Semestrální práce 17
Hierarchical View-Frustum Culling for Z-buffer Rendering

Tomáš Bubeníček[1]
Katedra počítačové grafiky a interakce,
Fakulta elektrotechnická, ČVUT Praha

Abstract

Implement hierarchical view frustum culling for large scale scenes consisting of triangles. First, construct a bounding volume hierarchy (BVH) using top-down method, middle point subdivision. Avoid rendering such BVH nodes that cannot be visible (out of viewing frustum) usually known as view frustum culling.

Keywords: Frustum culling, BVH, GPU rasterization

1. Úvod

Při vykreslování scény pomocí metod rasterizace se pro každý vrchol provádí transformace, jejímž cílem je projektovat body ve scéně do bodů v clip space. V tomto souřadném prostoru lze poté snadno určit zda primitiva náležící k daným vrcholům (v OpenGL trojúhelníky) budou ležet ve výsledném obraze. Tyto primitiva jsou poté rasterizovány.

Pro většinu scén je při transformaci do clip-space výsledkem, že velká část transformovaných primitiv neleží v té části clip space, ve které probíhá rasterizace. Cílem této práce je tedy implementovat systém, který omezí počet transformovaných vrcholů tak, aby se transformovaly pouze ty vrcholy, které náleží k objektům, které budou po transformaci vidět na výsledném obraze.

2. Popis algoritmu

Algoritmus pracuje s datovou strukturou BVH (Bounding Volume Hierarchy) postavenou nad jednotlivými objekty ve scéně. Jednotlivé uzly BVH mají osově zarovnanou obálku ve tvaru krychle (Axis Aligned Bounding Box – AABB) a každý vnitřní uzel má dva potomky. Obálky jednotlivých uzlů poté testujeme vůči pohledovému jehlanu (viewing frustum) kamery vykreslující scénu. Tento komolý jehlan reprezentuje objem, ve kterém bude po transformaci do clip-space probíhat samotná rasterizace.

Při testu průniku obálky a pohledového jehlanu může dojít k třem různým výsledkům, podle kterých se při vykreslování rozhodujeme:

- Obálka je mimo pohledový jehlan - Uzel a všechny jeho potomky nevykreslujeme.
- Obálka je celá uvnitř pohledového jehlanu - Uzel a všechny jeho potomky vykreslujeme.

[1] B4M39DPG – Tomáš Bubeníček, zimní semestr 2019/20

```
short AABB::checkFrustum(Frustum& f)
{
  short result = INSIDE;
  for (i = 0; i < 6; i++){
    glm::vec4& p = f.planes[i];
    m_prod = (p.x * m[p.x<0].x) + (p.y * m[p.y<0].y) + (p.z * m[p.z<0].z);
    if (m_prod > -p.w)
            return OUTSIDE;

    n_prod = (p.x * m[p.x>0].x) + (p.y * m[p.y>0].y) + (p.z * m[p.z>0].z);
    if (n_prod > -p.w)
            result = INTERSECTS;
  }
  return result;
}
```

Obrázek 1: Test průniku AABB frustum pomocí šesti testů průniku s rovinou. Pole `this->m` má v pozici 0 minimální bod a v pozici 1 maximální bod osově zarovnané obálky. Pole `f.planes` obsahuje implicitní zadání všech šesti rovin jehlanu.

- Obálka protíná stěnu jehlanu - Provádíme test průniku obálky na oba potomky uzlu, pokud uzel nemá potomky, vykreslíme.

Celý tento algoritmus je tedy založen na efektivní implementaci tohoto testu. V následujících podkapitolách je tedy popsán algoritmus testování a několik způsobů pro jeho zrychlení.

2.1. Test průniku AABB-frustum

V [1] jsou popsány dva přístupy, pomocí kterých lze test průniku obálky a pohledového jehlanu provádět. První metoda je provádět operaci transformace do clip space na samotné vrcholy obálek, jelikož ve výsledném souřadném systému je určit, zda obálka bude vidět na výsledném obraze je relativně snadná - Stačí porovnat obálku transformovaného AABB a AABB reprezentující rasterizovaný objem. Pro transformaci obálky do clip space je třeba ale 72 operací násobení, což nemusí být ve všech implementacích efektivní (obzvláště v implementacích nevyužívajících paralelismus či SIMD instrukcí).

Druhá, mnou implementovaná metoda, reprezentuje frustum jako šest rovin ve world space, vůči kterým se postupně provádí testy průniku. Pokud se AABB nachází na straně roviny reprezentující prostor vně frustum, AABB je mimo pohledový jehlan. Naopak pokud všech šest testů průniku vrátí že se AABB nachází v části poloprostoru reprezentujícího vnitřní část pohledového jehlanu, nachází se obálka uvnitř. Pokud obálka protíná alespoň jednu rovinu, protíná také stěnu výsledné obálky.

2.2. Test průniku AABB-rovina

Místo testování všech osmi vrcholů obálky je možné testování urychlit pomocí testování pouze dvou bodů n a p, ležících na úhlopříčce nejbližší normále roviny. Určení těchto dvou bodů je snadné: Jsou určeny podle jednotlivých komponent normálového vektoru – Pokud je komponenta kladná, bod p leží v dimenzi komponenty na pozici bodu $max(AABB)$ a bod n poté leží v dimenzi komponenty na pozici $min(AABB)$.

Pro určení průniku je třeba rozhodnout, zda bod n leží uvnitř poloprostoru definovaného stěnou pohledového jehlanu. Pokud ne, obálka leží celá vně. Pokud ano, provede se stejný test na bod p. Pokud tento bod leží vně, obálka protíná rovinu, pokud ano, obálka leží celá uvnitř poloprostoru. Tuto implementaci bez jakéhokoliv dodatečného zrychlení lze nalézt na obrázku 1, který je založen na implementaci v [3].

2.3. Maskování rovin

Zanořování se do vnitřních uzlů BVH nastává pouze v situaci, kdy se obálka protíná s alespoň jednou rovinou pohledového jehlanu. Nemá ovšem smysl testovat ve vnitřních uzlech ty roviny, o kterých již víme, že se neprotínají s rodičem. Jedno z zrychlení algoritmu je tedy při testu průniku určit, které roviny mají být testovány a které testovány být nemusí.

2.4. Časová koherence

Šance, že se mezi dvěma snímky změní rovina, u které zjistíme že se obálka nachází vně pohledový jehlan , je relativně nízká. Jelikož ukončujeme test průniku v moment, kdy je obálka vně jednu rovinu, je vhodné testovat rovinu která selhala test průniku v minulém snímku jako první.

