UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Verkenning van het gebruik van reinforcement learning in de context van progressive glTF 2.0 streaming

Jeroen Weltens

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Maarten WIJNANTS

COPROMOTOR :
Prof. dr. Jori LIESENBORGS

2024
2025

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Verkenning van het gebruik van reinforcement learning in de context van progressive glTF 2.0 streaming

Jeroen Weltens

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Maarten WIJNANTS

COPROMOTOR :
Prof. dr. Jori LIESENBORGS

Universiteit Hasselt

Masterproef voorgedragen tot het behalen van de graad van master in de informatica

Verkenning van het gebruik van reinforcement learning in de context van progressive glTF 2.0 streaming

***Auteur*:**

Jeroen Weltens

***Promotor*:**

Prof. dr. Maarten Wijnants

***Co-promotor*:**

Prof. dr. Jori Liesenborgs

Academiejaar 2024-2025

Dankwoord

Het afgelopen jaar heb ik onderzoek gedaan naar reinforcement learning binnen de context van progressive streaming, samengevat in deze thesis. Dit onderwerp sloot nauw aan bij mijn persoonlijke interesse in artificiële intelligentie en computernetwerken, twee domeinen die beide raakvlakken vertonen met dit thema. Hoewel ik dit traject individueel heb afgelegd, kon ik steeds rekenen op de steun en begeleiding van anderen. Daarom wil ik graag de volgende personen en instanties bedanken.

Allereerst wil ik mijn promotor, Prof. Dr. Maarten Wijnants, en co-promotor, Prof. Dr. Jori Liesenborgs, bedanken. Hun gecombineerde expertise in progressive streaming van 3D-formaten, netwerken en communicatie, en machine learning en deep learning, vormde een ideale ondersteuning voor dit onderzoek. Ze maakten wekelijks tijd vrij voor overleg, waarbij ik vrijuit vragen kon stellen, ideeën kon bespreken en de voortgang van het onderzoek kon toelichten. De kritische blik en waardevolle feedback tijdens deze momenten hebben me geholpen om het onderzoek in de juiste richting te houden en om veel bij te leren over academisch onderzoek binnen deze vakgebieden.

Daarnaast wil ik de Universiteit Hasselt bedanken, en in het bijzonder het Expertisecentrum voor Digitale Media (EDM), voor het faciliteren van dit onderzoek. Deze thesis bouwt voort op de paper, Progressive Network Streaming of Textured Meshes in the Binary glTF 2.0 Format, [28], een onderzoek dat binnen EDM werd uitgevoerd, onder andere door Prof. Dr. Wijnants. Voor rekenintensieve taken, zoals het trainen van machine learning modellen, kon ik bovendien gebruik maken van het Vlaams Supercomputer Centrum via mijn UHasselt account. Dankzij deze infrastructuur kon ik veel meer experimenten uitvoeren en het onderzoek verder uitdiepen.

Tot slot wil ik ook mijn familie bedanken, in het bijzonder mijn ouders, voor de morele steun en motivatie tijdens dit traject.

Jeroen Weltens

Overpelt, 2 juni 2025

Samenvatting

In dit deel wordt kort de structuur van deze thesis uiteengezet. De eerste sectie, *Achtergrond en gerelateerd werk*, beschrijft de nodige achtergrondinformatie om deze thesis te begrijpen. Eerst worden de kernthema's van deze thesis behandeld, zoals reinforcement learning, het glTF-formaat en andere basisinformatie die doorheen de gehele thesistekst aan bod zal komen. Nadat alle kerninformatie besproken is, wordt er gekeken naar enkele implementaties die als bronmateriaal dienden voor het ontwikkelen van de eigen implementatie.

Hierna volgt het Hoofdstuk *Projectstructuur*, waarin de grote lijnen van het ontwikkelde project worden toegelicht, dat ontworpen werd om de RL-agent te trainen en te testen. In dit hoofdstuk wordt niet per se diep ingegaan op implementatiedetails, maar eerder op de verschillende modules die samen de codebase vormen die tijdens het project tot stand kwam, en welke functie elk van deze modules vervult. Hier zullen bijvoorbeeld de ontwikkelde custom glTF-environt en de RL-agent besproken worden, die samen met een renderer geïmplementeerd zijn om de gebruikerservaring te simuleren. Deze modules dienen gezamenlijk om de probleemstelling zo accuraat mogelijk te representeren en een omgeving te creëren waarin het inladen van glTF-nodes gesimuleerd kan worden en het effect ervan geëvalueerd. Op deze manier wordt een reinforcement learning-framework opgezet dat toelaat dat de agent renderingsimulaties uitvoert en leert op basis van de representatie van het probleem welke acties optimaal zijn, zijnde: het selecteren van de juiste nodes om in te laden.

Vervolgens wordt in het Hoofdstuk *Technische keuzes* besproken welke implementatiedetails aan het project zijn toegevoegd om de leerprestatie van de agent te verbeteren. Er wordt uiteengezet hoe de kwaliteit van de acties van de agent wordt beoordeeld, en welke data, al dan niet met preprocessingstappen, aan de agent moet worden aangeboden om optimaal te kunnen leren. Concreet wordt een rewardfunctie voorgesteld, die in meerdere fasen is opgebouwd, waarbij elke uitbreiding leidde tot merkbare prestatieverbeteringen. Ook werd een methode ontwikkeld om de 3D-data van de scène te representeren en te combineren met cameradata: alle bounding boxes in de scène werden van world space naar cameraspace geconverteerd alvorens ze aan de reinforcement learning-agent werden aangeboden. Dit zorgde ervoor dat de agent de correlatie tussen cameradata en positionele data niet zelf expliciet moest leren, wat het leerproces aanzienlijk vergemakkelijkte. Na deze bijdragen worden ook de problemen besproken die zich tijdens het ontwikkelen van het framework hebben voorgedaan, zoals de *permutation dependence* van het model, wat het moeilijk maakt om het uit te breiden naar ongeziene scènes. Er wordt een hypothese geformuleerd over hoe dit probleem in toekomstig werk zou kunnen worden aangepakt.

Ten slotte wordt een diepgaand onderzoek uitgevoerd naar het effect van *action masking* tijdens zowel de trainings- als inferentiefase van het model. Dit is een concept dat uitgebreid aan bod komt in deze thesis en dat geïntroduceerd werd om het probleem te verhelpen waarbij de agent niet leerde dat het meermaals selecteren van dezelfde actie binnen één episode, of concreet: een glTF-fragment twee keer inladen, niet toegestaan is. Deze *masking* zorgt ervoor dat zulke acties conditioneel uit de actieruimte van de agent verwijderd worden, waardoor de agent toch correct kan functioneren. Verder wordt in dit experiment een grondige analyse gepresenteerd van de prestaties van de ontwikkelde agent, vergeleken met twee benchmarks: de *greedy benchmark*,

die een optimale nodesequentie benadert, en de *heuristische benchmark* vanuit een eerder werk, die we met deze thesis trachten te overtreffen. De bekomen resultaten van het onderzoek worden hier vergeleken, en de best presterende oplossing wordt voorgesteld als antwoord op de onderzoeksvraag. Deze oplossing past tijdens de trainingsfase *action masking* toe op reeds ingeladen nodes, en tijdens de inferentiefase op zowel deze nodes als op nodes die zich buiten het view frustum van de camera bevinden.

De conclusie van deze thesis is dat we een veelbelovend proof-of-concept hebben aangereikt voor het gebruik van reinforcement learning in de context van progressive streaming, toegepast op glTF 2.0-data. Het model, getraind op de Sponza-scène, wist de heuristische benchmark marginaal te overtreffen op die scène, maar verdere ontwikkeling is noodzakelijk om het uit te breiden naar een algemeen model dat toepasbaar is op arbitraire scènes. De thesis wordt afgesloten met voorstellen voor toekomstig werk, die het model dichter bij zijn beoogde doel kunnen brengen.

Inhoudsopgave

1 Inleiding **7**

2 Achtergrond en gerelateerd werk **8**
2.1 Reinforcement learning . . . 8
2.1.1 Q-learning . . . 10
2.1.2 Deep Q-learning . . . 10
2.1.3 Proximal policy optimization . . . 11
2.1.4 Action masking - maskable PPO . . . 13
2.2 3D rendering en format . . . 14
2.2.1 3D rendering . . . 14
2.2.2 glTF . . . 14
2.3 Adaptieve bitrate streaming . . . 15
2.4 Beeldkwaliteitsmetrieken . . . 16
2.4.1 PSNR . . . 16
2.4.2 SSIM . . . 18
2.5 Bestaande oplossingen . . . 19
2.5.1 Occlusion culling . . . 19
2.5.2 Point clouds & Gaussian splatting . . . 20
2.6 Gerelateerd werk . . . 21
2.6.1 Progressive streaming . . . 21
2.6.2 Machine learning in streaming . . . 22
2.6.3 Reinforcement learning-implementaties . . . 23

3 Projectstructuur **25**
3.1 glTF Environment . . . 25
3.1.1 Implementatie . . . 26
3.1.2 Gebruik . . . 29
3.2 PPO-agent . . . 30
3.2.1 Custom Policy . . . 30
3.2.2 PerNodePolicy . . . 31
3.3 WebSocket-communicatie . . . 32
3.4 Renderer via Three.js . . . 33
3.4.1 WebSocket-protocol . . . 34
3.4.2 PSNR-berekening . . . 36
3.4.3 Frustum culling . . . 36
3.4.4 Convert bounding boxes . . . 37
3.4.5 testtools . . . 37
3.5 Dataverzameling en Preprocessing . . . 38
3.6 Puppeteer-module . . . 39
3.6.1 Use-case . . . 40
3.6.2 Implementatie . . . 40

4 Technische keuzes **42**
4.1 Selectie kwaliteitsmetriek . . . 42
4.2 Benchmarking . . . 43
4.2.1 Heuristische benchmark . . . 43
4.2.2 Greedy benchmark . . . 43
4.2.3 Implementatie . . . 44
4.3 Ontwikkelingen . . . 45
4.3.1 Initiële setup . . . 45
4.3.2 Rewardfunctie . . . 46
4.3.3 StartPSNR . . . 48
4.3.4 Camera space bounding boxes . . . 49
4.3.5 Observation space . . . 50
4.3.6 Action Masking . . . 51
4.4 Problemen . . . 51
4.4.1 Willekeurige nodepermutatie . . . 51
4.4.2 Uitbreidbaarheid naar ongeziene scènes . . . 53

5 Action masking: experimenten en resultaten **55**
5.1 Experimentele opstelling . . . 56
5.1.1 glTF-environment . . . 56
5.1.2 Agent . . . 57
5.1.3 Structuur van het experiment . . . 57
5.2 Experiment . . . 58
5.2.1 Model zonder action masking . . . 58
5.2.2 Model met action masking op reeds ingeladen nodes . . . 58
5.2.3 Model met action masking op ingeladen nodes en FOV . . . 60
5.2.4 Vergelijkende analyse . . . 60
5.3 Discussie . . . 62
5.3.1 Gebruik van action masking tijdens inference . . . 62
5.3.2 Gebruik van action masking tijdens training . . . 63
5.3.3 Prestatie versus greedy- en heuristische methode . . . 63
5.4 Finaal model . . . 64

6 Conclusies **69**
6.1 Zelfreflectie . . . 70
6.2 future work . . . 72
6.3 Beantwoording van de onderzoeksvraag . . . 73

Hoofdstuk 1

Inleiding

Met de opkomst van toepassingen zoals virtual reality en augmented reality is de vraag naar on-demand streaming van 3D-data de afgelopen jaren sterk toegenomen. De gebruiker bevindt zich hierbij vaak in een virtuele scène die bestaat uit vele assets of afzonderlijke 3D-structuren, die samen het geheel van de scène vormen. Deze scènes zijn doorgaans te omvangrijk om in één keer via het web van server naar client verzonden te worden. Daarom worden ze meestal opgedeeld in kleinere fragmenten, die afzonderlijk verstuurd en aan de clientzijde opnieuw samengevoegd worden tot een volledige scène.

Het is echter niet wenselijk dat de gebruiker moet wachten tot alle fragmenten zijn aangekomen alvorens de scène weergegeven kan worden. Om deze bufferperiode te vermijden, kan een progressief streaming- en renderingparadigma worden toegepast: bij het ontvangen van een deel van de fragmenten kan reeds een gedeelte van de scène opgebouwd en weergegeven worden aan de gebruiker. Indien de fragmenten strategisch gekozen worden op basis van de positie en oriëntatie van de gebruiker in de scène, kunnen deze tussenresultaten reeds een gedeeltelijk tot volledig beeld geven van de structuur en de details in de nabijheid van de camera en diens kijkrichting.

In eerdere implementaties werd deze strategie hoofdzakelijk bepaald door het rechtstreeks toepassen van heuristieken die de Quality of Experience (QoE) van de gebruiker optimaliseren. Voorbeelden van dergelijke heuristieken zijn de positie ten opzichte van het gezichtsveld (field of view, FOV) van de camera, de afstand tot de camera en de grootte van het fragment. Complexere benaderingen maken gebruik van technieken zoals optical flow bij bewegende camera's of occlusion culling om verborgen fragmenten buiten beschouwing te laten. Op basis van zulke heuristieken wordt een model gevormd dat per fragment analyseert welke volgorde of resolutie (in het geval van adaptieve streaming) optimaal is om de gebruikerservaring tijdens het streamen te verbeteren. De ideale laadvolgorde hangt vaak af van het type toepassing (bijvoorbeeld interactief versus visueel) en het soort scène (bijvoorbeeld open versus gesloten), waardoor het moeilijk is om een universeel optimaal heuristisch model te definiëren.

In deze thesis wordt onderzocht of machine learning een rol kan spelen in de context van progressive streaming van 3D-data. In het bijzonder het bepalen van welke fragmenten prioritair moeten worden ingeladen. De heuristieken die in eerdere implementaties direct gebruikt werden om de laadvolgorde te bepalen, kunnen mogelijk dienen als input voor een intelligente agent die leert welke combinatie van heuristieken en structurele scènekenmerken optimaal is. Concreet ontwikkelen we een reinforcement learning-model dat, op basis van de kenmerken van een scène, een optimale volgorde probeert te bepalen waarin de fragmenten via het web verzonden worden. Optimaal wordt in de context van deze onderzoeksvraag gedefinieerd als: iteratief een zo compleet mogelijk beeld construeren, waarbij met zo weinig mogelijk fragmenten telkens een zo representatief mogelijk zicht voor de gebruiker opgebouwd wordt.

Hoofdstuk 2

Achtergrond en gerelateerd werk

In dit hoofdstuk wordt de nodige voorkennis en context rond het onderwerp van de thesis besproken om later in de verdere hoofdstukken dieper op details van de implementatie in te kunnen gaan. We beginnen met het beschrijven van de core-concepten die dit onderzoek omvat en later bekijken we in gerelateerd werk enkele implementaties die enkele van deze concepten samenbrengen en hoe we uit deze implementaties inspiratie haalden voor onze eigen oplossing. Ook wordt er steeds een terugkoppeling gemaakt naar de raakvlakken tussen de besproken concepten/werken en onze eigen probleemstelling.

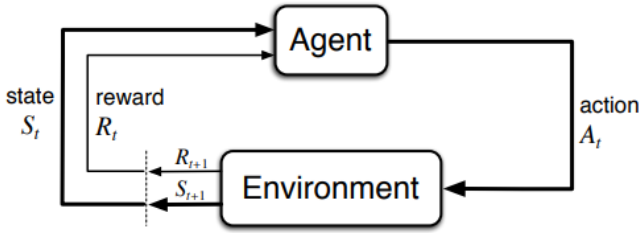
2.1 Reinforcement learning

Reinforcement learning (RL) is een vorm van machine learning waarbij een agent leert om optimale beslissingen te nemen door middel van interacties met een omgeving (environment). In tegenstelling tot traditionele supervised learning, waarbij het model getraind wordt op een vooraf gedefinieerde en gelabelde dataset, leert een RL-agent door middel van trial-and-error. Het doel is om een policy te ontwikkelen die acties kiest die op lange termijn maximale beloning (reward) oplevert.

Een agent observeert de toestand (state) van de omgeving en kiest op basis van zijn huidige policy een actie. Deze actie wordt vervolgens uitgevoerd in de omgeving, die daarop reageert door een nieuwe observatie en een bijbehorende beloning terug te geven. De agent gebruikt deze informatie om zijn policy bij te stellen en zijn toekomstige beslissingen te verbeteren. Dit leerproces wordt vaak geformaliseerd als een Markov Decision Process (MDP) (zie Figuur 2.1), waarbij het systeem zich ontwikkelt volgens probabilistische regels die alleen afhangen van de huidige toestand en gekozen actie. Voor diepgaandere uitleg hierover refereren we naar Hoofdstuk 3 van [43].

Reinforcement learning bevat ook specifieke terminologie die doorheen deze thesis gebruikt wordt. We zagen al dat het MDP bestond uit een agent die interageert met zijn environment via acties om zijn volgende state te bereiken. De agent stemt een policy af om de meest optimale actie vanuit een state te bepalen. Deze agent met zijn policy vormen in praktijk het model van het reinforcement learning framework dat getraind wordt door het verkennen van de environment. Iedere keer dat de agent een actie neemt noemen we dit een timestep. Wanneer dus een aantal timesteps vernoemd wordt, beduidt dit het aantal genomen acties tot op dat punt. Wanneer de agent een eindstate bereikt, is dit het einde van een episode. Episodes bestaan dus uit een aantal genoment timesteps (of acties) tot een eindstate bereikt is.

De environment is verantwoordelijk voor het verwerken van de actie via een `step`-functie, die



Figuur 2.1: Visualisatie van een Markov Decision Process die uitbeeldt hoe een agent interrageert met de environment. Overgenomen uit [43] chapter 3: Finite Markov Decision Processes

het nemen van een timestep behandelt. Deze functie neemt een actie als input, simuleert het effect ervan op de omgeving en retourneert:

- een nieuwe observatie van de state van de environment,
- een reward die de kwaliteit van de actie weerspiegelt,
- een boolean die aangeeft of een eindstate bereikt is, en dus de episode beëindigd is (bijvoorbeeld bij het bereiken van een doel of een game-over),
- en eventueel extra diagnostische informatie.

Een state stelt de parametrische waarden van de environment voor op een bepaald moment. Een observatie zijn de waarden geleverd aan de agent (of input vector) om via zijn policy (getraind model) de juiste actie uit de mogelijke acties (output vector) te kiezen. Zowel de mogelijke acties die een agent kan nemen als de observaties die hij ontvangt, worden gespecificeerd via een zogeheten *action space* en *observation space*. Deze ruimtes kunnen discreet of continu zijn, afhankelijk van de aard van de omgeving en het probleem. Een discrete action space betekent bijvoorbeeld dat er slechts een eindig aantal mogelijke acties zijn (bijvoorbeeld indrukken van een van de mogelijke toetsen van een interface), terwijl een continue action space oneindig veel mogelijke actieconfiguraties toelaat (bijvoorbeeld het aansturen van een robotarm via een bepaalde hoek).

De complexiteit in state-of-the-art reinforcement learning ligt vooral in de manier waarop optimale policies benaderd worden. Er bestaat een grote variëteit aan beschikbare algoritmen die geschikt zijn voor verschillende omstandigheden. Zo zijn er algoritmen die beter presteren bij taken met lage of hoge complexiteit, en die specifiek ontworpen zijn voor discrete of continue action- en observation spaces. De policy die uiteindelijk de hoogste cumulatieve reward oplevert, is het doel dat nagestreefd wordt. De manier waarop dit bereikt wordt, hangt sterk af van de gebruikte techniek en de onderliggende aannames van het algoritme.

Daarnaast ligt de moeilijkheid vaak in het balanceren van *exploratie* en *exploitatie*. Hiermee bedoelen we dat een agent tijdens de training kans moet hebben (misschien suboptimale) acties te verkennen, maar de policy mag niet te ver afwijken van de vorige policy om zijn voortgang te behouden. Ook het stabiliseren van het leerproces is tijdens de training een obstakel. Een training kan gedestabiliseerd geraken door uitschieters, lokale maxima, en andere randgevallen. Recente ontwikkelingen zoals Proximal Policy Optimization (PPO) [38], Soft Actor-Critic (SAC) [15] en Maskable DQN [16] bieden verbeteringen op het vlak van sample-efficiëntie, stabiliteit en veiligheid van het geleerde gedrag. De keuze van het algoritme wordt daarom sterk bepaald door het type omgeving, de aard van het probleem en de structurele eigenschappen van de observaties en acties.

Via deze algoritmes kan de agent zijn policy gradueel verbeteren om zijn cumulatieve beloning te optimaliseren. Wanneer deze beloning op een goede manier gedefinieerd is, kan de agent op deze manier door te interageren met zijn omgeving het gewenste gedrag aanleren. Dit maakt reinforcement learning bijzonder geschikt voor problemen waarbij beslissingen in een

sequentiële context genomen moeten worden, zoals robotica, spelstrategieën of adaptieve streaming van multimediacontent. Zeker omdat de handelingen vaak gesimuleerd kunnen worden en niet daadwerkelijk uitgevoerd (bijvoorbeeld robotarm kan in een digitale simulatie getraind worden). Voor uitgebreidere informatie over reinforcement learning wordt verwezen naar [43] of een beknoptere paper: [13].

We gaan hier dieper in op een paar voorbeelden van deze algoritmen en waar ze gebruikt kunnen worden.

2.1.1 Q-learning

Q-learning [48] is een value-based reinforcement learning techniek waarbij een tabel met Q-waarden, $Q(s, a)$, wordt opgebouwd. Elke Q-waarde stelt de verwachte cumulatieve beloning voor die een agent ontvangt wanneer hij in toestand s actie a kiest, en nadien de optimale strategie volgt. De actie met de hoogste Q-waarde wordt gekozen, behalve in het geval van exploratie: met een kans ϵ wordt een willekeurige actie geselecteerd (de zogenaamde ϵ-greedy policy).

De Q-waarden worden iteratief geüpdatet aan de hand van de Bellman-updateformule (zie [48] voor details). Deze methode werkt goed voor problemen met een beperkte en discrete state- en action space (zoals in grid worlds, binaire environments waarin agents kunnen bewegen die typisch als introductie in reinforcement learning gebruikt worden, zie [43], voorbeeld 3.8). Naarmate deze ruimtes groter worden, wordt het opslaan en updaten van de tabel onpraktisch.

2.1.2 Deep Q-learning

Deep Q-learning (DQN) [16] is een uitbreiding op Q-learning waarbij een neuraal netwerk de Q-functie $Q(s, a)$ approximeert in plaats van het opstellen van een tabel. Dit maakt het mogelijk om problemen met grote of continue state spaces aan te pakken. Het netwerk ontvangt de huidige toestand s als input en geeft als output de Q-waarden $Q(s, a_i)$ voor alle mogelijke acties.

Om de training stabieler te maken, wordt gebruikgemaakt van twee belangrijke technieken: de replay buffer en het target network.

- **replay buffer**:
  Slaat ervaringen op in de vorm van tuples (s, a, r, s'), waarbij s de oorspronkelijke toestand is, a de uitgevoerde actie, r de bijhorende beloning en s' de resulterende toestand. Tijdens training worden willekeurige batches uit deze buffer gesampled, wat zorgt voor decorrelatie tussen opeenvolgende voorbeelden en dus stabielere updates. kortom is dit het onderbreken van de learning cyclus door het opnieuw testen van een vorige state en kijken of deze nogsteeds een gewenste actie levert.

- **target network**:
  Een periodiek geüpdatete kopie van het Q-netwerk, en wordt gebruikt voor het berekenen van stabiele targetwaarden bij de Bellman-update. Aangezien het hoofdnetwerk continu verandert tijdens training, helpt een langzaam geüpdatet target network bij het stabiliseren van de leerdoelen. Simpelweg houden we een kopie van het netwerk bij dat slechts periodisch geupdate wordt om berekeningen mee te doen om stabiliteit te bekomen.

Zowel Q-learning als Deep Q-learning zijn value-based methoden: zij leren een valuefunctie die een waarde toekent aan alle mogelijke acties in een toestand, zonder een expliciete policy te modelleren. Deze valuefunctie wordt vervolgens gebruikt om in een nieuwe toestand opnieuw een actie te selecteren.

2.1.3 Proximal policy optimization

In ons project maakten we gebruik van *Proximal Policy Optimization* (PPO) [38] als het trainingsalgoritme voor onze reinforcement learning-agent. PPO is een veelgebruikt policy gradient-algoritme dat ontworpen is om de training van neurale netwerken in reinforcement learning stabieler en efficiënter te maken. Het algoritme optimaliseert de policy van een agent op basis van de beloningen die deze ontvangt tijdens het uitvoeren van meerdere episodes in zijn omgeving. Het is dus geen value based methode, maar een policy-based methode. Een policy-based methode geeft aan iedere actie een kans dat deze genomen wordt vanuit de state, wat intrensiek zorgt voor exploratie.

Het centrale idee van PPO is om de parameters van de policy op een gecontroleerde manier bij te werken door de geschatte *policy gradient* te gebruiken. De gradient of afgeleide van de policy in functie van de cumulatieve reward dient om de policy de juiste richting (van hogere rewards) in te duwen. Deze gradient wordt bepaald op basis van het verschil in verwachte beloning tussen de huidige policy en de vorige, zodat acties die tot hogere beloningen leiden vaker gekozen worden. Om ervoor te zorgen dat de policy tijdens training niet te sterk verandert (wat kan leiden tot instabiliteit), introduceert PPO een regularisatieprincipe gebaseerd op het idee van een "trust region".

PPO is ontwikkeld als een eenvoudiger en efficiënter alternatief voor het complexere *Trust Region Policy Optimization* (TRPO)-algoritme [37]. Dit algoritme houdt policy binnen een trust-region: een afbakening van hoe ver de volgende policy mag verschillen van de huidige. TRPO beperkt drastisch de afstand tussen opeenvolgende policies door een harde limiet op de *Kullback-Leibler (KL) divergence* tussen de oude en nieuwe policy op te leggen. Hoewel effectief, vereist TRPO het oplossen van een ingewikkeld optimalisatieprobleem met tweede-orde methodes, wat computationeel duur is. Voor verdere informatie over dit algoritme wordt verwezen naar [37].

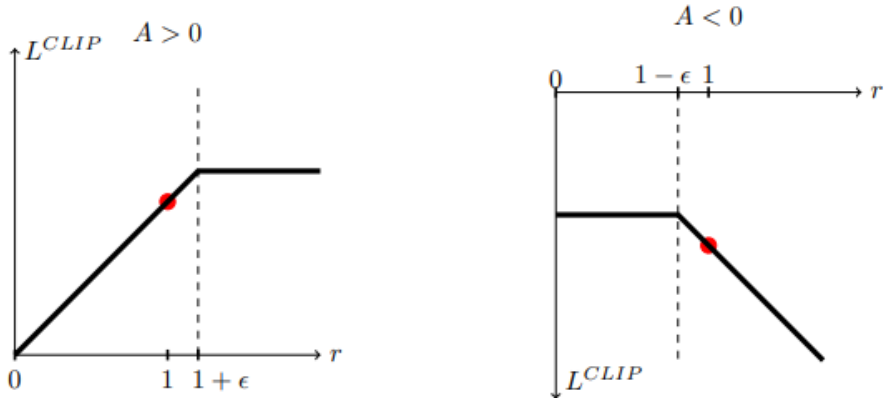
PPO behoudt hetzelfde idee van beperking van policy veranderingen, maar doet dit op een eenvoudigere manier door gebruik te maken van zogeheten *clipping*. Hierbij wordt de ratio tussen de nieuwe en oude policy likelihoods binnen een bepaalde marge gehouden. Dit voorkomt dat de policy te ver afwijkt van eerdere versies, zelfs als de reward-verbetering initieel groot lijkt. De objective function van PPO bevat dus een gemodificeerde term die beleidsupdates afstraft zodra ze te groot worden:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{2.1}$$

waarbij $\mathbb{E}_t$ de verwachtingswaarde over tijdstappen t voorstelt, typisch genomen over een batch van trajectories verzameld met de huidige policy. $r_t(\theta)$ is ratio is tussen de nieuwe en oude policy-probabilities, $\hat{A}_t$ de geschatte advantage, en ϵ een hyperparameter die bepaalt hoeveel de policy maximaal mag afwijken van de vorige versie. Stel dat $\epsilon = 0.2$: wanneer $r_t(\theta) = 1$ zal de policy onveranderd zijn. Wanneer $r_t(\theta) > 1$ wordt de betreffende actie meer gekozen dan in de vorige policy en vice-versa. Wanneer we streven naar een optimale policy zal deze policy in een stap fel kunnen veranderen, maar dankzij clipping zal $r_t(\theta)$ in dit geval enkel waarden aan kunnen nemen tussen [0.8, 1.2] (zie grafisch: fig 2.2), wat de maximale verandering van de policy limiteert, om zo stabiliteit aan het proces te voegen.

Hoewel PPO (Clip) en TRPO hetzelfde doel hebben: stabiele updates van de policy, onderscheiden ze zich in aanpak. TRPO gebruikt een complexe tweede-orde optimalisatie met een harde KL-constraint, terwijl PPO dit vervangt door een eenvoudiger clippingmechanisme dat grote policy-updates beperkt zonder dure berekeningen. Hierdoor behoudt PPO de stabiliteit van TRPO, maar is het veel eenvoudiger te implementeren, efficiënter in training en beter schaalbaar, wat het in de praktijk de voorkeur geeft.

Verder bestaat PPO niet enkel uit een value network zoals DQN, maar maakt het gebruik van



Figuur 2.2: Grafische voorstelling van het clipping algoritme, overgenomen uit [38]: links de visualisatie van de positieve clip wanneer een actie meer aangemoedigd zal worden in de nieuwe policy, rechts vice-versa.

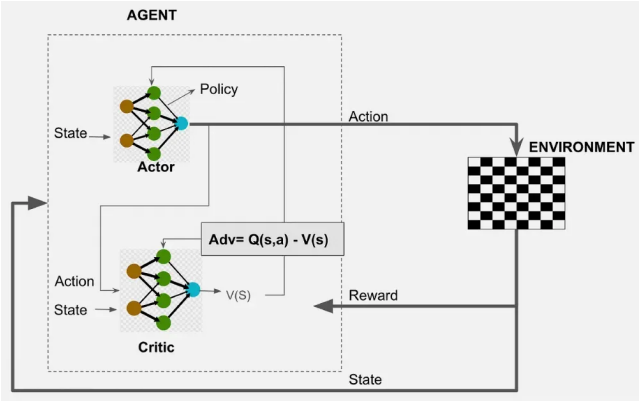
een **actor-critic architectuur**, waarbij twee neurale netwerken samenwerken. Het idee achter deze architectuur is om de voordelen van zowel policy-based (actor) als value-based (critic) methodes te combineren, en elkaars beperkingen aan te vullen. Een schematische visualisatie van dit systeem is terug te vinden bij fig 2.3. We vatten deze componenten kort samen op basis van [26].

- **Actor network**:
  Dit netwerk modelleert de policy $\pi_\theta(a|s)$ — het bepaalt hoe waarschijnlijk het is dat een actie a wordt gekozen in een bepaalde toestand s. De actor past zijn policy aan op basis van feedback uit de omgeving, met als doel het maximaliseren van de verwachte cumulatieve beloning. Deze feedback gebeurt via policy gradients, vaak gewogen door de *advantage* $\hat{A}_t$, die aangeeft hoe goed een actie presteerde t.o.v. het verwachte gemiddelde. Hoewel een actor-only netwerk direct kan leren vanuit gesimuleerde ervaringen, kan het instabiel of inefficiënt leren als het geen accurate feedback krijgt over hoe goed een actie werkelijk was.

- **Critic network**:
  Het critic-netwerk probeert een *valuefunctie* $V^\pi(s)$ of een gerelateerde functie (zoals de action-value $Q^\pi(s,a)$) te leren. Deze waarde voorspelt de verwachte toekomstige beloningen van een toestand, gegeven de huidige policy. In PPO wordt deze waarde onder andere gebruikt om het *advantage-schatting* $\hat{A}_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ te berekenen. De critic ondersteunt dus de actor door betrouwbaardere beoordelingen te geven over de kwaliteit van genomen acties, waardoor het leerproces stabieler en efficiënter verloopt. In tegenstelling tot DQN, waar er values voor iedere mogelijke actie berekend worden, zal in het geval van PPO gegeven de state en genomen actie een value voor enkel deze genomen actie berekend worden. Deze value wordt dan gebruikt om het actor network te beoordelen.

Door deze samenwerking profiteert PPO van de directe optimalisatie van het gedrag via de actor, terwijl het critic-netwerk fungeert als een soort coach die continu evalueert of het gedrag effectief is. Deze synergie zorgt voor snellere convergentie en minder variatie in de policy updates dan actor-only of critic-only benaderingen.

Door deze aanpak combineert PPO de voordelen van stabiliteit (zoals bij TRPO) met de eenvoud van first-order optimalisatie, waardoor het breed inzetbaar is in complexe omgevingen met hoge-dimensionale observaties en actie-ruimtes. Binnen ons project bood PPO een robuust kader om de agent gradueel te leren welke acties in bepaalde situaties het meest waardevol zijn. Voor meer gedetailleerde uitleg over deze policy gradient method verwijzen we naar [38].



