Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Преодоление ограничений графического конвейера GPU с использованием Mesh Shaders / Overcoming restrictions of GPU graphical pipeline with Mesh Shaders**

Работу выполнил / Thesis is executed by	**Алькабакиби Мохамад Зиад / Mohamad Ziad Alkabakibi**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Белявцев Иван / Ivan Belyavtsev**	подпись / signature

Иннополис, Innopolis, 2022

Contents

List of Tables

List of Figures

Abstract

GPU performance optimization has always been a concern for hardware and software designers because it is not only about drawing beautiful detailed models to make a game or other software successful but it is also about producing such models within the performance capabilities of users' hardware. In addition, surfaces which are made up of small triangles cause GPUs to do redundant shading and waste performance. However, Mesh shaders came out in 2018 and their features were notable. In this thesis, we study these features, correlate them with the previous 2 problems, compare performance with older features and try to understand the results.

Chapter 1

Introduction

1.1 Graphics Processing Units & Shaders

A graphics processing unit (GPU) is a specialized circuit for rapid manipulation and alteration of memory to create images so that users can interact with different types of electronics. With the development of programming, General-purpose computing on graphics processing units became possible [1]. In addition, GPUs have been expeditiously developing in terms of performance. However, this development comes with the cost of additional energy consumption [2]. Besides, graphics pipeline model has been developing slowly for the common graphics APIs (OpenGL, Direct3D, Vulkan and Metal) since they appeared. On the other hand, new software-side optimized innovations have been introduced especially in terms of rendering such as Mesh Shaders.

1.2 Direct3D 12 Graphics Pipeline

GPU uses a conceptional model called **Graphics Pipeline** to transform a 3D scene to a 2D scene that can be drawn on the screen for users. There

are different graphics pipelines depending on the software and hardware used. In particular, Direct3D since the Direct3D11 API uses the following graphics pipeline:[3]

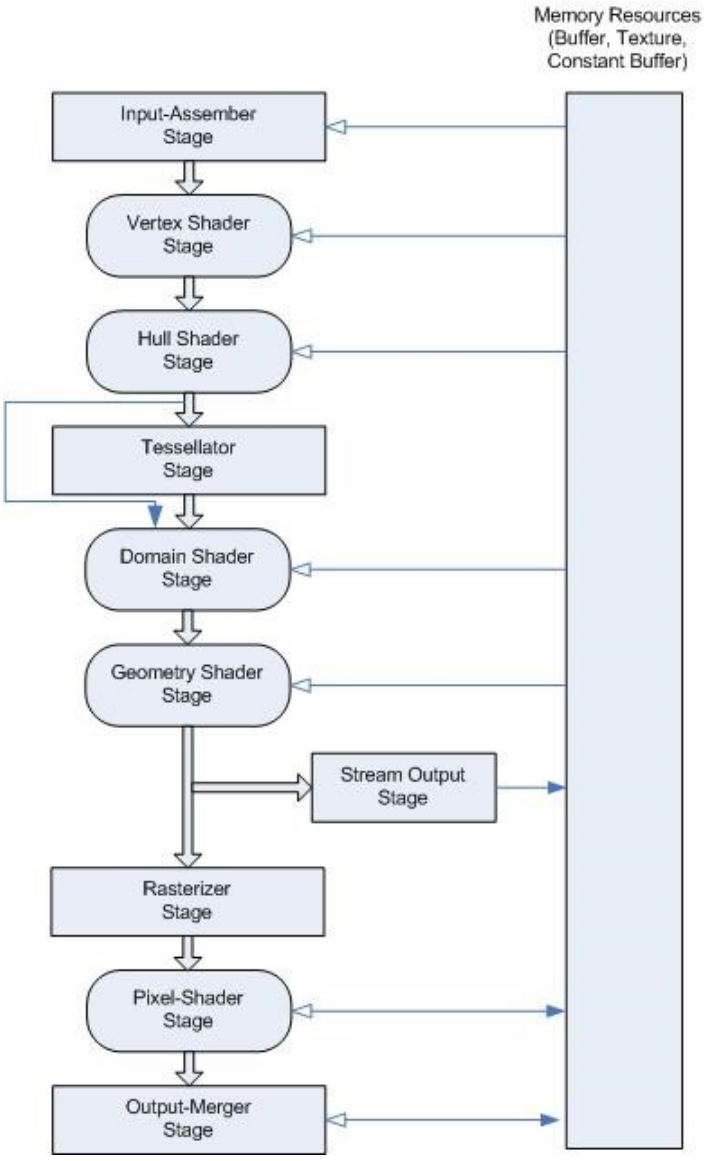

Figure 1.1: Direct3D11 (and 12) Graphics Pipeline

Some stages of the pipeline are **programmable** using the shader code while other stages are **fixed** and should be configured using the Direct3D API. Note that the resource says this is D3D11 graphics pipeline, but according to the changes between D3D11 and D3D12, so this still holds true. [4]

1. Input assembler:

   Reads user-defined data, such as points from buffers (vertex and index buffers for example), and assembles them in one of the user-defined ways (line lists, triangle stripes or any format available for formatting vertices assembly). Finally, it outputs the results to the Vertex Shader.

2. Vertex Shader:

   Retrieves vertex data from the previous stage and applies many operations per vertex such as transformations. This step is mandatory even if no operations are needed to be done where data at least should be passed from the first to the third stage through this second one.

3. Tessellation Stages (Hull-Shader, Tessellator, Domain-Shader stages): Converts low-detail subdivision surfaces into higher-detail primitives on the GPU. Tessellation tiles (or breaks up) high-order surfaces into suitable structures for rendering.

4. Geometry Shader:

   Takes sets of vertices as a full primitive each. For example, it can take each 2 vertices for a full primitive to draw a line and it connects vertices for each primitive depending on the adjacency and type of primitive used (line, triangle or any other available primitive type)

5. Stream Output:

   Continuously streams vertex data received from Geometry Shader (or from the Vertex Shader if the Geometry Shader is inactive) to one or more buffers in the GPU memory. As a result, written data can be read in a future rendering pass or it can be read by CPU.

6. Rasterizer:

   Converts the data from the vector and other stored formats to a raster image. Therefore, primitives are converted into pixels, clipping to the view frustum and division by Z axis are applied to prepare data for Pixel Shader to be in a suitable format.

7. Pixel Shader Invoked once per pixel to determine the final color value of each pixel by applying shading techniques in addition to the depth test. A NULL shader can be used also to avoid running a shader.

8. Output-Merger:

   Writes the colors to pixels after determining if the pixel should be drawn or not (using Depth-Stencil Testing). If blending is enabled, it applies blending equation to the Pixel Shader produced values using values set in the Pipeline State Object.

1.3 Mesh Shaders

Turing Architecture [5] by Nvidia providing a suitable architecture for extending parallelism which optimizes performance to draw objects using Mesh Shaders. In particular, each Mesh is divided into Meshlets [figure 2]. Each Meshlet is optimized for the vertex re-use within and each thread can be programmed to draw or do another task. Therefore, we can render more geometry while loading less data overall.

