SERVER SIDE WALL-HACK PREVENTION VIA HARDWARE-ACCELERATED RAY-TRACING

Author: Baktash Abdollah-shamshir-saz

Created: 21-12-2020

Updated: 5-11-2023

Version: 1.25

US Provisional Patents: 63/135,354

63/151,289

US Utility Patent: 11/771,997 (US20220219086A1)

CONTENTS

0. Glossary of Known Terms

Graphics Processing Unit (GPU): An electronic hardware unit designed to provide visuals for computers. It can either be a separate piece of hardware installable on a motherboard, integrated into a motherboard or provided alongside a CPU on the same chip. Modern GPUs are backed by massively parallel processors capable of intensive mathematical computations.

General Purpose Graphics Processing Unit (GPGPU): Utilizing a GPU's massively parallel processors for the purpose of performing processing that may not be visuals-related.

Ray-tracing: a computer graphics technique whereby a line segment is tested against an amalgamation of 3D triangles in space in order to find an intersection.

Hardware accelerated ray-tracing: ray-tracing accelerated via specialized hardware. Ray-tracing is otherwise known to be a computationally intensive process (i.e. when done using software in either a parallel or serial fashion.)

Path-tracing: a technique that utilizes multiple back-to-back and stochastic applications of ray-tracing to simulate a travelling photon. It starts casting a ray of light from the viewer and aims to end it at a light source. This technique and its modern variants are used to generate photorealistic imagery in movies and, more recently, in video games.

Transport Path (Path-tracing): the complete path that a photon travels from a light source to the eye.

Throughput (Path-tracing): the carried amount of light as the path of a photon is being discovered during the path-tracing process.

Next Event Estimation (Path-tracing): Next Event Estimation (dubbed NEE) is a technique in path tracing whereby a ray is extended from a hit point towards a light source intentionally to increase the chances of interaction with a light source. This is commonly done in graphical applications to reduce noise especially when light sources are small (and thus harder to hit) or exist as a single point in space.

Ray payload (Ray-tracing): a small data store that contains information about a sequence of rays traversing the scenery. This data unit has emerged in modern day GPU-based hardware accelerated ray-tracing as a necessary means of accumulating information along a transport path and is recommended to be kept small in order to avoid unnecessary memory and bandwidth pressure on the hardware.

Shader Binding Table or SBT (Ray-tracing): A table that contains sets of code used for ray-tracing. Each set usually contains code to execute when all geometry is missed, any piece of geometry is hit or the un-ignored geometry closest to the ray origin is found. A ray-tracing application may have one or more sets entered into such a table to represent different kinds of rays

each with custom logic. This term has evolved from modern-day GPU-based hardware-accelerated ray-tracing hardware-software ecosystems.

Acceleration Structure (Ray-tracing): a top-down structure used to hierarchically group geometric primitives such as triangles, distance-fields or any other shape representations. It is used to accelerate traversal and access of such primitives and shapes resulting from intersection queries defined by rays (two points defined in three-dimensional space). A ray intersection query will traverse down the tree, performing rough and quick intersection tests on higher-level nodes. If these rough tests do not yield an intersection, all children branching down from those nodes are ignored. Thus, significantly increasing the speed with which primitives or shapes of interest are obtained.

Bottom-level Acceleration Structure or BLAS (Ray-tracing): an acceleration structure that simply represents a small chunk of triangles or shapes arranged together on a top-down structure. This is yet another term evolved from modern-day GPU-based hardware-accelerated ray-tracing hardware-software ecosystems.

Top-level Acceleration Structure or TLAS: an arrangement of BLASs on yet another acceleration structure representing an entire scene. This two-level design allows one to compose different scenes by mixing and matching different BLASs together. Much like BLAS, this term has also evolved from modern-day GPU-based hardware-accelerated ray-tracing hardware-software ecosystems.

Pseudo Random Number Generator (PRNG): A computer subroutine that generates a seemingly random sequence of numbers.

Russian-roulette (Path-tracing): a mode of decision making in path-tracing whereby the continued existence of a ray is determined via a dice-roll.

Diffuse (material type): A type of material characterized by a very rough surface that has microstructures pointing in every direction. Unfinished wood is considered diffuse. This type of surface reflects incoming light to every direction.

Indirect Diffuse (Lighting): Light visible on a diffuse surface reflected from another surface. For example, a red table cloth reflecting direct sunlight on drywall. The red light reflected on drywall would be considered indirect diffuse lighting.

Ambient Occlusion or AO: Darkness visible in crevices due to increased difficulty for light to escape those geometric conditions.

Screen-space AO: A method for approximating AO by relying on geometric data available alongside color information on screen. This method is inaccurate as it cannot account for AO appearing from objects that are not immediately visible or out of view. It is normally used due to its memory requirements being solely tied to screen resolution.

Global AO: Any method that approximates AO via taking into account all relevant scene geometry. This approach does not suffer from the aforementioned issues with screen-space AO.

Voxel Accelerated AO or VXAO: A global AO method using brick-like (or 'voxelized') representations of a scene.

Game-client: the machine used by a player to play the game.

Game-server: an authoritative computer that exists as a broker of information between game clients in online video games. Video games that utilize the client-server model necessitate such machines. Games that utilize peer-to-peer knowledge distribution do not have traditional game servers. Due to this, they are also more susceptible to cheating and abuse.

Wall-hacking: a colloquial term that has emerged to describe a particular kind of cheating in online first person and third person shooter video games. This kind of cheating enables a user to be able to observe enemy combatants through otherwise obscuring geometry, thereby providing the user with otherwise inaccessible knowledge and an unfair advantage over the opposing team.

Aim-botting: a colloquial term emerged to describe a type of cheating in online shooter games whereby the computer assists the user to effortlessly aim at weak points of enemy combatants as described by game logic. These weak points could for example be the enemy's head or any other part known to be susceptible to increased damage from virtual hostile munitions.

DRM: Digital Rights Management. Software intended to prevent unauthorized copying and use of copyrighted material.

Banning (online games): forbidding a player from participating in an online game either temporarily or permanently.

Root-kit: software that operates at operating system level by posing as a system driver. When operating at this level, it is able to hide its presence by the virtue of pre-installing before known counter-measures. With this advantage it simply changes the operation of the system at any desired level with full impunity.

Player armature: a skeletal shape representing player poses at any point during a gameplay session.

Potentially Visible Set or PVS: a grouping of objects or individual primitives (triangles) within objects whereby elements of the set are potentially –but not guaranteed to be – visible.

Bounding Box: a virtual box used to represent all geometry contained within. Mostly used for a single entity or player. They tend to contain a lot of empty space.

Occlusion Culling: determining whether a bounding box or a complete piece of geometry is visible from an observer's vantage point given all – or the most important – obstructions.

Player Frustum: a virtual pyramid stretched out from the vantage point of an observer that defines the observer's field of view.

Graphics API: an application programming interface used to render computer graphics.

Headless Instance (Graphics API): an instance of a graphics API used to render only to a piece of memory instead of a display output (i.e. monitor). Such an instance is useful for GPGPU processing.

Texture: a single or a collection of 2D, 3D or cube images used in graphics APIs to store color information.

Sampler: commonly used to refer to an abstraction in graphics APIs around textures. Every texture can and usually does come with an accompanying sampler to filter the texture upon color retrieval in order to improve appearance. Samplers commonly have hardware units in video cards in order to speed up retrieval, filtering and provisioning of color output.

Material: a collection of samplers used to define the appearance of a surface. These samplers can define different aspects of the material such as roughness, emissivity, specularity amount, albedo color, specular color and other attributes of the material. They can also be procedural (i.e. defined by code). However, for the purposes of this application we are interested in how they distribute reflected light overall.

Pre-Pass (Computer Graphics): using computer rendering algorithms to produce results not yet ready for display. In fact, such results are then fed as input to another rendering algorithm (i.e. another 'pass') to produce final results or imagery.

Instance (Geometry): a piece of geometry (i.e. a collection of 3D triangles) usually with a single material. However, this is not a hard requirement as some game engines may support geometry instances with more than one material.

Centroid (Geometry): the geometric center (or 'mean') of a set of points in three dimensions.

Map: a colloquial term used to describe the environment in which players engage with each other, like a traditional city map.

Thread (code): computer instructions that run simultaneous to other instructions in a computer program.

Thread-safe: interacting with a thread in an unexpected manner results in corruption of its data, potential crashes or unwanted side effects. Operations dubbed thread-safe exist to prevent the aforementioned scenarios.

Temporal Super-sampling (a.k.a. Temporal Anti-Aliasing or TAA): a technique used in rendering to obtain more geometry and lighting information not available at a fixed resolution by perturbing/jittering initial rendering conditions (i.e. the frustum) and accumulating results over time. In its most prevalent form the acquired limited history of results is averaged to provide anti-aliased output.

Selective Super-sampling: a technique used in rendering to obtain more detail by casting more rays only when a certain selective criterion is met.

Boolean flag: a flag that can be either zero or one. The name Boolean refers to George Boole.

Bitwise operations: math operations involving bits (i.e. 'zeroes and ones'). For example a bitwise AND operation checks to see if there are common bits set to one in two different binary numbers, yielding a non-zero number in that case and yielding zero otherwise.

Ray/Object Masks (Ray Tracing APIs): a mechanism to allow rays to ignore or include certain objects based on bitwise AND operations involving binary numbers (dubbed 'masks') assigned to objects and rays alike. If a bitwise AND operation between an object and a ray's mask yields a non-zero number the intersection test occurs. Otherwise the intersection test is ignored.

Simplification (Geometry): Reducing the triangle count on a piece of geometry while roughly preserving its shape. Usually used in computer graphics to reduce rendering time for objects at a distance where reduction in geometric complexity is visually unnoticeable.

Dot Product (Algebra): an algebraic operation involving two vectors (i.e. 'directions'). It is commonly used to get a measure of the angle between the two.

Matrix (Algebra): a two-dimensional table containing numeric values.

Square Matrix (Algebra): a matrix where the number of rows and columns match.

Symmetric Matrix (Algebra): a matrix where the swapping the columns with rows does not yield different cell values.

Low Discrepancy Sequence: a sequence of pseudorandom numbers that provide even coverage of the range from which they're picked from. This is in contrast to high discrepancy sequences which may have too many similar results clumped together over a certain period of evaluations. An example of low discrepancy sequences is the Halton sequence.

Blue noise: in the domain of noise, blue noise is low discrepancy compared to white noise. Therefore, it can be used for random sampling of data when even coverage over any given range is desired.

Orthonormal basis (in 3D space): three vectors in 3D Cartesian space that are all at right angles to each other and with a length of exactly one. A viewer's look vector (pointing towards where the viewer is looking), up vector (pointing out of the viewer's head towards the sky) and side vector (pointing along either the left or right shoulder of the viewer) can construct such a basis.

Principal axes (in 3D space): the three axes determining X, Y and Z directions in 3 dimensions. An orthonormal basis would contain such axes albeit oriented for that basis.

Binary Space Partitioning or BSP: This is an algorithm that recursively splits the world and arranges the splits on a tree-like data structure until certain conditions are met. A BSP-tree is a special type of acceleration structure concerned with splitting the world in half on every node.

BSP Leaves: the bottom-most nodes on a BSP tree. They are at times used for PVS filtering as they can be wide enough to envelope a considerable amount of space.

Flood-fill: a class of algorithms that explore unexplored neighbours until a stop criterion is met. Potentially visible sets can be constructed via a flood-fill approach by recursively exploring open connected areas.

Traceroute (computer networking): a networking utility that measures the latency of every hop (i.e. stop) along a networking path to a certain host.

