Down the 1000 Rabbit Holes

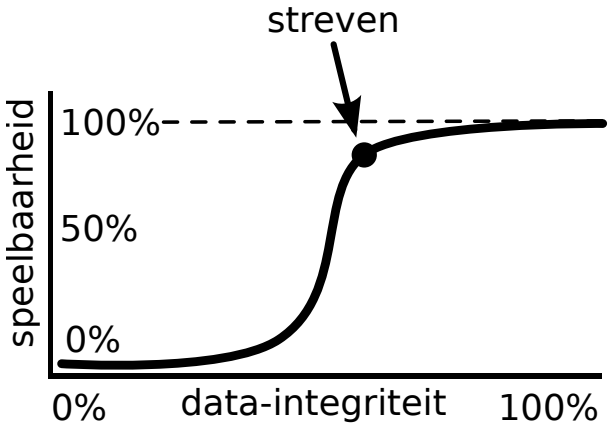
Nick Overdijk Rik Harink Joshua Moerman

2 april 2010

Introductie

Onze doelstelling is een vloeiende en ononderbroken (multiplayer-)game-ervaring. De vloeiendheid en speelbaarheid hangt af van in hoeverre de data met andere gebruikers is gesynchroniseerd en wat de gebruiker acceptabel vindt. Omdat dit laatste onmeetbaar is, gaan we proberen om een zo hoog mogelijke data-sync te krijgen zonder de computers van de gebruikers volledig te moeten benutten (dit omdat mensen naast de game vaak ook andere dingen op de computer hebben draaien, die mogelijk ook een connectie onderhouden). Om de synchronisatie zo flexibel mogelijk te maken en om synchronisatie te bewerkstelligen, gaan we een framework maken. Dit framework stelt gamedevelopers in staat makkelijk de gamedata te synchroniseren. Naast games is dit framework misschien ook bruikbaar in andere situaties.

De mate waarin de data gesynchroniseerd is noemen we data-integriteit. De data-integriteit heeft een verband met de speelbaarheid en vloeiendheid van de game. We verwachten dat het verband er als volg uit ziet:



Figuur 1: Een vloeiende game-ervaring eist niet 100% data-integriteit

We zouden dit kunnen testen, maar het is aannemelijk dat er een punt is waarbij de vloeiendheid niet meer wordt, naarmate de dataintegriteit hoger wordt. Omdat de bandbreedte een probleem kan vormen bij spellen met veel data, is het synchroniseren lastig. Maar doordat de dataintegriteit niet 100% hoeft te zijn, kunnen we het doel (een vloeiende game-ervaring) toch bereiken.

Met dit onderzoek en dit framework proberen we de vraag "Hoe maximaliseer je data-integriteit, terwijl je het bandbreedtegebruik laag houdt, bij online games?"te beantwoorden.

Inhoudsopgave

1 Data Synchronisatie **3**
1.1 Framework 3
1.2 Communicatie 3

2 Datasoorten **4**
2.1 Onafhankelijk en afhankelijk 4
2.2 Voorspellen 4

3 Triggers **5**
3.1 Time Based Triggers 5
3.2 Event Based Triggers 5

4 Out of Sync **7**
4.1 Out of Sync preventie 7
4.2 Out of Sync detectie 7
4.3 Checksum 7
4.4 Out of Sync Recovery 8

5 Flexibiliteit **10**

6 Conclusie **11**

1 Data Synchronisatie

1.1 Framework

Een vloeiend spel eist een heleboel communicatie, maar door beperkte bandbreedte is het vaak niet mogelijk alle data snel genoeg door te sturen. Dus er moet geoptimaliseerd worden. In ons framework hebben we de volgende structuren en methoden om de data-integriteit optimaal te maken.

- Datasoorten: onafhankelijke data en afhankelijke data
- Triggers
- Out of Sync preventie
- Out of Sync detectie
- Out of Sync herstel

Zoals je ziet zijn er twee soorten datasoorten en we zullen later zien wat deze zijn. Het is belangrijk om te weten dat je voor het een of het ander kunt kiezen, maar vooral ook dat je beide in een bepaalde verhouding kunt hebben.