Figure 1.2: Mesh decomposed into Meshlets

Mesh Shaders are used in Task/Mesh pipeline which differs from the traditional graphics pipelines and mainly has the following 2 new stages:

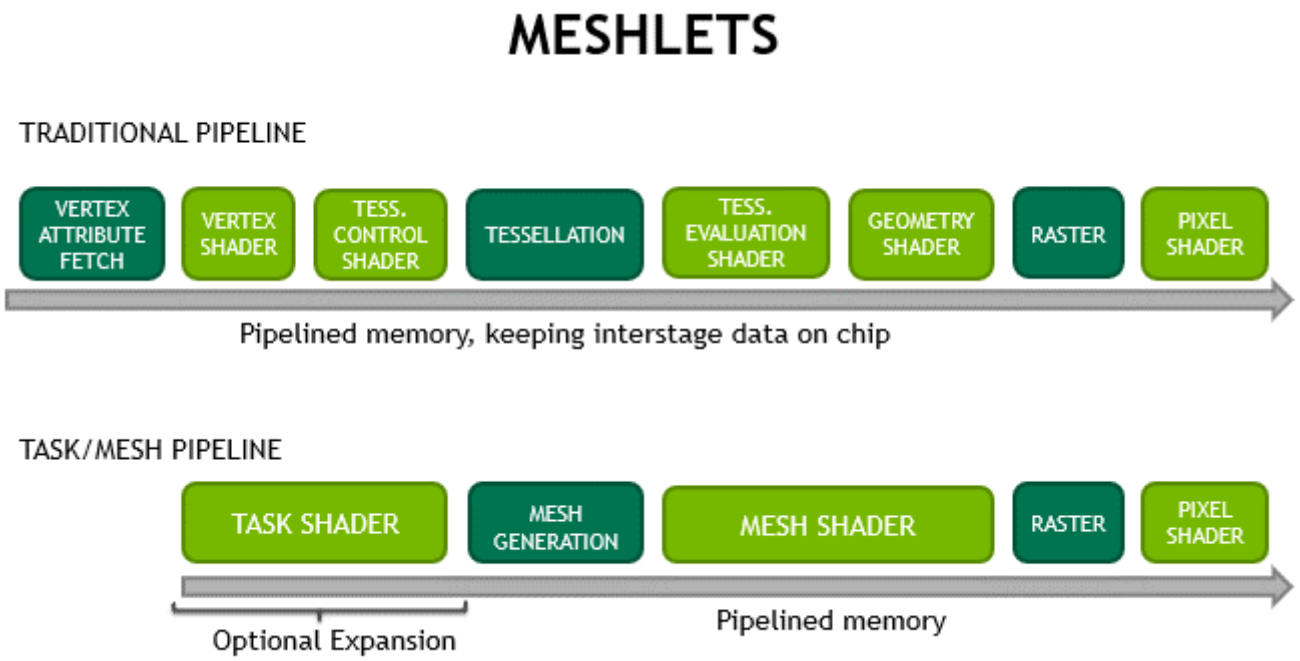

Figure 1.3: Traditional vs Task/Mesh pipeline

1. Task Shader:

   Optional stage which mainly works in workgroups for producing and executing Mesh Shaders' workgroups. For example, it runs mesh shader only after determining that a Meshlet is visible. Additionally, each Task Shader can share data with its Mesh Shaders children allowing them to access the shared memory for its group [6].

2. Mesh Shader: Also uses a collaborative threads model and allows the memory to be kept on the GPU after reading once only. Each group of threads produces primitives for the rasterizer without affecting the interfacing with the pixel and fragment shaders.

This new pipeline provides many advantages over the traditional:

- Flexibility: by being able to create the graphics work and define the mesh topology. On the other hand, insufficient threading was a problem in geometry shaders and only limited patterns could be used in tessellation shaders.

- Higher scalability: by providing more general purpose programming on the GPU and reducing fixed- function usage such as allowing applications to improve performance by adding more cores for example.

- Bandwidth-reduction: by allowing vertex reuse without the need to reread vertices in addition to providing developer with ability to use their own ways to store and compress data.

1.4 Traditional and Mesh pipelines implementation differences

In the traditional pipeline, vertex shaders read vertices data from the vertex buffer and read indices from the index buffer if needed. ID3D12GraphicsCommandList::IASetVertexBuffers function is used to inform the GPU how to view the data in the vertex buffer [7]. In addition, ID3D12GraphicsCommandList::IASetIndexBuffer is used to inform the GPU about the view of the index buffer [8]. Mesh shaders on the other hand, do not have input data and resources are the only way to upload data to them. Moreover, According to Direct3D Documentation [9], GPU cannot know what or how to read what is stored inside the resources by themselves. As a result, Descriptors, Descriptor heaps and root signature have to be used for the

following reasons:

- Root signature [10] is similar to a function signature because it defines root parameters which are the types of data the shaders should expect without defining the actual data or memory. On the other hand, root arguments are the actual values of root parameters passed at runtime and changing them changes the values that shaders read. Also root signature links the command list to resources needed by the shader. Furthermore, there are 3 types of parameters that a root signature might have:

    1. A Root Constant: an 32 bit value that appears in the shader as a constant buffer
    2. A Root Descriptor: these are inlined in the root signature and it might be Constant Buffer View, Raw Unordered Access View, Structured Unordered Access View or Shader Resource View. However, more complicated types cannot be used as a root descriptor such as 2D textures Structured Resource Views. Furthermore, there cannot be a check for out of bounds cases because the limit size is not included. On the other hand, descriptors that are contained in a descriptor heap have such feature.
    3. A Descriptor Table: each entry has:
        (a) A Descriptor
        (b) The bind name of the HLSL shader
        (c) Visibility flag: determines where this entry's descriptor is visible and for better performance it is advisable to make visibility where it is needed.

- Descriptors/Views: in order to inform the GPU what the resource contains and how to use it. For example, a descriptor of a resource containing a **float4** vector, informs the GPU that it has **x, y, z, w** fields and all of them have the type **float** so that developer can access the values and use them in the shader code.

- Descriptor Heaps [11]: Where descriptors are kept so that GPU can use them from its own memory. Furthermore, there are some restrictions about descriptor heaps: Any Constant Buffer Views, Unordered Access Views and Structured Resource Views are considered common resource views. On the other hand, Samplers entries cannot share the same descriptor heap with the common resource views. In addition, descriptor heaps can only be modified from the CPU.

  The following steps define how to upload data from CPU to the GPU memory: [12]

  1. Use ID3D12Device::CreateCommittedResource to create a normal heap, reserve the memory that will contain the data on the GPU and the state should be D3D12_RESOURCE_STATE_COPY_DEST
  2. Create an upload heap which is reserved on RAM and will contain the data we want to upload and it should have the state D3D12_RESOURCE_STATE_GENERIC_READ.
  3. Create a command list of type Direct which accepts all the types of commands (D3D12_COMMAND_LIST_TYPE_DIRECT) or just D3D12_COMMAND_LIST_TYPE_COPY. because only copy is needed in this case.

4. Create a command queue so that command list can be inserted into it and GPU will execute it.
5. Map the System memory to the upload buffer and copy data to it.
6. Ask the command list to add a command for copying data from the upload buffer to the default heap so that the GPU will execute it.

Chapter 2

Literature Review

2.1 Previous Works

Due to the fact that Turing Architecture was released at the end of 2018, researchers have been trying to optimize rendering operations performance using software and algorithms with the old hardware. University of Washington in 1993 made a research [13] about Mesh Optimization to solve the following problem: Given a set of data points scattered in 3D and a mesh M (which is already divided into triangles), find a mesh M' which has the same topological type, fits the data well and with a small number of vertices to decrease the processing needed to render M.

