_5 \triangle v_2 v_5 v_3 \triangle v_5 v_3 v_{10} \\
&TS_3 \rightarrow \triangle v_7 v_4 v_8 \triangle v_4 v_8 v_5 \triangle v_8 v_5 v_9 \triangle v_9 v_5 v_{10}
\end{aligned}
$$

The methodology has to be extended to include those situations in which both vertices in the extreme positions of a row are missing. As the insertion decisions are applied to the interior vertices in the same row, this implies that a complete row of vertices is missing. In this case fewer triangle strips are generated and a number of modifications have to be made to the method explained above. The first step is to update the TS lists of the non-adaptive case by identifying the two TS lists affected and eliminating the first one. After this the second list is updated by substituting the missing vertices by other vertices in the closest non-empty row of vertices located above. More specifically the substitution has to be performed by adhering to the following rules:

1. Upper row with no missing vertices. If the row above is complete, the vertices are directly employed to substitute the eliminated vertices in the TS list. As the number of vacancies is larger than the number of vertices, the vertices have to be replicated. To obtain a satisfactory tessellation and to prevent the

generation of large triangles, the pattern of substitution/replication has to be uniform.

2. Upper row with a missing vertex. If the row above has a missing vertex, this location will be occupied by a VV. The row of vertices is also employed to update the TS list under construction, but a number of considerations have to be taken into account for the VV. The vacancies are again covered by the vertices in the row, but the VV vertex can be employed only once; i.e., the VV cannot be replicated.

Examples of application are depicted in Figures 4.3(b) and 4.3(c). In these examples the row of vertices $\{v_4, v_5, v_6\}$ has not been inserted. The TS lists generated by the uniform tessellation (see Equation 4.1) have to be updated. In this case lists TS_2 and TS_3 are affected by the missing row and the list TS_2 is eliminated. List $TS_3 = \{v_7\ v_4\ v_8\ v_5\ v_9\ v_6\ v_{10}\}$ has to be updated by identifying the missing vertices (v_4, v_5 and v_6) and replacing them with the list of vertices above. Specifically, and for the example of Figure 4.3(b), the vertices to be employed are v_2 and v_3. The number of vacancies is larger, so the first two vacancies are covered with vertex v_2 and the last one with vertex v_3. The list becomes $TS_3 = \{v_7\ v_2\ v_8\ v_2\ v_9\ v_3\ v_{10}\}$.

Figure 4.3(c) shows an example where the row of vertices to be employed has a missing vertex (v_2). Following the methodology explained abouve, this vertex is substituted by a virtual vertex, in this case by v_1. This virtual vertex is employed only once while vertex v_3 is employed for the other two vacancies. So the updated list is $TS_3 = \{v_7\ v_1\ v_8\ v_3\ v_9\ v_3\ v_{10}\}$.

4.3. Experimental Results

In this section, the results of the evaluation of the different GPU tessellation methods described in the paper (non-adaptive, fully adaptive and semi adaptive subdivision) are presented and analyzed. The three proposals were coded in the Geometry Shader as this makes it possible to implement a free tessellation algorithm, even though it has the important constraint of limiting the maximum number of new primitives to be generated per input primitive to 1024 32-bit elements. Additionally, we compare them with the tessellation implemented by the DirectX11

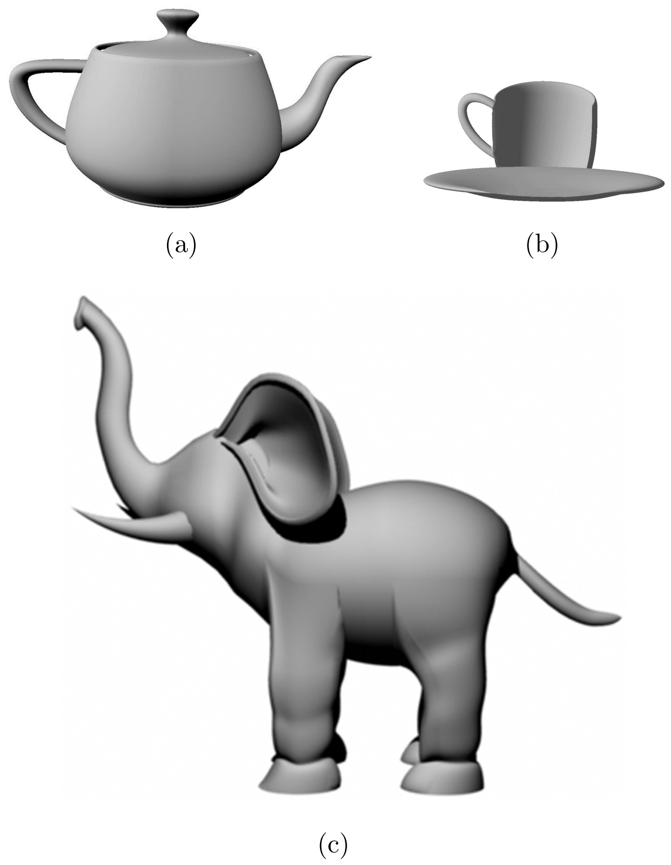


Figure 4.4: Models employed: (a) *Teapot* (b) *Teacup* and (c) *Elephant*.

tessellation unit. Specifically, we have used the code *SimpleBezier11* included in the DirectX11 SDK with two different tessellations, a simple uniform approach (*Tess*) and an adaptive solution (*AdptTess*) based on a distance test (as presented in Section 3.2.2) computed in the Hull Shader.

All the algorithms were implemented with Microsoft's HLSL DirectX11, and the tests were run on an Intel Core 2 2.4 GHz with 2 GB of RAM and three different GPUs: AMD/ATI Radeon 5870 (*ATI 5870*), GPU with 1600 processing elements distributed in 20 SIMD processors, each one having 16 cores with 5-way VLIW support; AMD/ATI Radeon 6970 (*ATI 6970*), with 1536 processing elements distributed in 24 *SIMD* processors, each one with 16 cores with 4-way VLIW support; and Nvidia Geforce GTX 580 (*Nvidia 580*), GPU based on the Fermi architecture that has 4 clusters, with 4 stream multiprocessor (SM) per cluster and 32 stream processors per SM for a total of $4 \times 4 \times 32 = 512$ physical processing elements.

Three different models were employed to evaluate the tessellation (see Figure 4.4): *Teapot*, *Teacup* and *Elephant*. These models were used to build three test scenes, *Teapots*, *Teacups* and *Elephants*, that consist of replicated versions of the models: 30, 100 and 10 models, respectively. To check the performance of the implemented methods a walk-through animation with the same movement of the camera for the three scenes was performed. The final images have a screen resolution of 1280×1024 pixels.

The next section focuses on the analysis of the experimental results in three different key points: the analysis of the quality in the final image, the performance in terms of frames per second for a medium degree of tessellation and, finally, the performance for a high degree of tessellation. As the tessellation scheme for high and medium degree has a different behavior, the analysis in terms of frames per seconds is split it up into two different subsections.

4.3.1. Performance in terms of quality

As a starting point of the analysis of the results from the tests, Table 4.1 presents the number of generated primitives for each method. In this table the maximum resolution employed is $L_{max} = 3$ (16×18 triangles for each input surface). The second row indicates the number of input Bézier surfaces per scene. The third row shows the number of triangles generated when a non-adaptive tessellation is applied. The rest of the rows show the average number of triangles generated by the semi adaptive and fully adaptive proposals. In both cases the *Distance test* was applied with three different thresholds on the basis of a quality criteria: high, medium or low degree of tessellation. The percentage of triangles obtained for each case with regard to the non-adaptive strategy is shown in parenthesis.

In our experiments, high quality meshes with no cracks or holes are obtained for all the methods. Obviously, the application of the adaptive proposals gives rise to a reduction in the number of primitives generated, where the decrement can be controlled by the adequate selection of the quality threshold applied. As an example of the tessellation obtained, Figure 4.5 shows the result of applying the

Table 4.1: Number of triangles generated (in thousands, $L_{max} = 3$) and number of input surfaces.

		Teapots	*Teacups*	*Elephants*
Input Data		960	2600	8110
Non Adaptive		270.00	731.32	2280.94
Semi adap- tive	**High**	263.58 (97.62%)	684.67 (93.62%)	1941.03 (85.10%)
	Medium	149.30 (55.30%)	379.98 (51.96%)	1155.70 (50.68%)
	Low	67.68 (25.07%)	197.85 (27.05%)	831.79 (36.46%)
Fully adap- tive	**High**	255.14 (94.50%)	680.21 (93.01%)	2112.71 (92.62%)
	Medium	142.73 (52.86%)	385.13 (52.66%)	1158.83 (50.80%)
	Low	59.73 (22.13%)	178.50 (24.41%)	519.51 (22.73%)

three different tessellation techniques to the *teacup* model with $L_{max} = 3$ and for a medium degree of subdivision. As can be observed, significantly fewer primitives are generated in the flat areas for the adaptive approximations. In these flat areas the coarse mesh is a good enough approximation to the Bézier surface, so introducing additional primitives do not result in a higher quality of the image for the quality threshold selected.

To further evaluate and compare the results in terms of quality, the numerical error of the resulting meshes is also analyzed. Thus, in order to estimate the distortion due to the adaptive tessellation we have computed the mesh error as the distance between the non-adaptive tessellation of the mesh and the resulting mesh obtained through the fully adaptive and semi adaptive techniques. Since a static object-space geometric error metric would not properly consider the point of view and perspective of the rendering, an image-space error metric has been used to determine the influence in the render quality of the non-inclusion of a candidate vertex for a given viewpoint.

More specifically, the error measure used in this work is defined by:

$$\varepsilon = \frac{|V_s - V_B|}{d_v - r_v} \tag{4.2}$$

where V_B and V_S are the coordinates of the candidate vertex on the Bézier surface and the corresponding sampling point on the control mesh, respectively (see

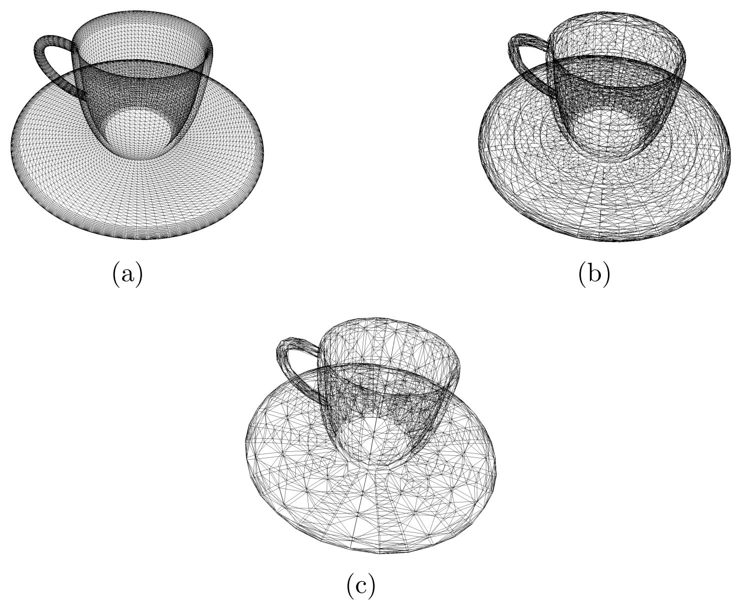


Figure 4.5: Examples of tessellation: (a) *Non adaptive* (b) *Semi adaptive* (c) *Fully adaptive.*

Equation 3.2.2); and d_v is the distance from the observer location to V_S. As can be noted, a bounding sphere with a conservative radius r_v is added to the expression to consider the influence of the relative triangle size, since the non-inclusion of a vertex with distance $|V_s - V_B|$ in a large triangle in a far away region should produce a higher error than a small triangle to the same distance. A similar metric is used in the context of multi-resolution models for interactive terrain rendering [75].

Figure 4.6 compares the quality of the semi adaptive and fully adaptive proposals. The graph of the figure shows the mean absolute error for the *Teacups* test scene with a maximum resolution level $L_{max} = 3$ for the same three quality levels of Table 4.1 and using low d_v values (approximately 70% of the screen is covered by the model).

As can be observed, and could have been expected, the best results were obtained with the semi adaptive proposal. The resulting tessellation patterns for the semi adaptive proposal are closer to the non-adaptive patterns. Consequently, an immediate benefit of reducing the flexibility in this proposal is a higher quality in the resulting meshes. In any case, both adaptive methods produce high-quality results comparable with the non-adaptive approximation, as Figure 4.6 proves.

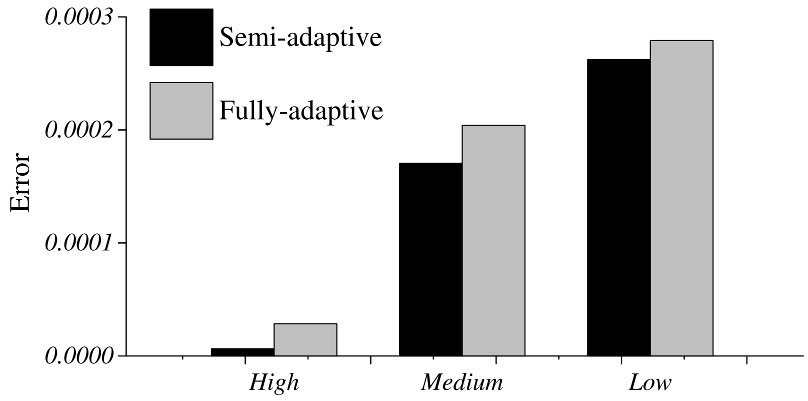
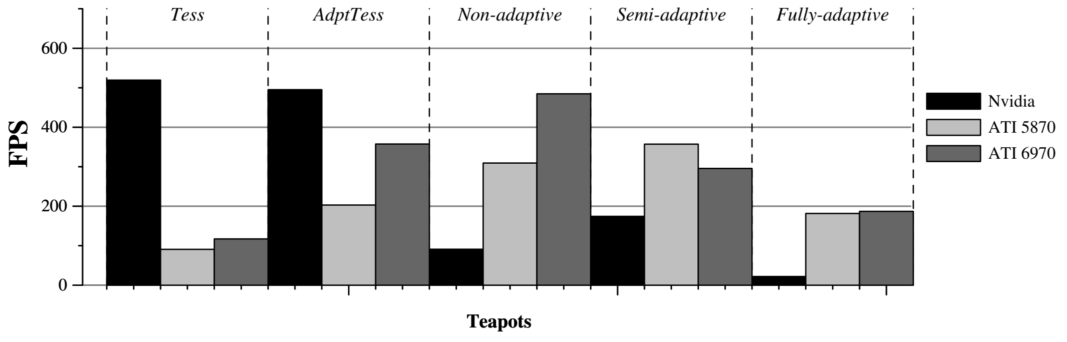


Figure 4.6: Mean absolute error obtained for the *Teacups* scene with $L_{max} = 3$ for the fully adaptive and semi adaptive proposals.

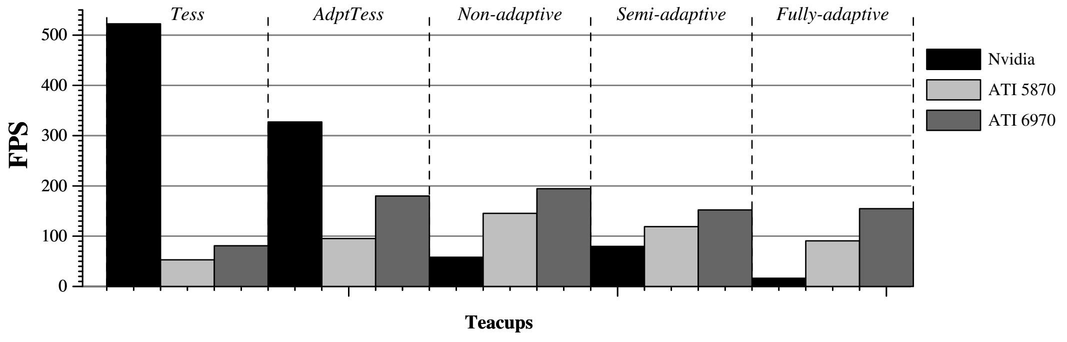
4.3.2. Performance in terms of fps for a medium degree tessellation

Performance in term of fps is another important aspect to be analyzed. Figure 4.7 shows the frames per second (fps) with three different GPUs for a medium-quality degree tessellation (over 50% of triangles of a non-adaptive tessellation, as shown in Table 4.1). The columns labeled as *Tess* and *AdptTess* display the performance obtained by the two *SimpleBezier11* methods implemented using the tessellation unit. Both alternative methods are based on the DirectX11 tessellation unit (see Section 1.1). *Tess* follows a non-adaptive tessellation in the Tessellator where all the tessellation factors have the same value. However, *AdptTess*, is also tessellated in the Tessellation although a certain degree of adaptively is introduced. In both alternatives, the tessellation factors have been selected to generate a number of triangles similar to those generated by the *Non-adaptive* and *Semi-adaptive* proposals, respectively, with a difference of less than 1 K for the non-adaptive case and less than 2 K for the adaptive one.

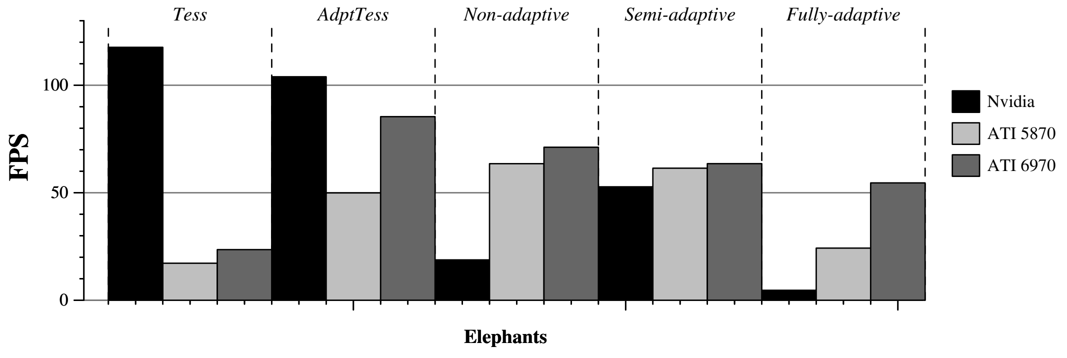
As can be observed, the performance values obtained are quite sensitive to the GPU architecture. Although a comprehensive analysis of the tessellation units in the different architectures is beyond the scope of this work, we will just mention some relevant issues that have an influence on the description of our proposals. The best performance results are achieved by the *Tess* proposal on the *Nvidia* card. On the whole, since the *Nvidia* platform has a larger number of tessellation units than the



(a)



(b)



(c)

Figure 4.7: Processing Speed in Frames per Second ($L_{max} = 3$) (a) *Teapots* (b) *Teacups* (c) *Elephants*.

ATI hardware, a higher patch tessellation rate is achieved, so the global performance of the *Tess* and *AdptTess* implementations on the *Nvidia* GPU is greater than on the two *ATI* GPUs. The reason why *AdptTess* performs worse than *Tess* in *Nvidia* is the competition in the processing units of the GPUs between the Hull Shader, which computes the tessellation factors for each edge according to the *Distance Test*, and the Domain Shader, which evaluates the position of each candidate point to be inserted. Nevertheless, in the case of ATI GPUs there is a completely different tendency, as the higher number of processing elements in this architecture means a greater performance for the *AdptTess* solution.

Regarding the three proposals analyzed in this chapter (non-adaptive, fully adaptive and semi adaptive) Figure 4.7 shows that good performance results in terms of frame rate (fps) were obtained, and a real-time adaptive tessellation was achieved even for a high number of triangles. Thus, for instance, the high quality result of the *Elephants* scene is achieved with a frame rate of 71.16 fps for the *non-adaptive* proposal on the *ATI 6970*, and 52.75 fps in the *AdptTess* on the *Nvidia*.

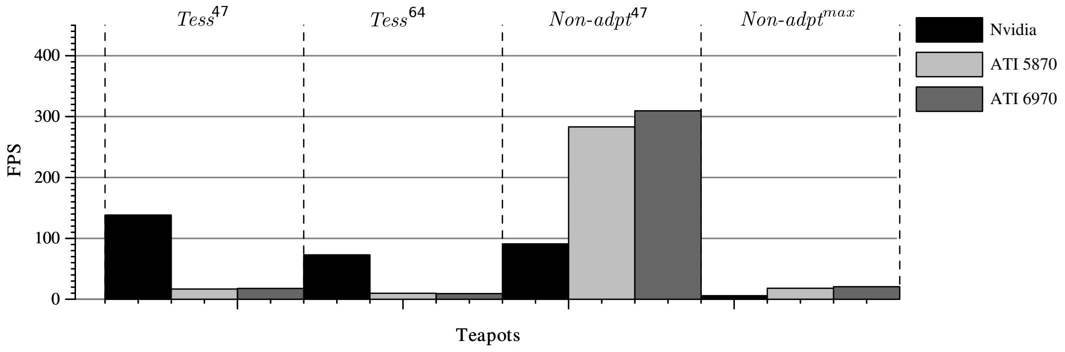
These three proposals are a representative set of the different approaches to the tessellation of parametric surfaces, since they provide different degrees of adaptability and flexibility which results in a quality/performance trade-off. An analysis of the experimental results enables us to study the reasons behind the difference in performance for each alternative, and to study their feasibility on current hardware. Thus, regarding the impact of one of the main performance penalty in current GPUs, branch divergence, the *non-adaptive* proposal is branch-free but needs two steps through the stream-output of the DirectX11 pipeline, which means a synchronization barrier. Meanwhile, the *semi adaptive* approach has a slight branch divergence, whereas this penalty factor has a great impact in the *fully adaptive* method. Looking at the results on the ATI GPUs, branch divergence has a significant impact on the performance, so the *fully adaptive* and the *non-adaptive* implementations (the latter is also optimized for a VLIW architecture) gets the worst and the best performance, respectively. Let us emphasize that branch divergence has less impact on *ATI 6970* where the performance of the *semi adaptive* and *fully adaptive* proposals are quite close. In short, out of the three methods mentioned here, the *non-adaptive* proposal obtains the best performance on ATI GPUs, mainly due to a much more optimized Geometry Shader and stream-output facility in this platform, together with the exploitation of the VLIW architecture of these GPUs.

The analysis of the performance on the *Nvidia* card causes us to come to a different conclusion with respect to the three proposals analyzed in this chapter. In this platform, the *semi adaptive* proposal achieves the best results, given that the performance dramatically decreases in this architecture when the stream-output facility is used [21]. Since the *semi adaptive* method passes only once through the GPU pipeline, i.e. the stream-output is not used, whereas three passes are needed in the *non-adaptive* proposal, this results in a substantial performance gain. Furthermore, the *non-adaptive* implementation based on vector optimization is not suited to the scalar architecture of the *Nvidia* GPU.

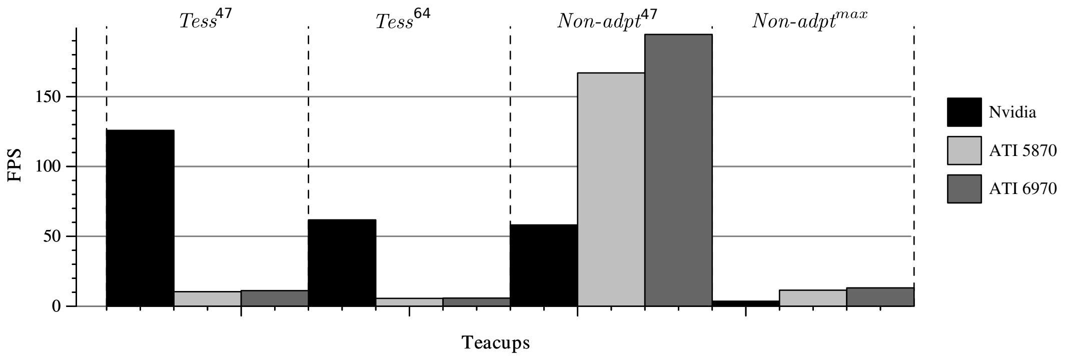
The three proposals analyzed in this work demonstrate better computational exploitation than the tessellator-based alternatives, since one computing core (shader) is used for each input primitive, instead of the one core per output primitive ratio of the DirectX11 tessellator-based proposals. As a result, an important feature of our approach is the exploitation of the spatial coherence of data, as shared common computations within the same patch are computed only once and reused when needed. Furthermore, from among the three methods the branch-free algorithm of the *non-adaptive* solution provides superior performance than the methods based on the DirectX11 tessellation unit, *Tess* and *AdptTess*. However, when a really large number of triangles is generated (see *Elephants* scene) the amount of data stored on the stream-output does not compensate for those advantages, due to the increase in read/write times, as depicted in Figure 4.8. As we go on to analyze in the following paragraph, if the second kernel of the *non-adaptive* method (with the GS implementation described in Chapter 2) generates a high number of primitives the performance of the branch-free iterative process has an important drop.

4.3.3. Performance in terms of fps with a high degree of tessellation

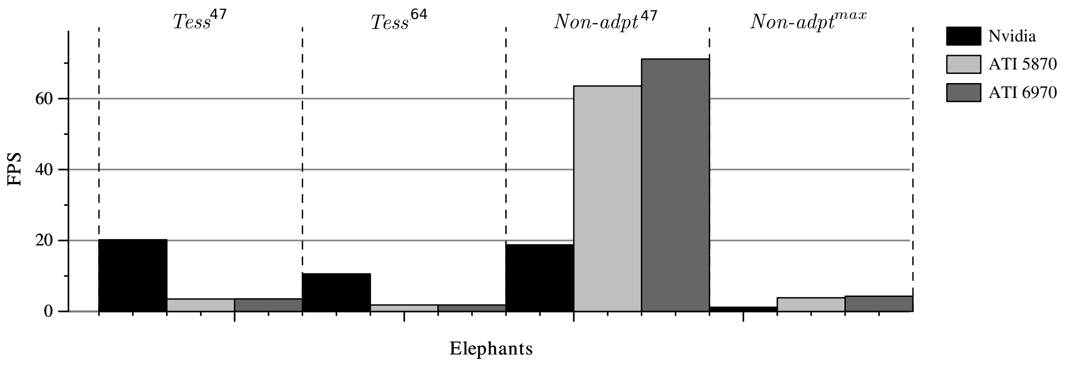
Figure 4.8 analyzes the impact on performance of the level of tessellation by showing the frame rate for four different options: *Tess*^{47} and *Tess*^{64} are the results obtained by the *Tess* implementation based on the DirectX11 tessellation unit, where the tessellation factor is 47 or 64 triangles for each input primitive, respectively (the



(a)



(b)



(c)

Figure 4.8: Frame rate using higher tessellation factors (a) *Teapots* (b) *Teacups* (c) *Elephants*.

latter corresponding to the maximun tessellator factor of the tessellator stage, see Section 1.1); whereas *Non-adpt*^{47} and *Non-adpt*^{max} result from the *non-adaptive* proposal, using in the first case the tessellation level necessary for obtaining with our proposal a similar number of triangles to those obtained with *Tess*^{47} and the maximun tessellation factor possible that can be implemented in our *non-adaptive* proposal (taking advantage of the absence of limit in this software proposal). Table 4.2 shows the number of output triangles for each case.