Figuur 2.3: Een visualisatie van het actor-critic paradigma binnen het kader van Reinforcement learning, afbeelding overgenomen van [12].

2.1.4 Action masking - maskable PPO

Voor vele RL toepassingen zijn acties in bepaalde states ongeldig (bijvoorbeeld het plaatsen van een schaakstuk buiten het bord, over eigen stukken heen, ...). De klassieke manier om deze acties af te leren aan een RL agent is om een negatieve belonging of penalty te geven wanneer zulke acties genomen worden. In applicaties met grote discrete action spaces (bijvoorbeeld schaken) is het leren van ongeldigheid van acties op deze manier echter niet zo simpel om aan te leren: er zijn zeer veel verschillende mogelijke states en zeer veel mogelijke acties, wat het moeilijk maakt voor de agent deze correlatie goed te leggen voor het pas echt kan beginnen leren.

Om dit probleem te omzeilen is het principe van Action Masking [18] naar de voorgrond getreden bij policy gradient methods zoals PPO. Het neuraal netwerk dat de policy van de agent voorstelt zal de observatie als inputvector nemen en zogenaamde action logits bekomen na het doorlopen van het neuraal netwerk. Dit zijn ongenormaliseerde values die aan iedere action toegekend worden, waarop samen een softmax functie op toegepast wordt om deze waarden om te zetten in een reeks probabiliteiten die de kans voorstellen dat iedere actie respectievelijk genomen wordt vanuit deze state. Wat de action masking in dit geval doet is het toepassen van een binair geldigheidsmask op deze logits: logits die overeen komen met een 0-bit in dit mask zullen hun score gereduceerd krijgen tot een zeer grote negatieve waarde, waardoor de uiteindelijke probability bekomen door de softmax praktisch 0 zal zijn. Masking wordt toegepast voor de softmaxberekening zodanig dat de probabiliteiten enkel over de geldige acties berekend wordt.

De resulterende gemaskeerde policy levert vervolgens geldige policy gradients op, omdat het maskeren kan worden geïnterpreteerd als een differentieerbare, state-afhankelijke transformatie op de logits. Er wordt dus bewezen dat de verkregen gradients nog steeds overeenkomen met de voorwaarden van het policy gradient theorem van Sutton et al. (2000) [44]. Met andere woorden: action masking verandert de vorm van de policy, maar niet de geldigheid van de gradient-updates. Voor gedetailleerdere beschrijving en een concreet voorbeeld, zie [18].

Uit de resultaten van dit onderzoek bleek dat action masking beter schaleert naar grotere action spaces, terwijl negatieve beloningen bij een invalid actie slecht schaleert. Er blijkt ook dat via action masking een agent sneller convergeert naar een versie die goede performantie realiseert: het model vindt hier sneller een werkende policy en heeft meer tijd deze te optimaliseren.

Action masking kan in dit onderzoek ook zeer interessant zijn: met de omvangrijke action space van de in te laden meshes kan het lastig zijn een agent aan te leren deze meshes niet dubbel in te laden. Via action masking zouden reeds ingeladen fragmenten invalid beschouwd kunnen worden zonder dat hier extra training voor nodig is.

2.2 3D rendering en format

2.2.1 3D rendering

Om de verwerking en weergave van 3D-data te kaderen, bespreken we in deze sectie kort de fundamentele principes van 3D rendering. Het doel van een graphics renderer is het omzetten van een representatie van een driedimensionale scène naar een tweedimensionaal beeld, op basis van een virtuele camera die de scène observeert. Deze transformatie gebeurt via een reeks stappen die samen de rendering pipeline vormen. Een cruciale stap hierin is *rasterisatie*, waarbij de continue wereld van 3D-geometrie wordt omgezet naar een discrete verzameling pixels die het uiteindelijke beeld vormen. Tijdens dit proces worden de positie van vertices, de toegepaste materialen en textures, en de verlichting in acht genomen om het uiteindelijke resultaat te bepalen. Een uitgebreide introductie tot deze pipeline wordt gegeven in de OpenGL Rendering Pipeline Overview [23], dat het fundament vormt van de meeste moderne graphics rendering engines. Voor het begrijpen van deze thesis is een diepgaande kennis van grafische rendering niet vereist. De rendering van glTF-data wordt namelijk geabstraheerd via een bestaande grafische library, met name Three.js [5].

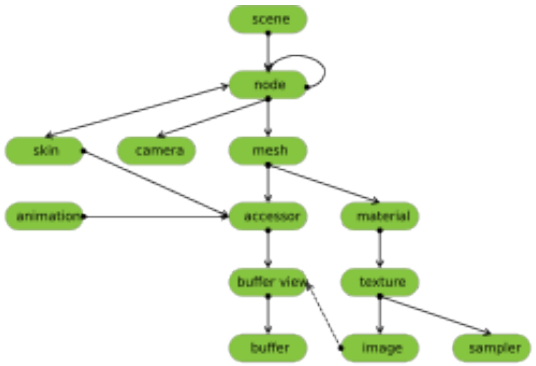
Graphics rendering toepassingen kunnen onderverdeeld worden in twee categorieën: *offline rendering* en *online rendering*. Offline rendering wordt typisch gebruikt in film en animatie, waarbij de rendering slechts één keer hoeft te gebeuren en het gegenereerde beeld het eindproduct vormt. Online rendering daarentegen, zoals toegepast in games en interactieve toepassingen, vereist continue herberekening van het beeld op basis van gebruikersinteractie of camerabewegingen. Deze thesis situeert zich binnen het domein van online rendering. Het doel is om het renderingproces zodanig te optimaliseren dat enkel de meest relevante fragmenten van een scène dynamisch naar de gebruiker worden doorgestuurd. Zo kan met onvolledige 3D-data toch een bruikbaar beeld van de scène gegenereerd worden.

2.2.2 glTF

In deze thesis werd gebruikgemaakt van 3D-scènes in het glTF-formaat (GL Transmission Format). glTF is een open standaard ontwikkeld door de Khronos Group, met als doel efficiënte uitwisseling en weergave van 3D-modellen over netwerken mogelijk te maken. Het formaat is geoptimaliseerd voor snelle laadtijden en minimale bestandsgrootte, en wordt vaak omschreven als het "JPEG van 3D".

glTF maakt gebruik van een hiërarchische datastructuur waarin 3D-objecten worden voorgesteld als knopen (nodes) in een scènegraaf. Elke node kan eigenschappen bevatten zoals een transformatie (positie, rotatie, schaal), referenties naar geometrie (meshes) en materiaalinformatie. Een glTF-node kan ook eventueel kinderen bevatten: subnodes die hiërarchisch geordend zijn onder de huidige node. Hierdoor kunnen complexe structuren zoals rigs (bottensysteem gebruikt voor fysisch animeren van 3D-modellen) of samengestelde objecten eenvoudig worden gemodelleerd. In deze thesis verwijst de term *node* naar een glTF-knoop die een 3D-object representeert, inclusief geometrische data, textuurinformatie, positie en andere relevante eigenschappen. In onze implementatie werd steeds uitgegaan van een "platte"nodestructuur: dit wil zeggen dat knopen zelf geen kinderen bevatten. Dit zou eventueel wel uit te breiden zijn door onze methode recursief toe te passen. Zo zou op een node met kinderen die gekozen wordt, de selectie intern tussen zijn kinderen toegepast kunnen worden om de volgorde van deze knopen te bepalen.

De daadwerkelijke geometrische data van de node, zoals vertexposities, normaalvectoren en textuurcoördinaten, worden opgeslagen in externe of ingesloten buffers. Elke buffer bevat ruwe binaire data, en via zogenaamde bufferViews en accessors wordt bepaald welk deel van deze data bij een specifieke node hoort. Een node in de graaf bevat dus indirecte verwijzingen naar de juiste byteranges in de bufferbestanden, waarmee de relevante data geëxtraheerd kan worden voor rendering of verwerking. Deze scheiding tussen structuur (de scènegraaf) en ruwe data (in buffers) maakt het glTF-formaat bijzonder geschikt voor toepassingen waarbij scènes



Figuur 2.4: Een schematische voorstelling van hoe een glTF-node in een scène gestructureerd is en welke componenten deze kan bevatten. Figuur overgenomen uit de glTF 2.0-specificatie van de Khronos Group [22].

incrementeel of selectief geladen moeten worden, zoals bij streaming van grote omgevingen of gebruik in resourcebeperkte omgevingen zoals mobiele toestellen en webapplicaties. Een schematisch overzicht van de eventuele structuur van een glTF-node die een asset in de scène beschrijft, is terug te vinden in Figuur 2.4.

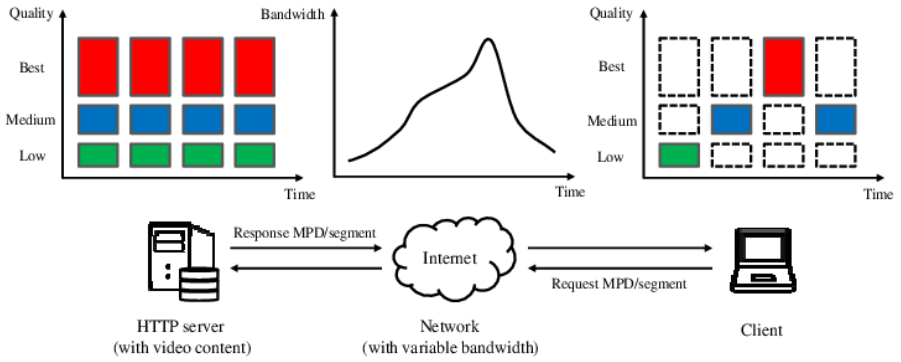
glTF 2.0 is een verder geoptimaliseerde versie van het glTF 1-formaat, dat voor ons mentaal model in deze thesis op dezelfde manier functioneert als de beschrijving hierboven. Deze optimalisaties zijn in de context van deze thesis te diepgaand. Voor een diepgaandere beschrijving van dit framework verwijzen we naar de pagina van de Khronos Group die dit formaat in detail beschrijft [22].

In deze thesis maken we voor onze glTF-scène geen gebruik van complexe textures of belichtingen. Er werd in de experimenten steeds gebruikgemaakt van de `Sponza`-scène. Dit is een veelgebruikte testscène die 142 nodes bevat in het glTF-formaat.

2.3 Adaptieve bitrate streaming

In de beginfase van mediastreaming werd content doorgaans aangeboden in één vast formaat en resolutie. Dit leidde vaak tot problemen wanneer de beschikbare bandbreedte tussen de client en de server onvoldoende was om de stream vloeiend af te spelen. Het gevolg hiervan was buffering. Dit is het tijdelijk onderbreken van de weergave om extra data te laden. Deze onderbrekingen zorgde voor aanzienlijke verslechtering van de user experience tijdens het consumeren van multimedia-content. Een voor de hand liggende oplossing is om de resolutie van de mediastream aan te passen aan de beschikbare bandbreedte van de gebruiker. Echter is de beschikbare netwerkbandbreedte in praktijk sterk variabel en afhankelijk van factoren zoals netwerkcongestie, het aantal actieve gebruikers, of de mobiliteit van de gebruiker (bijvoorbeeld bij gebruik van mobiele netwerken). Hierdoor is het kiezen van een vaste resolutie voor een volledige sessie vaak geen ideale aanpak. Om buffering te vermijden, zou men dan moeten kiezen voor een conservatieve resolutie wat betreft de bandbreedte, waardoor de resulterende visuele kwaliteit niet noodzakelijk de hoogst haalbare is.

Om dit probleem aan te pakken werd Adaptive Bitrate Streaming (ABR) geïntroduceerd. ABR is een techniek waarbij dezelfde mediacontent beschikbaar is in meerdere versies met verschillende resoluties en bitrates, typisch opgeslagen op een webserver. In plaats van één continue stream te verzenden, wordt de content opgedeeld in korte segmenten (bijvoorbeeld van enkele seconden lang). De clientsoftware beslist dynamisch, op basis van actuele netwerkcondities, welke versie van het volgende segment moet worden opgehaald. Een visualisatie van dit systeem is terug te vinden op Figuur 2.5. De selectie van de resolutie gebeurt op basis van metingen zoals de downloadtijd van vorige segmenten, de afspeelbufferstatus, en soms zelfs netwerkanalyses.



Figuur 2.5: Een simplistische weergave van het pincipe van adaptive streaming, overgenomen uit [24]

Als het clientprogramma merkt dat er sprake is van vertraging of congestie, zal het tijdelijk overschakelen naar een lagere resolutie om buffering te vermijden. Wanneer de netwerkomstandigheden verbeteren, kan er opnieuw geschakeld worden naar een hogere resolutie, wat leidt tot een betere visuele kwaliteit.

Dit dynamische aanpassingsmechanisme zorgt ervoor dat gebruikers op elk moment de hoogst mogelijke kwaliteit krijgen die hun netwerk toelaat, terwijl het risico op buffering en onderbrekingen tot een minimum beperkt blijft. ABR vormt hierdoor de ruggengraat van moderne streamingdiensten zoals YouTube, Netflix, en andere video-on-demandplatformen, en is essentieel voor een consistente en kwalitatieve gebruikerservaring in variabele netwerkomstandigheden. Een veelgebruikte techniek die dit ABR-principe toepast is MPEG-DASH (Dynamic Adaptive Streaming over HTTP). Het is de open standaard voor het streamen van multimediacontent via het internet. Dit framework zal de stream van deze content opdelen in fragmenten, typisch van enkele seconden lang per fragment. Elk van deze fragmenten is op de server beschikbaar in verschillende resoluties. In MPEG-DASH zal de clientapplicatie steeds de meest optimale resolutie voor het volgende fragment opvragen op basis van waargenomen netwerkcondities. Voor een gedetailleerde uitleg over de werking van MPEG-DASH wordt verwerzen naar het sourcemateriaal van IEEE [40].

ABR kan in de context van deze thesis toegepast worden op de nodes van de scene die over het web verzonden worden. Zo zou het mogelijk zijn per node deze data in verschillende resoluties beschikbaar te stellen, waar gegeven de relevante netwerkcondities, de RL-agent een keuze maakt tussen deze resoluties om de bandbreedte zo goed mogelijk te benutten om met minder round trips een beter beeld te kunnen leveren voor de gebruiker. In de finale staat van het project is ABR echter niet aan bod gekomen vanwege tijdsgebrek. Het zou een mogelijk verder onderzoek kunnen zijn aangezien ABR-streaming via reinforcement learning al eerder goede resultaten opgeleverd heeft, zoals te lezen in het werk [10]. Voor een diepgaande uitleg over adaptive streaming wordt verwezen naar [42].

2.4 Beeldkwaliteitsmetrieken

Om de kwaliteit de beoordelen van een beeld t.o.v. het volledige beeld tijdens het inladen van een scene bestaan er enkele veelgebruikte metrieken. Deze scores worden berekend op basis van het verschil in pixelwaarden van de beelden.

2.4.1 PSNR

In deze thesis maken we gebruik van PSNR of peak signal-to-noise ratio. Dit is een zeer globaal toepasbare techniek. PSNR is gemaakt om signalen te beoordelen: signalen in deze context

omvat informatie die verandert in functie van de tijd of ruimte. Zo zijn beelden signalen omdat hier pixel values veranderen in de functie ruimte, terwijl audio dan weer geluidsgolven zijn die veranderen in functie van de tijd. Video's zijn signalen met een extra dimensie: hier veranderen pixelvalues in functie van ruimte en tijd. Hier kan ook bijkomend audio in de tijd veranderen. PSNR is toepasbaar deze typen signalen en is daarom interessant naar uitreidingen van het project toe, wanneer men agents zou willen ontwikkelen die op verschillende signaaltyes werken.

Peak signal slaat in deze context erop dat de opgemerkte verschillen tussen informatie van de signalen zullen worden geschaleerd ten opzichte van de maximale amplitude van deze informatie. Zo zal voor beelden met 8-bit RGBA waarden de peak signal 255 zijn (mogelijke values liggen tussen 0 en 255 per kleur/alfawaarde), van een genormaliseerde PCM (audio) zal deze 1 zijn (-1 tot 1). De noise uit PSNR slaat op de ruis of vervorming van het signaal. Ruis is vooral bekend bij audio, maar is evenzeer toepasbaar bij andere signalen. Het beduidt het verschil tussen de verwachtte informatie en verkregen informatie, voor beeld zijn dit de pixel values die verschillen ten opzichte van het volledige beeld. PSNR is effectief in het berekenen van zowel lokale als globale ruis: kwaliteitsverschillen tussen beelden door compressie zullen een verschil in PSNR opleveren net zoals missende fragmenten t.o.v. een volledig beeld.

De formule van PSNR is als volgt:

$$\mathrm{PSNR} = 10 \cdot \log_{10} \left(\frac{(\mathrm{MAX}_I)^2}{\mathrm{MSE}} \right) \tag{2.2}$$

Hier is MAX_I de peak signal waarde die we hierboven besproken hebben en MSE de noise uitgedrukt via de Mean Squared Error van de values van het signaal. De peak signal value wordt gekwadrateert omdat dit ook gebeurt bij de MSE. De formule van deze MSE is op zijn beurt:

$$\mathrm{MSE} = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} [I(i,j) - K(i,j)]^2 \tag{2.3}$$

Waar I het originele signaal is en K het signaal wat we willen beoordelen is. Deze MSE itereert over alle informatie in alle dimensies en zal de exponent van het verschil tussen deze optellen. De exponent wordt hier gebruikt om zowel positieve als negatieve verschillen gelijk in rekening te nemen. Dit totaal wordt dan gedeeld door het totaal aantal optellingen die er gedaan werden, wat het totaal aantal values over alle dimensies heen is. In context van beelden is dit de som van het kwadraat van het verschil tussen het originele pixelwaarden en die van het te vergelijken beeld, gedeeld door het totaal aantal pixels.

Om PSNR te bekomen wordt de bekomen fractie uitgedrukt in decibel. Dit doen we door het resultaat van de breuk logaritmisch te schalen met een factor 10. We gebruiken hierbij 10 in plaats van 20 omdat het gaat om een verhouding tussen vermogens (energie per sample, dus amplitude in het kwadraat) en niet rechtstreeks tussen amplitudes. De logaritme maakt het resultaat leesbaarder en intuïtiever, omdat het grote verschillen tussen signaal en ruis herleidt tot een schaal die beter aansluit bij hoe mensen kwaliteit subjectief ervaren. Een hogere PSNR-waarde betekent een kleiner verschil tussen origineel en gereconstrueerd signaal, met een theoretisch maximum van oneindig wanneer er geen enkel verschil is (dus MSE = 0).

Op het eerste zicht lijkt het misschien dat het vreemd is dat PSNR oneindig is bij gelijke beelden en niet 0 zoals eerder intuitief lijkt. Hierbij moet onthouden worden dat we niet de verhouding tussen de intensiteiten van de 2 signalen vergelijken met PSNR, maar de verhouding van de maximale intensiteit, wat ook het maximale verschil tussen 2 signalen is, en de MSE tussen de 2 signalen. PSNR geeft dus eigenlijk eerder aan hoe dicht de gemiddelde ruis tussen de 2 signalen zich verhoudt tot de maximale ruis mogelijk tussen 2 signalen van dat type. De PSNR score zou bij een beeld dus enkel 0 kunnen zijn wanneer elke overeenkomstige pixelwaarden een verschil hebben van 255 en zal oneindig zijn wanneer 2 beelden compleet overeenkomen (zie ook nuldeling door MSE = 0).

Een punt wat het werken met PSNR moeilijker maakt in sommige gevallen is doordat de schaal logaritmisch is en niet lineair. Zo zullen verschillen in MSE een andere impact hebben op de uiteindelijke PSNR wanneer de MSE hoger ligt dan in het ander geval. Zeker in ons geval zal bij het inladen van de eerste correcte fragmenten de PSNR veel minder snel stijgen in verhouding met hetzelfde fragment als laatste (PSNR naar oneindig) of een van de laatste ontbrekende fragmenten in te laden. Dit terwijl deze fragmenten in absolute mate evenveel bijdragen aan ons te bekomen beeld.

2.4.2 SSIM

Een andere metriek die we in overweging genomen hebben, is de **Structural Similarity Index Metric (SSIM)**. Dit is een metriek die specifiek wordt gebruikt voor beelden en verder gaat dan alleen het vergelijken van absolute verschillen tussen pixelwaarden. SSIM berekent de overeenkomst tussen twee beelden door te focussen op drie belangrijke aspecten: structuur, luminantie en contrast.

Luminantie verwijst naar de helderheid van een beeld. Het beschrijft hoe goed de intensiteit van de pixels in een bepaald gebied overeenkomt tussen het originele beeld en het verwerkte beeld. Verschillen in luminantie kunnen optreden als gevolg van belichting, schaduwen of andere visuele factoren.

Contrast betreft het verschil tussen de lichtste en donkerste delen van een beeld. Het contrast meet de variatie in intensiteit van de pixels. Een verschil in contrast kan optreden als de helderheid van bepaalde beelddelen meer verandert dan andere, wat leidt tot verlies van visuele details.

Structuur heeft betrekking op de ruimtelijke arrangementen van de pixelwaarden in het beeld. Het verwijst naar de relatieve posities van intensiteiten, wat cruciaal is voor het behouden van de structuur van het beeld. Wanneer de structuur tussen twee beelden overeenkomt, betekent dit dat de ruimtelijke patronen van de objecten en vormen in beide beelden gelijk zijn, zelfs als de intensiteit of het contrast kan variëren.

De SSIM metriek combineert deze drie factoren om een meer polyvalente vergelijking te maken van de beeldkwaliteit. SSIM meet niet alleen de zichtbare verschillen in helderheid en contrast, maar ook hoe goed de onderliggende structurele informatie van het beeld behouden blijft. SSIM wordt om enkel beelden te beoordelen gezien als de superieure techniek t.o.v. PSNR omdat het in plaats van pure pixelwaarde, zich meer focust op aspecten die meer aansluiten bij de manier hoe de mens deze beelden perceptueert. Voor diepgaandere uitleg en details over SSIM wordt verwezen naar de paper [7].

De formule voor SSIM is als volgt:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{2.4}$$

Waarin:

- μ_x en μ_y de gemiddelde intensiteiten van de twee beelden zijn.
- σ_x^2 en σ_y^2 de varianties van de beelden zijn.
- σ_{xy} de covariantie tussen de twee beelden is.
- C_1 en C_2 kleine constante waarden zijn om deling door nul te voorkomen.

Voor zowel PSNR als SSIM zou het een interessante uitbreiding kunnen zijn om het beeld eerst spatiaal onder te verdelen in gelijke “panelenën deze afzonderlijk te beoordelen. Zowel PSNR als SSIM kunnen moeite hebben met het feit dat kleine mankementen in het beeld grote verschillen in de uiteindelijke score kunnen veroorzaken. Dit is in ons geval echter niet altijd een probleem, aangezien er tijdens het inladen van het volledige beeld regio’s kunnen zijn die al

volledig geladen zijn, terwijl andere delen nog ontbreken. Dit zou kunnen leiden tot een sterke daling van de globale PSNR of SSIM, terwijl het resultaat in werkelijkheid in vele relevante regio's compleet is. Het idee om deze panelen afzonderlijk te beoordelen en het gemiddelde van de scores te gebruiken, kan deze abrupte dalingen vermijden en een meer lineaire beoordeling bieden van missende details in de scene. Bijkomend zou het mogelijk zijn een prioriteitscore aan deze panelen toe te kennen om apart belang te hechten aan verschillende panelen. Zo zouden centrale regio's voor een gebruiker belangrijker kunnen zijn ten opzichte van QOE dan regio's in de hoeken.

2.5 Bestaande oplossingen

In deze sectie bespreken we rond het thema van progressive streaming van 3D data en 3D data in het algemeen, welke methodieken de dag van vandaag gebruikt worden. Ook wordt een state-of-the-art rendering techniek, Gaussian splatting, die de dag van vandaag zeer relevant is en gaan na hoe deze zou kunnen passen in het kader van onze onderzoeksvraag.

Huidige systemen passen voor het progressive streaming van graphics t.o.v. QOE voornamelijk heuristieken toe. Zo zal de volgorde vaak afhankelijk zijn van de afstanden tot de camera, grootte van de fragmenten, intersection van een fragment met het view frustum etc. Een combinatie van zulke heuristieken zullen een aanzienlijke verbetering zijn t.o.v. het inladen in een willekeurige volgorde van fragmenten.

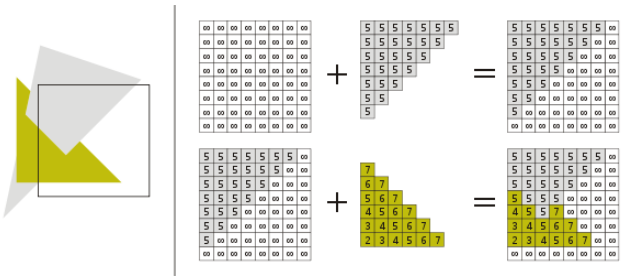
2.5.1 Occlusion culling

Dit is een techniek die tracht objecten te labelen die niet in beeld zijn voor de camera, gegeven het view frustum en andere objecten die het object volledig bedekken. In 3D-rendering wordt deze techniek voornamelijk gebruikt om geen onnodige GPU berekeningen te doen op objecten die voor de gebruiker niet te zien zijn. Bij de probleemstelling van deze thesis zou het op een gelijkaardige manier toegepast kunnen worden om te bepalen welke fragmenten prioriteit krijgen om ingeladen te worden. Objecten volledig buiten het view frustum worden in deze techniek al direct buiten beschouwing gehouden: berekeningen op deze objecten is verloren processing power. Dit wordt frustum culling genoemd.

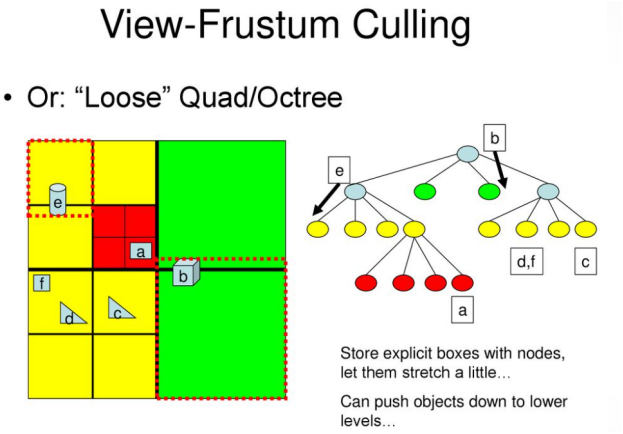
We overlopen kort enkele methodes om frustum culling te kunnen realiseren.

- **z-buffer algoritme**:
  Een voorbeeld van een Occlusion culling methode is het z-buffer algoritme: de renderer onderhoudt een z-buffer die geinitialiseerd is op een maximale z-waarde (overeenkomstig met de far-plane van het view frustum) en zal voor iedere pixel bepalen wat de dichtstbijzijnde te renderen figuur is volgens de z-as. Zo kunnen delen van objecten die een hogere z-waarde hebben in view space dan de overeenkomstige waarde van de z-buffer genegeerd worden omdat deze bedekt zijn door een ander object. De z-buffer wordt zo door de meshes in de scene te overlopen steeds geupdate samen met de beeldbuffer. Grafische weergave van het z-buffer algoritme op Figuur 2.6.
- **Uniform grids**:
  Een efficiëntere manier om botsingen of overlap tussen objecten langs de z-as vanuit de camera te detecteren, is door het view frustum spatiaal te verdelen. Dit kan op verschillende manieren. Een eenvoudige methode is een uniforme grid, waarbij het frustum wordt opgedeeld in gelijke cellen. Alleen cellen die meerdere objecten bevatten, worden verder onderzocht. Deze aanpak is echter niet flexibel: bij een ongelijke verdeling van objecten kunnen veel cellen leeg zijn of slechts delen van grote objecten bevatten, wat leidt tot overbodige of repetitieve testen.
- **Octrees**:
  Een geavanceerdere aanpak gebruikt hiërarchische spatiale datastructuren zoals octrees. Hierbij wordt de ruimte (of een bounding volume rond het frustum) telkens opgesplitst in

8 gelijke subvolumes. Als een cel meerdere objecten bevat, wordt ze verder onderverdeeld, recursief, totdat objecten geïsoleerd zijn of aan een bepaalde drempel voldoen. Zo kunnen botsingen of zichtberekeningen efficiënter gebeuren omdat irrelevante delen van de ruimte snel uitgesloten kunnen worden. Grafische weergaven van octrees op Figuur 2.7.



Figuur 2.6: Z-buffering en occlusion culling (bron: [11])



Figuur 2.7: Octree datastructuur voor 3D-scène-indeling (bron: [2])

Dit zijn methodes die reeds gebruikt worden in het domein van computer graphics in de rendering pipeline zelf. Deze zouden naast de simpelere heuristieken gebruikt kunnen worden om in het kader van onze onderzoeksvraag een sterke heuristiek te vormen voor de volgorde van de progressive streaming van glTF fragmenten te bepalen. Voor diepgaandere uitleg over deze occlusion culling methodes en uitgebreidere methodes verwijzen we naar de paper [33] waar een breed overzicht van de basismethodes van occlusion culling beschreven worden.

2.5.2 Point clouds & Gaussian splatting

3D-scènes werden traditioneel gerepresenteerd aan de hand van polygonale meshes, voornamelijk opgebouwd uit driehoeken. Bij Structure-from-Motion-technieken is het reconstrueren van dergelijke polygonale oppervlakken echter relatief complex. Structure-from-Motion (SfM) is een methode waarbij op basis van meerdere 2D-beelden, genomen vanuit verschillende hoeken rond een scène of object (bijvoorbeeld uit een video), automatisch de 3D-structuur van de omgeving wordt gereconstrueerd.

Een eenvoudiger en succesvoller alternatief voor het representeren van zulke 3D-data is het gebruik van point clouds in plaats van polygonen. Een point cloud bestaat uit een verzameling discrete punten in de 3D-ruimte, waarbij elk punt een positie, kleur en eventueel een normaalvector bezit (in functie van belichting). Deze punten vormen samen een uitgedunde (“sparse”) wolk die de geometrische structuur van de scène benadert. Zonder verdere verwerking zijn point clouds echter vaak onvolledig: bepaalde delen van de scène kunnen niet voldoende gedekt zijn door punten, wat resulteert in gaten of ruis in de visualisatie. Daarom worden point

clouds in veel toepassingen verder verwerkt of verrijkt, bijvoorbeeld met interpolatie, meshing, of technieken zoals Gaussian Splatting.

Gaussian Splatting [21] is een state-of-the-art techniek om op basis van een point cloud een fotorealistische reconstructie van een 3D-scène te genereren. In plaats van polygonale meshes maakt deze methode gebruik van zogenaamde *Gaussian splats*: anisotrope, ellipsvormige Gaussiaanse functies die in de 3D-ruimte gepositioneerd zijn. Elke Gaussian bezit een positie, een vorm (bepaald door een covariantiematrix), een kleur, een alpha-transparantie en eventueel een oriëntatie. Deze "splats"worden geprojecteerd naar het 2D-beeldvlak en daar visueel gecombineerd (geblend) om zo een realistisch beeld van de scène te creëren. De parameters van de Gaussians worden geoptimaliseerd op basis van een set van inputbeelden, zodat de gerenderde weergaven zoveel mogelijk overeenkomen met de originele beelden. Dit resulteert in vloeiende overgangen tussen verschillende gezichtspunten en een hoge visuele kwaliteit, zelfs zonder expliciete oppervlaktestructuur.

Gaussian Splatting is relevant binnen het kader van deze thesis, waarin gefocust wordt op het progressief streamen en laden van 3D-inhoud. Omdat deze techniek gebruikmaakt van afzonderlijke Gaussians als basisentiteiten, kunnen deze eenheden, net zoals fragmenten in een glTF-scène, individueel geëvalueerd worden op hun relevantie voor het huidige camerastandpunt. Hierdoor wordt het mogelijk om de laadvolgorde van Gaussians (of fragmenten) te optimaliseren op basis van hun visuele bijdrage aan het eindbeeld, bijvoorbeeld door hun zichtbaarheid, afmetingen op het scherm of occlusie. Gaussian Splatting biedt zo een conceptueel kader waarin fragment-prioritering als optimalisatieprobleem kan worden benaderd, en vormt daarmee een interessante inspiratiebron voor het ontwerp van heuristieken of leeralgoritmes binnen deze thesis.