Re-orthonormalization: a previously orthonormal basis modified to re-satisfy the condition.

1. INTRODUCTION

Cheating in video games is one of the most problematic issues facing video game developers and publishers alike. A survey by Irdeto[1] revealed a staggering 37% of respondents admitting to cheating. It further revealed that 76% of respondents stressed the importance of preventing cheating in online games. The CEO of Irdeto further stressed that not countering these issues will lead to lower engagement and shrinking revenues which is a commonly reported[2] phenomenon. In a webinar hosted by GamesBeat, Riot Games revealed that cheaters roughly cause 33% of affected players to discontinue playing an online game[3].

In this document we provide a solution to prevent one of the most subtle types of cheating: the wallhack. The reason wallhacks are notoriously difficult to detect are their very nature: the player while knowing where enemy combatants are, may intentionally not look their way to deceive spectators observing gameplay. This provides for incredible difficulty in definitively proving whether the observed player was indeed engaging in cheating. Such inquires themselves result in paranoia amongst other players and reduce trust in the fairness of the game itself.

This especially becomes a problem in the world of e-Sports where large prize pools are at stake and fairness is incredibly important. A notable scandal is that of the 2014 banning of professional competitors Hovik Tovmassian (Team Titan), Simon Beck (Team Alternate) and Gordon Giry (Team Epsilon) which made headlines[4]. They were caught and banned mere days before the Dreamhack Winter 2014 CS:GO championship tournament where a prize pool of $250,000 was at stake[4]. It is important to note that this prize pool is by no means the largest of such pools as there are competitions with much larger prize pools such as the $1M at stake for B Site Inc.'s Flashpoint 2 championships[5]. Prize pools for CS:GO alone have totalled more than $100M in a breakdown provided by Valve Corporation[6].

2. BACKGROUND

Many attempts have been made in the past to eliminate or detect not just wall-hacking but other forms of online cheating for first/third person shooter video games. In the section we shall cover notable attempts.

Most anti-cheat software includes technology that runs on client systems. Examples include VAC (Valve Anti-Cheat), PunkBuster™ and BattlEye™. There are two considerable issues with this. Firstly, many consumers feel that these technologies are very invasive as they operate at a privileged level to scan the client machine for signatures of known cheat applications. In this sense they are comparable to intrusive DRM software (used by the music and film industries) or intrusive Anti-virus software. This issue of invasiveness is so bothersome that Blizzard's anti-cheat solution was classified as malware by the EFF[7] at one point. Secondly, cheat applications can indeed be released with new signatures or equipped with sophisticated techniques that place them below the anti-cheat application at operating system level (by prior analysis of the host anti-cheat software) and effectively operate as a root-kit. Detection or removal of such technology at such points becomes either incredibly difficult or virtually impossible. State of the art academic proposals such as BlackMirror[8] released at the ACM SIGSAC this year propose placing anti-cheat software inside Intel's SGX enclave. In fact, Intel's patent[9] also proposes executing anti-cheat code in their SGX secure execution environment embedded in their CPUs which cooperates with an anti-cheat server periodically. SGX has been shown to be vulnerable via multiple vectors[10] and exclusive to Intel CPUs only. nVidia's recent success with running a modern Wolfenstein title on ARM CPUs [11] highlights the necessity for a solution that does not rely on vendor specific guarded execution environments. Additionally, BlackMirror does not address complex rendering pipelines such as Quake II RTX's and simply defers handling it to future work. Valve's patent[12] proposes challenges that can be run inside virtual execution environments crafted specifically to report false negatives in combination with sophisticated heuristics to differentiate real from fake (i.e. 'no-op') challenges. The patent itself emphasises the fact that its proper operation rests on the client machine not having had enough time to develop countermeasures to challenges provided within server-supplied 'black boxes'. Given the development history of cheating software, this is a never ending struggle for anti-cheat (and more generally DRM) software development. SecureBoot Attestation is slowly gaining popularity as a client-side anti-tamper measure. However, its adoption remains largely voluntary and low as many clients with older hardware have found it difficult to procure the TPM hardware required for its operation. More recently, SecureBoot via TPM 2.0 was dealt a decisive blow via the BlackLotus bootkit [13]. Also, without strong session layer security there is no need to circumvent TPM2 in order to wall-hack as demonstrated by hardware wall-hacks[14] that examine traffic externally and place markers on translucent displays. Even with such a measure, CV/AI-based hardware aimbots[15] are possible though this is not a focus of ours. Initiatives like Stadia have claimed to put an end to cheating[16]. Aside from still being vulnerable to CV/AI-based hardware aimbots, their additional input latency[17] introduced by the cloud provider becoming the man in the middle makes them unacceptable for competitive gaming.

Some technology has emerged as solely server-side anti-cheat technology. Notable mentions are that of FairFight™ by GameBlocks, Valve's machine-learning based anti-aimbot solution and Nexon's machine-learning based anti-wallhack solution. FairFight proposes a statistical approach called Algorithmic Analysis of Player Statistics (AAPS)[18] which tracks player behaviour and compares it to a wide variety of data. The issue with this approach is that there can be false positives as they have been reported by the community at large. This further reduces faith in the system amongst players. It is notable that Microsoft[19] and Gamblit Gaming[20]'s patents also use player tracking to achieve a similar goal: flagging players that advance too quickly in a game. Our system by contrast is deterministic and simply prevents wall-hacks from happening in the first place. Valve's machine-learning based application [21] solely targets aim-botting. While they allege that it is achieving high accuracy, this approach unfortunately still provides for community frustration as it does not prevent the act but rather addresses it after the fact. Needless to say, it does not address the issue that we address in this document whatsoever. Nexon's approach [22] is another interesting solution that randomly takes screenshots (pictures) of the player's screen and uploads them to Nexon servers for analysis. While this may work initially, it provides no guarantee that sophisticated root-kit like cheating software won't circumvent the screen-recording mechanism and momentarily turn off wall-hacking when screen-shots are being taken.

Conceptually speaking, the closest solutions that we have seen to date is that of Valorant's[23] initial server side occlusion culling attempt or works derived from it such as CornerCulling[24]. While conceptually similar, our solution is far more accurate with far fewer false negatives. This is made possible by: 1) making the hardware necessary to effectively solve the problem a pre-condition to running the algorithm and 2) making good use of the hardware by using an algorithm built upon extensive ray-tracing research to solve a difficult dynamic visibility determination problem at low resolutions for many viewers simultaneously.

3. SOLUTION DESCRIPTION

Our invention is provided as middleware to server-side developers for integration. It is dubbed SauRay. SauRay requires a server environment that has a real-time ray-tracing-capable GPU or provides real-time ray-tracing in some reasonable capacity. This includes frameworks such as Optix, real-time rendering pipelines that utilize voxels, (signed) distance fields (known as SDFs) or other geometric primitives. Even though our current implementation is crafted around modern GPU-based hardware-accelerated ray-tracing platforms, our algorithm follows the same procedure in the absence of such utilities.

SauRay starts off by launching a headless instance of a graphics API. This instance will not be headless if we are launching SauRay's debugging view. The API then requests feeding of samplers, materials and map geometry instances. It is prudent to cache the range of materials that describe the varieties of player appearances at this stage to avoid feeding them later (if the game supports player appearances with greatly varying materials). This information is then retained

until the next map change. Once fed, our visibility determination system is kicked off on a separate thread which uses our proprietary algorithm. This thread then consumes updated player information every frame: viewing angles, positions and – potentially – animation states (if armature support is present). Using updated player information and previously cached player materials it updates its representation of the players inside SauRay. It may also get updated lighting information per-frame (if the game supports dynamic lighting that casts shadows). Additionally, dynamic changes to environment geometry are retrieved if that is supported as well (such as doors openings, elevators moving, obstructions being add or removed etc.). Our algorithm then proceeds to conduct the innovations outlined below to determine pair-wise visibility between players. Pair-wise visibility information is subsequently fed back to game-server code for packet filtering.

Launching our process in a separate thread is considered our 'threaded' mode and is always one frame behind the server. Part of our algorithm is crafted around dealing with challenges resulting from this (see **The need for forward projection**). A 'non-threaded' mode is also available but it will increase server-frame cost and the increase must be examined to avoid violating server refresh-rate requirements. Receiving map information can be threaded as well as it would allow the game server to perform other tasks simultaneously thus, not increasing the server's load time with that of ours. The most ideal scenario would have both map loading and per frame pre-passes and passes all executing on one long-running thread – with proper synchronization of course – to save on thread creation and destruction overhead. However, we did not go this far for any of our prototypes.

3.1 HIGH LEVEL OBJECTIVE

As mentioned, the objective of the algorithm is to filter outgoing traffic to each player based on computed pair-wise visibility information between players. This computed information is stored into what is called a **visibility matrix**. The visibility matrix is a square matrix representing the visibility of each player against every other player. The diagonal elements of this matrix are of no interest to our application. This matrix is not symmetric as one player seeing another does not guarantee the reverse. Each cell in this matrix is configured to have a set number of Boolean flags representing player visibility in different frames. Our application can track a limited history of visibility which can be as low as one frame. Any bits within a cell being one represents a transmit decision on the server's part. Only a cell of all zeroes prevents a packet transmit decision. Figures 1 and 2 are used to illustrate this matrix.

The reason for maintaining this temporal history is that our algorithm employs techniques for capturing players with sub-pixel footprints in low resolution virtual frusta that at heart have some stochastic element to them (see **Selective Super-Sampling**). When those techniques are not necessary, it still employs a per-ray jitter much like that in **Temporal Super-Sampling** (with the option to choose either **blue** or **white noise**). The ideal use case for this jitter is to capture slightly visible players at medium range through holes when **Force-Casters** are forgotten or not placed at all.

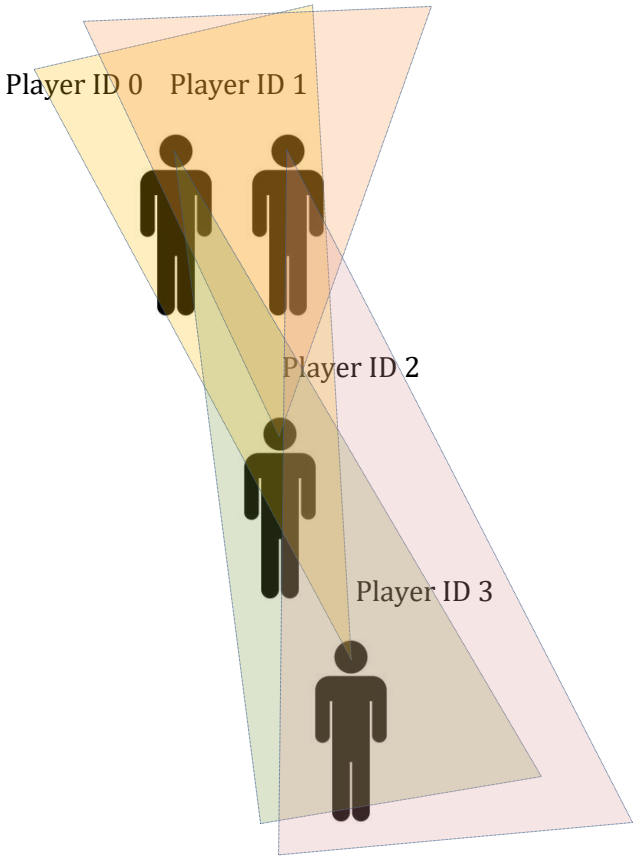


Figure 1.