In respect to *Tess*^{47} and *Non-adpt*^{47}, the best performance is obtained by our *non-adaptive* proposal on ATI GPUs, since the exploitation of the data locality and vector computation is emphasized. As an example, the frame rate of *Non-adpt*^{47} for *Teacups* with *ATI 6970* is 194.51 fps, whereas *Tess*^{47} with *Nvidia* results in 125.83 fps. *Tess*^{64} achieves real time rendering except for scenes with a high number of input primitives. However, in *Non-adpt*^{max} real-time rendering is even obtained (33.96 fps for *Teapots*) with 16.8 Mtriangles generated.

In summary, considering the good results obtained, the flexibility of the adaptive proposals, the exploitation of the locality and the prevention of redundancy computations, our proposals are good candidates to be integrated as a specific tessellation unit in future graphics cards, as nowadays the existing tessellation units included in current GPUs do not offer the desirable adaptability.

4.4. Conclusions

This chapter presents a proposal for the tessellation of Bézier surfaces on the GPU. It has been designed as a halfway solution between a non-adaptive tessellation scheme, GST detailed in Chapter 2, and a fully adaptive proposal, DABT presented

Table 4.2: Number of output triangles for the three test models (in thousands).

	Teapots	*Teacups*	*Elephants*
Tess^{47}	1088.13	2947.00	9192.39
Tess^{64}	2009.69	5442.90	16977.67
Non-adpt^{47}	1080.00	2925.00	9123.75
Non-adpt^{max}	17236.10	46800.00	145980.00

in Chapter 3. This semi adaptive proposal is based on the exploitation of the spatial coherence and it assigns a unique level of resolution per patch.

This tessellation scheme reduces the divergence in order to achieve an optimum utilization of the computational resources of the GPU; however, a remarkable degree of adaptivity has been introduced. Hence, this proposal processes considerably fewer triangles than a non-adaptive proposal, although the divergence caused by this adaptivity is reduced.

The purpose of the implementations was testing not only considering the exploitation of the spatial coherence proposal but also the behavior of DirectX11 tessellation unit. The behavior is quite sensitive to the GPU architecture. The tessellation unit obtains the best performance in the Nvidia architecture owing to the high number of tessellation units. On the other hand, the adaptive proposals obtain better performance on the ATI architecture, owing to the high number of processing elements and its better behavior with respect to the divergence, above all in the most recent ATI GPUS.

In addition to the quality and performance favorable results, we hope that the good results obtained, the flexibility of the adaptive proposals and the simplicity of the computations involved will encourage the inclusion of more flexible tessellation units in future graphics cards.

Chapter 5

Rendering of Bézier surfaces on Handheld Devices

This chapter describes the vertex shader Tessellation adapted to handheld devices (VSTHD) which has been designed according to the constraints of GPUs implemented on handheld devices. More specifically, this proposal is based on the virtual parametric grids and memory exploitation described in VST alternative (see Section 2.2). As handheld devices use GPUs with no hardware for generating primitives, no other tessellation proposal detailed in this dissertation can be implemented in these devices.

Handheld devices, such as smartphones, consoles or tablets, are widely available, virtually omnipresent devices and this is one of the fastest growing technology markets. Graphics processing has become a significant factor on handheld devices, as many contents in different platforms require high-quality visual contents, such as video player, TV, game console or camera. Hence, they are by far the most widely available device with rendering capabilities in the world [3, 13]. As consumers' expectations have increased, demanding complex rendering capabilities, a new GPU generation has been specifically designed to fit in the constraints of handheld devices. As handheld devices are small in size and they are battery powered, they have been designed according to a restricted set of features. Hence, the GPUs in these devices implement only a subset of the features available in desktop GPUs. Specifically, these GPUs have been designed to offer high performance graphics while reducing power consumption. Their GPUs implement a subset of the features available in desktop GPUs. Hence, a new graphic API called OpenGL ES [52] has been developed as a stripped-down version of the well-known graphics API as OpenGL.

In previous works, the tessellation of parametric surfaces directly on GPU of handheld devices were oriented toward GPUs with a lower degree of programmability. For this reason the tessellation was designed not to be programmed but to be implemented in an additional and specific hardware unit [18, 19, 53]. In [18, 19] tessellation limitations of the current hardware of GPU implemented in handheld devices are dealt with and an architecture for tessellating spline surfaces in a mobile multimedia processor is proposed. These proposals have been specifically designed for reducing memory access or for saving bandwidth and consequently reducing energy consumption. In [53] a hardware unit for an efficient tessellation in handheld devices was also proposed, but this proposal describes a tessellation procedure for subdivision surfaces.

In this work we present a novel approach to the tessellation of Bézier surfaces on the GPU of handheld devices. In contrast to previous proposals a generalization of our VST solution aimed at the new programmability capabilities available in current GPUs in handheld devices is proposed. Our proposal tessellates parametric surfaces into high-quality triangle meshes that accurately represent complex surfaces and contain no artifacts such as T-junctions or cracks. It is based on the utilization of a parametric maps of virtual vertices [11, 21, 47], which makes it possible to work on GPUs with no primitive generator. In addition, this design allows the efficient exploitation of the information stored in the GPU and the minimization of the CPU-GPU communications. Three main parameters are exposed to allow a fine tuning of the method to the hardware resources available: maximum resolution level, number of surfaces to be rendered per draw call and number of draw calls per frame.

In order to test our approach, we have made an OpenGL ES implementation of the method for Android systems [45] and we have designed a full set of experiments to analyze its performance. The tests were focused on locating the main performance bottlenecks and identifying possible enhancements and tuning opportunities. Thus, the results obtained could be a useful tool to introduce architecture improvements.

The rest of the chapter is organized as follows: Section 5.1 details hardware features and introduces OpenGL ES API; in Section 5.2 our approach to tessellate Bézier surfaces on handheld devices is presented; Section 5.3 the implementation on Android smartphones with OpenGL ES is detailed and in Section 5.4 the experimental results obtained are described and analyzed; finally, in Section 5.5 the main conclusions are highlighted.

5.1. GPU Architectures on Mobile Devices

There is a gread deal of variation in hardware features on handheld devices, such as memory bus bandwidths, cache memories, etc.; thus a wide range of different performance profiles are available. Furthermore, each handheld device architecture has a different GPU hardware, which has a significant impact on the performance.

Two constraints are considered in the design of handheld devices, their size and the fact that they are battery powered. The size of these devices is quite small in order to be portable and they are battery-driven platform [4]. Hence, the components that can be fitted in these devices are limited and consequently handheld devices provides a restricted set of features, such as a limited memory, a limited memory bandwidth, a restricted CPU instruction set or a low clock frequency.

A handheld processor is usually implemented as a system on chip (SoC). As components, such as CPU or GPU, are integrated into a single chip, they compete for the shared bus, which becomes the main bottleneck in the chip. Furthermore, the computational power of GPUs is usually under-utilized, owing to the fact that memory accesses are very expensive in terms of computation thus an off-chip memory access takes tens of cycles while an on-chip memory access takes only one cycle [53]. However, off-chip memory accesses have a higher impact in terms of power than in terms of computation [43] as each component drives high capacitance for the buses, which has a considerable cost in terms of energy. Parametric surfaces are a suitable proposal in this architecture due to the fact that, as they provide a more compact representation, the CPU-GPU communication is diminished.

As the power for handheld devices is supplied by batteries, it is important to reduce their energy consumption to provide a long-use time of battery, thus the clock frequency is kept as low as possible as higher clock frequency means higher

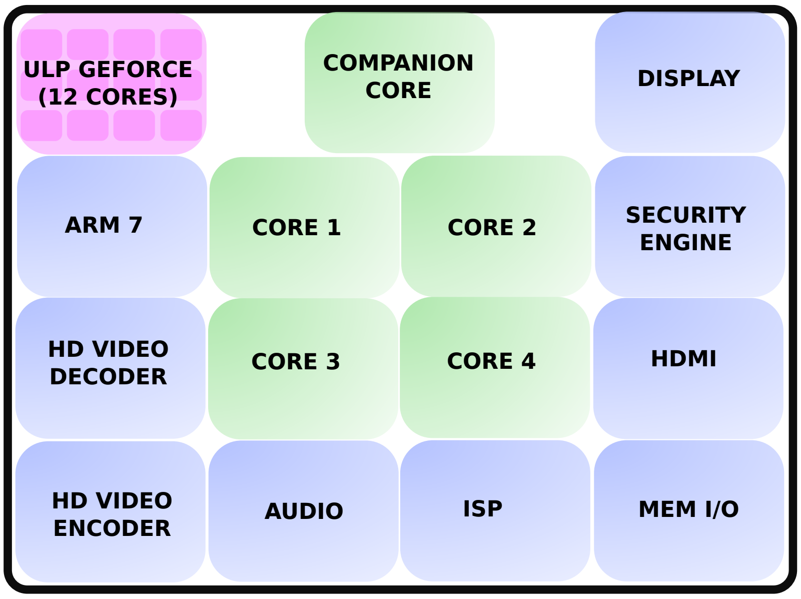


Figure 5.1: Tegra 3 structure

power consumption [8].

Nowadays, the screen and the communication subsystems are components with the highest consumption and their consumption is significantly increased with each new generation [14]. Meanwhile, the trend is to reduce as far as possible the consumption of all the other components, such as CPU or GPU. Nevertheless, as screens are continuously increasing in size and resolution, and a better screen requires a better graphics render, GPU performance is quite relevant for these devices.

More specifically, there are principally three different GPU architectures: immediate mode rendering (IMR), tile base rendering (TB) and an trade-off solution called tile based immediate mode rendering (TBIMR).

Traditional IMR is a brute force approach designed for the desktop environment, where bandwidth is plentiful and power consumption is largely irrelevant. IMR is the traditional proposal for a graphics pipeline, where each vertex and each pixel are independently processed. IMR architecture is similar to DirectX 9 pipeline detailed in Section 1.1 (see Figure 1.1), where a per vertex computation takes place and, after a rasterization step, generated pixels are also processed. Despite several resulting pixels being occluded pixels which will be discarded, every IMR primitive is sent

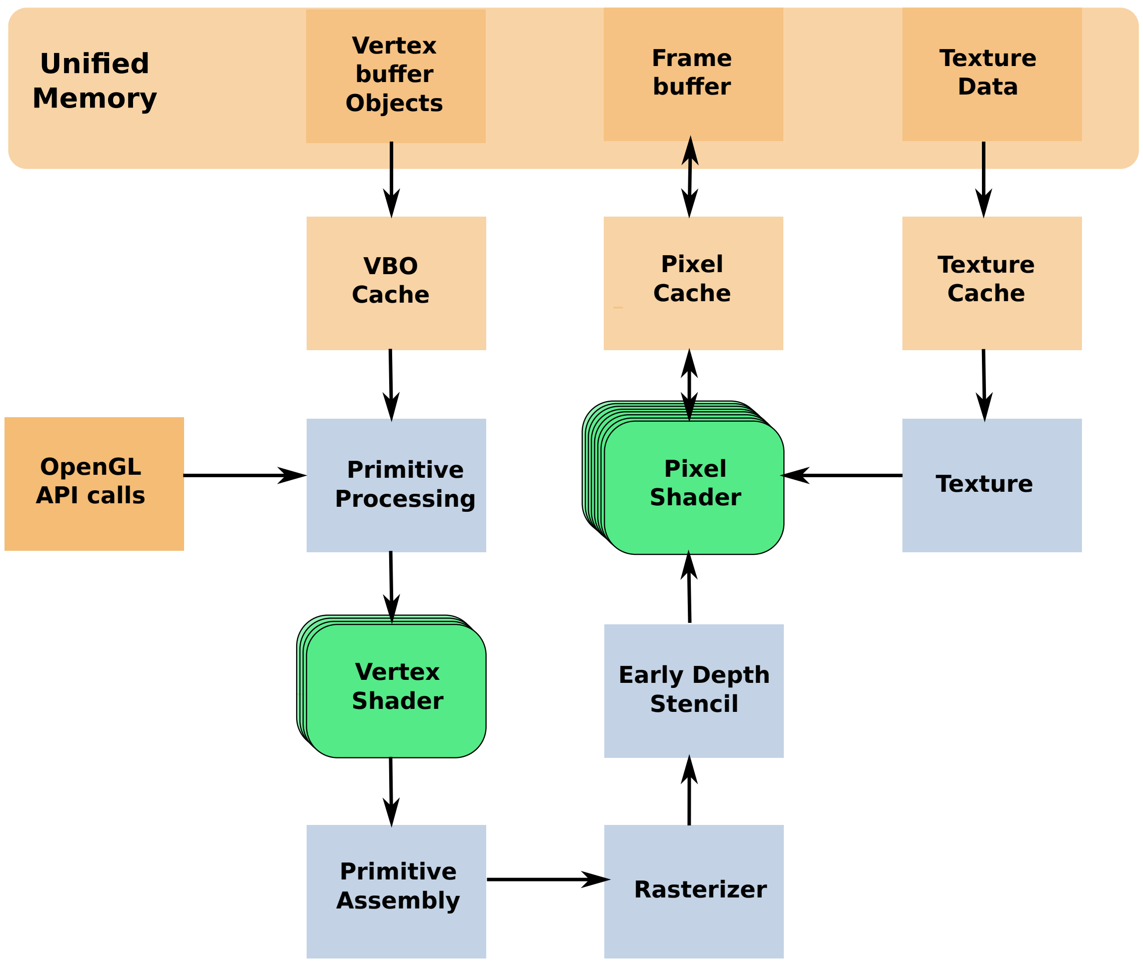


Figure 5.2: ULP Geforce Architecture in Tegra 3

down through the graphics pipeline. As a significant amount of processing steps and memory transactions are involved in the computation of these discarded pixels, a considerable amount of power is consumed, which is not desirable for a battery supplied device as the battery power is drained.

Tegra 3 architecture is a fixed function pipelined architecture that includes fully programmable fragment and vertex shaders [73]. Nvidia Tegra 3 processor is a multi-core system on a chip specifically designed for handheld devices. This processor implements a Variable Symmetric Multiprocessing with a Variable SMP (4-plus-1) (see Figure 5.1) focused on extending battery life [74]. That is, a fifth CPU core, called the *Companion* core or the battery saver core, which operates at low power frequency for active standby mode or music playback, while the four main "quad" cores are built to reach higher frequencies. All five cores are individually enabled and disable based on the workload. On the other hand, there are twelve cores, eight pixel shaders and four vertex shaders, in the ultra low power (ULP) Geforce GPU implemented in the Tegra 3.

Despite the fact that the Nvidia Tegra has a traditional IMR pipeline (see Figure 5.2), it has been designed to deliver the performance of a pipelined GPU architecture with low power consumption. Tegra implements an IMR architecture where the vertices are received as input primitives which are then processed to create polygons; once they have been rasterized, pixels are colored or textured in the pixel shader. As shown in Figure 5.2 the ULP Geforce architecture is a fixed function pipelined architecture with a fully programmable vertex and pixel shader (see green squares in Figure). Tegra 3 provides four vertex shaders and eight pixels shaders. This architecture includes a pixel cache, texture cache and vertex buffer objects cache to reduce memory transactions. Furthermore, this architecture includes several features for optimizing power consumption, such as an early Z-buffer placed prior to the pixel shader, and which discards occluded primitives in the early stages of the pipeline, or an optimized memory controller.

With respect to Tile-Based (TB) rendering architecture [7, 13, 44, 96] (see TB pipeline in Figure 5.3) is based on the rendering of tiles, small rectangular blocks of pixels. A tile based rendering architecture is a slightly different approach to IMR architecture. As shown in Figure 5.3, the scene is divided into tiles while geometries are sorted into tile lists and each tile is stored in the graphics chip; thus, the tile's frame buffer accesses are extremely inexpensive. As soon as the rendering of a particular tile is finished, it is written in an off chip frame buffer with an efficient block transfer. Therefore, tile-based architecture fits perfectly with the characteristics of handheld devices owing to the fact that per pixel bandwidth inefficiencies are reduced by breaking each frame into tiles, rendering them independently and finally assembling them together before display. Furthermore, this tiling approach provides a multitude of optimization and culling possibilities. Nevertheless, as the scene is projected to the screen space and its projection is divided into tiles, triangle sorting is an expensive process. Hence, tile-based rendering offers better bandwidth utilization than IMR in low complexity models, but as scene complexity increases,

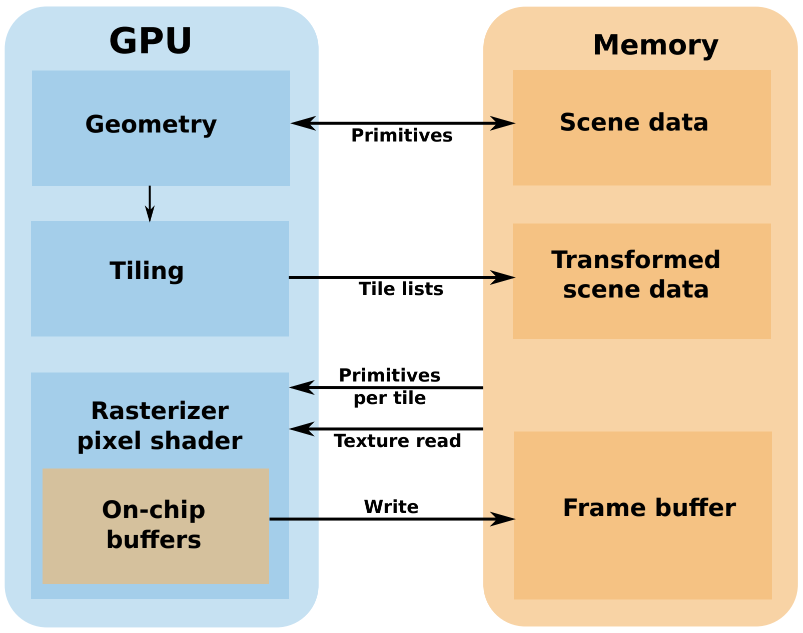


Figure 5.3: Tile base deferred pipeline

the bandwidth saved per pixel is reduced.

As the optimal approach also depends on the scene being rendered, the modern handheld device GPU implements a hybrid between tile-based and immediate mode renderings (TBIMR), exploiting the best characteristics of both approaches. In a TBIMR approach, polygons are transformed before being assigned to tiles. The rendering of the tiles itself is done in a manner more reminiscent of IMR as each tile computes values for all pixels in that tile before starting work on the following one. As this tile-based approach uses a fast, on-chip tile buffer, the GPU only writes the tile buffer contents to the frame buffer in main memory at the end of each tile, in contrast to traditional immediate-mode renderings, which require more frame buffer accesses.

Mali 400 (ARM) and Adreno 200 (Qualcomm) implement a TBIMR, which reduces memory bandwidth overhead and lower power consumption. More specifically, Mali 400 architecture tiles the scene but no full order independent hidden surface removal is performed. The Mali 400 architecture is depicted in Figure 5.4 and it is scalable from 1 to 4 cores. As shown in Figure, Mali 400 has a non-unified architecture with vertex and fragment processors and an optimized memory interface for an

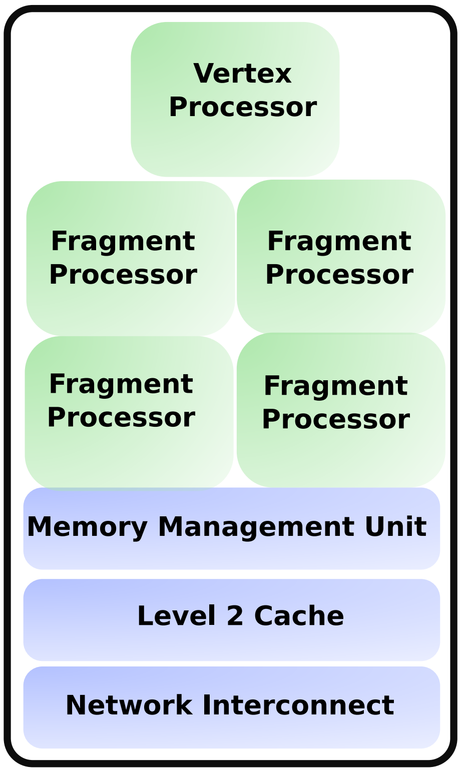


Figure 5.4: Mali 400 Architecture

efficient connection to other bus architectures.

As has been detailed, Mali GPUs and Adreno GPUs use tile-based immediate-mode rendering (TBIMR); however, the Mali GPU is closer to a tile based approach as the frame buffer is divided into tiles of size 16×16 pixels, a small tile size, while Adreno GPU is closer to an immediate mode rendering approach as tiles are split into relative large tile as 256K. As shown in Figure 5.5, different size of tiles can be defined according to hardware features. In Figure 5.5(a) the scene is split it into small tiles, such as in a Mali architecture, while another approaches, such as Adreno architecture, splits the screen into large tiles (see Figure 5.5(b)).

5.1.1. OpenGL ES

In the desktop world there are two standard 3D APIs: DirectX [66] and OpenGL [51, 87]. DirectX is the de facto standard 3D API for any system running the Microsoft

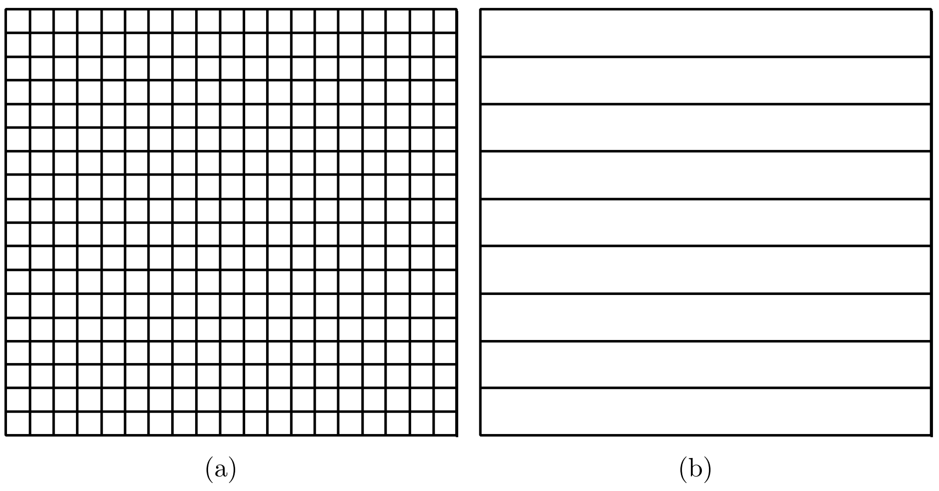


Figure 5.5: Screen split into tiles: (a) *small tiles* (b) *large tiles*

Windows operating system, and is used by the majority of 3D games on that platform. OpenGL (Open Graphics Library) is another cross-platform standard 3D API.

Nowadays, high-end mobile phones use a simple graphics API called OpenGL for Embedded Systems (OpenGL ES) [52, 81] as they cannot support any desktop graphic APIs, such as DirectX or OpenGL. As small handheld devices became increasingly common, OpenGL ES was developed, which was a stripped-down version of the desktop version. It removed a lot of redundant API calls and simplified other elements to make it run efficiently on the low-power GPUs in the market. As a result, it has been widely adopted across many platforms such as iOS (Operating System for Apple handheld devices) or Android [69, 94]. The device constraints that OpenGL ES addresses are highly limited processing capabilities and memory availability, low memory bandwidth, sensitivity to power consumption and lack of floating-point hardware.

There are three main flavors of OpenGL ES, 1.x, 2.x and 3.x, the latter of which was published in August 2012 and as of yet it is not supported for any device. However, many of general market devices support OpenGL ES 1.x and 2.x. Version 1.x implements a fixed function pipeline and it is derived from the original OpenGL specifications. Version 2.x implements a programmable graphics pipeline instead. Also, OpenGL 2.0 specification was used as the baseline for determining the feature

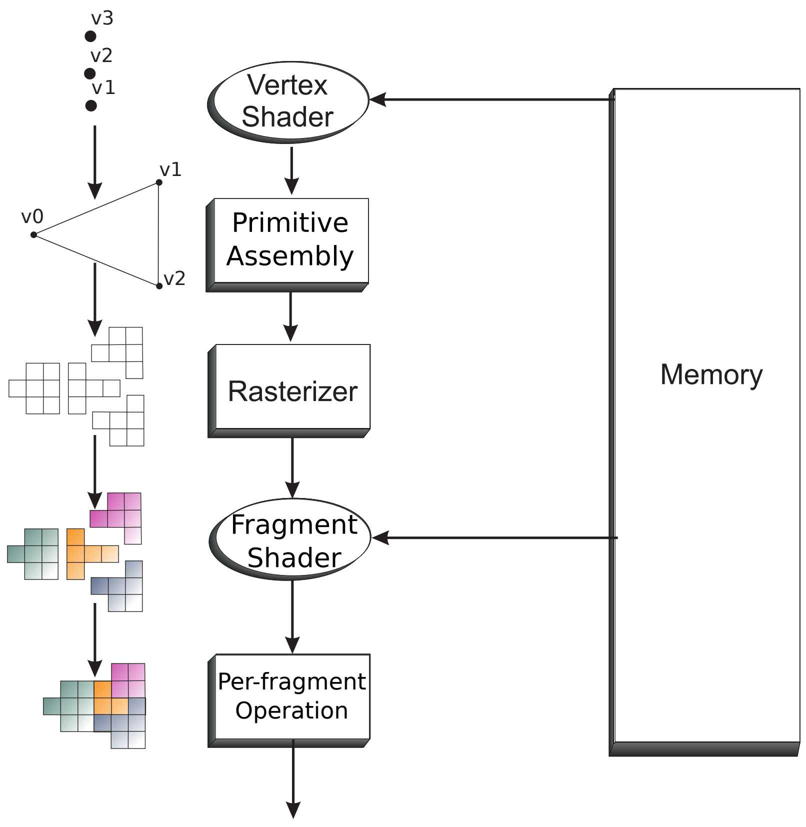


Figure 5.6: OpenGL ES 2.0 Graphic Pipeline

set in OpenGL ES 2.0.

