RIVERZ
The Anatomy of a Physics Farm

Miguel Saraiva
Game Designer and Architect
Vosdal Studios Lda
miguel.saraiva@vosdal.com

Abstract
This paper describes RIVERZ, our solution for the architecture of the physics simulations of "Rivers Under Roar", a single shard, single instance, persistent massively multiplayer online (MMO) game, using a commercial cloud computing service

Author Keywords
MMO; Single shard; Single instance; Distributed physics engine; Cloud computing; Game development

ACM Classification Keywords
D.1.3 [Concurrent Programming]: Distributed programming; K.8.0 [Personal Computing]: General – Games

Physics Engines
Physics engine based games are great!

After computers appeared in our society, there were some games that simulated or tried to simulate the physical world, or parts of. They lend us to believe that the effects of our and the environment's actions had "real" effects on the actors. You got PONG [1] that had the collision effect of the ball with the walls and the paddles, the different bounce effects and not much more. It looked great, even in old oscilloscopes.

Some definitions

Sharding: The practice of creating separate copies of virtual worlds each of which cannot interact with each other.

Instancing: Similar to sharding but deployed on smaller content segment, like a dungeon. Two different parties of players appear to enter the same dungeon but do not see each other, as they are in their own instanced space.

Persistent world: is one where the world is persistent, meaning the world continues to exist even after users have exited the world and that changes made to the world state by its users remain intact for all users

A big chunk of years later, and we got some hardcore physics games, like the ragdoll ones (the great Porrasturvat™ [2], for instance), curious driving games (the muddy SpinTyres [3]), as well as others.

Other types of games also applied physical rules, but they were not simulations. Most of the better known are simple puzzle games that use "physics" as the rule of movement. We all know about the effects of collisions, throws and so on, so we already master the basic rules of those games. You know when you throw a (non-flying) bird into the air, it will eventually fall down… eventually hitting some other animal on the ground, like, for instance… a green pig.

Less hostile ones using physics as their foundations are Jeff Weber's. He made Diver and Krashlander (as well as others), and he is also the founder of Farseer Games and works on the Farseer Physics Engine (a derivative port from Box2D) [4]

Almost all of today's game have some kind of physics simulation… I bet the guys from Candy Crush Saga use some physics for particle effects. Even being a turn-based strategy game, "Crimson: Steam Pirates", for instance, uses a physics engine (Box2D) for the movement and collisions for a "more natural-looking motion" [5].

"Rivers Under Roar" (RUR)

So, what is RUR and why are we here?

In our own single line technical description, "RUR is a physics based single shard, single instance, persistent world MMO". That's it. And the interesting bits start here.

Physics engine

We had to make a choice of the physics engine to work with and there are lots. Lots of them. We needed something cross-platform and mobile-friendly, that we knew or that was easy to grasp (almost all of them are) and keeping out of the paid ones, we decided to work with Box2D. Box2D is stable, used all over the industry (the 2D part of Unity3D, Angry Birds, Tiny Wings, Limbo and many others [6]), so it was a good bet. Unfortunately, the latest C# port in existence is several years old, so we kept on using Farseer's Engine. Farseer is not a strict port of Box2D, it is C# optimized and has some extra physics features. And we already worked with it. Now, the only thing left is to put all players into the world and it is done.

Engine Server

So here we are. Having a physics engine put into a server. All physical objects will be in the server, running even if the player is offline, and interacting with all the other objects.

In a local device scenario, whenever you have a player action, you extract the physical actions from it, tell the engine about them, and simulate. Each drawing frame, you get the needed properties of the visible objects (at least, position and orientation, but we could also get velocity for some drawing effects), and display them… and wait for the next simulation or drawing frame. Memory to memory exchange not only is easy and "free" to code, but fast as well.

The physics engine being in a server, you don't get an easy and fast dialog with it. So, some work is needed.