1.2 Communicatie

In ons onderzoek gebruiken we in eerste plaats een server-client model. Waarbij de client alleen communiceert met de server en de server op zijn plaats die data doorstuurt naar de andere clients. De gebruiker hoeft zijn data dus maar naar 1 plek te versturen. Door het gebruik van een server hebben we een plek waar de game vast staat, we noemen de server dan de baas van de game. De baas heeft altijd gelijk, en door de server de baas te maken, is elke gebruiker gelijk en is er dus een eerlijk spel.

Een andere manier van communicatie zou zijn een peer-to-peer netwerk, waarbij de gebruikers direct een connectie hebben met de andere gebruikers. Dit is in de praktijk vaak lastig, omdat een connectie maken naar een andere gebruiker vaak onmogenlijk is, door routers, firewalls, etc. Een ander groot nadeel van peer-to-peer is dat de gebruikers naar iedereen data moeten sturen en dus een grotere upload-bandbreedte moeten hebben. Veel mensen hebben niet zo veel upload-bandbreedte, servers vaak wel, want die zijn daarop gebouwd. In peer-to-peer netwerking is iedereen gelijk en kun je dus eigenlijk geen baas kiezen (een gebruiker als baas zou oneerlijk kunnen zijn).

We kiezen in ons framework voor een server-client model. In feite zou ons framework net zo goed werken met peer-to-peer, maar dingen als out of sync detectie zullen minder helder zijn.

2 Datasoorten

2.1 Onafhankelijk en afhankelijk

We kunnen de data opdelen in onafhankelijke data, dat is data die, wanneer deze wordt ontvangen, altijd exact de juiste staat van een speler aangeeft op een bepaalde tijd (positie, snelheid, draaiing, etc.). De afhankelijke data is dan data waarbij er vanuit wordt gegaan dat de onafhankelijke data nog klopt, het zijn dus meer de veranderingen van een speler (bijvoorbeeld; "Speler X gaat nu naar links", "Speler X gaat dood"). De afhankelijke data betekent alleen iets als de onafhankelijke data ook bekend is. Als iemand zegt dat hij naar links gaat en je weet niet waar hij eerst stond, betekent het niks. Maar als je eenmaal weet waar die stond, is de onafhankelijke data eigenlijk niet meer nodig (in het ideale geval dat de berekeningen 100% goed uitgevoerd worden).

De data die een typische game-client over zijn eigen speler zal versturen, zou als volgt kunnen zijn:

- Onafhankelijke data
  - Positie
  - Draairichting (noord, west, oost, zuid)
  - Levensstaat (dood/levend/levenloos)
- Afhankelijke data
  - Ik draai nu naar links
  - Ik draai nu naar rechts
  - Ik ben gestopt met draaien
  - Ik bots tegen speler x

Onafhankelijke data is over het algemeen groter in het aantal bytes (positie moet bijvoorbeeld zo precies mogelijk worden meegegeven). Berichten zoals "Ik draai naar links"zijn heel erg klein te maken. We verwachten dat in grotere games de afhankelijke data beperkt blijft, terwijl de onafhankelijke data enorm kan groeien (denk bijvoorbeeld aan wapenkeuze, kleding van karakters, etc). Het is dus preferabel om afhankelijke data te versturen.

2.2 Voorspellen

Met alleen afhankelijke data, kun je dingen zoals positie voorspellen. Dit is te doen met extrapoleren [1]. Een voorbeeld van extrapolatie: als je op een gegeven moment de positie van een speler weet en de speler beweegt in een bepaalde richting, dan kun je die positie steeds iets verder zetten. Nu zegt de speler dat hij naar links gaat. Zonder dat hij zijn positie meegeeft kan je toch doorgaan met voorspellen, maar nu extrapoleer je naar links.