On a classical approach, the model is evaluated and tessellated in the CPU and the generated vertices of these triangle meshes are sent to the GPU. These vertices are stored on attributes variables, which are inputs to the vertex shader, as shown in Figure 5.6. Every vertex shader receives a new vertex as input and this vertex can be modified to be stored in varying variables, which are outputs of the vertex shader.

After the vertex shader, the next stage in the pipeline is a fixed primitive assembly stage. In this stage, those vertices are assembled into individual geometric primitives that can be drawn, such as triangles, lines or point-sprites (see Figure 5.6). Clipping, a procedure to identify and remove those vertices outside the view cone, and backface culling, a process to remove those vertices which are no visible from the camera point of view, can also be performed in this stage in order to remove

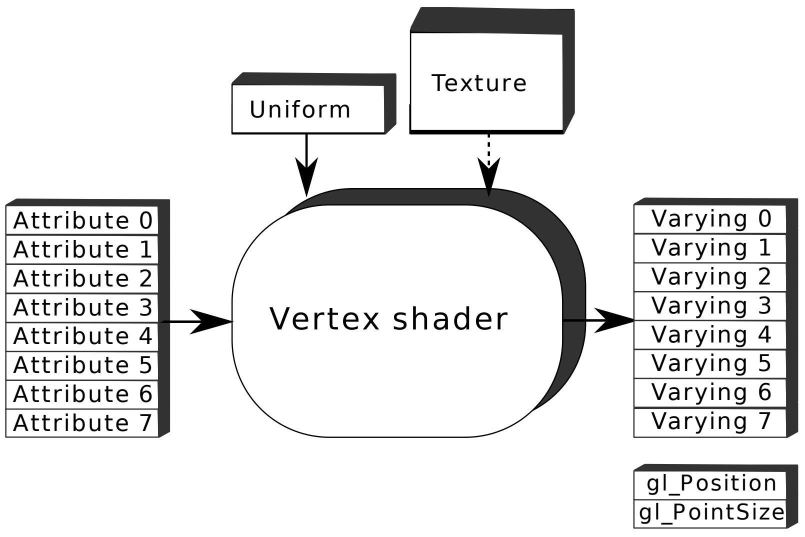


Figure 5.7: OpenGL ES 2.0 vertex shader

those invisible primitives. After clipping and culling, the primitive is ready to be passed to the next stage of the pipeline, which is the rasterization stage.

Rasterization is the process that converts primitives into a set of two-dimensional fragments, which are processed by the fragment shader. All the triangles built in the primitive assembly stage are converted into a group of fragments in this stage (see Figure 5.6). These two-dimensional fragments are pixels that will be drawn on the screen.

The vertex shader is the programmable stage in the rendering pipeline that handles the processing of individual vertices. It receives a single vertex composed of a series of vertex attributes. There must be a 1:1 mapping from input vertices to output vertices. Figure 5.7 shows a vertex shader and its inputs on the left side and outputs on the right side. Vertex shader inputs consist of:

- Attributes. Data supplied for each vertex using vertex arrays.
- Uniforms. Constant data used by the vertex shader.
- Texture. Data storage larger than uniform storage and used by the vertex shader. Samplers in vertex shaders are optional.

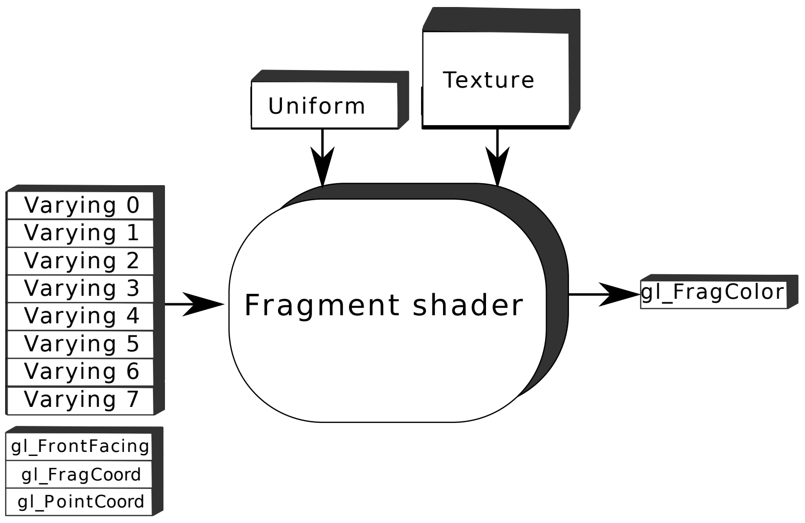


Figure 5.8: OpenGL ES 2.0 fragment shader

- Shader program. Source code or executable that describes the operations that will be performed on the vertex.

Varying variables are the output of the vertex shader. These primitives are passed through the rasterizer to the fragment shader and they are computed for each generated fragment.

Furthermore, another variables called built-in special variables are the output of the vertex shader. These variables contain useful information for the rendering and they are built by OpenGL instead of the program computed in the vertex shader. The built-in special variables available to the vertex shader are as follows:

- gl_Position. Variable used in the rasterizer to perform the clipping process and to convert the vertex position from model coordinates to screen coordinates.
- gl_PointSize. Variable used when point sprites are rendered and which describes the size of the point sprite in pixels.

Fragment shader implements a user-supplied program that, when executed, will process a fragment from the rasterization process into a set of colors and a single depth value. The fragment shader (see Figure 5.8) is executed for each generated fragment by the rasterization stage and takes the following inputs:

- *Varying* variables. Outputs of the vertex shader that are generated by the rasterization unit for each fragment.
- Uniform. Constant data used by the fragment shader.
- Texture. Data storage larger than uniform storage and used by the vertex shader.
- Shader program. Source code or executable that describes the operations that will be performed on the fragment.

Furthermore, fragment shader receives as input built-in special variables, which are read-only and contain information from the vertex shader and the rasterizer stages. As shown in Figure 5.8, these variables are the following:

- gl_FrontFacing. Boolean variable which indicates whether the fragment is part of a front facing primitive.
- gl_FragCoord. Variable that holds the screen coordinates of the current fragment.
- gl_PointCoord. Variable used in the rendering of point sprites and which holds the texture coordinate for the point sprite.

The output of the fragment shader is a built-in special variable called gl_FragColor which is used to send the fragment color into the per-fragment operation stage.

The color, screen coordinate location, depth and stencil generated by the rasterization stage become inputs to the per-fragment operation stage of the OpenGL ES 2.0 pipeline.

Finally, in the per-fragment operation, either a fragment is rejected or is written to the frame buffer. Therefore, the per-fragment operation stage performs the following functions on each fragment: pixel ownership test, scissor test, stencil and depth test, blending and dithering. A fragment produced by the rasterization with screen coordinates (x_i, y_j) can only modify the pixel at that location in the frame buffer.

5.2. Vertex Shader Tessellation on Handheld devices

Since the GPUs in current handheld devices do not generate any new geometry, the design of our tessellation proposal is based on the main features of the VST proposal [21], which analyzes the capabilities of classical desktop GPU, so VST for handheld translates characteristics of VST to the OpenGL ES 2.0 pipeline depicted in Figure 5.6. Let us remind ourselves of the main features of VST alternative. VST is a tuning strategy that permits the choice of an suitable relation between the requirements of storage and the number synchronization CPU-GPU according to the underlying GPU architecture. This technique uses a parametric map as vertex shader input, with as many positions in the parametric domain as output vertices are needed for the desired resolution of the triangle mesh. Then, by accessing the control points on the Bézier surface to be tessellated, these virtual vertices are evaluated on the vertex shader, generating the resulting triangle mesh. Hence, the resolution of the triangle mesh is chosen by the parametric map being used.

This approach subdivides the parametric domain into uniform squares, where the granularity is selected in function of the desired resolution. More specifically, it tessellates the surface in the parametric space (u, v) in $2^l \times 2^l$ squares of size $\frac{1}{2^l} \times \frac{1}{2^l}$, for a resolution level l that is previously selected by the application taking into account different factors, such as computational power, screen space error or model complexity. Therefore, the Bézier surface is evaluated for each one of the $2^{l+1} \times 2^{l+1}$ to obtain the corresponding Euclidean space points (see Equation 1.6). The resulting vertices are conveniently arranged to output a triangle strip.

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

$$P^l = \begin{bmatrix} (u_1, v_1) & (u_2, v_1) & \cdots & (u_{2^{l+1}}, v_1) \\ (u_1, v_2) & (u_2, v_2) & \cdots & (u_{2^{l+1}}, v_2) \\ \vdots & \vdots & \ddots & \vdots \\ (u_1, v_{2^{l+1}}) & (u_2, v_{2^{l+1}}) & \cdots & (u_{2^{l+1}}, v_{2^{l+1}}) \end{bmatrix} \tag{5.1}$$

where

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

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

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

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

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

with $1 \leq N_d \leq N_S$.

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

$$M = M_{[B^s]} \times N_d \tag{5.2}$$

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

Figure 5.9 depicts an example of our approach for a resolution level of 2 ($l = 2$) and a couple of Bézier surfaces to be processed concurrently ($N_d = 2$). The parametric map for $l = 2$ is replicated and the resulting samples are the input primitives for the vertex shader (middle box in the figure). The control points of the two surfaces are transferred to the GPU (left box, texture memory is used in this example) and a draw call causes the evaluation of the samples that results in the meshes of the right box.

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

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

5.3. Implementation details

In this subsection, we summarize the details of our implementation. The kernel implemented processes bicubic Bézier surfaces and exploits the capabilities of OpenGL ES 2.0. The structure of our algorithm is shown in Figure 5.10. In the

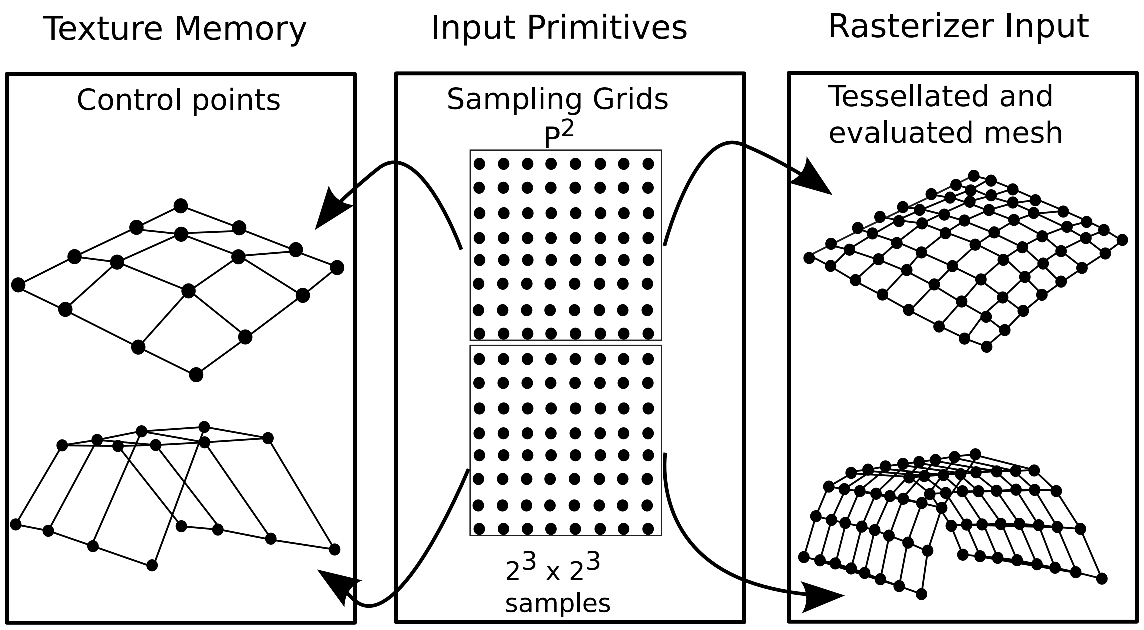


Figure 5.9: Example of parametric maps for $l = 2$

preprocessing stage the grids of virtual vertices P^l, $1 \leq l \leq L$ are transferred from CPU to GPU. During the synthesis process the level of resolution per surface is selected and the control points of N_d surfaces are sent down from CPU to GPU.

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

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

Instead of applying the de Casteljau algorithm [88], in our kernel a direct evaluation strategy is used to compute the tessellation, as it results in a more efficient GPU implementation. As will be shown in the results section, the simplicity of this strategy and the efficient management of the data storage are key points, together with the CPU-GPU transfers, for the real time rendering of high quality models.

According to the vertex shader structure of OpenGL ES, this work proposes two

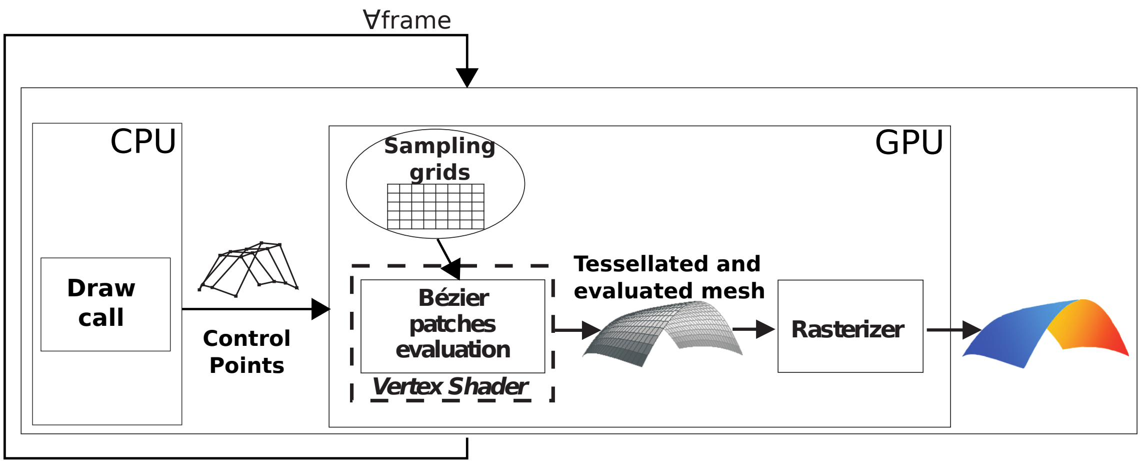


Figure 5.10: Structure of the method

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

5.3.1. Uniform method

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

In order to check the suitability of the uniform variables storage, a couple of features should be clarified: firstly, the overhead related to the memory access of uniform variables; and, secondly, its storage capacity. From a performance point of view, and according to hardware manufacturers [64, 73], any access to uniform variable memory is simple and fast and it has a low impact on execution speed. Moreover, this access overhead is similar to an arithmetic operation such as addition or subtraction, and considerably faster than other operation, such as division or square root.

With respect to the storage capacity, uniform variables are generally stored in hardware in constant storage. This physical storage space is organized into a grid with four columns and a row for each storage location. As storage is typically of a fixed size, there is a limit on the number of uniforms that can be used in a program. According to the standard, any implementation of OpenGL ES 2.0 must provide at least uniform memory in the vertex shader, M_{uv}, to store 128 vectors and uniform memory in the pixel shader, M_{uf}, to store 16 vectors, each vector comprising four floats. Hence, the maximum number of surfaces per chunk would be

$$N_d = \frac{M_{uv}}{M_{[B^S]}} \tag{5.3}$$

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

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

In Section 5.4, the overhead of the draw calls, N_{DP}, and N_d are analyzed in order to identify the bottleneck of different configurations.

5.3.2. Texture VST for handheld devices

An alternative to the uniform variables is to store the control points in texture memory, M_T, as is shown in Figure 5.7. As texture memory can store a higher number of surfaces than uniform memory, this alternative prevents an important number of draw calls per frame, N_{DP}, one of the main drawbacks of using uniform variables.

$$N_d = \frac{M_T}{M_{[B^S]}} \tag{5.4}$$

where M_T is considerably larger than M_{uv} and subsequently more primitives can be stored in texture memory than in the uniform variables memory simultaneously. More specifically, for Mali 400 $M_T = 16MB$, which is four times larger than M_{uv}.

Although Bézier control points can be stored in the texture memory, this storage space has not been designed for store floats. In OpenGL ES 2.0 texture memory formats have been implemented to store color information as a 4-byte vector. There are different formats in texture storage, but typically each color is 32 bit data and they are split up into 4 groups of 8 bits: rgba [42] red, green and blue colors and the alpha channel. The texture method defines a codification process to store and recover float values from texture memory. This encoding process to pack a float into a rgba texture is a simple process based on multiplications and divisions by the largest number that can appear.

On the other hand, the texture method solves the main disadvantages of uniform approach for handheld devices, however it has yet to be implemented on Android platforms. As it is shown in Figure 5.7 with a dashed line, access to texture memory from vertex shader is not implemented in any commercial OpenGL 2.0 device at this moment. First devices implementing this feature are expected in lately 2013.

5.4. Experimental results

In this section, the results of the evaluation of VST for handheld devices on different GPU architectures is analyzed. In particular, the platforms employed are a Samsung Galaxy S2 (*Mali*), a Samsung Galaxy ACE (*Adreno*) and a Asus Transformer TF 300 (*Tegra 3*).

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

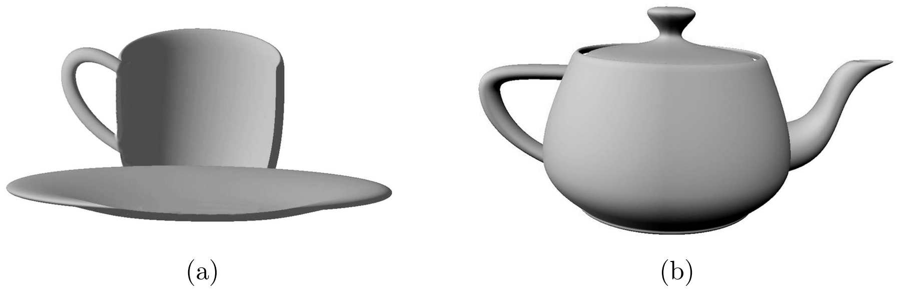


Figure 5.11: Models employed in the test scenes (a) *Teacup* (b) *Teapot*

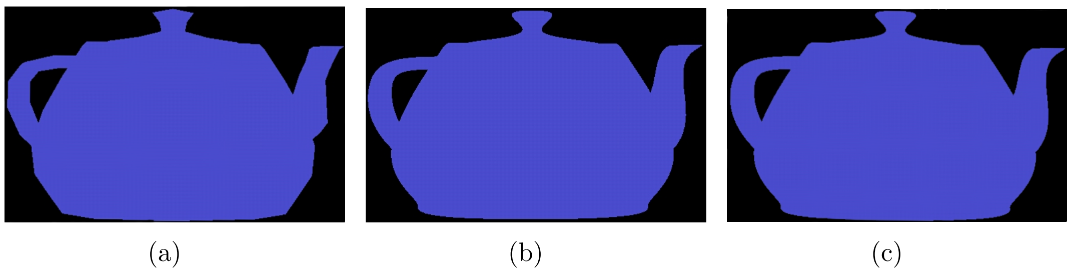


Figure 5.12: Screenshots of the *teacup* model with (a) L=1, (b) L=3 and (c) L=5

implements a Nvidia Tegra 3 Quad-core at 1.2GHz and a ULP Geforce 12-core: 4 vertex shaders and 8 fragment shaders.

Different scenes, composed of replicas of a set of models, have been used in our tests. The models (*Teacup* and *Teapot*) are depicted in Figure 5.11. The number of primitives generated for the different resolution levels is shown in Table 5.2. Column N_s presents the number of Bézier surfaces whereas the rest of columns include the number of triangles generated for the corresponding level of detail with a uniform tessellation; i.e. all surfaces are tessellated with the same level of detail. Figure 5.12 depicts a screenshot of the *teacup* model rendered with $L = 1$, $L = 3$ and $L = 5$ in *Tegra 3*. Table 5.2 shows the KBytes stored in the vertex buffer for different scenes and different resolution levels. As shown in the figure, performance is dramatically reduced increasing the tessellation level.

First of all, we must emphasize that our proposal obtains superior performance than the best CPU results: up to 3 fps in *Mali* and 5 fps in *Tegra* for the scene S_{5pots} with $L = 1$. In this CPU implementation each sample is evaluated in the CPU and

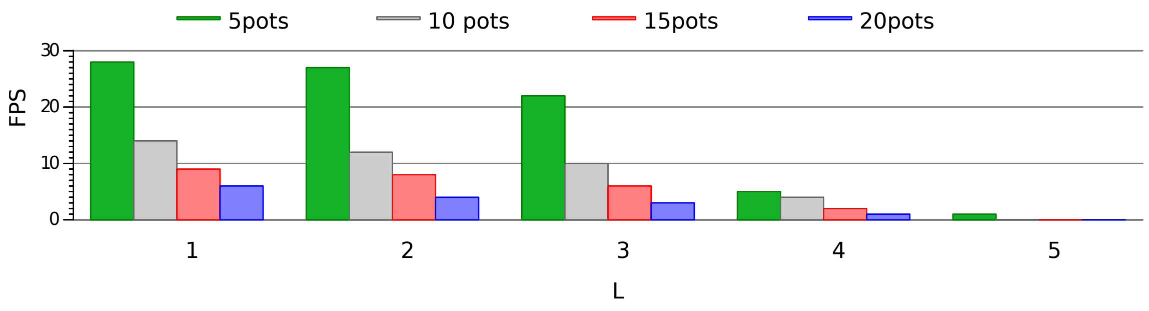


Figure 5.13: FPS of our proposal implementated in *Mali* with different levels of resolution

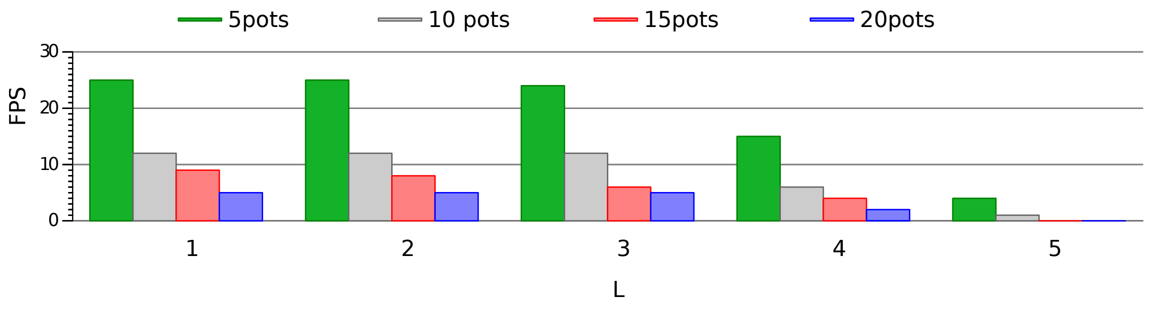


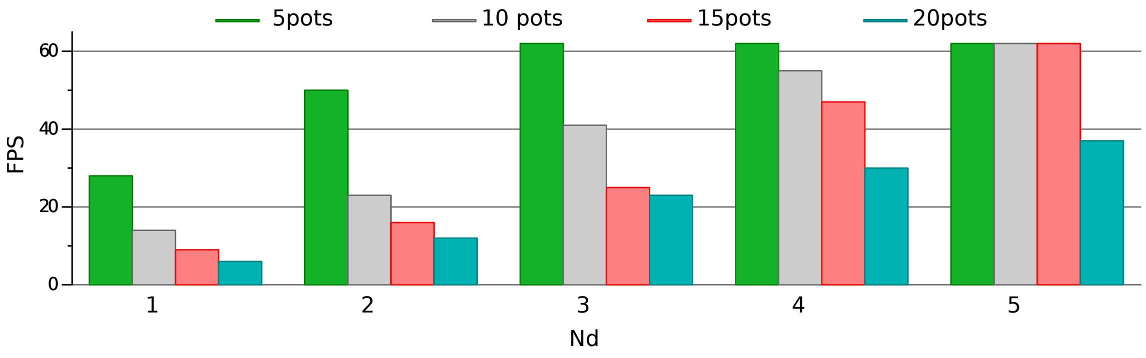
Figure 5.14: FPS of our proposal implementated in *Tegra* with different levels of resolution

the whole vertex buffer is sent down to the GPU for each frame.

