Optimizing Shader Processing Pipelines using Graph Theory for Mobile Game Development

Renata Puspanegara Ninagan - 13525041
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: renataninagan1@gmail.com , 13525041@std.stei.itb.ac.id

***Abstract*—The rapid evolution of PC hardware has enabled open-world games with hyperrealistic graphics trends; however, a massive performance and thermal disparity remains between PC and mobile platforms. Running unoptimized hyperrealistic graphics on mobile devices causes severe power consumption, extreme overheating, and causing performance drops. To address these hardware constraints, this paper introduces a dynamic shader optimization framework that prunes unnecessary shader operations based on the devices real-time temperature and the hyperrealism aspect of games. All the shaders are modeled as Directed Acyclic Graphs (DAGs). Computational costs (weights) are quantified using CPU clock cycles for individual operations, while visual importance (values) is assigned based on an established hyperrealism framework. The optimization task is framed as a Precedence-Constrained Knapsack Problem (PCKP), where a shader feature can only be rendered if its parent nodes are active. To maximize performance, the problems are solved using dynamic programming. Experimental results under controlled simulated temperatures between 25°C–50°C and a minute observation demonstrate a positive correlation between temperature and games FPS. As temperatures rise, non-essential local textures are incrementally reduced while critical global illumination is prioritized to preserve hyperrealism. The implementation successfully increases mean FPS from 38.7 at 25°C to 42.6 at 50°C, proving that graph theory can maintain stable mobile game FPS with minimal compromises to hyperrealistic rendering.**

Keywords—Shader Optimization, Graph Theory, Directed Acyclic Graph (DAG), Precedence-Constrained Knapsack Problem (PCKP), Mobile Game Development, Thermal Management.

I. Introduction

Open-world games with hyperrealistic graphics are currently trending in the world of game development. In this genre of game, players are able to explore vast, lifelike environments from their computers. Some of the famous games in this genre are Assassin's Creed Shadows and Crimson Desert, with release dates in 2025 and 2026. This hyperrealistic graphics can be achieved as a result of rapid evolution of PC hardware and the maximum usability of it. For example, the game Crimson Desert needs over 150 GB of storage, a minimum of 16 GB of RAM, and at least an RX 5500 XT or GTX 1060 graphics card.

Fig. 1.1 Crimson Desert
Adapted from [1].

This hardware usability created a massive disparity between PC and mobile gaming. While PCs handle hyperrealistic rendering with ease, mobile devices simply cannot. The hardware constraint, primarily on the power consumption and thermal management, prevents mobile developers from pushing graphical limits. Mobile devices are cramped and lack fans, hence running hyperrealistic graphics on mobile rapidly drains the battery, causing severe performance drops, and triggers extreme overheating that can permanently damage the hardware.

However, this does not mean that realistic games are completely unplayable on mobile. Pruning unnecessary shaders at a time is available as an option to make realistic mobile games. With the adjustment based on temperature, players can enjoy realistic games on their mobile phone without the risk of permanently damaging the phone.

II. Theoretical Basis

A. Thermal Management in Mobile Phone

Mobile phones, specifically Android, have their own alarm if a device begins to overheat. There are specific thermal status codes that are translated into specific throttling levels, which can be used for gathering data and for designing an optimal UX. Below is the table and description for each thermal status code[2].

TABLE I. THERMAL STATUS CODE

Thermal status code	Descriptions
THERMAL_STATUS_NONE (0x00000000)	No throttling
THERMAL_STATUS_LIGHT (0x00000001)	Light throttling, UX is fine.
THERMAL_STATUS_SEVERE (0x00000003)	Moderate throttling, UX isn't greatly impacted.
THERMAL_STATUS_SEVERE (0x00000003)	Severe throttling, UX is impacted.
THERMAL_STATUS_CRITICAL (0x00000004)	Platforms tried to reduce power.
THERMAL_STATUS_EMERGENCY (0x00000005)	Key components in the platform are shutting down.

B. Graph Theory

A graph is a set consisting of vertices (nodes) and edges. A graph may contain no edges, but it must contain at least one vertice. Graphs are generally used to represent discrete objects and the relationships between them. Highway route is one example of graphs in real life.