3. Potíže při implementaci

Implementaci jsem se rozhodl provést pomocí knihoven SDL a OpenGL. Nejprve jsem se rozhodl implementaci testovat na velkých trojúhelníkových sítí poskytnutých pro testování, což se ale při samotném vývoji ukázalo jako relativně nevhodná volba - Tyto testovací soubory zabírají často skoro až gigabajt místa, což při spuštění programu v debug módu může trvat mému jednoduchému .obj loaderu až minutu a půl na načtení. Pro načítání dat jsem tedy později implementoval systém, kde se při prvním načtení .obj souboru uloží binární soubor reprezentující data. Při dalším načítání se nejprve zkontroluje, zda soubor existuje a pokud ano, načte se ten. Načítání i velkého binárního souboru je poté velmi rychlé.

Samotné modely poskytnuté na testování neměly vždy přiřazené normály, takže jsem se rozhodl vrcholům ve vertex shaderu přiřazovat náhodně barvu, aby byly jednotlivé vrcholy rozlišitelné. Za zmínku také stojí postup na získání jednotlivých rovin pohledového jehlanu z projekční matice, k čemuž jsem využil postup popsaný v [2].

Poskytnuté modely bylo možné na grafické kartě GTX 1070 vykreslovat celé relativně rychle (s jedním modelem *asianDragon.obj* byl výsledek více jak 600 fps), takže v testovaných scénách bylo nutno těchto modelů vložit více (cca 60). Pro definici této scény je využit textový soubor, kde je možno také určit, zda jednotlivé modely ve scéně jsou vykreslovány v celku, či zda mají být předzpracovány a rozděleny pomocí midpoint split metody na menší modely s předem definovaným maximálním počtem trojúhelníků.

Při implementaci BVH jsem také vytvořil jednoduché vykreslování jednotlivých vrstev stromu, abych se ujistil, že mnou implementované midpoint split dělení je správně. Na semestrální práci jsem pracoval cca 100 hodin, s většinou času strávenou na jiných věcech než je samotné programování osekávání pohledovým jehlanem (načítání obj, vykreslování, získání rovin z projekční matice, pohyb kamery, vytváření scén, měření výsledků psaní reportu). Na samotné tvorbě BVH a implementaci frustum culling algoritmu jsem strávil cca 20 hodin.

4. Naměřené výsledky

Hlavní část měření probíhala na mém domácím stolním počítači, který má následující hardwarové a softwarové parametry:

- Procesor Intel Core i5-7600K 64bit@3.80GHz (4 jádra)
  - L1 4×32 KB

  - L2 4×256 KB
  - L3 6 MB
- Grafická karta Nvidia GTX 1070 8GB
- 24 GB DDR4 RAM
- Samsung SSD 960 Evo 250GB NVMe disk
- OS Windows 10 Pro verze 1903
- Microsoft Visual Studio Community 2017
- Parametry překladače:
  - Intel C++ Compiler 19.0
  - /std=c++17
  - /O2
  - /Oi (Intrinsic funkce)
  - /Qipo (Mezisouborová optimalizace)
  - /GA (Optimalizace pro Windows OS)
  - /Qopt-matmul (Optimalizace maticového násobení)
  - /Qparallel (Optimalizace pomocí paralelizace)
  - /Quse-intel-optimized-headers (Hlavičkové soubory optimalizované pro intel)

Při testování jsem měřil několik odlišných proměnných po vteřinovém intervalu:

- *FPS* - Počet snímků od posledního zapsaného měření
- *Frametime* - Doba, kterou trval výpočet posledního snímku (v milisekundách)
- *Drawcalls* - Počet volání vykreslení na grafické kartě
- *GLTimerQuery* - Skutečný čas vykreslování na GPU získaný pomocí OpenGL dotazu `GL_TIME_ELAPSED` (v milisekundách)
- *CPUtime* - Čas od počátku procházení BVH až po konec včetně volání OpenGL vykreslovacích funkcí (v milisekundách)
- *Traversaltime* - Čas trvání odděleně zavolaného procházení BVH, které nevolá žádné OpenGL příkazy a pouze navštíví potomky a spočítá počet objektů, které by byly zobrazeny (v milisekundách)

Důležité je zmínit, že samotná optimalizace testů průniku bude znatelná nejvíce ve výsledku *traversaltime*, jelikož ten reprezentuje pouze čas průchodu stromem bez volání funkcí OpenGL, které poté pracují s trojúhelníkovou sítí uloženou v jednotlivých listech.

Všechny testy jsem prováděl vícekrát (cca 10 běhů), každý průběh uložil do .csv souboru a jednotlivé průběhy poté zprůměroval pomocí přiloženého Python skriptu. Pro vykreslení grafů byly poté výsledky vyhlazeny konvolucí s hammingovým okénkem, aby byl vidět průměrný výsledek ve větším časovém úseku. Pro některé scény jsem porovnával implementaci s zapnutou optimalizací (pomocí maskování a s využitím koherence) a bez optimalizací (testováním vždy všech šesti rovin). Také jsem otestoval scény bez ořezávání pohledovým jehlanem.

4.1. asianDragons-Split

Pro hlavní část měření jsem vytvořil scénu *asianDragons-Split*. Scéna obsahuje 60 instancí modelu *asianDragon.obj* s velkými rozestupy a má varianty, ve kterých se model rozdělí do listů s maximální velikostí 25 tisíc, 50 tisíc, 100 tisíc, 200 tisíc, 500 tisíc, milion a dva miliony trojúhelníků. Samotný model obsahuje 7,2 milionů trojúhelníků a byla také vytvořena varianta scény, kde se model na menší části nedělí. Kamera scénou prolétává po Bézierově křivce.

Výsledky měření této scény bez dodatečných optimalizací je vidět na obrázku 2. Můžeme si povšimnout, že pro scénu s modely rozdělených do listů s 25 tisíci trojúhelníky až 200 tisíc trojúhelníků je celková rychlost vykreslování přibližně stejná. Scéna s nejméně trojúhelníků v listech je nepatrně rychlejší. U listů s 500 tisíců trojúhelníků a více je poté pokles ve výkonu, kde vykreslování plného modelu bez dělení na další listy může mít ve výsledku méně jak poloviční počet snímků za vteřinu oproti ostatním scénám.

Je také možné si povšimnout, že moderní dedikované grafické karty nejsou tolik limitovány samotným počtem volání vykreslovacích příkazů – dvojnásobný nárůst v počtu vykreslení mezi scénou s 25 tisíc trojúhelníků v listech a scénou s 50 tisíc způsobuje minimální změnu ve výsledném běhu na grafické kartě. Kde tento výsledek známý již je je vidět při porovnávání běhu *GLTimerQuery* a *CPUtime*. *CPUtime* ukazuje, jak rychle driver grafické karty předá běh zpět samotnému programu a grafická karta může běžet asynchronně vůči CPU. V tomto případě je vidět, že drivery karty nepracují nejlépe pokud se vykreslují příliš velké modely či pokud je volání vykreslování příliš. Pro nejlepší využití driverů je v listech vhodné mít pro testovanou konfiguraci přibližně 100 tisíc trojúhelníků.