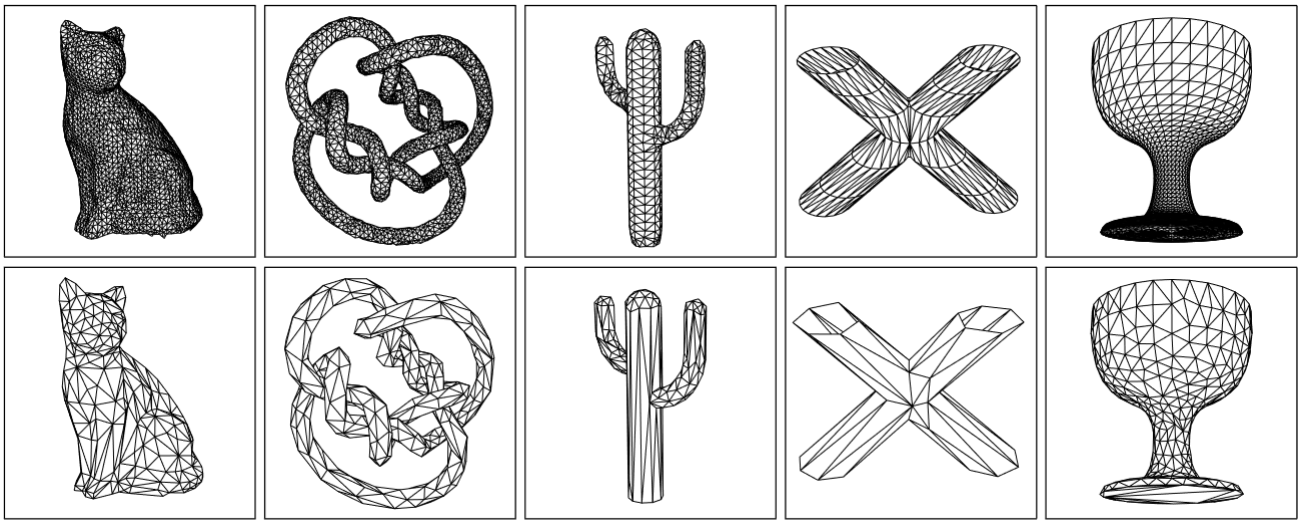

Figure 2.1: Initial Meshes (top) vs Optimized Meshes (bottom)

The algorithm aims to achieve 2 goals:

1. Minimize the number of Vertices
2. Minimize Energy Function EF where EF(M,M') is the deviation of M' from M

In terms of Mesh Simplification, the algorithm achieves interesting results:

1. Edges and Vertices of M' are located near sharp features of M
2. Low curvature causes long edges in its direction
3. Gaussian curvature has a positive relationship with the density of the vertices in each region

But this approach has some disadvantages:

- Despite the fact that the topological type is the same, M' is rather dense and lacking the sharp features of the original Mesh M. Therefore, M' is far away from the original data.

The previous approach can be used for the reconstruction of a surface given its unorganized points or simplifying a dense mesh by reducing its vertices.

To conclude, the previous study provides a solution by minimizing the processed data but with the cost of losing resolution and some properties from the original mesh since it forces mesh simplification as a second stage of it. On the other hand, mesh shaders do not change meshes but only provide better collaboration between threads to render the original mesh without modifications. However, these 2 methods can even be applied together if we have a very complicated mesh, picking one of the solutions is not efficient enough and we have a space to allow for some data loss.

In another research[2] by Graz University of Technology and Max Planck Institute for informatics which used Direct3D9 API for delivering real world scenes real-time rendering within a bounded memory where the pipeline architecture was made to make use of GPU parallelization. Types of sorting in the pipeline differ by the point when the sorting algorithm is applied after taking the data during from object-space to screen-space redistribution. As a result it has 4 types: sort-first, sort-last, sort-middle and sort-everywhere with the last method being the latest. The overall approach in the paper can be identified as sort-everywhere despite the fact that it uses this type only between the cores inside each of the multiprocessors while it uses sort-middle between them. One of the benefits of sort-everywhere is exposing many points in the pipeline and give more opportunities for optimization. However, a full sort-everywhere affects the memory bandwidth negatively. The optimal design point was found

to be between geometry processing and rasterization using a tiled rendering approach. To clarify, when a local redistribution processing from vertex to primitive occurs. The steps are as follows:

1. In vertex processing batches of the indices are loaded and sorted (to exclude duplicates) and the outputs are passed to primitive processing using shared memory.

2. The viewport is divided into sub-areas with a configurable size and mapped to a rasterizer (one rasterizer can be mapped to many sub-areas but not the opposite) which reads the triangles it got from the previous stage in the global memory but only the ones included totally or partially in its sub-area(s)

3. Each rasterizer reads the triangles which are relevant to its mapped sub-areas from the global memory. As a result, each rasterizer's thread block has its own exclusive access to its framebuffer regions and related data and using shared memory for communication purposes only.

4. Inside each rasterizer R_i the *Tiles* covered by its sub-area are determined

5. If $t_i \in R_i$ where t_i is the tile with index i, its pixel coverage masks are compacted further.

6. Rasterizer threads are reassigned for pixel shading either to individual pixels or to pixel quads

7. Only with local synchronization, pixel shading threads perform in-order Framebuffer blending.

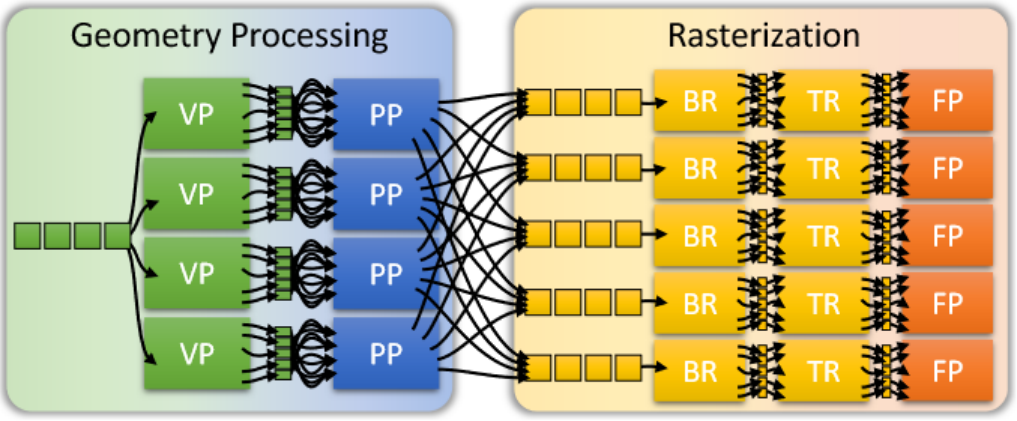

Figure 2.2: pipeline overview for the second research. Geometry processing uses on-chip memory and the register file to locally redistribute data between vertex processing (VP) and primitive processing (PP). Rasterization uses on-chip memory to locally redistribute data between bin rasterizer (BR), tile rasterizer (TR), and fragment processing (FP).

Implementation strategy was inspired by the mega-kernel approach used in Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU by Graz University of Technology in Austria [14]. The kernel function contains the rendering pipeline scheduler and mega-thread scheduler. Rasterizer and geometry processor thread blocks are assigned on demand by the scheduler considering the following:

1. The block should execute rasterization only if there is enough work for it.

2. Otherwise the block runs geometry processing If all rasterizer queues can take enough additional triangles and there are enough available input primitives for geometry processing .

3. The block process any rasterization input data remaining in the queue If nothing among the previous conditions suceeeds.

4. If nothing is available to do and other blocks are doing nothing in regards of geometry, then the block will terminate.