An example scene involving four players

	Player 0	Player 1	Player 2	Player 3
Player 0	000	000	100	100
Player 1	000	000	100	100
Player 2	100	100	000	100
Player 3	100	100	000	000

Figure 2.

Visibility matrix associated with Figure 2 at frame zero.
Temporal memory is limited to three frames in this setup.

3.2 TOOLKIT FOR LOW RESOLUTION VIRTUAL FRUSTA

Though we are employing graphics hardware to perform visibility determination, our ray budget is not infinite: certainly not large enough to account for the collective native client-side resolutions of all connected players. Therefore, we need to be prudent on how this budget is spent. We have devised three main tools that we use throughout our algorithm to deal with low resolution virtual frusta:

3.2.1 BOUNDS STRETCHING

With insufficient virtual resolutions it is possible for distant players to become invisible as their occupied virtual screen real-estate shrinks. While – as mentioned – our algorithm employs a jitter similar to temporal super-sampling to capture sub-pixel detail, it is important to employ additional measures as well. A pre-pass is employed in our algorithm to determine how far we can stretch player bounding boxes or armatures without exceeding twice their initial volume or intersecting the world at large. In practice one can be flexible with the stretched volume limit though caution is provided to not exceed that amount by much more than two as it can break the algorithm. Information gathered at this stage is used to increase player sizes as the virtual resolution shrinks. This is referred to as **bounds stretching**.

3.2.2 SELECTIVE SUPER-SAMPLING

Another tool used to counter a lowered resolution is as follows: from every pixel we trace a cone starting from the eye with the area of the base being roughly pixel sized. Players with bounding spheres that are enveloped by this cone or overlap this cone are deemed sub-pixel. If a cone encounters a sub-pixel player, it will launch a configurable number of additional rays – jittered differently – to attempt intersecting the sub-pixel player. This is a form of selective super-sampling and is done with the sole purpose of increasing the likelihood of visibility for a sub-pixel player. Our approach to this test is actually simple and more optimized than using exact cones. We simply check if the centroid of the player's collated armature/geometry is sufficiently far enough from the viewer by using a predetermined multiple of its bounding sphere radius shrunk with increasing viewer pixel footprint. If the player is sufficiently far enough, we further measure angle difference via a dot product between the originally traced ray and a ray pointing to the player's geometric centroid. If this difference falls under a certain threshold increased with increasing viewer pixel footprint, the player is deemed sub-pixel and the above mechanism kicks in. In fact, given the above approach, players may not actually be sub-pixel for this mechanism to effectuate. This helps with visibility determination even before players actually become sub-pixel. We provide an option to use low discrepancy sequences such as Halton or blue noise for randomization instead of the default white noise. Figure 3 illustrates these two conditions.

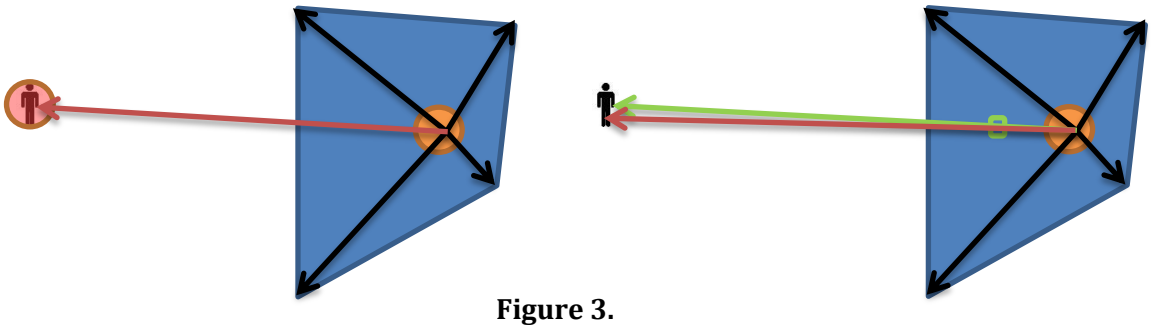


Figure 3.

Our simplified two-step checking process for sub-pixel players

The aforementioned process is not exclusively reserved for rays casted from frusta directly. Secondary ray cones overlapping very distant players can also increase sample count while accounting for intersected surface or luminaire properties especially when the primary ray footprint is large. A detailed examination of propagated footprints is covered in [25].

3.2.3 FORCE-CASTING GEOMETRY

This approach, much like selective super-sampling, is a mechanism to force launching of more rays. A **force-casting** geometry is a piece of geometry that launches additional rays uniformly across its surface area when appearing within a player's frustum. It need not be sufficiently close or even visible to do so. Its sole purpose is to cover visibility for tiny holes that may not be caught using our usual mechanisms. A straightforward application of this concept simply associates a virtual pixel's ID with that of a limited number of fore-casters present for a scene. The code running for that virtual pixel will check to see if the force-caster is within the player's frustum. If so, it will launch as many additional rays necessary to cover its surface area. If the virtual frustum has a sufficiently high resolution, more than one virtual pixel can be associated with a force-caster to further spread the additional trace load even if that means more virtual pixels have to individually check to see if their associated force-casters are in view.

3.3 VISIBILITY REGISTRATION

Our algorithm commences by casting rays from every player's virtual frustum using the player's aspect ratio. This ratio is provided by the server which should theoretically match that of the player's. An under-reported ratio places the player at a disadvantage and an over-reported ratio should be rejected server-side and regarded as misinformation. This mechanism prevents players from abusing the system. In scenarios where this information is not available, a reasonable maximum frustum aspect ratio is used for all players equally (i.e. 16:9). Our algorithm works best when player armatures are available server-side. However, if only their bounding boxes are available, the rays will register player visibility as they intersect player bounding boxes and continue along the ray direction as if these intersections did not occur until encountering a solid

wall. This prevents players' bounding boxes from hiding each other in the event that they line up behind each other visually. To account for limited API guarantees, a map-exclusive ray can be launched followed by another potentially shrunk overlapping ray exclusively for player-geometry registering each encountered player along the way. This is shown in Figure 4. A reason why the server may not replicate player armatures is increasing computational overhead. The registration of player visibility is recorded into the visibility matrix described above.

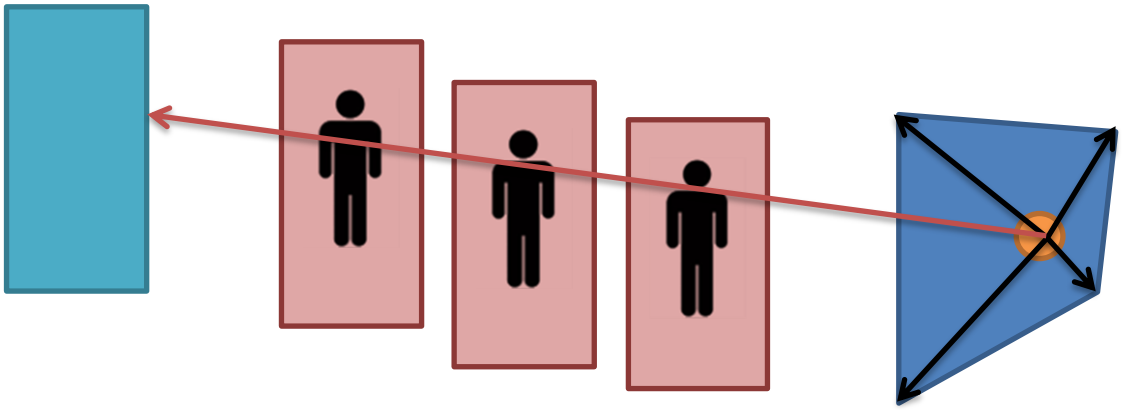


Figure 4.

Player registration for servers providing only bounding boxes

A replication of the above can also account for secondary rays (i.e. shadows) as long as the second overlapping ray is only launched if the map is missed. This ensures that the shadow does indeed belong to a player and not an obscuring building or structure behind.

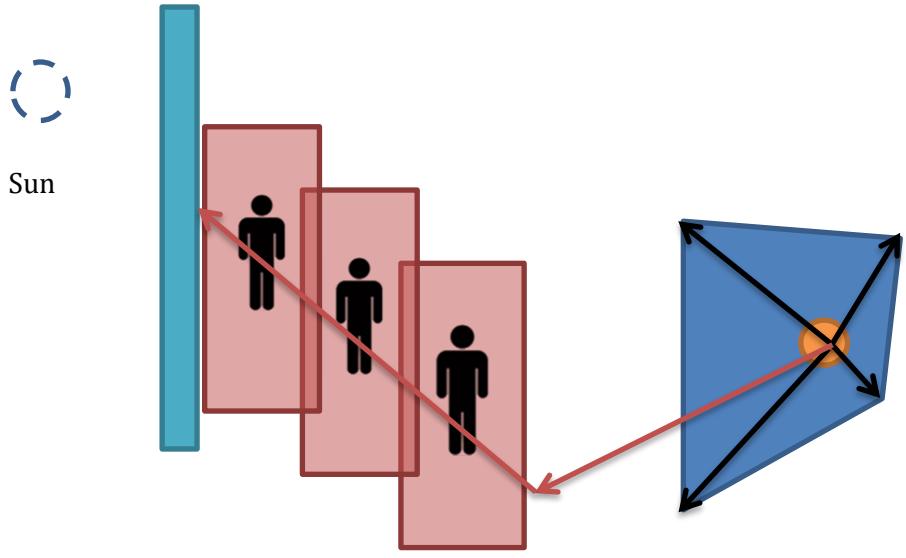


Figure 5.

Player registration for a shadow ray blocked by an obscuring wall

3.4 THE NEED FOR FORWARD PROJECTION

When pre-existing mechanisms for packet filtering – i.e. BSP visibility leaves – are replaced with something far less conservative – i.e. ray-tracing – new challenges are introduced that need to be addressed. In this section we examine these challenges and how they are resolved in SauRay.

3.4.1 HIGH-SPEED SCENARIOS

When traditional means like BSP visibility leaves are used to filter packets, player information is exchanged between players within the same visibility leaf which can be quite large mostly obviating the need for any extra care related to velocity. However, when information transmission is only reduced to instances where one player can see another – if that duration is brief – this can be cause challenges. Imagine a scenario where a player's velocity is extremely high and within one server frame, they are going to be past a gap in a wall. In that circumstance, the player's information should be transmitted briefly regardless. This scenario is illustrated in Figure 6.

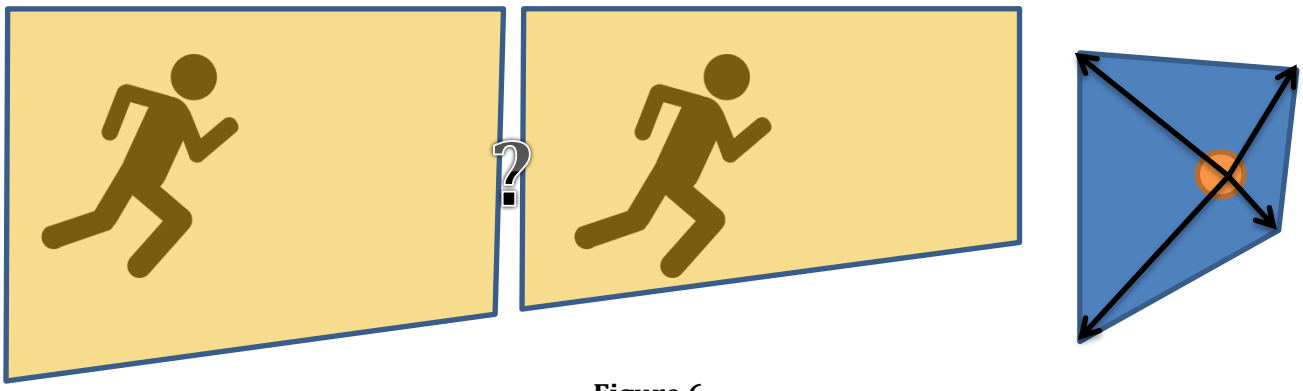


Figure 6.