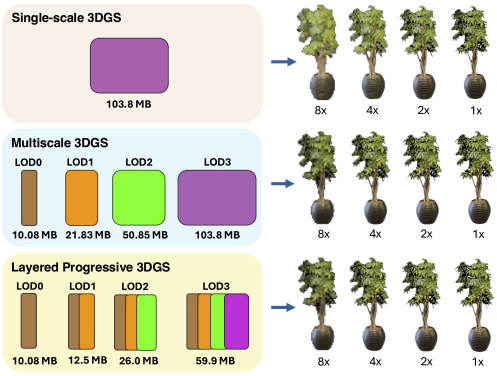
2.6 Gerelateerd werk

Om de probleemstelling van deze thesis te beantwoorden werd eerst onderzoek gedaan naar bestaande implementaties rond de relevante deeldomeinen om beschikbare frameworks rond deze domeinen te leren kennen en designprincipes te overwegen die geïntroduceerd werden. Hoewel tijdens de literatuurstudie geen voorgaand project gevonden is dat direct aanleunt bij de onderzoeksvraag van het gebruik van reinforcement learning bij progressive streaming van glTF 2.0 data, kon nuttige informatie gehaald worden uit implementaties die een of meerdere deeldomeinen van deze thesis behandelden. In deze sectie geven we een overzicht van implementaties die gebruikt werden als inspiratie voor het project dat ontwikkeld werd in deze thesis geordend volgens een van de deelthema's waar het werk onder valt. Verder wordt beschreven hoe deze aanleunde bij en relevant waren voor dit onderzoek en welke designprincipes hiervan meegenomen werden in eigen ontwikkeling.

2.6.1 Progressive streaming

Door de mogelijk grote bestandsgrootte van 3D-scènes, opgebouwd uit *textured meshes* (zoals opgeslagen in het glTF-formaat), is het vaak niet haalbaar om deze volledig in één keer via het web te versturen. Bij conventionele dataoverdracht is het gebruikelijk om te wachten tot de volledige dataset ontvangen is alvorens deze te gebruiken. In het geval van complexe of omvangrijke 3D-scènes leidt dit tot aanzienlijke opstartvertragingen, wat de *Quality of Experience* (QoE) van de gebruiker negatief beïnvloedt.

In het werk *Progressive Network Streaming of Textured Meshes in the Binary glTF 2.0 Format* [28] wordt een methode voorgesteld om glTF 2.0-binary data progressief te streamen. De volgorde waarin meshes verzonden worden, wordt bepaald aan de hand van een heuristiek die prioriteit geeft aan meshes binnen het gezichtsveld (field of view, FOV) van de camera en die dichter bij de camera liggen. In de praktijk werd hiervoor een strafwaarde van 100 toegekend aan meshes buiten het FOV, die werd opgeteld bij de afstand van het middelpunt van de mesh tot de camera. De meshes met de laagste gecombineerde score werden als eerste verstuurd.



Figuur 2.8: Een overzicht van hoe de gelaagde LOD-structuur van Gaussian Splatting verschilt ten opzichte van een single-scale Gaussian splat en een adaptive multiscale Gaussian splat. Overgenomen uit de paper [39].

De segmentatie van de glTF-data gebeurt door het parsen van de metadata van het bestand, waarin onder meer bounding boxes en bufferviews per glTF-node zijn gespecificeerd. Deze metadata wordt gecombineerd met camerainformatie om de volgorde te bepalen, waarna via de bufferviews de overeenkomstige binaire data wordt geëxtraheerd. Op basis van een selectie van meshes (afhankelijk van de maximale bestandsgrootte per segment) wordt een nieuw glTF-bestand samengesteld en verzonden naar de client. Deze procedure wordt herhaald tot de volledige scène beschikbaar is. Voor verdere technische details wordt verwezen naar het originele artikel [28]. Deze thesis bouwt voort op dit onderzoek door te onderzoeken of de laadvolgorde van meshes geoptimaliseerd kan worden via een reinforcement learning (RL) framework, dat op basis van cameradata en glTF-metadata een efficiënte sequentie van meshtransmissies leert bepalen. Zo wordt de manier van segmentatie alsook het gebruik van glTF 2.0 data overgenomen van dit project, waar het bepalen van de volgorde van deze segmenten onderzocht wordt.

Progressive streaming hoeft echter niet enkel te gebeuren door de scène te segmenteren in volledige fragmenten, maar kan ook op andere manieren worden benaderd. Zo kan het *Level-Of-Detail* (LOD) van het streamen van een scène ook progressief verlopen. In dit soort applicaties worden fragmenten aangeboden in een structuur van lagen: de basislaag vormt het silhouet van het object zonder hoge LOD; details zoals kleine submeshes en materialen worden in deze versie weggelaten. Progressief kunnen er per fragment extra lagen van detail worden opgevraagd, zodat op deze manier incrementeel detail wordt toegevoegd aan assets in de scène, met voorrang voor de meest prominente assets die zichtbaar zijn voor de gebruiker. Een interessante toepassing van deze LOD-progressieve streaming, toegepast op *Gaussian Splatting*, is het werk *LapisGS: Layered Progressive 3D Gaussian Splatting for Adaptive Streaming* [39]. Dit is een recent werk waarin de Gaussian splats worden opgedeeld in splats die bestaan uit verschillende lagen, met incrementeel meer detail per splat. Een verschil met de principes van *adaptive streaming* is dat hier, om een asset in volledig detail te verkrijgen, alle splats incrementeel toegevoegd moeten worden. De splats vormen als het ware lagen van detail die aan het silhouet worden toegevoegd, en bevatten niet opnieuw de informatie uit de vorige, minder gedetailleerde splat. Hierdoor kan, in het kader van progressive streaming, bandbreedte worden bespaard door het vermijden van herhaalde transmissie van data uit eerdere lagen. Voor een visualisatie van deze gelaagde splats verwijzen we naar Figuur 2.8.

2.6.2 Machine learning in streaming

Het gebruik van Machine learning in streaming, is geen nieuw concept. Voornamelijk in het veld van ABR zijn verschillende implementaties ontstaan die aan de hand van netwerkcondities als input, gewenste resoluties kunnen bepalen via een RL-agent.

Een voorbeeld van zulke implementatie die bestudeert is voor deze thesis is het werk “Adaptive

streaming algorithm based on reinforcement learning" [10]. In deze studie wordt een ABR-algoritme (Adaptive Bitrate) ondersteund met een RL (Reinforcement Learning) model, dat op basis van de actuele netwerkcondities een geschikte bitrate selecteert voor videostreaming zoals eerder uitgelegd in Sectie 2.3. MPEG-DASH is een ABR-framework waarbij voor elke video meerdere versies in verschillende resoluties beschikbaar zijn, met als doel de meest geschikte versie te selecteren om buffering te minimaliseren en de resolutie te optimaliseren ten voordele van de QoE (Quality of Experience) van de kijker. Traditioneel wordt de bitrate rechtstreeks gekozen op basis van gemeten netwerkcondities. In deze studie daarentegen worden deze condities als input doorgegeven aan een RL-agent, die op basis daarvan leert om de meest geschikte resolutie te selecteren. Hoewel het doeleinde (bepalen bitrate) en onderwerp (video) verschilt van deze probleemstelling, kan het toch interessant zijn hier te kijken naar de manier hoe men hier een RL agent traint en wat deze als observation space aangeboden krijgt. Er werd een PPO-gebaseerd model ontwikkeld voor adaptieve bitrate selectie, waarbij historische netwerkgegevens en bufferniveau als input dienden. Het model overtrof bestaande heuristieken zoals Buffer-Based en MPC-methodes (Model predictive control) in termen van gemiddelde totale reward. Het wist een stabielere en hogere bitrate te selecteren met minder schommelingen en herbuffering, wat resulteerde in een betere kijkervaring.

OFB-VR (Optical flow based VR) [34] is een iets ouder werk uit 2020, maar is in onze context relevant omdat het een RL-agent traint, waar de training gebeurt op basis van een bewerkte versie van PSNR. Verder gebruiken ze ook een tiling techniek, wat voor verdere implementaties op deze thesis ook overwogen is. In dit werk tracht men de Quality of Experience (QoE) te optimaliseren voor streaming naar VR-headsets. De focus ligt opnieuw op ABR (adaptive bitrate streaming), waarbij een reinforcement learning-agent getraind wordt om de meest geschikte kwaliteit voor elk segment of tile van de video te selecteren. De centrale QoE-factor in deze studie is gebaseerd op *optical flow*: een techniek die de beweging van pixelgroepen tussen opeenvolgende frames beschrijft (bijvoorbeeld door camerabeweging doorheen een scène zullen pixelgroepen verplaatsen). Deze informatie laat toe om te voorspellen welke delen van het beeld belangrijk(er) zullen zijn voor de kijker, wat cruciaal is in de context van VR, waar gebruikers zich vrij kunnen bewegen binnen een 360° omgeving.

In dit onderzoek wordt een tile-based streamingmethode toegepast binnen het ABR-kader. Hierbij wordt de originele video opgesplitst in spatiale "tegels"(tiles) die elk een apart kwaliteitsniveau kunnen krijgen. Door gebruik te maken van optical flow worden tegels in regio's met hoge visuele prioriteit geïdentificeerd en kunnen deze een hogere resolutie toegekend krijgen. Bovendien worden tegels met gelijkaardige visuele eigenschappen gegroepeerd tot grotere tileblokken met uniforme kwaliteitsinstellingen, om zo de bandbreedte-efficiëntie te verbeteren. Om te kunnen beoordelen welke tegels belangrijk zijn voor de gebruiker, ontwikkelden de auteurs een nieuwe QoE-metriek: **PSNR-OF** (Peak Signal-to-Noise Ratio - Optical Flow). Deze metriek combineert traditionele PSNR met informatie uit optical flow en dieptekaarten om de waargenomen kwaliteitsimpact van compressie per regio beter te modelleren. Concreet wordt hier bij het berekenen van de MSE tussen het originele beeld een score in rekening gehouden, die aangeeft hoe "merkbaar"de fout is voor een gebruiker op basis van dept en optical flow. Op basis hiervan wordt vervolgens een reinforcement learning-agent (gebaseerd op het A3C-algoritme) getraind om adaptieve bitratebeslissingen te nemen die de gemiddelde PSNR-OF-score maximaliseren. Dit resulteert in een streamingstrategie die zowel efficiënter omgaat met beschikbare bandbreedte als beter inspeelt op de perceptuele gevoeligheden van VR-gebruikers. De voorgestelde OFB-VR-methode presteert beter dan de toen bestaande VR-streamingmethoden zoals Plato [19] en Pano [14], zowel qua bandbreedtebesparing als Quality of Experience (QoE), gemeten via de PSNR-OF-metric. Dit zijn heuristische aanpakken die de kwaliteit direct bepalen op basis van gebruikersperceptie, hoofdbewegingen, en bandbreedtebeperkingen.

2.6.3 Reinforcement learning-implementaties

Voordat het implementatieproces van start ging, werd eerst gekeken naar bestaande reinforcement learning-projecten waarvan de implementatie publiek beschikbaar was. Deze dienden

als inspiratiebron voor het ontwerp van het systeem dat in deze thesis is ontwikkeld. De eerste referentiepunten waren eenvoudige, maar goed gedocumenteerde voorbeelden van de Gymnasium-bibliotheek [9], met name de implementatie van de omgeving "CartPole-v1". Deze voorbeelden boden inzicht in de basisstructuur van een reinforcement learning-project—zoals de opbouw van episodes, interactie tussen agent en omgeving, en de trainingsloop. Dit framework vormde uiteindelijk de ruggengraat van de eigen codebase. Naast deze 'out-of-box' voorbeelden werd ook inspiratie gehaald uit complexere realisaties, waarbij meer geavanceerde technieken en omgevingen aan bod kwamen. In wat volgt, worden enkele van deze studies kort besproken, met aandacht voor hun belangrijkste bijdragen en hun relevantie voor de aanpak in deze thesis.

Een klassiek voorbeeld van de toepassing van reinforcement learning is te vinden in het werk [30], waarin een Deep Q-Network (DQN, zie Sectie 2.1.2) ontwikkelen dat Atari-spellen leert spelen op menselijk niveau. Deze studie was een mijlpaal in het veld, aangezien het de eerste succesvolle grootschalige integratie van Q-learning met diepe neurale netwerken liet zien. In plaats van handmatig regels te definiëren, leerde het model autonoom complexe sequentiële beslissingen te nemen puur op basis van pixels als input. Deze aanpak markeert het beginpunt van veel modern reinforcement learning-onderzoek, waarin visuele input, sequentiële besluitvorming en generalisatie over taken heen centraal staan.

In het verlengde daarvan wordt in deze thesis een ander reinforcement learning-algoritme gebruikt, namelijk Proximal Policy Optimization (PPO zie Sectie 2.1.3). De keuze hiervoor wordt mede ondersteund door de studie The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games [49]. In dit werk wordt PPO—ondanks zijn relatief eenvoudige, on-policy aard—met succes toegepast in complexe coöperatieve multi-agent omgevingen zoals StarCraft en Google Research Football. Dit zijn omgevingen waar meerdere agents tegelijk ingezet worden als NPCs. De auteurs laten zien dat PPO, mits zorgvuldig geïmplementeerd en goed afgesteld qua hyperparameters, in staat is om prestaties te leveren die vergelijkbaar zijn met veel complexere off-policy methodes. Een belangrijk aspect in hun aanpak is de toepassing van centrale training met decentrale uitvoering (CTDE), waarbij agents leren op basis van lokale observaties, maar het gezamenlijke doel centraal staat. Dit sluit aan bij het scenario in deze thesis, waarin het model leert om, op basis van lokale node-informatie, een globale laadstrategie voor 3D-fragmenten te optimaliseren.

Om het leerproces verder te sturen en te stabiliseren, wordt in deze thesis gebruikgemaakt van action masking: het dynamisch uitsluiten van ongeldige of irrelevante acties (zoals het opnieuw laden van reeds geladen nodes). De relevantie van actiebeheer wordt uitvoerig besproken in het werk van Kanervisto et al. [20], waarin verschillende technieken worden onderzocht om de actieruimte van een RL-agent te optimaliseren. Door zelden succesvolle acties te verwijderen (RA), continue acties te discretiseren (DC) of multi-discrete ruimtes te herstructureren (CMD), tonen de auteurs aan dat een meer gerichte actie-ruimte leidt tot snellere en stabielere training. Het gebruiken van action masking als een soort chaos-parameter die de optimale acties sporadisch uitmasked is een zeer interessante vorm van gebruik. Een toekomstige test voor dit project kan zijn om op deze zelfde wijze acties uit te masken om regelmatig nieuwe sequenties proberen te vormen.

Tot slot biedt de paper die eerder aangehaald werd in de Sectie 2.1.4, A Closer Look at Invalid Action Masking in Policy Gradient Algorithms [18], waardevolle richtlijnen voor de correcte implementatie van action masking. De implementatie die gedaan werd was ook zeer interessant om te bekijken en de manier waarop masking toegepast werd was een goede introductie naar dit concept toe.

Gezamenlijk illustreren deze werken hoe reinforcement learning effectief kan worden toegepast in uiteenlopende omgevingen—van klassieke Atari-games tot multi-agent systemen—en hoe praktische keuzes in de opzet van actie-ruimtes en algoritmes de sleutel vormen tot stabiel en efficiënt leren. De implementatie in deze thesis staat in directe lijn met deze inzichten.

Hoofdstuk 3

Projectstructuur

In deze thesis wordt een reinforcement learning model iteratief ontwikkeld door stapsgewijs complexiteit toe te voegen. Om deze taak te realiseren is er een codebase opgesteld met de nodige functionaliteit om de reinforcement agent te trainen om progressive streaming sequenties te genereren. Hiernaast ondersteunt het de functionaliteit om de performantie van deze getrainde agent te visualiseren en te analyseren. De trainingsloop van deze agent zelf bestaat uit meerdere delen: deze moet tijdens het trainen van deze agent renders kunnen uitvoeren om zijn prestatie te beoordelen om zijn strategie correct aan te kunnen passen. Omdat we de vruchten willen plukken van de voordelen van meerdere programmeertalen zullen we een systeem ontwikkelen waar meerdere processen parallel lopen en communiceren met elkaar. In dit hoofdstuk van de thesis gaan we over een overzicht van de codebase die ontwikkeld werd om het onderzoek uit te voeren. Bij ieder deel wordt de use-case binnen het project en wijze van implementatie, zodanig dat deze codebase duidelijk wordt voor eventuele uitbreidingen.

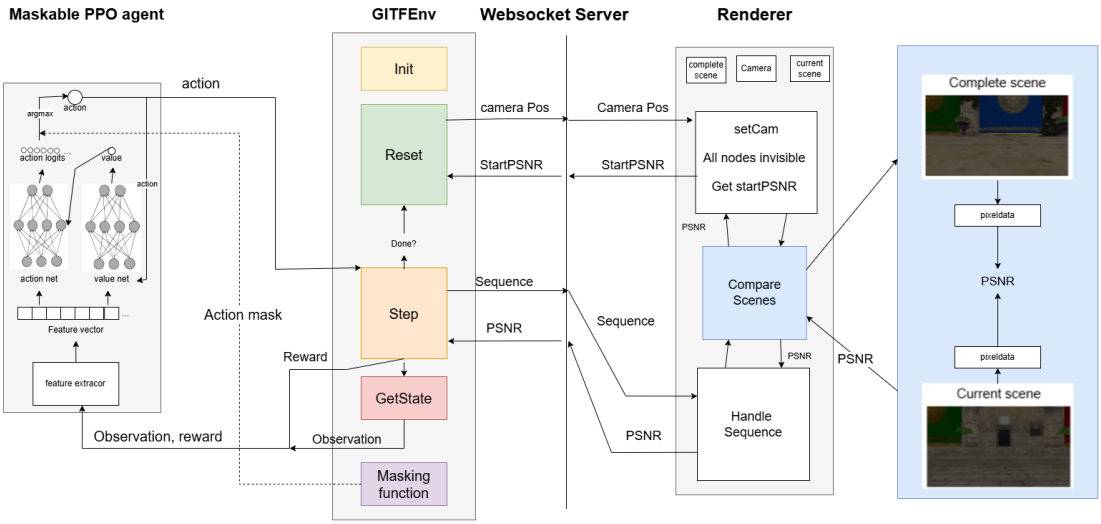
Het project waarmee het model ontwikkeld is, bestaat uit verschillende componenten. De reinforcement learning code is geschreven in Python, terwijl de rendering van de scene gebeurt in JavaScript. De processen communiceren met elkaar via een WebSocket server, waar de Python code voornamelijk te renderen sequenties van fragmenten verstuurt en de JavaScript code deze zal renderen en PSNR t.o.v. de volledige scene zal berekenen en terugsturen. Een globaal overzicht van de basis van het gehele project wodrdt schematisch weergegeven op figuur 3.1. De details op deze figuur worden in verdere secties in dit hoofdstuk toegelicht.

Voor de JavaScript module zijn twee versies ontwikkeld: een versie in React waar niet alleen de functionele rendering plaats vond, maar ook functionaliteit om te experimenteren met de glTF data en het visualiseren van performantie van getrainde modellen. Anderzijds werd een simpelere versie met plain HTML/JS ontwikkeld die aangeroepen wordt via een puppeteer script om rendering via headless Chrome te kunnen dirigeren. Deze versie werd ontwikkeld om efficiënter modellen te kunnen trainen op de VSC cluster waar geen visuele component beschikbaar is.

3.1 glTF Environment

Het doel van de glTF environment (of kort GlTFEnv) is om het probleem van het sequentieel streamen van nodes zo realistisch mogelijk voor te stellen aan de agent. Daarbij moet de omgeving relevante data aanleveren aan de RL-agent, zodat die een geïnformeerde keuze kan maken over welke node als volgende geladen moet worden. Daarnaast moet de environment ook het effect van een actie, het toevoegen van een node aan de huidige sequentie, kunnen simuleren en evalueren, om zo bij te dragen aan het bepalen van een optimale laadvolgorde.

We willen bovendien dat deze omgeving voldoende flexibel is om gebruikt te worden met ver-



Figuur 3.1: Een overzicht de gehele structuur van het project waarmee de agent getraind wordt. De componenten zullen in de loop van dit hoofdstuk toegelicht worden.

schillende glTF-scenes, camerastandpunten, en uitbreidingen in de toekomst.

3.1.1 Implementatie

Zoals eerder vermeld, wordt deze reinforcement learning environment geïmplementeerd als een Python-project. Deze keuze is gebaseerd op de brede ondersteuning en de beschikbaarheid van bestaande frameworks in Python die het ontwikkelen van RL-agents en -omgevingen vereenvoudigen.

Een van de meest gebruikte libraries voor het bouwen van reinforcement learning environments is Gymnasium, een community-maintained fork van de oorspronkelijke OpenAI Gym-library [9]. De originele Gym-library [32] verwijst voor verdere ontwikkeling expliciet door naar het Gymnasium-project, dat wordt onderhouden door de Farama Foundation. Op de officiële websites van beide projecten zijn uitgebreide richtlijnen beschikbaar over de interfaces en het gebruik van deze libraries.

Daarnaast zijn er op de website van Gymnasium ook tal van bestaande testomgevingen beschikbaar. Deze kant-en-klare environments maken het mogelijk om snel te experimenteren met reinforcement learning zonder eerst een eigen omgeving te moeten implementeren. Tot slot biedt de Farama Foundation ook een wetenschappelijke publicatie aan met een diepgaandere toelichting op het Gymnasium-project [47].

Buiten het gebruik van voorgemaakte omgevingen biedt `Gymnasium` ook de mogelijkheid om een eigen, op maat gemaakte omgeving te definiëren. Dit gebeurt door een klasse te definiëren die overerft van de `Env`-klasse uit de `Gymnasium`-library. Daarbij moeten enkele methods en attributen worden geïmplementeerd in de custom environment om compatibel te zijn met het Gymnasium-framework:

- `__init__`: Omdat de environment wordt gedefinieerd als een klasse in Python, zal deze steeds een `__init__`-functie bevatten. Deze functie wordt opgeroepen bij het initialiseren van een nieuwe instantie van de environment. Binnen deze functie worden typisch de noodzakelijke interne variabelen geïnitialiseerd, zoals counters, buffervelden en koppelingen met externe componenten (zoals de WebSocket-server en renderer). Ook worden hier de `action_space` en `observation_space` gedefinieerd. Deze spaces zijn geen functies, maar objecten die aangeven hoe de actie- en observatieruimte eruitzien waarbinnen de agent mag opereren. Ze vormen een essentieel deel van de Gymnasium-API.

- `action_space`: De `action_space` beschrijft welke acties geldig zijn binnen deze environment. Dit is een object (geen functie) dat door de Gymnasium-API wordt gebruikt om o.a. te controleren of een actie geldig is, of om willekeurige geldige acties te genereren tijdens testen. In onze implementatie gebruiken we een `Discrete(n)` space, waarbij `n` gelijk is aan het aantal nodes in de glTF-scène. Dit betekent dat de agent telkens één actie kiest uit `n` mogelijke opties, waarbij elke actie staat voor het toevoegen van een specifieke node aan de huidige sequentie. Voor andere scenario's zouden ook andere types mogelijk zijn, zoals `MultiDiscrete`, `Box` of `MultiBinary`, afhankelijk van de aard van de beslissingsruimte.

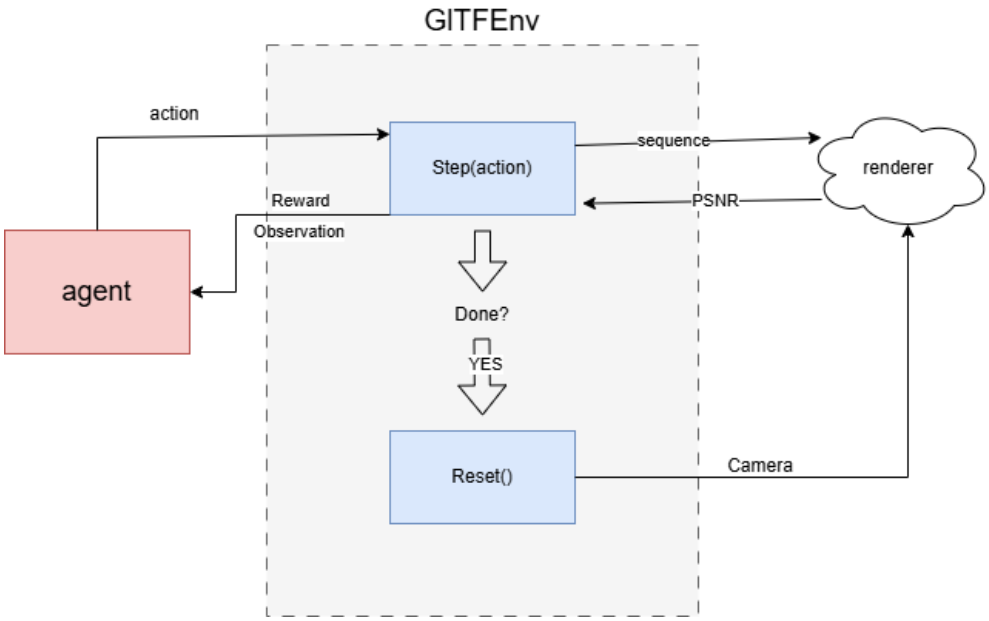
- `observation_space`: De `observation_space` definieert de vorm en structuur van de observaties die de agent ontvangt van de environment. Het is een object dat aangeeft wat voor soort en welke waarden de observaties kunnen aannemen, zodat de agent weet welk type informatie hij kan verwachten en verwerken. De observation space zal tijdens het doorlopen van de environment steeds ingevuld worden met de relevante data die de agent zal gebruiken om zijn volgende actie te bepalen. Typische types van observation spaces in Gymnasium zijn bijvoorbeeld `Box`, wat gebruikt wordt voor continue numerieke observaties zoals vectors of matrices van reële getallen binnen bepaalde grenzen; `Discrete`, dat een eindig aantal discrete waarden beschrijft; en `Dict`, dat complexere observaties mogelijk maakt door verschillende componenten met uiteenlopende types te combineren in een dictionary. Door de `observation_space` te definiëren, kan het reinforcement learning framework de input van de agent valideren en begrijpen, en kan de agent de omgeving op een consistente manier waarnemen en interpreteren. Het declareren van de observation space en ook de action space, is eigenlijk het bepalen van het formaat van de input- en outputvector waarmee de agent uiteindelijk mee te werk zal gaan.

- **step**: De `step`-functie wordt aangeroepen bij het nemen van een actie, met deze actie als argument. Haar taak is om de environment te updaten op basis van de actie, en een rewardscore toe te kennen. In ons geval ontvangt deze functie een index van de in te laden node. Deze index wordt toegevoegd aan een lijst van genomen acties en via WebSocket verstuurd naar de WebSocket-server. De server stuurt deze sequentie door naar de renderer in JavaScript, die de sequentie rendert en vergelijkt met de volledige scene vanuit hetzelfde camerastandpunt (voor verdere uitleg over dit proces wordt verwezen naar Sectie 3.4). De PSNR wordt vervolgens op dezelfde manier teruggestuurd naar de `step`-functie, die hiermee de reward berekent. Ten slotte wordt bepaald of deze actie de episode beëindigt (bijvoorbeeld wanneer alle nodes geladen zijn), en wordt dit samen met de reward en de nieuwe observatie als resultaat teruggegeven. Hierbij wordt in ons geval de observatiestate aangepast, waarbij de `loaded`-bit van de betrokken node wordt geüpdatet.

- **reset**: De `reset`-functie initialiseert de omgeving aan het begin van een nieuwe episode. Meestal betekent dit dat interne lijsten en parameters leeggemaakt worden. In het geval van glTF-env zal het camerastandpunt van de volgende episode naar de renderer verzonden worden, de sequentielijst terug leeggemaakt worden, preprocessing van de renderer in functie van de observation space gedaan worden etc. De `observation_space` wordt hier ook opgebouwd. Deze informatie wordt opgeslagen in de environment. De reset-functie retourneert de startobservatie via de `getState`-functie. Voor een schematisch overzicht hoe de reset-functie en step-functie met elkaar interageren verwijzen we naar Figuur 3.2.

- **getState**: Deze functie vult de `observation_space` in en retourneert de huidige observatie. Voor deze probleemstelling kunnen zaken als camera-coördinaten, mesh coördinaten, loaded state etc. relevant zijn om deze observation mee te vullen. Let op: te veel irrelevante of dubbele data kan de agent ervan beletten zijn optimale performantie te halen. Hierbij is het belangrijk dat er goed onderzocht wordt welke data cruciaal zijn voor de agent om zijn keuze van actie te maken.

- **masking function**: We lopen hier even voor op de feiten: in deze thesis onderzoeken we



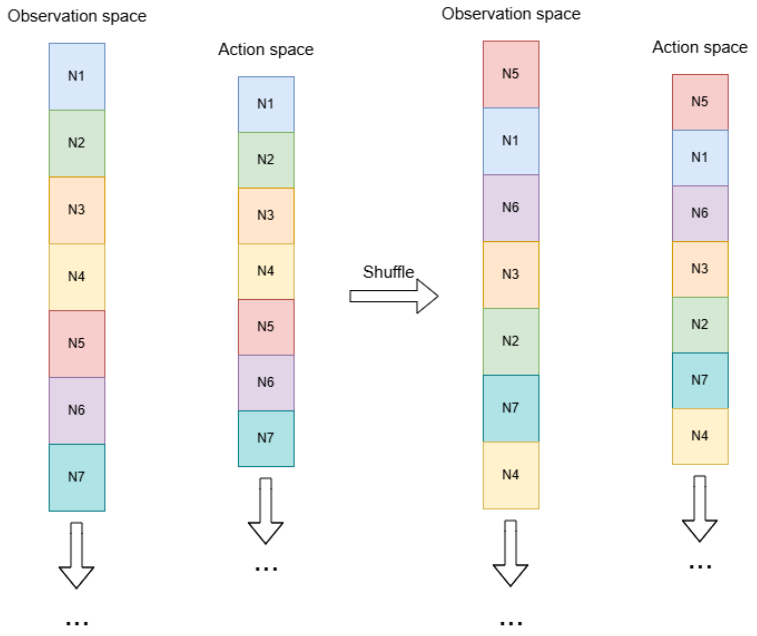
Figuur 3.2: Schematische voorstelling van het leerproces van een agent, in het bijzonder hoe de step en reset functie met elkaar interageren.

onder andere de effectiviteit van het gebruik van action masking (zie Sectie 2.1.4) in deze context. Diepgaandere uitleg sparen we voor Sectie 5, waar het experiment uitgevoerd wordt en resultaten die onze oplossing opleverde besproken worden. De `masking`-functie wordt intern aangeroepen bij het gebruik van een `Maskable`-agent om aan te geven welke acties geldig zijn. Voor een discrete `action_space` retourneert deze functie een binaire array, waarbij elk element aangeeft of de overeenkomstige index (actie) nog toegestaan is. Een voorbeeld in deze environment kan zijn dat een actie een positief bit krijgt als de bijhorende node nog niet eerder is ingeladen.

Door deze componenten in functie van onze use-case in te vullen vormen wordt de glTF environment gevormd. Deze environment zal per episode een sequentie of permutatie van alle in te laden meshes of nodes moeten kiezen, waarbij dus iedere node slechts 1 keer in mag voorkomen. Dit kan verwezelijkt worden door de eindstate van een episode te definiëren als de states waarin alle nodes van de huidige glTF scene in de huidige sequentie aanwezig zijn. Verder kunnen de acties die eenzelfde node een tweede keer probeert in te laden verbieden door deze niet uit te voeren of te masken.

Verder zal de step functie van deze environment communicatie naar de renderer moeten voeren via de WebSocket server om het effect van de gekozen actie te simuleren. Van deze renderer zal dan een score terugkomen die we kunnen gebruiken om een reward op te stellen die we aan deze actie toekennen. De reset functie gaat alle variabelen die de episode betreffen terug moeten initialiseren en eventuele variatie toevoegen aan het proces. Zo zou het mogelijk zijn om tussen episodes van camerastandpunt te wisselen, van volgorde waarin de nodes beschreven zijn, of zelfs de gehele scene te wisselen. Merk op dat we dit laatste ook kunnen verwisselen door na elkaar verschillende glTF-environments te declareren die elk andere scenes bevatten en de agent beurtlings hierop traint. Deze methode kan echter voor overhead zorgen van het initialiseren van een nieuwe Gymnasium environment.

De environment bevat ook functionaliteit om de volgorde van acties willekeurig te herschikken (shuffling). Zonder deze functionaliteit bestaat het risico dat het model tijdens de training een harde correlatie leert tussen specifieke node indices en gunstige acties—zoals het inladen van grote muren of achtergronden—zonder deze voorkeuren af te leiden uit de observation space.



Figuur 3.3: Visualisatie van hoe de observation space en action space op gelijke wijze geshuffled worden om te vermijden dat node indices op harde wijze vanbuiten geleerd worden.

Zolang er met dezelfde scène gewerkt wordt, vormt dit geen probleem. Wanneer het model echter toegepast wordt op nieuwe, ongeziene scènes, verliezen deze specifieke node indices hun betekenis, wat kan resulteren in slechtere prestaties. Om dit te vermijden, werd een mechanisme geïmplementeerd dat zowel de action space als de observation space op identieke wijze herschikt. Dit heeft als doel het model te dwingen relevante informatie af te leiden uit de observaties zelf, in plaats van uit de volgorde van de knopen. In de uiteindelijke testfase werd deze shuffle-functie achter wege gelaten door tijdsgebrek. Een visualisatie van deze shuffle te zien op Figuur 3.3.

Daarnaast werd de environment ontworpen met het oog op uitbreidbaarheid en generalisatie. Indien een glTF-bestand minder knopen bevat dan het maximum waarvoor het model getraind werd, zal de action mask automatisch acties deactiveren die geen corresponderende node hebben. Hierdoor is het mogelijk een agent te trainen op scènes met een vast maximaal aantal knopen, en deze later toe te passen op ongeziene scènes met een lager aantal knopen. Op die manier kunnen agents getraind worden op een gevarieerde set van scènes en gegeneraliseerd worden naar nieuwe situaties, zonder aanpassing van het netwerk.