Jako poslední se zaměříme na dobu samotného průchodu všemi uzly BVH bez volání OpenGL vykreslovacích funkcí a můžeme si povšimnout, že i při stálém testování pohledového jehlanu pomocí šesti rovin je čas traverzace cca jedna setina času vykreslování a další optimalizace budou mít tedy minimální vliv na celkové trvání snímku. Toto bylo potvrzeno i pomocí profilování s programem Intel VTune Amplifier, který ukazoval výsledek, kde více jak 97% času běhu aplikace byl běh samotného OpenGL driveru (buď volání vykreslovacích příkazů, nebo čekání na jejich dokreslení).

Poté jsem provedl stejný test na této scéně se zapnutou optimalizací pomocí maskování stěn a časové koherence s podobnými výsledky. Měření je možno vidět na obrázku 3 a je ve výsledném počtu snímků za vteřinu velmi podobné. Je vidět lehké zrychlení v čase traverzace pro případ největšího stromu (s listy s méně jak 25 tisíci trojůhelníků), a z pohledu efektivního využití driverů je stále nejlepší využít BVH s počtem listů okolo 100 tisíc.

Pro tuto scénu jsem prováděl také měření s vypnutým ořezávání pohledovým jehlanem (ale s trojúhelníky rozdělenými opět do různě velkých listů), které je možno vidět na obrázku 4. Výsledky jsou časově konstantní (jelikož se nemění počet vykreslovacích příkazů) a pro asynchronní vykreslování opět vede scéna s modely s max 100 tisíc trojúhelníky.

Čas dělení objektu *asianDragon.obj* na různě velké listy, počet celkových listů po vytvoření 60 instancí dělených objektů ve scéně a čas stavby výsledného BVH nad celou scénou je vidět v tabulce 1. Pro další scény byly časy velmi podobné.

4.2. asianDragons-Packed

Scéna *asianDragons-Split* je pro testování efektu optimalizace možná nevhodná, protože mezi jednotlivými modely je relativně velká mezera a tak se často stává, že stěnu pohledového jehlanu neprotíná žádný model ve scéně. Tím pádem neprobíhá při traverzaci tolik testování průniku AABB-frustum, protože se testuje pouze tehdy, když byl průnik rodičovské obálky.

	Čas dělení	Počet listů	Čas stavby
25k	4853 ms	26880	52.22 ms
50k	4383 ms	13980	26.06 ms
100k	3876 ms	6780	12.39 ms
200k	3358 ms	3360	5.57 ms
500k	2569 ms	1200	1.79 ms
1000k	1916 ms	540	0.87 ms
2000k	1465 ms	300	0.52 ms

Tabulka 1: Čas dělení objektu *asianDragon.obj* a stavby BVH nad 60 instancemi děleného objektu.

V jiných případech se potomci buď rovnou vykreslí bez testování, nebo zahodí a nevykreslují vůbec. Proto jsem vytvořil novou scénu *asianDragons-Packed* podobnou scéně předchozí s tím rozdílem, že modely mezi sebou mají podstatně menší mezery a při průletu kamerou je jich více v průniku se stěnou pohledového jehlanu. Výsledky těchto měření jsou na obrázcích 5(bez optimalizace), 6(s optimalizací) a 7(s vypnutým ořezáváním).

Výsledek je prakticky totožný s předchozí scénou a větší zrychlení traverzace není viditelné. Další testované scény jsem tedy dále netestoval ve variantě bez optimalizací, protože rozdíl ve výsledcích byl na mé grafické kartě zanedbatelný.

4.3. part_of_pompeii

Jako další scéna byla vytvořena scéna *part_of_pompeii*. Tato scéna obsahuje 20 instancí modelu `part_of_pompeii.01.final.combined.obj`, který obsahuje cca 16 milionů trojúhelníků a byla opět testována pro různě velký počet trojúhelníků v listech. Při testování (na obrázku 8) byl framerate opět nejlepší pro nejjemnější dělení a stejně jako v minulých případech byl pro nejlepší využití driverů a asynchronního vykreslování ideální počet trojúhelníků v listu v rozmezí 50 až 100 tisíc.

4.4. testování na dalším stroji

Implementaci jsem také testoval na notebooku s integrovanou grafickou kartou a těmito parametry:

- Procesor Intel Core i5-8250U 64bit@3.40GHz (4 jádra, 8 vláken)
  - L1 4×32 KB
  - L2 4×256 KB
  - L3 6 MB
- Integrovaná grafická karta Intel® UHD Graphics 620
- 8 GB DDR4 RAM
- Seagate ST1000LM035 1TB hard disk
- OS Fedora 31, Linux 5.3.7
- Parametry překladače:

	Čas dělení	Počet listů	Čas stavby
25k	1768 ms	26880	21.18 ms
50k	1697 ms	13980	9.67 ms
100k	1449 ms	6780	4.04 ms
200k	1279 ms	6780	1.89 ms
500k	1026 ms	1200	0.67 ms
1000k	800 ms	540	0.27 ms
2000k	640 ms	300	0.18 ms

Tabulka 2: Čas dělení objektu *asianDragon.obj* a stavby BVH nad 60 instancemi děleného objektu na druhém stroji.

- GCC 9.2.1
- -std=gnu++17
- -O3

Měření bylo provedeno na scéně *asianDragons-Packed* a výsledky je možno vidět na obrázku 9. Na datech je zřejmý slabší výkon grafické karty, ale poukazují také na rozdílnou implementaci grafického driveru, jelikož *CPUtime* je nejrychlejší pro scénu s nejméně voláním vykreslovacích příkazů a nejvíce trojúhelníky v listech. Naopak nejdéle trvá grafickým driverům činnost při velkém množstvím vykreslovacích příkazů – Graf byl oříznut, aby byly vidět menší hodnoty, ale scéna s 25 tisíci trojúhelníků v listech měla *CPUtime* více jak 330 milisekund za snímek při cca 10,000 volání vykreslování.

Ačkoliv je stroj grafickým výkonem podstatně horší, rychlost dělení objektu a stavby stromu je podstatně vyšší – částečně způsobeno také jinou volbou překladače a agresivnějším nastavením optimalizace. Tabulka 2 znázorňuje rychlost dělení a stavby stromů s různě velkými listy.