A very fast player dashing past a gap will be missed intra server-frame if non-conservative pairwise visibility determination is used naïvely

The same issue can happen if a viewer is moving too fast past a player hiding in a gap in the wall. Figure 7 illustrates this:

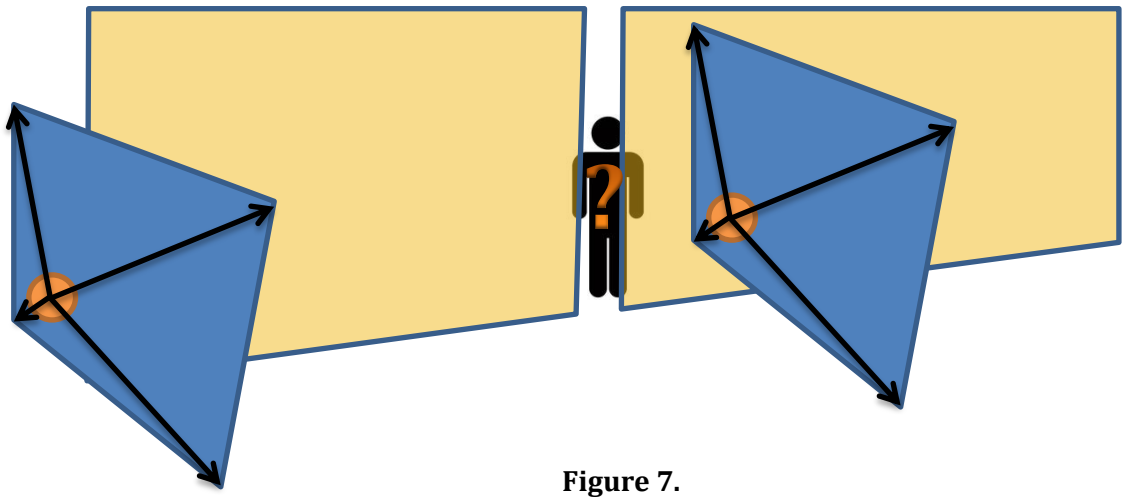


Figure 7.

A very fast viewer dashing past a gap will be miss a hiding player intra server-frame if non-conservative pairwise visibility determination is used naïvely

This issue can be exacerbated as the server refresh-rate goes down. It can be especially problematic in games like Quake II where the server refresh-rate is as low as 10 ticks per second.

3.4.2 THREADED-MODE

As previously mentioned, one may run our middleware in 'threaded' mode in order to save on performance. In this mode SauRay does visibility determination for the previous frame in parallel as the server evolves the world to arrive at the current frame. This causes SauRay to always be a single frame behind the server. This can be problematic especially when the tick-rate is low (i.e. Quake II as discussed above) as illustrated in Figures 8 and 9.

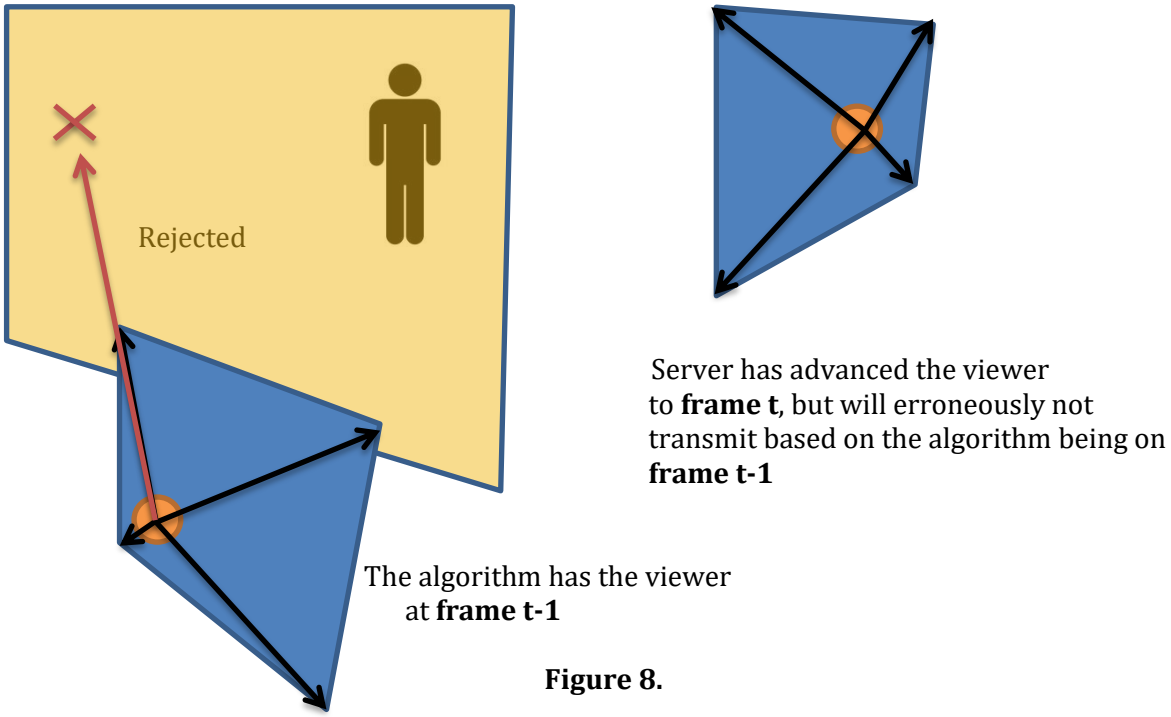


Figure 8.

A naïve application of threaded visibility determination can cause popping for viewers

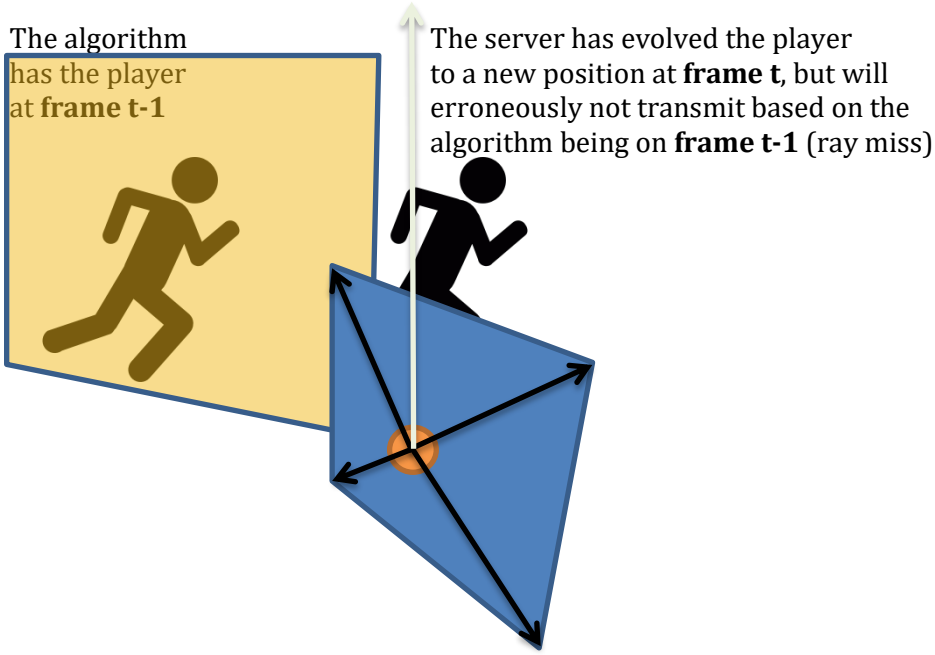


Figure 9.

A naïve application of threaded visibility determination can cause popping of players

3.4.3 FORWARD PROJECTION AS A SOLUTION

To combat the aforementioned scenarios, we employ a technique that forward interpolates frustum data with a future frustum provided alongside the initial frustum. This future frustum is essentially built using the current frustum plus a small amount of time in the future given the viewer's linear and rotational velocities. Our API explicitly requests both frusta along with the player ID when the developer is given the opportunity to supply vantage point data. Rays traced for the viewer are randomly casted from in between frusta made via linearly interpolating between the supplied initial and future frusta using a uniform random variable in the range 0.0 to 1.0 inclusive. These in between frusta are re-orthonormalized after interpolation. The user again has the choice to either use blue or white noise for the random variable. This way we have a high probability of catching an enemy combatant that is not currently visible but will be in the next frame. A diagram is provided in Figure 10 to present this concept.

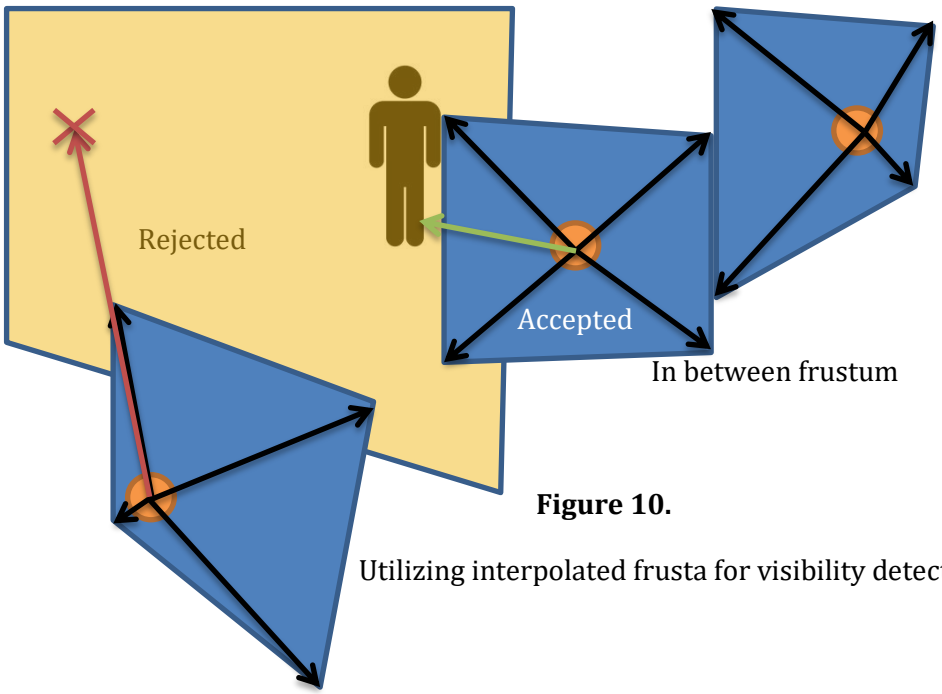


Figure 10.

Utilizing interpolated frusta for visibility detection ahead of time.

In that same sense, when the developer is supplying player armature/geometry data to SauRay they should take care to overlap several versions of the armature/geometry using animation data looking into the next frame. The armature data supplied to our API is essentially constructed low polygon count pose geometry that represents the armature. As previously mentioned this may not always be available and may simply be replaced with bounding boxes. With this approach combatants that are not currently visible by a viewer but will be in the next frame, will have their data transmitted ahead of time to avoid popping on the viewer's screen. Compared to Valorant's bounding box solution described in [23], our solution is far more accurate with far less room for both false positives and negatives. Figure 11 aims to visualize this as well.