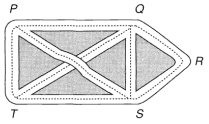
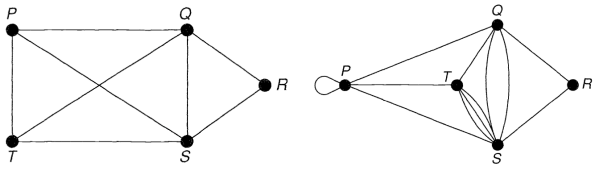


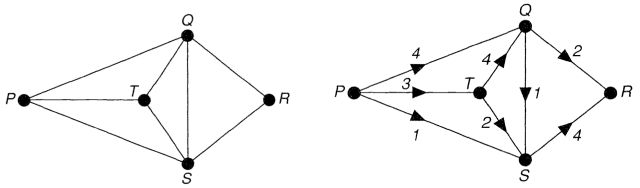
Fig. 2.1 Highway route
Adapted from [3], p. 1.

Based on the existence of loops and multiple edges, graphs are classified into simple graphs and unsimple graphs (or multigraphs) (Fig. 2.2). Based on the orientation of the edges, graphs are classified into undirected graphs and directed graphs (Fig. 2.3). Based on their weight, they are categorized into weighted graphs and unweighted graphs (Fig. 2.4)[5].



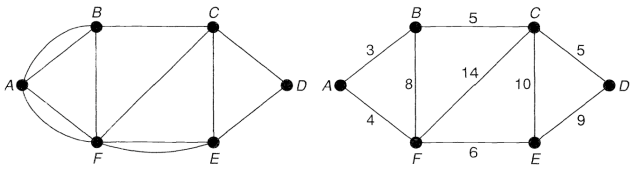
(a) (b)

Fig. 2.2 Simple graph (a) and unsimple graph (b)
Adapted from [3] p. 2–3.



(a))b)

Fig. 2.3 (a) Undirect graph and (b) direct graph
Adapted from [3], p. 3.



(a) (b)

Fig. 2.4 (a) Unweighted graph and (b) weighted graph
Adapted from [3], p. 40.

C. Directed Acyclic Graph

A directed acyclic graph (DAG) is a type of graph that the nodes are linked by directed edges that do not form any cycles. DAGs are used to illustrate one way relationships, DAGs usually used for. A basic example of such one-way relationships is a family tree.

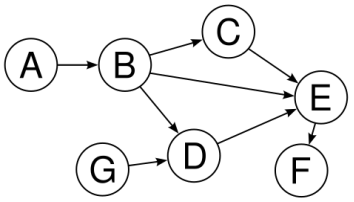


Fig. 2.5 Directed Acyclic Graph
Adapted from [4].

The application of DAGs is common in computer science, with developers and engineers using it for data processing, network architecture, and more. In the context of shader processing, DAGs structure is the best way to process shader because of the nature of the compiler and the processor. The DAGs eliminate dead codes, have no loops which are faster for processors, and enable reuse of the calculated resources.

D. 0/1 Knapsack Problem

Given n distinct items and a knapsack of capacity W. Each item has 2 attributes, weight(w) and value(v). The objective of this problem is to take a subset of items that has maximum total value without exceeding the knapsack's capacity of W. This problem is called 0/1 knapsack. The 0/1 represents two choices of whether an item was taken or not[6].

$$Maximize\ F = \sum_{i=1}^{n} v_i x_i,$$

$$subject\ to \sum_{i=1}^{n} w_i x_i \leq W, x \in \{0,1\},\ i \in \{1,\ldots,n\}$$

The 0/1 Knapsack problem is an NP-hard optimization problem. It can be solved optimally using Dynamic Programming in pseudo-polynomial time, or via direct methods such as Brute Force. The 0/1 Knapsack Problem is usually used for deciding maximizing profit, optimizing logistics, without exceeding limits.

E. *Precedence-Constrained Knapsack Problem*

Precedence-Constrained Knapsack Problem or PCKP is an extension of the well-known 0/1 knapsack problem in which a set of items *I* are in precedence constrain. PCKP is typically modeled using DAGs constraint, so an item *i* can only be chosen if all of its parent items *j* are also chosen. The objective of this problem is the same as the default knapsack, that is to get maximum of total value without exceeding the knapsack's capacity.

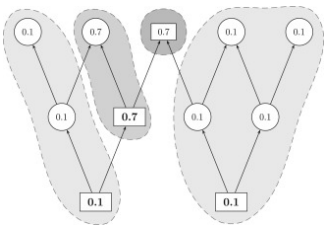


Fig. 2.4 DAGs of Knapsack Problem with precedence-constrained
Adapted from [7].