5. Závěr

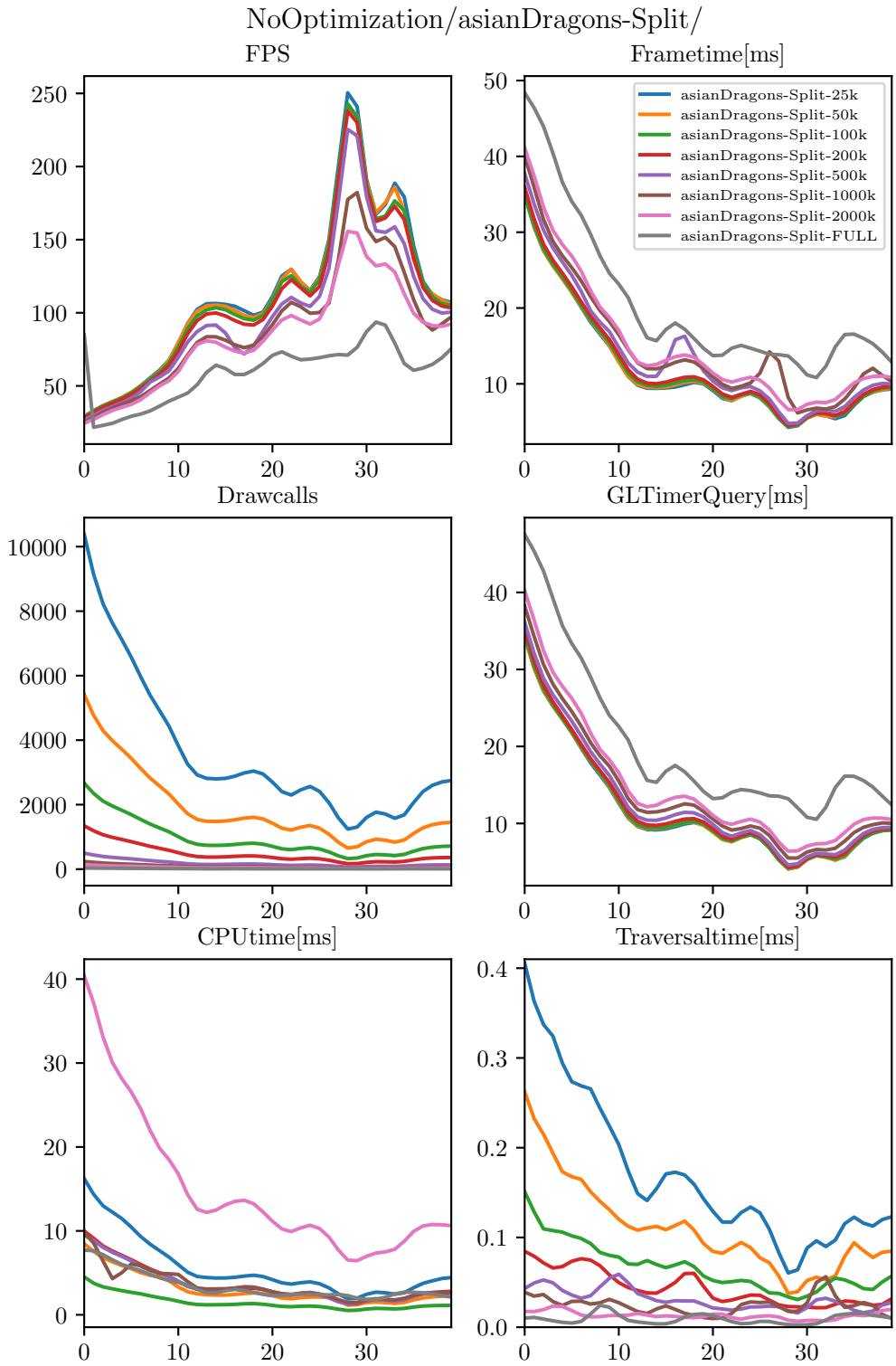
V této práci jsem implementoval algoritmus osekávání pohledovým jehlanem pomocí hierarchické datové struktury BVH. Ačkoliv zrychlení pomocí samotného testu intersekce je značné (závislé na pohledu kamery do scény), nepodařilo se mi získat znatelné zrychlení při využití maskování rovin a časové koherence. Zjistil jsem také, že pro nejlepší využití driveru grafické karty může být počet trojúhelníků v listech závislý na zařízení a pro různé platformy je ideální počet jiný.

Reference

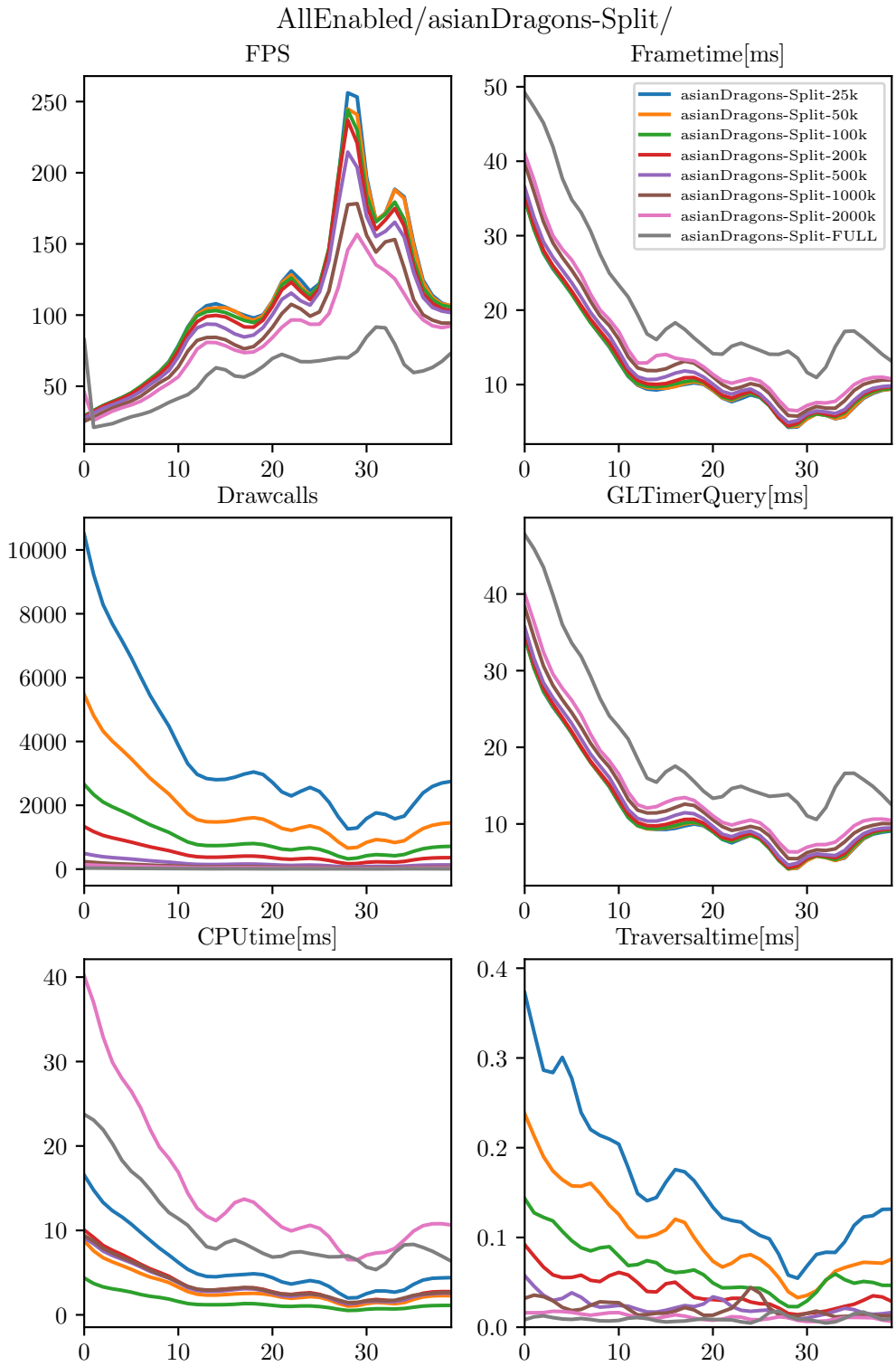
[1] U. Assarsson and T. Moller. Optimized view frustum culling algorithms for bounding boxes. *Journal of graphics tools*, 5(1):9–22, 2000.