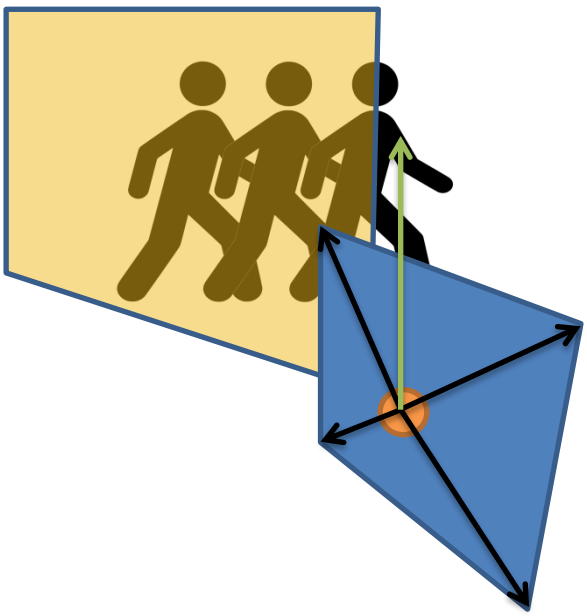


Figure 11.

Ahead of time visibility detection via overlapping armature geometry

In all of our implementations we clip the velocity vector for forward projection against the map using a single additional ray cast per player to ensure that final frusta do not end up inside the map, thus denying a side-channel attack whereby players can leak information by slamming themselves into walls. Also note that this mechanism does not replace lag compensation[26] but is rather additive to such a mechanism (i.e. forward projection should happen on top of lag compensated player information if the server has this implemented).

3.4.4 HANDLING ULTRA-HIGH-LATENCIES VIA SPECIALIZED LOOK-A-HEADS

Once round-trip time exceeds a 100 milliseconds it becomes apparent that further measures are required. The reason for this is distinction is what we call the issue of intent. Envision a scenario where a player with a ping of a 100 milliseconds is behind a pillar facing an enemy combatant. Assuming symmetric bandwidth, it would take 50 milliseconds for the player's movement intent – i.e. to strafe either left or right – to arrive at the server. In other words, we cannot predict the intent of the lagging player fast enough in order to forward projection their actions into the future for ahead-of-time transmission purposes.

For such a scenario we introduce specialized look-a-heads. Specialized look-a-heads provide the ability to cover very high latencies (as high as 250ms round-trip) by leaking a bit more data in order to prevent popping. While this technique reduces our technique's effectiveness it still maintains an edge over PVSs by ensuring that players sufficiently obscured behind walls are still not transmitted. Such distinctions are not made via PVS approaches (whether utilizing flood-fills or BSP visibility leaves). To ensure that players are not reporting fake latencies, we can perform a traceroute operation on their originating IPs. If their second-to-last hop latency is wildly different

from their latency, they are banned for attempting to exploit the algorithm through traffic rate manipulation. Since measures like SauRay are practically intended to prevent cheating in high-stakes low-latency competitive matches this mechanism is not to be overly relied upon other than in casual settings. They simply act as a reasonable fall-back in the event that high latencies are acceptable (such as using a SauRay enabled server to temporarily host a causal match). Competitive-only servers such as those of Valorant that enforce strict player latencies need not rely on this section of the algorithm.

The actual implementation of a look-a-head involves what we call sub-frusta. We effectively divide a frustum into 9 pieces and launch rays from each piece as if it was a frustum all by itself. This necessarily shrinks the resolution of each virtual sub-frustum and necessitates the mechanisms described in the previous section for low resolutions (i.e. stretch-bounds and selective super-sampling). In fact, the bounds-stretched player geometry can be placed in a separate BLAS, grouped into a separate TLAS and only served to the lagging player. This denies players with low latency an enlarged version of enemy geometry. In the event that enough bits are left over for geometry masks, separate stretched BLASes/TLASes become unnecessary and bound-stretched geometry can share half of the bitmasks for geometry masks. This was possible for our implementation for CS:GO and the resulting geometry masks were solidified into what is demonstrated in Figure 12:

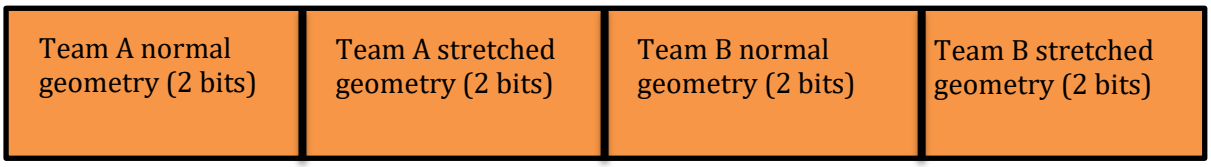


Figure 12.

Our final geometry masks for CS:GO with look-a-heads implemented

However, this may not always be possible as these bits maybe used for selectively presenting certain obscuring geometry to different players (i.e. players that can 'see' through certain portals). In that event, the overall solution will still be efficient as the map's BLASes will be shared amongst the two TLASes saving memory and compute.

The sub-frusta mentioned previously are composed of one that maps to the original frusta and another 8 placed at corners surrounding the player's interpolated eye. The interpolated look direction is maintained in all cases. The corners are visualized below in Figure 13 and are aligned to world principal axis:

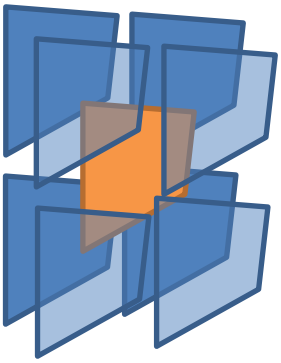


Figure 13.

Approximate arrangement of sub-frusta with the main one in orange

Please note that look-a-heads arranged on 8 corners are not in and of themselves novel and can be found in prior art such as CornerCulling[24]. However, our approach maintains all the benefits of a full frustum – including forward projection and shadow rays –by simply subdividing the appropriated space and re-running the same algorithm. This is very unique to our approach. Figure 14 is a screen-shot of sub-frusta in action in CS:GO (debug view):

Figure 14.

Sub-Frusta in action: note the player's split view of the player above the stairs

The look-a-head corners are clipped during the pre-pass phase described above to ensure that they do not end up inside walls. Every look-a-head corner also conducts line-of-sight tests to every other enemy player in the game. This is necessary as look-a-head corners alone will not be able to prevent popping caused by rapid head swings. They can only account for popping resulting from lateral movement. This is the only location in the algorithm that we conduct line-of-sight tests. Line-of-sight tests occur on three locations along player velocity vectors to account for forward projection of enemy positions. Much like force-casters each virtual pixel within a sub-frustum can either take on one or more line-of-sight tests dependent on sub-frusta resolutions. For example, in our CSGO implementation with a base resolution of 640x640, each sub-frustum will approximately

have a resolution of 213x213. Given that the upper bound on player count is 64, the first 64 virtual pixel rows can naturally be associated with line-of-sight tests. Since each virtual row has also 213 virtual pixels – well above the three suggested above – the first three virtual pixels can trace along the combatant's velocity vector. This is our implemented approach for evenly distributing line-of-sight tests across virtual pixels.

3.5 Path-Tracing for Future Pipelines

While the simplest use case of our API involves few (usually no more than three) bounces per virtual pixel per player (accounting for reflections or AO combined with direct shadows), we can provide standard uni-directional path-tracing code that allows further bounces of light based on hit material properties making our technique relevant for path traced online multiplayer games in the future. Our procedure for visibility flag determination remains the same for further bounces effectively. This makes our solution very adaptable for rendering technology that may become fairly commonplace in the future. One interesting aspect of our path-tracing approach, adapted to our solution, is that diffuse secondary rays (as in diffuse rays cast after the first ray) need not be arbitrarily long. They can be as short as two to three times the height of the average player in order to capture relevant combatants in the environment. This again provides for natural optimization that lends itself well to our approach. In the event that player shadows for nearby light sources are visible to clients, NEE secondary rays can be launched for player registration as well. Light sources can be picked at random (again using either high or low discrepancy sequences) as well for distributing the cost across samples and frames much like the explicit many-light sampling approaches found in computer graphics literature. However, all of this is done in service to player visibility detection and not rendering visuals. As a result, small dim lights too distant to cast player shadows or leave noticeable imprints can be safely ignored. Effects such as smoke, which cause loss of perceived light through scattering, can be implemented in our algorithm via passing an additional strength value on the ray payload. This value can be diminished as the ray travels through absorbing material with per-vertex values describing the absorption factor. When that value falls below a certain threshold it can terminate the ray. Alternatively, one could implement stochastic termination of the ray based on the vertex absorption factor as well (i.e. 'Russian-roulette' style). In practice, when we had to account for smoke in our CS:GO integration our solution turned out to be far simpler (see 'dynamic geometry' below).

Current generation multiplayer games such as CS:GO or regular Quake2 will usually need 1 or 2 rays per virtual pixel as they do not account for complex light interactions on screen. Modified multiplayer games such as Quake2 with RTX do provide us with a window into what could be commonplace looking forward. However Quake 2 RTX's multiplayer component is not popular at the moment even though our approach can fully support it.

A diagram is provided in Figure 15 (with only two path segments) to demonstrate our approach to repurposing path-tracing for player visibility determination.

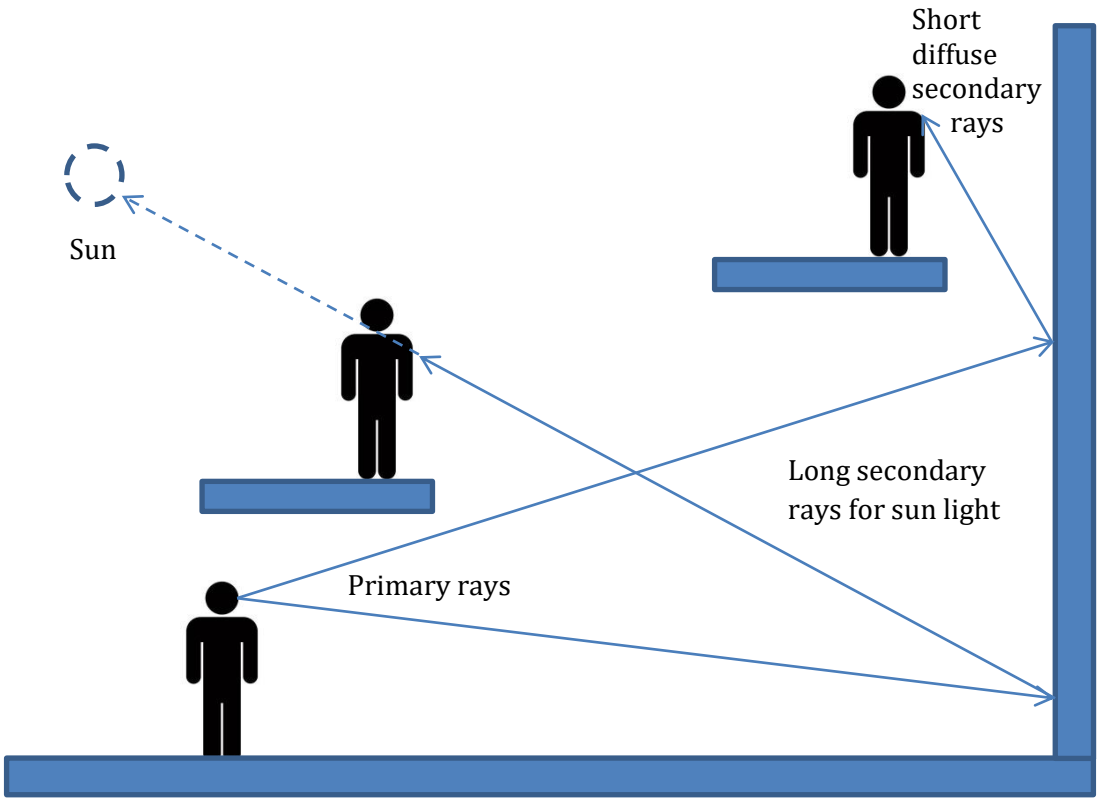


Figure 15.