Het gevaar hiervan is dat als jij een dergelijk berichtje iets te laat binnenkrijgt, de speler niet op tijd naar links zal gaan. Zijn positie zal dus verschillen van zijn werkelijke positie. Het is dus niet verstandig om geheel te vertrouwen op afhankelijke data.

[1] In andere synchronisatie situaties, zoals google wave, is dit soort voorspellen lastig. Ons framework werkt dus het beste in situaties met voorspelbare data, zoals in games.

3 Triggers

Een trigger is een abstract begrip, een trigger geeft aan dat de client of server data moet versturen. De gamedeveloper zal zelf kunnen kiezen wanneer hij wil dat de data wordt gesynchroniseerd wordt, dit kan hij doen door middel van een trigger. Triggers zijn dus de momenten waarop de data gesynchroniseerd moet worden. Dit kan bijvoorbeeld een timer zijn (die elke seconde triggert) of een event (als er dus iets belangrijks gebeurt). We analyseren zowel de timebased triggers als de eventbased triggers, dit zijn gebruikelijke triggers in games, maar je zou ook andere triggers kunnen bedenken. Om dat we twee soorten data hebben en deze dus afzonderlijk kunnen sturen, zijn er twee triggers. Een onafhankelijke trigger zal alle data van een client of server versturen en een afhankelijke trigger zal alleen de verandering in data sturen.

3.1 Time Based Triggers

De timebased trigger vuurt om de zoveel seconden af. Dit is de meest eenvoudige trigger. Bij dergelijke triggers is het nauwlijks mogelijk een verandering door te geven (want op het exacte moment van deze trigger verandert er waarschijnlijk niks), dus is een time based trigger geschikt voor onafhankelijke data. De time based trigger is dus een onafhankelijke trigger.

Voordelen

- Simpel in te bouwen. Dit is weliswaar geen voordeel voor de uiteindelijke game, maar development tijd is ook belangrijk.
- Regelmatige synchronisatie. Bij het voorspellen zagen we dat er een fout kan optreden. Door regelmatig een onafhankelijke trigger te doen, verhelp je dit probleem.

Nadelen

- Potentieel hogere belasting op het netwerk. Omdat onafhankelijke data zal worden verstuurd. Het is mogelijk om niet alle onafhankelijke data te versturen, maar alleen de belangrijke delen. Hiervoor is een op prioriteit gesorteerde lijst nodig met alle data.
- Er wordt mogelijk onnodig data verstuurt. Als er niets verandert in je game, wil je eigenlijk geen data hoeven sturen, maar dat gebeurt met een time based trigger toch.

3.2 Event Based Triggers

Met eventbased synchronisatie bedoelen we dat je alleen synchroniseert als er iets gebeurt. Events kunnen bijvoorbeeld zijn:

- Speler draait
- Speler stopt met draaien

- Speler botst

Aangezien er met deze triggers iets verandert in de gamestate is het typisch een afhankelijke trigger. Ons framework kan zien welke data veranderd is en kan slechts deze (kleine hoeveelheid) data sturen.

Voordelen

- Geen onnodige synchronisatie. Bij time based triggers zagen we dat er onnodig data werd verzonden als er niks gebeurt. Dit is nu niet het geval.
- Ideaal voor afhankelijke data. Omdat afhankelijke data vaak kleiner is, wordt hiermee de gebruikte bandbreedte beperkt gehouden.

Nadelen

- Mogelijk veel data. Als er veel veranderingen zijn, kan het voorkomen dat dit losse triggers worden, waardoor er vaak data wordt verstuurd. Dit is te verhelpen door deze afhankelijke triggers samen te voegen in een enkele trigger (deze kan dan zowel afhankelijk als onafhankelijk zijn).
- Mogelijk incorrecte voorspelling Als een client en bericht niet goed ontvangt, zal de voorspelling tot het volgende bericht niet kloppen (en misschien zelf na dat bericht niet). Om dat fout te corrigeren zal onafhankelijke data nodig zijn. Je kunt die onafhankelijke data meesturen met deze trigger of later een keertje sturen met bijvoorbeeld een time based trigger.