[2] G. Gribb and K. Hartmann. Fast extraction of viewing frustum planes from the world-view-projection matrix. *Online document*, 2001.

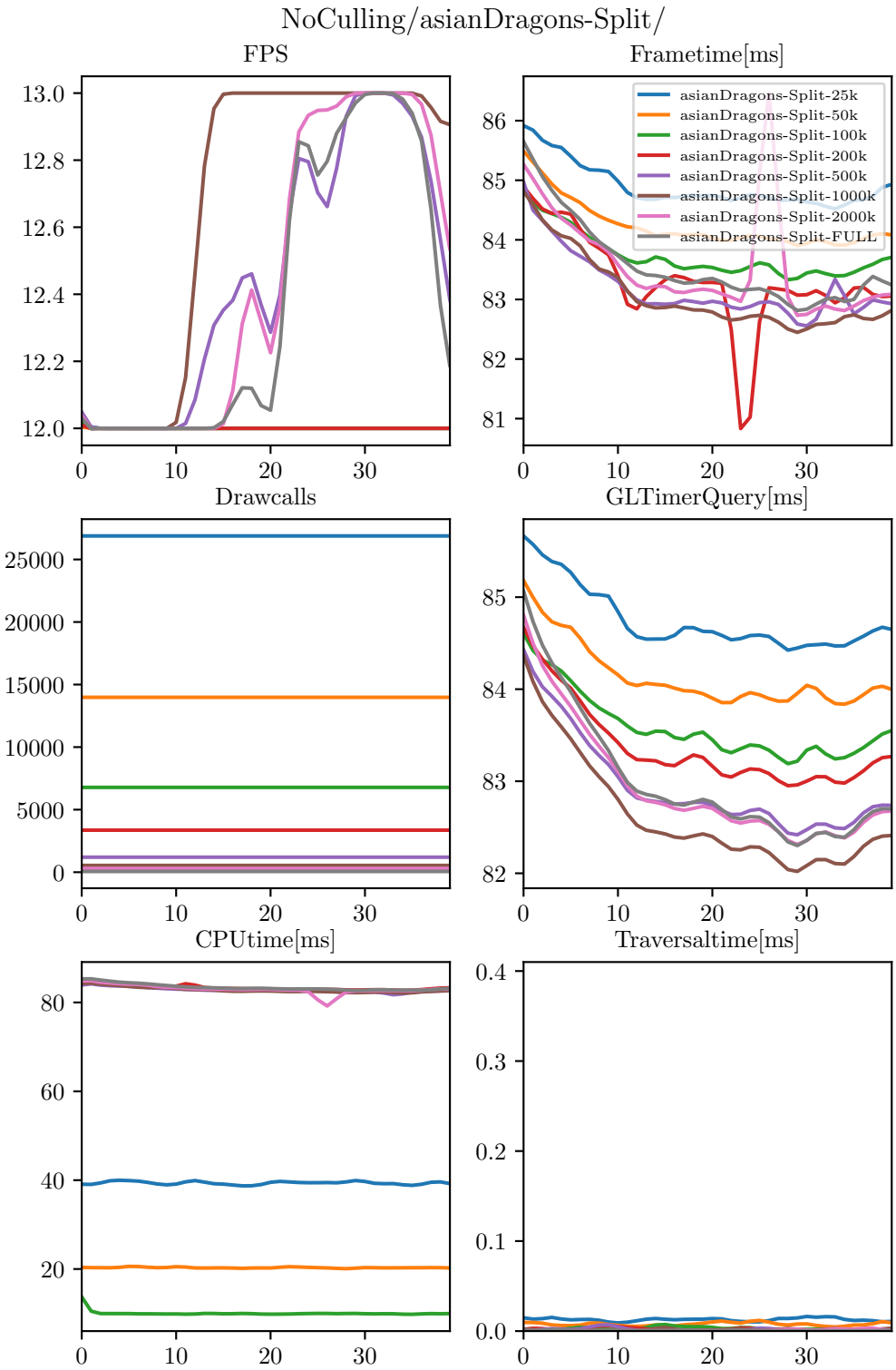
[3] D. Sýkora and J. Jelínek. Efficient view frustum culling. In *Central European Seminar on Computer Graphics*, 2002.