The buffer storing the triangles is external and referenced by the elements in the rasterizer input queue in order to reduces the size of the output when doing geometry processing and helps to get rid of the duplicates in case a triangle covers many memory bins. In addition, input queues for the rasterizer use the same ring buffer approach.

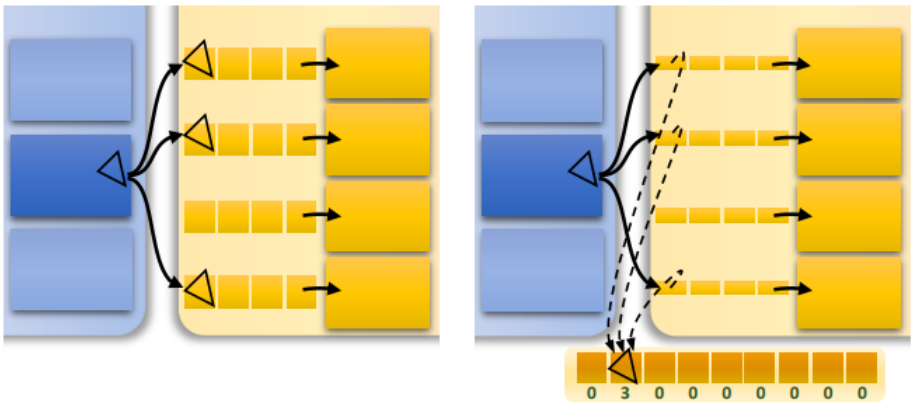

Figure 2.3: a. Many rasterizer queues may contain the same triangle.
b. Reducing memory requirements by storing indices in the rasterizer queues and using a buffer specially for triangles.

1. During geometry processing, the vertex shader is run and duplicate vertices are removed in order to avoid the costly call for the shader which runs for every vertex. In addition, supporting vertex reuse results in threads collaboration over the triangles' attributes. By setting a limit and upper bound for the number of loaded indices from the previous stage, a sufficient number of threads is guaranteed. Furthermore, using register shuffles, the other threads can access the vertex shader results.

2. During Clipping and culling, many triangles are discarded because they are not in the view frustum. This is possible since triangles data at this stage is ready and available. Furthermore, this is done on the GPU cores.

3. During Triangle setup, needed data is all calculated and triangles' locations are pushed to the input queues.

4. Primitive order will already be satisfied since rasterizers consume their input queues in order after they reorder the data in them such that it will be the required output. However, a triangle might not be ready yet during the sorting step so a bitmask for the ring buffer is used as an indicator.

5. During raterization, triangle data is read from the queues, processed to get the resulting fragments after getting each 8 x 8 tile by dividing the 64 x 64 bins and assigning all threads to the same stage to collaborate which enables the possibility of using prefix sums. Furthermore, ensuring memory consumption to be bounded is done using the strategy of encoding only and expanding in the next stage.

This research has shown noticeable performance improvement as shown in the figure:

		bud	fry	san	sib	tr12	tw12	sh23	sg13
GTX 1080	OpenGL	1.1	0.2	7.5	0.1	2.9	2.3	0.8	0.4
	OpenGL$_{fi}$	1.2	0.3	7.5	0.1	4.0	3.9	2.1	0.9
	CUDARaster								
	cuRE	7.8	2.8	35.8	1.6	28.4	39.9	16.8	6.4
	cuRE$_{w/q}$	5.4	2.4	29.2	1.4	24.0	21.5	14.8	6.1
	cuRE$_{w/p}$	8.3	2.8	35.5	1.4	25.3	37.9	14.2	5.8
	cuRE$_{w/o}$	5.5	2.3	28.0	1.4	20.9	19.4	12.1	5.4
	Piko	8.4	3.6	44.0	2.9	37.1	25.1	12.5	7.7
	FreePipe	0.8	72.3	292.7	68.1	261.3	66.2	141.4	145.0
GTX 780 Ti	OpenGL	1.8	0.4	12.3	0.2	5.1	3.4	1.4	0.8
	OpenGL$_{fi}$								
	CUDARaster	4.2	2.1		1.5	20.1	14.3	7.1	4.6
	cuRE	23.1	8.0	143.3	4.8	88.9	95.2	37.4	18.3
	cuRE$_{w/q}$	17.1	7.4	105.7	4.1	70.7	56.9	33.8	16.7
	cuRE$_{w/p}$	23.6	7.8	143.7	4.3	82.8	93.5	31.1	16.7
	cuRE$_{w/o}$	16.9	7.0	95.6	3.6	63.5	50.4	28.3	16.7
	Piko	19.4	8.5		7.5	89.9	62.3	28.9	19.8
	FreePipe	2.1	156.3		149.1	903.6	182.5	463.8	492.4

Figure 2.4: Frame times for the scenes which are selected in the study measured with ms and with resolution 1024×768 for Buddha (bud), Fairy Forest (fry), San-Miguel (san), Sibenik (sib), and four game scenes

However, this research is hard to apply because it is not used with the latest graphics APIs such as Direct3D 12 and Vulkan. In addition, the study should give better results with these new APIs because they give developers more control over the graphics pipeline which allows for more optimizations. Furthermore, it will be hard to apply this study to be used by all graphics developers because the industry these days use the latest APIs in addition to the required knowledge to be able to use this new pipeline for different purposes in case the API developers do not integrate it or change their documentation for the changes which is not the case currently. As a result, developers cannot get the benefits of this study in the same way they can use the available APIs.

Chapter 3

Methodology

To understand the relation between performance, the size of the triangles, and how the 2 experiments explain that, we need to introduce some definitions:

- N is the number of triangles drawn
- $P = \{p_1, p_2, ..., p_i, ..., p_n\}$ where p_i is the number of pixels in triangle with index i
- $P_s = \Sigma_{i=N}^{i=1} p_i$ is the total number of pixels drawn on the screen which belong to the triangles
- C is a constant number of our choice which value was picked equal to 4800 so that it can be easily divided by the powers of 2 without a remainder and suitable for performance as explained in the Implementation section.
- *Area* is a function $\mathbb{R}^3 \to \mathbb{R}$ calculating the area of triangle t using Shoelace Formula given its 3 points' coordinates (Note that coordinates are in the range $[-1, 1]$ because of the way Direct3D works).
- G is a function $\mathbb{R} \to \mathbb{R}$ that applies **LinearInterpolation** to map the

results of $Area(t)$ from $[Area_min, Area_max]$ to $[0, 255]$ and used to visually make the triangle more greenish the bigger its area is. Note that $Area_min, Area_max$ differ between the 2 experiments. Thus, the function only tries to show visually the effect of tuning parameters in each experiment's domain of work and does not have the same formula for each experiment.4

The 2 Experiments that have been done with the previously introduced parameters and definitions:

1. $\forall\, p \in P, p = mC_2$ where $m \in \{1, 2, 4, 8, 16, 32, 64\}$ and $n = C$.
   The number of triangles was always the same but the size of each triangle was increasing as we are changing m. Since triangles were very big, it was too hard to measure the count of pixels. As a result, $Area$ and G functions were only used to analyze the results without Ps.

2. $P_s = C$ (for all values of m), $\quad n = C/m, \;\; \forall\, p \in P$ we have:
   $p = m$ and $m \in \{1, 2, 4, 8, 16, 32, 64, 80, 100, 120, 160, 200\}$ so that we have a constant number of pixels drawn on the screen and filling the equal-sized triangles despite changing triangles' count and the size for each.