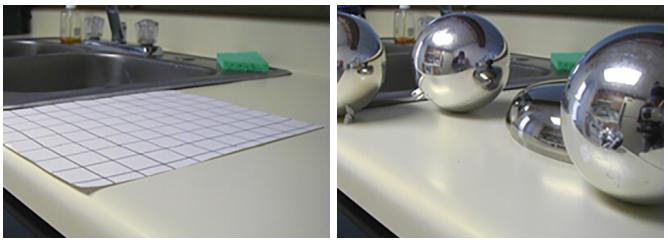
The PCKP problem is also an NP-hard optimization problem. It can be solved optimally using Branch and Bound algorithms, or using Dynamic Programming if the DAGs is tree-structured.

F. *Hyperrealism Aspect of Images*

While art is often said as subjective due to its lack of a formal theory, establishing a standard methodology for art, including replicating realistic images remains a challenge. However, there are some theoretical frameworks that attempt to formalize these aspects of hyperrealism. According to [4], hyperrealism is going through two process of rendering:

1. Global Illumination, including reflections, refractions, shadows and subsurface scattering (e.g., how light passes through liquids).
2. Local Features, including shading, color, textures, and geometry.

This hierarchy of hyperrealism aspect works under four principles: Complete algebraic structures of colors and textures, allowing inconsistent geometry, decomposition of illumination and shading, and statistic-based shader.



(a) (b)
Fig. 2.6: (a) Original photograph. (b) The same photograph with rendered spheres added.
Adapted from [8].

III. Optimizing Shader Processing Pipelines using Graph Theory for Mobile Game Development

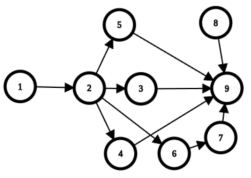
To simulate thermal-based shader optimization, this project is using Unity 2022.62f2 to construct realistic shaders. The scene used is from Unity Universal 3D template. The program will simulate what shader is turning on and off based on the manipulated temperature and how it affects the game FPS. The higher FPS when the temperature is high shows that the optimization works, with minimal cutting of the hyperrealistic graphics.

The idea is to convert the High-Level Shading Language (HLSL) code and the shader graph to a DAGs, then weighted it based on how much operation is done each process and assign a value based on how much it contributes to hyperrealism graphics. Then, using the PCKP Problem, the program turns on/off the less necessary process based on manipulated temperature and capacity of the devices.

A. *DAGs Structure of The Shader*

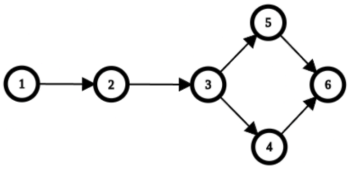
Before optimization can be executed using PCKP, each shader must be converted into a graph first. This project is using 2 shaders, the lit shader (one of the Universal Rendering Pipelines [URP] in Unity) for the environment and a customized shader graph to render the water. The water shader graph has 7 processes: Normal mapping, probe reflection, raymarch reflections, fresnel, refraction, depth and color. The environment shader has 4 processes: Parallax, normal, specular, and detail.

Some of the processes are depending on the previous process in order to make it usable, the relationship of each node can be represented as DAGs. The nodes represent the value and weight, the edges represent the processes, and the node represents the value of each process based on four aspects of hyperrealism. These aspects are ranked by importance on a scale from 4 (highest) to 1 (lowest): (1) Geometry and Shapes, (2) Colors and Textures, (3) Local Shading, (4) Global Illumination. The weight is assigned based on the CPU clock cycle for each operation. Fig. 3.1 illustrates the Directed Acyclic Graph (DAG) representation for each shader, where individual nodes are denoted as Name(Weight, Value).



1. Start
2. Normal (16, 4)
3. Probe Reflections (46, 4)
4. Raymarch Reflections (59, 3)
5. Fresnel (40, 3)
6. Refraction (4, 3)
7. Color (20, 2)
8. Depth (6, 2)
9. End

(a)



1. Start
2. Parallax (34, 1)
3. Normal (30, 1)
4. Specular (50, 3)
5. Detail (36, 1)
6. End

(b)

Fig. 3.1: (a) Environment shader DAGs. (b) Water shader DAGs.

Although modern mobile devices utilize the GPU to process shaders, CPU clock cycles are used to define node weights because they provide a more noticeable difference in computational cost. To quantify these weights, operations are categorized by their complexity: Bitwise operation cost 1 clock cycle, basic operation cost 2 clock cycle. If the process contains 5 basic operations and 1 bitwise operation, the weight of the node is calculated as 11 clock cycles. Operations with explicit mathematical equations are calculated exactly, whereas more complex operations are approximated (Table I).