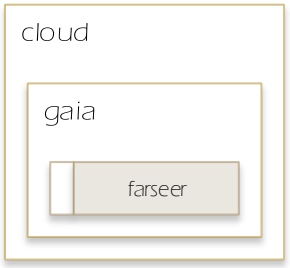

Figure 1 Gaia is the software "exoskeleton" around the physics engine

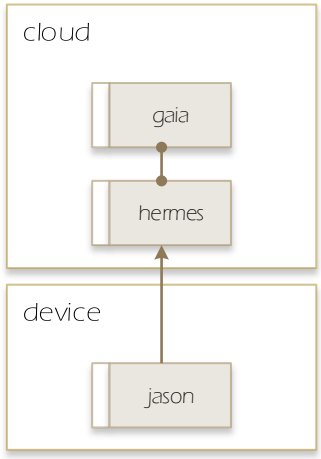

Figure 2 Global view of current state. JASON is the generic RUR client

First, we started by decoupling the Player->Engine and Engine->Player channels.

Player->Engine is a stress less channel. If you look at the APM (actions per minute) tables from StarCraft[2] [7], for instance, you can see it climbs above 300 for the most experienced players. And those values are for a game that encourages greater APM, with many possible actions accessible by keyboard shortcuts, and that is ok, but we are not going to have a high APM game. It will be a touch or mouse or controller based, so we are talking in the hottest battles at about 100 APM or less. That is around 2 per second. That is 1 every 500 milliseconds. So many milliseconds to work with! Great! Almost any net protocol will do. The only stress is that you cannot (or should not) lose any action from the player, but having reliable, ordered and error-checked delivery protocols available everywhere, it goes again stress free.

Engine->Player is a pain. Increasing the frequency of updated information decreases visual artefacts, jumps, staggered movements, ghost characters, while displaying the screen and acting on what happens on the screen. So, we need to target a very high number of FPS… or at least, the minimum that can be used without the user perceiving the steps in the simulation. Also, the Engine to Player must also send information of ALL the physical objects in the players' viewing frustum. There are some optimizations in here. But the rule of thumb for this we are using is: ANY object that the player can interact within the next frames, must have its information update sent in the simulation time. The others, the ones a bit farther, can be updated less frequently. It goes without saying, only the changed information will be sent.

Now we have a physics engine living in a server. That is really good, but as we are a very small team, very small, we do not have resources to build an adaptable server farm, nor have the time and skills to maintain all the servers in prime condition. For that, and for scalability issues and security and easy deployment around the world, we have decided to put the servers in the cloud. Not for any other reason than good past experiences, we have chosen Microsoft Azure. Hey, Forza, Titanfall [8] and Halo [9] could not be all wrong, right? We could have used the other cloud players, but past experience weights a lot in a small team.

The Server

To reduce potential issues like man-in-the-middle cheats, for instance, we took care that no physical properties are changed outside of the simulator. From the outside, players only state what action they want, and not which state they want to be in. So, a player does not state where to shot, but the will to do it, and the parameters for "when" are limited to a single "now". That action is sent to the server, and the server checks for several parameters, available bullets, orientation of the gun, time passed from the last shot, and only after that it decides to fire a gun or not. With this, a rogue installation of the game cannot do things outside of the box. No super-men allowed.

Decoupling the physics engine from the player also allows us to prevent attacks on it, and thus the world continues to move as desired without any external bad influence. It is the middle man server that can be attacked, and it can even go down without affecting the world. The only issue in here is that no one can enter the game, but their characters continue to live… unless

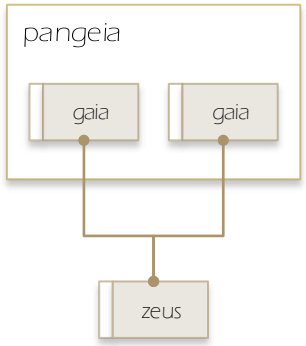

Figure 3 Coordinated PANGEIA

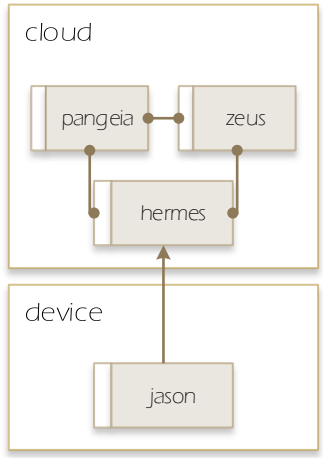

Figure 4 Global view of current state.

killed by some NPC (Non-Player Character) or the environment.