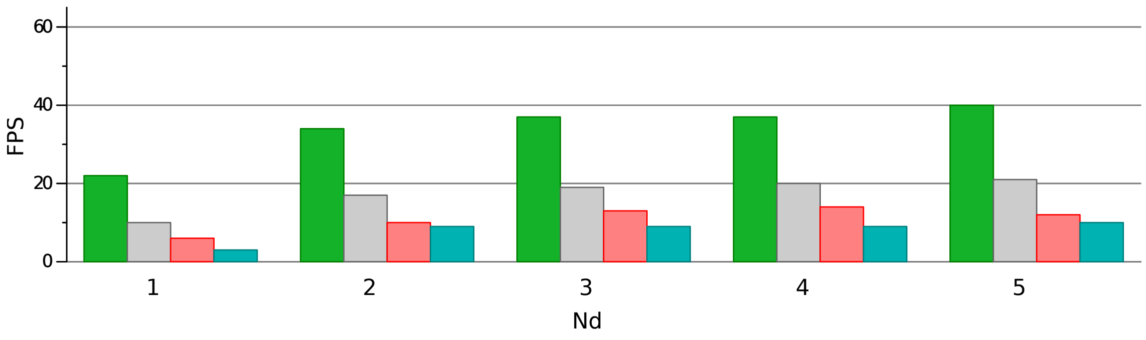
Our analysis is principally focused on obtaining the optimal tuning factors for the three parameters used to characterize the behavior of our method: $\{L, N_d, N_{DP}\}$. The first factor we have analyzed in our method is the resolution level, L. As the evaluation and tessellation of Bézier surfaces are computed for every vertex in each GPU shader, the number of vertices to be computed is defined by the selected tessellation level; thus, the greater the increase in the tessellation level of Bézier surfaces, the lower the performance obtained. More specifically, each of the $2^{l+1} \times 2^{l+1} \times N_d$ samples is evaluated in each GPU vertex shader. The graphs in Figures 5.13 and 5.14 show the frame rate on two distinct platforms for 4 of the test scenes with the different resolution levels. In both devices, *Mali* and *Tegra*, the frame rate drop when the resolution level increases, but a number of differences between the two platforms can be observed. *Mali* obtains better results when the tessellation level is low (less synchronization penalty) whereas *Tegra* performs better when the resolution level increases (4 vertex shaders vs. 1 vertex shader in *Mali*).

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

Table 5.1: Number of surfaces and triangles generated (in K) for each scene



(a)

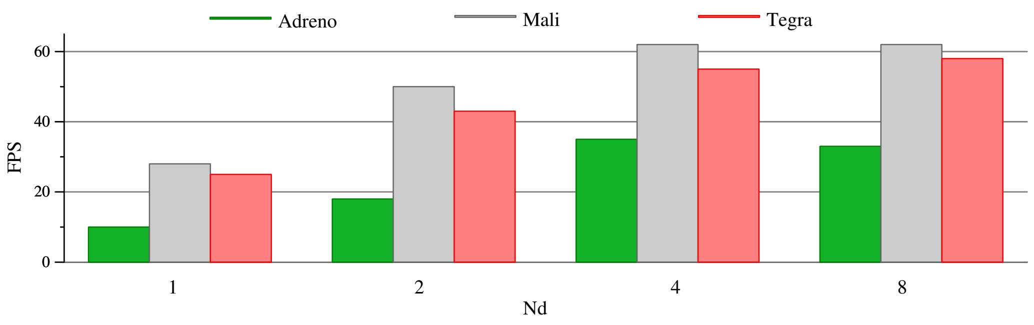


(b)

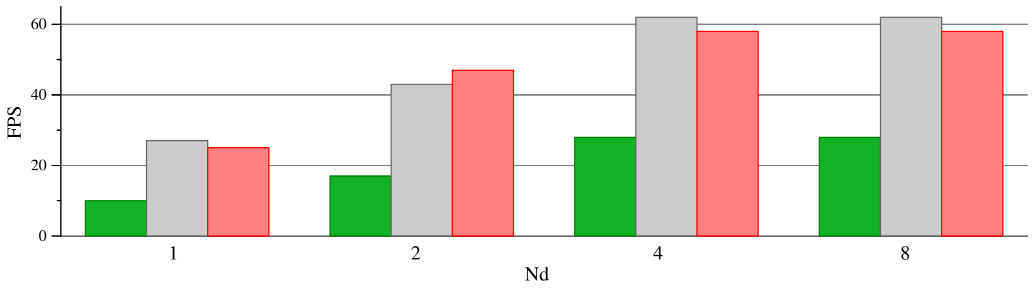
Figure 5.15: Frame rate in *Mali* with different N_d and considering: (a) $L = 1$ (b) $L = 3$

In any case, frame rate dramatically drops for $L = 5$ in both cases, since a vertex buffer size greater than 16 MB is needed.

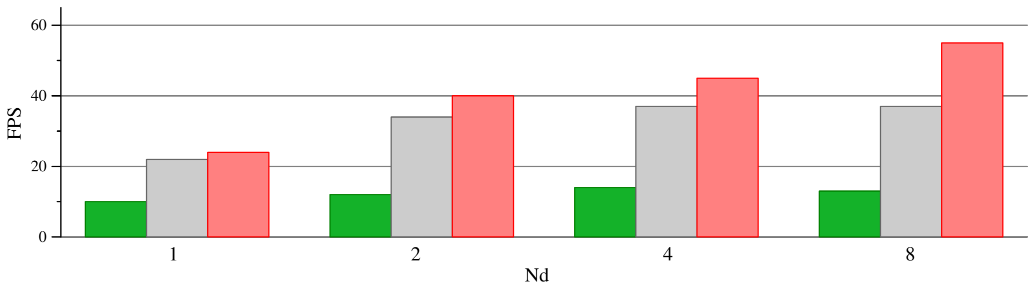
In short, the main problems of current GPUs in handheld devices are the computing power and the low number of vertex shaders. This implies a limit on the



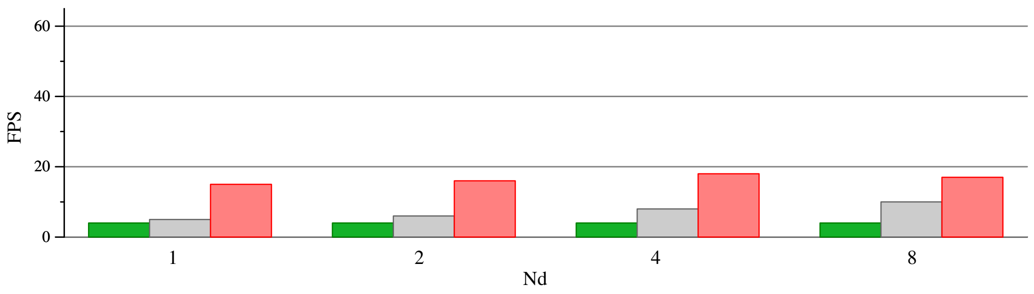
(a)



(b)

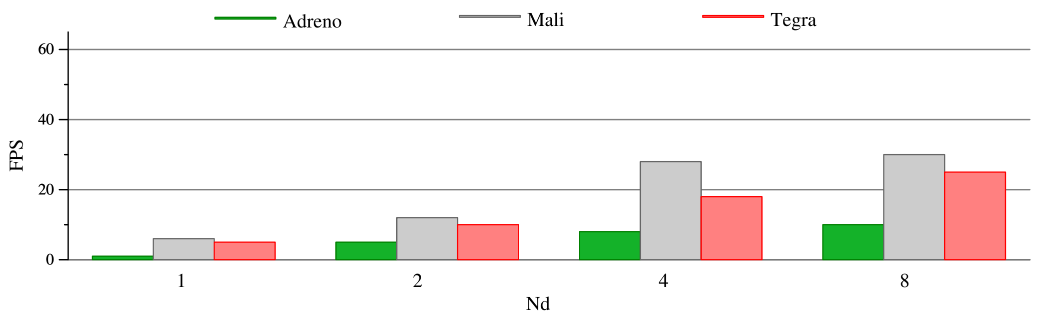


(c)

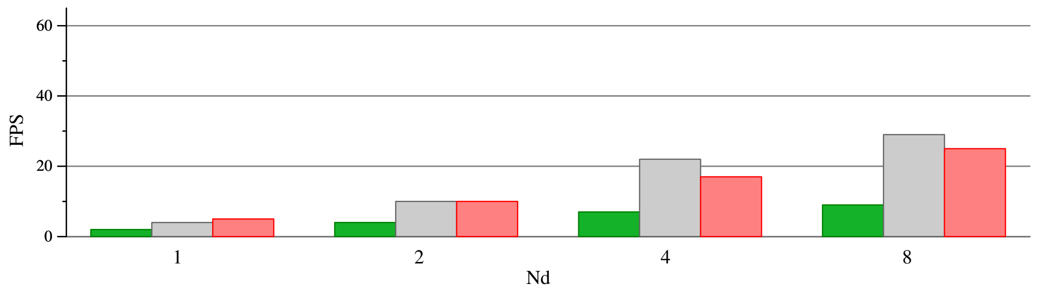


(d)

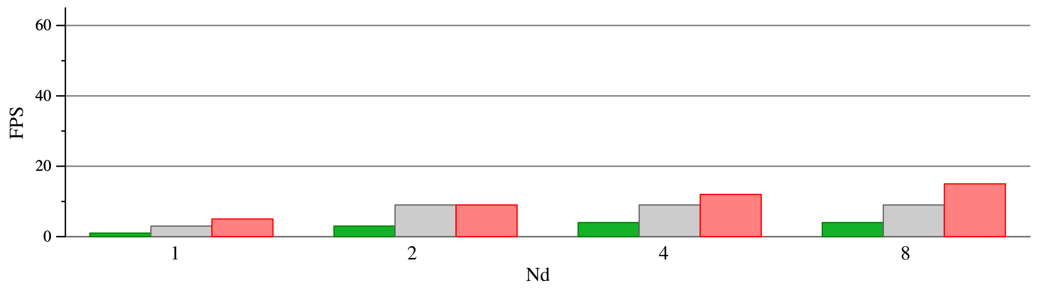
Figure 5.16: Frame rate comparative in *Adreno, Mali and Tegra* with S_{5pots} and different resolution levels: (a) L=1 (b) L=2 (c) L=3 (d) L=4



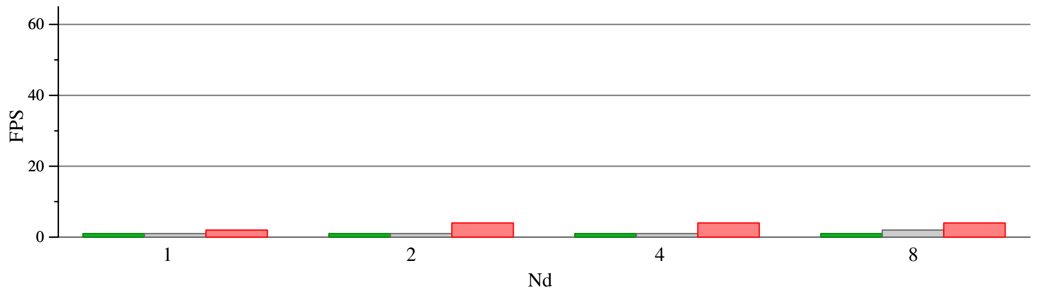
(a)



(b)



(c)



(d)

Figure 5.17: Frame rate comparative in *Adreno, Mali and Tegra* with S_{20pots} and different resolution levels : (a) L=1 (b) L=2 (c) L=3 (d) L=4

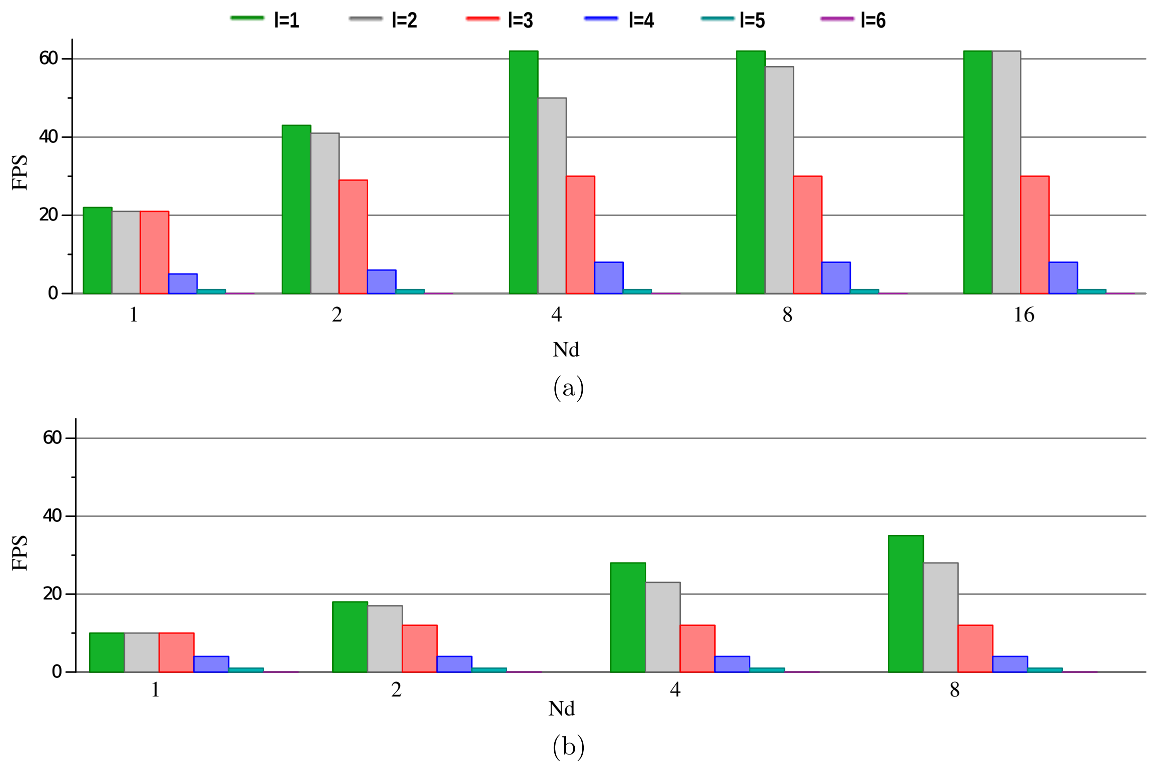


Figure 5.18: Performance of scene S_{5pots} with texture access in: (a) *Mali* (b) *Adreno*

resolution that can be achieved and the complexity of scenes that can be rendered.

With respect to the rest of factors $\{N_d, N_{DP}\}$ where $N_{DP} = N_S/N_d$, four different scenes have been considered and their performance is depicted with different resolution levels in Figure 5.15 on the *Mali*. This graph analyzes how the number of surfaces that can be tessellated by a single draw call affects the GPU performance. Similar behavior is observed on *Tegra* and *Adreno* and other scenes. Table 5.3 presents the number of N_{DP} for these scenes with different number of draw calls. As can be observed, values lower than $N_{DP} = 40$ reach maximum performance on *Mali*, that is 62 fps. Broadly speaking, if the complexity of the model increases (more surfaces, Ns), maintaining a high performance usually implies to try to reduce N_{DP}. For example, S_{5pots} with $N_S = 160$ for $\{L = 1, N_d = 4, N_{DP} = 40\}$ achieves 60 fps, and S_{15pots} with $N_S = 480$ also achieves 60 fps for a configuration $\{L = 1, N_d = 16, N_{DP} = 30\}$. Thus, a trade-off between the number of draw calls and the primitives processed in parallel is needed to increase the performance as

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

Table 5.2: Vertex buffer size (in KB) for each scene

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

Table 5.3: N_{DP} for each scene

much as possible.

Figure 5.16 and Figure 5.17 present a final comparison of our best results in different platforms: *Adreno*, *Mali* and *Tegra*. Different models such as S_{5pots}, and S_{20pots} have been depicted as a wide range of rendered primitives and vertex buffer size are considered. As has previously been explained, the uniform method stores Bézier control points in the vector uniform variables. Hence, as there are only 128 vector uniform variables in the Adreno 200 architecture, only 8 surfaces can be stored for each draw call, $N_d = 8$, meanwhile up to 16 surfaces can be stored in Mali 400 or in a Tegra 3 in 256 vector uniform variables, $N_d = 16$.

For a low level of resolution, $L = 1$ or $L = 2$, *Mali* and *Adreno* offer the best performance for $N_d = 8$; i.e. 62 and 35 fps, respectively. *Tegra* also has the best performance, that is 58 fps but scales perfectly as the level of resolution increases, up to 55 fps for $L = 3$. Nonetheless, in *Mali* and *Adreno* the performance drop for $L \geq 3$ (37 and 13 fps are achieved, respectively, for $L = 3$ and $N_d = 8$) confirms that the number of computational cores (i.e. the computational power) becomes a limiting factor and is noticeable in the performance. More specifically, the Mali 400 MP GPU has one vertex shader while Tegra 3 has four. For larger levels of resolution, $L = 4$, the performance is below 20 fps for all GPUs due to the low number of vertex shaders.

Although, in the uniform method the number of primitives processed in parallel in the GPU is restricted by the vertex uniform vectors, the Texture method cannot be tested on any market devices, as the access to texture memory from the vertex shader has yet to be implemented on the Android platform. This approach would solve the main problem of the uniform method, since texture memory provides a larger storage space than uniform variables. To test how this approach would be, we have designed a group of tests to measure the texture access latency. Although the evaluation of a Bézier surface cannot be carried out in the fragment shader, the texture memory accesses are processed in this stage to analyze the access latency. Figure 5.18 shows the performance of a texture memory access from the fragment shader in *Mali* and *Adreno* architectures. A simple model comprising 160 surfaces (S_{5pots}) has been chosen for this test to reduce the impact of computational power and CPU-GPU communication as far as possible. According to the results, the overhead associated with the texture access is about 20% of the final performance in Mali architecture and under 10% in an Adreno devices, as a unified architecture as implemented in Adreno devices, dynamically configuring its GPU cores to allocate vertex or fragment processing.

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

5.5. Conclusions

In this chapter we have presented a proposal for the tessellation of Bézier surfaces on the GPU of embedded devices. VST for handheld devices has been designed to test the bottleneck of the GPU implemented on handheld devices and to evaluate how the performance of VST for handheld devices is sensitive to important architectural parameters.

Parametric surfaces cannot be directly rendered in the current GPUs of modern handheld devices, thus our first contribution is to achieve a rendering of Bézier surface and the analysis of the most relevant hardware constraints in current GPUs using in the handheld devices. Another related contribution is to describe some of the tuning techniques employed.

VST for handheld permits the analysis of the most relevant hardware constraints in graphics computing. It has basically been implemented as a benchmark set to analyze hardware capabilities of embedded devices, as the evaluation and tessellation of Bézier surfaces on the GPU enable the analysis of hardware constraints.

As one of the main features of GPUs implemented in handheld devices is to reduce power consumption as far as possible, the computational power of the GPU's shader has been dramatically reduced compared to a desktop GPU. Hence, computational power and CPU-GPU communication, a typical bottleneck in GPUs performance, are the two main topics when analyzing VST for handheld devices.

VST for handheld devices is based on the rendering of Bézier surfaces, considering that these surfaces fit perfectly into the analysis of these hardware constraints. Bézier surfaces have been evaluated on the GPU cores to test their computational power, and a different number of surfaces has been rendered per draw call to analyze the performance overhead related to this communication.

VST for the analysis of handheld devices has been tested on handheld devices from different manufacturers, giving rise to the conclusion that CPU-GPU communication is the main drawback of these devices, despite the reduced processing power.

Chapter 6

Rendering Pipeline for NURBS Surfaces

In this chapter we present a proposal for rendering NURBS surfaces directly on the GPU without a tessellation preprocess. Our proposal, Rendering Pipeline for NURBS Surfaces (RPNS), is based on a new primitive, KSQuad, which 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. Additionally, to provide a versatile example of this new primitive, different culling strategies are applied using the KSQuad primitive and its strong convex hull property. An implementation of RPNS using current GPUs is presented, achieving real-time rendering rates of complex parametric models.

NURBS surfaces [37, 38, 79, 82] have been widely employed in CAD/CAM tools and graphics applications owing to their capabilities for modeling complex geometries. In addition to the high quality of NURBS models, another advantage of NURBS representations is the compactness of the description and, in consequence, the low storage and transmission requirements. Furthermore, graphics designers can produce models and animations in a simpler and faster way as they need 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, depending on the required LOD.

NURBS surfaces are commonly decomposed into a series of Bézier patches by the well-known technique called knot refinement [79], 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 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 adjusts 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. Thus, 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 pre-processing times as well as the introduction of artifacts, especially at the surface boundaries [1].

Thus, even though a NURBS surface can be decomposed into a set of Bézier surfaces, which makes efficient rendering and evaluation possible, Bézier surfaces are not a good replacement for NURBS surfaces in fields such as modeling or animation [54], since NURBS properties are not projected into these new Bézier surfaces. For example, any change in one Bézier point can reduce the continuity from C^1 to C^0 on the edge between surfaces. This effect is not an artifact on NURBS surfaces as, according to local support property, $p \times q$ consecutive points are needed to generate a corner or a change from C^1 to C^0. 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.

The rest of the chapter is organized as follows: Section 6.1 presents the structure of RPNS, Section 6.2 presents KSQuad, the primitive our pipeline is based on. Section 6.3 describes the rendering primitive, KSDice. Section 6.4 details the culling techniques implemented in RPNS. Section 6.5 introduces the Stair Strategy that achieves an efficient evaluation of NURBS surface. Section 6.6 describes implementation details of RPNS on current GPUs using DirectX11 and Section 6.7 presents the experimental results obtained in our tests. Finally, in Section 6.8 the main conclusions are highlighted.

This work has been published in [26].

6.1. Rendering Pipeline for NURBS Surfaces

This section presents RPNS, Rendering pipeline for NURBS Surface, a novel solution for the direct evaluation of NURBS surfaces on the GPU with no previous tessellation procedure or pre-processing. The objective is the efficient rendering of each surface so that the final image is free of cracks and holes, either inside each surface or between neighbor surfaces, making it possible to exploit the parallelism of the GPU to perform common operations such as sketching on surfaces, interactive trimming or surface-surface intersection.

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, which computes what, how and where the things are drawn. Finally, the rasterizer stage renders an image; i.e., it sets the color for the pixels covered by the different object in the scene.

Figure 6.1 is a block diagram from our pipeline proposal for NURBS surfaces. It consists of three modules: geometry, sampler and rasterizer. In this work we focus on the first two stages of RPNS, geometry and sampler, leaving the discussion about the rasterization of KSDices as future work.

As mentioned below, RPNS adds a new primitive, KSQuad (see Section 6.2), to the input stream of the geometry shader. *KSQuad* is based on the regions defined by the projection on the parametric cell delimited by the different knot spans. This primitive provides an efficient and accurate evaluation of NURBS surfaces and in RPNS they are processed in the geometry stage (see Figure 6.1). *KSQuad* needs no pre-processing stage and intrinsically maintains the main geometric properties of NURBS surfaces, such as local support and strong convex hull. The exploitation of these properties enables us to improve performance by applying acceleration algorithms, such as culling techniques.

As depicted in Figure 6.1, an intermediate stage, sampler, between the geometry and the rasterizer stages is added in our pipeline proposal. In this stage an adaptive sampling of the KSQuad primitives is performed according to the viewpoint, the

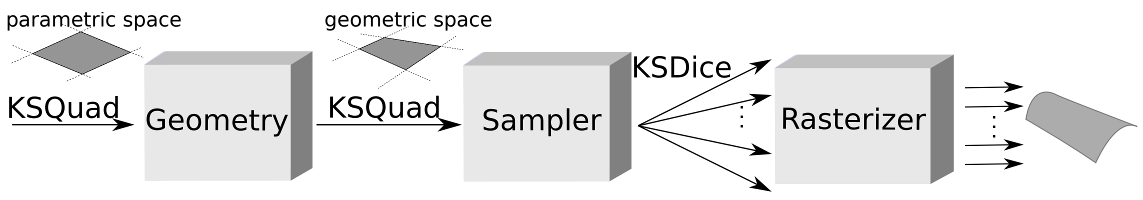


Figure 6.1: Generic structure of the rendering pipeline for NURBS surfaces based on KSQuad

geometric characteristics of the surface and the boundary edges between surfaces. More precisely, the geometry stage precedes the sampler stage in RPNS pipeline and all this geometric information is used to guide the sampling process.

In sampler stage, RPNS carries out an adaptive discretization of KSQuad primitives into *KSDices* (see Section 6.3) according to the level of resolution needed or the geometric properties. This sampling process results in a set of sampled points or dices that we have named *KSDice* and which make it possible to render the surface without cracks or holes. Here, KSDice, a new object analogous to the idea of surfels in the context of point rendering [78], is introduced. Each KSDice consists of a sampled point and additional information such as the parametric size of the dice and the degree of the corresponding surface,and it does not save any explicit connectivity information. The KSQuad discretization makes it possible to find an optimal rendering of the geometry of surfaces with minimum redundancy. Thus, a suitable discretization is obtained when it can be guaranteed that there is at least one KSDice projected into the region of each output pixel for orthographic projection. Therefore, the objective is to reduce the number of positions to be evaluated for each KSQuad primitive while keeping the quality of the resulting image.

On the other hand, the utilization of the KSQuad primitive allows the RPNS pipeline to improve performance in two ways: culling techniques and evaluation of NURBS surfaces using a non-recursive strategy.

One beneficial side effect of bringing forward the geometry stage prior to the sampling is that we can improve the performance by also moving forward the application of other techniques in the pipeline, such as culling. Whereas backface culling is usually performed after a tessellation step, RPNS follows recent proposals [48] which cull before tessellation, so culling is performed before discretization in RPNS (see Section 6.4). By applying a culling technique in the first stages of the pipeline, the number of primitives to be processed is dramatically reduced [48]. To prove this, several culling algorithms that exploit the versatility and flexibility offered by the use of KSQuad in the geometry stage have been implemented. These culling proposals are based on the strong convex hull property of the NURBS surfaces that is preserved in the KSQuad primitive, and it efficiently avoids the evaluation of points of knot spans which do not contribute to the final image.

An important feature of RPNS is the evaluation of NURBS surfaces without any approximation; thus, 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. RPNS introduces a non-recursive strategy, called stair strategy (see Section 6.5), to the evaluation of the basis functions which makes it suitable for a GPU architecture. This strategy performs an explicit evaluation of the basis functions by exploiting the local support property of NURBS surfaces; i.e., the fact that the number of nonzero basis functions in any knot span is, at most, equal to the basis function degree.

In summary, by means of the KSQuad primitive, RPNS exploits the main features of NURBS surfaces to accomplish their real time evaluation and direct display, rivaling approaches based on the REYES pipeline in quality and performance. Moreover, whereas other similar approaches really compute the basis functions previously on the CPU [54], RPNS goes one step further and evaluates the whole NURBS equation in the GPU in real time with no pre-computation on the CPU. To test our proposal, and although this chapter 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 real-time rendering rates.