Using path tracing solely for visibility determination in our algorithm

4. OTHER CONSIDERATIONS

This algorithm by nature places constraints on player information availability. Thus it is important to account for that in code related to gameplay and beyond. In this section we explore such areas of importance.

4.1 THIN-CLIENT LEANING

Our algorithm desires network code design that authorizes the server to do most of the game world's processing. This is not to say that it is not applicable to thick-client models. For thick-client network code, additional checks must be placed in order to avoid exploits such as spamming different frusta across the environment to break the algorithm. Such measures could also avoid cheats such as invulnerability, speed-hacking, phantom bullets and false deaths.

4.2 Audio cues

As enemy player information is no longer consistently available, it is important to have a strategy for emitting sounds related to unseen players. Either of three strategies is possible:

- Randomizing the origin of sounds emitted from players. The amount of randomization can increase as the origin itself becomes more distant from the listener. This would maintain correct sound spatialization while decreasing information leakage with distance. We currently employ this for our third party integrations.
- Sending only stereo channel volumes along with a sound ID. While not stereo-correct this could be a bandwidth efficient way of providing audio cues without unveiling audio origin.
- Live audio streaming. This is an interesting strategy as it provides complete origin anonymity while technically being able to emulate spatialization server-side. Games like Call of Duty: Warzone use it albeit not for avoiding player information leakage but as a means of purposefully leaking nearby combatant radio chatter. It can be repurposed for origin obfuscation as well.

4.3 Radar and screen-based pins

When transmitting such information, one should be careful to only transmit two-dimensional information and withhold as much data as possible without breaking functionality. Pins that project a 3D point onto the screen can simply do this server-side and only transmit the 2D screen location when it is in view for example.

4.4 Non-visible physical cues

These should be handled via transmitted server-side signals. An example of this is a player walking backwards into another player who is not in view. The server should check for this condition server-side and transmit a stop signal to the back-stepping player. We have seen this in action in CS:GO which is quite server-authoritative.

4.5 Impact on lag-switching

Lag switching is slightly stunted under this solution compared to other filtering mechanisms (i.e. BSP visibility leaves). This is due to there being far fewer opportunities for player data to be needlessly available. Thus lag switchers will have far fewer advantageous scenarios where withholding outgoing updates will allow them to advance the game state in their favor.

4.6 Trust of user-supplied frustum information

As hinted in **section 4.1**, careful scrutiny of user-supplied data is important. User-supplied frustum specifications such as vertical field-of-view angles and aspect-ratios are no exception. Fortunately, there are no real avenues of abuse in this regard: if a client under-reports these values, they are placed at a disadvantage. Values supplied beyond a reasonable range can be rejected server-side. In our integrations, such values were not available server-side and thus we assumed reasonable maximums of 90 degrees and 16:9 aspect ratios for all y-FOV angles and aspect ratios respectively.

4.7 Using simplified geometry

Using simplified geometry when available is absolutely desirable. However, it should be noted that said geometry should fit in the geometry it represents in order to avoid false negatives during gameplay (i.e. players being falsely invisible). In other words, it should not be a conservative 'wrapper' of its associated geometry. Primitives reserved for client-side occlusion culling could be great candidates for this purpose.

4.8 Player-associated geometry (i.e. vehicles, projectiles)

Such geometry should be traced and have their outgoing packets filtered like any player. The only difference is that these entities will not be coupled with any ray-tracing frusta.

4.9 Dynamic geometry

This is one of the most interesting points of care in our application. Our philosophy with respect to dynamic geometry is to err on the side of more visibility (i.e. false positives). For example, if a door is signalled to be opened on the server, any obstruction representing it in our acceleration structure(s) should be removed before the door begins to open. If it is signalled to be closed, the server should wait until the door is completely closed and only then re-introduce the obstruction representing the door in the acceleration structure(s). In CS:GO we represent smoke plumes as opaque occluders. In game world, they are indeed fully opaque for a period of 15 seconds before they slowly become fully transparent (this takes 3 seconds). We simply remove our smoke representation at the 15 second mark which both forbids a great deal of information leakage while allowing the algorithm to maintain its simplicity and avoid any additional overhead associated with computing scattering or performing Russian-roulette. This also keeps it in line with our philosophy outlined above as packet transmissions during those 3 seconds occur as if the plumes were not there, avoiding any sensation of popping client-side.

The outlined philosophy becomes important when considering large moving vehicles or debris capable of obstructing player vision. In order to correctly allow such occluders to obstruct

players without popping, their current and future models can be quickly and roughly sphere packed. Spheres that are common to both models can then be isolated and transformed into obstructive geometry placed on our acceleration structure(s).

5. FLOWCHART

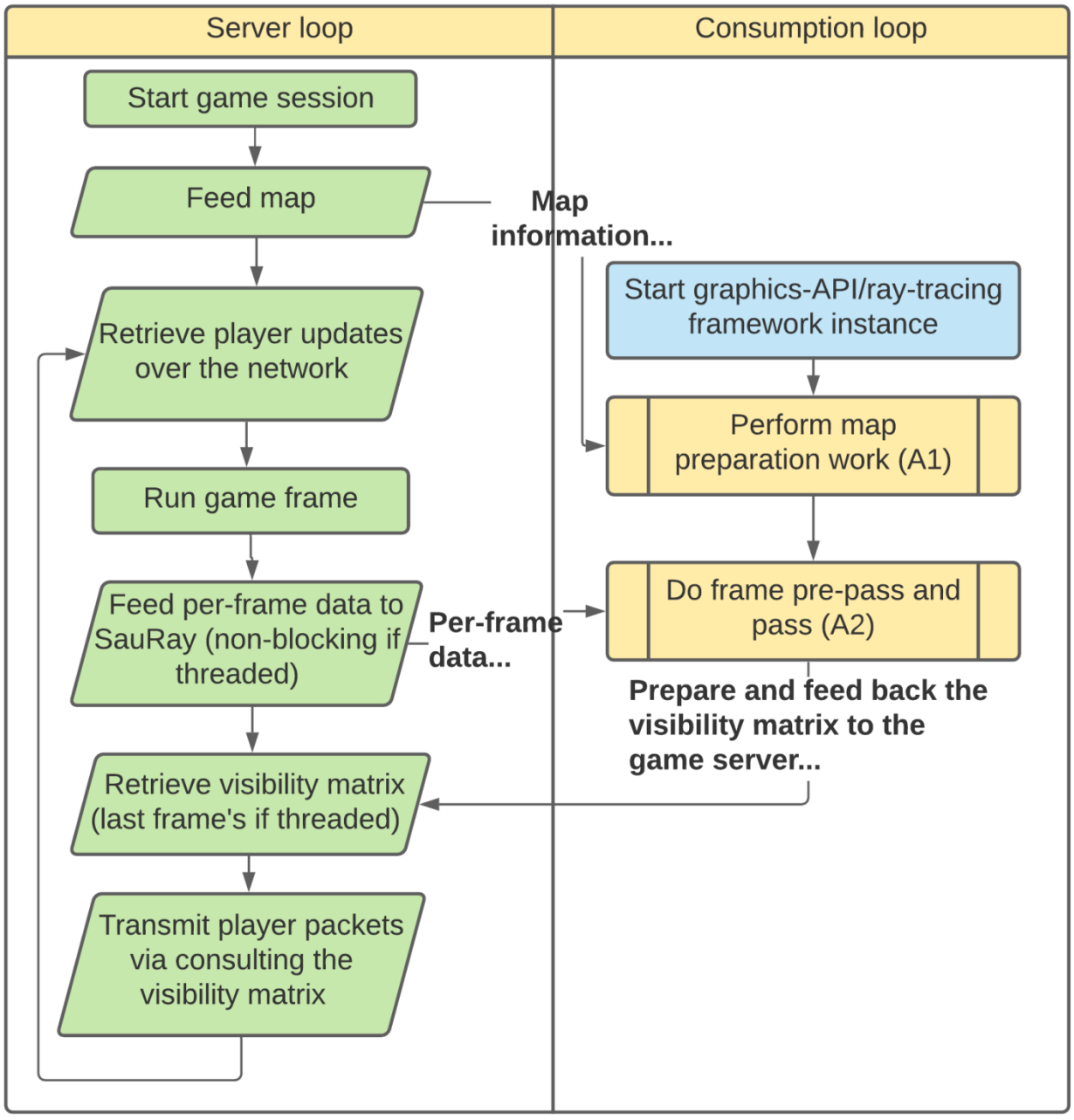
The following is our flow chart of the algorithm. The simple route is crafted to handle current generation games with few visibility requirements (i.e. CS:GO or Fortnite RTX) and the path-tracing route is intended for games with more complex visibility requirements (i.e. Q2 RTX). There are similarities and differences in both approaches worth mentioning.

The simple (AABB) approach requires that rays penetrate player bounding boxes and continue. The path-tracing approach does not and indeed bounces rays against players as if they were no different from the surrounding environment (i.e. using surface material properties). This difference in approach is due to the fact that player AABBs – given their conservative dimensions – can occlude other player AABBs whether in a player's frustum or when casting shadows. This is in contrast to player geometries resembling armatures which can be fitted enough to prevent such obstructions and thus idea for path-tracing scenarios. This necessitates encoding player IDs on ray payloads until the entire transport path reaches a light source in the path-tracing case.

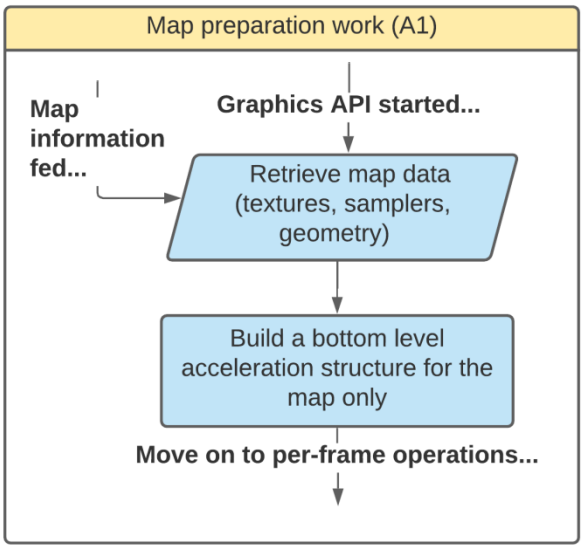
All rays other than shadow rays (i.e. primary, reflections, global AO etc.) in the simple algorithm are un-interested in light sources. This is due to the fact that games such as Fortnite RTX or Rise of the Tomb Raider (which features global AO via VXAO) may always have a background amount of ambient light, enabling player visibility even in low-light conditions. Thus, reaching a light source is not necessary for such rays. Conversely, the path-traced route – whether NEE or material scattered rays are concerned – cannot ignore light sources as both independent and dependent visibility is achieved due to interactions (or lack thereof) with them.