And that middle server, HERMES, the one receiving the commands from the player, is the only one to know some of the inner secrets of the physics server and as everything is internal, we can make it completely opened and unsecured and therefore, faster.

Intra-engine communication

Regarding network speed, the connection from JASON (the local device RUR client) and HERMES is from home to datacenter and back, whose pings vary a great deal. Between HERMES and GAIA, we use a virtual private network within the same cloud datacenter, and we get a great internal speed. From our tests, this intra-datacenter connections have normally less than 1ms of ping in a typical production situation.

Having an online game, we should have a 24h up time. With only one single server, when for any reason it crashes the datacenter by itself reboots the service and again we have it running. But this operation takes time. Some minutes time, and if you are in a battle, 5 minutes downtime is not good and you lose players. So, we should have (at least) two GAIAs running at the same time. We can create them in mirror mode so that when one goes out, HERMES knows about it and only communicates with the other. Hopefully both servers do not crash at the same time, but it could happen. Also, the entire datacenter could go off. With at least two GAIAs, we can cover most of the situations.

Distributed engine

But we have a new situation. Being multiplayer, hopefully being massive, it will be rather impossible to simulate all player's objects in one single engine. For some thousands, one engine could be enough, but there is a cap somewhere, so we opt to put this distributed aspect of the engine in before getting into that ceiling later on.

So, we are going to have multiple GAIAs (PANGEIA), running side by side. Each will have its own set of physical objects. All objects spread out on all GAIAs are all the active physical objects.

As they can go on and off almost at will, we need to have one coordinator. This coordinator, ZEUS, keeps track of everything that is going on. ZEUS commands the simulation frequency and informs GAIAs of which simulation tick to execute.

It also keeps track of the engines' workload and, if needed, launches a new GAIA if it finds some overloaded. In that case, it tells the overloaded to stop simulating some objects, and sends those objects to a new engine. Again, this happens in a local network, a single datacenter, so the performance is pretty good for our needs, keeping it under a couple of milliseconds of housekeeping work. The client (the player), will not be aware of this change, and that is our goal.

All of the above situations happen either in memory or on a very low-latency network. So speed is good. But if we need to get something from or set to storage, we get a heavy penalty. By heavy we mean 10ms or above per transaction, which is absolutely a no!

To solve this, we introduce a cache server. This server is the only one that loads and saves information from storage and, most importantly, pre-caches information

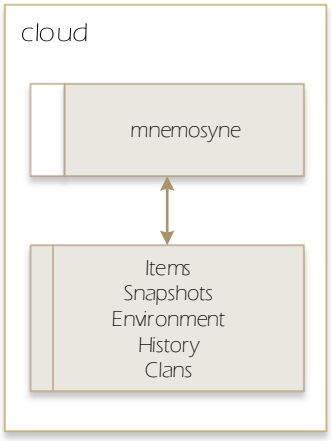

Figure 5 Access to "slow" storage (>10ms) is always via a cache server

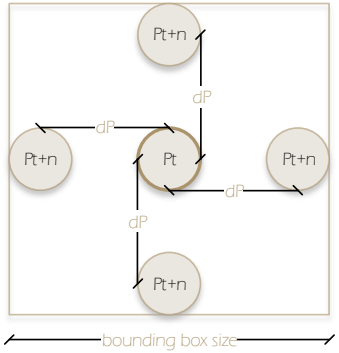

Figure 6 Bounding box dimensions

that it thinks PANGEIA will need in a near future. That way, when PANGEIA requests something, it will already be in memory at some engine, and it will have just a few milliseconds delay.

It is this cache server, MNEMOSYNE that gets all state from all GAIAs and from time to time, dumps it to the storage. This is currently done once per second, which in a catastrophic event (everything goes down) means that we have just lost less than one second of world time. And, as we keep all the player actions in storage, we can recreate that lost second.

ZEUS, if it finds a dead GAIA, knowing which objects were being worked on, just asks MNEMOSYNE for those objects' state, and send it to a working GAIA (remember, we have at least two GAIAs running on the PANGEIA).

Islands

One question that was put early on was how to separate objects into different engines.