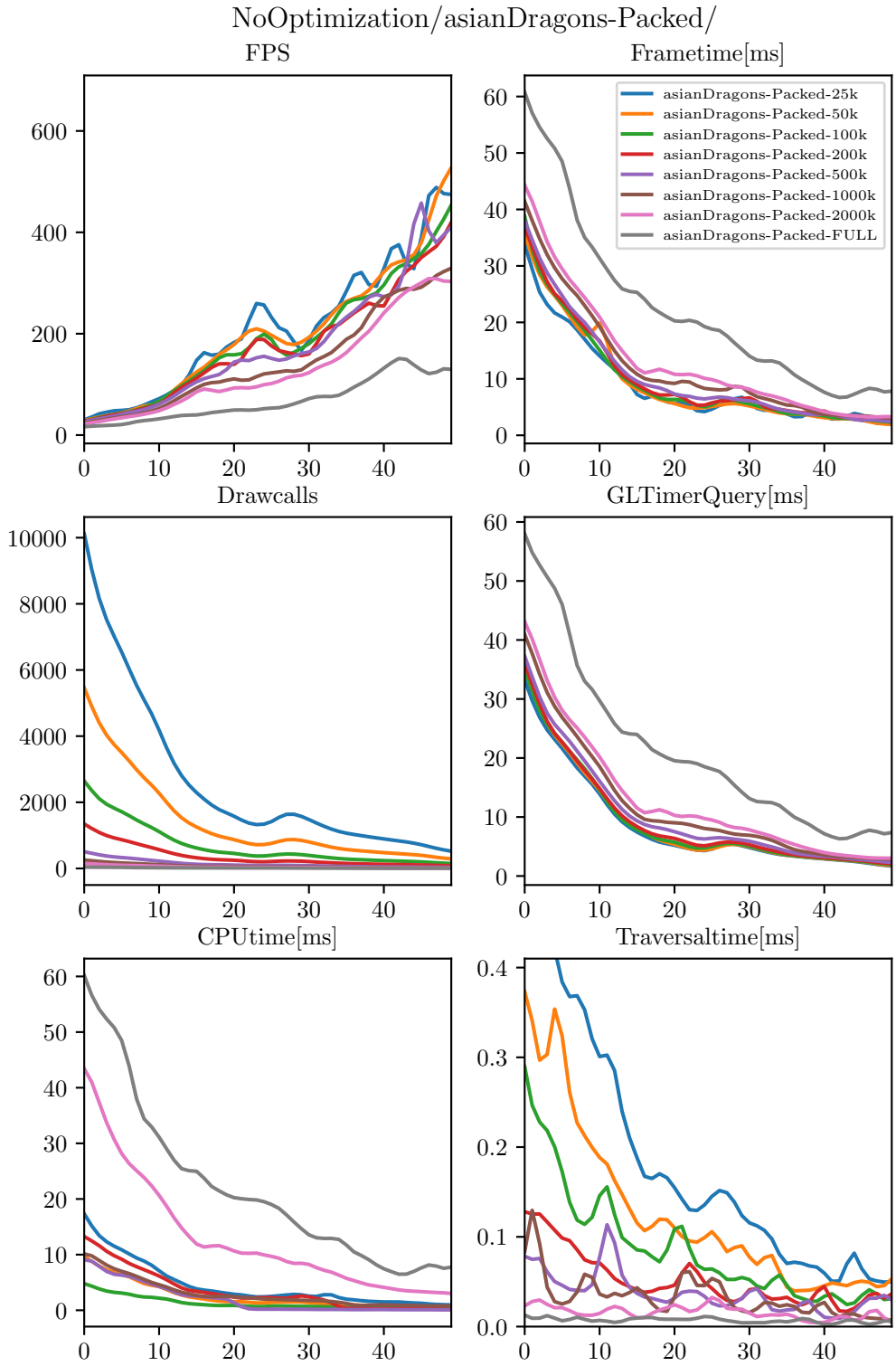
Obrázek 2: Měření na scéně *asianDragons-Split* bez dodatečných optimalizací s různě velkým množstvím trojúhelníků v listech. Osa x reprezentuje čas měření v sekundách.



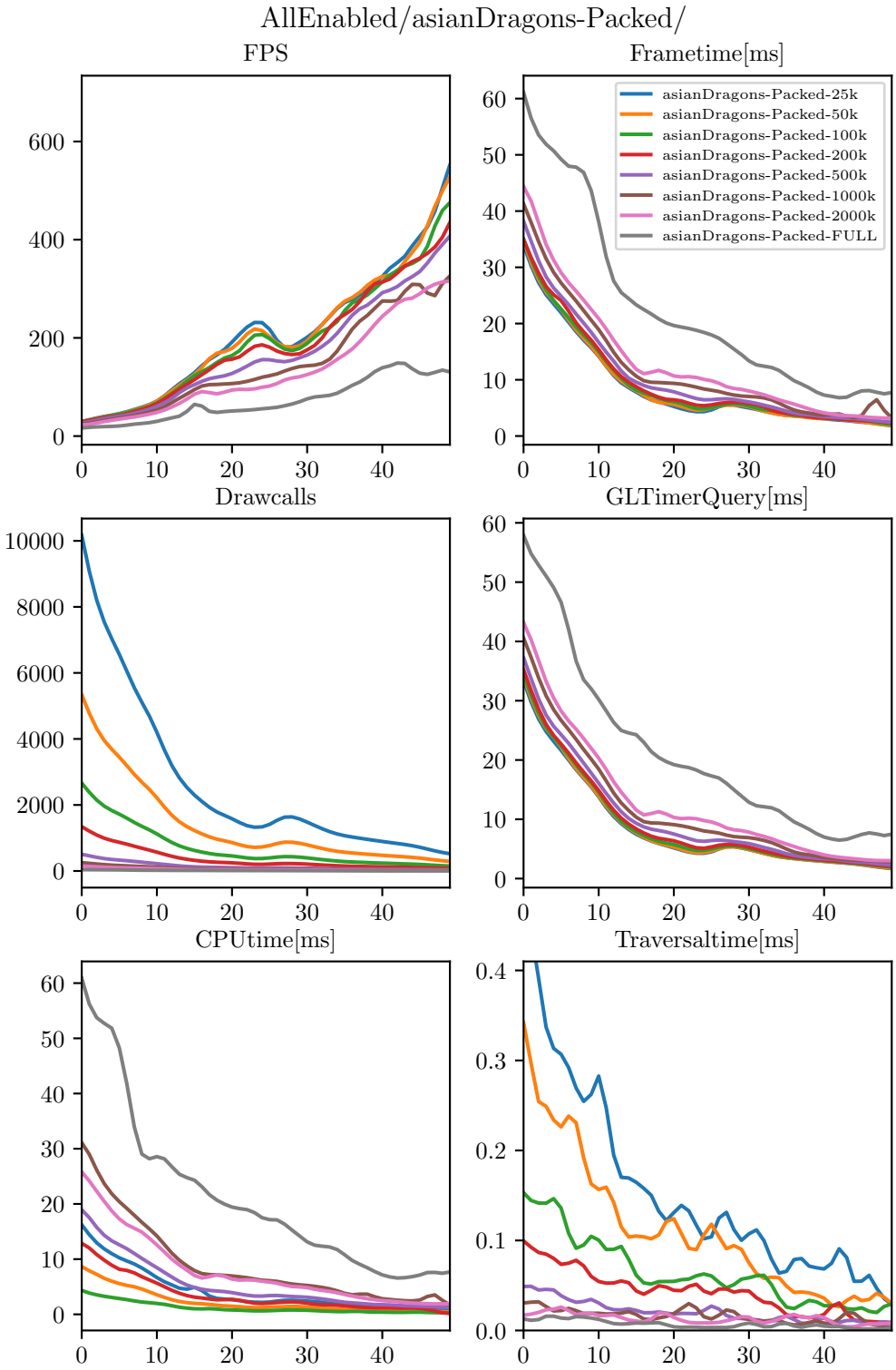
Obrázek 3: Měření na scéně *asianDragons-Split* s dodatečnými optimalizacemi s různě velkým množstvím trojúhelníků v listech. Osa x reprezentuje čas měření v sekundách.