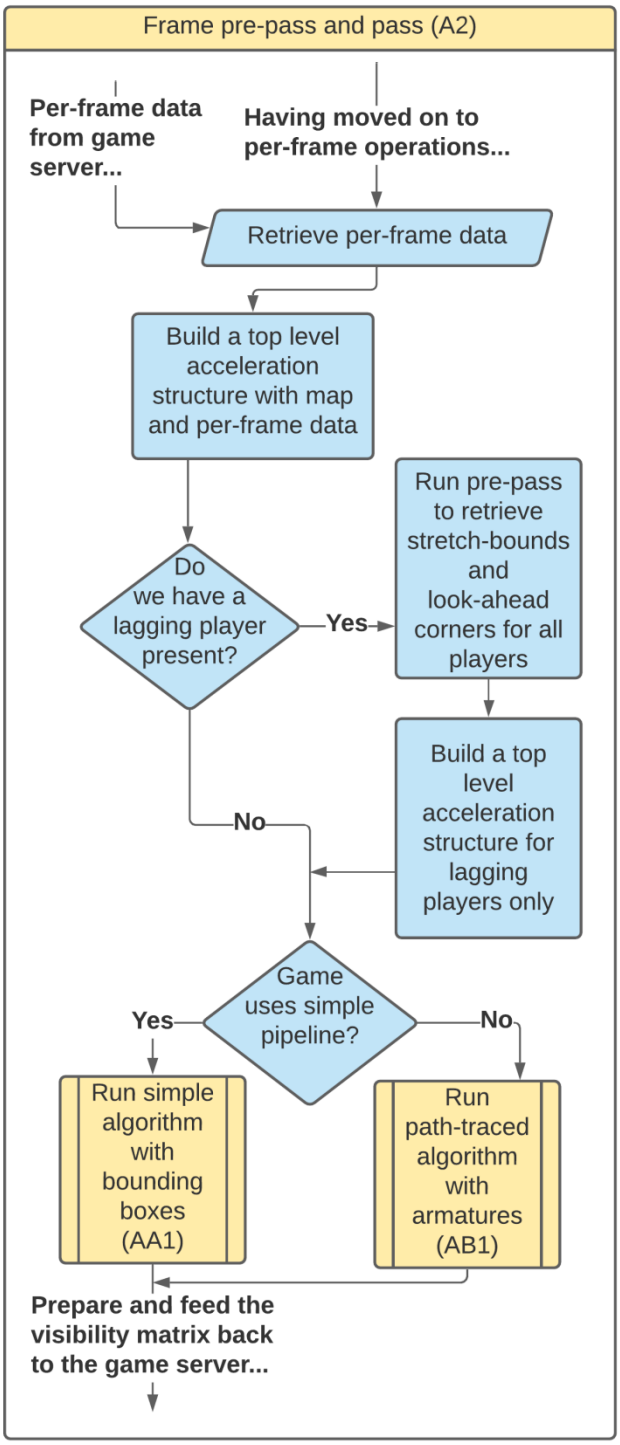
AO or reflective mirror rays in the simple case or NEE rays in the path-traced case utilize their own custom logic. This necessitates additional entries in the SBT when using modern hardware-accelerated ray-tracing APIs; thus, reducing additional burden on ray payloads and preventing code from cluttering into a single SBT entry. The path-tracing route does not keep track of hit-types precisely because of such a choice. When conducting alternative forms of tracing (i.e. against SDFs, voxels, etc.) this also implies custom ray logic in line with this philosophy. When tracing against SDFs for example, the AO or NEE query may only have had to come close to a present player along its path to initiate a player registration. The minimum necessity is a change in visible lighting according to the client-side rendering pipeline.

In the event where the game falls somewhere in between (such as Quake II RTX where client-side rendering is path-traced and the server provides AABBs), it is worthwhile to create armatures server-side in the interest of correct visibility determination.


Server loop
Consumption loop
Start game session
Feed map
Map information...
Start graphics-API/ray-tracing framework instance
Retrieve player updates over the network
Perform map preparation work (A1)
Run game frame
Feed per-frame data to SauRay (non-blocking if threaded)
Per-frame data...
Do frame pre-pass and pass (A2)
Prepare and feed back the visibility matrix to the game server...
Retrieve visibility matrix (last frame's if threaded)
Transmit player packets via consulting the visibility matrix