3.1.2 Gebruik

Nu we de structuur van de environment besproken hebben, overlopen we kort hoe deze gebruikt kan worden om een agent in deze omgeving te laten leren.

De environment dient geïnitialiseerd te worden met behulp van enkele parameters:

- node: Een array die voor elke node uit het glTF-bestand de noodzakelijke data bevat voor onze environment. Deze array kan bekomen worden via de functie `parse_gltf` uit ons project. Voor een gedetailleerde uitleg van deze parsing verwijzen we naar de sectie over dataverzameling en preprocessing (3.5).
- `send_queue` **en** `receive_queue`: Deze queues vormen het communicatiekanaal tussen de environment en de WebSocket-server. Via deze kanalen kunnen node-sequenties en con-

troleberichten verzonden worden. Voor een uitgebreide uitleg verwijzen we naar Sectie 3.3.

- `num_valid_nodes`: Een variabele die aangeeft hoeveel van de nodes in de `node_data`-array als geldig beschouwd moeten worden. Meestal komt dit overeen met de lengte van de array, maar indien dit getal kleiner is, wordt bij elke episode willekeurig een subset van dit aantal nodes gekozen. Wanneer de waarde groter is dan het aantal nodes in de scene zal deze overschot aan nodes met nulwaarden aangevuld worden en uitgemaskeerd worden door de masking function. Deze functionaliteit werd voornamelijk experimenteel toegevoegd.
- `camera_positions`: Een array van koppels van 3D-vectoren: de cameraposities en bijhorende kijkrichtingen waarop getraind moet worden. Indien slechts één koppel wordt opgegeven, blijven de camerapositie en kijkrichting constant tijdens de volledige training. Wanneer meerdere posities opgegeven worden, wordt na elke episode willekeurig een nieuw standpunt gekozen uit deze lijst.

Eens het environment-object geïnitialiseerd is, kan het gebruikt worden als trainingsomgeving voor een reinforcement learning-agent. In het bijzonder raden we de reinforcement learning-bibliotheek Stable Baselines3 aan [35] (zie ook de uitgebreidere paper [36]) wegens compabiliteit met deze environment. Deze bibliotheek bevat alle courant gebruikte algoritmes voor policy learning binnen RL en werkt naadloos samen met `Gymnasium`-environments. Via de SB3-API kan men het agent-model trainen op de environment en het resulterende model toepassen op nieuwe situaties.

3.2 PPO-agent

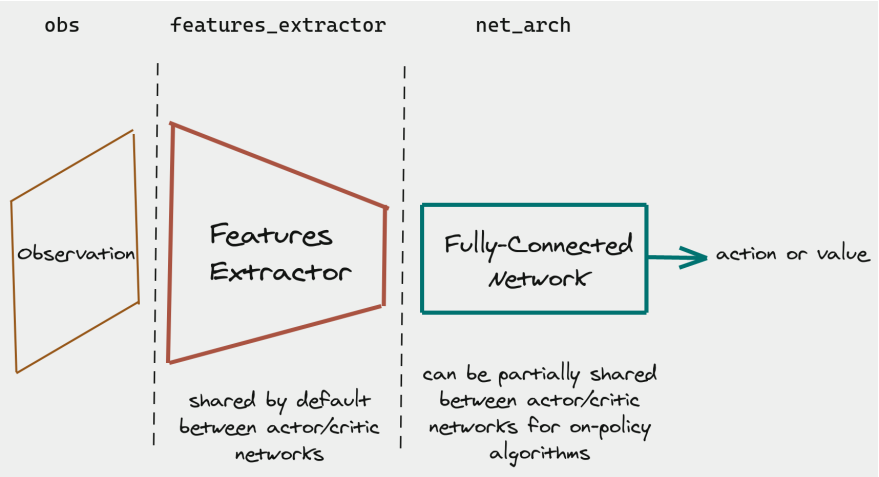
In dit project maken we gebruik van een reinforcement learning-agent die PPO (Proximal Policy Optimization, zie Sectie 2.1.3) toepast. PPO is om meerdere redenen geschikt voor onze toepassing. Ten eerste is PPO een stabiele *policy gradient method*: de policy-updates worden geclipped (Voor extra uitleg zie vergelijking 2.2), waardoor de nieuwe policy niet te sterk afwijkt van de vorige. Dit zorgt voor een gecontroleerd en consistent leerproces. Ten tweede is PPO compatibel met onze gedefinieerde action- en observation spaces. Niet alle policy-optimalisatiealgoritmes kunnen omgaan met arbitraire spaces, maar PPO is hierin bijzonder flexibel. In eerdere projecten werd PPO ook met succes toegepast zoals aangehaald in Sectie 2.6.3. Voor meer informatie over het PPO-algoritme verwijzen we naar Sectie 2.1.3.

We stelden al vroeg vast dat het gebruik van een *action mask* nuttig kon zijn voor onze opgave. Zo kan het probleem van dubbel ingeladen nodes bijvoorbeeld vermeden worden door bepaalde acties (nodes) te maskeren zodra ze reeds geselecteerd zijn. `MaskablePPO` bleek de meest geschikte policy-optimalisatiemethode die dit mechanisme ondersteunt. Voor extra uitleg over action masking verwijzen we naar Sectie 2.1.4.

`MaskablePPO` bevindt zich in de `contrib`-versie van de *Stable-Baselines3* library. De `contrib`-module (voluit: `sb3-contrib` [41]) bevat experimentele of geavanceerde uitbreidingen van de hoofd-`SB3` library die (nog) niet officieel opgenomen zijn in de standaardversie. Deze uitbreidingen zijn bedoeld voor meer gespecialiseerde toepassingen, zoals ondersteuning voor action masking. Hoewel de algoritmes in `sb3-contrib` doorgaans minder uitgebreid getest zijn dan die in de hoofdmodule, worden ze actief onderhouden door de community en zijn ze goed inzetbaar voor onderzoeksdoeleinden zoals de onze.

3.2.1 Custom Policy

Net zoals bij `Gymnasium` environments biedt SB3 de mogelijkheid om de interne werking van policy-optimalisatie aan te passen. Dit gebeurt via het definiëren van een `CustomPolicy`, waarin onder andere de architectuur van het neurale netwerk, de feature extractors en de actor- en critic-heads aangepast kunnen worden. Voor een schematisch overzicht van zulke policy wordt verwezen naar Figuur 3.4.



Figuur 3.4: Schematische voorstelling van hoe een policy network eruitziet in Stable Baselines 3. Voor het PPO-algoritme zal zowel het actor network als het critic network deze vorm aannemen, waar ze beiden door dezelfde feature extractor gestuurd worden. (Afbeelding overgenomen van [6])

Feature extractors worden toegepast op de observation space voordat deze aan de policy-netwerken wordt doorgegeven. De standaard feature extractor is de `FlattenExtractor`, die geneste structuren (zoals dictionaries of arrays) platdrukt tot één vector. Deze platte vector vormt vervolgens de input van de netwerken.

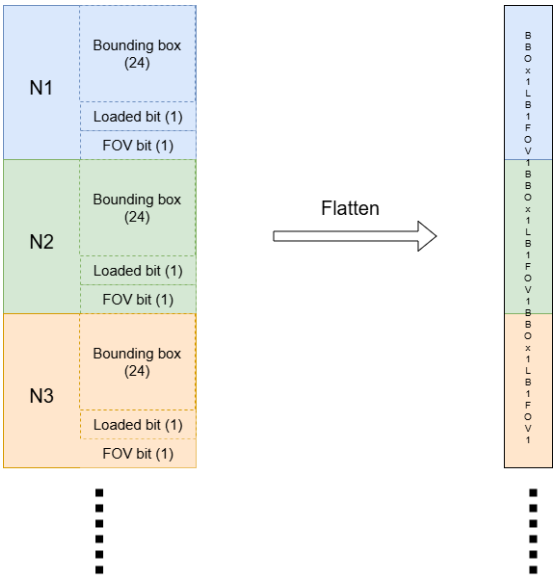
Daarnaast is het mogelijk om het actor- en critic-netwerk aan te passen (in SB3 respectievelijk `policy_net` en `value_net` genoemd). Dit kan op verschillende manieren: door de `net_arch`-parameter van het model in te vullen bij het instantiëren van het model, of door een aangepaste policyklasse te schrijven. In een custom policy kunnen zowel het netwerk als de feature extractor volledig overschreven worden. Voor specifieke instructies verwijzen we naar de documentatie van SB3, en in het bijzonder naar de webpagina over custom policies [6].

De standaard policy-architectuur bestaat uit een inputlaag die overeenkomt met de dimensie van de feature vector, gevolgd door twee volledig verbonden (*fully connected*) verborgen lagen met telkens 64 neuronen, en een outputlaag. Bij het actor-netwerk produceert de outputlaag zoveel logits als er acties in de action space zijn (in ons geval het aantal glTF-nodes). Bij het critic-netwerk bestaat de output uit één enkele waarde, die de verwachte cumulatieve reward representeert gegeven de huidige observatie.

De activatiefuncties tussen de lineaire lagen zijn standaard `Tanh`-functies. De afwisseling tussen lineaire transformaties en niet-lineaire activaties zoals `Tanh` geeft het netwerk het vermogen om complexe niet-lineaire relaties in de data te modelleren. Deze keuze is algemeen toepasbaar en maakt het netwerk flexibel inzetbaar, maar is niet specifiek afgestemd op de karakteristieken van onze taak.

3.2.2 PerNodePolicy

Tijdens het trainen van onze agent met een willekeurige volgorde van nodes tussen episodes, bedoeld om correlaties tussen specifieke node-indices en hun relevantie te verminderen, observeerden we dat de agent moeite had om effectief te leren onder deze omstandigheden. Een mogelijke verklaring hiervoor is het gebruik van de standaard feature extractor, die de gehele observatieruimte eendimensionaal maakt door middel van een flatten-operatie. Hierdoor worden alle individuele node-informatie en structuur gecomprimeerd tot één enkele vector, wat het vermogen van het model om onderscheid te maken tussen verschillende nodes mogelijk beperkt (zie Figuur 3.5).



Figuur 3.5: Visualisatie van hoe de flatten operatie alle nodedata platdrukt en het onderscheiden van nodes vermoeilijkt voor het model.

Ter bevordering van een betere representatie hebben we een aangepaste policy ontwikkeld, genaamd *PerNodePolicy*, welke expliciet is ontworpen om nodes individueel te verwerken in plaats van alle nodes te combineren tot één vector. Deze implementatie, terug te vinden in de `custom_rl` module binnen de Python-map van het project, maakt gebruik van een *PerNodeFeatureExtractor* die per node een vector van kenmerken extraheert via een dedicated fully-connected neuraal netwerk. De zodoende verkregen features worden per node afzonderlijk geëxtraheerd met een uitgebreidere embedding en bijgehouden. Het actor-netwerk doorloopt vervolgens iedere node afzonderlijk om de corresponderende actie-logits te berekenen. Deze aanpak beoogt een index-onafhankelijke representatie waarbij de policy enkel op basis van observaties de optimale nodes leert selecteren. Het critic-netwerk daarentegen werkt met een geaggregeerde, gevlakte representatie van alle node-features, waarbij de vectorlengte is aangepast aan het aantal features, om zo een betere verwerking van langere feature-vectoren mogelijk te maken.

Hoewel deze methode potentieel bood, presteerde de PerNodePolicy in de huidige configuratie niet bevredigend. De leerprogressie van de agent bleef vrijwel uit, met instabiele prestaties over opeenvolgende episodes. Door tijdsbeperkingen konden we geen verdere experimenten of optimalisaties uitvoeren. Echter sluit deze bevinding niet uit dat de sleutel ligt in het effectief decorreleren van node-indices en het leren van representaties die uitsluitend gebaseerd zijn op omgevingsdata. De PerNodePolicy is semantisch correct geïmplementeerd en biedt een flexibele basis om toekomstige configuraties experimenteel te onderzoeken, om een geschikte policy-architectuur voor deze probleemstelling te vinden.

Om bovengenoemde redenen is ervoor gekozen om te blijven werken met de standaard PPO-implementatie, waarbij enkel de `FlattenExtractor` wordt gebruikt en zowel het actor- als critic-netwerk bestaat uit twee volledig verbonden verborgen lagen met elk 64 neuronen. Verdere studie naar optimale netwerkarchitecturen kan aanbevolen worden als vervolgonderzoek.

3.3 WebSocket-communicatie

WebSockets zijn een netwerkprotocol dat het mogelijk maakt om in twee richtingen, persistent en real-time data uit te wisselen tussen een client en een server via één enkele TCP-verbinding. In tegenstelling tot traditionele HTTP-requests, waarbij de client telkens een nieuwe verbinding

moet opzetten voor elke request, laat een WebSocket-verbinding toe om voortdurend berichten te versturen en ontvangen zonder extra overhead. Dit maakt het protocol bijzonder geschikt voor interactieve toepassingen zoals live datafeeds, chatapplicaties en in ons geval: continue communicatie tussen een renderer en reinforcement learning-agent.

De communicatie tussen onze glTF-environment en de renderer (zoals het renderen van sequenties in de `step`-functie, het omzetten van bounding boxes en het berekenen van de FOV-bit in de `reset`-functie, etc.) verloopt via een client-servermodel. Binnen dit model fungeert de environment als client die requests verstuurt, terwijl de renderer deze requests ontvangt, verwerkt en een resultaat terugstuurt. Een belangrijke technische beperking hierbij is dat het in een browseromgeving niet mogelijk is om een WebSocket-server op te zetten die inkomende connecties accepteert. Browsers kunnen enkel fungeren als WebSocket-client en dus zelf een verbinding initiëren met een externe WebSocket-server. Daarom moet de servercomponent van de communicatie zich buiten de browser bevinden, bijvoorbeeld in een Node.js- of Python-omgeving, zodat de renderer als client kan blijven functioneren. Om dit probleem te omzeilen werd een aparte WebSocket-server opgezet tussen de environment en de renderer. Deze server accepteert inkomende verbindingen van zowel de RL-environment als de renderer, en fungeert louter als relay-server: hij stuurt berichten die van de ene partij binnenkomen automatisch door naar de andere. Op deze manier vormt de server een tussenlaag die toelaat om browser-gebaseerde communicatie mogelijk te maken, ondanks de beperking dat een WebSocket-server niet rechtstreeks in de browser kan worden opgezet.

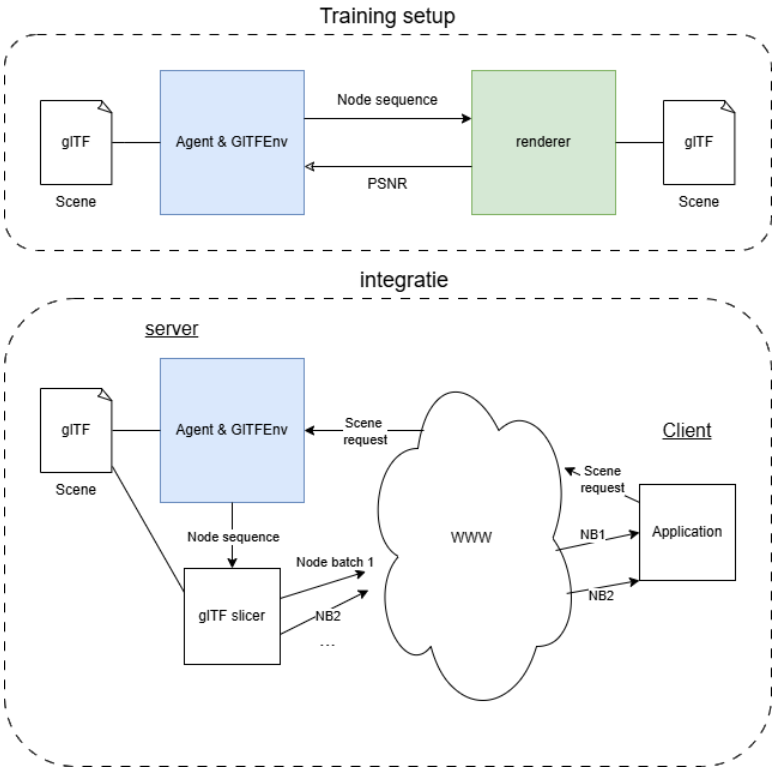
Een tweede technisch obstakel deed zich voor in de `step`- en `reset`-functies van de environment: het is niet mogelijk om hierin rechtstreeks asynchrone WebSocket-communicatie te gebruiken, aangezien de architectuur van Gymnasium dergelijke `async`-definities niet toelaat. Het wachten op inkomende Websocket messages maakt een functie echter per definitie asynchroon, wat in strijd is met de restricties van Gymnasium. Om dit op te lossen is de code van de glTF-environment opgesplitst in twee threads: een hoofdthread die verantwoordelijk is voor het trainen, testen en uitvoeren van de agent, en een tweede thread die instaat voor de WebSocket-communicatie. De communicatie tussen deze twee threads gebeurt via een *queue*-mechanisme, een wachtrij die een First-In-First-Out (FIFO)-model hanteert om berichten tussen de threads te verzenden. Bij de start van het programma worden een send-queue en een receive-queue geïnitialiseerd. De WebSocket-thread wacht telkens op een bericht in de send-queue, stuurt dit door naar de WebSocket-server, en plaatst bij ontvangst van een antwoord het resultaat in de receive-queue. De hoofdthread kan zo eenvoudig een bericht plaatsen in de verzend-queue en vervolgens wachten op een antwoord in de ontvangst-queue, zonder zelf asynchrone logica te vereisen.

De sequenties die via de server naar de renderer worden gestuurd, bevatten steeds de effectieve node-namen in plaats van de index van de node in de glTF-scene. Hoewel dat laatste op het eerste gezicht voldoende lijkt, merkten we dat de indices of volgorde van de nodes aan de kant van de renderer soms verschilden van die aan de kant van de environment. Dit verschil is mogelijk te verklaren doordat het parsen van het glTF-2.0-bestand aan beide zijden gebeurt met verschillende libraries: in Python met *pygltflib* [29], en in JavaScript met *Three.js* [5].

3.4 Renderer via Three.js

Om het model te trainen en toe te passen, hebben we een module nodig die bekomen sequenties van meshes kan renderen en hier een PSNR-score aan kan toekennen. Hiervoor werd gebruikgemaakt van Three.js [5], een 3D-graphicsbibliotheek gebouwd bovenop WebGL, die een intuïtieve interface biedt om grafische scènes te renderen, manipuleren en definiëren in de browser. Three.js ondersteunt bovendien rechtstreeks het glTF 2.0-formaat, dat in deze thesis gebruikt wordt. Ook in het werk waarop deze thesis verder bouwt [28], werd Three.js gebruikt als rendering-engine.

De renderer fungeert niet als een client die verbinding maakt met een service waarin het RL-



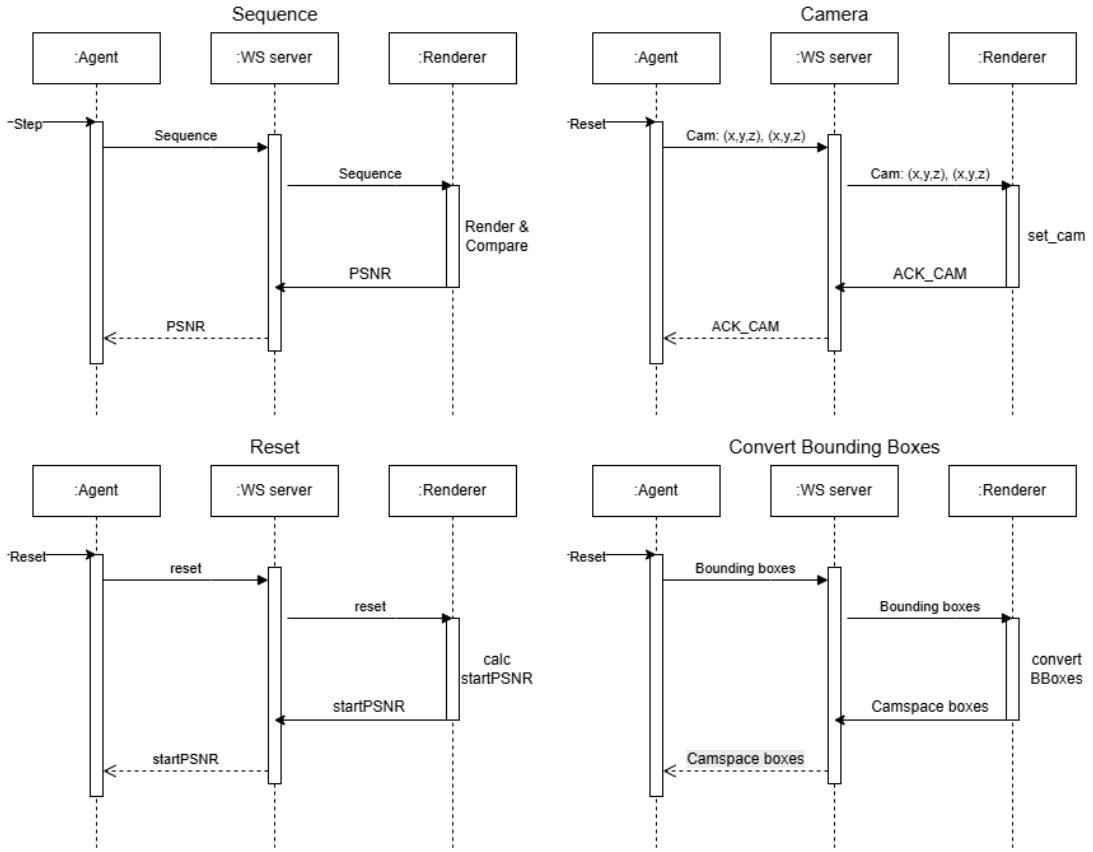
Figuur 3.6: Schematische voorstelling van hoe de systeemarchitectuur van de renderer verschilt van wanneer dit model door een clientapplicatie gebruikt zou worden. In dit geval zal het nodig zijn een extra glTF-slicing en verzend component te koppenlen omdat deze client geen toegang heeft tot het glTF bestand.

model wordt toegepast, maar als een render training engine. In deze implementatie is het volledige glTF-bestand zowel aan de kant van de agent als aan de kant van de renderer beschikbaar. Via de WebSocket-interface worden enkel identifiers doorgestuurd van de corresponderende nodes of meshes die gerenderd moeten worden. Het doel van deze renderer is om het trainen van de agent te ondersteunen, waarbij de agent via zijn policy tracht een optimale volgorde van meshes te bepalen. Om dit model in een realistische toepassing te integreren, zal een extra component nodig zijn die de gegenereerde volgorde van identifiers aanneemt en op basis daarvan de bijhorende meshes naar de client streamt door de glTF file te slicen en de correcte nodes stuurt naar de client (zie Figuur 3.6 voor een verduidelijking van dit scenario). Ook de React-versie van de renderer fungeert niet als een client: deze zal achterliggend ook het gehele gLTF-bestand ter beschikking hebben en dient louter voor resultaten te visualiseren.

3.4.1 WebSocket-protocol

De rendering-engine wordt aangestuurd via WebSocket-berichten afkomstig van onze glTF-environment, waarin de agent zijn model traint of toepast. Om deze processen te ondersteunen, moet de renderer verschillende types berichten kunnen verwerken: (zie Figuur 3.7 voor schematische weergave)

- **node sequence**:
  De renderer ontvangt een sequentie van in te laden nodes of meshes, rendert deze, en berekent vervolgens een PSNR-score. Deze score wordt via de WebSocket teruggestuurd naar de agent.
- **reset**:
  Bij het ontvangen van een reset-bericht moet de renderer alle momenteel gerenderde



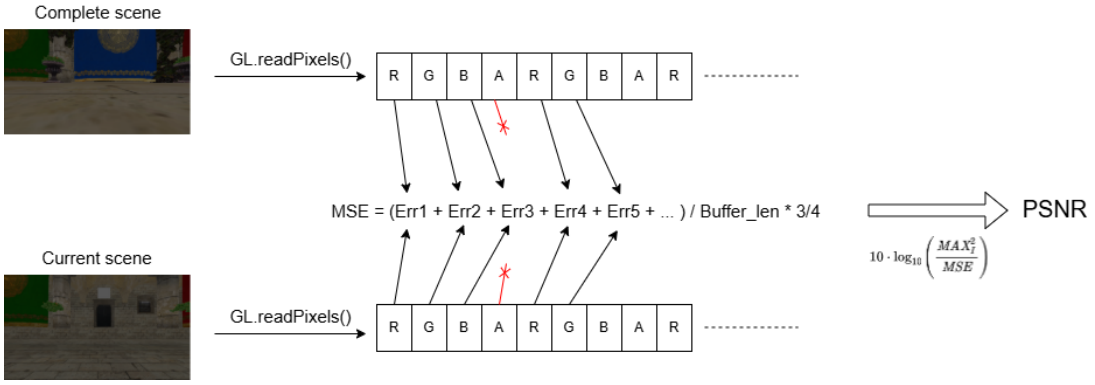
Figuur 3.7: Schematische voorstelling van WebSocketscommunicatie tussen agent en renderer over de WebSocket server heen voor de verschillende types communicatie.

nodes uit de sequentie verwijderen, zodat enkel de achtergrond zichtbaar blijft. Van deze achtergrond moet eveneens een PSNR-score berekend worden, die naar de agent wordt verzonden als *startPSNR*.

- **camera**:
  Aan het begin van een nieuwe episode kan het camerastandpunt gewijzigd worden om variatie in het trainingsproces te introduceren. De renderer ontvangt hierbij de positie en kijkrichting van de camera. Voorlopig wordt aangenomen dat de camera rechtop staat en dus niet om zijn as is geroteerd.

- **frustum culling**:
  Voor het opbouwen van de observation space van het RL-model is per mesh een bit vereist die aangeeft of de bounding box van die mesh het viewing frustum snijdt. Dankzij ingebouwde ondersteuning in Three.js kan dit eenvoudig bepaald worden tijdens het renderen. Bij ontvangst van dit bericht antwoordt de renderer met een dictionary waarin per node-naam de overeenkomstige bit is opgenomen.

- **bounding boxes**:
  Wanneer de renderer een bounding box request krijgt, dient het een dictionary samen te stellen van alle nodenames, gekoppeld aan hun bounding box in camera space. Deze functionaliteit is nodig voor de preprocessing van de data die aangeleverd wordt aan de agent zoals beschreven in Sectie 3.5. Dit wordt gedaan om het effect van de camerapositie en kijkrichting impliciet te maken in de data zodat dit niet door de agent gecorreleerd moet worden.



Figuur 3.8: Schematische voorstelling van hoe de PSNR van een subresultaat van een sequence berekent wordt.

3.4.2 PSNR-berekening

Zoals eerder aangegeven zal er door de renderer een PSNR-score berekend worden. Dit gebeurt door het huidige beeld van de scène, bekomen door het renderen van een sequentie van nodes, te vergelijken met het volledige beeld van de scène, bekeken vanuit het huidige camerastandpunt. In Three.js kunnen meerdere scène-objecten simultaan bijgehouden worden, die na elkaar of parallel gerenderd kunnen worden. Rendering in Three.js gebeurt via de `THREE.WebGLRenderer`, die toegang heeft tot de onderliggende functionaliteit van WebGL. Hierdoor kunnen de pixelwaarden van het gerenderde beeld opgevraagd worden via de `readPixels`-methode van WebGL. Deze methode levert de RGBA-waarden van de momenteel gerenderde framebuffer op.

Om de PSNR van een sequentie te berekenen, wordt eerst de volledige scène gerenderd, waarvan de pixelwaarden worden opgeslagen in `buffer1`. Vervolgens wordt het beeld gegenereerd dat overeenkomt met de opgegeven node-sequentie, waarvan de pixelwaarden in `buffer2` terechtkomen. Op basis van deze twee buffers wordt de MSE (Mean Squared Error) berekend door beide buffers te doorlopen en het verschil tussen de corresponderende pixelwaarden te bepalen. De alfa-waarden werden bij deze berekening systematisch genegeerd. De training werd immers hoofdzakelijk uitgevoerd op de Sponza-scène, waarin alle meshes volledig ondoorzichtig zijn en de alfa-waarden dus steeds gelijk zijn. Het meenemen van de alfa-component zou daardoor artificieel een 1/4 overeenstemming opleveren tussen beide buffers, wat de focus van het model op structurele verschillen in RGB verstoort. Een schematische weergave van deze PSNR berekening is te vinden bij Figuur 3.8. Indien de alfa-component in toekomstige toepassingen wél relevant zou zijn, kan de MSE-functie eenvoudig aangepast worden door ook de alfa-waarden mee te nemen en de buffergrootte hierop af te stemmen. De bekomen MSE wordt vervolgens omgezet naar een PSNR-waarde zoals beschreven in Sectie 2.4.1.

3.4.3 Frustum culling

Hiernaast wordt, zoals hierboven vermeld, een *frustum culling* uitgevoerd op alle nodes om een dictionary van bits te bekomen die aanduiden of de bounding box van de bijhorende node van de key (node-identifiers) het view frustum snijdt. Het **view frustum** is een afgeplat kegelvormig volume dat vertrekt vanuit de lens van de camera tot aan de *far-plane* (de maximaal zichtbare afstand), en omvat dus alles wat vanuit de camera zichtbaar is.

Zonder ondersteuning kan het berekenen van deze intersecties relatief complex zijn, maar dankzij de interface van Three.js is dit eenvoudig op te lossen via ingebouwde matrixtransformaties. Om te bepalen welke nodes het frustum snijden, volgen we deze stappen:

1. **Berekening van de view-projection matrix:**
   We vermenigvuldigen de `projectionMatrix` met de `matrixWorldInverse` van de camera.

De viewmatrix (`matrixWorldInverse`) zet wereldcoördinaten om naar cameracoördinaten, terwijl de `projectionMatrix` cameracoördinaten omzet naar clip-space. De combinatie ervan transformeert dus rechtstreeks van worldspace naar clip-space.

2. **Initialisatie van het frustum:**
   In Three.js kunnen we een `Frustum`-object aanmaken en initialiseren met deze samengestelde matrix via `frustum.setFromProjectionMatrix(...)`.

3. **Berekenen van bounding boxes:**
   We itereren over alle nodes. In Three.js kan een axis-aligned `Box3`-object rechtstreeks geïnitialiseerd worden op basis van een glTF-node. Dit object stelt de bounding box van die node voor.

4. **Intersectietest:**
   Voor elk van deze bounding boxes gebruiken we de methode `frustum.intersectsBox(...)` om na te gaan of de box (deels) binnen het frustum valt.

3.4.4 Convert bounding boxes

Een andere taak die in de renderer geïmplementeerd werd, is het converteren van bounding boxes die behoren tot de nodes in de scene van *world space* naar *camera space*. Deze omzetting is noodzakelijk in de preprocessingstap van de data die aan de RL-agent wordt aangeboden, om optimalisatieredenen die verder worden toegelicht in Sectie 4.3.4. Net zoals bij de frustum culling (Sectie 3.4.3) wordt hiervoor gebruikgemaakt van de *view matrix* (`matrixWorldInverse`) om coördinaten van world space naar camera space te transformeren.

In tegenstelling tot bij frustum culling, zijn in deze context geen verdere clippingoperaties vereist. Een belangrijk geometrisch inzicht hierbij is dat een axis-aligned bounding box (AABB) na een dergelijke transformatie naar camera space niet langer gealigneerd is met het assenstelsel. In camera space kan de box namelijk geroteerd zijn, waardoor de representatie met slechts twee uiterste hoekpunten (zoals in een AABB) niet langer volstaat. Daarom wordt in de implementatie gekozen om expliciet alle acht hoekpunten van de box te berekenen. Hoewel dit redundant is voor een volledige geometrische reconstructie (aangezien een box door minder punten uniek bepaald kan worden), biedt het volledigheid en eenvoud bij verdere verwerking.

De implementatie verloopt volgens volgende stappen:

1. **Bepalen van de AABB:**
   Voor elke node in de glTF-scene wordt een axis-aligned bounding box berekend via `THREE.Box3().setFromObject(node)`. Dit object biedt toegang tot de minimale en maximale coördinaten in de x-, y- en z-richting. Op basis van deze twee punten worden vervolgens de acht hoekpunten van de box expliciet samengesteld.

2. **Transformatie naar camera space:**
   De matrix `camera.matrixWorldInverse` wordt gekopieerd in een nieuwe `THREE.Matrix4`-instantie. Deze matrix zet coördinaten om van world space naar camera space. Elk van de acht hoekpunten wordt getransformeerd met behulp van `corner.clone().applyMatrix4(cameraMatrix)`.

3. **Opbouw van de datastructuur:**
   Voor elke node wordt een entry toegevoegd aan een dictionary, waarbij de key overeenkomt met de nodenaam (`node.userData.name`) en de value een lijst bevat van de getransformeerde hoekpunten in camera space. Deze dictionary wordt later omgezet naar JSON-formaat en verzonden over de WebSocket-verbinding naar de RL-agent.