6.2. KSQuad Primitive

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

A NURBS surface is obtained as the tensor product of two NURBS curves, as explained in Section 1.2.2, parametric curves that are a generalization of Bézier curves and are defined by their 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:

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)\ N_{j,q}(v)\ w_{i,j} B_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)\ N_{j,q}(v)\ w_{i,j}}, \quad 0 \le u, v \le 1 \tag{6.1}$$

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 non-rational B-spline basis function defined on two knot vectors of $p+n+1$ and $q+m+1$ elements, respectively:

$$\begin{aligned} U &= \left[\underbrace{0, \cdots, 0}_{p+1}, x_{p+1}, \cdots, x_{r-p-1}, \underbrace{1, \cdots 1}_{p+1}\right] \\ V &= \left[\underbrace{0, \cdots, 0}_{q+1}, y_{q+1}, \cdots, y_{s-q-1} \underbrace{1, \cdots 1}_{q+1}\right] \end{aligned} \tag{6.2}$$

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

$$N_{i,p}(u) = \frac{u - x_i}{x_{i+p} - x_i} N_{i,p-1}(u) + \frac{x_{i+p+1} - u}{x_{i+p+1} - x_{i+1}} N_{i+1,p-1}(u) \tag{6.3}$$

with

$$N_{i,0}(u) = \begin{cases} 1 & \text{if} \quad x_i \le u < x_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

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

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 this notion to propose a suitable input primitive that requires no 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 surface in this interval without any recursive computation, making it suitable for GPU implementation. It is important to emphasize that at no time does our proposal decompose the NURBS surface. Each position of the surface is directly evaluated on the NURBS surface.

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

$$\mathrm{KSQuad}_{i,j} = \{\overbrace{x_i, x_{i+1}, y_j, y_{j+1}}^{\text{knot span}}, \overbrace{B_{i-p,j-q}, \cdots, B_{i,j}}^{\text{control points}} \underbrace{w_{i-p,j-q}, \cdots, w_{i,j}}_{\text{weights}}\} \tag{6.5}$$

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

Each KSQuad$_{i,j}$ is defined in the parametric domain by a knot span, 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 6.2. As a bicubic surface is shown in the figure, sixteen control points are evaluated in each knot span projection. Hence, the NURBS surface in the model space is composed of patches which are generated as a projection of the knot span into the model space, see shadow patch and shadow knot span in the figure.

A KSQuad preserves the many desirable geometric properties presented in NURBS curves and surfaces, 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 $\{B_{i-p,j-p}, \ldots, B_{i,j}\}$. This property assumes that all the weights in the NURBS surfaces are positive values, and it allows us to propose

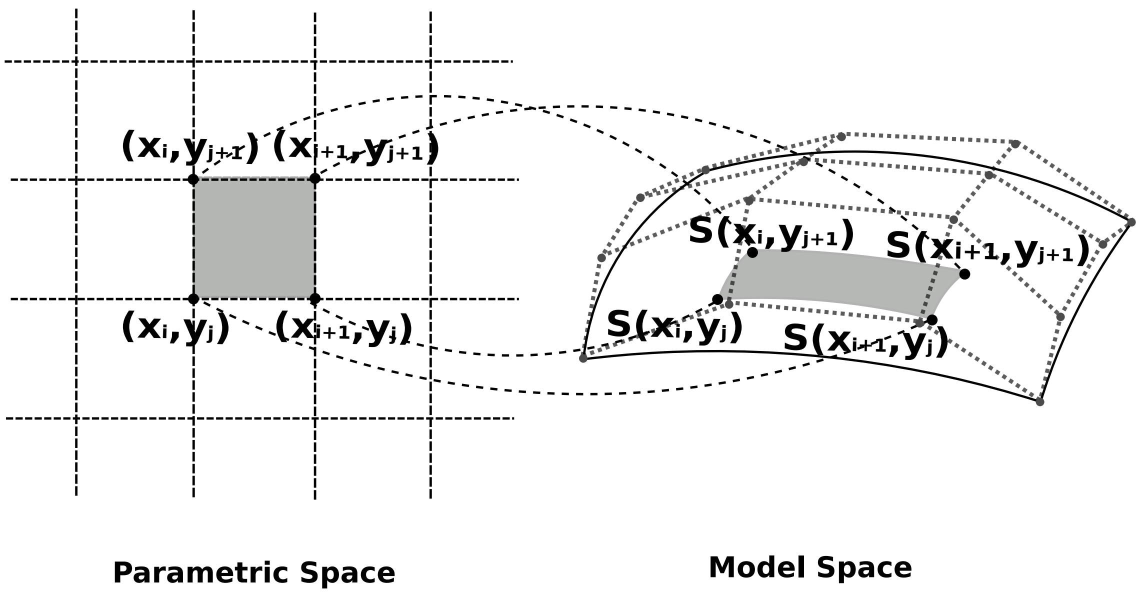


Figure 6.2: KSQuad primitive defined by a knot interval

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 highly 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 means that the data calculated for a given point into a cell can 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 $(r-2p)\times(s-2q)$:

$$\begin{aligned}
U &= \left[0,\cdots 0, \underbrace{0, \overbrace{x_{p+1}, x_{p+2}}^{KS_{p+1}}}_{KS_p}, \cdots \underbrace{x_{r-p-2}, \overbrace{x_{r-p-1}, 1}^{KS_{r-p-1}}}_{KS_{r-p-2}}, \cdots 1\right] \\
V &= \left[0,\cdots 0, \underbrace{0, \overbrace{y_{q+1}, y_{q+2}}^{KS_{q+1}}}_{KS_q}, \cdots \underbrace{y_{s-q-2}, \overbrace{y_{s-q-1}, 1}^{KS_{s-q-1}}}_{KS_{s-q-2}}, \cdots 1\right]
\end{aligned} \tag{6.6}$$

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

Therefore, a surface is defined by the following set of knot span:

$$\begin{aligned}
S \rightarrow \{&\underbrace{[0, x_{p+1}) \times [0, y_{q+1})}_{KS_{q,p}}, \underbrace{[x_{p+1}, x_{p+2}) \times [y_1, y_2)}_{KS_{q+1,p}}, \cdots, \\
&\underbrace{[x_r, x_{r+1}) \times [y_s, y_{s+1})}_{KS_{r-p-1,s-q-1}}\}
\end{aligned} \tag{6.7}$$

As a visual example, Figure 6.3 shows the *Head* model where each surface has a different color (see Figure 6.3(a)), and the KSQuads for this model (see Figure 6.3(b)). In this figure, each KSQuads has been rendered as a pair of triangles, since the model has been rendered in a triangle oriented pipeline with DirectX11.

6.3. 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) for providing high-quality, hole-free rendering, while 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 improved exploitation of the surface's local geometric features. Therefore, it solves an important problem in current GPUs: they cannot 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

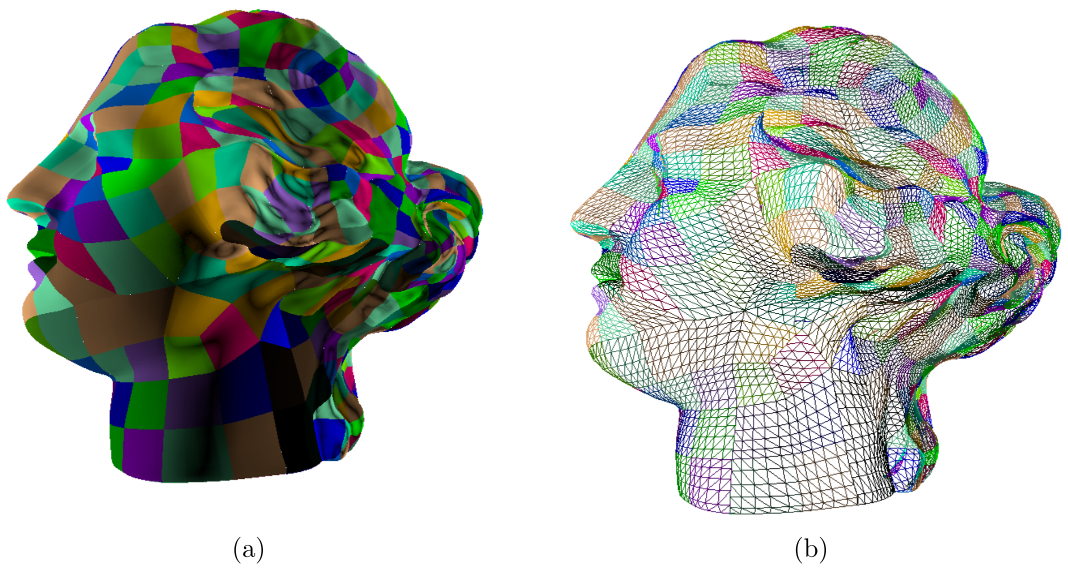


Figure 6.3: *Head* model: (a) Surfaces (b) KSQuads

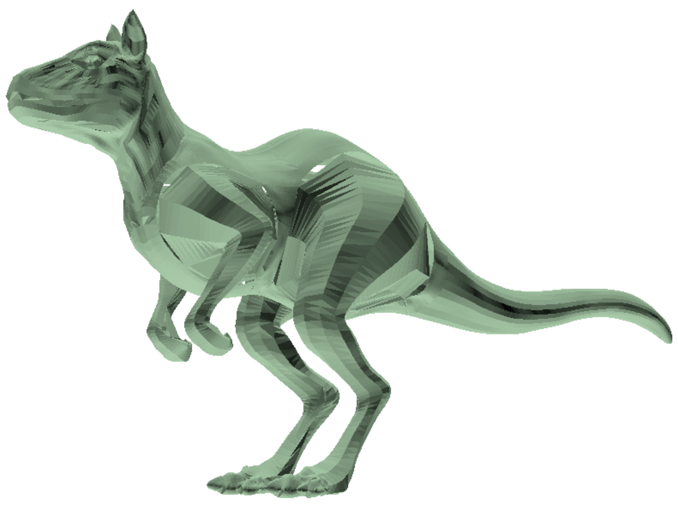
tessellation of a NURBS surface into triangles, it is preferable to select a tessellation factor per KSQuad than to compute a tessellation considering the whole NURBS surface. This is shown in Figure 6.4, where a model, *Killeroo*, is first rendered using surfaces (Figure 6.4(a)) as input primitives, and then KSQuads Figure 6.4(b)). As can be observed, the quality of the final image is much higher with KSQuads, even though significantly fewer triangles are used: 12016.05 K vs. 385.56 K. Figure 6.4(a) contains numerous artifacts, such as cracks, holes and creases, since it possesses insufficient triangles to follow the curvature of the surface. Thus, thirty times more primitives are evaluated but a lower quality rendered model is obtained.

A KSDice obtained from a KSQuad is defined as

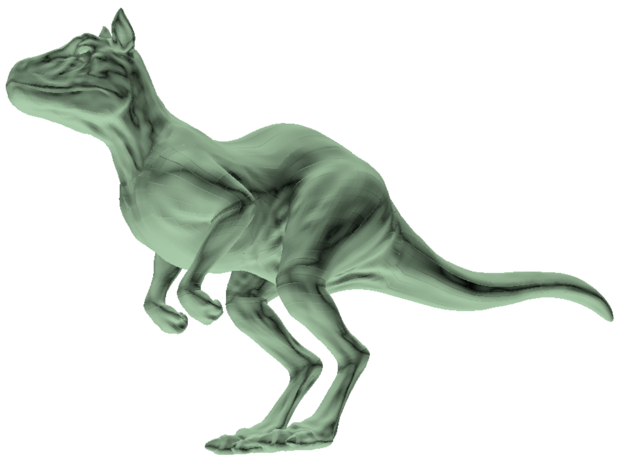
$$KSDice = \{(x_k, y_l), \delta_x, \delta_y, S^{p,q}, f\}$$

where (x_k, y_l) is the starting parametric coordinate of the range covered by the KSDice, 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 KSDice are boundary edges with another surface.

The key to obtaining good performance with RPNS is the number of KSDice that are sampled and rendered. An adaptive sampling that focuses on the geomet-



(a)



(b)

Figure 6.4: *Killeroo* model rendered using as input primitives: (a) NURBS surfaces with a uniform tessellation per surface (b) KSQuads, which are adaptive allowing triangles to be placed more appropriately

ric 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 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 viewpoint is evaluated. Then, a sampling factor τ is chosen and applied in parametric space to obtain the appropriate number of KSDice primitives from the projected KSQuad, considering a maximum pixel-size for each KSDice of μ:

$$\begin{aligned} \mathrm{dist}(p(S(x_k, y_l)) - p(S(x_m, y_n)))/\tau < \mu, \\ \forall k, m \in \{i, i+1\} \textit{ and } k \neq m \\ \forall l, n \in \{j, j+1\} \textit{ and } l \neq n \end{aligned} \tag{6.8}$$

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 KSDice 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

$$\begin{aligned} max\ abs(p(S(u, v)) - p(S'(u, v))) < \epsilon, \\ \forall(u, v) \in [x_k, y_l) \times [x_k + \delta_x, y_l + \delta_y) \end{aligned} \tag{6.9}$$

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:

$$\begin{aligned} max\ abs(p(S(x_k, y_l)) - p(S(x_k + \delta_x, y_l + \delta_y))) < \eta, \\ \eta < \mu \end{aligned} \tag{6.10}$$

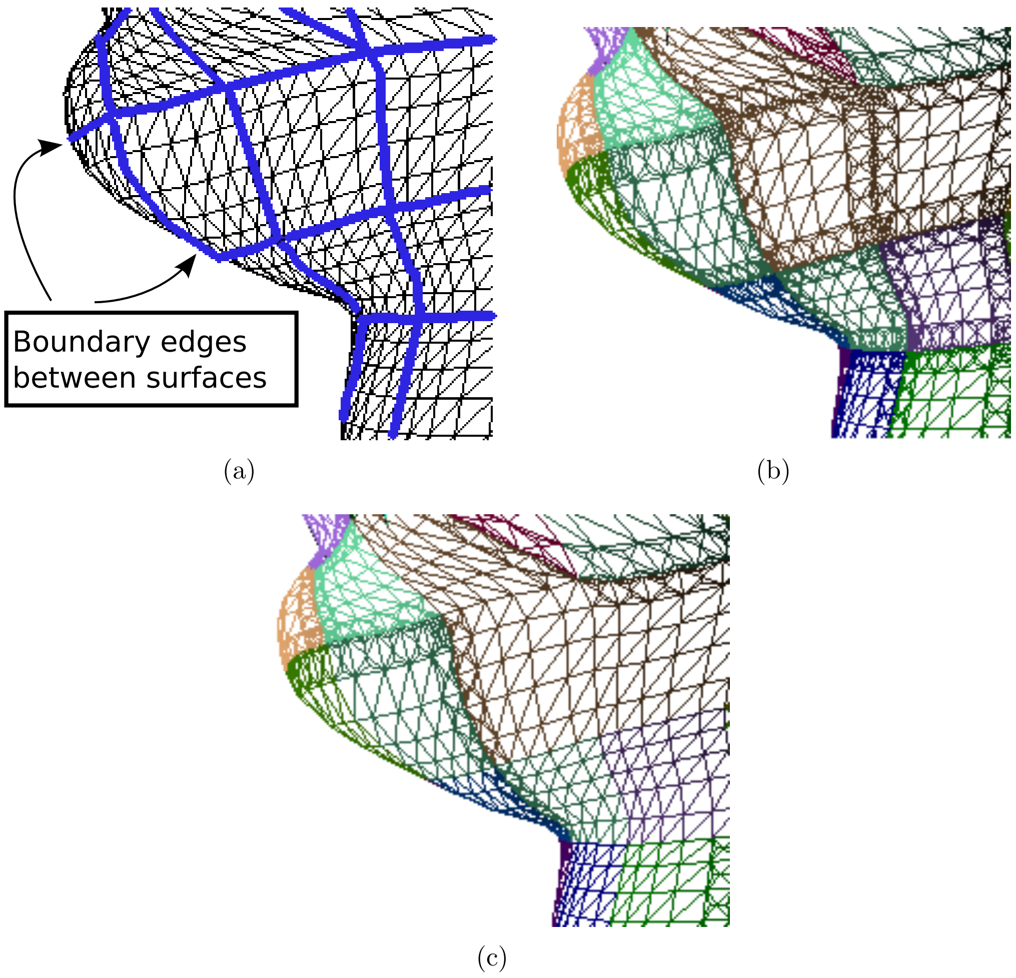


Figure 6.5: Boundary edges between surfaces (a) KSQuads with boundary edges (b) Oversampling all boundary edges (c) Oversampling only non G^1 boundary edges

An example is depicted in Figure 6.5(b): thick lines mark the boundary edges between surfaces in Figure 6.5(a), and Figure 6.5(b) shows how the boundary regions are higher sampled.

A better solution for dealing with possible discontinuities but avoiding over-tessellation is our second approach to this test, depicted in Figure 6.5(c). This new implementation is based on the notion 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 [17] 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) are analyzed in the application stage on the CPU only once. This results in a *boundary region* test that only applies a higher sampling on the boundary edges of a KSQuad when it is really needed, as can be observed in Figure 6.5(c).

These tests measure the improvement achieved in image quality with each new KSDice inserted by the sampler stage. As with other similar proposals [41], our tests work with a series of thresholds that must be pre-computed for each surface in order to reach the required quality level.

6.4. Culling techniques for NURBS surfaces

Culling is the process of removing those portions of the scene that do not contribute to the final rendering. The advantage of culling in the early stages of the rendering pipeline is that entire objects that are invisible can be removed, saving a great deal of computation in the rest of the pipeline. Examples of culling techniques are backface culling and viewing frustum culling. To optimize the rendering of NURBS surfaces in RPNS as well as a proof of the versatility of the primitive KSQuad, backface and viewing frustum culling methods for the geometry stage are proposed in RPNS. This culling makes possible an important reduction in the number of rendered KSDice, which has a high impact on the overall performance.

First of all, a view frustum culling based on the NURBS bounding box is performed at KSQuad level. A view frustum is a pyramid truncated by six planes: near, far, left, right, top, and bottom. This culling proposal is based on the strong convex hull property, described above in Section 6.2. Hence, those KSQuads placed outside of the viewing cone will be removed and only those KSQuads inside or partially inside the viewing cone will be considered in the following steps.

Lastly, a backface culling is applied. In RPNS, backface culling and backpatch culling strategies have been considered. These culling techniques are based on different processing characteristics and they are placed in different RPNS pipeline stages. Backface culling is placed at the end of the RPNS sampler stage (see Figure 6.1), where samplers called KSDices have already been built. Meanwhile, backpatch

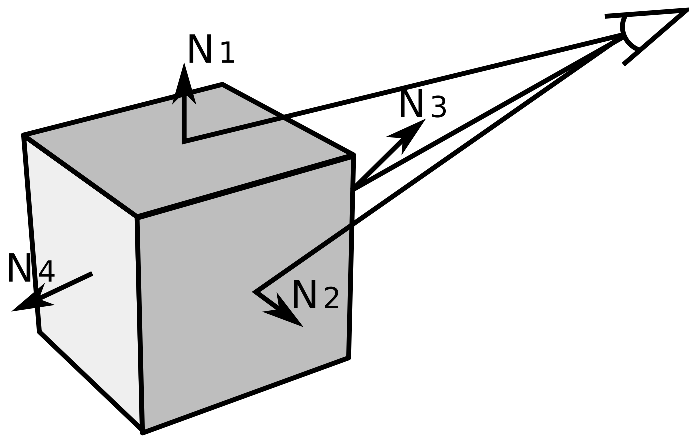


Figure 6.6: Backface culling

culling should be placed at the beginning of the geometry stage, the first step in the pipeline proposed (see Figure 6.1). In summary, backface culling is at KSDice level, while backpatch culling is at KSQuad level.

Broadly speaking, a culling algorithm generates what is called the *potentially visible set* (PVS), which is a prediction of the *exact visible set* (EVS) [2]. A PVS is conservative if it fully includes the EVS, so that only invisible geometry is discarded. Otherwise, a PVS is approximate when the EVS is not fully included, which results in rendered images with a certain error. In RPNS, three culling proposals have been implemented, EVS follows a backface culling algorithm while two backface culling algorithms, approximate and conservative, have been implemented.

6.4.1. Backface culling

Backface culling [2] is a traditional and standard technique used by GPUs and it is based on removing those polygons which are invisible from the viewpoint as early as possible. Backface culling is usually obtained by the computation of the dot product between the normal to the plane defined by the polygon and the viewing direction of each polygon (see Figure 6.6); i.e., a triangle t defined by $v1$, $v2$ and $v3$ is culled out, evaluating the dot product of the plane normal to the triangle t and the camera - to - polygon vector. In this figure, light gray face is placed on the backface to the camera, so this face is removed. Meanwhile, dark gray faces of the cube are turned to the camera and they will be rendered.

RPNS pipeline includes a backface culling proposal called DiceCulling (DC). As traditional pipelines are triangle oriented in the rendering stages, culling is usually computed on triangle polygons. However, as RPNS is based on the rendering of pieces of parametric surfaces, specifically KSDice, the DiceCulling proposal removes those KSDices not turned to the camera as early as possible. Although in RPNS a KSDice is the render primitive sampled from a KSQuad, in the implementation of RPNS on a DirectX11 pipeline, each KSDice is approximated as a pair of triangles.

The DiceCulling test is placed at the end of the sampler stage, after the screen-mapping procedure (see Figure 6.1). DiceCulling decreases the number of KSDices sent to the rasterizer, since backfaced KSDice are culled out. However, as backface culling computations are done in the sampler stage, in this stage workload is slightly increased meanwhile the rasterizer workload is reduced.

6.4.2. Backpatch culling

Nevertheless, backface triangles have proved to be an unsuitable solution for complex models due to their computational cost; thus, in order to reduce the cost of a dot product and normal computation for each polygon in backface culling, several proposals have been developed to approximate the plane normal. For instance, hierarchical proposal like [56, 57] cluster polygons by normal and cull the whole group. However, as RPNS proposes a direct evaluation of NURBS surfaces with no tessellation into polygons, a backpatch NURBS culling is also considered with the aim of reducing culling computations. Figure 6.7 depicts a sphere of surfaces. In this figure light gray surfaces are backfaced, so they are removed; meanwhile dark gray surfaces are frontfaced and consequently they are rendered.

Backpatch culling [58] is an extension of backface culling to parametric surfaces. Like backface culling, backpatch culling is based on removing these invisible patches as early as possible. In RPNS, the KSQuad is removed before being tessellated into KSDices. As with a polygon, a KSQuad is culled out by evaluating the dot product of the normal to the KSQuad and the $camera-to-patch$ vector.

Although backpatch culling is not a novel idea, to date it had only be applied

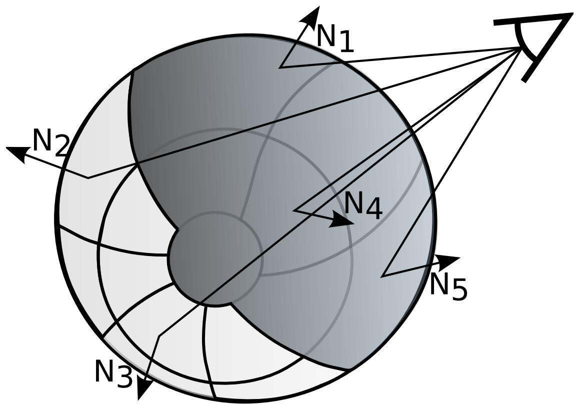


Figure 6.7: Backpatch culling

to Bézier patches. There are basically two different groups of proposals to compute backpatch culling: based on cone-of-normals [68, 86, 89] and based on the bounding box [58, 61]. In the first group, [86] proposes a cone-of-normals derived from tangent and bi-tangent patches, whose main drawback is the coarse bounds that are obtained. [89] presents a preprocessing step to compute a normal patch for a given Bézier patch and to compute its bounding cone-of-normals. Lately, a simple test is used to compute the culling on the fly. The main drawback of this approach is that dynamic models are not rendered in real time owing to the high computational cost. A recent proposal [68] is focused on fitting this algorithm into modern GPUs, which in turn means an approximation in the computation of the tangent and bi-tangent cone. RPNS, like [58] and [61], is based on the computation of the bounding box of the patches. [58] and [61] focus on Bézier surfaces, though. [58] computes the bounding box of the normalized vectors of the normal patch, whereas [61] constructs the Bézier convex hull of the parametric tangent plane.

Unlike Dice Culling which has been designed to remove KSDice, backface culling proposals, Light Quad Culling (LQC) and Strong Quad Culling (SQC), have been designed to remove KSQuad. Quad Culling algorithms are independently applied to each KSQuad. Hence the KSQuad of a NURBS can be culled out, while other KSQuads from the same NURBS surface are rendered. Both Quad Culling proposals are based on the potentially visible set (PVS) and they are placed at the beginning of the geometric stage; consequently, KSQuads are removed before to be sampled into KSDices. Although the workload in geometric stage is increased, sampler stage and rasterizer workload are reduced. As both Quad Culling techniques have been specifically designed for NURBS surfaces, they are based on the strong convex hull property detailed in Section 6.2. Hence, the normal to the plane defined by the convex hull of each KSQuad is computed instead of computing the normal to the KSQuad. Therefore, the normal computation workload is reduced and a more efficient culling technique is implemented.

The proposed Light Quad Culling algorithm culls a KSQuad$_{i,j}$ by using this simple square:

$$\square_{i,j} = \{S(x_i, y_j),\ S(x_{i+1}, y_j),\ S(x_i, y_{j+1}),\ S(x_{i+1}, y_{j+1})\} \tag{6.11}$$

LQC is an approximate PVS technique and although a fast and efficient culling computation is provided, the EVS is not fully included thus the quality of the render is slightly decreased as will be detailed in Section 6.7. $\square_{i,j}$ cannot guarantee that the normal of all surface points have the same direction.

