УДК 004.021

ADVANCED BULLET COLLISION REACTION SYSTEM FOR VIDEO GAMES

DOI 10.36994/2788-5518-2021-01-01-13[13]

***Dmitriy Markov**, NTUU "Igor Sikorsky KPI", Kiev, Ukraine, markov.dmitriy.dev@gmail.com*

***Waleriy Rogoza**, Dr.habil.Prof. NTUU "Igor Sikorsky KPI", Kiev, Ukraine, rosvetnik@gmail.com*

НТУУ «КПИ имени Игоря Сикорского», г. Киев, Украи

***Abstract**. The article describes different bullet collision detection and collision reaction systems and proposes an advanced bullet collision reaction system that allows game developers to achieve robust and believable reactions with a minimal performance impact. The propos НТУУ «КПИ имени Игоря Сикорского», г. Киев, Украина ed system shows rules for ricochet, penetration and jam mechanics. Also, complex penetration logic was introduced in the given system, allowing gamers to use the en НТУУ «КПИ имени Игоря Сикорского», г. Киев, Украина vironment to achieve the best results.*

***Keywords**: video game, bullets, algorithm, collision reaction, penetration, ricochet, shooter*

ПОКРАЩЕНА СИСТЕМА РЕАКЦІЇ ПУЛЬ НА КОЛІЗІЮ ДЛЯ ВІДЕО ІГОР

***Марков Д. К.**, НТУУ "КПІ імені Ігоря Сікорського", м. Київ, Україна, markov.dmitriy.dev@gmail.com*

***Рогоза В. С.**,. Профессор Западнопоморского Технического Университета, Щецин, Польша, rosvetnik@gmail.com*

***Анотація**. У статті описуютьсяпристрої систем пострілу, знаходження колізії і реакції на колізію в відео іграх. Описуються такі підходи як рейкастінг або хітскан який заснований на зборі інформації про колліжене променя, посланого з зброї у напрямку пострілу. Розбираються плюси і мінуси такого підходу. У статті також розглядається балістичний підхід, в якому формується фізичний об'єкт і випускається в ігровий світ. На основі цих підходів був розроблений гібридний підхід, який полягає у використанні балістичної логіки заснованої на послідовних рейкастах.*

На основі наведених технологій розроблена система реакцій куль на колізію. Технічно вона отримує всі об'єкти на шляху проходження кулі на даному кадрі. Далі на основі цих даних система розбиває всі ділянки прольоту за матеріалами і обробляє їх окремо один від одного.

Описана вдосконалена система реакції на зіткнення кулі може забезпечити набагато більш глибоке занурення в будь-який шутер. З точки зору

[13] Dmitriy Markov, Walery Rogoza

продуктивності, запит інтервалів променів заснований на універсальному запиті променів, тому єдина різниця полягає в постобробці результатів, що означає, що вплив на продуктивність просунутої системи мінімально. Беручи до уваги, що пропонована система також заснована на модифікованому гібридному балістичному підході, вона може використовуватися як є в різних типах відеоігор, забезпечуючи відмінні результати і роблячи приємність вашим гравцям.

***Ключові слова**: відео ігри, пулі, алгоритм, реакція на колізію, пробивання, рикошет, шутер*

УЛУЧШЕННАЯ СИСТЕМА РЕАКЦИЙ ПУЛЬ НА КОЛЛИЗИЮ ДЛЯ ВИДЕО ИГР

***Марков Д. К.** НТУУ «КПИ имени Игоря Сикорского», г. Киев, Украина, markov.dmitriy.dev@gmail.com*

***Рогоза В. С.**, Д.т.н.,проф. НТУУ «КПИ имени Игоря Сикорского», г. Киев, Украина, , rosvetnik@gmail.com*

***Аннотация**. В статье описываются устройства систем выстрела, нахождения коллизии и реакции на коллизию в видео играх. Описываются такие подходы как рейкастинг или хитскан который основан на сборе информации о коллижене луча, посланного из оружия по направлению выстрела. Разбираются плюсы и минусы подобного подхода. В статье также рассматривается баллистический подход, в котором формируется физический объект и выпускается в игровой мир. На основе этих подходов был разработан гибридный подход, который заключается в использовании баллистической логики основанной на последовательных рейкастах.*

На основе приведённых технологий разработана система реакций пуль на коллизию. Технически она получает все объекты на пути следования пули на данном кадре. Далее на основе этих данных система разбивает все участки пролёта по материалам и обрабатывает их отдельно друг от друга.