Chapter 4

Implementation

4.1 Tools and Programming Languages Used

For the CPU side part of the experiment, C++ programming language and Direct3D 12 were used. In addition, for implementing Vertex, Pixel and Mesh shaders, HLSL was used. Both experiments' code have been built on Microsoft Direct3D samples to ensure optimization. In addition, depth and stencil buffers were turned off to make sure all non-culled triangles consume the GPU performance despite overlapping for more precise results.

The code is available as a GitHub Repository with instructions on building them. Also, Nvidia RTX graphics card is required to be able to run the Mesh Shader parts of the experiments due to these graphics card exclusive new technology compared to the older ones.

Hardware used to do the experiments:

- Asus ROG Strix SCAR 17 G733QM-HG028;
- Nvidia RTX 3060 8 GB mobile graphics card (115V with the 130V dynamic boost disabled);

- RAM: 8 GB DDR4-3200 SO-DIMM x 2;
- 512 GB M.2 NVMe PCIe 3.0 SSD;

Frame view software from Nvidia was used to benchmark the resulting exe in release mode. Microsoft Hello Triangle example was used as a start because its the code and classes are organized and convenient for the reader to understand. However, Depth buffer was switched off to force the GPU to draw all the triangles for the correctness of the experiments' results. In addition, Mesh pipeline parts which do not exist in the traditional pipeline were added also.

4.2 Generation Algorithm

The following algorithm was used to generate N triangles in every run:

```
class Point:
    def __init__(self, X, Y, Z):
        self.x = X
        self.y = Y
        self.z = Z

    def on_screen(self):
        return -1 <= self.x <= 1 and -1 <= self.y <= 1

    def move(self, x_diff, y_diff):
        return Point(self.x + x_diff, self.y + y_diff, self.z)


class Triangle:
    def __init__(self, p1: Point, p2: Point, p3: Point):
```

```
        self.p1 = p1
        self.p2 = p2
        self.p3 = p3

    def move_triangle(self, x_diff, y_diff):
        return Triangle(self.p1.move(x_diff, y_diff),
                        self.p2.move(x_diff, y_diff),
                        self.p3.move(x_diff, y_diff))


def generate_triangles():
    triangles_list = []
    for i in range(triangles_count):
        t = initial_triangle
        while True:
            t = initial_triangle.move_triangle(random.uniform(-1.0, 1.0),
                random.uniform(-1.0, 1.0))
            if t.on_screen():
                break

        triangles_list.append(t)

    return triangles_list
```

Note: this generation only runs once at the start of the program

N was chosen to be 4800 because it can be divided by all the powers of 2 up to $2^6 = 64$. In addition, all triangles were chosen to have the same color and size but the color was determined depending on the size picked for the initial triangle such that triangles are greener the smaller the area using

affine transformation. Size of triangles in the first experiment was being scaled such that the number of triangles multiplied by the pixels filling a triangle is always N. However, in the second experiment scaling was not done. Finally, the final version of the generated triangles is stored in std::vector without any modification after.

4.3 Shaders Used

Meanwhile, Shaders in the traditional pipeline were only simple Vertex and Pixel shader taking the default parameters when drawing a triangle. Where vertex shader was just passing coordinates and color to the pixel shader and pixel shader only returned the passed color. Moreover, we did not use an index buffer in this pipeline because all triangles have unique points and index buffer will not add any optimization or save more memory. On the other hand, Mesh pipeline used the following Mesh shader:

```
struct MSvert
{
    float4 pos : SV_POSITION;
    float4 color : COLOR;
};

struct SubsetIndicator
{
    uint start;
};

StructuredBuffer<float4> coords : register(t0);
```

```
ConstantBuffer<SubsetIndicator> subsetStart : register(b0);

[outputtopology("triangle")]
[numthreads(75, 1, 1)]
void main(in uint3 groupID : SV_GroupID,
          in uint3 threadInGroup : SV_GroupThreadID,
          out vertices MSvert verts[225],
          out indices uint3 idx[75]) // Three indices per primitive (triangle)
{
    const uint numVertices = 225;
    const uint numPrimitives = 75; // 225 / 3
    SetMeshOutputCounts(numVertices, numPrimitives);

    uint tid = threadInGroup.x;

    // each thread 1 triangle
    for(int i = 0; i < 3; i++)
    {
        int j = tid * 3 + i;
        int k = subsetStart.start + j;
        verts[j].pos = float4(coords[k].x, coords[k].y, coords[k].z, 1.f);
        verts[j].color = float4(coords[k].w, 1.f, coords[k].w, 1.f);
    }

    idx[tid] = uint3(tid * 3, tid * 3 + 1, tid * 3 + 2);
}
```

75 was chosen as the number of threads because all triangles count in the experiments N/x were always divisible by 75 (etc $4800/64 = 75$). Moreover, DispatchMesh(75, 0, 0) was called until all the triangles in the experiment are covered by the drawing commands. In order to know where does the subset

starts in each DispatchMesh(75, 0, 0) call, a constant root integer root value *subsetStart* was used and was wrapped in a struct SubsetIndicator because HLSL does not allow uint to be the direct type of constant buffers. In addition, a structured buffer was used to access the data in the default heap in the GPU where the triangles data resides because constant buffers can contain only up to 64 Kilobytes that can contain at most 4096 vectors (points) which is much less than needed for storing 4800 triangles where 3 vectors are needed for each. The 4^{th} component of float4 was used to upload the color and the first 3 were used to indicate the vertex position because uploading 2 heaps will consume more memory without the need for it. Furthermore, Pixel shader used was the same one in the traditional pipeline.

4.4 Experiments and Results

For tables, the letter **A** will be used to represent the case of traditional pipeline and the letter **B** will be used for the case of mesh pipeline.

4.4.1 First Experiment

The following values were used and average FPS for each value:

Scale factor	Number of triangles	Pixels per triangle	A avg FPS	B avg FPS
0.01	5000	1	67.834	944.858
0.015	5000	2	56.165	906.579
0.02	5000	4	54.897	897.422
0.029	5000	8	50.329	863.404
0.043	5000	16	46.044	852.397
0.062	5000	32	26.492	703.934

Table I: Values used and results in the first experiment

- Using traditional pipeline, we got the following chart:

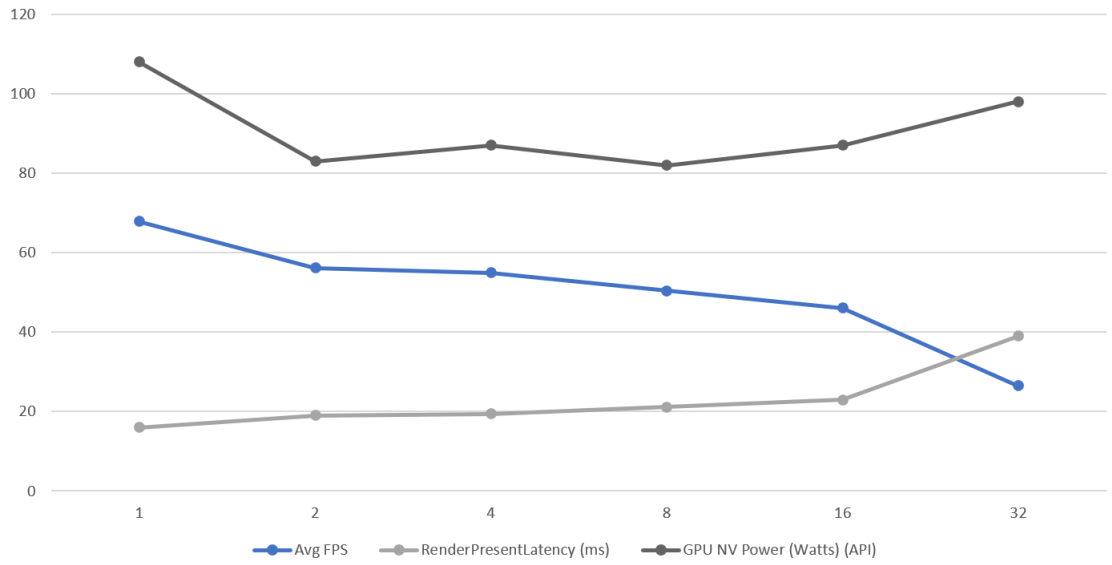

Figure 4.1: Render Present Latency, GPU power consumption and FPS when drawing 5000 triangles with scale x each using traditional pipeline for the first experiment case A **(Note that dynamic boost is not enabled)**.