On the other hand, the Strong Quad Culling (SQC) algorithm culls a $KSQuad_{i,j}$ by computing the culling for a set of $p \times q$ squares

$$\square_{i,j}^{k,l} = \{B_{i-p+k,j-q+l},\ B_{i-p+k+1,j-q+l}, B_{i-p+k,j-q+l+1},\ B_{i-p+k+1,j-q+l+1}\} \tag{6.12}$$

with $0 \leq k \leq p-1$ and $0 \leq l \leq q-1$. Each square is the convex hull polygon corresponding to the adjacent control points. If any of these squares is not culled, then the KSQuad is not culled. A $KSQuad_{i,j}$ is contained in the convex hull defined by the control points $\{B_{i-p,j-q}, \cdots, B_{i,j}\}$. That is, the NURBS surface fragment that defines the parametric subset $KSQuad_{i,j}$ is contained by the control net fragment, $p \times q$ squares. If any square is oriented to the viewpoint, then it is possible that any point in the surface is frontfaced. The control polygon represents a piecewise bilinear approximation to the surface. This approximation is improved by applying either knot insertion or degree elevation. As a general rule, the lower the degree, the closer the surface follows its control polygon, reaching the extreme case with $p = 1$, when the surface is the control polygon. SQC is a conservative PVS technique where the EVS is fully included, as only invisible geometries are discarded, and a high quality images are rendered.

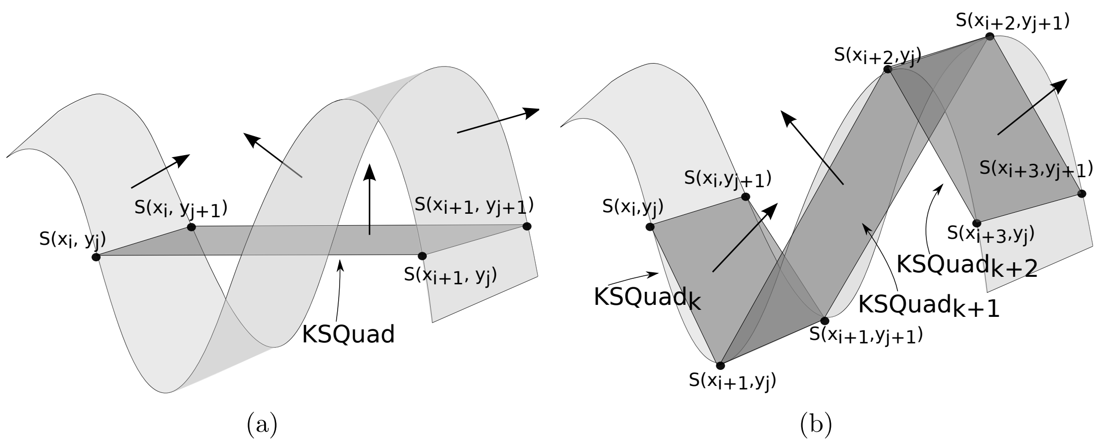


Figure 6.8: KSQuad-based culling (a) high degree NURBS (b) low degree NURBS

Figure 6.8 shows two different scenarios for applying our culling strategies: for a high-degree surface (Figure 6.8(a)), there are few knots in the NURBS and the difference between the results obtained by the approximative and the conservative strategies, $\square_{i,j}$ and $\square_{i,j}^{k,l}$, is greater; however, for a low-degree surface (Figure 6.8(b)) the squares $\square_{i,j}$ comes close to the actual surface, so a similar PVS is obtained in both strategies, although with an marked reduction in performance in the approximative method (as shown in Section 6.7).

The introduction of these Quad Culling techniques in RPNS results in an important reduction in the computational load of the sampler and rasterizer stages, although this is accompanied by a slight increase in the computation of the geometry stage.

6.5. Explicit equations: Stair strategy

Another relevant contribution in this chapter 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 which makes it suitable for implementation on current GPUs. Nowadays, NURBS surfaces evaluation on the GPU is usually based on the *de Boor* algorithm [55]. Thus, evaluating the B-splines basis function of degree p requires the evaluation of the B-spline basis function of degree $p-1$. In

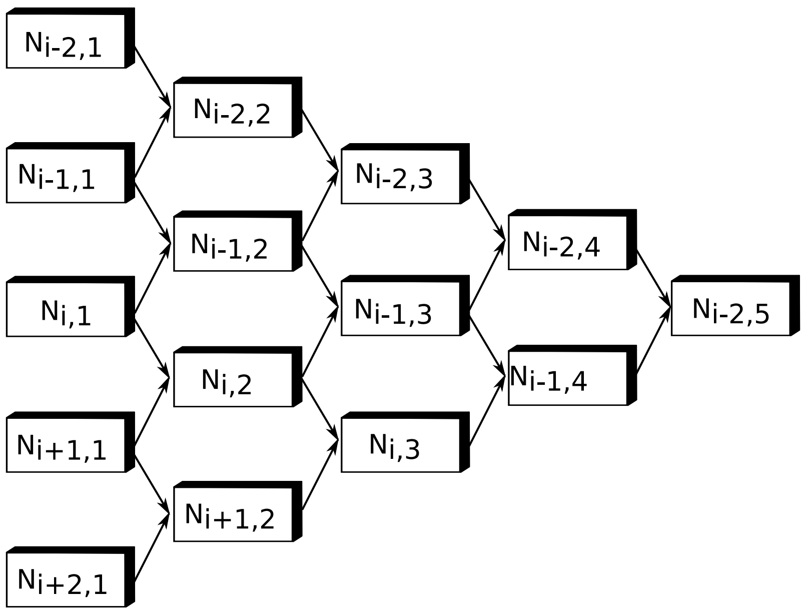
[55] the B-spline basis function of degree $p-1$ is stored as a texture on the GPU and this intermediate result is used as input for evaluating the B-spline basis function of degree p. In [1, 49] several approaches are presented for improving the performance on CPU of the computationally expensive *de Boor* recursion 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 *de Boor* recursive expression shown in Equation 6.3. As mentioned in Section 6.2, 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 6.1 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 $\{N_{i-5,5}, N_{i-4,5}, N_{i-3,5}, N_{i-2,5}, N_{i-1,5}, N_{i,5}\}$.

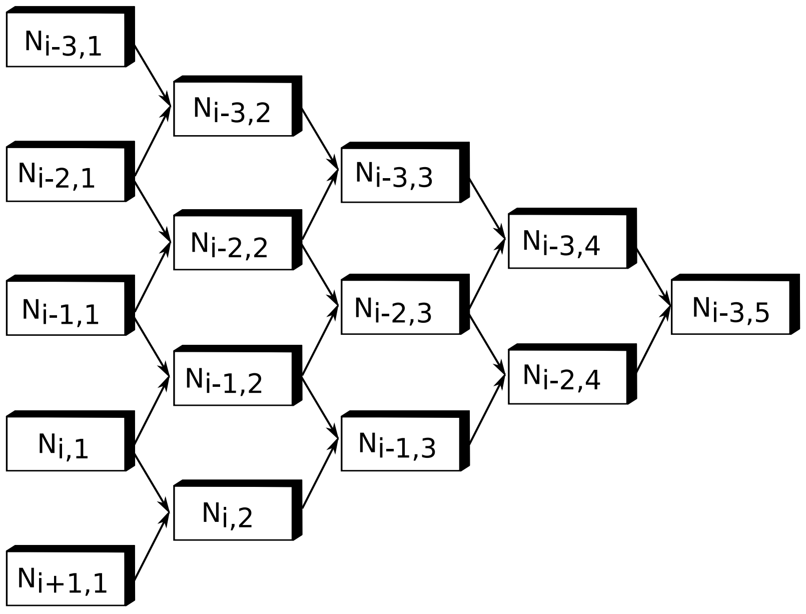
The RPNS proposal is based on the non-recursive reformulation of the $N_{i-k,p}$ functions, replacing the recursion that can be represented by a truncated triangular table [79] with a simple expression of additions and multiplications. Thus, each basis function can be represented like a rectangle table with size $(p-k) \times k$. Whereas *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 N basis functions. Figure 6.9 shows the dependence of $N_{i-2,5}$ and $N_{i-3,5}$, where according to the NURBS basis function definition each of these basis functions should be computed in a five-step recursive process. However, each basis function can be expressed as a rectangle table with a remarkable reduction of computation, as shown in Figure 6.10. This figure shows the rectangle tables for $N_{i-2,5}$ (Figure 6.10(a)) and for $N_{i-3,5}$ (Figure 6.10(b)), where the influence of each basis function is shown and subsequently the recursive representation is avoided.

Table 6.1: Non-zero 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	$i+1$



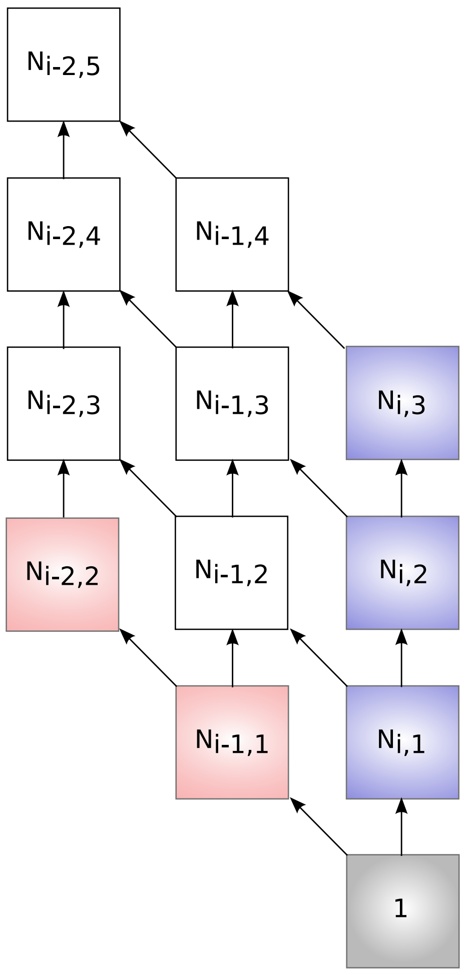
(a)



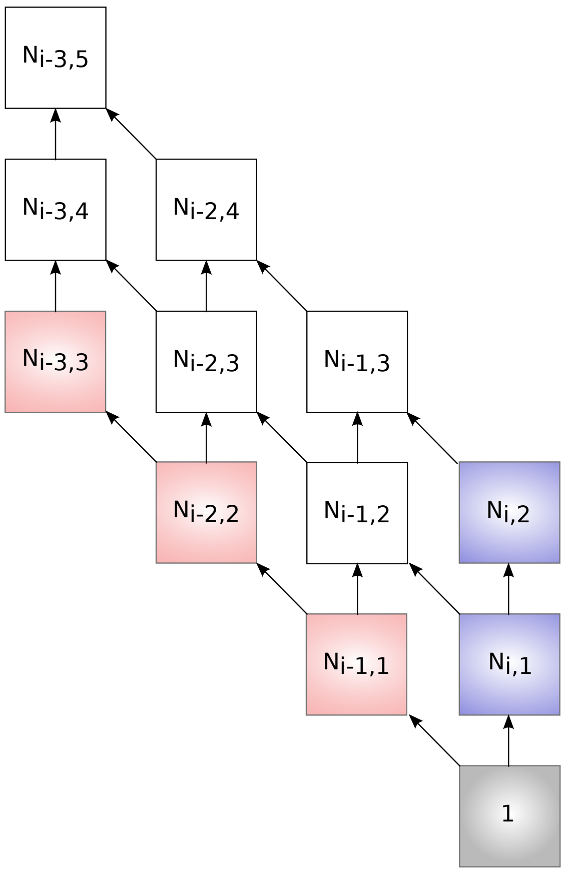
(b)

Figure 6.9: Dependence of (a) $N_{i-2,5}$ and (b) $N_{i-3,5}$

Consequently as this stair strategy is optimized by an efficient diagonal and columns computation, these diagonals and columns are highlighted in Figure 6.10. The basis functions $N_{i-k,p}$ with $k = \{0, \cdots, p\}$ can be formulated according to some of the



(a)



(b)

Figure 6.10: Stair strategy (a) $N_{i-2,5}$ (b) $N_{i-3,5}$

$N_{i,c}$ and $N_{i-d,d}$ with $c = \{1, \cdots, p-k\}$ and $d = \{1, \cdots, k\}$, that is, only a total of p non-recursive basis functions. We designate the basis functions $N_{i,c}$ as *column* functions and $N_{i-d,d}$ as *diagonal* functions. Stair strategy formulation is expressed as a piecewise function with three subdomains. The first subdomain details the formulation of column basis functions, where $k = 0$ (depicted as blue squares in Figure 6.10), the second subdomain comprises diagonal basis functions, where $k = p$ (depicted as red squares in figure) and, finally, the rest of basis functions, where $k \neq 0$ and $k \neq p$ (depicted as white squares). Stair strategy allows a simple and efficient computation of the column and diagonal basis functions by simply applying the following expressions:

$$N_{i,c} = \frac{(u-x_i)^c}{\displaystyle\prod_{l=1}^{c}(x_{i+l}-x_i)} \quad N_{i-d,d} = \frac{(x_{i+1}-u)^d}{\displaystyle\prod_{l=1}^{d}(x_{i+1}-x_{i-l+1})} \tag{6.13}$$

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

$$\begin{aligned} N_{i-k,p} = &\sum_{j=0}^{k-1} \frac{(u-x_{i-k+j})^{p-k}(x_{i+k-1+j}-u)^j}{\displaystyle\prod_{l=1}^{p-k+j}(x_{i+l}-x_{i-k+j})} N_{i-k+j,k+j} \\ &+ \sum_{j=0}^{p-k-1} \frac{(x_{i+p-k+1-j}-u)^k(u-x_{i-k})^j}{\displaystyle\prod_{l=0}^{k-1+j}(x_{i+p-k+1-j}-x_{i-l})} N_{i,p-k-j} \end{aligned} \tag{6.14}$$

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

$$C_{j,k,p} = \frac{1}{\displaystyle\prod_{l=1}^{p-k+j}(x_{i+l}-x_{i-k})} \quad D_{j,k,p} = \frac{1}{\displaystyle\prod_{l=0}^{k-1+j}(x_{i+p-k+1-j}-x_{i-l})} \tag{6.15}$$

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

$$\begin{aligned} N_{i-k,p} = &\sum_{j=0}^{k-1}(u-x_{i-k+j})^{p-k}(x_{i+p-1+j}-u)^j(x_{i+1}-u)^{k+j}C_{j,k,p}D_{0,k+j,k+j} + \\ &\sum_{j=0}^{p-k-1}(x_{i+p-k+1+j}-u)^k(u-x_{i-k})^j(u-x_i)^{p-k-j}C_{0,0,p-k-j}D_{j,k,p} \end{aligned} \tag{6.16}$$

Hence, Stair Strategy piecewise expression is defined as:

$$N_{i-k,p} = \begin{cases} (u - x_i)^p C_{0,0,p}, & if\ k = 0 \\ (x_{i+1} - u)^p D_{0,p,p}, & if\ k = p \\ Equation\ 6.16, & if\ k \neq p\ and\ k \neq 0 \end{cases}$$

Considering a bicubic NURBS surface, basis functions are:

$$\begin{aligned} N_{i,3} &= (u - x_i)^3 C_{0,0,3} \\ N_{i-1,3} &= (u - x_i)^2 (x_{i+1} - u) C_{0,1,3} D_{0,1,1} + (x_{i+3} - u)(u - x_i)^2 C_{0,0,2} D_{0,1,3} + \\ &\quad + (x_{i+2} - u)(u - x_{i-1})(u - x_i) C_{0,0,1} D_{1,1,3} \\ N_{i-2,3} &= (u - x_{i-2})(x_{i+1} - u)^2 C_{0,2,3} D_{0,2,2} + (x_{i+2} - u)(u - x_{i+1})(x_{i+1} - u) C_{0,1,3} D_{0,1,1} + \\ &\quad + (x_{i+2} - u)^2 (u - x_i) C_{0,0,1} D_{1,1,3} \\ N_{i-3,3} &= (x_{i+1} - u)^3 D_{0,0,3} \end{aligned}$$

6.6. 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.

Although the new configurable stage introduced in DirectX11, the tessellator unit, does not follow the same philosophy as the GS as it does not focus on data locality, this new stage solves the main limitation of the GS, the amount of primitives generated in the GPU on the fly. The Tessallator 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 KSDice in our implementation. The DS receives both the

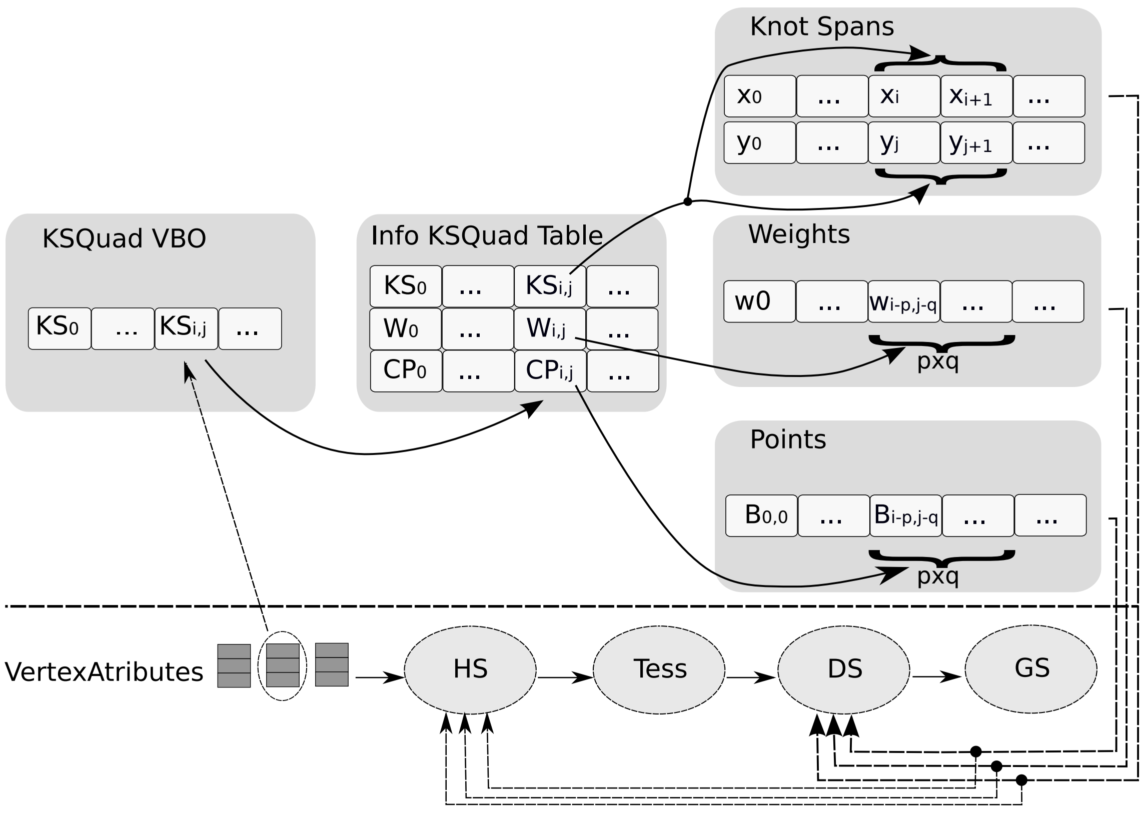


Figure 6.11: Memory layout of the data structures

parametric positions created by the tessellator and the KSQuad data directly from the HS. The four corners of each KSDice, $S(x_k, y_l)$, $S(x_k+\delta_x, y_l)$, $S(x_k, y_l+\delta_y)$, 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. The output from the DS is sent to the GS, where two triangles are generated for each KSDice due to the triangle-oriented graphics pipeline of current GPUs. We should emphasize that, like Reyes vertex shading, RPNS also allows the user to specify an arbitrary shading rate. In Reyes, the shading rate is expressed in samples per pixel, while in RPNS we specify samples per KSDice, with a value of 4.0 in our implementation.

Although DirectX11 introduces a *patch* primitive to deal with parametric surfaces, this primitive is only suitable for working with simple regular surfaces, such as bicubic 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 surface degree. Owing 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 6.11, 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 easily be 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 the Info KSQuad Table and fetch all the data needed to completely evaluate the KSQuad from texture memory. The access to all this information is needed in the HS and DS stages (see Figure 6.11), which correspond to the input of the geometry and sampler stages in RPNS, as shown in Figure 6.1.

We have implemented and tested three different alternatives to map RPNS on DirectX11, one which exploitsg the GS capability of generating new geometry, and the other two which use the HS, tessellator and DS stages to overcome the GS limitations. One of these two proposals implements a non-uniform version of RPNS which 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 evaluating KSQuads in the GS means that part of the computations can be reused, the bounds in the maximum number of KSDices that can be generated for each input KSQuad make it impossible to achieve high quality renderings. This proposal is not shown in the results section as its performance and the rendering quality is considerably worse than the other proposals as well as the GS not providing enough primitives for a smooth rendering. As it is shown in Figure 6.12, the number of generated primitives with two passes through the GS stage is insufficient to achieve an acceptable quality. Furthermore, two GS stage passes decreases the performance and real-time rendering is not achieved.

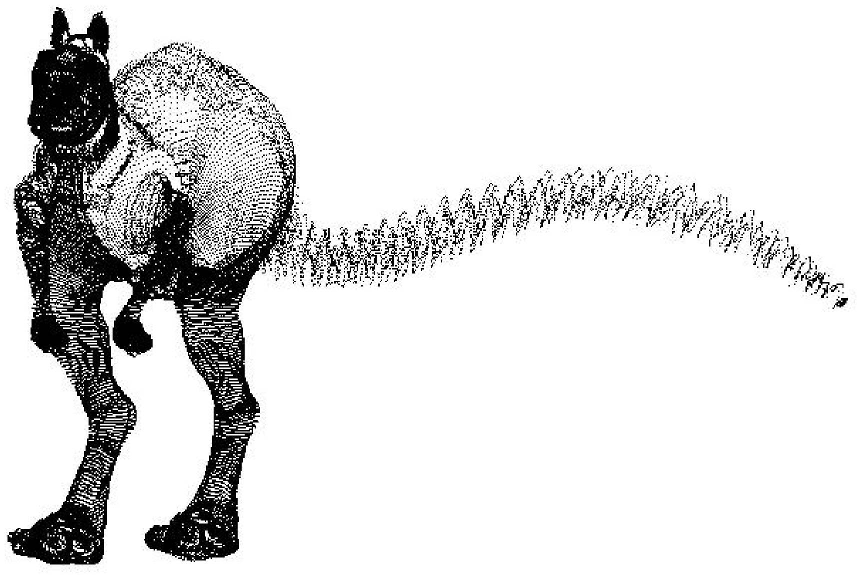


Figure 6.12: Killeroo model with the GS-based RPNS and two passes through the GS stage

Uniform RPNS (RPNS-U). This implementation uses the HS, Tessellator and DS stages to map the stages in RPNS. The work in HS is performed 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 KSDices to be generated (Equation 6.8). All this work corresponds with the geometry stage of RPNS.

Once the KSDices are created by the tessellator, they are sent to the DS. Thus, the DS is called once for each (still empty) KSDice. In this stage, each KSDice is evaluated in the NURBS surface using the *stair strategy* described in Section 6.5. This stage, along with the tessellator, corresponds with the sampler stage of RPNS.

Non-uniform RPNS (RPNS-NU). This non-uniform implementation of RPNS follows the same structure as the previous one, but with the addition of a GS stage which implements the adaptability in the sampler stage of RPNS [10]. In this case, the HS previously sets the tessellator to create a fewer number of KSDices. The output from DS is sent to GS, where the *linear approximation* and the *boundary region* test are used to guide the subdivision level applied to each KSDice. Thus, the linear approximation test (Equation 6.9) ensures that a higher subdivision level is applied to non-flat regions. Moreover, the boundary region test (Equation 6.10) detects the regions of KSQuads that are boundaries to other surfaces and applies the highest subdivision factor to prevent cracks between adjacent surfaces.

6.7. 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 6.13 and Table 6.2 depicts the number of NURBS surfaces and KSQuads, $\#NS$ and $\#KS$, respectively, in the models. As shown in Table 6.2 a high $\#KS$ and a low $\#NS$ are desirable due to the fact that a high amount of $\#KS$ provides a high flexibility and adaptivity inside the NURBS surface meanwhile a low $\#NS$ decreases the continuity gaps, because they can only be introduced on surface edges. The final images were rendered with a screen resolution of 2048×1152 pixels.