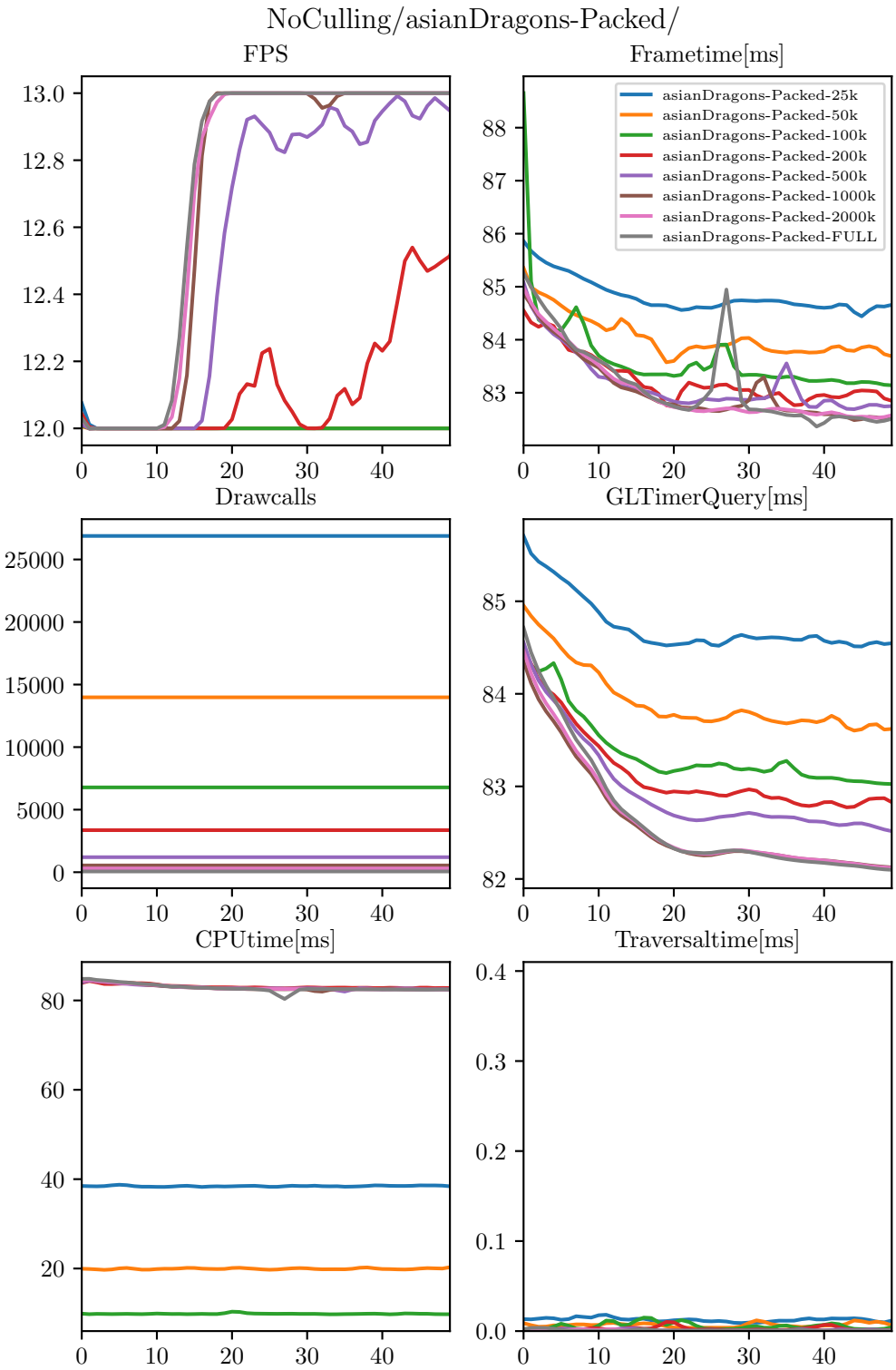
Obrázek 4: Měření na scéně *asianDragons-Split* s vypnutým ořezáváním pohledovým jehlanem. Osa x reprezentuje čas měření v sekundách.