3.4.5 testtools

Buiten deze functionaliteit, die essentieel is voor de werking van onze RL-agent training, zijn er in de `React`-implementatie van de renderer ook nog enkele handige testtools aanwezig. Dit

omdat de ontwikkeling van het project en het analyseren van de resultaten voornamelijk gebeurde via de `React` renderer, terwijl het daadwerkelijke trainingswerk hoofdzakelijk via de `Puppeteer`-versie verliep. Deze testtools zijn nuttig om onder andere de werking van PSNR te onderzoeken, visueel te inspecteren hoe een render sequence er daadwerkelijk uitziet door het doorlopen van sequenties, of om na te gaan welke node-index met welke mesh overeenkomt. De gebruikersinterface van deze testtools laat voorlopig echter nog te wensen over: in plaats van invoervelden in de applicatie te voorzien, werden tijdens het inspecteren van sequences deze vaak rechtstreeks in de broncode van de pagina ingevoerd. Met wat extra werk aan de UI zouden deze testtools enkele handige functionaliteiten kunnen bieden om sequences efficiënt te debuggen:

- **Node Explorer:**
  Nodes of meshes worden in het project doorgestuurd aan de hand van hun node-naam (bijv. `Mesh_2.01`) of hun index in de lijst van meshes. Wel opletten dat indices tussen de environment en renderer kunnen verschillen, dus is het hier van belang dat deze overeenkomen met de versie van de renderer. Hoewel dit duidelijk en identificeerbaar is voor de renderer, is het voor de gebruiker moeilijk om in te schatten welke mesh hiermee bedoeld wordt. In de *Node Explorer* begint men in een lege scene en kan men via index of node-naam een mesh toevoegen om visueel te zien welke mesh dit betreft. Bovenaan wordt steeds de huidige PSNR-score weergegeven van de verzameling momenteel zichtbare nodes.
- **Sequence Viewer:**
  Via deze tool kan een sequentie van node-indices of node-namen (gemakkelijk te kopiëren uit een print van het model) ingevoerd worden om deze stapsgewijs te doorlopen. Met de toetsen `f` (forward) en `b` (backward) kan de sequentie manueel worden afgespeeld. Ook hier wordt telkens bovenaan de PSNR weergegeven, zodat het effect van bepaalde meshes op deze score bekeken kan worden. Momenteel moeten de sequentie en het camerastandpunt nog rechtstreeks in de code ingegeven worden.

Deze tools zijn niet volledig uitgewerkt, aangezien ze organisch gegroeid zijn tijdens de ontwikkeling voor persoonlijk gebruik, en daardoor geen prioriteit vormden voor deze thesis. Met wat extra werk aan de UI en enkele uitbreidingen zouden ze echter een waardevolle debugtool kunnen vormen voor soortgelijke projecten.

3.5 Dataverzameling en Preprocessing

Preprocessing is in de context van Reinforcement Learning en daarbij machine learning in het algemeen cruciaal voor een goede performantie te behalen. Het zorgt ervoor dat ruwe of inconsistente data wordt omgezet in een gestandaardiseerd en informatief formaat dat het leerproces van het model bevordert. Door irrelevante of redundante informatie te verwijderen en belangrijke kenmerken expliciet te maken, wordt niet alleen het leerproces efficiënter, maar ook stabieler. In het bijzonder bij 3D-scenes, zoals in deze thesis, helpt preprocessing om complexe ruimtelijke informatie — zoals bounding boxes, zichtbaarheid en afstanden tot de camera — op een consistente manier aan te bieden aan het RL-model. Zonder deze verwerking zou de agent zelf verantwoordelijk zijn voor het afleiden van deze ruimtelijke relaties, wat resulteert in een veel moeilijkere leeropgave en slechtere generalisatie.

Voor deze thesis doen we onderzoek naar glTF 2.0 (Graphics Library Transmission Format) data. Dit is een bestandstype dat faciliteerd om efficiënt 3D-scenes voor te stellen en te verzenden over het internet door de hiërarchische structuur die het omvat. Een uitgebreidere uitleg over het glTF format werd reeds gegeven in Sectie 2.2.2. In het bijzonder werd steeds gebruik gemaakt van de Sponza-scene, een alombekende testscene binnen de literatuur rond computer graphics. Wanneer men uit zou willen breiden naar verschillende glTF-scenes zijn deze breed beschikbaar op het internet of zelf te construeren via een graphics applicatie zoals Blender [4]. Deze applicatie werd in een initieel stadium van het project gebruikt om zelf simpele scenes te

genereren wanneer de RL-agent moeite had met de omvang van de Sponza scene.

Tenvorens deze data klaar is om aan het RL-framework beschikbaar te stellen moet deze eerst geparsed worden. Hiervoor werd een parse_gltf module gemaakt in Python, die met de hulp van pygltflib [29], een library gespicificeerd op het verwerken, aanpassen en creëren van glTF 2.0 data. Deze module diende de gewenste data van iedere node in de glTF scene te extraheren, verwerken en te ordenen volgens de conventies van het RL-framework. Er wordt een geordende lijst opgesteld met de volgende elementen per node.

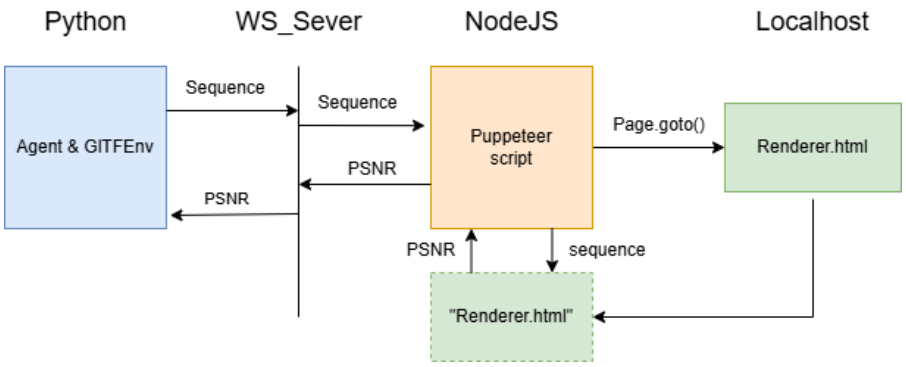
- **Node name:**
  Zoals in eerdere secties aan bod is gekomen worden doorheen het project nodes geïdentificeerd volgens de bijhorende name. Typisch wordt in glTF-data gebruik gemaakt van unieke namen van nodes, al is dit niet per implementatie verplicht. Dit was ook het geval van de Sponza scene. Wanneer dit niet het geval zou zijn kan dit met aanpassingen aan de implementaties van het project omgevormd worden naar hun daadwerkelijke identifiers. Dit is in functie van de leesbaarheid van de resultaten niet gedaan.
- **Node index:**
  Positie van de node in de glTF file. Gebruikt in de interne implementaties van het project.
- **Bounding box:**
  Per node zal de axis aligned bounding box (AABB) uit de glTF-data geëxtraheerd worden. Deze zullen uiteindelijk verder omgevormd worden naar cameraspace-boundingboxes in een volgende preprocessing stap.
- **Distance to Camera:**
  Aan de hand van hun AABBs, zal voor iedere node zijn afstand tot de camera berekend worden. Dit door het verschil te berekenen van het standpunt van de camera en het centrum van de bounding box van de node.

Een belangrijke opmerking bij het berekenen van de bounding boxes, is dat deze beschreven wordt in object-coördinaten in de bijhorende node-data. Deze omschrijven de bounding boxes van de node wanneer deze op de oorsprong van het assenstelsel gepositioneerd is. Op deze node dienen nog zijn rotatie en translatie die ook te vinden zijn in zijn object toegepast te worden. Deze fout werd in de beginfase gemaakt van het project en slechts pas op een later moment opgemerkt wanneer de bounding boxes en bijgevolg de distances naar de camera bij nadere inspectie onjuist leken.

Deze set aan data wordt in verschillende fasen van de trainingsloop gebruikt en zal uiteindelijk ook gebruikt worden om de observatie van de agent te vormen, waarmee deze zijn volgende actie zal selecteren. Specifiek zullen de bounding boxes van de nodes van worldspace naar cameraspace omgevormd worden door de renderer, zoals beschreven in Sectie 3.4.4 alsook zal een FOV-bit berekend worden per node die beschrijft of deze zijn bounding box intersecteerd met het view frustum (of deze in het beeld van de camera ligt) en zal een bit gebruikt worden om aan te geven of bijhorende node reeds via een vorige actie ingeladen is. De combinatie van deze data per node zal de observatie van onze agent vormen. Deze wordt alvorens het betreden van het policy-network van de agent door een feature extractor gestuurd. In de huidige staat van het project zal deze extractor louter een flatten-operatie op deze vector uitvoeren.

3.6 Puppeteer-module

Naast de renderer die geïmplementeerd is in *React*, bevat dit project ook een alternatieve renderer gebaseerd op *Puppeteer*. *Puppeteer* is een JavaScript-bibliotheek die in de eerste plaats bedoeld is om een browser (specifiek Chromium) *headless* aan te sturen. Dit houdt in dat de browser niet visueel wordt geopend, maar in plaats daarvan via een Node.js-proces in de terminal wordt uitgevoerd. Op die manier kan een webpagina automatisch geladen en gemanipuleerd worden zonder tussenkomst van een grafische gebruikersinterface. *Puppeteer* biedt uitgebreide mogelijkheden voor het automatiseren van interacties met webpagina's. Zo kan het



Figuur 3.9: Schematische voorstelling van hoe de environment met puppeteer interageert om een PSNR score van een nodesequentie te verkrijgen.

programma via de DOM interageren met elementen zoals knoppen, invoervelden en menu's. Veelvoorkomende toepassingen zijn onder andere web scraping, automatische formulierverwerking, visuele regressietesten, en monitoring van websites. De tool wordt eveneens gebruikt om PDF's of schermafbeeldingen van webpagina's te genereren, of om de prestaties van een website automatisch te evalueren met behulp van Lighthouse-integratie.

Hoewel browserautomatisering soms wordt geassocieerd met controversiële toepassingen, zoals het automatisch klikken op advertenties, kunstmatig verhogen van weergaves of interacties op sociale media, kent het ook tal van legitieme en maatschappelijk waardevolle gebruikstoepassingen. Enkele voorbeelden zijn:

- Automatisch genereren van toegankelijke versies van dynamische websites.
- Testen van gebruikersinterfaces op consistentie en toegankelijkheid (bijv. *a11y*-checks, een industry-gedeclareerde accesibility standaard [46])
- Verzamelen van publieke gegevens voor wetenschappelijk onderzoek of journalistieke doeleinden.

Puppeteer hoeft in feite ook niet *headless* gerund te worden; het kan eveneens in *headful* modus gebruikt worden, waarbij er wel een visuele component aanwezig is. Dit laat toe om visuele interacties met webpagina's te automatiseren, zoals het uitvoeren van visuele regressietests, het opnemen van gebruikerssessies, of het demonstreren van een applicatie in een realistische gebruikersomgeving. Voor meer algemene informatie verwijzen we naar de website van het puppeteer project [45]. Voor een schematisch voorbeeld van de structuur van deze puppeteer module zie Figuur 3.9.

3.6.1 Use-case

In het kader van dit project wordt *Puppeteer* ingezet als autonome renderer die geschikt is voor gebruik in een batch- of serveromgeving. Voor deze thesis werd gebruikgemaakt van rekeninfrastructuur van het Vlaams Supercomputer Centrum (VSC), waarbij toegang werd verleend tot verschillende compute-clusters voor het uitvoeren van trainingssessies. Meer informatie over het VSC is te vinden op de website [1]. Aangezien deze trainingen via het Slurm-taakbeheersysteem moesten worden uitgevoerd, was het niet mogelijk om een fysieke browser te gebruiken. Puppeteer bood in dit geval een geschikte oplossing door het toelaten van *headless* rendering, waardoor de trainingspipeline alsnog in een niet-interactieve omgeving kon worden uitgevoerd.

3.6.2 Implementatie

Voor de Puppeteer-versie van de renderer werd een vereenvoudigde variant van de *React*-renderer ontwikkeld. In tegenstelling tot de *React*-versie, die interactieve functionaliteiten biedt

zoals het bekijken van sequenties of het inspecteren van individuele meshes binnen scènes, richt deze implementatie zich uitsluitend op het automatisch renderen van frames en het berekenen van de bijbehorende PSNR-score. Daarom werd gekozen voor een implementatie in *plain HTML/JavaScript*, waarbij alle debugtools en gebruikersinterfaces werden weggelaten.

Om de training via Puppeteer correct te laten verlopen, is het noodzakelijk dat de rendererpagina over een functie beschikt die als *hook* kan fungeren voor het afhandelen van control messages (zoals sequenties, resets, camerastandpunten, enz.). Deze berichten zijn inhoudelijk identiek aan die van de *React*-renderer, zoals besproken in Sectie 3.4. De centrale functie die deze berichten verwerkt, werd gedefinieerd als `handleAction()`. De HTML/JS-renderer wordt lokaal gehost via een eenvoudige *Express*-server [8]. Zodra deze renderer beschikbaar is, kan een script geschreven worden dat via Puppeteer de nodige interacties uitvoert. Dat script wordt binnen dit project aangeduid als `puppeteer_script.js`. De implementatie verloopt in de volgende stappen:

1. **Launch**: Via `puppeteer.launch()` wordt een instantie van de Chromium-browser gestart. Parameters zoals *headless mode*, schermresolutie en andere configuraties kunnen hier gespecificeerd worden. Het resultaat wordt opgeslagen als een `browser`-object.
2. **newPage**: Met `browser.newPage()` wordt een nieuw virtueel tabblad geopend, dat bijgehouden wordt als een `page`-object.
3. **goto**: Via `page.goto()` wordt een webpagina geladen. In dit project verwijst dit naar de lokale instantie van de HTML-renderer via `localhost` en de corresponderende poort.
4. **evaluate**: Met `page.evaluate()` kunnen functies worden uitgevoerd binnen de context van de geladen pagina. Door de eerder gedefinieerde `handleAction()`-functie aan te roepen met berichten afkomstig van de WebSocket-server, kan de renderer geautomatiseerd bestuurd worden.

Om communicatie met de trainingsomgeving mogelijk te maken, onderhoudt het Puppeteerscript een actieve WebSocket-verbinding met de WebSocket-server. Binnen de handlerfunctie worden binnenkomende berichten geparsed en vervolgens doorgegeven aan de renderer via `page.evaluate()`. Een belangrijk aandachtspunt is dat het Puppeteer-script actief moet blijven om inkomende berichten te blijven verwerken. Zonder bijkomende maatregelen zou het script onmiddellijk worden afgesloten na initialisatie van de WebSocket en puppeteer. Als oplossing wordt aan het einde van het script `process.stdin.resume()` aangeroepen. Dit zorgt ervoor dat het proces actief blijft tot een expliciete onderbreking wordt uitgevoerd.

Hoofdstuk 4

Technische keuzes

Om de onderzoeksvraag kort te herhalen: we trachten op experimentele wijze een systeem te ontwikkelen dat voor een gegeven 3D-scène een optimale strategie bepaalt om deze scène progressief te streamen en renderen. Optimaal betekent in deze context dat de gebruiker zo snel mogelijk een zo volledig mogelijk visueel beeld van de scène krijgt, zonder dat alle fragmenten of nodes/meshes al ingeladen zijn.

Nu de theoretische achtergrond en de modules van het project in de vorige hoofdstukken besproken zijn, kunnen we onderzoeken hoe een RL-agent optimaal getraind kan worden om voor een glTF-scène een zo gunstig mogelijke sequentie van nodes te voorspellen. Deze sequentie moet ervoor zorgen dat tijdens het inladen zo snel mogelijk een representatief beeld van de scène wordt opgebouwd.

In dit hoofdstuk definiëren we eerst wat precies bedoeld wordt met een "gunstige sequentie" en hoe deze geëvalueerd kan worden. Vervolgens bespreken we de strategieën en methodes die gehanteerd werden om de prestaties van de agent te verbeteren. Dit doen we door eerst een beeld te scheppen van de simplistische initiële staat van het project en te beschrijven welke voornaamste veranderingen we hebben gedaan om de performantie van de agent te verbeteren.

4.1 Selectie kwaliteitsmetriek

Vooraleer we overgaan tot het trainen van een agent, is het essentieel om vast te leggen hoe de kwaliteit van een gegenereerde nodesequentie geëvalueerd zal worden. Zoals besproken in Sectie 2.4 bestaan er verschillende methodes om de kwaliteit van een gedeeltelijk gerenderd beeld objectief te beoordelen ten opzichte van een volledig referentiebeeld. In het kader van progressive streaming beoordelen we daarom de tussenresultaten van een nodesequentie via een beeldkwaliteitsmetriek. Het doel van de agent tijdens de training is dan om deze beeldkwaliteit zo snel mogelijk te maximaliseren. Met andere woorden: om met een minimaal aantal gestreamde nodes een zo accuraat mogelijk beeld van de volledige scène te reconstrueren.

Uit de besproken methodes werd in deze thesis gekozen voor PSNR (Peak Signal-to-Noise Ratio) als evaluatiemetriek voor de progressive streaming sequence. Zowel PSNR als SSIM (Structural Similarity Index) bieden mogelijkheden om objectieve beeldkwaliteit te meten, maar verschillen fundamenteel in aanpak. PSNR evalueert enkel op pixelniveau en is daarom eenvoudiger en sneller te berekenen. SSIM daarentegen maakt gebruik van perceptuele eigenschappen zoals luminantie, contrast en structurele informatie, en benadert daarmee beter de menselijke waarneming van beeldkwaliteit (zie Sectie 2.4). Beide metrieken zijn echter niet volledig onafhankelijk. Zoals aangetoond in [17] bestaat er een analytisch verband tussen PSNR en SSIM onder bepaalde degradaties, wat impliceert dat ze vergelijkbare informatie kunnen leveren, afhankelijk

van de context. PSNR is bijzonder geschikt voor het detecteren van willekeurige ruis en levert een snelle, reproduceerbare meting. SSIM biedt daarentegen meer robuustheid ten aanzien van structurele verschuivingen, wat in dynamische scènes met bewegende camera's relevanter kan zijn.

Omwille van computationele overwegingen kozen we uiteindelijk voor PSNR. Tijdens training bleek beeldvergelijking de meest intensieve stap in de 'step()'-functie van de environment. SSIM voegde daarbovenop een sterke surplus aan computationele complexiteit toe. In Tabel 4.1 vergelijken we de gemiddelde tijdsduur (in milliseconden) van een volledige 'step()' met de afzonderlijke beeldvergelijkingsstap (PSNR of SSIM). De metingen gebeurden op de React-module van het project, uitgevoerd op een persoonlijke desktop (niet op de computing cluster, al blijven de verhoudingen representatief).

Tabel 4.1: Gemiddelde tijdsduur (in milliseconden) voor een volledige stap versus enkel scoreberekening, opgesplitst per metric.

	PSNR (ms)	SSIM (ms)
Tijdsduur volledige step	15–30	30–60
Alleen scoreberekening	10–15	15–30

SSIM blijkt gemiddeld tweemaal zo traag als PSNR. Bovendien werkten we in deze thesis met een stilstaande camera tijdens de training: binnen één episode bleef de camerapositie en -oriëntatie constant. Daardoor komen typische nadelen van PSNR (zoals zware bestraffing van kleine structurele verschuivingen) minder tot uiting. In deze context levert PSNR dus een voldoende robuuste maat zonder de hoge berekeningskost van SSIM.

Op basis van deze overwegingen werd PSNR weerhouden als beeldkwaliteitsmetriek voor het trainen van onze agent.

4.2 Benchmarking

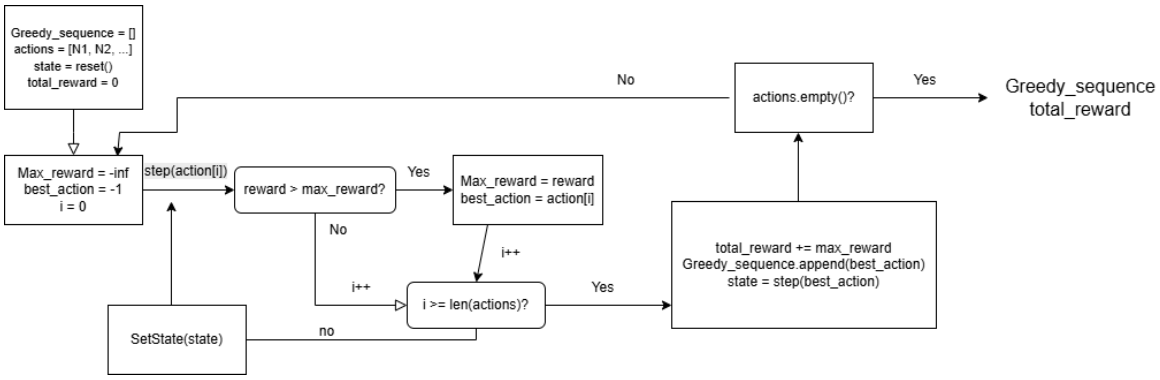
Om de prestaties van onze getrainde agent te kunnen beoordelen, is het noodzakelijk om de behaalde scores te vergelijken met referentiemethodes. Zonder een referentiekader is een kwaliteitsscore zoals PSNR moeilijk te interpreteren. In deze thesis definiëren we daarom twee benchmarks: een heuristische methode gebaseerd op eerder werk en een greedy methode die lokaal de beste PSNR-stijging nastreeft.

4.2.1 Heuristische benchmark

De eerste benchmark is gebaseerd op de methode uit [28], waarop dit onderzoek voortbouwt. In deze aanpak worden nodes geselecteerd op basis van hun afstand tot de camera. Concreet worden eerst de nodes gekozen waarvan het middelpunt zich het dichtst bij de camera bevindt. Daarbij wordt voorrang gegeven aan nodes die zich binnen het view-frustum bevinden: nodes buiten het frustum krijgen een extra penalty van 100 toegevoegd aan hun afstandsscore. een node die bijvoorbeeld op een vectorafstand van 6 zal liggen, maar buiten het frustum, zal een score van 106 krijgen. Een node die op een afstand van 50 binnen het frustum ligt krijgt geen penalty en dus behoudt deze de score van 50. Hierdoor zal deze tweede node eerder gestreamd worden. Op basis van deze geordende lijst wordt een sequentie opgebouwd.

4.2.2 Greedy benchmark

De tweede benchmark, die we zelf introduceren, is een greedy benadering. Deze methode kiest bij elke stap de node die op dat moment de grootste stijging in PSNR veroorzaakt ten opzichte van het vorige tussenresultaat. Dit gebeurt door elk van de resterende nodes afzonderlijk toe



Figuur 4.1: overzicht van het algoritme voor de berekening van de greedy sequence

te voegen aan de huidige sequentie, de nieuwe PSNR te berekenen, en vervolgens de node te selecteren die de grootste winst oplevert. Dit proces wordt herhaald tot alle nodes gekozen zijn.

Hoewel deze aanpak in elke stap een lokaal optimale keuze maakt, garandeert dit niet noodzakelijk een globaal optimale sequentie. Bijvoorbeeld, in sommige gevallen kunnen twee bedekkende nodes op de voorgrond in totaal een hogere PSNR-verbetering geven dan één grote node op de achtergrond. Als deze grote node op de achtergrond later bedekt wordt door andere, zal een groot deel van deze vroege PSNR-winst teniet doen (PSNR wordt berkent met het verschil in pixelvalues, dus gelijkaardige kleur kan PSNR ook beïnvloeden). Dit fenomeen werd ook in onze experimenten waargenomen: in bepaalde episodes kon de sequentie geproduceerd door onze getrainde agent momenteel een hogere cumulatieve PSNR-score behalen dan de greedy aanpak.

4.2.3 Implementatie

De greedy benchmark werd geïmplementeerd binnen de glTF-environment van de reinforcement learning agent. Voor elke mogelijke actie of keuze van node werd een simulatie uitgevoerd via de 'step()'-functie, waarna de PSNR-score werd geëvalueerd. Om het effect van een keuze ongedaan te maken en de environment terug in de oorspronkelijke staat te brengen, werd een aangepaste 'setState()'-methode toegevoegd aan de environment. Deze liet toe om iteratief alle opties te evalueren zonder de environment permanent te wijzigen. De node met de hoogste marge in PSNR-winst werd vervolgens definitief toegevoegd aan de sequentie, waarna het proces werd herhaald. Op deze manier konden we een robuuste en systematische greedy sequentie opbouwen die diende als sterke benchmark voor vergelijking met de resultaten van onze agent. Een diagram dat het algoritme voorstelt om de greedy sequence te berekenen wordt afgebeeld op Figuur 4.1. De heuristische benchmark werd toegevoegd door iedere node in de scene een score toe te kennen volgens zijn afstand en ligging in het FOV volgens de uitleg in Sectie 4.2.1. Dit is in feite niet meer dan een enkele optelling per node. De nodes werden hierna geordend van laagste naar hoogste score om de heuristische sequentie te bekomen.

Met deze twee benchmarks beschikken we over een praktische bovengrens en vergelijkbare benchmark voor de prestaties van ons systeem. De greedy methode vertegenwoordigt een theoretisch ideaal: hoewel deze aanpak nauwkeurig en effectief is, is de complexiteit ervan $O(n^2)$, wat betekent dat de tijd die nodig is om een volledige sequentie te genereren snel groter wordt dan de tijd die nodig is om de nodes zelf te streamen ($O(n \cdot c)$, met c de tijd om één node van server naar client te verzenden). Daarom is deze methode in de praktijk niet bruikbaar als streamingstrategie. De heuristische methode is daarentegen een haalbare benadering die gebruik maakt van eenvoudige, op afstand gebaseerde regels. Onze ambitie is om een intelligente agent te trainen die op basis van dergelijke heuristieken en bijkomende mesh-data in staat is om deze heuristische aanpak te overtreffen en zich zo tussen de heuristsche- en greedy score te

vestigen.

4.3 Ontwikkelingen

In deze sectie bespreken we de belangrijkste moeilijkheden en doorbraken die tijdens het ontwikkelen van onze omgeving, renderer en reinforcement learning-agent naar voren kwamen. Om het ontwikkelingsproces in context te plaatsen, schetsen we eerst kort de initiële opzet van deze componenten, waarna we toelichten welke verbeteringen we gerealiseerd hebben.

4.3.1 Initiële setup

glTF-environment:
De environment binnen reinforcement learning bestaat uit een aantal kerncomponenten:

- **Observation space**: de invoer die de agent ontvangt vanuit de environment. Deze bevat de informatie waarop de agent zijn beslissingen baseert.
- **Action space**: de lijst van mogelijke acties waaruit de agent kan kiezen, afhankelijk van de observatie.
- **Step-functie**: de functie die simuleert wat er gebeurt wanneer een actie wordt uitgevoerd. Ze past de toestand van de environment aan en berekent een bijhorende reward.
- **Reset-functie**: herstelt de omgeving naar de beginsituatie aan het einde van een episode.

Voor een meer diepgaande beschrijving verwijzen we naar Sectie 3.1.

In de oorspronkelijke implementatie van de glTF-environment bestond de observation space uit een verzameling opeenvolgende data-arrays die bij het omzetten naar een inputvector geflattened werden tot één lange vector. Deze arrays omvatten de volgende informatie:

1. **Axis-aligned bounding boxes (AABB)** van elke node: voorgesteld door twee uiterste hoekpunten per node (dus 6 waarden per node).
2. **Loaded bits**: een binaire array die per node aanduidt of deze reeds geladen is (de index komt overeen met de positie in de action space).
3. **Afstand tot camera**: een array die per node de afstand tussen het middelpunt van de bounding box en de camera beduidt.
4. **Camera-informatie**: een vector van 6 elementen die de positie en kijkrichting van de camera representeren.

De action space was gedefinieerd als een discrete ruimte met evenveel acties als er nodes aanwezig zijn in de glTF-scène.

De oorspronkelijke step-functie van de environment bevatte drie hoofdonderdelen:

1. **Validatie van de actie**: als de agent een node probeert in te laden die reeds geladen is, werd de timestep onmiddellijk beëindigd en kreeg de agent een negatieve reward toegekend.
2. **Uitvoeren van de actie**: de identifier van de gekozen node werd naar de renderer gestuurd, die vervolgens de bijbehorende PSNR berekende en terugstuurde.
3. **Rewardberekening**: in de initiële versie werd de reward berekend als de genormaliseerde PSNR van het huidige tussenresultaat gedeeld door het aantal verlopen timesteps.

De reset-functie zette alle interne waarden van de environment terug naar hun oorspronkelijke staat en stuurde een reset-signaal naar de renderer, die vervolgens een bevestiging (acknowledgement) teruggaf.

Renderer:
De initiële renderer hoefde slechts twee zaken te ondersteunen: het uitvoeren van renders met de berekening van de PSNR score op basis van ontvangen node-sequenties, en het afhandelen van resetberichten met een acknowledgement. Het renderen van sequenties gebeurde bij het ontvangen van een bericht van het type `"step"`, waarin een sequentie van node-identifiers geparsed werd. Deze sequentie werd vervolgens doorgestuurd naar de renderfunctie. Deze functie zette alle nodes van het gedeeltelijk geladen scene-object op `invisible`, en activeerde enkel de nodes die in de sequentie voorkwamen door ze opnieuw `visible` te maken. Daarna werd de PSNR-score berekend zoals beschreven in Sectie 3.4.2. Deze score werd via de WebSocket teruggestuurd naar de environment. Merk op dat met deze aanpak de renderer in principe onafhankelijk is van een reset: bij elke nieuwe sequentie worden alle nodes opnieuw geïnitialiseerd qua zichtbaarheid. Toch werd de resetfunctionaliteit reeds voorzien, om toekomstige uitbreidingen mogelijk te maken waarbij bijkomende communicatie tijdens een reset vereist is.

RL-agent:
De initiële agent was een onaangepaste versie van het PPO-object, aangeboden door de Stable Baselines3-library [35]. Deze agent werd gebruikt als uitgangspunt voor verdere experimenten en uitbreidingen. Verdere informatie over deze implementatie is terug te vinden in [3] en [31].

Nu de initiële setup geschetst is, bespreken we de belangrijkste ontwerpkeuzes die geleid hebben tot een verbetering in de prestaties van de agent.

4.3.2 Rewardfunctie

De rewardfunctie, die bepaalt welke beloning de agent ontvangt na het nemen van een actie, is gedurende het project in meerdere fasen aangepast. Het doel van deze functie is om gewenst gedrag te stimuleren door hoge beloningen toe te kennen aan optimale acties en negatieve beloningen te geven voor inefficiënte beslissingen.

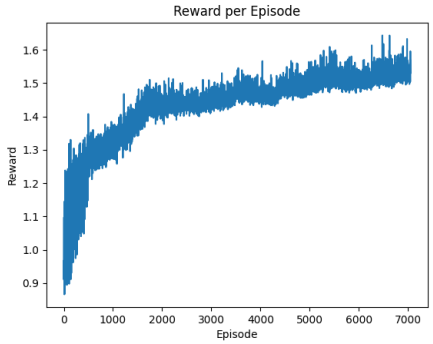
In de eerste versie was de reward per stap eenvoudig gedefinieerd: de totale PSNR-score van het gedeeltelijk gerenderde beeld, "genormaliseerd"door te delen door 100 (dit werd gekozen omdat de maximale waargenomen PSNR die niet oneindig was net onder 100 lag), ten opzichte van het volledige beeld werd berekend en gedeeld door het aantal genomen stappen. Indien een node meermaals werd ingeladen, werd dit beschouwd als inefficiënt gedrag en kreeg de agent een vaste negatieve reward van -1. Het idee achter deze aanpak was dat optimaal gedrag, waarbij de nodes met de grootste PSNR-bijdrage het eerst worden ingeladen, een hogere reward zou opleveren. De bijdrage van een node daalt immers wanneer deze later in de sequentie wordt toegevoegd, omdat de totale PSNR-score gedeeld wordt door het toenemende aantal stappen. Deze aanpak vertoonde echter twee belangrijke tekortkomingen.

Ten eerste werd de reward telkens gebaseerd op de globale PSNR, wat betekent dat eerdere verbeteringen ook in volgende stappen blijven meetellen. Hierdoor konden acties die eigenlijk geen visuele meerwaarde bieden, zoals het inladen van een node buiten het frustum, toch een positieve beloning opleveren. De reward op een individuele stap weerspiegelde dus niet de relatieve bijdrage van die actie.

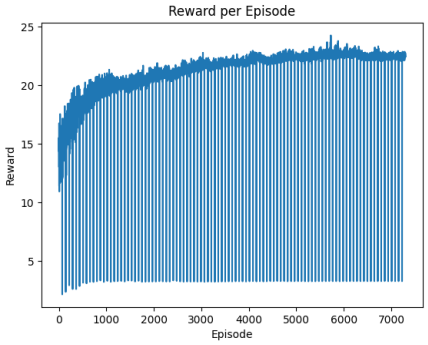
Ten tweede werd de tijdsdegradatie van de reward te agressief toegepast. Deze degradatie had als doel om het inladen van belangrijke nodes in de beginfase van een episode aan te moedigen. In onze oorspronkelijke implementatie gebeurde dit door de globale PSNR-score te delen door het aantal genomen stappen:

$$R_t = \frac{\text{MAX}(\text{PSNR}_{\text{global}}, 100)/100}{t}$$

Hierbij is R_t de reward op timestep t, $\text{PSNR}_{\text{global}}$ de globale PSNR-score die voortkomt uit het renderen van de huidige versie van de opgebouwde sequentie, en t de huidige timestep.