4 Out of Sync

Een client noemen we out of sync als hij niet meer de juiste data heeft over de game. Dit is dus in feite wat we met ons framework bestrijden. Ten eerste willen we dit preventeren, maar mocht het toch voorkomen, dan willen we dat detecteren en vervolgens herstellen. Je bent dus out of sync als je voorspelling niet meer klopt.

4.1 Out of Sync preventie

We hebben bij de afhankelijke en onafhankelijke data al methoden gezien om out of sync te voorkomen. Zo hebben we de volgende punten al gezien:

- Je kunt bijvoorbeeld om de zoveel afhankelijke triggers onafhankelijke data meesturen. Hierdoor heb je weer een nieuw beginpunt van je voorspelling.
- Veel time based triggers gebruiken. Hierdoor is je voorspelling om de zoveel tijd correct, omdat je veel onafhankelijke data stuurt.

Andere preventieve maatregelen zijn:

- Als er bandbreedte over is, kan de server ervoor kiezen meer onafhankelijke data te sturen (want hij heeft alle correcte data).
- Clients met een hoge ping (en dus hoge latency) niet mee laten doen. Dit klinkt misschien flauw, maar het zal wel helpen. Veel games doen dit al.

4.2 Out of Sync detectie

Om te controleren of een client out of sync is moet hij al zijn data controleren met de data die de server heeft. Nu is het niet handig als alle data zomaar verstuurd wordt en gecontroleerd wordt, dit kan namelijk heel veel data zijn (meer dan de bandbreedte aan kan). We zoeken dus een manier om veel data te vergelijken, zonder al te veel te hoeven sturen.

Dit kunnen we bereiken met een checksum. Hierbij vat de client zijn gamestate (dus bijvoorbeeld posities en levenstaten van alle spelers) samen in een enkel getal. De client stuurt dit getal naar de server en de server doet dezelfde berekening. Nu kan de server de twee getallen vergelijken en bepalen of de client out of sync is (namelijk als de getallen niet gelijk zijn).

4.3 Checksum

Er is een hele simpele checksum denkbaar; alle posities en andere data van alle spelers worden opgeteld en afgerond tot een getal. Deze checksum levert zeker een klein getal op en geeft dus geen overhead aan het netwerkgebruik. Maar er zijn twee nadelen aan zulke simpele checksums:

- Het geeft niet aan in welke mate de client out of sync is.
- Het geeft niet aan welke data de client fout heeft.

Voor ons framework willen we een slimmere checksum die deze punten wel nastreeft en toch klein is (zodat je netwerk niet overbelast wordt en toch vaak kunt checken). We willen de mate van out of sync zijn kunnen meten, zodat we een treshold kunnen instellen. Zodat je pas ingrijpt als de client te veel out of sync is. Daarnaast willen we ook selectief kunnen herstellen, zodat we nooit te veel data sturen en daarvoor moeten we weten welke data de client fout heeft.

We zullen een paar voorbeelden van slimme (delen van) checksums bespreken met betrekking tot specifieke data.

- Positie, dit is de positie van een speler. We gaan er even van uit dat dit per speler wordt verstuurd en we gaan dus kijken hoe we dit het best kunnen beperken tot een klein aantal bits. We kunnen de positie bijvorbeeld afronden op tientallen [2]. Je controleert dan alleen op de globale positie van een speler. Het nadeel hiervan is dat de globale positie vaak wel goed is en een kleine afwijking is dan dus niet te controleren. Beter zou dus zijn om wel de kleine afwijkingen op te nemen in de checksum. Om toch weinig bits te versturen kan je dus juist alleen de laatste getallen van je positie doorsturen. Je weet dan niet waar de speler in het speelveld zit, maar aangezien je vaak globaal de positie wel goed hebt, kun je wel exact zeggen waar de speler is. Dit laatste is dus wel handig voor de checksum.