- On the other hand, using a mesh pipeline:

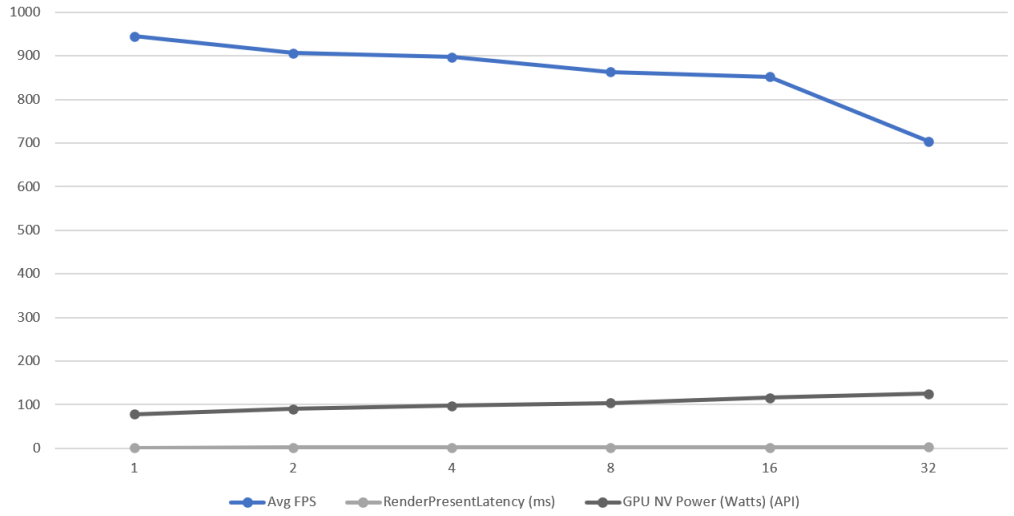

Figure 4.2: Render Present Latency, GPU power consumption and FPS when drawing 5000 triangles with scale x each using mesh shaders for the second experiment case B **(Note that dynamic boost is not enabled)**.

4.4.2 Second Experiment

The following values were used in addition to the average FPS we got:

Scale factor	Number of triangles	Pixels per triangle	A avg FPS	B avg FPS
0.01	4800	1	109.502	960.122
0.015	2400	2	315.12	1025.367
0.02	1200	4	827.462	1098.669
0.029	600	8	1004.164	1136.964
0.043	300	16	1082.601	1134.504
0.062	150	32	1112.565	1168.13
0.0905	75	64	1114.322	1175.311

Table II: Values used and results in the first experiment

- Using traditional Direct3D 12 graphics pipeline we got the following chart:

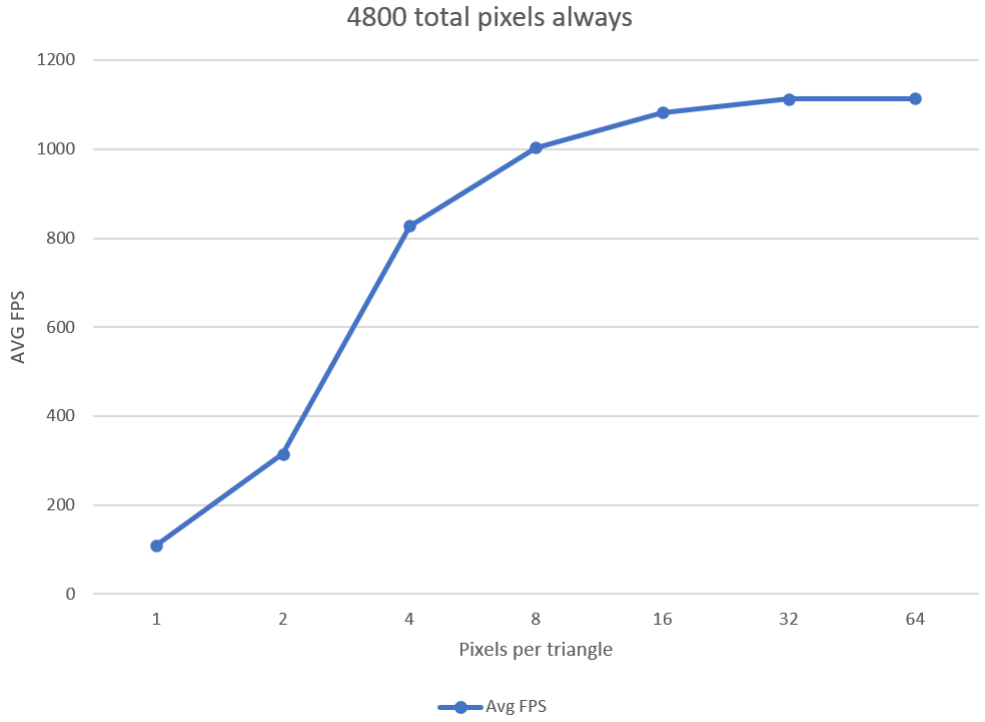

Figure 4.3: Benchmarks when drawing $4800/x$ triangles using normal vertex and pixel shaders for the first experiment with traditional pipeline.

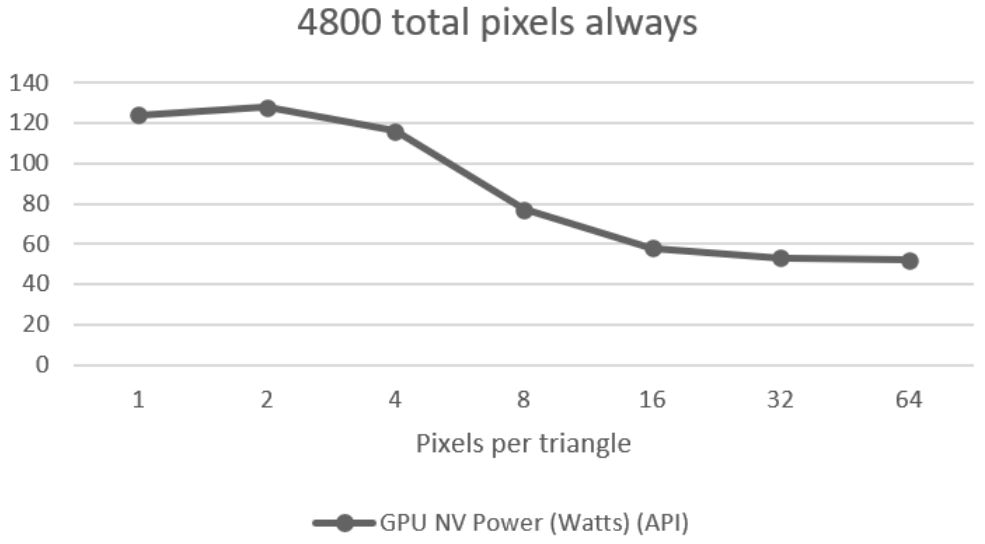

Figure 4.4: Power consumption when drawing $4800/x$ triangles using normal vertex and pixel shaders for the first experiment with traditional pipeline **(Note that dynamic boost is not enabled)**.