Figuur 4.2: Cumulatieve reward tijdens het trainen van een agent met de originele rewardfunctie op basis van globale PSNR met hyperbolische degradatie.



Figuur 4.3: Cumulatieve reward tijdens het trainen van een agent met de aangepaste rewardfunctie met lineaire degradatie.

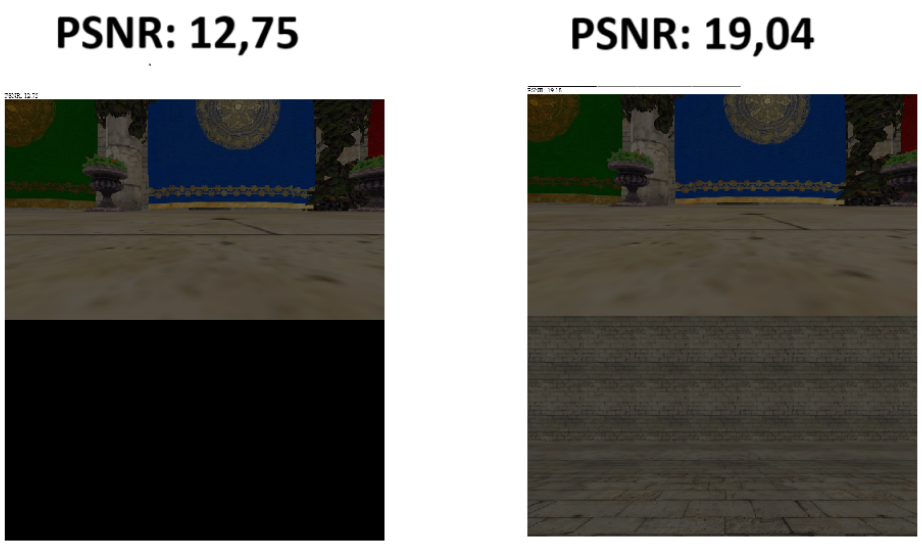
Hoewel dit initieel logisch leek, bleek deze hyperbolische degradatie over het aantal timesteps in de praktijk te streng: enkel de eerste ingeladen nodes hadden een significante impact op de cumulatieve reward, terwijl latere stappen nauwelijks nog meetelden. Dit maakte het moeilijk om uit de scores af te leiden wat het model precies geleerd had, en waarom.

Om dit probleem aan te pakken, vervingen we de deling door een lineaire schaalcoëfficiënt die afneemt over de lengte van de episode:

$$R_t = \text{MAX}(\text{PSNR}_{\text{global}}, 100) \cdot \frac{\text{Episode\_length} - t}{\text{Episode\_length}}$$

Hier is Episode_length het totale aantal nodes in de scene (en dus ook het aantal stappen per episode bij een correcte sequentie zonder dubbele nodes). Bij timestep $t = 0$ is de schaalcoëfficiënt gelijk aan 1; bij $t = \text{Episode\_length} - 1$ is deze bijna nul. Hierdoor wordt de reward lineair gedegradeerd over tijd, in plaats van hyperbolisch.

Hoewel deze aanpassing de scores beter interpreteerbaar maakte, bleven de rewards afhankelijk van de globale PSNR en dus van accumulatie van eerdere acties. Daarom besloten we de reward verder aan te passen door niet langer de globale PSNR te gebruiken, maar de `PSNR-increment` tussen opeenvolgende stappen. Dit wordt berekend als het verschil in PSNR tussen de huidige en vorige timestep. Deze incrementele benadering maakt de reward veel gevoeliger voor het directe effect van een actie. Een bijkomend probleem was dat PSNR exponentieel stijgt, en dus naar oneindig gaat bij perfecte reconstructie (bijvoorbeeld wanneer de laatste node wordt toegevoegd). Om dit numeriek te beperken, kappen we de increment af op een maximum van



Figuur 4.4: De PSNR van de scene zonder ingeladen nodes versus de PSNR van enkel de node die het meeste PSNR-winst oplevert naast hun volledig referentiebeeld. Door de PSNR van het zwarte beeld in rekening te brengen, wordt de bekomen score aanzienlijk realistischer.

1 in het geval van een oneindige sprong. Hoewel deze aanpak niet perfect is, voorkomt ze dat de reward van de laatste stap het model onevenredig sterk beïnvloedt.

Deze rewardmethode, gebaseerd op PSNR-increment, zorgde er voor het eerst voor dat de prestaties van het getrainde model de greedy baseline benaderden bij eenvoudige scènes met één camerastandpunt. Dit toont aan dat het gebruik van relatieve rewardinformatie per stap beter overeenkomt met het daadwerkelijke effect van individuele acties, en dus leidt tot effectievere training.

De finale bekomen rewardscore is als volgt:

$$R_t = \text{PSNR}_{\text{Increment}} \cdot \frac{\text{Episode\_length} - t}{\text{Episode\_length}}$$

4.3.3 StartPSNR

In de oorspronkelijke versie van de environment werd de PSNR-score aan het begin van een episode geïnitialiseerd op 0. Wanneer de eerste actie werd uitgevoerd door het inladen van een bijbehorende node, werd de resulterende PSNR-score onmiddellijk meegenomen in de rewardberekening, zoals beschreven in de vorige sectie. In het geval van de eerste timestep betekende dit een verschil tussen de PSNR van het beeld met enkel de eerste ingeladen node en de initiële waarde 0. Deze aanpak leidt echter tot een onjuiste voorstelling van de kwaliteit van de eerste actie. Een volledig zwart beeld heeft immers een niet-nul PSNR ten opzichte van het volledig gerenderde beeld. Een PSNR-waarde van 0 treedt enkel op in het extreme geval waarbij elke corresponderende pixel een verschil van 255 vertoont, bijvoorbeeld tussen een volledig zwart en volledig wit beeld. Voor een verdere toelichting over de interpretatie van PSNR verwijzen we naar Sectie 2.4.1.

Wanneer de PSNR dus op 0 wordt geïnitialiseerd, resulteert dit in een overschatting van de bijdrage van de eerste ingeladen node (zie Figuur 4.4). Dit probleem wordt versterkt door het exponentieel verloop van PSNR: vroege nodes leveren in absolute termen vaak al een grote

bijdrage aan pixelovereenkomst, maar het verschil in PSNR tussen opeenvolgende timesteps is in dit stadium relatief klein. Hierdoor zullen de rewardverschillen tussen mogelijke eerste acties zeer beperkt zijn, wat het leerproces van de agent bemoeilijkt. De keuze van een optimale eerste node wordt zo onvoldoende gedifferentieerd in termen van beloning. De oplossing voor dit probleem bleek relatief eenvoudig: bij het resetten van de scene vragen we aan de renderer om de PSNR te berekenen van het initiële (lege) beeld ten opzichte van het volledige doelbeeld. Deze waarde wordt vervolgens gebruikt als startwaarde voor de PSNR bij aanvang van een nieuwe episode.

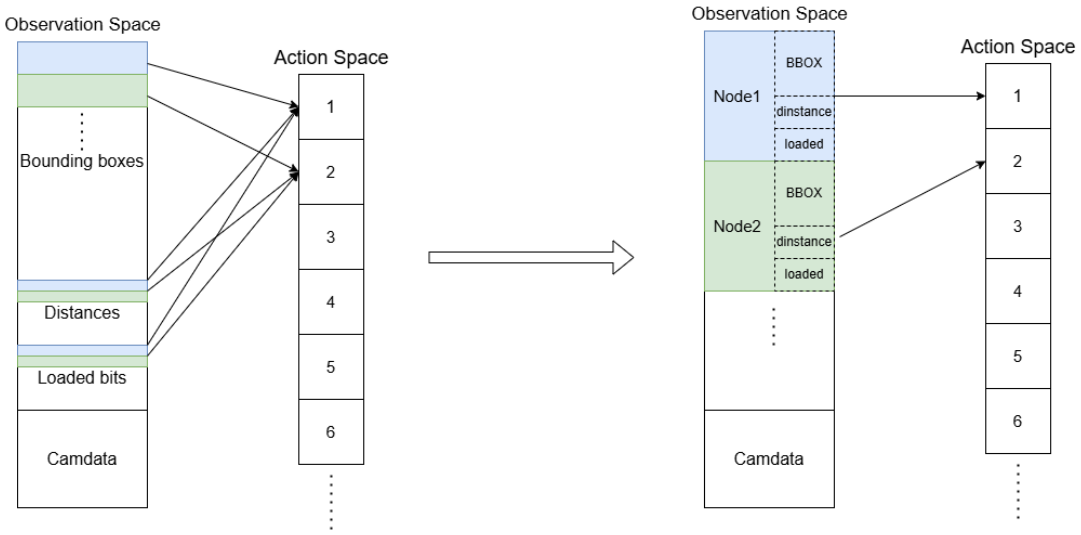
Het bleek bovendien van cruciaal belang om, bij gebruik van variabele camerastandpunten, eerst de correcte camerapositie in te stellen vóór de berekening van deze StartPSNR. Tijdens het testen bleek dit fout te lopen: inconsistenties in rewardscores bij identieke sequences konden worden herleid tot een verkeerde volgorde van initialisatie. De PSNR van de achtergrond werd soms berekend ten opzichte van een referentiebeeld met een verkeerd camerastandpunt, wat leidde tot foutieve beloningen. Hoewel deze correctie geen directe stijging in de cumulatieve rewardscore opleverde (rewardscore was lager door verminderen reward eerste node), zagen we dat de gegenereerde sequenties beduidend meer leken op de greedy-volgordes, wat wijst op een verbeterde trainingsdynamiek.

4.3.4 Camera space bounding boxes

Tijdens de ontwikkeling merkten we op dat het model, ongeacht het gekozen camerastandpunt, vaak dezelfde volgorde van nodes inlaadde. Dit wees erop dat de agent moeite had om de cameraview te correleren met de ruimtelijke structuur van de scene. Op dat moment bestond de observation space uit drie componenten: de axis-aligned bounding boxes (AABBs) van alle nodes in world space, de afstand van elke node tot de camera, en een vector met camerapositie en kijkrichting (zes waarden in totaal). De bounding box-data heeft echter een veel grotere dimensie dan de cameragegevens (aantal nodes $\times$ 2 hoekpunten), waardoor de cameradata relatief ondervertegenwoordigd was. We vermoeden dat het model hierdoor onvoldoende in staat was om camerageboden correlaties te leren.

Om dit probleem te verhelpen, transformeerden we de bounding boxes van world space naar camera space. In dit coördinatenstelsel bevindt het nulpunt zich op de positie van de camera en zijn de assen georiënteerd volgens de kijkrichting. Hierdoor varieert de representatie van de scene sterk afhankelijk van het standpunt en de kijkrichting van de camera, wat het voor het model eenvoudiger maakt om sequenties te leren die afhangen van specifieke camera-instellingen. De conversie van wereld- naar cameracoördinaten werd uitgevoerd via matrixtransformaties, met ondersteuning van `Three.js`. De technische implementatie hiervan wordt besproken in Sectie 3.4.4. Doordat de camerageegevens in de bounding boxes verwerkt werden, konden deze apart uit de observatie worden weggelaten om redundantie te vermijden.

Het effect van deze aanpassing was aanzienlijk: het model stelde voor elk camerastandpunt duidelijk verschillende node-sequenties op. Bovendien nam de performantie over de verschillende camerastandpunten heen sterk toe. Waar het model aanvankelijk slechts in beperkte mate beter presteerde dan een willekeurig gekozen volgorde, benaderden de gegenereerde sequenties nu vaker de greedy score op de camerastandpunten waarop getraind was. Deze resultaten vormden een belangrijke doorbraak. Voor het eerst hadden we bewijs dat de agent effectief leert op basis van de observatie, en niet enkel uit het hoofd leert welke node-indices doorgaans goede keuzes zijn om vervolgens die een hoge logitwaarde toe te kennen. Daarnaast leerden we hieruit dat het in sommige gevallen voordelig kan zijn om het model expliciet te helpen met het leggen van relevante correlaties, zoals die tussen camerageegevens en bounding boxes. Hoewel het ideaal zou zijn als het model deze relaties autonoom leert, blijkt het in de praktijk zinvol om correlaties vooraf expliciet aan te bieden wanneer deze computationeel haalbaar zijn.



Figuur 4.5: Visualisatie van het interpoleren van de data per betreffende node. Hierdoor zal het netwerk niet nog bij moeten leren welke data met welke node te associeren is.

4.3.5 Observation space

Aanvankelijk werd gebruik gemaakt van afzonderlijke `Box`-spaces om de observatieruimte van de agent te modelleren. Een `Box` is een continue of discrete ruimte met gespecificeerde grenzen, dimensies en datatypes. Zo stelden we bijvoorbeeld de axis-aligned bounding boxes (AABBs) van nodes voor als een `Box` van vorm $(n, 6)$, waarbij n het aantal nodes in de scène is en zes waarden volstaan om de uiterste hoekpunten van elke bounding box te beschrijven. Omdat het bereik van deze hoeken niet expliciet begrensd is, gebruikten we $-\infty$ en ∞ als respectieve minimum- en maximumwaarden. het gebruikte datatype was `float32`. Ook andere per-node eigenschappen zoals de afstand tot de camera en de geladen-status (*loaded bits*) modelleerden we op gelijkaardige wijze als `Box`-spaces. Daarnaast bevatten de observaties ook cameragerelateerde informatie zoals positie en kijkrichting. Al deze spaces groepeerden we in een `Dict`-space, een structuur die toelaat om de observatie logisch op te splitsen en elk onderdeel een herkenbare key te geven. Tijdens de preprocessing wordt deze `Dict`-space geflattened tot een enkele vector, waarbij de inhoud van elke `Box`-space sequentieel toegevoegd wordt. Dit resulteert in een lange vector waarbij de data per datatype gegroepeerd is en vervolgens per node gerangschikt. Het nadeel hiervan is dat het model zelf moet leren dat specifieke posities in de vector bij elkaar horen, bijvoorbeeld dat het zesde element in een subvector een visibility flag is die betrekking heeft op de bounding box op de zesde positie in de subvector van deze bounding boxes.

Zoals besproken in de vorige sectie over de converted bounding boxes, is het in sommige gevallen efficiënter om het model expliciete correlaties aan te bieden via preprocessing, in plaats van deze impliciet te laten aanleren. In dit geval werd gekozen om de per-node features (bounding boxes, afstand, geladen-status, ...) te interleaven, zodat alle relevante informatie per node gegroepeerd zit in de uiteindelijke observatievector. Dit vergemakkelijkt het leerproces doordat het netwerk de features van eenzelfde node gelijktijdig te zien krijgt. Zie Figuur 4.5 voor een visualisatie van deze veranderde structuur van de observation space.

Een nadeel van deze aanpak is dat we hierdoor alle per-node gegevens moeten samenbrengen in één enkele `Box`-space, die uniforme grenzen en datatype vereist voor alle dimensies. Concreet betekent dit dat bijvoorbeeld de binaire geladen-status (ideaal voorgesteld als een boolean) hetzelfde datatype en grenzen moet krijgen als continue waarden zoals bounding box-coördinaten. In de praktijk gebruiken we daarom een gemeenschappelijk datatype (`float32`) en een ruime bovengrens, bijvoorbeeld het interval $[0, 1]$ voor de binaire features, binnen het grotere interval van de andere data.

4.3.6 Action Masking

Een belangrijk obstakel in de initiële ontwikkelingsfase van de agent was het herhaald selecteren van reeds uitgevoerde acties, wat resulteerde in het meerdere keren inladen van dezelfde nodes. Dit gedrag leidde tot een inefficiënt gebruik van computationale middelen, aangezien dergelijke herhaalde acties in geen enkele context performant zijn. In de eerste implementaties werd dit probleem aangepakt door een negatieve beloning toe te kennen telkens een actie overeenkwam met een reeds ingeladen node. Hoewel deze actie in werkelijkheid niet werd uitgevoerd, werd de agent er toch voor bestraft. Een vergelijking tussen het aantal voltooide episodes in deze niet-gefilterde benadering en een model met action masking toonde echter aan dat er gemiddeld veertig ongeldige acties ondernomen werden per geldige actie. Dit wees erop dat het model niet in staat was om louter op basis van negatieve beloningen af te leren dat het herladen van nodes verboden is. Om dit probleem te verhelpen, werd onderzocht of action masking een geschikte oplossing kon bieden. Bij action masking worden bepaalde acties conditioneel uitgesloten van de action space van de agent, door de corresponderende logitwaarden te manipuleren zodat ze niet geselecteerd kunnen worden. Een meer gedetailleerde toelichting over deze techniek wordt gegeven in Sectie 2.1.4. Door action masking te integreren in het model, werd het probleem van het herhaald inladen van dezelfde node effectief omzeild in plaats van opgelost via leerprocessen. Deze keuze werd gerechtvaardigd door het inzicht dat het opnieuw inladen van een reeds geladen node in geen enkele situatie wenselijk of nuttig is. Het aanleren van dergelijk gedrag aan de agent werd dan ook als overbodig beschouwd, omdat het leren van de semantiek hiervan niet direct zou zorgen voor betere beslissingen zonder deze masking. Voor een analyse van de impact van action masking op het leerproces en de wijze waarop deze techniek werd toegepast in het experiment, verwijzen we naar Hoofdstuk 5.

4.4 Problemen

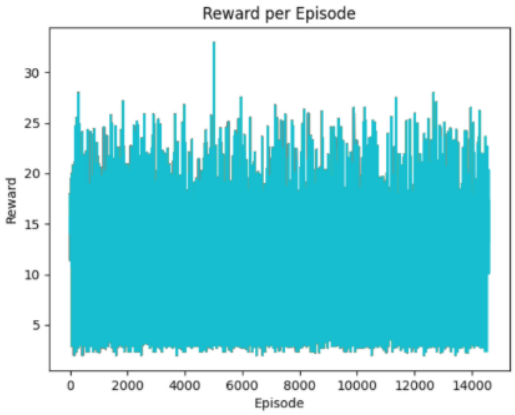
Ondanks deze ontwikkelingen waren er toch nog tekortkomingen van onze getrainde agent met situaties waar deze nog geen goede prestaties behaalde. Deze situaties bespreken we kort in deze sectie.

4.4.1 Willekeurige nodepermutatie

Om te vermijden dat het model simpelweg leert om specifieke actie-indexen te koppelen aan vaste nodes, implementeerden we een mechanisme dat bij het resetten van de environment een willekeurige permutatie toepast op de volgorde van de nodes. Deze permutatie wordt consistent toegepast op zowel de observation space als de action space, zodat de koppeling tussen actie en node behouden blijft, maar de indexen geen vaste betekenis meer hebben. Het doel van deze aanpak is om het model te stimuleren om beslissingen te baseren op de inhoud van de observatie (zoals de getransformeerde bounding boxes), en niet op de positie van een actie binnen de action space. Hoewel we in deze fase enkel met de Sponza-scène werkten, konden we via deze methode toch al variatie injecteren in het leerproces van de agent.

Deze strategie draagt bij aan een grotere generaliseerbaarheid: een model dat niet afhankelijk is van vaste node-indices, maar eerder leert op basis van node-eigenschappen, heeft potentieel een betere overdraagbaarheid naar ongeziene scènes. Dit principe wordt binnen machine learning aangeduid met de term *permutation invariance* [25]. Permutation invariance betekent dat een model dezelfde output moet genereren, ongeacht de volgorde (of permutatie) waarin de invoerdata wordt aangeboden. In onze context is dit principe gedeeltelijk van toepassing: hoewel de volgorde van de nodes in de observation space verandert, verandert ook de volgorde van de acties mee. De betekenis van een actie, het selecteren van een specifieke node, blijft echter behouden, wat betekent dat de semantiek van de beslissing invariant zou moeten zijn onder permutatie.

Bij het trainen van de agent op een environment waarin de nodevolgorde telkens bij het resetten geshuffeld werd, merkten we dat het model moeite had om te leren (zie Figuur 4.6). Dit is



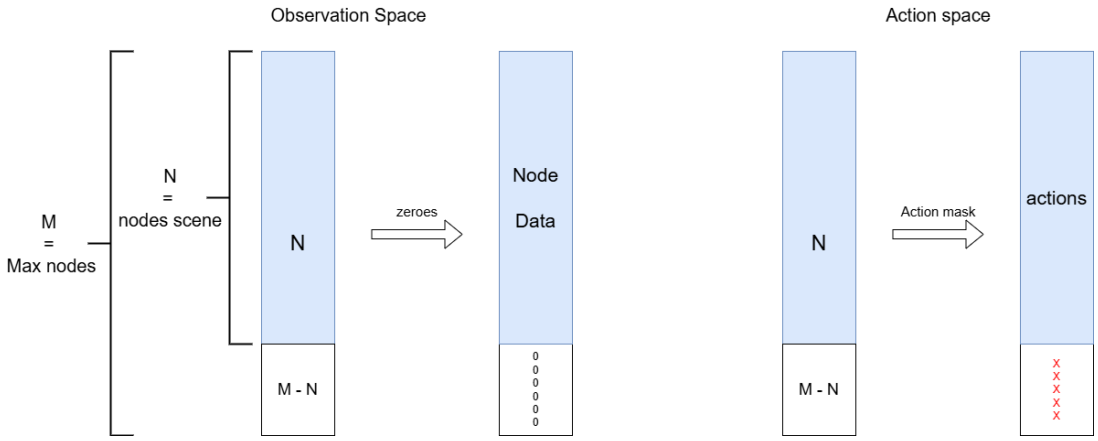
Figuur 4.6: Grafiek van de trainingsfase van de agent die per episode onderworpen werd aan een willekeurige nodpermutatie. Er is duidelijk te zien da ter geen voorwaartse trend in het proces is.

verklaarbaar binnen het reinforcement learning-paradigma, waar een actie vanuit een bepaalde toestand beloond of bestraft wordt. Als de betekenis van een actie (de bijbehorende node) voortdurend wisselt, wordt het moeilijk om stabiele associaties tussen observaties, acties en beloningen te vormen. Het traditionele RL-mechanisme, waarbij actie-indices impliciet een vaste semantiek hebben, valt daardoor grotendeels weg.

Toch bestaan er technieken om reinforcement learning *permutation-invariant* te maken. Zo zijn er neurale architecturen die ontworpen zijn om te werken op verzamelingen (sets) van objecten, zoals *Deep Sets* of *Set Transformers* [50] [27]. Deze modellen verwerken inputs op een manier die expliciet onafhankelijk is van de volgorde waarin de elementen aangeboden worden. Daarnaast bieden ook *Graph Neural Networks (GNNs)* [51] een structuur waarbij individuele elementen (zoals nodes in een scène) met elkaar in verband gebracht worden via relaties, en waarbij permutation invariance ingebouwd is in de wijze waarop informatie over het netwerk geaggregeerd wordt. Dergelijke modellen zouden het mogelijk maken om een observation space met permuteerbare elementen robuuster te verwerken binnen een reinforcement learning-kader.

Het aanpassen van de netwerkstructuur van de agent is echter geen triviale taak. Het integreren van architecturen zoals *Deep Sets*, *Set Transformers* of *Graph Neural Networks* vereist een grondige herwerking van het model, met mogelijk ingrijpende gevolgen voor andere onderdelen van de agent. Belangrijke eigenschappen zoals de *actor-critic*-structuur en de manier waarop policy-updates verlopen, moeten immers behouden blijven om het leerproces correct te laten verlopen. Gezien de beperkte tijd en de complexiteit van dergelijke integraties, was het in het kader van deze thesis niet haalbaar om zulke fundamentele aanpassingen door te voeren.

In plaats daarvan kozen we voor een experimentele aanpak binnen de bestaande architectuur, waarbij we een eigen poging ondernamen om permutation invariance te benaderen. Dit gebeurde via de implementatie van een aangepaste `PerNodePolicy` en een bijhorende `PerNodeFeatureExtractor`, zoals uiteengezet in Sectie 3.2.2. Het kernidee achter deze aanpak is dat elke node afzonderlijk wordt verwerkt door de feature extractor, in plaats van alle features samen te voegen tot één grote inputvector. Elke node genereert op deze manier een aparte embedding, die onafhankelijk door het neurale netwerk wordt doorgestuurd. Dit levert per node een logit-waarde op, waarbij de hoogste logit overeenkomt met de geselecteerde actie. Omdat de volgorde van de nodes hierbij geen invloed heeft op de verwerking, ontstaat er in theorie een vorm van permutation invariance. Deze aanpak botst echter met een fundamenteel aspect van onze PPO-agent: het *actor-critic*-principe. Binnen deze architectuur is het de bedoeling dat het actor-netwerk een volledige policy leert over de actiespace, terwijl onze aanpak eerder neigt naar het afzonderlijk scoren van acties, vergelijkbaar met een Q-function. Voor een uitgebreide toelichting over deze



Figuur 4.7: Visualisatie van hoe we een maximum aantal nodes kunnen declareren en scenes met minder nodes hier kunnen mee behandelen.

spanningsvelden verwijzen we naar Sectie 2.1.3. Ondanks het potentieel van deze benadering bleek ze in de praktijk onvoldoende krachtig om de agent effectief te laten leren. Door tijdsbeperkingen hebben we dit spoor niet verder uitgewerkt, maar het vormt een interessante basis voor toekomstig onderzoek naar permutation-invariant reinforcement learning.

4.4.2 Uitbreidbaarheid naar ongeziene scènes

Alle trainingen die in het kader van deze thesis werden uitgevoerd, vonden plaats op één enkele scène. In de eerste fasen van het project gebeurde dit op een eenvoudige *room*-scène met slechts 10 meshes. Nadien werd een vereenvoudigde versie van de Sponza-scène gebruikt (30 meshes), om uiteindelijk over te gaan naar de volledige Sponza-scène met 142 meshes. Om een agent te trainen die generaliseerbaar is naar ongeziene scènes, is het echter noodzakelijk om meer variatie in de trainingsdata aan te brengen dan enkel via één specifieke scène.

Zoals eerder besproken, probeerden we reeds enige variatie te introduceren door tussen episodes de kijkrichting van de camera te wijzigen. Hoewel dit een stap in de goede richting is, volstaat dit niet om echte generalisatie naar ongeziene scènes te bekomen. Daarom overwogen we aanvullende strategieën, zoals het willekeurig verplaatsen van de camera en, zoals in de vorige sectie besproken, het shuffelen van de volgorde van de nodes en bijhorende data bij elke `reset`. Het combineren van variaties in het camerastandpunt met willekeurige nodepermutaties zou reeds binnen één enkele scène zoals Sponza tot een aanzienlijke diversificatie van de trainingssituaties kunnen leiden. Aangezien bounding boxes worden omgezet naar camera space (zie Sectie 4.3.4), resulteert elke wijziging van de camera in een volledig andere weergave van de scène in de observation space, wat de training robuuster zou kunnen maken. Hiernaast zou zoals beschreven in de vorige sectie de willekeurige nodepermutatie kunnen leiden tot het loskoppelen van actie-nummers en daadwerkelijke nodes voor de agent.

De implementatie van de environment werd reeds voorbereid voor uitbreidbaarheid naar meerdere glTF-scènes toe. Via de parameter `num_valid_nodes` kan het maximum aantal nodes dat de agent moet kunnen ondersteunen, worden gespecificeerd. Deze parameter bepaalt de grootte van de observation space en action space. Een agent moet immers getraind worden op een vaste grootte van input- en outputlagen, wat noodzakelijk is voor de structuur van het neurale netwerk. Dit werd opgelost door, wanneer een scène minder nodes bevat dan dit maximum, de resterende posities in de observation space op te vullen met nulwaarden, en de overeenkomstige acties uit te schakelen via het action masker. Op deze manier kan het netwerk toch flexibel omgaan met scènes van verschillende groottes, al moet de agent wel leren dat observaties bestaande uit nulwaarden overeenkomen met niet-relevante nodes die niet gekozen mogen worden. Voor een schematische voorstelling van dit systeem verwijzen we naar Figuur 4.7.

Aangezien we uiteindelijk niet zijn overgestapt op training met meerdere, verschillende scènes, is het huidige model in de praktijk nog niet toepasbaar op ongeziene scènes. Momenteel moet de agent getraind worden op exact dezelfde scène waarop het later zal worden toegepast. Het model Kan wel leren om per camerarichting vanuit een standpunt een redelijke laadvolgorde van nodes te bepalen binnen die specifieke scène. Toch is deze werkwijze in de praktijk weinig efficiënt: de tijd die nodig is om een agent te trainen voor één specifieke scène met een vast camerastandpunt kan moeilijk concureren met de flexibiliteit van de heuristische methode (zie Sectie 4.2.1), die direct toepasbaar is op iedere scene en ieder camerastandpunt. Het geleverde project kan wel als een Proof-of-concept dienen voor de principes die we introduceerden om een reinforcement agent te trainen een optimale node sequentie voor een scene te genereren.

Hoofdstuk 5

Action masking: experimenten en resultaten

Om onze onderzoeksvraag te kunnen beantwoorden, namelijk of een reinforcement learning-agent kan helpen bij het efficiënt inladen van glTF 2.0-scenes, trachten we een agent op te stellen die een optimale node-sequentie genereert. Deze sequentie moet toelaten om de glTF-data op een effectieve manier progressief te streamen en renderen bij een eindgebruiker. We bouwen voort op het werk van [28] en vergelijken de heuristische methode die daar werd voorgesteld met de prestaties van ons eigen reinforcement learning-model.

In de voorgaande hoofdstukken werd beschreven uit welke componenten het project bestaat en hoe deze met elkaar interageren (zie Hoofdstuk 3), evenals de technische keuzes die tijdens de ontwikkeling gemaakt zijn om de agent in de juiste richting te sturen (zie Sectie 4). Deze hoofdstukken vormen de aanloop naar het reinforcement learning-framework dat ontworpen is om de centrale onderzoeksvraag zo adequaat mogelijk te beantwoorden. In dit hoofdstuk wordt experimenteel onderzocht wat het effect van *action masking* op de prestaties van onze reinforcement learning-agent is. Voor een meer theoretische bespreking van action masking verwijzen we naar Sectie 2.1.4. We evalueren de prestaties van modellen die op verschillende manieren gebruikmaken van action masking, en vergelijken deze onderling en met de greedy- en heuristische methodes voor het genereren van node-sequenties (zie Sectie 4.2 voor meer informatie over deze benchmarks).

Hoewel action masking in veel situaties nuttig kan zijn, moeten we er rekening mee houden dat het tijdens de training ertoe kan leiden dat de agent bepaalde ongeldige acties nooit heeft kunnen exploreren. Hierdoor bestaat het risico dat cruciale aspecten van de probleemstelling onvoldoende geleerd worden. Zoals besproken in Sectie 2.1.4 beïnvloedt het action mask het neurale netwerk zelf niet rechtstreeks. In plaats daarvan worden de logitwaarden van ongewenste acties, na het doorlopen van het action network, naar een zeer lage waarde gezet, zodat de kans dat deze acties geselecteerd worden effectief nul is. De agent leert deze acties dus niet actief af, maar ze worden door een externe ingreep uitgesloten. Overmatig gebruik van action masking tijdens de trainingsfase kan daardoor nadelig zijn voor het leerproces van de agent. In dit experiment wordt geanalyseerd hoe de verschillende versies van de modellen presteren bij het genereren van een node-sequentie, zowel voor geziene als ongeziene kijkrichtingen, afhankelijk van de wijze waarop het action mask toegepast wordt. Op basis van deze experimenten selecteren we uiteindelijk de best presterende variant als onze finale agent, die de onderzoeksvraag het meest doeltreffend weet te beantwoorden.

5.1 Experimentele opstelling

Om het effect van *action masking* op de reinforcement learning-taak, namelijk het genereren van een efficiënte node-sequentie, te onderzoeken, voeren we een reeks experimenten uit waarin enkel de wijze van action masking verschilt. Alle overige onderdelen van het systeem blijven identiek, zodat verschillen in prestaties enkel aan het gebruik van action masking kunnen worden toegeschreven. De basisopstelling van dit experiment is opgebouwd uit de modules beschreven in Hoofdstuk 3, samen met de technische optimalisaties uit Hoofdstuk 4. In deze sectie vatten we de belangrijkste componenten kort samen en beschrijven we de experimentele procedure die gevolgd werd.

5.1.1 glTF-environment

De gebruikte versie van de glTF-environment combineert de eerder besproken implementaties en optimalisaties uit deze thesis. De *observation space* bestaat per node uit drie onderdelen.

1. De bounding box van de node getransformeerd naar camera space (zie Sectie 4.3.4)
2. Een binaire indicator of de node reeds werd ingeladen
3. Een binaire indicator of de node zich binnen het field of view (FOV) van de camera bevindt

Elke bounding box wordt voorgesteld door de acht hoekpunten, wat resulteert in 24 floating-point waarden per node. Aangezien deze bounding boxes in camera space niet langer axis-aligned zijn, volstaan twee uiterste hoekpunten niet om de volledige vorm correct te representeren. Hoewel in theorie minder punten voldoende zouden zijn om een 3D-bounding box uniek te definiëren, werd ervoor gekozen om alle acht hoekpunten op te nemen simpliciteit.