Описанная усовершенствованная система реакции на столкновение пули может обеспечить гораздо более глубокое погружение в любой шутер. С точки зрения производительности, запрос интервалов лучей основан на универсальном запросе лучей, поэтому единственная разница заключается в постобработке результатов, что означает, что влияние на производительность продвинутой системы минимально. Принимая во внимание, что предлагаемая система также основана на модифицированном гибридном баллистическом подходе, она может использоваться как есть в различных типах видеоигр, обеспечивая отличные результаты и доставляя удовольствие вашим игрокам.

***Ключевые слова**: видео игры, пули, алгоритм, реакция на коллизию, пробивание, рикошет, шутер*

Introduction

The video game industry consists of thousands of companies and thousands of games and many of them use bullet mechanics in them. A first-person shooter (FPS) is a video game genre focused on gun and another weapon-based fighting in a first-person perspective, in which the player sees the action through the protagonist's eyes. The genre has a lot in common with other shooting games, which is why it's classified as an action game. Since the genre's inception, advanced

3D/pseudo-3D graphics, realistic and believable logic and visual effects are must-haves.

Generally speaking, there are several ways to implement bullet mechanics in your game: hitscan, physics ballistic and hybrid approaches.

Hitscan approach

In video games, a hitscan is where the programming system decides where the gun/object is pointing, casts a ray in that direction when fired for a certain range built into the system, and sees if that ray comes into contact with another object in the line of fire. Hit scan is a term that may also be referred to as "hit instantly" because it hits the target immediately when fired.

A hitscan weapon is a projectile weapon that uses unaltered hitscan data to determine whether or not it has hit its target. The hitscan function is called when the weapon is deployed, and if an object is detected in the path, a hit is recorded. After determining whether or not anything was hit, the device calculates damage based on where the ray struck the target. The bullets essentially fly at infinite speed and have a linear or otherwise simple trajectory, which is a realistic approximation of a bullet's speed and accuracy over short distances since the effect is immediate.

The hitscan method can be tweaked by making certain surfaces reflective, making the hitscan rays go on indefinitely, or allowing several objects in the same line to be penetrated at the same time. To improve the realism, programmers may use hitscan functions in slightly different ways; for example, applying a random perturbation to the calculated path to simulate inaccuracy.

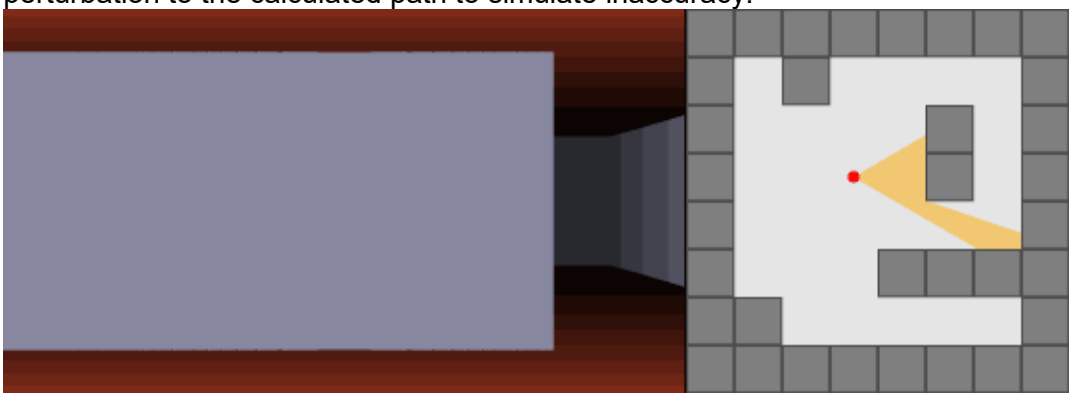

Fig. 5. An example of raycasting

Advantages

The primary advantage is the simplicity of the simulation, which uses relatively simple mathematics to calculate hits. Although bullets do not move at infinite speed via perfectly straight trajectories, they move fast enough that a hitscan solution is normally a reasonable approximation. It leaves the question of where a weapon has hit to just one function, streamlining the programming of weapons.

In terms of game design, it readily ties cause (the player presses a 'fire' button, executing a hitscan function) to effect (the hitscan returns a result, the player sees

the weapon's effect at that location). While a simplified model of real world ballistics, it makes games more accessible in that there is no need to aim slightly ahead of a moving target in order to compensate for the time it takes for the projectile to reach it. Although less realistic this model requires no understanding of real weapon handling in order to play the game, and reinforces the intuitive understanding that whatever the reticle is placed over will be hit.