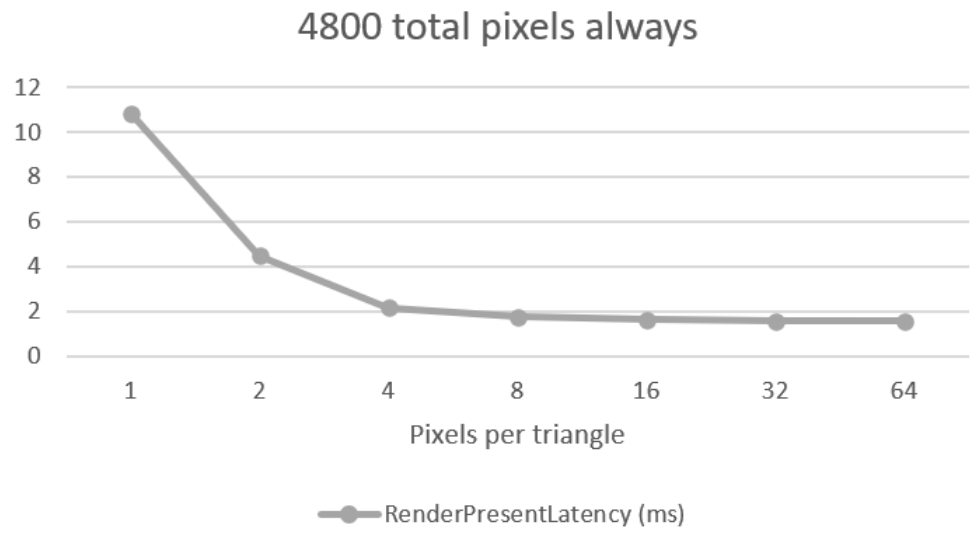

Figure 4.5: Render Present Latency when drawing $4800/x$ triangles using tradtional pipeline for the second experiment with traditional pipeline **(Note that dynamic boost is not enabled)**.

- On the other hand, Using Mesh pipeline pipeline, results represent:

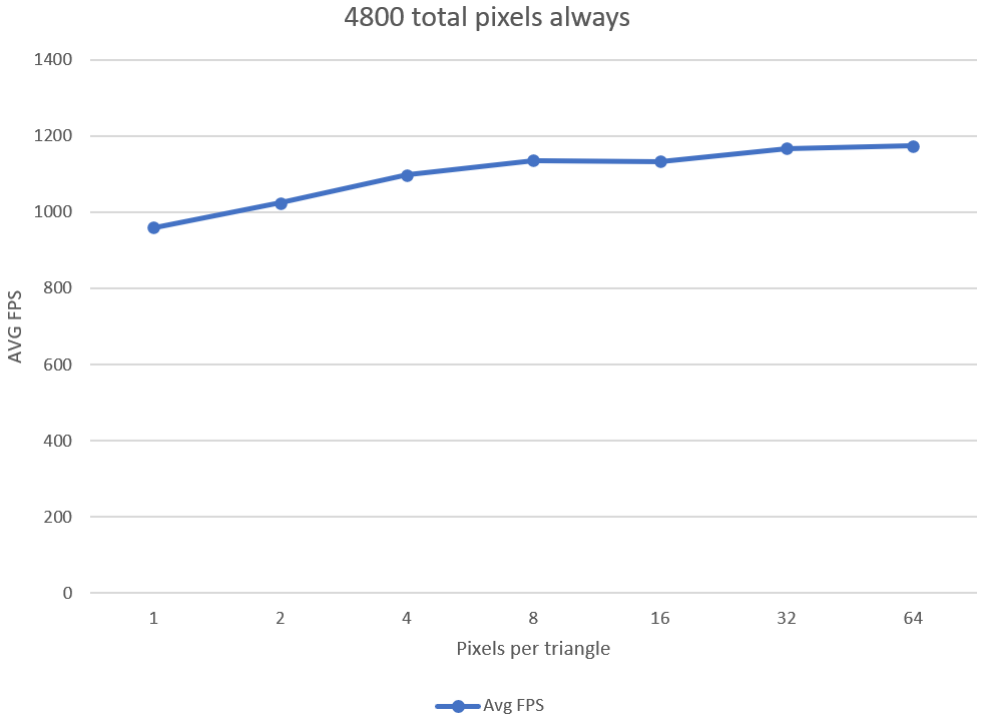

Figure 4.6: Benchmarks when drawing $4800/x$ triangles using mesh pipeline for the second experiment with mesh pipeline.

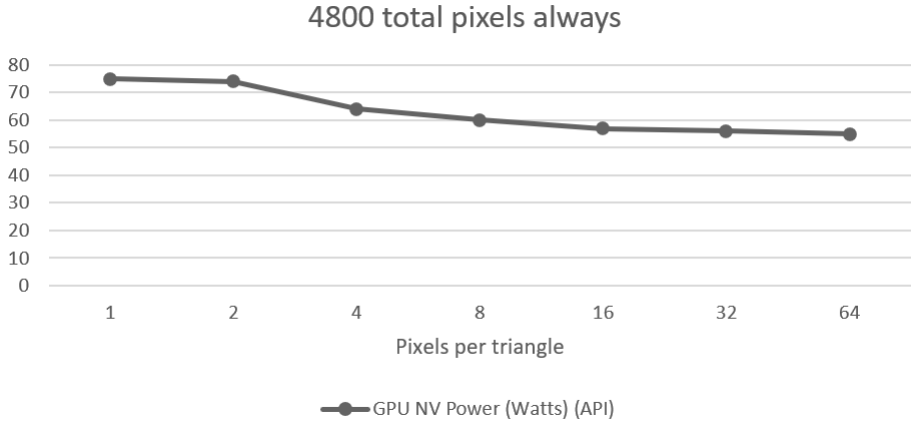

Figure 4.7: Power consumption when drawing $4800/x$ triangles using normal vertex and pixel shaders for the first experiment with mesh pipeline **(Note that dynamic boost is not enabled)**.

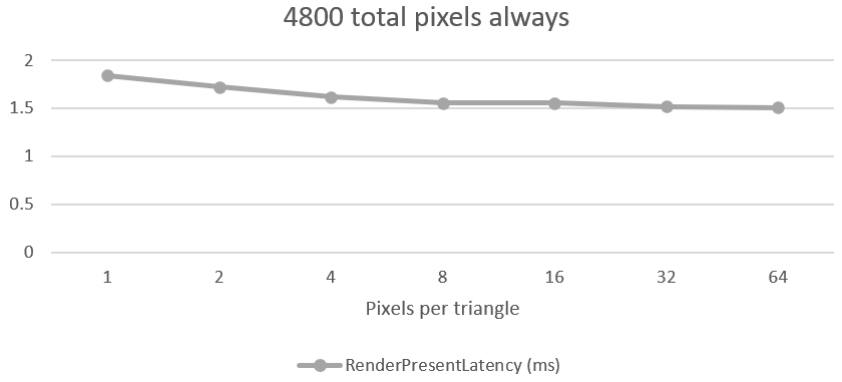

Figure 4.8: Render Present Latency when drawing $4800/x$ triangles using normal vertex and pixel shaders for the first experiment with mesh pipeline **(Note that dynamic boost is not enabled)**.