De *action space* bestaat uit 142 discrete acties, overeenkomend met het aantal nodes in de Sponza-scène. De reward die per timestep wordt toegekend, is gebaseerd op de PSNR-toename sinds de vorige stap, gedeeld door een normalisatiefactor die rekening houdt met het aantal reeds ingeladen nodes, zoals beschreven in Sectie 4.3.2. Wanneer de agent een node selecteert die reeds geladen is (wat enkel mogelijk is bij afwezigheid van loaded bits action masking), ontvangt deze een negatieve reward van -1 en wordt de timestep onmiddellijk beëindigd zonder de node toe te voegen aan de sequentie. Een gelijkaardige negatieve reward van -1 wordt toegekend wanneer een node buiten het frustum wordt ingeladen terwijl het maximale PSNR nog niet bereikt is. Deze actie zal wel uitgevoerd worden. Belangrijk is dat deze negatieve rewards uitsluitend dienen voor het sturen van het leerproces tijdens de training. Ze worden niet meegeteld in de cumulatieve reward die gebruikt wordt voor de evaluatie van volledige sequenties in de experimenten.

Tijdens een reset worden eerst alle controlevariabelen gereset, waarna een nieuwe camerakijkrichting geselecteerd wordt. De camera bevindt zich steeds in het centrum van de scène op positie $(0, 0, 0)$, en één van tien vooraf gedefinieerde kijkrichtingen wordt gekozen. Deze richtingen werden handmatig geselecteerd omdat willekeurige kijkrichtingen vaak resulteerden in onbruikbare observaties (zoals wanneer de camera naar de grond kijkt of zich vlak voor een muur bevindt). Ook variabele cameraposities bleken in deze context minder effectief: de Sponza-scène is relatief smal en gesloten, waardoor slechts enkele cameraposities een redelijk zicht op meerdere nodes mogelijk maken. Hoewel dit interessante uitbreidingen waren, konden ze wegens tijdsbeperkingen niet worden gerealiseerd.

Na het selecteren van de cameradata voert de omgeving nog de volgende stappen uit:

- **Frustum culling:** de zichtbaarheid van elke node wordt bepaald via frustum culling (zie Sectie 3.4.3).
- **Start-PSNR:** de beginwaarde van de PSNR voor het niet-gerenderde beeld wordt opgevraagd (zie Sectie 4.3.3).

- **Bounding boxes in camera space:** de omgeving vraagt de omgezette bounding boxes van alle nodes opnieuw op bij de renderer (zoals eerder beschreven in Sectie 4.3.4).

Deze informatie wordt samengevoegd volgens de structuur die eerder in deze sectie werd toegelicht om de volledige observation spaces, geordend in een interweavede structuur zoals besproken in Sectie 4.3.5, voor de agent op te bouwen in de volgende episode.

5.1.2 Agent

De agent die getraind werd is gebaseerd op de standaardimplementatie van `MaskablePPO` uit de `sb3-contrib` library [41]. Hierbij maken we gebruik van de `MultiInputPolicy`, wat betekent dat de agent een observation space in de vorm van een dictionary ondersteunt. In de praktijk worden de verschillende inputs echter eenvoudigweg geconcateneerd en geflattened alvorens ze aan het neurale netwerk gevoed worden. Buiten het specificeren van de environment (zoals besproken in de vorige sectie), worden er geen bijkomende hyperparameters opgegeven, waardoor de actor- en critic-netwerken elk bestaan uit twee verborgen lagen van telkens 64 neuronen, naast de respectievelijke input- en outputlagen. Voor een meer gedetailleerde bespreking van de agentstructuur verwijzen we naar Sectie 3.2.

De keuze voor een `MaskablePPO`-agent laat toe om in de environment een maskingfunctie te definiëren die conditioneel bepaalde acties uitschakelt tijdens het beslissingsproces van de agent. In dit experiment trainen we drie verschillende agents, elk met een andere invulling van deze maskingfunctie:

1. **Zonder action masking:** de maskingfunctie geeft steeds een vector van enen terug, wat impliceert dat alle acties toegestaan zijn.
2. **Masking van reeds ingeladen nodes:** deze agent mag geen nodes selecteren die reeds in de gegenereerde node-sequentie voorkomen.
3. **Masking van reeds ingeladen nodes en nodes buiten het frustum:** bij deze agent worden acties uitgesloten voor nodes die reeds geladen zijn, evenals nodes die zich op dat moment buiten het camera-*frustum* bevinden. Wanneer er geen nodes meer binnen het frustum beschikbaar zijn, wordt deze tweede maskingsregel tijdelijk opgeheven.

Alle agents worden getraind binnen dezelfde implementatie van de environment zoals beschreven in de vorige sectie, met de Sponza-scène als enige gebruikte scène. Elke agent werd getraind gedurende 2 miljoen timesteps, wat resulteerde in ongeveer 14.000 voltooide episodes. Hierbij dient opgemerkt te worden dat het aantal episodes bij de agent zonder masking lager ligt: wanneer een agent een reeds ingeladen node selecteert, wordt de actie weliswaar genomen (met bijhorende negatieve reward), maar levert deze geen voortgang op voor de episode. Een episode wordt pas als voltooid beschouwd wanneer alle 142 unieke nodes één keer zijn ingeladen.

5.1.3 Structuur van het experiment

Met de drie getrainde versies van onze agent, elk met een verschillende configuratie van action masking, voeren we een experiment uit om te bepalen welke aanpak de beste prestaties levert onder uiteenlopende scenario's. We starten met een analyse van de training van elk model en evalueren hun prestaties ten opzichte van de benchmarks gebaseerd op het greedy- en heuristisch algoritme (zie Sectie 4.2 voor een toelichting van deze benchmarks). Deze evaluatie voeren we zowel uit op camerakijkrichtingen die tijdens training gezien zijn als op ongeziene richtingen, om de generalisatiecapaciteit van elk model te onderzoeken.

Vervolgens voeren we een kruisvergelijking uit: elk van de drie modellen wordt geëvalueerd in de scenario's van de andere modellen. Concreet betekent dit dat we de modellen sequenties laten genereren onder drie varianten van action masking:

1. zonder enige vorm van action masking,
2. met enkel masking op reeds ingeladen nodes, en

3. met masking op zowel reeds ingeladen nodes als nodes buiten het gezichtsveld (FOV) van de camera.

Ook onderzoeken we per model hoeveel acties het gemiddeld nodig heeft onder iedere vorm van masking om een volledig beeld te realiseren. Ook vergelijken we voor iedere case het aantal keer dat dit model hogere cumulatieve reward wist te halen dan de heuristische methode. Op deze manier krijgen we een idee hoe deze modellen presteren en hoeveel hulp ze nodig hebben van de action masking om een acceptabel resultaat te kunnen bereiken. Alsook is het interessant om te kijken of action masking in de trainingsfase het leren van de environment belemmert.

5.2 Experiment

Met de opstelling zoals beschreven in Sectie 5.1 voeren we in deze sectie het experiment uit. Eerst trainen we de drie modelvarianten zoals uiteengezet in Sectie 5.1.2, en analyseren we hun individuele prestaties. Vervolgens vergelijken we deze modellen onderling volgens de structuur die we in Sectie 5.1.3 hebben beschreven. De resultaten van deze vergelijkingen worden in deze sectie gepresenteerd. In de volgende sectie gaan we dieper in op de interpretatie van deze resultaten en formuleren we een conclusie over de impact van *action masking* bij reinforcement learning in de context van progressieve glTF-streaming.

5.2.1 Model zonder action masking

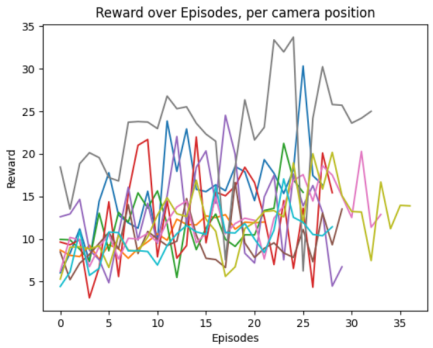
De training zonder enige vorm van action masking leverde beperkte resultaten op. Door het ontbreken van masking op reeds ingeladen nodes bleef de agent herhaaldelijk ongeldige acties uitvoeren, wat het leerproces aanzienlijk vertraagde. Dit uitte zich in het lage aantal succesvol afgewerkte episodes: gemiddeld slechts 30 per camerastandpunt, tegenover ongeveer 1400 bij modellen mét action masking op reeds genomen acties. Deze verhouding is gevisualiseerd in figuur 5.1. Aangezien het uitvoeren van een ongeldige actie geen vooruitgang in de episode oplevert, maar wél een timestep kost, verliest de agent tijdens training veel tijd. Op basis van de verhouding van afgewerkte episodes kunnen we ruwweg schatten dat het model gemiddeld meer dan 40 ongeldige acties uitvoert voor elke geldige actie. Hoewel deze schatting licht vertekend is. De training verloopt immers in batches van 200×10.000 timesteps waarbij episodes worden afgebroken tussen deze batches heen. Toch blijft het duidelijk dat het model moeite heeft om episodes succesvol te voltooien zonder masking.

Deze resultaten onderstrepen waarom action masking al in een vroeg stadium van dit project werd geïntroduceerd. Louter het toekennen van negatieve rewards volstaat niet om de agent af te leren dubbele acties te nemen. Een mogelijke verklaring is dat acties die initieel tot een hogere PSNR leiden, door de agent positief worden versterkt. Wanneer die acties in latere timesteps opnieuw gekozen worden wanneer ze ongeldig zijn, leidt dit tot negatieve feedback, wat verwarrend kan zijn in het leerproces. Dit is problematisch in het kader van reinforcement learning, zeker bij permutatie-afhankelijke modellen zoals besproken in Sectie 4.4.1. Hoewel deze verklaring voorlopig hypothetisch blijft, suggereert ze dat verder onderzoek naar het interne gedrag van de agent nodig is om deze dynamiek te bevestigen of te weerleggen.

Niettemin is het waardevol om na te gaan of de agent ondanks alles enige informatie over de relevantie van nodes op basis van de converted bounding boxes heeft geleerd. In de vergelijkende analyse zullen we daarom dit model evalueren op environments met masking, samen met de andere modellen.

5.2.2 Model met action masking op reeds ingeladen nodes

In deze opstelling werd in de masking function enkel gemasked op acties die reeds genomen waren. Het trainen van deze agent verliep volgens de reward plot 5.2. Op deze grafiek is voornamelijk te zien hoe snel het model verbetert over de episodes heen. Voor de ene kijkrichting is het leerproces sneller dan voor het andere. Vaak is dit omdat sommige kijkrichtingen minder



Figuur 5.1: Deze plot beeldt de training van de agent zonder masking uit op de 10 verschillende kijkrichtingen waar deze op getraind is.

Figuur 5.2: Deze plot beeldt de training van de agent met alleen masking op dubbele acties uit op de 10 verschillende kijkrichtingen waar deze op getraind is.

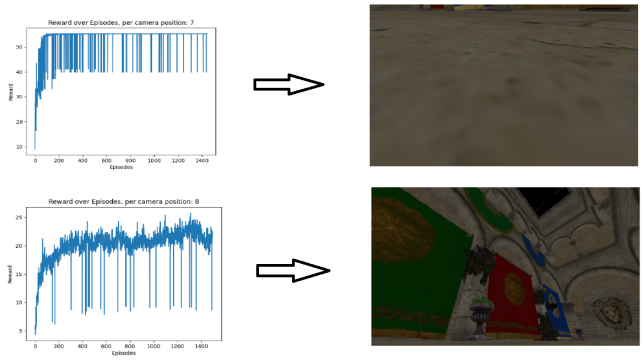
complex zijn dan andere: wanneer je rechtstreeks naar de vloer kijkt zullen minder nodes de PSNR van het beeld bepalen dan wanneer je door de gang heen kijkt. Voor een concreet voorbeeld zie figuur 5.3: hier hebben we een plot die een relatief zeer hoge reward haalt (~ 55) waarvan het beeld voor het overgrote deel bestaat uit de vloer, en een plot die een matigere reward haalt relatief gezien (~ 25) die door de scene heen kijkt. Het is ook te merken dat de plot met het groter aantal nodes in beeld nog meer aan het fluctueren is, terwijl de eerste plot al snel een plafond lijkt te bereiken.

In de praktijk zijn daarom de cumulatieve rewards, de som bekomen van de reward van iedere timestep op te tellen, van verschillende camerakijkrichtingen niet met elkaar te vergelijken. De maximale cumulatieve reward is verschillend afhankelijk van de kijkrichting, omdat deze PSNR veel sneller gemaximaliseerd is bij een standpunt waar slechts enkele nodes ingeladen moeten worden dan een standpunt waar vele nodes ingeladen moeten worden.

Voor de kijkrichtingen waar de agent expliciet op getraind heeft zijn volgende cumulatieve rewardscores bereikt te vinden in Tabel 5.1. We vergelijken hier de score van de agent met de greedy en heuristische laadmethodes.

Camera direction	[0, 0, 1]	[0, 1, 0]	[1, 0, 0]	[0, 0, -1]	[-1, 0, 0]	[1, 1, 1]	[1, 1, -1]	[1, -1, 1]	[-1, 1, 1]	[-1, 1, -1]
Greedy	41.3	38.1	47.6	41.6	45.1	49.5	49.1	55.1	37.0	44.3
Agent	41.2	18.2	46.0	41.4	19.2	47.9	19.4	55.2	22.3	15.1
Heuristic	31.2	16.7	35.1	32.7	36.0	28.1	28.3	54.9	19.8	20.5

Tabel 5.1: Vergelijking van sequence scores per camerakijkrichting tussen de greedy, onze getrainde agent en de heuristic aanpak.



Figuur 5.3: Hier de plots te zien van de cumulatieve rewards over de episodes heen in de training samen met een beeld van hun kijkrichting.

5.2.3 Model met action masking op ingeladen nodes en FOV

Ten slotte werd het model getraind met een uitgebreid action mask, waarbij zowel reeds geladen nodes als nodes buiten het view frustum uitgesloten werden tijdens de trainingsfase. Van zodra er geen nodes meer binnen het frustum beschikbaar zijn, vervalt het FOV-mask automatisch. De reward curves van deze training zijn te vinden in Figuur 5.4. Bij vergelijking met de vorige trainingssessie zonder FOV-masking valt op dat de globale performantie gelijkaardig blijft, maar enkele opvallende verschillen vertoont. Zo zien we dat de scores in richtingen met complexere zichtvelden, doorgaans richtingen met meer zichtbare nodes, gemiddeld hoger liggen in het model met FOV-mask. Bovendien stijgen de beloningen initieel sneller en bereiken ze sneller een plateau. Dit wijst op een versnelde convergentie, wat verklaard kan worden door het FOV-mask dat suboptimale acties uitsluit en zo het zoekproces van het model vergemakkelijkt.

Opmerkelijk is echter dat het model met FOV-mask niet op alle camerastandpunten beter presteert. In vijf van de tien richtingen presteert het model zonder FOV-mask tijdens de training beter (zie Figuur 5.5 voor een voorbeeld). Na controle kon uitgesloten worden dat het FOV-mask noodzakelijke nodes ten onrechte uitsloot. Dit suggereert dat FOV-masking niet per definitie tot betere prestaties leidt op alle zichtvelden. De cumulatieve rewards per camerakijkrichting zijn samengevat in Tabel 5.2. Op basis van deze tabel lijkt het dat FOV-masking voornamelijk voordelig is bij camerastandpunten met lagere baseline scores, mogelijk indicatief voor hogere complexiteit door een voller frustum. Anderzijds zien we juist een daling in prestaties bij richtingen met zeer hoge scores, waar minder visuele complexiteit verwacht wordt. Een analyse van de gegenereerde node-sequenties toonde aan dat de volgorde waarin belangrijke nodes gekozen worden verschilt. In meerdere gevallen koos het FOV-getrainde model niet dezelfde startnode als de greedy baseline, terwijl het model zonder FOV-mask dat wel deed. Deze observatie kan een mogelijke verklaring bieden voor de prestatieverschillen en zou verder onderzocht kunnen worden.

Camera direction	[0, 0, 1]	[0, 1, 0]	[1, 0, 0]	[0, 0, -1]	[-1, 0, 0]	[1, 1, 1]	[1, 1, -1]	[1, -1, 1]	[-1, 1, 1]	[-1, 1, -1]
Greedy	41.3	38.1	47.6	41.6	45.1	49.5	49.1	55.1	37.0	44.3
Agent	31.2	25.4	35.0	31.8	40.6	43.7	31.4	40.3	33.0	25.5
Heuristic	31.2	16.7	35.1	32.7	36.0	28.1	28.3	54.9	19.8	20.5

Tabel 5.2: Vergelijking van sequence scores per camerakijkrichting tussen de greedy, onze getrainde agent en de heuristic aanpak.

5.2.4 Vergelijkende analyse

In deze sectie vergelijken we de modellen die werden getraind met verschillende vormen van action masking, zoals besproken in de voorgaande paragrafen. Het doel is om het effect van action

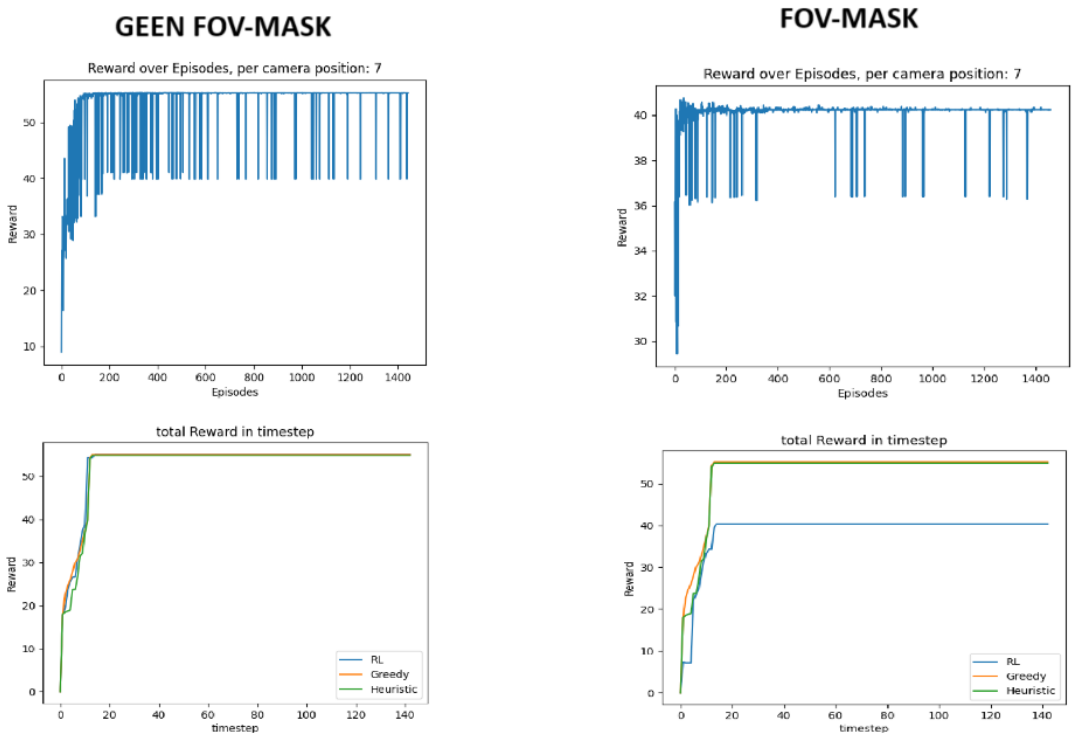
Figuur 5.4: Deze plot beeldt de training van de agent met masking op dubbele acties en nodes die buiten het viewfrustum liggen uit op de 10 verschillende kijkrichtingen waarop deze getraind werd.

masking tijdens training te evalueren door elk model te testen op drie varianten van de environment: zonder masking, met masking op reeds geladen nodes, en met masking op zowel geladen nodes als nodes buiten het view frustum. Concreet wordt elk getraind model geëvalueerd in alle drie de masking-configuraties, ongeacht de configuratie waarin het oorspronkelijk werd getraind. Hierdoor kunnen we nagaan in welke mate het gebruik van masking tijdens training bijdraagt tot generalisatie en prestatieverbetering. We maken een onderscheid tussen *geziene coördinaten* (zichtvelden gebruikt tijdens training) en *ongeziene coördinaten* (nieuwe camerastandpunten). Naast de reinforcement learning (RL) modellen zijn ook de prestaties van de greedy baseline en de heuristische methode opgenomen. Voor deze methodes wordt, ondanks hun onafhankelijkheid van masking, dezelfde opsplitsing in geziene en ongeziene coördinaten gehanteerd om een eerlijke vergelijking te maken.

Tabel 5.3 toont voor elke situatie de gemiddelde reward, het gemiddelde aantal stappen om maximale PSNR te bereiken, en het aantal keer dat de RL-agent beter presteerde dan de heuristische benchmark over de camerastandpunten heen.

Uit de tabel blijkt dat alle RL-modellen niet presteren wanneer er geen enkele vorm van action masking aanwezig is. Tijdens evaluatie zonder masking blijven agents vaak in een oneindige lus van dezelfde actie hangen. Wanneer we expliciet de inferentie pas stop wilden zetten wanneer deze een volledige nodesequentie bereikte bleef deze oneindig lang lopen. Dit is te verklaren doordat inference gebeurt in deterministische modus: voor een gegeven state kiest de agent telkens dezelfde actie. Tijdens training gebeurde dit probabilistisch, wat verkenning stimuleerde. Verder is het echter wel opvallend dat alle drie de modellen nagenoeg identiek presteren op elk van de verschillende environments. Tussen de modellen is er weinig verschil in performantie volgens de gebruikte meetstaven, tussen de gebruikte enironments dan weer wel. Bij evaluatie op de environments met masking presteren de modellen aanzienlijk beter. De gemiddelde rewardscore bij geziene coördinaten is nagenoeg gelijk tussen de environments met enkel masking op reeds genomen acties en die met extra FOV-masking. Het aantal benodigde stappen tot maximale PSNR is echter bijna dubbel zo hoog bij enkel action masking, wat op minder efficiënte sequenties wijst. Bij ongeziene coördinaten is het verschil tussen deze twee masking-configuraties duidelijker: FOV-masking leidt zowel tot hogere rewards als kortere sequenties.

Een opvallende observatie is dat het aantal ‘wins’ ten opzichte van de heuristische methode groter is bij ongeziene coördinaten wanneer volledige masking wordt toegepast. Dit duidt op betere generalisatie. Bij enkel action masking zijn de modellen echter succesvoller op de geziene coördinaten. Volledige masking zorgt er bovendien voor dat elk model in minstens de helft van de testgevallen beter presteert dan de heuristiek. Deze resultaten tonen aan dat action masking tijdens inferentie niet alleen het leerproces vergemakkelijkt, maar ook de generaliseer-



Figuur 5.5: Vergelijking van trainingsprestaties op een bepaald camerastandpunt: het model zonder FOV-mask presteert hier beter dan het model met FOV-mask.

baarheid van het model verbetert, vooral wanneer frustum-informatie mee in rekening wordt gebracht.

5.3 Discussie

Nu de resultaten van het experiment besproken zijn, wordt ingegaan op de conclusies die getrokken kunnen worden over het gebruik van action masking binnen het reinforcement learning framework dat ontwikkeld werd om optimale nodesequenties te genereren voor progressieve glTF-streaming.

5.3.1 Gebruik van action masking tijdens inference

Uit de experimenten blijkt dat action masking tijdens inference essentieel is om volledige nodesequenties te verkrijgen en het herhaald laden van reeds ingeladen nodes te vermijden. Pogingen om dit gedrag af te leren via negatieve beloning (bv. $r = -1$ bij dubbele loading) bleken onvoldoende effectief: het model bleef repetitieve, niet-productieve acties uitvoeren. Hierdoor concluderen we dat action masking op reeds ingeladen nodes een noodzakelijke voorwaarde is voor efficiënte inference. Wat betreft het gebruik van FOV-masking tijdens inference blijkt dat het model met en zonder FOV-masking vergelijkbare prestaties behaalt op geziene camerastandpunten. Beide modellen overtreffen daarbij nipt de heuristische baseline. Wel is het zo dat FOV-masking leidt tot kortere sequenties (minder stappen tot maximale PSNR), wat duidt op een efficiëntere beslissingsstrategie. Op ongeziene camerastandpunten blijkt FOV-masking bovendien noodzakelijk om acceptabele prestaties te behalen. Het model zonder FOV-masking generaliseert hier minder goed.

Toch is het gebruik van FOV-masking niet zonder nuance. In scenario's met een bewegende camera kan het bijvoorbeeld wenselijk zijn om ook nodes net buiten het frustum vroegtijdig in te laden, vooral wanneer deze binnenkort in beeld zullen komen of nodes binnen het frustum

weinig waarde toevoegen (bv. bedekt door andere objecten). Daarom kan geconcludeerd worden dat het gebruik van FOV-masking tijdens inference afhankelijk is van het gewenste gedrag van de agent en de dynamiek van de scène. In de huidige setup blijkt het echter noodzakelijk om robuuste prestaties op ongeziene standpunten te bereiken.

5.3.2 Gebruik van action masking tijdens training

De keuze voor action masking tijdens training blijkt minder simplistisch te zijn. De drie gemaskeerde trainingsmodellen (zonder mask, enkel geladen nodes en extra FOV-mask) behalen gelijkaardige prestaties, zowel tijdens training als op de validatiesets. Dit zou erop kunnen wijzen dat het model op dit moment voornamelijk leert op basis van de positie van acties in de outputvector in plaats van op basis van semantiek in de input. Dit besproken we eerder al uitgebreid in Sectie 4.4.1 als *permutation dependence*. Om dit te bevestigen zou echter verder onderzoek genoodzaakt zijn.

Zolang het model nog afhankelijk is van de permutatie van de acties in de outputvector, zoals besproken in Sectie 4.4.1, lijkt de keuze van action masking tijdens de trainingsfase slechts een beperkte impact te hebben. De modellen met verschillende maskingstrategieën behalen immers vergelijkbare prestaties. In dit stadium leert het model eerder welke actie-indexen goed werken in specifieke toestanden dan dat het semantisch betekenisvolle verbanden legt tussen inputeigenschappen en acties. Wanneer het model echter verder ontwikkeld wordt en deze permutatieafhankelijkheid weggenomen wordt, zou de keuze van masking tijdens training mogelijk wel een invloed kunnen hebben. De hypothese hier naar voren geschoven is dat masking van reeds ingeladen nodes in alle gevallen optimaal blijft. Dubbele loads bieden in geen enkel scenario een voordeel en het is niet zinvol dat de agent moet leren waarom dit ongewenst gedrag is; het kan dus efficiënter geëlimineerd worden via masking. De rol van FOV-masking tijdens training is minder evident. Het kan waardevol zijn voor de agent om ook toegang te krijgen tot nodes buiten het frustum, om zo te leren welke nodes semantisch minder relevant zijn om in te laden. Dit draagt mogelijk bij tot een beter begrip van de geometrie van de scène en de interpretatie van de zogeheten *converted bounding boxes*, zoals besproken in Sectie 4.3.4. Wanneer nodes buiten het frustum automatisch gemaskeerd worden, wordt deze semantiek mogelijk moeilijker bereikbaar voor het model. Het zal de kans niet hebben kennis te maken met suboptimale acties, dus hoe zal het optimale acties kunnen onderscheiden van suboptimale acties? In plaats van masking kan dit ongewenste gedrag ook op andere manieren worden bijgestuurd, bijvoorbeeld via negatieve beloning. In dit experiment werd gebruikgemaakt van een eenvoudige binaire penalty ($r = -1$) voor het laden van een node buiten het FOV zolang het beeld nog niet volledig was. In toekomstig werk kan dit rewardmechanisme verfijnd worden via complexere strategieën, die bijvoorbeeld rekening houden met de afstand tot de camera, de oriëntatie van de bounding box ten opzichte van de camera, of zelfs met optical flow — zeker in dynamische scènes met bewegende camera's.

Samengevat bevelen we voor toekomstig werk aan om tijdens training enkel action masking toe te passen op reeds ingeladen nodes. Voor FOV-masking lijkt het aangewezen om dit achterwege te laten tijdens training, zodat het model vrij blijft om semantische relaties tussen zichtbaarheid en relevantie van nodes zelfstandig te leren.

5.3.3 Prestatie versus greedy- en heuristische methode

Bij het vergelijken van de prestaties van de getrainde modellen met de greedy- en heuristische methode blijkt dat de modellen met FOV-masking met een kleine marge beter presteren dan de heuristische methode. De prestaties van de greedy methode liggen momenteel nog buiten bereik, zeker in de huidige opzet. Deze globale observatie verbergt echter een aanzienlijke variatie per camerastandpunt. In bepaalde richtingen presteert de agent vergelijkbaar met de greedy methode, en in sommige gevallen zelfs beduidend beter dan de heuristische aanpak. In andere richtingen scoort de agent echter slechter dan beide benchmarks. Voorbeelden van zulke variabele prestaties zijn te zien in figuur 5.6.

Voor de geziene camerastandpunten overschrijden de modellen met enkel masking op ingeladen nodes, evenals de modellen met aanvullend FOV-masking, de heuristische methode op vlak van cumulatieve reward. Het verschil in aantal stappen om de maximale PSNR te bereiken is echter groot: zonder FOV-masking duurt dit significant langer. Dit verschil wordt vaak veroorzaakt door kleine, minder visueel significante meshes die pas laat geladen worden en nauwelijks bijdragen aan de visuele volledigheid, maar die wel het aantal stappen verhogen. FOV-masking tijdens inferentie voorkomt dit: doordat nodes buiten het view frustum niet geselecteerd worden, ontstaat er consistent snel een volledig beeld. Het gevolg is een consistente en efficiënte volgorde bij progressive streaming. Het effect op de PSNR over de sequentie heen bij het laat inladen van enkele nodes wordt gevisualiseerd in figuur 5.7.

Voor ongeziene camerastandpunten zien we dat modellen zonder FOV-masking veel slechter presteren dan de heuristische methode, zowel qua rewardscore als aantal stappen. Verrassend genoeg presteren de modellen met FOV-masking tijdens inferentie bijna even goed op ongeziene camerastandpunten als op de geziene. Deze versie scoort bovendien beter dan de heuristische methode op vlak van cumulatieve reward. Dit suggereert dat FOV-masking de capaciteit tot generaliseren naar ongeziene kijkrichtingen van het model aanzienlijk verhoogt.

5.4 Finaal model

Nu het experiment rond action masking uitgevoerd en besproken is, schuiven we in deze sectie de finale opstelling naar voren die volgens onze evaluatie het beste presteert binnen het kader van onze probleemstelling. Die probleemstelling bestaat erin een reinforcement learning-agent te ontwikkelen die progressive streaming van glTF 2.0-bestanden ondersteunt. Concreet betekent dit dat de agent een optimale volgorde van nodes leert genereren, zodat met zo weinig mogelijk dataverkeer toch snel een visueel bruikbaar beeld van de scene ontstaat, vanuit het oogpunt van de gebruiker.

De implementatie die we als finaal model voorstellen, komt voort uit het project zoals beschreven in Hoofdstuk 3, en bevat de volgende details:

- **MaskablePPO-agent**
  Voor het ondersteunen van action masking tijdens zowel training als inferentie gebruiken we de standaardversie van de MaskablePPO-agent. Pogingen om een eigen per-node model te ontwikkelen leverden nog geen bevredigende resultaten op (zie Sectie 3.2.2). Om deze reden gebruiken we de standaard implementatie geleverd door de Stable-Baselines 3 contrib library [41].

- **glTF-environment**
  Deze custom environment simuleert de interactie tussen de agent en de 3D-scene. Elke actie komt overeen met het laden van een specifieke node. De implementatie van deze omgeving is ontworpen om representatieve observaties en rewards te genereren, zodat de agent effectieve beslissingen kan leren nemen. Voor details verwijzen we naar Sectie 3.1.

- **Rewardfunctie**
  De rewardfunctie is een cruciale component van het RL-framework. We gebruiken de functie beschreven in Sectie 4.3.2, waarbij de PSNR-toename als resultaat van het laden van een node wordt gedegradeerd volgens het aantal genomen stappen. Penalty's van -1 worden toegekend bij het herladen van reeds ingeladen nodes of bij het laden van nodes buiten het view frustum zolang er zich nog niet-ingeladen nodes binnen het frustum bevinden. Verder wordt aandacht besteed aan een correcte initialisatie van de start-PSNR (zie Sectie 4.3.3), aangezien dit de effectieve beloningswaarden beïnvloedt.