Disadvantages

Visually representing the firing effect of a hitscan weapon can be difficult: since the weapon hits its target instantaneously, any bullet or projectile that comes from the weapon is merely a 'ghost', and where it lands may not necessarily represent its actual hit. In particular, a projectile bullet effect will always lag behind the effect of its hit, a problem which can be compounded by internet latency in online multiplayer gaming.

It's hard to change the path of a projectile once it leaves the gun with things such as wind and gravity. Since it is shooting out a beam that hits the target almost immediately there can be no real changes to the path. An example of this would be shooting someone at a full sprint 200 yards away and aiming right on them and still hitting them. If it were a realistic shooting engine you would have to lead that target but with hitscan you don't. It is impossible to have things like projectile ballistics or projectile motion in hitscan games. There are hybrid systems that use both hitscan and projectile ballistics but not at the same time. Some games detect with the hitscan method, then provide an animation with projectile ballistics.

With advances in processing and internet bandwidth, it has become more practical to simulate the ballistic nature of real-world firearms in real-time games by using a more realistic "projectile" model, spawning bullets as actual game objects with mass and velocity and continuously simulating them until they reach their target.

Projectile ballistics

It sounds pretty fancy, but the high-level idea is straightforward. Every bullet or projectile shot out of a weapon creates a new physics object in the environment. It has its own mass, velocity, and hitbox that the engine will track.

The advantages of using projectile ballistics shine in games where realism is the top priority. Since every projectile exists on its own, you can now factor in wind, friction, gravity, temperature; any force that should act on the bullet. Now that you can change the physics, players can now use weapons other than simple guns and lasers; you can now add grenade and rockets to your arsenal.

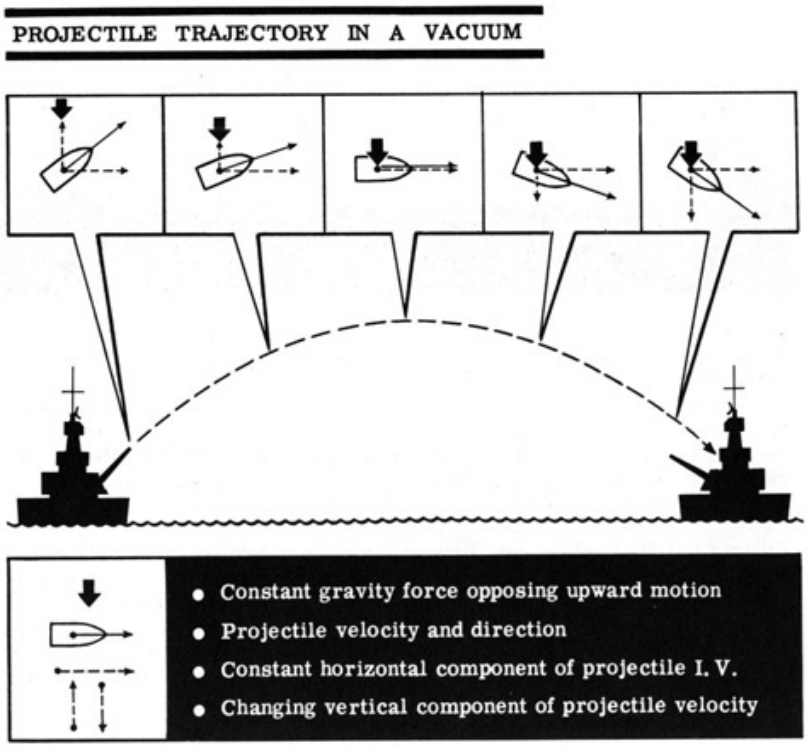

Fig. 6. Illustration of ballistic approach

Since bullets under this system aren't moving at the speed of light, you can also implement temporal features:

"Bullet-time" as seen in Max Payne, Sniper Elite or Superhot is feasible.

Travel time for projectiles, which means if you're taking a long-distance shot (or shooting a slow-moving projectile), aiming ahead becomes crucial.

Delayed explosions on projectiles, like grenades

With these additional computations, the processing is more taxing relative to using hitscan. Servers will have to do a lot more work to make sure all the objects are in sync, and discrepancies or conflicts in logic across clients have to be resolved not to create inconsistent experiences for players on the same server.