Figure 4.9: Second experiment drawing 4800/1 triangles.

Figure 4.10: Second experiment drawing 4800/64 triangles.

Chapter 5

Evaluation and Discussion

5.1 Results Evaluation

Looking at the results in the first table we can notice the performance improvement for mesh pipeline over the traditional in the case of 32 pixels per triangle to be **2657%**. Moreover, in case of 1 pixel per triangle it will be **1393 %** which shows the performance advantages of mesh shaders with them making more use of the hardware and parallelism in addition to providing tools for the developers to use those benefits as it is more suitable instead of asking the driver to do it for different cases in a limited number of ways. Furthermore, latency for mesh pipeline was 2.37 milliseconds in its worst case while in traditional pipeline case it is 38.97 milliseconds. However, There is no much difference in terms of GPU power used.

In the second experiment, we used the same total number of pixels because we want to focus on the effect of small triangles on the GPU performance with both pipelines. We can notice also the performance improvement over the traditional pipeline. In addition, Render latency and Power consumption are

much less. However, small triangles draining GPU and causing overshading still exists but with much less difference with the bigger sizes case (dropping from 1114.322 to 1175.311 average frames per second . On the other hand, a traditional pipeline has a drastic drop from 1114.322 to 109.502 average frames per second).

5.2 Mesh shader Limitations

- A single task workgroup has a limited number of how many mesh workgroups it can assign tasks to. 64K was the the limit for the first generation hardware. However, In a draw call, the total number of mesh children has no limits. In addition, if only a mesh shader is used, there is no limit for the number of work groups that its call can generate. [5]
- Direct3D 12 only allows 256 number of Vertices and 256 number of Indices in its output parameters in the main function. [15]
- The primitive topology has to be known
- Dividing into meshlets has to be done the CPU side before using the mesh shader
- In order to be able to use them, RTX graphics card is a must because only the new architectures of GPU support mesh shaders.

5.3 Challenges faced

In this thesis many challenges were encountered that slew down the progress:

- Direct3D 12 is not beginner friendly to start with and usually people learn Direct3D11 first and then learn the differences while still having a difficulty because the documentation is too abstract and resources are few compared to Vulkan which has the same capabilities.

- Mesh shaders are a new technology and only a few research is available for similar methods to try and optimize the GPU because before Nvidia's RTX graphics cards, the hardware architecture was limiting the possible software side solutions even if they were low level and hard to apply commonly.

- Debugging the code which was running on the GPU was hard (such as the ability to use debug break points and see the values of variables defined in the shaders) because IDE does not support it by default and tools available do not have a beginner friendly documentation or only for advanced purposes. In our case, Microsoft PIX was used which contained also many bugs and sometimes the only solution is waiting for a new Nvidia graphics card driver to be released which is not convenient or reliable.

Chapter 6

Conclusion

Performance has always been on the biggest major challenges in the technical industry. For computer graphics technology companies, shipping a graphics card with affordable price and ability to satisfy a wide range of purposes such as video games is one of the major goals for success. In this thesis we took a look at traditional graphics pipeline in Direct3D 12 in the first chapter in addition to the mesh pipeline and mesh shaders' features. Secondly, we compared this new technology with previous research about optimizing the graphics pipeline in the second chapter. Third, we run 2 experiments to compare the performance and analyze the reasons behind those results in the third chapter. In the end, we discussed the results and the reasons behind them in addition to noting the difficulties faced.

6.1 Future works

Further experiments can be done such as model loading and compare the performance in addition to special cases models. Moreover, particle systems and animations are also worth experimenting.

Bibliography cited

[1] *Shaders overview*, https://learnopengl.com/Getting-started/Shaders, Accessed: 2021-09-30.

[2] M. Kenzel, B. Kerbl, D. Schmalstieg, and M. Steinberger, "A high-performance software graphics pipeline architecture for the gpu," *ACM Trans. Graph.*, vol. 37, no. 4, Jul. 2018, ISSN: 0730-0301. DOI: 10.1145/3197517.3201374. [Online]. Available: https://doi.org/10.1145/3197517.3201374.

[3] Microsoft Docs, *Direct3d11 Graphics Pipeline Overview*, https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline, Accessed: 2021-10-01.

[4] Microsoft docs, *Important Changes from DirectX11 to DirectX12*, https://docs.microsoft.com/en-us/windows/win32/direct3d12/important-changes-from-directx-11-to-directx-12, Accessed: 2021-10-01.

[5] Nvidia Developer Blog, *Introduction to Turing Mesh Shaders*, https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/, Accessed: 2021-10-01.

[6] Khronos Combined OpenGL Registry, *NV_mesh_shader OpenGL extension*, https://www.khronos.org/registry/OpenGL/extensions/NV/NV_mesh_shader.txt, Accessed: 2021-10-01.

[7] Microsoft Docs, *IASetVertexBuffers Method Documentation*, https://docs.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12graphicscommandlist-iasetvertexbuffers, Accessed: 2021-10-10.

[8] ——, *IASetIndexBuffer Method Documentation*, https://docs.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12graphicscommandlist-iasetindexbuffer, Accessed: 2021-10-10.

[9] ——, *Introduction to a Resource in Direct3D 11*, https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-resources-intro, Accessed: 2021-10-11.

[10] ——, *Root Signatures*, https://docs.microsoft.com/en-us/windows/win32/direct3d12/root-signatures, Accessed: 2021-10-11.

[11] ——, *Descriptor Heaps Overview*, https://docs.microsoft.com/en-us/windows/win32/direct3d12/descriptor-heaps-overview, Accessed: 2021-10-11.

[12] ——, *Uploading Resources*, https://docs.microsoft.com/en-us/windows/win32/direct3d12/uploading-resources, Accessed: 2021-10-11.

[13] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Mesh optimization," in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '93, Anaheim, CA: Association for Computing Machinery, 1993, pp. 19–26, ISBN: 0897916018. DOI: 10.1145/166117.166119. [Online]. Available: https://doi.org/10.1145/166117.166119.

[14] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg, "Whippletree: Task-based scheduling of dynamic workloads on the gpu," vol. 33, Nov. 2014. DOI: 10.1145/2661229.2661250.

[15] microsoft.github.io, *Mesh Shader DirectX Specs*, https://microsoft.github.io/DirectX-Specs/d3d/MeshShader.html, Accessed: 2021-10-11.

Appendix A

Code Repository

The code is available on Gihub in the following link: https://github.com/TheSharpOwl/Mizu/tree/cc5b809d014fafb6883d8b1c43918734e5d93eb8. The name of the repository is Mizu which is a game engine made for different purposes. In order to be able to run the experiments, an RTX graphics card is required to run the experiment with a mesh shader pipeline. In addition, Direct3D 12 Ultimate should be supported through the latest Windows SDK installed on the machine. The same instructions in README file should be followed but the startup project should be ThousandTriangles. Finally, In order to enable mesh pipeline, the #define MESH_SHADER should be written. On the other hand, it should be removed or commented out in order to use the traditional pipeline.

Appendix B

Tools Used

1. Frameview: an application by Nvidia for measuring frame rates, their times, power, and per-watt-performance on different graphics cards.

2. Nvidia Nsight Graphics: a tool that allows developers for debugging, profiling or expoerting frames in the applications built with Direct3D 11, Direct3D 12, OpenGL, Vulkan or Oculus SDK.

3. PIX: A tool for debugging and performance tuning Direct3D 12 applications on Windows by Microsoft. We used it to debug shaders and make sure that the GPU has the data we wanted to upload.