- **Observatie**
  De observatie van de environment die in iedere stap gegeven wordt aan de agent om zijn volgende actie te bepalen, is ook cruciaal voor een functionerend RL-framework. Dit zal de inputvector vormen na een preprocessingstap die naar het feed-forward policy network

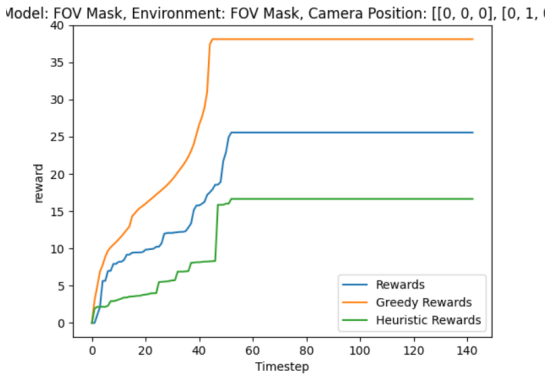
gestuurd wordt om de keuze van de volgende actie te bepalen. Onze observatie, zoals in detail beschreven in Sectie 4.3.5, zal de scene voorstellen door een grote vector op te stellen die per node zijn camera space bounding boxes bevat, samen met een loaded bit en FOV-bit. Deze camera space bounding boxes zijn de bounding boxes van elke node in de scene, omgevormd van world space naar camera space (uitgebreid beschreven in Sectie 4.3.4). De loaded bit geeft aan of de bijhorende node reeds werd ingeladen, en het FOV-bit geeft aan of de bounding box van deze node het view frustum snijdt.

- **Maskingfunctie**
  De manier waarop we action masking toepassen op onze agent is gebaseerd op de bevindingen uit het experiment in dit hoofdstuk. Tijdens de trainingsfase raden we aan om masking toe te passen op reeds ingeladen nodes. Dit voorkomt dat de agent expliciet compleet nutteloze acties moet afleren, zodat hij zich beter kan focussen op het leren van de structurele eigenschappen van de 3D-scène. We raden echter aan om FOV-masking tijdens training achterwege te laten, zodat de agent suboptimale acties kan verkennen en de semantiek van bounding boxes buiten het frustum kan leren begrijpen. Voor de inferentiefase bevelen we aan om zowel masking op reeds ingeladen nodes als op nodes buiten het view frustum toe te passen. Dit verhoogt de prestaties van het model, vooral bij ongeziene cameraposities en kijkrichtingen. Voor verdere onderbouwing van deze keuzes verwijzen we naar de discussie in Sectie 5.3. In toekomstige uitbreidingen met krachtigere modellen zou het weglaten van FOV-masking overwogen kunnen worden om ook scenario's met een bewegende camera of andere use-cases beter te ondersteunen.

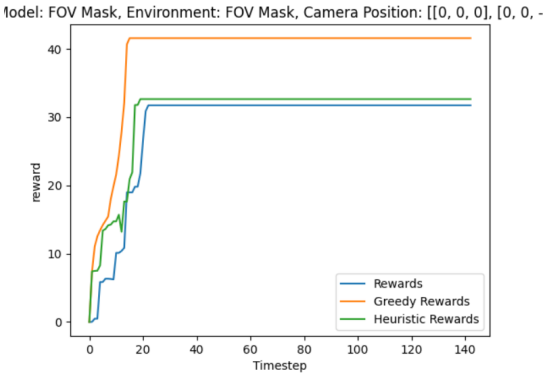
Deze finale opstelling presteert beter dan de heuristische benchmark, maar blijft nog onder het prestatieniveau van de greedy benchmark binnen de Sponza scene met een vast camerastandpunt en variërende kijkrichting. Uitleg over beide benchmarks is terug te vinden in Sectie 4.2.

Model	Geziene Coördinaten	Ongeziene Coördinaten
Greedy	Gem. reward: 44.6 Gem. stappen: 37.2	Gem. reward: 45.4 Gem. stappen: 47.7
Heuristic	Gem. reward: 30.4 Gem. stappen: 43.6	Gem. reward: 29.6 Gem. stappen: 56.6
RL No Mask	**No Mask:** Reward: 1.4, Stappen: N/A Wins: 0, Losses: 10 **Action Mask:** Reward: 33.0, Stappen: 79.7 Wins: 8, Losses: 2 **FOV Mask:** Reward: 33.6, Stappen: 45.3 Wins: 6, Losses: 4	**No Mask:** Reward: 0.7, Stappen: N/A Wins: 0, Losses: 15 **Action Mask:** Reward: 23.5, Stappen: 119.4 Wins: 3, Losses: 12 **FOV Mask:** Reward: 31.0, Stappen: 58.8 Wins: 9, Losses: 6
RL Action Mask	**No Mask:** Reward: 2.6, Stappen: N/A Wins: 0, Losses: 10 **Action Mask:** Reward: 32.5, Stappen: 79.9 Wins: 8, Losses: 2 **FOV Mask:** Reward: 33.6, Stappen: 45.5 Wins: 6, Losses: 4	**No Mask:** Reward: 0.5, Stappen: N/A Wins: 0, Losses: 15 **Action Mask:** Reward: 22.8, Stappen: 119.1 Wins: 2, Losses: 13 **FOV Mask:** Reward: 31.5, Stappen: 58.7 Wins:10, Losses: 5
RL FOV Mask	**No Mask:** Reward: 2.1, Stappen: N/A Wins: 0, Losses: 10 **Action Mask:** Reward: 33.2, Stappen: 81.4 Wins: 8, Losses: 2 **FOV Mask:** Reward: 33.6, Stappen: 45.5 Wins: 6, Losses: 4	**No Mask:** Reward: 1.4, Stappen: N/A Wins: 0, Losses: 15 **Action Mask:** Reward: 23.5, Stappen: 121.7 Wins: 3, Losses: 12 **FOV Mask:** Reward: 31.5, Stappen: 58.9 Wins: 10, Losses: 5

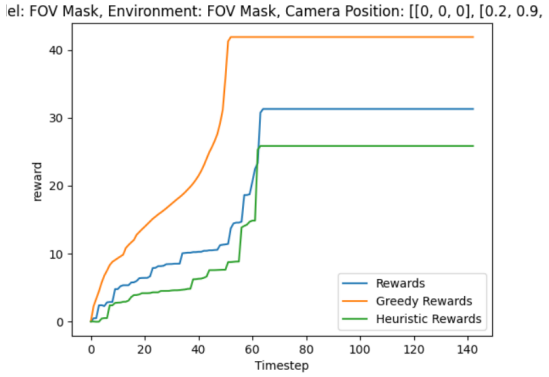
Tabel 5.3: Vergelijking van modellen op geziene en ongeziene coördinaten. Voor de RL-agenten worden prestaties gegeven onder drie evaluatie-masks: geen mask, action mask, en action + FOV mask.



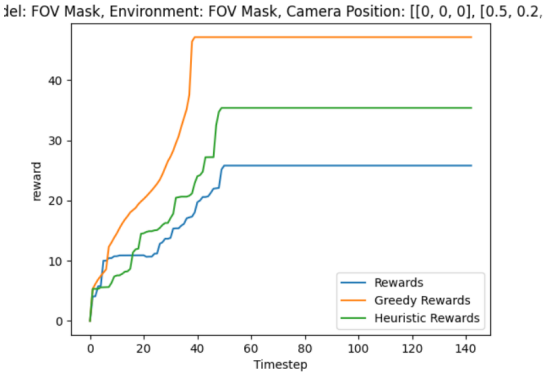
Goede prestatie op geziene kijkrichting



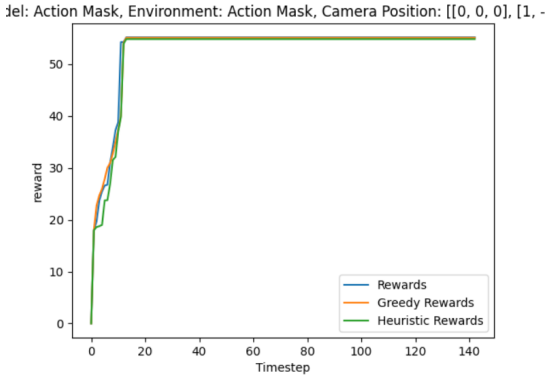
matige prestatie op geziene kijkrichting



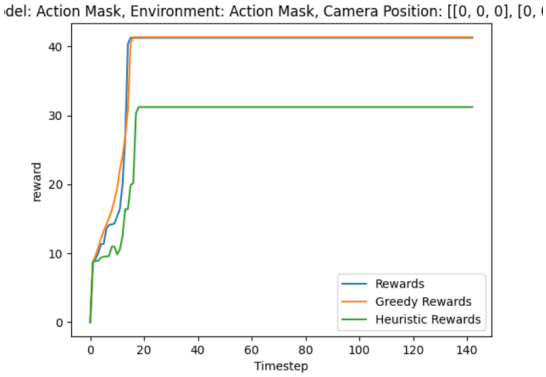
Goede prestatie op ongeziene kijkrichting



Zwakke prestatie op ongeziene kijkrichting

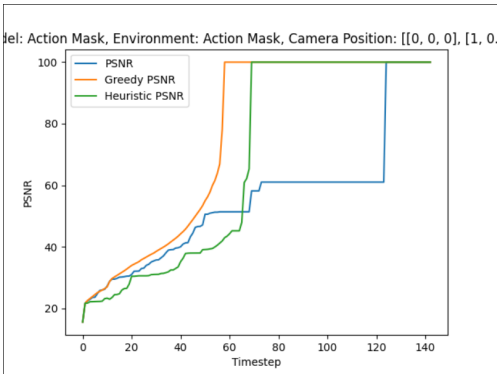


ijkrichting waar het getrainde model en de heuristische methode greedy evenaren



Kijkrichting waar het getrainde model de greedy mehtode evenaart, maar de heuristische methode niet

Figuur 5.6: Een overzicht van de variabele prestatie van de getrainde modellen ten opzichte van de heuristische en greedy benchmark.



Figuur 5.7: De plot die de PSNR aangeeft over het inladen van de nodes over de gegenereerde sequentie heen die tijdens inference geen gebruik maakt van FOV-masking. Hier is te zien dat enkele nodes pas laat geladen worden wat zorgt voor een hoger gemiddeld aantal stappen tot maximale PSNR

Hoofdstuk 6

Conclusies

In deze thesis probeerden we de onderzoeksvraag te beantwoorden of reinforcement learning gebruikt kon worden om progressive glTF 2.0 streaming te ondersteunen en eventueel verbeteren in vergelijking met bestaande technieken. Specifiek werd onderzocht hoe we een reinforcment learning agent konden trainen om een optimale sequentie van glTF nodes te bepalen, die wanneer ze in deze volgorde progressief gestreamd worden met zo weinig mogelijk gestreamde nodes een compleet beeld van de scene kunnen vormen. Dit onderzoek is in feite een verderzetting van eerder onderzoek naar progressive glTF 2.0 streaming te lezen in de paper, [28], waar men een heuristische methode voorstelt die aan de hand van de afstand van de nodes tot de camera en een FOV flag, die aangeeft of de betrekkende node zich in het view-frustum bevindt, de laadvolgorde van de nodes zal bepalen. Deze oplossing gebruikten we dan ook als benchmark in ons eigen onderzoek.

Om dit probleem op te lossen hebben we eerst uitgebreid onderzoek gedaan naar de relevante topics, voornamelijk rond reinforcement learning en glTF 2.0-, of andere graphics typen streaming. Met deze kennis ontwikkelden we een codebase waar de reinforcement agent in ontwikkeld kon worden. Hiervoor werd een systeem ontwikkeld waarbij glTF 2.0 data geparsed werd en aan de custom glTF environment werd geboden. Deze environment is een Python-module waarmee de agent zal interrageren om het correcte gedrag of policy aan te leren om een optimale nodesequentie te kunnen voorspellen aan de hand van de data uit het glTF bestand. De interacties van de agent met de environment is in deze probleemstelling het selecteren van de volgende node om deze toe te voegen aan de nodesequentie. Om dit effect te simuleren en de kwaliteit van de actie te beoordelen werd er een rendering engine in JavaScript naast deze environment geïmplementeerd die met elkaar communiceren via WebSocket-communicatie. De taak van deze rendering engine zal zijn om sequenties van nodes te beoordelen op de gelijkaardigheid van het samengestelde beeld resulterend uit deze sequentie in vergelijking met het beeld van de volledig ingeladen scene. Deze beoordelingen deden we door de Peak Signal-to-Noise Ratio te berekenen van deze 2 beelden ten opzichte van elkaar. Dit is een metriek die gelijkheid van signalen uitdrukt, waaronder beelden met hun RGBA-waarden ook vallen. Deze score werd gebruikt om de actie van de agent te beoordelen zodanig dat deze zijn policy aan kan passen om PSNR maximalisatie na te streven. Met dit framework zijn we aan de slag te gaan om te onderzoeken welke delen van deze learning cyclus we konden manipuleren om de agent te stimuleren om zo goed mogelijk het gewesnte gedrag aan te leren.

Echter werd al snel duidelijk dat dit geen triviale opdracht was om aan een reinforcement learning agent te leren. Een zeer groot probleem bleek om de agent afgeleerd te krijgen om repititieve acties te doen, in ons geval een node 2 keer inladen. Een negatieve reward teruggeven volstond niet om de agent dit gedrag af te leren. Het is echter niet onlogisch dat dit in klassieke RL modellen onintuïtief gedrag is: wanneer een actie een positieve belonging oplevert zal de policy deze actie aanmoedigen, wat motiveert om deze in een volgende state opnieuw te kiezen.

Een node 2 keer inladen is in het geval van deze probleemstelling in ieder geval compleet nutteloos. Dit heeft geleid tot verder opzoekwerk naar soortgelijke problemen, waar al snel het gebruik van action masking binnen reinforcement learning naar boven kwam. Dit is een systeem waarbij per timestep acties ongeldig verklaart kunnen worden waardoor de agent deze nooit zal selecteren. Door deze masking toe te passen op reeds genomen acties zagen we dan dat de agent er in slaagde zijn eerste vooruitgang te boeken.

Na verdere ontwikkelingen werd iteratief een beter en beter presterend model ontwikkeld dat uiteindelijk met een kleine marge beter wist te presteren op de scène waarop het model getraind is dan de heuristische methode voorgesteld in het voorgaande werk. Het gebruik van action masking op reeds ingeladen nodes was cruciaal om het model efficiënt episodes te laten voltooien in de trainingsfase en uitgebreidere masking hielp tijdens de inferentie van het model goede resultaten te halen. Het gebruik van bounding boxes van iedere node getrasformeerd van world-space naar camera-space was tevens ook een cruciale ontwikkeling. Dit zorgde ervoor dat de agent niet zelf het effect van de cameradata op de geometrische data moest ontdekken, maar we deze al impliciet in de invoerdata konden verwerken. Hiernaast zijn er nog kleinere vooruitgangen gerealiseerd door opitimalisaties te doen van de rewardscore, observation space en de glTF environment in het algemeen. Uit de ontwikkelingen die voortgang brachten voor de agent kon vooral geconcludeerd worden dat het belangrijk is om waar mogelijk correlaties te leggen tussen data in de preprocessing stap. De getransformeerde bounding boxes zijn hier een voorbeeld van. In eerste instantie werden de boxes in world space gebruikt en een aparte vector die de cameradata voorstelde. De agent sloeg er echter niet in in de trainingsfase de correlatie tussen deze data te leggen, wat leidde tot bijna identieke nodesequenties voor ieder camerastandpunt. Het impliceren van deze cameradata op de bounding boxes verhielp dit probleem. Idealiter zou men willen dat een Reinforcement learning agent deze correlaties tijdens de training ontdekt, maar met de tools voor handen was dit eerder onrealistisch. Bij het trainen van een taak-specifieke agent blijkt het noodzakelijk om de learning zoveel mogelijk te focussen op de daadwerkelijke taak voor handen, in dit geval het samenstellen van een goede nodesequentie vanuit een bepaalde cameraorientatie, in plaats van het model zelf nog onnodig correlaties leren leggen.

Echter laat het ontwikkelde resultaat nog veel ruimte voor verbetering. De huidige staat van het project is eerder een proof-of-concept van de principes om een reinforcement learning agent te verbeteren gegeven onze probleemstelling, maar is in praktijk nog niet toepasbaar op realistische scenario's. Training en inferentie van de agent gebeurden steeds enkel op de Sponza scëne vanuit een enkele camerapositie. De kijkrichting werd aangepast om variate te bieden, maar dit is slechts de eerste stap naar een generaliseerbaar model dat presteert op arbitraire glTF 2.0 scënes. Een probleem naar verdere uitbreidingen toe is het principe van permutation invariance wat dit model nog niet beschikt. Dit beduidt het feit dat het model de indices van gunstige acties vanbuiten leert, waardoor het goede acties eerder koppelt aan zijn plaats in de output vector dan aan de positie van bijhorende data in de observatie of input vector. Dit viel op wanneer we tussen episodes heen een willekeurige shuffle deden van de volgorde van de nodes in de scene in de inputvector en bijgevolg ook de bijhorende acties in de outputvector. Hier kon de huidige implementatie van het model echter compleet niet mee om, wat zorgde dat het model niet leerde tijdens de trainingsfase. Na een studie naar dit probleem wijst het erop dat het probleem zit in de interne arrchitectuur die het niet toelaat deze permutation invariance te bereiken. Er is een poging geweest om een permutation invariant policy model te maken die de nodes niet als een grote vector behandeld, maar individueel. Dit was echter nog onsuccesvol. verder zou er wel al binnen dezelfde scene uitgebreid kunnen worden naar verschillende camerstandpunten tussen episodes, maar door tijdsgebrek is dit niet in de thesis opgenomen.

6.1 Zelfreflectie

Het uitwerken van deze thesis is niet altijd een vlak parcours geweest. Bij de eerste fase waar de lijnen van het onderzoek afgebakend werden waren we zeer optimistisch over wat er mogelijk was

om uit te werken in deze thesis. Zo waren er ideeën om adaptive streaming toe te voegen aan dit probleem: het ondersteunen van streaming van verschillende formaten per mesh en netwerkcondities mee te gebruiken om de agent zijn keuzes te laten nemen. Anderzeids waren er ideeën om paden te voorspellen die de gebruiker door de scëne zou kunnen nemen om de nodesequentie hierop te optimaliseren. In realiteit had ik de basis van het probleem onderschat: een reinforcement learning agent kennis te laten nemen met de geometrische structuur van een 3D scëne en aan de hand hiervan een optimale volgorde te selecteren voor de progressive streaming. Het onderschatten van deze taak leidde tot problemen en misconcepties over reinforcement learning in het begin van het project. Er werd al snel duidelijk tijdens de eerste implementaties dat een betere verstandhouding van de principes van reinforcement learning nodig was om de agent het correcte gedrag aan te leren. De innerlijke werking van het gebruikte model die de agent voorstelde had ik op voorhand ook niet genoeg in detail onderzocht waardoor de problemen die naar voren kwamen in deze fase moeilijk te verklaren en verhelpen waren. Na diepere studie op de werking van deze systemen bleken de oorzaken van deze problemen ook logisch en simpel te voorkomen. Hieruit leerde ik al snel dat het belangrijk was een uitgebreidere literatuurstudie te doen over de kern van de probleemstelling, zijnde reinforcement learning, voordat ik naar de daadwerkelijke implementatie van de oplossing van de probleemstelling overging.

Op vlak van de onderzoeksmethode en ontwikkelingsproces naar aritificiele intelligentie in het bijzonder heb ik ook zeer veel bijgeleerd. In de beginfasen van het project leverde ik redelijk willekeurig de data aan aan het policy netwerk met als doel een zo volledig mogelijk beeld van de situatie voor te stellen aan de agent. Samen met een zeer simpele rewardscore dacht ik op deze manier al een versie te ontwikkelen die suboptimaal de probleemstelling kon beantwoorden. Dit bleek echter niet het geval te zijn. De data aangeleverd aan de agent, en machine learning modellen in het algemeen, moet zorgvuldig gekozen worden en met een eventuele preprocessing stap geprepareerd worden voor het policy network. Dit om geen overbodige data aan te bieden die de focus van de agent zou kunnen verleggen.

Alsook heb ik geleerd dat het van groot belang is tijens het iteratief verbeteren van deze agent dat de resultaten van een iteratie steeds grondig onderzocht moeten worden. Alsook werd duidelijk dat naar een volgende iteratie ik geen te grote verandering moet introduceren om controle te behouden over het verbeterproces van de agent. De fout die ik maakte was dat ik vaak enkel naar de rewardcurve over trainingsessies heen keek en niet naar de daadwerkelijke performance van het model, namelijk hoe de gegenereerde sequenties eruitzien in de praktijk ten opzichte van het gewenste resultaat. Alsook bracht ik tussen iteraties vaak meerdere aanpassingen aan aan het systeem waardoor het moeilijk te beseffen was welke verandering een positief effect hadden en welke handelingen een negatief effect hadden op de performantie van de agent. In deze fase werd ik door de promotoren van deze thesis hier vaak terecht op gewezen, wat me zeer hard geholpen heeft dit iteratief verbeterproces gecontrolleerd te behandelen. Vaak kreeg ik ook de aanbeveling te controlleren of na nieuwe ontwikkelingen versimpelde versies van de probleemstelling, zoals een stilstaande camera, nogsteeds presteerden zoals gewenst. Deze sturing heeft me veel bijgeleerd over de noodzaak om structuur tussen iteraties van het project te behouden en geen stappen over te slaan in dit proces. Ik ben sterk overtuigd dat als ik dit besef voor aanvang van deze thesis had dat het model nog verder kon staan dan de huidige implementatie.

Om te concluderen heb ik zeer veel bijgeleerd over het efficiënt ontwikkelen van een project van deze grootte, waar ik mijn promotoren zeer dankbaar voor ben. Gaandeweg zijn er regelmatig bijsturingen nodig geweest om het project gefocust te houden op zijn eerstvolgende doel voor ogen, wat op termijn wel wat tijd gekost heeft. Echter denk ik dat dit leerproces tijdens de Masterproef niet abnormaal is en naar het einde van het project toe het ontwikkelingsproces en beheersing van de kennis rond het onderwerp al een stuk beter waren. Ik was zeer tevreden met de feedback die ik tijdens de proef kreeg en stond steeds open voor een kritische kijk om bij te leren hoe ik mijn proces kon verbeteren. Als ik deze proef met deze voorkennis opnieuw zou moeten beginnen zou ik zeker meer inzetten op structuur van de werkwijze en een bredere onderbouwing van de kennis rond de gebruikte systemen voor het aanvangen van de ontwikkeling

van de agent. Echter ben ik wel tevreden met de ontwikkelingen die naar boven zijn gekomen om een reinforcement learning agent effectiever te laten leren voor deze probleemstelling. Ook was het zeer interessant om dieper in te gaan op de topics van Reinforcement learning, progressive streaming, optimalisatie van graphics rendering en nog veel meer omdat dit voor mij persoonlijk werkvelden zijn die me zeer interesseren die samenkomen in deze thesis.

6.2 future work

Zoals eerder in dit hoofdstuk besproken laat de huidige state van het project nog veel ruimte voor verbetering. In dit deel lijsten we op wat de volgende ontwikkelingen zijn die op de planning stonden om toegevoegd te worden aan het project om de agent generaliseerbaar te maken naar een goede prestatie op ongeziene glTF scenes. Op deze manier kan het model in de prakijk algemeen ingezet worden om 3D scenes progressief te streamen.

- **Ontwikkeling custom policy network:**
  De huidige versie van het project traint een agent die een standaard policy network omvat vanuit de SB3 [35] library zonder expliciete parameters om deze architectuur te wijzigen. We merken echter dat deze oplossing geen permutation invariance beheerst, wat noodzakelijk is om uit te breiden naar een algemeen model dat werkt op meerdere 3D-scenes. Door onderzoek te doen naar de architectuur van het policy network en dit deel eventueel als een custom policy te herschrijven zoals beschreven in Sectie 3.2 zou het mogelijk kunnen zijn deze permutation invariance te realiseren en de agent algemeen toepasbaar te maken.

- **variabele camerastandpunten:**
  Op dit moment staat de camerapositie steeds vast op het middelpunt van de scene. Een volgende stap van het ontwikkelingsproces zou kunnen zijn om enkele interessante camerastanpunten en kijkrichtingen te gebruiken tijdens de training. Zo zou er gekeken kunnen worden of het model dan ook presteert op ongeziene standpunten, gelijkaardig aan het experiment in deze thesis rond de kijkrichting van de stilstaande camera. Zelf hebben we deze stap niet gerealiseerd omdat het probleem van permutation dependence een hogere prioriteit kreeg, maar dit zou een logische volgende stap in de ontwikkeling zijn.

- **bewegende camera tijdens training:**
  Om nog een stap verder te nemen dan in de vorige bullet zou een bewegende gebruiker gesimuleerd kunnen worden tijdens de training door de camera tussen timesteps heen laten verplaatsen en draaien volgens een interpolatie van een willekeurig traject. Bij deze toevoegingen zouden zaken zoals optical flow eventueel mee in de rewardberekening in rekening genomen kunnen worden om betere performance op bewegende gebruikers te realiseren.

- **Node shuffle:**
  Wanneer permutation invariance van het model gerealiseerd zou kunnen worden kan er in de trainingsfase een volgende stap genomen worden naar generalisatie door tussen episodes heen de volgorde van de nodes geleverd aan de agent willekeurig te herschikken. De combinatie van deze willekeurige volgorde en het gebruik van cameraspace bounding boxes zouden zelfs in een enkele scene voor een grote varieteit aan trainingsdata kunnen leveren. Hiermee zal de focus volledig verlegd worden van het zoeken naar interessante indices in de action space naar het linken van goede bounding boxes aan de juist acties.

- **Uitbreiding naar ongeziene scënes:**
  Het uiteindelijke doel van het model is te functioneren op eender welke glTF scënes. Om dit te realiseren zullen in de trainingsfase meeerdere scenes gebruikt kunnen worden waartussen de agent alterneerd tussen episodes. Er is in de custom environment functionaliteit voorzien om deze uitbreiding te ondersteunen, echter functioneerde het model nog niet goed genoeg om deze stap te maken.

6.3 Beantwoording van de onderzoeksvraag

Ten slotte, om de onderzoeksvraag van deze thesis te beantwoorden, zijnde of dat reinforcement learning succesvol toegepast kan worden op progressive streaming van glTF 2.0-data, kunnen we stellen dat er absoluut zeer veel potentieel zit in deze methode. Met verdere verscherpingen van het project, voornamelijk gefocust op uitbreiding naar arbitraire scènes, is het mogelijk deze aanpak in de context van 3D-streaming daadwerkelijk toe te passen. De methodes zijn reeds succesvol toegepast op ABR-streaming, wat aantoont dat reinforcement learning effectief kan inspelen op dynamische omgevingen waarin beslissingen in real-time genomen moeten worden op basis van schaarse middelen zoals bandbreedte. Door deze principes te vertalen naar het domein van 3D-graphics, opent dit de deur naar efficiëntere en adaptieve laadstrategieën die de gebruikerservaring in streamingtoepassingen kunnen verbeteren.

Er kon ook reeds vastgesteld worden dat deze proof-of-concept op een vast standpunt in een scène met bewegende camera een voorgaande heuristische methode kon overtreffen. Dit toont aan dat het model in staat is om relevante visuele informatie uit camerastandpunten te benutten om efficiëntere laadbeslissingen te nemen, zelfs binnen een dynamische context.

Deze resultaten vormen een stap richting het ontwikkelen van intelligente streamingalgoritmes die zich niet enkel aanpassen aan netwerkcondities, maar ook aan de visuele structuur van een scène en de beleving van de gebruiker.

Bibliografie

[1] Vlaams supercomputer centrum. https://www.vscentrum.be, 2024. Geraadpleegd op 27 mei 2025.

[2] Joshua Barczak. Octree acceleration structures. https://slideplayer.com/slide/14820139/, 2018. CMSC435 UMBC, gepubliceerd door Lillian Hart. Geraadpleegd op 18 juni 2025.

[3] Stable Baselines3. Stable baselines3 - ppo documentation. https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html, 2023. Accessed: 2025-05-22.

[4] Blender Online Community. Blender - a 3d modelling and rendering package. https://www.blender.org, 2024. Version 4.1, accessed June 2025.

[5] Ricardo Cabello. Three.js. https://threejs.org/. Accessed: 2025-05-22.

[6] Stable-Baselines3 Contributors. Policy networks — stable baselines3 documentation, 2025. Accessed: 2025-05-27.

[7] Richard Dosselmann and Xue Dong Yang. A comprehensive assessment of the structural similarity index. *Signal, Image and Video Processing*, 5(1):81–91, 2011.

[8] Express Contributors. Express - node.js web application framework. https://expressjs.com/. Geraadpleegd op 27 mei 2025.

[9] Farama Foundation. Gymnasium. https://gymnasium.farama.org/, 2023. Accessed: 2025-05-22.

[10] Suliu Feng, Caihong Wang, and Xiuhua Jiang. Adaptive streaming algorithm based on reinforcement learning. *IOP Conference Series: Materials Science and Engineering*, 768(7):072069, mar 2020.

[11] GamersNexus. Deep dive: How vxao, frustum tracing, & fluid combustibles work. https://gamersnexus.net/guides/2355-deep-dive-how-vxao-frustum-tracing-and-flow-work, March 2016. Geraadpleegd op 18 juni 2025; bevat illustratieve figuren over z-buffering, frustum tracing en VXAO.

[12] GeeksforGeeks. Actor-critic algorithm in reinforcement learning. https://www.geeksforgeeks.org/actor-critic-algorithm-in-reinforcement-learning/, 2025. Geraadpleegd op 20 mei 2025.

[13] Majid Ghasemi and Dariush Ebrahimi. Introduction to reinforcement learning, 2024.

[14] Yu Guan, Chengyuan Zheng, Zongming Guo, Xinggong Zhang, and Junchen Jiang. Pano: Optimizing 360° video streaming with a better understanding of quality perception. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2019. Extended version available at arXiv.

[15] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications, 2019.

[16] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[17] Alain Horé and Djemel Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th International Conference on Pattern Recognition*, pages 2366–2369, 2010.

[18] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. In Roman Barták, Fazel Keshtkar, and Michael Franklin, editors, *Proceedings of the Thirty-Fifth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2022, Hutchinson Island, Jensen Beach, Florida, USA, May 15-18, 2022*, 2022.

[19] Xiaolan (federerjiang) Jiang. Plato: Viewport adaptation based bitrate adaptive vr video streaming. `https://github.com/federerjiang/Plato`, 2018. Software project, MIT License. Laatste versie gecloned in juni 2025.

[20] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning. In *IEEE Conference on Games (CoG)*, pages 479–486. IEEE, 2020.

[21] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering, 2023.

[22] Khronos Group. glTF 2.0 Specification. `https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html`, 2022. [accessed 2025-06-15].

[23] Khronos Group. Rendering Pipeline Overview. `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`, 2022. [accessed 2025-06-15].

[24] Minsu Kim and Kwangsue Chung. Adaptive quality control scheme based on vbr characteristics to improve qoe of uhd streaming service. *Advances in Electrical and Computer Engineering*, 19:89–98, 02 2019.

[25] Masanari Kimura, Ryotaro Shimizu, Yuki Hirakawa, Ryosuke Goto, and Yuki Saito. On permutation-invariant neural networks, 2024.

[26] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[27] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks, 2019.

[28] Wouter Lemoine and Maarten Wijnants. Progressive network streaming of textured meshes in the binary gltf 2.0 format. In *Proceedings of the 28th International ACM Conference on 3D Web Technology*, Web3D '23, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Luke Miller. pygltflib: Python library for reading, writing and handling gltf files, 2025. Version 1.16.4, accessed May 26, 2025.

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[31] OpenAI. Proximal policy optimization. `https://openai.com/index/openai-baselines-ppo/`, 2017. Accessed: 2025-05-24.

[32] OpenAI and Contributors. Gym library. `https://www.gymlibrary.dev/`, 2024. Accessed: 2025-05-22.

[33] Ioannis Pantazopoulos and Spyros Tzafestas. Occlusion culling algorithms: A comprehensive survey. *Journal of Intelligent and Robotic Systems*, 35(2):123–156, 2002.

[34] Wei Quan, Yuxuan Pan, Bin Xiang, and Lin Zhang. Reinforcement learning driven adaptive vr streaming with optical flow based qoe. In *2020 IEEE 18th International Conference on Industrial Informatics (INDIN)*, volume 1, pages 231–236, 2020.

[35] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[36] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[37] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[38] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[39] Yuang Shi, Géraldine Morin, Simone Gasparini, and Wei Tsang Ooi. Lapisgs: Layered progressive 3d gaussian splatting for adaptive streaming, 2025.

[40] Iraj Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[41] Stable-Baselines-Team. Stable-baselines3 contrib. `https://github.com/Stable-Baselines-Team/stable-baselines3-contrib`, 2021. Accessed: 2025-05-26.

[42] Thomas Stockhammer. Dynamic adaptive streaming over http –: standards and design principles. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, page 133–144, New York, NY, USA, 2011. Association for Computing Machinery.

[43] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction, 2018.

[44] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[45] Google Chrome Team. Puppeteer: Node.js library for controlling headless chrome, 2024. Accessed: 2025-05-27.

[46] The A11Y Project. A11y project checklist, 2025. Geraadpleegd op 18 juni 2025.

[47] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024.

[48] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[49] Chengxi Yu, Yali Du Liu, Haitham Xu, Yuan Tian, Ziyu Wu, Aravind Rajeswaran, and Ryan Lowe. The surprising effectiveness of ppo in cooperative multi-agent games. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent*

Systems (AAMAS), pages 425–433. International Foundation for Autonomous Agents and Multiagent Systems, 2021.

[50] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets, 2018.

[51] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.