TABLE II. COST OF OPERATIONS BASED ON CLOCK CYCLE

Operations	Cost (Clock Cycle)
Bitwise	1
Basic operation (+,-,*)	2
Lerp	6
Smoothstep	8
Vector Cross	18
Special Mathematical Functions (Logarithms, Trigonometry)	20
Tiling and Offset	4
Matrix Transformation	36
Reflection	24
Fresnel	40

B. Programs

Now that the weighted DAGs have been modeled, all the data are converted into a list of nodes and pointers. This format enables the code to parse and traverse the graph structure. All code is written in C#.

LISTING I. CLASS AND REPRESENTATION OF GRAPH

```
public class Node {
        public int id;
        public string name;
        public string shaderKeyword;
        public int cost;
        public int priority;
        public bool isTurnedOffKeyword;
        public List<Node> dependents = new List<Node>();
}
private void BuildGraph2()
    {
        graph2Nodes.Clear();
        Node n2 = new Node { id = 2, name =   "Parallax",
shaderKeyword = "_NODE_PARALLAX_OFF", cost = 34, priority
= 1, isTurnedOffKeyword = true };
        Node n4 = new Node { id = 4, name = "Normal",
shaderKeyword = "_NODE_NORMAL_OFF", cost = 30, priority =
1, isTurnedOffKeyword = true };
        Node n5 = new Node { id = 5, name = "Specular",
shaderKeyword = "_NODE_SPECULAR_OFF", cost = 50, priority
= 3, isTurnedOffKeyword = true };
        Node n6 = new Node { id = 6, name = "Detail",
shaderKeyword = "_NODE_DETAIL_OFF", cost = 36, priority =
1, isTurnedOffKeyword = true };

        n2.dependents.Add(n4);
        n4.dependents.Add(n5);
        n4.dependents.Add(n6);

        graph2Nodes.AddRange(new List<Node> { n2, n4, n5,
n6 });
    }
```

Because the DAGs here are structured like a tree, all the nodes are converted into a flat array for dynamic programming best performance. The array is structured so the parent is always at the front of the children. If the algorithm decides to skip a parent node, this pointer tells it exactly how much index to jump to skip all the children completely.

LISTING II. GRAPH-TO-ARRAY FLATTENER

```
private void TraverseAndFlatten(Node node, List<Node>
orderedNodes, int[] nextPeerIndex){
        if (orderedNodes.Contains(node)) return;

        int currentInsertionIdx = orderedNodes.Count;
        orderedNodes.Add(node);

        foreach (var child in node.dependents){
            TraverseAndFlatten(child, orderedNodes,
nextPeerIndex);
        }

        nextPeerIndex[currentInsertionIdx] =
orderedNodes.Count;
    }
```

The array is then calculated using a dynamic programming algorithm to find a subset of processes to include. The program then turns on the maximum shader process possible based on maximum capacity and phone temperature. Maximum capacity for each shader is half the total of initial capacity. While the program runs, FPS are being noted at maximum, minimum, and mean.

C. Game Scene

To simulate thermal-based shader optimization, a realistic shader is required at first. Using the Unity Universal 3D Terminal Scene, the interactions between lights, shadows, and several material types, including concrete, corroded metal, regular metal, glass, and water, can be analyzed.

The Unity Inspector is set to display detailed information for each enabled shader, including temperature adjustments, maximum initial capacity, and frame rate metrics (minimum, maximum, and mean FPS).

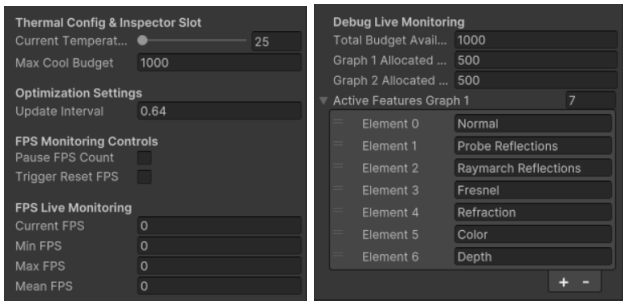


Fig. 3.2: Data displayed on Inspector

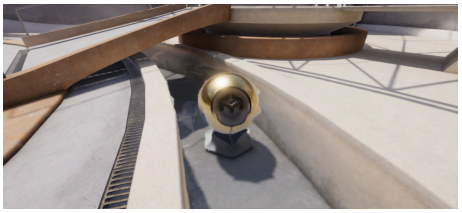
IV. Result

Several tests were conducted across a temperature range of 25°C to 50°C with a fixed maximum capacity of 500. Each test run lasted for 1 minute. The total number of tests was determined by the thermal status codes, beginning at a baseline temperature of 25°C, increasing 5°C each.