There are many ways around this to squeeze out as much performance as possible. An example of engine optimization is to have a "pool" of objects loaded before playtime, and "warp in and enable" them when needed. Once it hits a surface, you can play a ballistics animation and disable the projectile, saving it for later. This method will reduce some computation and memory costs from creating and destroying objects over and over again.

There are also multiple ways to do the computations, but the high-level difference is where they decide to process a "tick" of a game, a unit of time measurement:

The tick is calculated separately from the rendering logic, which means the game will have a more accurate representation of the objects even if there are frame skips. More logic is needed to calculate the exact time that passed since the last render.

Calculating the tick on every frame; binding the physics to the frame rate. If you disable frame rate caps or start to drop frames, you can see the accelerated or choppy effects on the world.

Hybrid approach

The developed approach is based on both described above. The main idea is to imitate the ballistic approach and bullet's travel time using speed value while still using raycasting as a basic mechanism of collision detection.

Basically in C++, this will be like

```
if (!bullet_manager.calc_bullet(bullet, delta_time)) {
        remove_bullet(bullet);
        break;
}
```

For now we will be using much lighter version of calc_bullet in order to demonstrate the initial idea of hybrid approach.

```
bool bullet_server::calc_bullet (ref_cbullet &bullet, fp32 &delta_time)
{
  if (fp_zero(bullet->speed))      // there is not reason to update it!
        return      (false);

  fp32 final_range = _max(EPS_3, bullet->speed * delta_time);

  const vec3f     bpos = bullet->pos.vec3();
  const vec3f     bdir = bullet->dir.vec3();

  uspatial::ray_defs RD (bpos, bdir, final_range + 1.f,
SPATIAL_FLAG_OBJECT|SPATIAL_FLAG_STATIC,
uspatial::O_FACE_NORMAL);

  rq_result_vec results;
  uspatial::ray_query(RD, results);

  process_results(results);

  bullet->fly_dist += final_range;
  bool use_line = !bullet->flags.use_curve || !bullet->target;
  vec pos = bullet->pos;
```

```
    vec dir = bullet->dir;
    fp32 speed = bullet->speed;

    vec  off; off.mul(dir, speed);
    off.y -= _gravity_const * delta_time * bullet->gravity_mod; //
connection to real gravity
    speed = off.len3();
    if (use_line) {
        pos.mad(dir, final_range);
        dir.mul(off, 1.f / speed);
    }
    bullet->pos = pos;
    bullet->dir = dir;
    bullet->speed = speed;
    delta_time = 0.f;
    return     (true);
  }
```

This leads us to the following result, where arrows representing distance that bullet has travelled during current frame.

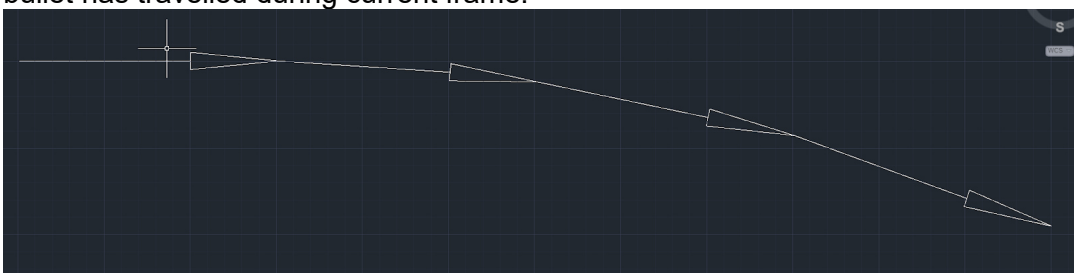

Fig. 7. Trajectory and updates of bullet in hybrid approach

Collision reaction system

The main idea of proposed reaction system is that we want to be able to interact with different objects and materials in the game. So we know that each and every object in the game is covered with corresponding material which describes its physical properties in a very simple way.

Based on these properties and on properties of a bullet, a gun, current trajectory, current energy of the bullet we can have good enough approximation for real-life collision reaction system. There are only 3 types of events that are used in current system: ricochet, penetration and jam. In the next sections we will cover ricochet and penetration, and if nothing of that has happened then on impact we will jam into the object.