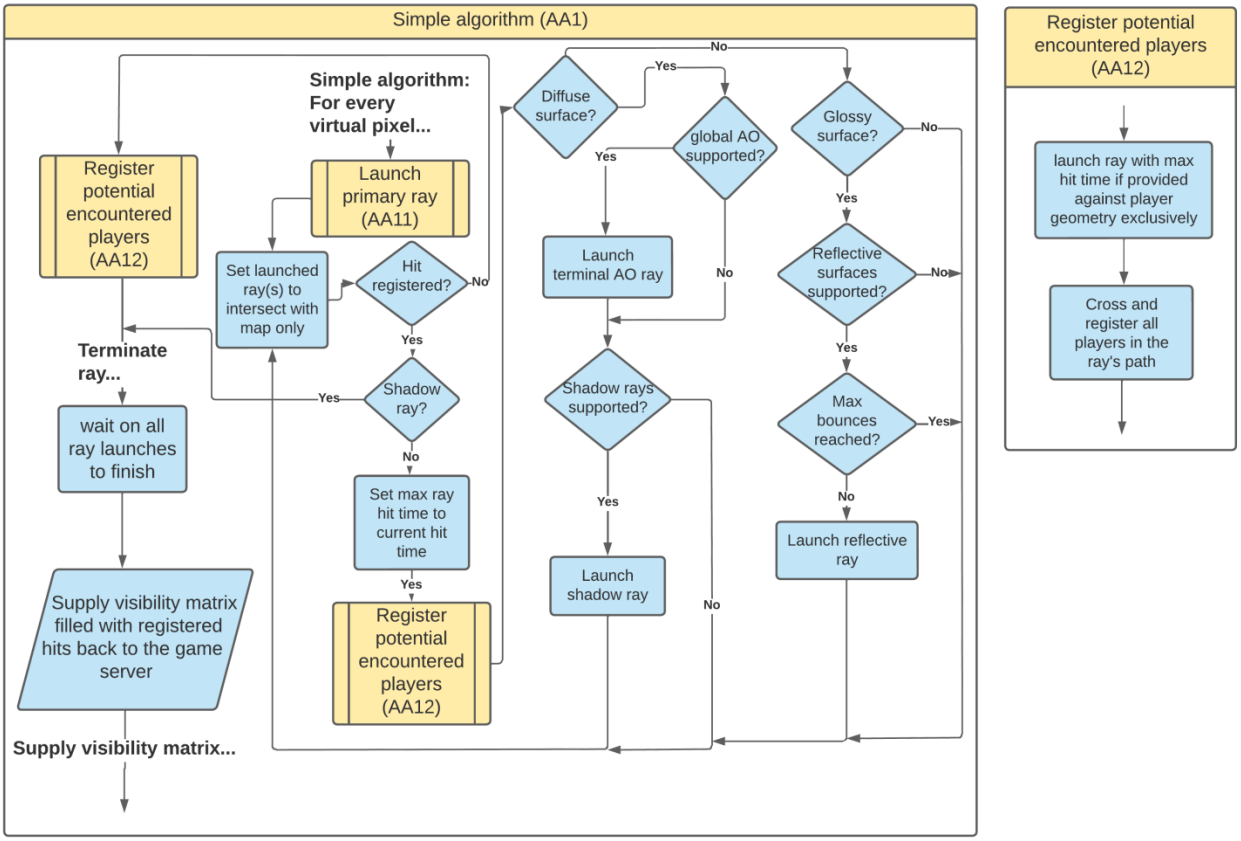
5.1. SUBROUTINES



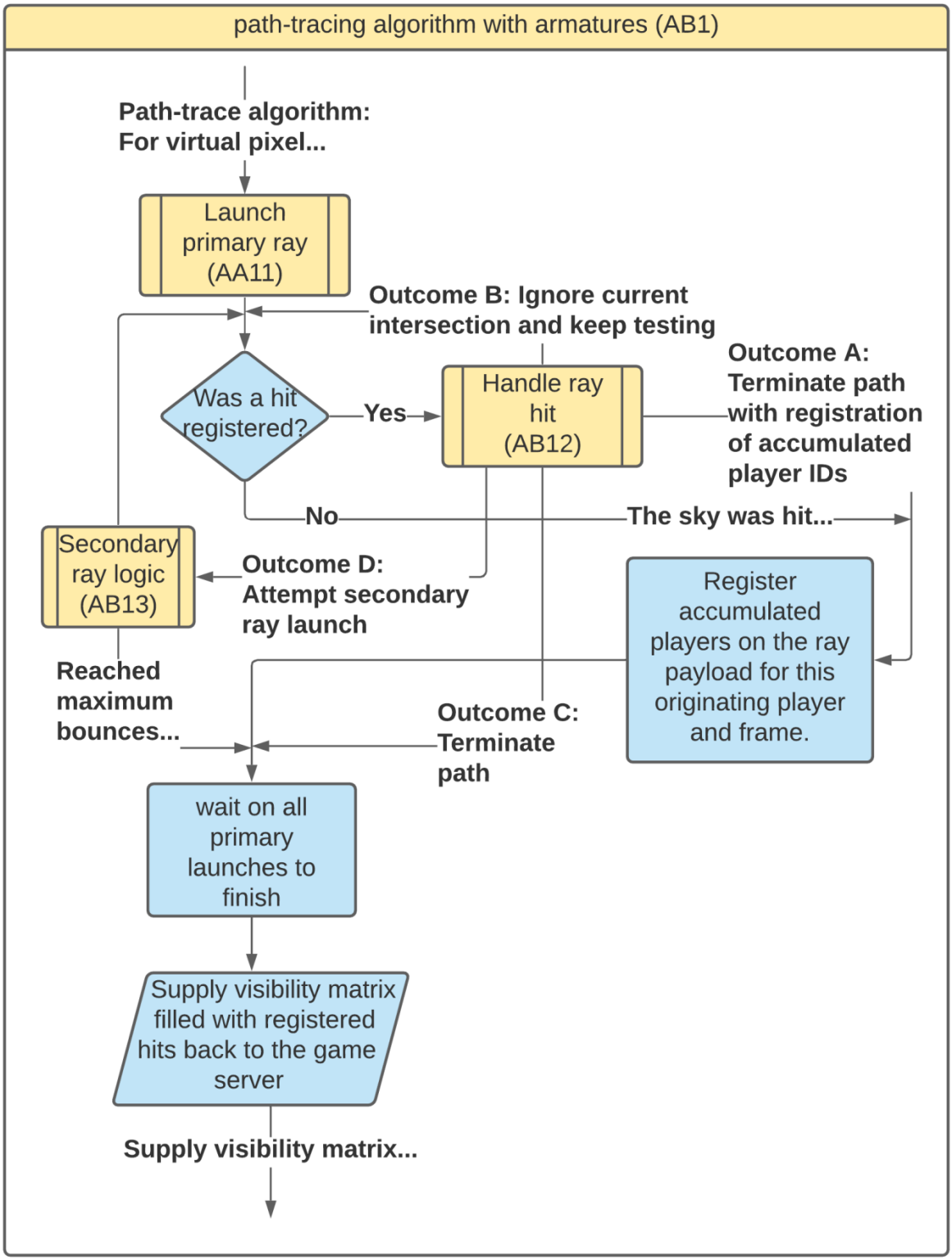


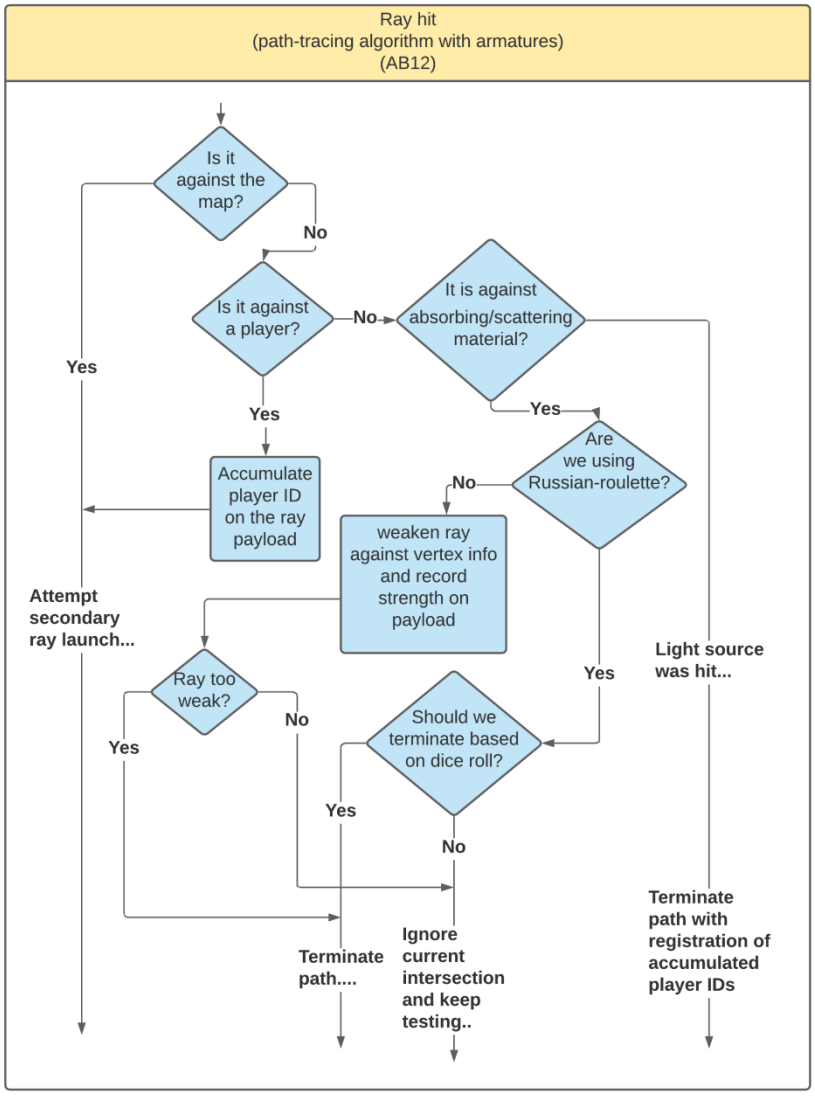
5.1.1. SIMPLE ALGORITHM

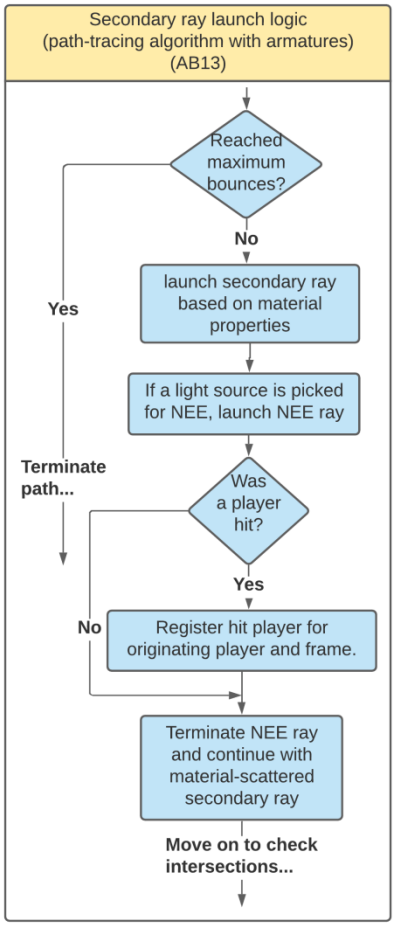
Please note that subroutine AA11 is common with the next section and will be provided further below.



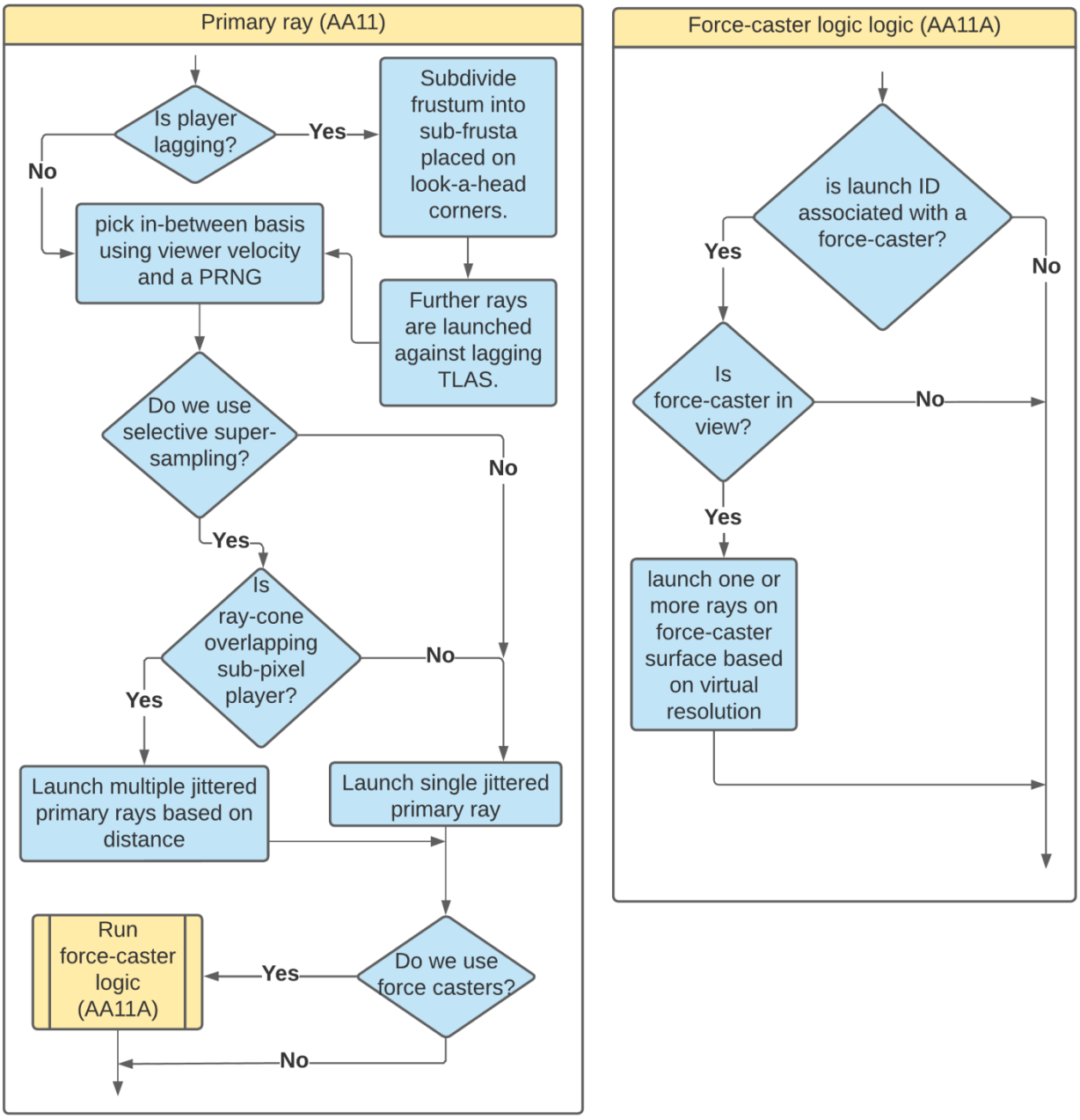
5.1.2. PATH-TRACING ALGORITHM WITH ARMATURES




Ray hit
(path-tracing algorithm with armatures)
(AB12)
Is it against the map?
No
Is it against a player?
No
It is against absorbing/scattering material?
Yes
Yes
Yes
Accumulate player ID on the ray payload
Are we using Russian-roulette?
No
weaken ray against vertex info and record strength on payload
Attempt secondary ray launch...
Ray too weak?
Yes
No
Yes
Should we terminate based on dice roll?
Yes
No
Light source was hit...
Terminate path....
Ignore current intersection and keep testing..
Terminate path with registration of accumulated player IDs


Secondary ray launch logic
(path-tracing algorithm with armatures)
(AB13)
Reached maximum bounces?
Yes
No
launch secondary ray based on material properties
If a light source is picked for NEE, launch NEE ray
Terminate path...
Was a player hit?
Yes
No
Register hit player for originating player and frame.
Terminate NEE ray and continue with material-scattered secondary ray
Move on to check intersections...

5.1.3. COMMON SUB-ROUTINES



6. REFERENCES

[1] https://www.forbes.com/sites/nelsongranados/2018/04/30/report-cheating-is-becoming-a-big-problem-in-online-gaming/?sh=f613d3676637

[2] https://www.theverge.com/2020/5/6/21246229/pc-gaming-cheating-aimbots-wallhacks-hacking-tools-developer-response-problem

[3] https://www.brighttalk.com/webcast/12339/489903?utm_source=vb&utm_medium=marketing-post&utm_content=-jun-17&utm_campaign=june-29-denuvo-webinar

[4] https://www.ign.com/articles/2014/11/25/csgo-esports-community-shaken-following-revelation-of-cheating

[5] https://www.prnewswire.com/news-releases/1-million-on-the-line-for-flashpoint-2-and-the-largest-csgo-prize-pool-of-2020-301158145.html

[6] https://www.esportznetwork.com/csgo-esports-reaches-100-million-milestone-for-prizes/

[7] http://news.bbc.co.uk/2/hi/technology/4385050.stm

[8] https://lifeasageek.github.io/papers/seonghyun-blackmirror.pdf

[9] https://www.freepatentsonline.com/y2020/0206635.html

[10] https://en.wikipedia.org/wiki/Software_Guard_Extensions#Attacks

[11] https://blogs.nvidia.com/blog/2021/07/19/geforce-rtx-arm-gdc/

[12] https://patents.google.com/patent/US20060247038A1/en

[13] https://www.welivesecurity.com/2023/03/01/blacklotus-uefi-bootkit-myth-confirmed/

[14] https://www.youtube.com/watch?v=AIbkt6Rl8FA

[15] https://arstechnica.com/gaming/2021/07/cheat-maker-brags-of-computer-vision-auto-aim-that-works-on-any-game/

[16] https://www.pcgamesn.com/stadia/google-stadia-cheating-aimbot

[17] https://piunikaweb.com/2022/06/20/nvidia-geforce-now-input-lag-issue-persists-but-there-is-no-fix-in-sight/

[18] https://www.i3d.net/products/hosting/anti-cheat-software/

[19] https://patents.google.com/patent/US20180182208A1/en

[20] https://patents.google.com/patent/US20150194016A1/en

[21] https://www.youtube.com/watch?v=kTiP0zKF9bc

[22] https://www.gdcvault.com/play/1026331/ML-Tutorial-Day-Beating-Wallhacks

[23] https://technology.riotgames.com/news/demolishing-wallhacks-valorants-fog-war

[24] https://github.com/87andrewh/CornerCullingSourceEngine

[25] https://cseweb.ucsd.edu/~ravir/belcour.pdf

[26] https://developer.valvesoftware.com/wiki/Lag_Compensation