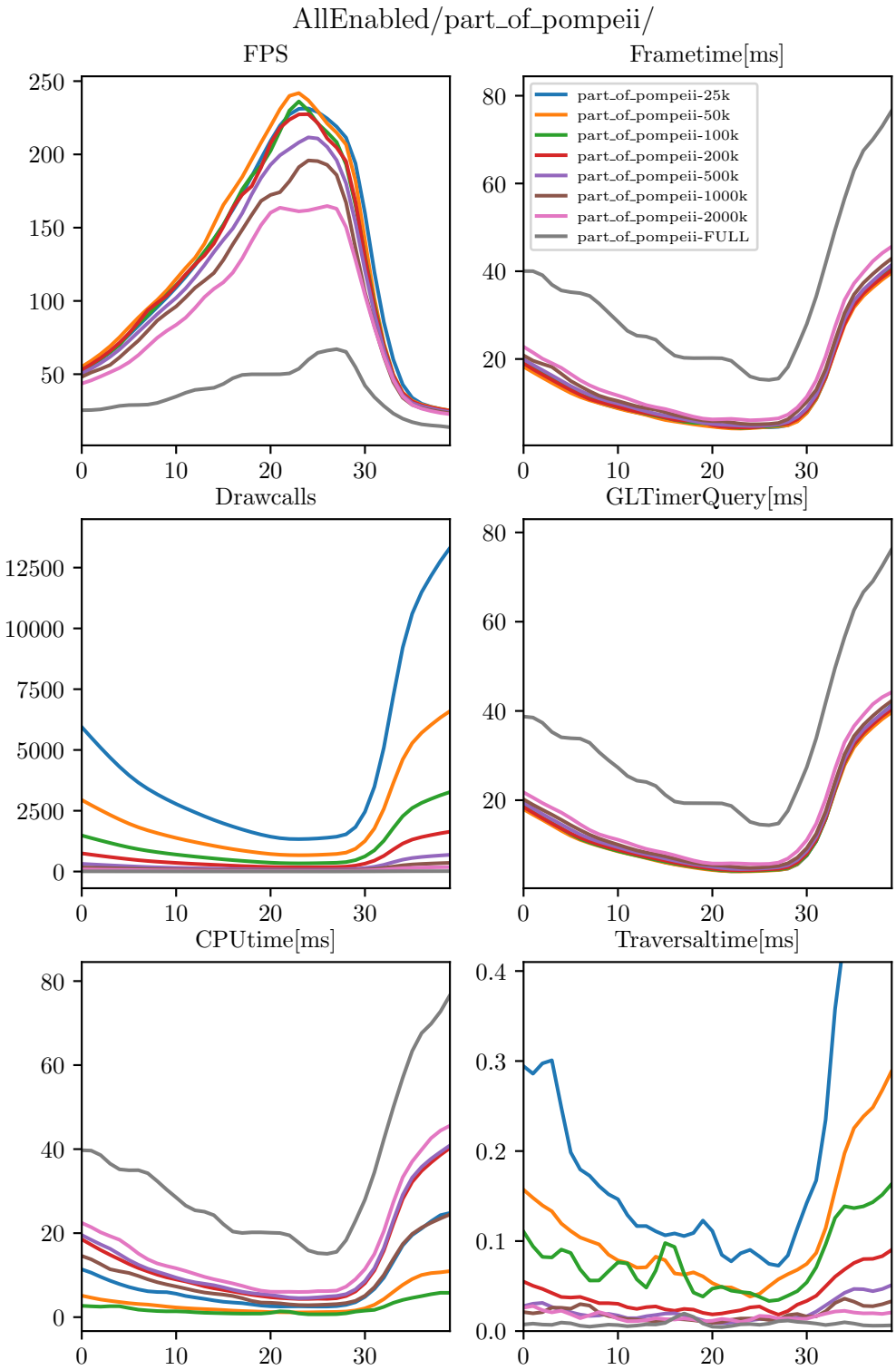
Obrázek 5: Měření na scéně *asianDragons-Packed* bez dodatečných optimalizací. Osa x reprezentuje čas měření v sekundách.



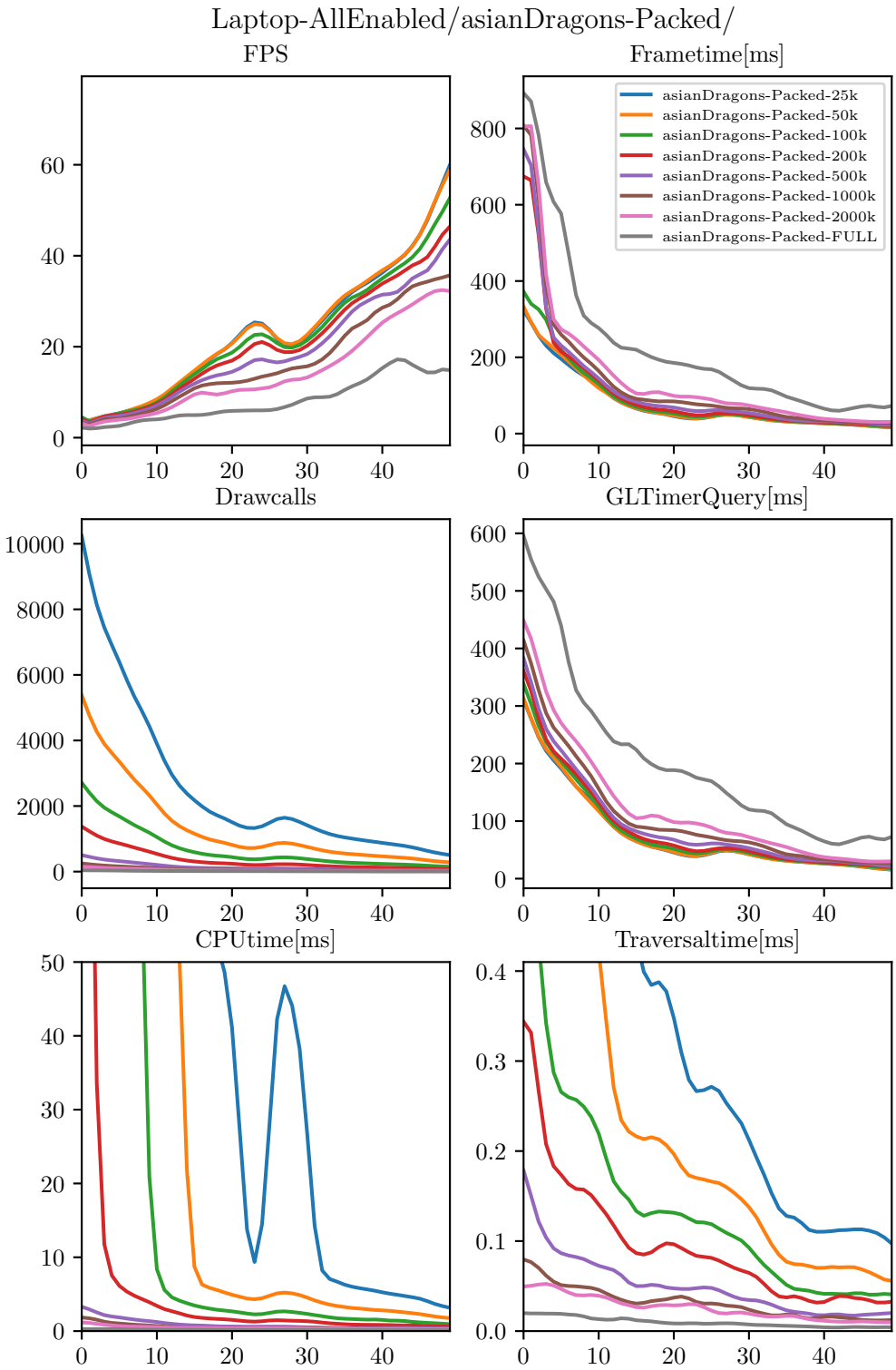
Obrázek 6: Měření na scéně *asianDragons-Packed* s dodatečnými optimalizacemi. Osa x reprezentuje čas měření v sekundách.



Obrázek 7: Měření na scéně *asianDragons-Split* s vypnutým ořezávání pohledovým jehlanem. Osa x reprezentuje čas měření v sekundách.



Obrázek 8: Měření na scéně *part_of_pompeii* s dodatečnými optimalizacemi s různě velkým množstvím trojúhelníků v listech. Osa x reprezentuje čas měření v sekundách.



Obrázek 9: Měření na scéně *asianDragons-Packed* na zařízení s integrovanou Intel GPU pod OS Linux. Osa x reprezentuje čas měření v sekundách.