Ricochet

In order to calculate if we have to ricochet first we have to check if object's depth (distance inside the object between 'in' and 'out' point). That is needed

because different materials have different ability to resist incoming impulse. For approximation we have used ‘min_depth_for_ricochet’ as a global parameter and we won’t be able to ricochet if the depth of an item is so low.

```
const BOOL depth_check = item_depth > game_level().bullet_manager().min_depth_for_ricochet();
```

The next part is we have to check angle between hit direction and normal of the surface. Here we have two parameters that will influence this check: bullet’s ‘piercing factor’ and material’s ‘ricochet factor’.

```
const BOOL angle_check = (piercing_factor * angle_coef + EPS_5) < desc.ricochet_factor;
```

Adding some sanity checks and debug information this is the result code for ricochet check:

```
bullet_server::ricochet_result bullet_server::calc_ricochet (fp32 &angle_coef, const vec &dir, const vec &hit_normal, const fp32 item_depth, const fp32 piercing_factor, const game_material::bullet_desc &desc) {
    if (!fp_similar(dir.len3sq(), 1.f)) {
        rlog("!can't ricochet, direction isn't normalized [%2.4f, %2.4f, %2.4f]", vec3push(dir));
        return (ricochet_result(false));
    }
    if (!fp_similar(hit_normal.len3sq(), 1.f)) {
        rlog("!can't ricochet, hit normal isn't normalized [%2.4f, %2.4f, %2.4f]", vec3push(hit_normal));
        return (ricochet_result(false));
    }
    const fp32 hit_angle = rad2deg(acos(_abs(dir.dp3(hit_normal))));
    angle_coef = clampr((90.f - hit_angle) / 90.f, 0.f, 1.f);

    const BOOL depth_check = item_depth > game_level().bullet_manager().min_depth_for_ricochet();
    const BOOL angle_check= (piercing_factor * angle_coef + EPS_5) < desc.ricochet_factor;
    ricochet_result result (depth_check && angle_check);
    return (result);
}
```

After we have decided that ricochet is what we have - we have to apply new params to a bullet. We have to change the direction of the bullet as reflection rule

suggests. Also, we want to show that part of energy was lost upon the impact and that is what *ric_infl* is for.

```
//:shared params
const fp32 item_depth= res.back.range - res.front.range;

//:trying to ricochet
fp32 ric_angle_coef = 1.f;
const ricochet_result ricochet = calc_ricochet(ric_angle_coef,
bullet->dir, hit_normal, item_depth, bullet->pierce, desc);
if (ricochet) {
    const fp32 ric_infl = 1.f - (desc.ricochet_factor *
ric_angle_coef);
    bullet->cur_power *= ric_infl;
    bullet->cur_impulse *= ric_infl;
    bullet->speed *= ric_infl;

    result.power = power - power * ric_infl;
    result.impulse = impulse - impulse * ric_infl;

    // in order to avoid penetration after ricochet:
    bullet->pos.mad(end_point, hit_normal, EPS_4);
    bullet->dir.reflect3(bullet->dir, hit_normal);
    bullet->flags.need_interrupt = true;
    bullet->on_ricochet();

    return;
}
```

Now, when bullet could have changed its trajectory after ricochet - we have to reiterate that same bullet until we have calculated all its path during the given delta_time

```
while (bullet->update_time_left(delta_time)) {
 if (!bullet_manager.calc_bullet(bullet, delta_time)) {
        remove_bullet(bullet);
        break;
    }
}
```

Penetration

Penetration calculation is quite easy - bullet has ‘piercing factor’ and we know percentage of energy bullet has maintained till that moment simply calculating

```
const fp32 power_factor = bullet->cur_power / bullet->init_power;
```

The whole code for deciding if we can penetrate object is quite simple

```
BOOL bullet_server::calc_penetration(fp32 &depth, const fp32 item_health, const fp32 item_depth, const fp32 power_factor, const fp32 piercing_factor, const game_material::bullet_desc &desc, const BOOL through) {
    depth = power_factor * piercing_factor * desc.max_penetration_depth;
    const BOOL pen_depth_check = through ? (depth > item_depth) : (depth > 0.f);
    const BOOL can_penetrate = !fp_zero(desc.max_penetration_depth);
    return (pen_depth_check && can_penetrate);
}
```

And after calculation we have to apply influence of the penetration to the bullet

```
    const BOOL penetration = calc_penetration(pen_depth, target_hp, item_depth, power_factor, bullet->pierce, desc, true);

    if (penetration) {
        const fp32 pen_infl = 1.f - clampr(item_depth / pen_depth, 0.f, 1.f);
        bullet->cur_power  *= pen_infl;
        bullet->cur_impulse *= pen_infl;
        bullet->speed *= pen_infl;

        result.power  = power;
        result.impulse = impulse;
        bullet->pos.mad(bullet->pos, bullet->dir, EPS_5);
        back_valid = item_depth > EPS_2;
        return;
    }
```

Sequential penetration

But the main problem is to decide how to penetrate a sequence of objects. First step is to learn how to penetrate object after object.