- Spelers. Aangezien we een checksum willen waarbij we aan kunnen geven welke data out of sync is, kunnen we bijvoorbeeld elke speler in het veld een eigen checksum geven en een lijst checksums doorgeven naar de server. Nu is het dan wel van belang dat de server weet bij speler de checksum hoort, zodat hij zijn eigen checksums kan vergelijken. Een mogelijkheid om dit te bewerkstelligen is door de spelers een id te geven en dit ook in de checksum op te nemen. Maar aangezien dat extra bits zijn is dit niet preferabel. Beter zou dan zijn dat de server een vaste lijst heeft en dat de client zich aan deze lijst houdt, hierdoor bespaar je bits en kan je toch selectief de data checken. Een andere optie zou een checksum zijn warbij de volgorde van de spelers niet uitmaakt. Je kan hierbij denken aan een checksum per data-element (bijvoorbeeld positie en richting). Hierbij gooi je wel alle spelers op een hoop, maar heb je toch verschillende checksums en kan je alsnog specifiek data herstellen.

4.4 Out of Sync Recovery

Als een client out of sync is geraakt, zijn er een paar manieren dit te herstellen. De server kan alle data naar een client sturen, of slechts een deelverzameling van alle data.

Alle data versturen
Voordelen

- Erg simpele methode

Nadelen

- Extreme overkill als de data van de client maar voor, stel 10% out of sync is.

[2] Aangezien we op een computer zitten is het beter om met machten van 2 te werken, dus afronden op tientallen zal niet gebeuren. Afronden gebeurt eigenlijk door de laatste zoveel bits te negeren.

- Kan voor bursts van netwerk-gebruik zorgen, die zorgen weer voor een hoge latency en daardoor nog meer out of sync data.

Een deelverzameling van alle data versturen
Voordelen

- Er wordt nu alleen verstuurd wat nodig is en er is dus meer van de bandbreedte over.

Nadelen

- Er is een slimmere checksum vereist, waarbij duidelijk is waar de client fout zit.

5 Flexibiliteit

We hebben gezien hoe verschillende concepten een synchronisatie tot stand kunnen brengen. Maar het is wel de vraag welke concepten beter zijn. Kies je voor veel afhankelijke data of juist alleen voor onafhankelijke data? Wat voor Triggers zijn het handigst? We willen een flexibel framework dat tijdens het draaien tussen deze concepten kan wisselen, sterker nog, het framework moet ook er tussenin kunnen zitten. We nemen het hele spectrum van onafhankelijke data en afhankelijke data en willen op het optimale punt zitten. Hiervoor is een stukje AI nodig in het framework, wat de data-integriteit meet en vervolgens dingen varieert en nagaat of er een verbetering is.

6 Conclusie

Optimale synchronisatie is dus afhankelijk van een aantal onderdelen. Je synchronisatie framework moet ten eerste zo in elkaar zitten dat het voorkomt dat de data out-of-sync raakt. Hier kun je enkele methodes en trucjes voor gebruiken die er op neer komen dat je extra data of extra informatie verstuurd in de bandbreedte die je anders over zou hebben.

Mocht het nu zo zijn dat dit niet voldoende is dan moet je een manier hebben om te detecteren dat een client out-of-sync is. Hiervoor zijn checksums ideaal maar binnen deze checksums heb je ook weer twee mogelijkheden, een slimme en een simpele checksum. De simpele checksum kost minder rekenkracht maar hiermee weet je alleen of een client out of sync is en niet hoeveel hij out-of-sync is en waar hij precies out-of-sync is. Met de slimmere checksum weet je deze twee dingen wel en kun je dus intelligenter repareren, deze checksum kost echter wel wat meer rekenkracht en vereist een hogere upload capaciteit van de client.

Als je eenmaal hebt gedetecteerd dat een client out-of-sync is dan moet je dit repareren. Bij de simpele checksum betekend dit de gehele gamestate opsturen naar de client. Bij de slimme checksum kun je echter alleen dat deel van de gamestate sturen die ook daadwerkelijk out-of-sync is.

Als je al deze stappen hebt geimplementeerd dan heb je een goed synchronisatie-framework.