Different aspects of the RPNS implementation have been analyzed in this section. Firstly, different culling techniques are analyzed in the standard RPNS-U approach. This analysis focuses on the number of primitives generated, the quality of the rendered images, as well as the frame rate achieved. Recently, different RPNS implementations have been considered to measuring the effectiveness of the adaptive process that our KSQuad primitive allows us to introduce in RPNS. Specifically,

Table 6.2: Number of surfaces and KSQuad for each test model

Test model	$\#NS$	$\#KS$
Killeroo	89	11532
Head	601	15025
Hinge	427	34891
Car	1364	63000

Table 6.3: #KS with different culling techniques for *Killeroo* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	1564.79	485.76	178.47	83.16	51.20	45.05
SQC	1564.79 100%	378.63 77.95%	139.71 78.29%	83.16 100%	51.19 100%	45.05 100%
LQC	841.98 53.81%	259.90 53.5%	94.82 53.13%	43.60 52.43%	26.51 51.78%	23,10 51.27%
DC	847,16 54.14%	261,89 53.91%	95,72 53.63%	44,18 53.13%	26,98 52.7%	23,95 53.17%

Table 6.4: #KS with different culling techniques for *Head* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	2309.04	742.09	225	115.96	69.09	58.69
SQC	1829.71 79.24%	554.62 47.74%	199.82 88.81%	91.32 78.75%	54.26 78.53%	45.91 78.23%
LQC	1211.38 52.45%	368.28 49.63%	131.23 58.33%	58.87 50.76%	34.24 49.55%	29.46 50.20%
DC	1223.83 53%	368.87 49.71%	131.85 58.6%	59.78 51.55%	34.96 50.59%	29.74 50.68%

Table 6.5: #KS with different culling techniques for *Hinge* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	4245.46	1593.56	576.54	263.69	163.00	139.56
SQC	2977.54 29.86%	907.97 43.02%	325.36 43.56%	236.42 10.34%	91.07 44.13%	77.52 44.45%
LQC	2373.27 44.09 %	716.07 55.06%	251.76 56.33%	111.00 57.90%	66.49 59.21%	55.11 60.51%
DC	1578.48 62.81 %	716.35 55.04 %	252.02 56.28%	83.96 68.16%	66.78 59.02%	55.57 60.18%

Table 6.6: #KS with different culling techniques for *Car* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	4448.50	1493.51	607.11	332.07	257.90	246.09
SQC	3444.57	1149.10	459.18	247.62	190.12	180.09
	56.81%	56.22%	55.25%	53.89%	52.53%	52.39%
LQC	2527.32	839.68	335.44	178.94	135.47	128.93
	77.43%	76.94%	75.63%	74.57%	73.72%	73.18%
DC	2520.33	839.08	311.96	180.35	136.83	123.05
	56.66%	56.18%	51.38%	54.31%	53.06%	50%

Table 6.7: PSNR with different culling techniques for *Killeroo* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	44.51	42.95	40.75	39.28	39.30	36.74
SQC	44.53	42.95	40.75	39.28	39.30	36.71
LQC	39.16	38.64	38.22	37.51	37.486	36.22
DC	44.50	42.95	40.75	39.28	39.30	36.72

Table 6.8: PSNR with different culling techniques for *Head* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	42.37	43.93	42.97	40.87	40.53	36.26
SQC	42.36	43.95	42.57	40.87	40.52	36.26
LQC	38.88	38.41	38.21	38.18	38.05	35.07
DC	42.36	43.86	42.57	40.87	40.51	36.26

Table 6.9: PSNR with different culling techniques for *Hinge* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	41.89	41.39	39.77	40.94	40.30	38.77
SQC	41.11	41.42	39.78	40.20	39.93	38.55
LQC	40.60	41.29	39.70	40.03	39.26	37.89
DC	41.02	41.65	39.94	40.18	39.92	37.92

Table 6.10: PSNR with different culling techniques for *Car* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	38.64	38.35	40.47	38.47	37.58	30.45
SQC	38.54	38.34	40.48	38.47	37.58	30.18
LQC	34.87	34.47	35.25	34.78	34.23	30.02
DC	38.53	38.34	40.45	38.44	37.57	30.17

Table 6.11: FPS with different culling techniques for *Killeroo* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	27.91	76.14	152.7	199.86	265.85	281.73
SQC	34.46	87.51	162.78	189.21	213.12	225.61
LQC	47.44	107.12	179.9	204.24	231.65	234.87
DC	26.43	73.1	165.21	250.68	487.543	405.87

Table 6.12: FPS with different culling techniques for *Head* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	17.90	41.84	113.03	115.25	134.128	218.98
SQC	21.44	49.83	54.56	69.99	104.39	195.53
LQC	33.67	90.24	122.52	131.8	144.86	208.95
DC	20.66	54.6	158.48	300.51	404.93	434.12

Table 6.13: FPS with different culling techniques for *Hinge* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	9.27	27.57	35.22	117.42	134.16	158.27
SQC	15.35	44.49	83.42	105.50	122.17	130.03
LQC	19.01	53.73	100.65	113.36	127.73	132.92
DC	8.55	26.21	64.94	117.95	156.30	159.82

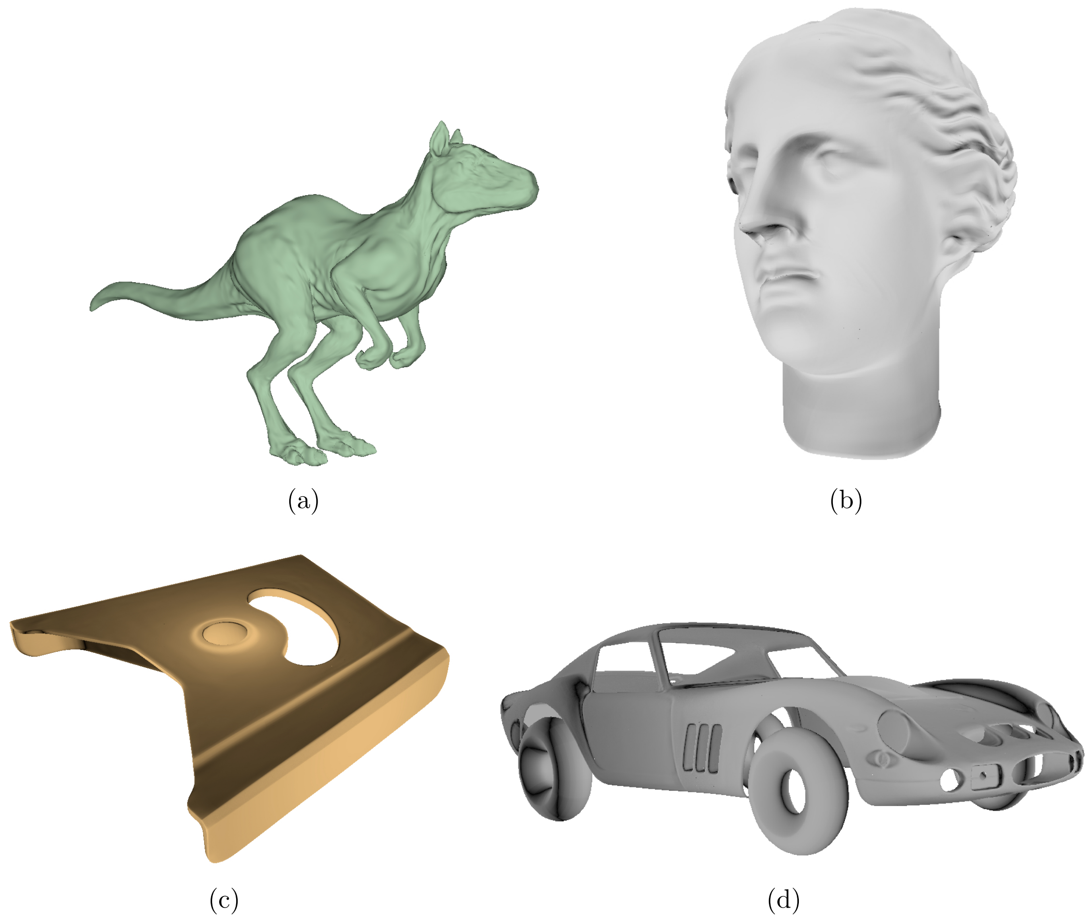


Figure 6.13: Test models: (a) *Killeroo* (b) *Head* (c) *Hinge* (d) *Car*

RPNS approach with different adaptivity degrees have been tested considering *de Boor* approach as the baseline.

The results obtained from the tests are shown in detail in Tables 6.3 - 6.14 and in Figure 6.14 with the RPNS-U approach. The experiments were carried out for the four test models for different culling strategies and with different values of the threshold μ (maximum pixel-size for each KSDice, see Equation 6.8).

Results have been grouped into three different groups of tables. Tables 6.3 - 6.6 indicate the thousands of KSDices that are rendered, #KS. The second group of tables (Tables 6.7 - 6.10) detailes PSNR (Peak Signal-to-Noise Ratio in dB) in order to indicate the performance in terms of quality. Peak Signal-to-Noise Ratio is the distortion between the maximum possible power of a signal and the power

Table 6.14: FPS with different culling techniques for *Car* model

	$\mu = 1$	$\mu = 2$	$\mu = 4$	$\mu = 8$	$\mu = 16$	$\frac{KSDice}{KSQuad}$
No Culling	10.23	20.31	33.3	91.36	96.84	104.73
SQC	12.85	34.9	58.9	65.59	66.09	84.4
LQC	17.06	44.37	70.56	79.98	82.73	85.6
DC	9.59	29.91	59.18	93.65	102.09	105.65

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 $\text{PSNR} = 20 \cdot \log_{10}(\text{MAX}/\sqrt{\text{MSE}})$. As can be observed, similar results have being obtained in all cases. Finally, the fame rate achieved, FPS, is shown in Tables 6.11 - 6.14 in order to depict the performance in terms of rendering time.

An adaptive sampling of KSQuad allows the utilization of larger KSDices where the geometry is less detailed, concentrating smaller KSDices at silhouette edges and curves. As shown in Tables 6.3 - 6.6 the $\#KS$ is not linear and increases when μ is increased. As adaptive sampling focuses on the geometric characteristics of the NURBS surfaces, $\#KS$ depends on the geometric characteristic of the model. Regarding the performance of RPNS implementation in the pipeline of current GPUs, the graphs of Figure 6.14 show the good results in terms of frame rate obtained by RPNS with $\mu > 1$ for the four test models. Furthermore, four strategies achieve good quality results, always over $30dB$ and close to $40dB$ for $\mu \leq 16$, with no significant loss of quality. Commonly accepted reference values for PSNR are between 20 and $40dB$ (the bigger, the better). A value higher than 30 dB usually means that a good quality result has been achieved.

For each culling technique implemented, the table shows the percentage of KSDices eliminated. The Strong Quad Culling approach culls up to 44.45% of primitives, but there are still 1.4 times more KSDices than the strictly necessary for the rendering. The Light Quad Culling culls some KSDices that should be in the rendering, so the quality of the render (PSNR value) decreases slightly. Regarding the frame rate obtained by the different culling approaches for the four test models, the graphs of Figure 6.13 show that the proposals based on the KSQuad primitive discussed in this work achieve the best results for $\mu < 8$. Thus, for example, speedups of 2.86 and 2.37 are obtained for $\mu = 4$ by the LQC and SQC, respectively. However, as the value of μ rises, the number of KSDices generated for each KSQuad decreases (see Table 6.5); this means that for values lower than 300 K KSDices the high computational cost of the HS software implementation, that has a significant degree of divergence, spoils the advantage achieved by reducing the number of KS-Dices to be rendered in cases with similar tessellation factors for all the surfaces, so the frame rate drops. In this case, the best results are obtained by the Dice Culling implementation, since the backface KSDices are removed with very simple computations with no divergence on the GS. Otherwise, the Quad Culling implementations may still be worthwhile for values between 300 K and 200 K KSDices in models such as *Killeroo*. In this model, there are KSDices with much higher tessellation factors than others as they have a greater area in the projected image. These large KSQuads are an important bottleneck, so performance dramatically improves when they are culled. Thus, 182.7 K KSDices are generated without culling with $\mu = 4$ for the *Killeroo* model, achieving 179.9 and 165.21fps with the LQC and the DC solutions, respectively.

The tables and the graphs show that the introduction of DC culling dramatically improves the performance of the pipeline, with speed-ups of more than 3x in the frame rate in some cases, and without decreasing the quality in the rendering, since the PSNR values are mostly identical. In these cases, the number of primitives culled in an early stage of the pipeline is not worth the additional computation introduced with these backpatch strategies such as LQC and SQC. Backpatch culling strategies are applied to KSQuad primitives in an early pipeline stage, while KSDice primitives are culled out in a latter GPU pipeline stages in a more classical backface culling approach. Consequently, backpatch culling strategies evaluate considerable fewer primitives than the backface culling strategy. However, as more computational complexity is introduced by the backpatch approaches, better results in terms of FPS have been obtained with a backface design. In any case, the final renderoing obtained by RPNS is a high quality image, as shown in Figure 6.15. To sum up, the culling techniques proposed in Section 6.4 remove an average of around a 20% of the KSDices generated in the case of the SQC method and over a 41% when the LQC is applied. Both strategies achieve good results in terms of quality, always over 30 dB, as has previously been mentioned.

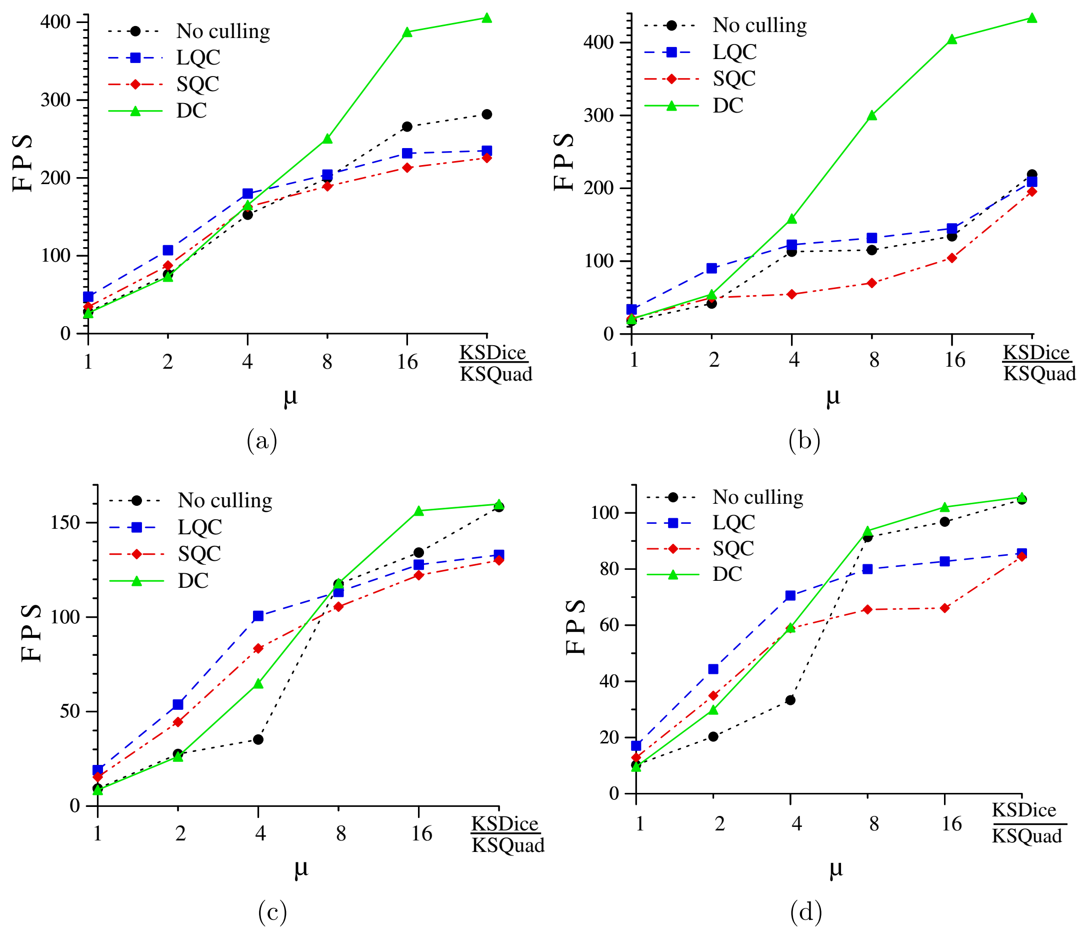


Figure 6.14: Frame rate with the different culling approaches for the four test models: (a) *Killeroo* (b) *Head* (c) *Hinge* (d) *Car*

The performance of several RPNS implementations is analyzed in Table 6.15 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 speed-up values in parentheses. The first method, *de Boor*, is the result of applying Dice Culling on the KSQuads in the GS, evaluating the NURBS Surfaces in a traditional way with $\mu = 16$. Simply by changing the surface evaluation to our *stair strategy*, second column in the table, we achieve a significant improvement in performance. Hence, a NURBS surface evaluation specifically designed for a GPU computation, which has been detailed in Subsection 6.5, provides a more efficient evaluation than the classical *de Boor* proposal, where the speed-up is up to 1.62. The next two columns, which both follow the stair strategy, *RPNS-U* which is a

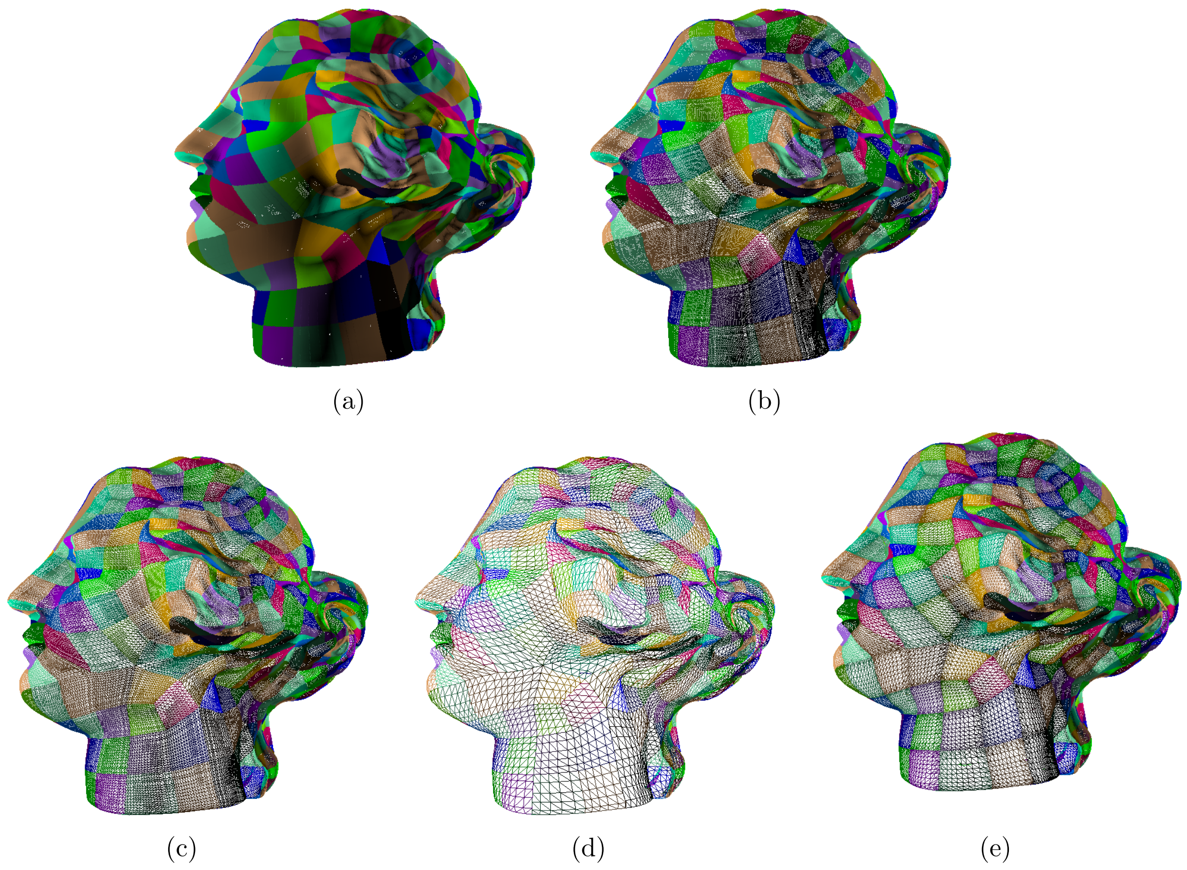


Figure 6.15: *Head* model rendered with different screen area (a) $\mu = 1$, (b) $\mu = 8$ and (c) $\mu = 16$ (d) 1 KSDice per KSQuad (e) RPNS-NU

uniform RPNS and *RPNS-NU (GS)* which is a-non uniform RPNS in the geometry shader, correspond to the implementations proposed in Section 6.6. As can be observed, in general the uniform approach obtains the best results, since it allows the HS, tessellator and DS to be explointed by using regular operations. However, the non-uniform proposal, based on some conditional branches, introduces a greater divergence into the control flow of the GS stage, and also calls a supplementary kernel, which results in a significant drop in performance. Finally, the last column, which also follows the stair strategy *RPNS-NU (HS)* which is a non-uniform RPNS implemented in the hull shader, 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.

Table 6.15: Frame rate (FPS) of RPNS implementations with Dice Culling

						(GS)		(HS)
Model	de Boor		Stair Strategy		RPNS-U	RPNS-NU		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

In conclusion, although the DC approach provides a better performance in terms of fps, the RPNS pipeline has been implemented in a triangle-oriented pipeline, and theoretically in a RPNS hardware implementation improved performance should be obtained by applying the backface culling, where a considerable less amount of KSQuad and KSDice will be processed. Furthermore, this chapter describes different RPNS implementations where different GPU stages are exploited. As has already been detailed, the stair strategy has been specifically designed for a GPU evaluation avoiding recursive computation and consequently better performance than the *de Boor* strategy has been obtained. Furthermore, as expected, a uniform implementation (RPNS-U) provides better performance than a non-uniform implementation (RPNS-NU), owing to the fact that computational divergence has been avoided. Finally, in a non-uniform implementation, HS provides a more efficient divergence implementation owing to the well-known GS restrictions as well as to a more adaptive tessellation pattern than the HS.

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, performing a test implementation on current GPUs and achieving real-time rendering for complex models.

6.8. Conclusions

In this chapter 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, which provides a regular and flexible management of NURBS surfaces but maintains their main geometric properties. This primitive allows an efficient evaluation of the NURBS surface in all its parametric knot spans, which is especially suitable for achieving high performance GPU implementations.

To provide a versatile example of RPNS, we implemented several culling strategies using the KSQuad primitive and its strong convex hull property. RPNS performs an efficient adaptive discretization of KSQuads into KSDices, which allows us to fine-tune the density of primitives needed to avoid cracks and holes in the final image. It also applies an efficient non-recursive evaluation of the basis function of a NURBS surface on the GPU. An implementation of RPNS using current GPUs is presented, achieving real-time rendering rates of complex parametric models. Different GPU implementations of RPNS are proposed on DirectX11, exploiting the programmable and configurable stages.

Chapter 7

Conclusions and Future Work

The graphics processing unit (GPU) is a specialized processor designed to accelerate the rendering of computer images. From this initial design, GPU hardware architecture has gone from a single core with a fixed function hardware pipeline specifically designed for graphics, to a set of highly parallel and programmable cores for more general purpose computation. Despite the rapid development of the GPGPU (general purpose graphics processing unit), the main purpose of GPUs is the rendering of graphics computing.

Although GPU pipelines were initially designed as a triangle-oriented pipeline, it has proved to be an unsuitable solution for the rendering of the complex images employed on a wide range of areas, such as medicine, computer-aided design (CAD) or virtual reality. As those models employed in scientific areas can be more precisely described by equations than by a triangles mesh, current research in the computer graphics area is considering other render primitives instead of triangles; more specifically, parametric surfaces have been proved as a flexible solution. The main objective of this dissertation is to develop a real-time rendering of parametric surfaces solution aimed to the new programmability capabilities presented in current GPUs.

This thesis proves that a parametric surface-oriented pipeline is a perfect fit for the rendering of complex models, as these surfaces provide mathematical characteristics specifically suitable for graphic computation. These surfaces have a compact representation, thus there is a saving in memory storage and much fewer data need to be sent than when sending a triangle mesh. Furthermore, this representation allows a more flexible evaluation and tessellation of the model because if a parametric surface is tessellated in the GPU, then the level of detail can be selected on the fly. As parametric surfaces are represented as a mathematical expression, they provide a more precise representation than a mesh of triangles.

This dissertation proposes different GPU approaches for the rendering of parametric surfaces. Firstly, the Bézier surface is considered as the input parametric surface and, finally, a NURBS (Non Uniform Rational B-Spline) surface approach is proposed.

Firstly, this thesis details a non-adaptive proposal for the tessellation of Bézier surfaces on the GPU based on the exploitation of spatial coherence of data within each surface. Furthermore, each Bézier surface is considered as an input pipeline primitive instead of an independent set of samples or triangles, since this non-adaptive proposal has been designed to tessellate and evaluate Bézier surface in the GPU. Additionally, an efficient Bézier evaluation as well as an optimized memory access have been designed. Two alternatives have been designed and implemented considering different GPU architectures, thus the VST (Vertex Shader Tessellation) alternative has been designed for those GPUs without any primitive generator and the GST (Geometry Shader Tessellation) alternative has been designed for GPUs with a geometry shader.

In one hand, VST is a tuning strategy that permits the choice of a suitable relation between the requirements of storage and the number synchronization CPU-GPU according to the underlying GPU architecture. It focuses on the efficient utilization of a parametric maps of virtual vertices, owing to the impossibility of generating geometries in the GPU. The VST alternative is based on three different keys: firstly, API draw calls and CPU-GPU communications are reduced and finally, data locality is exploited.

In the other hand, GST is based on the capabilities of a primitive generation in the GPU. As in this case parametric maps of virtual vertices are generated on the fly, storage requirements are reduced and primitive generator capabilities are exploited. The GST alternative performs an efficient Bézier evaluation prior to the sampling process and although Bézier computation is reused for each new generated point, it is only computed once for each surface. As the number of new primitives that can be generated is restricted, owing to hardware features, the level of detail of the render is limited. Therefore, in the GST alternative the resolution level is increased by partitioning the parametric map into zones, where a Bézier surface is computed once for each partition it is split into.

Both alternatives prove the impact on the performance of the shader without divergence and they highlight the benefits of the exploitation of the spatial coherence. The non-adaptive proposal demonstrates that an evaluation based in parametric surfaces can produce a real-time rendering of complex models where the model is tessellated and rendered in the GPU with a level of detail selected on the fly.

This thesis also analyzes the advantages and disadvantages of an adaptive tessellation; thus a fully adaptive approach called Dynamic and Adaptive Bézier Tessellation (DABT) is proposed. Unlike non-adaptive proposals, which can generate meshes with a high number of triangles without any contribution to the quality to the rendered image, DABT proposes an adaptive tessellation and, consequently, a considerable reduction in the number of triangles is obtained without reducing the quality of the final scene.

DABT provides a tessellation procedure without any recursive structure and where the positions of the candidate vertices can be easily evaluated through their barycentric coordinates. Furthermore, the tessellation procedure can be determined on the fly without any set of pre-computed patterns according to the tessellation level and the resolution level can be dynamically selected. The methodology employed is based on three main strategies: the use of a fixed tessellation pattern to guide the procedure, the use of local tests for the adaptive tessellation decisions and an efficient meshing procedure to reconstruct the resulting meshes. As DABT follows an adaptive approach, the resolution level can be dynamically modified along the path, thus preventing different resolution levels being selected in neighbor triangles. DABT has three different resolution levels inside a triangle, one resolution level for each triangle edge. In order to apply three levels per triangle, resolution levels are projected through the barycentric center and each resolution level is applied to one third of the triangle.