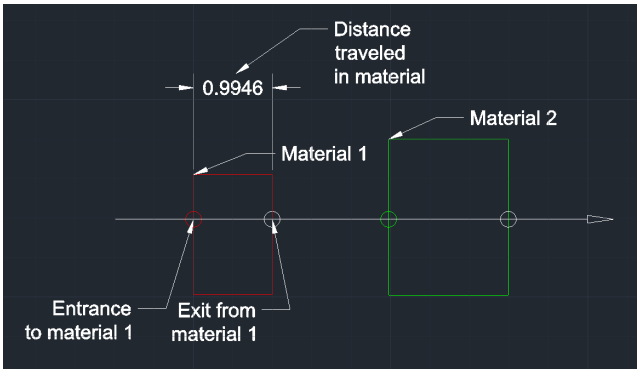

Fig. 8. Simple penetration illustration

There are two boxes, different colours represent different materials. Circles represent point of 'in' or 'out' collision with object. 'Out' collision is really the same change of material, because we had some 'material 1' and now we have 'air'. Color of the circle represents material in this particular collision.

In that case, bullet can easy determine 'in' and 'out' points of objects and knowing their materials we can bring up calculations from above.

But not always in the games models of objects are perfect. Sometimes they can be overlapped (which in the real world is not the case), for example some kind of block in the water inside video game not really pushing water out using Archimedes' principle. Thus, we have to invent a way to calculate penetrations of objects using 'game reality'. This idea was called *'Ray Intervals Query'*, because the basic approach was made using 'ray query' which is basically raycast but returning all objects on their way. Now we have to divide those objects into parts with only one material each.

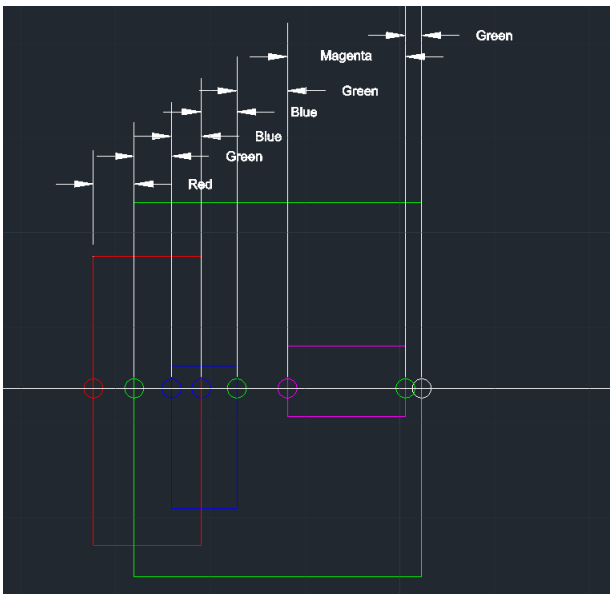

Fig. 9. Complex penetration illustration

This technique allows us to penetrate each part individually as if they were divided physically which is a good approximation for a video game with believable results.

Conclusion

Described advanced bullet collision reaction system can provide much deeper immersion for any shooter game. From the performance point of view, Ray Intervals Query based on general-purpose ray query, so the only difference lies in post processing of the results which means that performance impact of the advanced system is minimal. Taking into account that proposed system also based on modified hybrid ballistic approach it can be used as is in a various types of video games providing great results and enjoying your players.

References

5. Wikipedia, Shooter game, https://en.wikipedia.org/wiki/Shooter_game
6. Wikipedia, Hitscan, https://en.wikipedia.org/wiki/Hitscan
7. Tristan Jung, (2019, July 12), How do bullets work in video games, https://www.gamasutra.com/blogs/TristanJung/20191206/355250/How_Do_Bullets_Work_in_Video_Games.php
8. Arti Sergeev, (2020, July 17), How does shooting work in games: Hitscan and Projectile ballistics, https://80.lv/articles/how-does-shooting-work-in-games-hitscan-and-projectile-ballistics
9. Chris Language, Raycasting & Physics, https://www.raywenderlich.com/books/apple-augmented-reality-by-tutorials/v1.0/chapters/14-raycasting-physics
10. Maritime Park Association, (2013), Exterior Ballistics Part C, https://maritime.org/doc/firecontrol/partc.htm