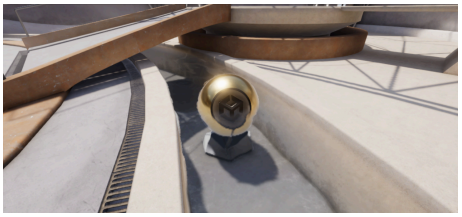
TABLE III. Test Result Based on Temperature

Temp (°C)	FPS Count in 1 Minute		
	Minimum	***Maximum***	***Mean***
25	6.8	50.3	38.7
30	6.2	50.2	38.9
35	10.3	56.7	40.5
40	9.4	55.9	40.1
45	8.3	57.2	42.8
50	8.7	56.4	42.6

Water shader processing reduces at three temperature points: 35°C, 45°C, and 50°C. Similarly, environment shader processing reduces at 40°C and 45°C. The data shows a positive correlation between temperature and FPS. Because the simulation was conducted under controlled temperature conditions, the FPS continuously increases rather than stabilizing.



(a)



(b)

(c)

(d)

Fig. 4.1: (a) Game Shader at 25°C, (b) Game Shader at 35°C, (c) Game Shader at 40°C, (d) Game Shader at 45°C.

Fig. 4.1 shows that this shoulder optimization still holds in principle of hyperrealism. As in the image, global illumination becomes the top priority to preserve. At 40°C, the textures are pruned for optimization, but hardly noticeable. At 45°C, the program had no choice but to cut most of the detail and make the games look no longer realistic.

V. Conclusion

Graph theory, specifically the DAGs, combined with knapsack problems, can be used to optimize shader processing in mobile games. The node represents the weight and value each process carries, and dynamic programming chooses the subset with maximum value, shader processing can be optimally pruned without risking the hyperrealism aspect of the games.

Acknowledgment

First, the author would like to thank God for all the guidance throughout the process of learning thus writing the contents of this paper. The author would also like to thank the lecturer of IF2120 Discrete Mathematics, Mr. Rinaldi Munir for sharing their knowledge, guides the students throughout the learning process in the class. Not to forget, the author would also like to thank her family and friends who have given the support throughout the entire semester.

References

[1] J. Thang, "Crimson Desert review: The most gorgeous disaster I've ever played," Men's Journal, Feb. 27, 2025. [Online]. Available: https://www.mensjournal.com/entertainment/crimson-desert-review-the-most-gorgeous-disaster-ive-ever-played. [Accessed: Jun. 19, 2026].

[2] "Thermal mitigation," Android Open Source Project, Documentation. [Online]. Available: https://source.android.com/docs/core/power/thermal-mitigation?hl=id. [Accessed: Jun. 19, 2026].

[3] R. J. Wilson, *Introduction to Graph Theory*, 4th ed. Harlow, England: Longman, 1996.

[4] L. Norton, "Using directed acyclic graphs to store transfer patterns," Linus Norton, Dec. 10, 2018. [Online]. Available:

https://ljn.io/posts/using-directed-acyclic-graphs-to-store-transfer-patterns. [Accessed: Jun. 19, 2026].

[5] R. Munir, “Graf (Graph) - Bagian 1,” Bahan Kuliah IF2120 Matematika Diskrit, Program Studi Teknik Informatika, STEI-ITB, 2026. [Online]. Available:https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf. [Accessed: Jun. 19, 2026].

[6] “Knapsack Problem,” Algorithms for Competitive Programming, Jun. 8, 2024. [Online]. Available: https://cp-algorithms.com/dynamic_programming/knapsack.html. [Accessed: Jun. 19, 2026].

[7] D. Espinoza, M. Goycoolea, and E. Moreno, “The precedence constrained knapsack problem: Separating maximally violated inequalities,” Discrete Applied Mathematics, vol. 194, pp. 65–80, Oct. 2015.

[8] E. Akleman et al., “Hyper-Realist Rendering: A Theoretical Framework,” 2024, arXiv:2401.12853v1. [Online]. Available: https://arxiv.org/html/2401.12853v1. [Accessed: Jun. 18, 2026].

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Renata Puspanegara Ninagan
13525041