DABT introduces an early adaptive tessellation where much fewer triangles are rendered than in a non-adaptive proposal; however, performance is worse than in the non-adaptive proposal in current GPUs. Hence, this dissertation reasserts the drawbacks of the divergence in a GPU core, since DABT performance is reduced owing to the divergence introduced by the adaptivity. Therefore, and in order to maintain the more relevant features of both schemes, a semi-adaptive proposal is also outlined.

The semi-adaptive scheme is proposed as a halfway solution which exploits the best characteristic from the non-adaptive and the fully adaptive proposals. The objective of semi-adaptive proposal is to increase the processing speed of the DABT by reducing its flexibility. Hence this tessellation scheme reduces the divergence in order to achieve an optimum utilization of the GPU's computational resources, even though a significant degree of adaptivity has been introduced. Hence, this proposal processes considerably fewer triangles than a non-adaptive proposal, although the divergence caused by this adaptivity is considerably reduced.

The semi-adaptive algorithm is a simplified version of the fully adaptive strategy, and it is based on a single level of resolution per triangle and a regular grid pattern in the parametric directions. Furthermore, the tests are only applied in the candidate positions located in the original edges of the coarse triangle. According to the inserted vertex in the triangle edges, the insertion is also performed along the row in all the candidate positions inside the triangle. Hence as the semi-adaptive scheme is a more simplified scheme than DABT, and similar to the non adaptive proposal, it allows the GPU's computation capabilities to be exploited.

As nowadays, handheld devices are by far the most available device with rendering capabilities in the world, this dissertation also describes a proposal for the tessellation of Bézier surfaces in these devices. Handheld devices are widely available, virtually omnipresent and one of the fastest growing markets. As consumers demand complex rendering capabilities and their expectations for these devices are increasingly higher, a new GPU generation has been specifically designed to fit with the constraints of handheld devices. Handheld devices are small in size and are battery powered, thus they have been designed according to a restricted set of features. Hence, the GPUs in these devices implement only a subset of the features available in desktop GPUs. More specifically, these GPUs have been designed to offer a high performance graphics while reducing power consumption. This dissertation describes the design of several hand-tuned Bézier surfaces real-time rendering, Vertex Shader Tessellation in Handheld Devices (VSTHD), and identifies the key graphics processor performance limitations, enhancements and tuning opportunities.

VSTHD has been designed according to the constraints on the GPU in handheld devices. As handheld devices implement GPUs with no hardware for generating primitives, VSTHD is based on virtual parametric grids and memory exploitation, like the VST proposal for desktop GPUs. However, a small memory is implemented on current handheld devices, thus two different alternatives have been implemented to analyze different memory characteristics: Uniform VSTHD, which exploits uniform variables memory; and texture VSTHD, which stores data in texture memory.

In summary, this dissertation designs different Bézier tessellation schemes and the capabilities of current GPUs, from both desktop PCs and handheld devices, to the tessellation and evaluation of Bézier surfaces have been tested. As Bézier surfaces have a very simple and regular representation, they have commonly been used for the rendering of parametric surfaces. This thesis asserts that these surfaces fit well into the evaluation and tessellation in the GPU.

Finally, as NURBS surfaces are much more complex than Bézier surfaces, in a parametric surface oriented pipeline NURBS are usually converted into Bézier surfaces in the CPU and finally these Béziers are sent down the GPU pipeline to be rendered. However, this dissertation goes a step further and proposes an efficient scheme for the evaluation and tessellation of NURBS in the GPU.

This thesis also introduces two new primitives: the input primitive, called KSQuad, and the rendering primitive, called KSDice. KSQuad is a input primitive which mantains the main geometric properties from the original NURBS surfaces and it also provides a regular and flexible management of NURBS surfaces, which reduces divergence on GPU evaluation. This primitive allows an efficient evaluation of the NURBS surface in all its parametric knot spans, which is especially suitable for achieving high performance GPU implementations. This scheme is based on analyzing the mathematical characteristics of NURBS surfaces and a non-recursive evaluation of the basic functions of NURBS expression is performed.

Furthermore, this thesis proposes and designs a new hardware pipeline for the real-time rendering of complex models designed with NURBS surfaces. THe NURBS pipeline, called Rendering Pipeline for NURBS Surfaces (RPNS) consists of three modules: geometry, sampler and rasterizer. Firstly, KSQuad are processed in the geometry stage. Lately, in the sampler stage, each KSQuad is tessellated into a set of KSDices. RPNS performs an efficient adaptive discretization of KSQuads into KSDices, and it allows the fine tuning of the density of primitives needed to avoid cracks and holes in the final image. Finally, this KSDices are rasterized.

This thesis also implements a RPNS pipeline using current GPUs, achieving real-time rendering rates of complex parametric models. To provide a versatile example of our proposal, backpatch and backface culling strategies have been designed.

Although this thesis focuses on the design of a parametric surface oriented pipeline, it has also proved that our proposals are capable of achieving real-time rendering in the current triangle-oriented pipelines.

7.1. Future Work

As this dissertation proposes an effective solution for the rendering of NURBS surface in the GPU, several features are considered as extensions to the Rendering Pipeline for NURBS Surfaces. NURBS surfaces are defined as a tensor product of NURBS curves, thus their borderlines are defined by NURBS curves. However, as current GPUs are triangle-oriented pipelines and only triangles, lines or points can be projected from a 3D representation in the model world to a 2D representation in the screen world, a higher sampling is applied in the boundary edges in order to prevent discontinuities, especially if the continuity between patches in the boundary region is not G^1. Although RPNS introduces several proposals for reducing these cracks between surfaces, employing a piecewise representation such as friend surfaces detection, current GPUs are triangle oriented, thus each NURBS boundary edge defined by an NURBS curve is approximated as a set of linear segments. Consequently, triangle-oriented rendering produces numerous artifacts, such as cracks, holes and creases, since a considerable amount of triangles are needed to follow the curvature of the surface.

As a future extension to the NURBS pipeline, a rasterizer for the rendering of curved KSDice will be proposed. This rasterizer will be designed to project curves into the screen space and any patch artifacts due to different tessellation in the border will not be introduced; thus, the KSDice rasterizer will modify the features of the tessellation procedure of parametric surfaces. This rasterizer will provide more efficient rendering in which neither friend surface nor overtessellation techniques will be needed as the rasterizer will project the NURBS curve to the screen space and consequently a more exact representation of the parametric model will be achived. A complete parametric surface oriented pipeline will be designed including a KSDice rasterizer. In short, a NURBS model will be tessellated and evaluated in the GPU and finally small pieces of the NURBS model called KSDices will be render according to the NURBS representation.

Furthermore, current proposal of texture implementations on NURBS surfaces are based on a individual mapping of the texture. Usually, a parametric texture is associated with a surface, thus the coordinates of the parametric point evaluated in the NURBS surface can be used to obtain the corresponding texel. However and from a designer's point of view, it should be easier to assign one texture to the whole model. However, CAD or designer software as well as more common NURBS file formats have been designed to store one texture for each surface instead of one surface for the whole object.

In order to assign a texture to the whole model, textures should include a reference to those surfaces it is associated with, and a preprocessing step should be performed to identify which piece of the texture corresponds to each parametric patch. In this case each NURBS surface will access a piece of the texture and a more complex index than parametric coordinates should be needed to reference the corresponding texels.

Bibliography

[1] O. Abert, M. Geimer, and S. Müller. Direct and Fast Ray Tracing of NURBS Surfaces. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 161–168, Salt Lake City, UT, USA, September 18-20 2006. IEEE Computer Society. pages 3, 138, 156

[2] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering.* A. K. Peters, Ltd, third edition, 2008. pages 4, 35, 39, 46, 89, 151

[3] T. Akenine-Möller and J. Ström. Graphics for the masses: A hardware rasterization architecture for mobile phones. *ACM Transactions on Graphics*, 22:801–808, 2003. pages 107

[4] T. Akenine-Möller and J. Ström. Graphics Processing Units for Handhelds. In *Proceedings of the IEEE, special issue on Cutting-Edge Computing*, volume 96, pages 779–789, 2008. pages 109

[5] M. Amor, M. Bóo, J. Hirche, M. Doggett, and W. Strasser. A meshing scheme for efficient hardware implementation of butterfly subdivision surfaces using displacement mapping. *IEEE Computer Graphics and Applications*, 25(2):46–59, 2005. pages 53, 55

[6] A. Amresh and C. Fünfzig. Semi-uniform, 2-Different Tessellation of Triangular Parametric Surfaces. In *Proceedings of the ISVC'10: 6th International Conference on Advances in Visual Computing*, pages 54–63, Berlin, Heidelberg, 2010. Springer-Verlag. pages 6

[7] I. Antochi, B. Juurlink, and S. Vassiliadis. Scene management models and overlap tests for tile-based rendering. In *In Proc. EUROMICRO Symp. on*

Digital System Design (DSD 2004), pages 424–431, Washington, DC, USA, 2004. IEEE Computer Society. pages 112

[8] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Boosting mobile gpu performance with a decoupled access/execute fragment processor. *SIGARCH Computer Architecture News*, 40(3):84–93, 2012. pages 110

[9] D. Blythe. The Direct3D 10 System. *ACM Transations on Graphics*, 25(3):724–734, 2006. pages 4, 56, 74

[10] M. Bóo, M. Amor, R. Concheiro, and M. Doggett. Efficient adaptive and dynamic mesh refinement based on a non-recursive strategy. *The Computer Journal*, 2012. pages 26, 54, 55, 163

[11] T. Boubekeur and C. Schlick. Generic Mesh Refinement on GPU. In *Proceedings of the HWWS '05: ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 99–104, New York, NY, USA, 2005. ACM. pages 30, 32, 36, 108

[12] T. Boubekeur and C. Schlick. A Flexible Kernel for Adaptive Mesh Refinement on GPU. *Computer Graphics Forum*, 27(1):102–113, 2008. pages 53

[13] T. Capin, K. Pulli, and T. Akenine-Möller. The state of the art in mobile graphics research. *Computer Graphics and Applications, IEEE*, 28(4):74 –84, july-aug. 2008. pages 107, 112

[14] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association. pages 110

[15] I. Castaño. Next-Generation Hardware Rendering of Displaced Subdivision Surfaces. In *Exhibition Tech. Session at the SIGGRAPH'08*, New Orleans, LA, 2008. ACM Press, USA. pages 5

[16] E. Catmull and J. Clark. Recursively generated B-Spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, 1978. pages 1

[17] X. Chea, X. Liangb, and Q. Lib. G^1 continuity conditions of adjacent NURBS surfaces. *Computer Aided Geometric Design*, 22(4):285–298, 2005. pages 149

[18] K. Chung, Y. Chang-Hyo, K. Donghyun, and K. Lee-Sup. Shader-based tessellation to save memory bandwidth in a mobile multimedia processor. *Computer and Graphics*, 33(5):625–637, 2009. pages 108

[19] K. Chung, C.-H. Yu, D. Kim, and L.-S. Kim. Tessellation-enabled shader for a bandwidth-limited 3D graphics engine. In *Proceedings of CICC 2008: Custom Integrated Circuits Conference*, pages 367 –370, sept. 2008. pages 108

[20] R. Concheiro, M. Amor, and M. Bóo. Evaluation of Parametric Surfaces on the GPU. In *Proceedings of the ACACES'10: Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems.*, pages 13–16. HiPEAC, 2010. pages 25, 30

[21] R. Concheiro, M. Amor, and M. Bóo. Synthesis of Bézier Surfaces. In P. Richard, J. Braz, and A. Hilton, editors, *Proceedings of the GRAPP'10: International Conference on Computer Graphics Theory and Applications*, pages 110–115. INSTICC Press, 2010. pages 3, 25, 29, 30, 102, 108, 120

[22] R. Concheiro, M. Amor, M. Bóo, and R. Doallo. Explotación de la tarjeta gráfica para la síntesis de modelos basados en superficies Bézier. In *Proceeding of the XIX Jornadas de paralelismo*, pages 448–453, 2008. pages 25, 30

[23] R. Concheiro, M. Amor, M. Bóo, and R. Doallo. Síntesis de superficies paramétricas en tiempo real. In *Proceedings of the ANACAP 2008: Workshop de Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP 2008)*, 2008. pages 25, 30

[24] R. Concheiro, M. Amor, M. Bóo, and M. Doggett. Dynamic and Adaptive Tessellation of Bézier Surfaces. In P. Richard and J. Braz, editors, *Proceedings of the GRAPP'11: International Conference on Computer Graphics Theory and Applications*, pages 100–105. SciTePress, 2011. pages 26, 55

[25] R. Concheiro, M. Amor, M. Bóo, I. Iglesias, E. J. Padrón, and R. Doallo. Synthesis of Multiresolution Scenes with Global Illumination on a GPU. In P. Richard, M. Kraus, R. S. Laramee, and J. Braz, editors, *Proceedings of*

the GRAPP'12: International Conference on Computer Graphics Theory and Applications, pages 274–279. SciTePress, 2012. pages 26, 55, 56

[26] R. Concheiro, M. Amor, M. Bóo, E. J. Padrón, and M. Doggett. Interactive Rendering of NURBS Surfaces (under review). *High Performance Graphics (HPG)*, 2013. pages 28, 138

[27] R. Concheiro, M. Amor, M. Gil, and E. J. Padrón. Bézier tessellation in Handheld Devices (under review). *International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 2013. pages 27

[28] R. Concheiro, M. Amor, E. J. Padrón, and M. Bóo. Tessellation Techniques for Bézier Surfaces based on the Exploitation of the Spatial Coherence on GPU (under review). *Computer Aided Geometric Design*, 2012. pages 26, 86

[29] R. L. Cook, L. Carpenter, and E. Catmull. The Reyes Image Rendering Architecture. *SIGGRAPH Computer Graphics*, 21:95–102, 1987. pages 3

[30] C. DeCoro and N. Tatarchuk. Real-Time Mesh Simplification using the GPU. In *In the Proceedings of the I3D'07: Symposium on Interactive 3D Graphics and Games*, pages 161–166, Seattle, WA, 29 April–02 May 2007. ACM, USA. pages 4

[31] M. Doggett and J. Hirche. Adaptive view dependent tessellation of displacement maps. In *Proceedings of HWWS '00: ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 59–66, Interlaken, Switzerland, 21–22 August 2000. ACM Press. pages 53, 55

[32] D. Doo and M. Sabin. Behaviour of Recursive Division Surfaces Near Extraordinary Points. *Computer-Aided Design*, 10(6):356–360, 1978. pages 1

[33] C. Dyken, M. Reimers, and J. Seland. Semi-uniform Adaptive Patch Tessellation. *Computer Graphics Forum*, 28(8):2255–2263, 2009. pages 3, 29, 37

[34] C. Eisenacher and C. Loop. Data-Parallel Micropolygon Rasterization. In *Proceeding of the Eurographics 2010*, pages 53–56, Switzerland, 3–7 May 2010. The Eurographics Association. pages 3, 53, 81

[35] C. Eisenacher, Q. Meyer, and C. Loop. Real-Time View-Dependent Rendering of Parametric Surfaces. In *Proceedings of the I3D '09: Interactive 3D Graphics and Games*, pages 137–143, NY, USA, 27 February–01 March 2009. ACM Press. pages 3, 53

[36] F. J. Espino, M. Bóo, M. Amor, and J. D. Bruguera. Hardware Support for Adaptive Tessellation of Bézier Surfaces Based on Local Tests. *Journal of Systems Architecture*, 53(4):233–250, 2007. pages 72

[37] G. E. Farin. *NURBS - from projective geometry to practical use.* AK Peters, second edition, 1999. pages 11, 16, 137

[38] G. E. Farin. *Curves and surfaces for CAGD: a Practical Guide.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2002. pages 11, 16, 137

[39] K. Fatahalian, S. Boulos, J. Hegarty, K. Akeley, W. R. Mark, H. Moreton, and P. Hanrahan. Reducing Shading on GPUs Using Quad-Fragment Merging. *ACM Transations on Graphics*, 29:67:1–67:8, 2010. pages 3

[40] M. Fisher, K. Fatahalian, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan. DiagSplit: Parallel, Crack-free, Adaptive Tessellation for Micropolygon Rendering. *Proceedings of ACM SIGGRAPH Asia 2009*, 28(5), December 2009. pages XXXI, 7, 9

[41] M. Fisher, K. Fatahalian, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan. DiagSplit: Parallel, Crack-Free, Adaptive Tessellation for Micropolygon Rendering. *ACM Transations on Graphics*, 28:150:1–150:10, 2009. pages 3, 150

[42] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics: principles and practice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 1990. pages 4, 10, 126

[43] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick. The energy efficiency of IRAM architectures. *SIGARCH Computer Architecture News*, 25(2):327–337, 1997. pages 109

[44] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '89, pages 79–88, New York, NY, USA, 1989. ACM. pages 112

[45] Google. *Android documentation.* pages 108

[46] K. Grya. *Microsoft DirectX9 Programmable Graphics Pipeline*. Microsoft Press, 2003. pages 4

[47] M. Guthe, A. Balázs, and R. Klein. GPU-Based Trimming and Tessellation of NURBS and T-Spline Surfaces. *ACM Transations on Graphics*, 24(3):1016–1023, 2005. pages 3, 29, 30, 32, 36, 37, 44, 48, 108

[48] J. Hasselgren, J. Munkberg, and T. Akenine-Möller. Automatic Pre-Tessellation Culling. *ACM Transations on Graphics*, 28:19:1–19:10, 2009. pages 140, 141

[49] K. Jankauskas. Time-efficient NURBs curve evaluation algorithms. In *Proceedings of the IT2010: 16th International Conference on Information and Sofware*, pages 60–69, 2010. pages 156

[50] T. Kanai. Fragment-Based Evaluation of Non-Uniform B-spline Surfaces on GPUs. *Computer-Aidded Design and Applications*, 4(1-4):287–294, 2007. pages 3

[51] Khronos group. OpenGL specifications. Technical report. pages 114

[52] Khronos group. OpenGL ES. Technical report, 2010. pages 108, 115

[53] S.-H. Kim, S.-E. Yoon, S.-H. Chung, Y.-J. Kim, H.-Y. Kim, K. Chung, and L.-S. Kim. A mobile 3-d display processor with a bandwidth-saving subdivider. *IEEE Trans. VLSI Syst.*, 20(6):1082–1093, 2012. pages 108, 109

[54] A. Krishnamurthy, R. Khardekar, and S. McMains. Direct Evaluation of NURBS Curves and Surfaces on the GPU. In *Proceedings of SPM'07: The 2007 ACM Symposium on Solid and Physical Modeling*, pages 329–334, New York, NY, USA, 2007. ACM. pages 138, 141

[55] A. Krishnamurthy, R. Khardekar, and S. McMains. Optimized GPU Evaluation of Arbitrary Degree NURBS Curves and Surfaces. *Computer Aided Design*, 41(12):971–980, 2009. pages 3, 155, 156

[56] S. Kumar and D. Manocha. Hierarchical visibility culling for spline models. In *Proceedings of the conference on Graphics interface '96*, GI '96, pages 142–150, Toronto, Ont., Canada, Canada, 1996. Canadian Information Processing Society. pages 152

[57] S. Kumar, D. Manocha, B. Garrett, and M. Lin. Hierarchical back-face culling. In *In 7th Eurographics Workshop on Rendering*, pages 231–240, 1996. pages 152

[58] S. Kumar, D. Manocha, and A. Lastra. Interactive display of large-scale nurbs models. *IEEE Transactions on Visualization and Computer Graphics*, 2(4):323–336, 1996. pages 152, 153

[59] C. Loop. *Smooth Subdivision Surfaces Based on Triangles*. PhD thesis, University of Utah, Utah, 1987. pages 1

[60] C. Loop and S. Schaefer. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics*, 27(1):8:1–8:11, Mar. 2008. pages 2

[61] C. T. Loop, M. Nießner, and C. Eisenacher. Effective Back-Patch Culling for Hardware Tessellation. In *Proceeding of the VMV 2011: Vision, Modeling, and Visualization Workshop*, pages 263–268, 4-6 October 2011. pages 153

[62] H. Lorenz and J. Döllner. Dynamic Mesh Refinement on GPU using Geometry Shaders. In *Proceedings of the WSCG '08: 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2008*, pages 97–104, Plzen–Bory, Czech Republic, 4–7 February 2008. ACM Press, USA. pages 4, 53, 81

[63] F. Luna. *Introduction to 3D Game Programming with DirectX 11*. Mercury Learning Series. International Pub Marketing, 2012. pages 5

[64] Mali. Mali GPU OpenGL ES. Application Development Guide. Technical report, 2009. pages 124

[65] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical Ray Tracing of Trimmed NURBS Surfaces. *Journal of Graphics Tools*, 5:27 – 52, 2000. pages 3

[66] Microsoft. *DirectX SDK Documentation*, 2007. pages 36, 42, 44, 75, 114

[67] J. Munkberg, J. Hasselgren, and T. Akenine-Möller. Non-uniform Fractional Tessellation. In *Proceedings of the GH '08: 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 41–45, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. pages 6

[68] J. Munkberg, J. Hasselgren, R. Toth, and T. Akenine-Möller. Efficient bounding of displaced bezier patches. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 153–162, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association. pages 153

[69] A. Munshi, D. Ginsburg, and D. Shreiner. *OpenGL(R) ES 2.0 Programming Guide*. Addison-Wesley Professional, 1 edition, 2008. pages 115

[70] H. Nguyen. *GPU GEMS 3*, chapter third. Addison-Wesley Professional, first edition, 2007. pages 44

[71] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray Tracing Trimmed Rational Surface Patches. *ACM SIGGRAPH 1990*, 24:337 – 345, 1990. pages 3

[72] Nvidia. *Technical Brief. Microsoft DirectX 10: The Next-Generation Graphics API*, 2006. pages 4

[73] NVIDIA. Technical Brief. Bringing High-End Graphics to Handheld Devices. Technical report, 2011. pages 111, 124

[74] NVIDIA. Technical Brief. The Benefits of Quad Core CPUs in Mobile Devices. Technical report, 2011. pages 111

[75] R. Pajarola and E. Gobbetti. Survey of Semi-regular Multiresolution Models for Interactive Terrain Rendering. *Visual Computing*, 23(8):583–605, July 2007. pages 98

[76] A. Patney, M. S. Ebeida, and J. D. Owens. Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *Proceedings of the High Performance*

Graphics, pages 99–108, New York, NY, USA, 01–03 August 2009. ACM. pages 53

[77] A. Patney and J. D. Owens. Real-Time Reyes-Style Adaptive Surface Subdivision. *ACM Transations on Graphics*, 27:143:1–143:8, 2008. pages 3

[78] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *Proceedings of the SIGGRAPH'00: 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. pages 140

[79] L. Piegl and W. Tiller. *The NURBS Book.* Springer, 1997. pages 11, 16, 29, 137, 138, 156

[80] H. Prautzsch and T. Gallagher. Is there a Geometric Variation Diminishing property for B-Spline or Bézier Surfaces? *Computer Aided Geometric Design*, 9(2):119–124, 1992. pages 16, 23

[81] K. Pulli, J. Vaarala, V. Miettinen, T. Aarnio, and K. Roimela. *Mobile 3D Graphics: with OpenGL ES and M3G.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. pages 115

[82] D. F. Rogers. *An Introduction to NURBS with Historical Perspective.* Morgan Kaufmann, 2001. pages 11, 16, 29, 137

[83] A. L. Sarmiento, M. Amor, E. J. Padrón, C. V. Regueiro, R. Concheiro, and P. Quintía. Evaluating Performance of Android Systems as a Platform for Augmented Reality Applications (in press). *International Journal of Advances in Software*, 2012. pages 27

[84] M. Schwarz and M. Stamminger. Fast GPU-based Adaptive Tessellation with CUDA. *Computer Graphics Forum*, 28(2):365–374, 2009. pages 3, 81

[85] M. Schwarz and M. Stamminger. Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum (Proceedings of Eurographics 2009)*, 28(2):365–374, 2009. pages 53

[86] T. W. Sederberg and R. J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric Design*, 5(2):161–171, 1988. pages 153

[87] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 3.0 - August 11, 2008)*, 2008. pages 114

[88] P. Shirley. *Fundamentals of Computer Graphics.* Addison-Wesley, 2003. pages 4, 38, 77, 123

[89] L. A. Shirman and S. S. Abi-Ezzi. The cone of normals technique for fast processing of curved patches. *Computer Graphics Forum*, 12(3):261–272, 1993. pages 153

[90] T. Ni and I. Castaño. Efficient substitutes for Subdivision Surfaces. In *SIGGRAPH Course Notes*, New Orleans, LA, August 2009. ACM Press, USA. pages 5, 74

[91] P. Walsh. *Advanced 3D Game Programming with DirectX 10.0.* Wordware, 2008. pages 4

[92] D. Wexler, L. Gritz, E. Enderton, and J. Rice. GPU-Accelerated High-Quality Hidden Surface Removal. In *Proceedings of the HWWS '05: ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–14, New York, NY, USA, 2005. ACM. pages 3

[93] Y. I. Yeo, L. Bin, and J. Peters. Efficient Pixel-Accurate Rendering of Curved Surfaces. In *Proceedings of the i3D'12: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 165–174, New York, NY, USA, 2012. ACM. pages 2, 3, 6

[94] M. Zechner and R. Green. *Beginning Android 4 games development.* Apress, Berkeley, CA, USA, 2011. pages 115

[95] K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, and B. Guo. RenderAnts: Interactive Reyes Rendering on GPUs. *ACM Transations on Graphics*, 28:155:1–155:11, 2009. pages 3

[96] R. Zioma. Unity: iOS and Android: cross platform challenges and solutions. In *Exhibition Tech. Session at the SIGGRAPH'12*, Los Angeles, 2012. ACM Press, USA. pages 112