We are not trying to simulate the "butterfly effect" where every object influences every other one. We have an area of influence, more specifically, we have each objects' bounding box expanded with the maximum theoretical movement for the next *n* seconds. That way, we know that two objects that do not clash their expanded bounding box, will not directly interact in the next *n* seconds, and that is good for us. They can exist in a different island and therefore in a different engine. An island is just a collection of objects that are potentially in collision trajectory… or not. We have no need to have atomic islands. So one island can have objects that are not going to collide but, if they do, for sure those objects are in the same island. Apart from the collisions, we also use the islands to move objects between servers.

For that, we use a simple but effective brute force method. Taking into account the actual velocity value, plus acceleration and potential acceleration, plus some external effects (environment, mainly), we determine the maximum displacement possible for the next *n* seconds. To make it faster for processing, we ignore rotation and rotation effects, but take into account the maximum length of the object, and we assume it to be a sphere, and this way we can ignore its change in orientation. We end up with a greedy bounding box for the next *n* seconds, and we can compare this with the other islands' bounding boxes. Ignoring velocity orientation, also simplifies the bounding box as it is a simple square now. All of this shortcuts are put into place to minimize the needs of the islands processing.

So, moving objects between GAIAs is in fact moving some of its islands. And as we group objects taking into account their possible collisions in the future, we have that time to change islands composition. Again, it is not for PANGEIA to check this. It is the coordinator's job to do this, while it waits for the simulations to finish their frames. If ZEUS finds some colliding islands, it will reorganize them appropriately, and inform the corresponding GAIA to restructure its job before the next simulation frame. It will continue to work with exactly the same objects, but from a different view, they will be organized differently so that when needed, complete islands can be moved between the engines. Also, islands are the minimum unit that gets saved to storage. No individual object are. As each engine's

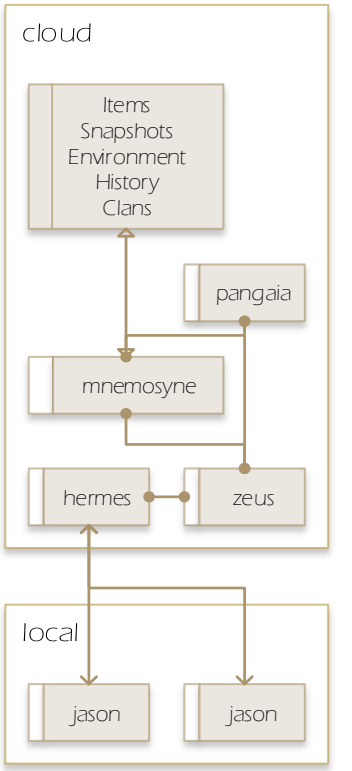

Figure 7 The complete scenario

atomic unit is an island, this does reduce the number of cloud and storage operations which are not free.

Conclusion

This is the physics framework for the RUR and as it should be running 24/7, one of the most important metrics we wanted to target was the stability. The other two are the transparency for the player (as much as possible), and the simplicity of the programming on the local device part of the game. If we have this cloud architecture building as solid as Wurtzite Boron Nitride [10], we have a very good foundation for what is coming in the next months.

More Information

This is part of the online architecture behind RUR, and the most demanding. The game itself is in constant development, and only some pieces are completed, RIVERZ being one of them. If you want to know more about ourselves and this new IP, just head over to our Twitter [11, 14], Facebook [12, 15] or Indie DB [13] pages and ask for directions.

References

[1] Atari PONG - The first steps - http://www.pong-story.com/atpong1.htm

[2] Porrasturvat™ http://secretexit.com/freeware

[3] Spintires is full of mud and hardy Soviet trucks http://www.pcgamesn.com/spintires/spintires-full-mud-and-hardy-soviet-trucks-its-also-top-selling-new-release-steam

[4] Farseer Physics Engine http://farseerphysics.codeplex.com/

[5] How Crimson: Steam Pirates (#1 game on iPad) Was Built in just 12 Weeks http://www.gamasutra.com/blogs/AljernonBolden/20110926/90267/How_Crimson_Steam_Pirates_1_game_on_iPad_Was_Built_in_just_12_Weeks.php

[6] Unity Game Engine To Get Official 2D Game Support And A Built-In Ad Solution http://techcrunch.com/2013/08/28/unity-game-engine-to-get-official-2d-game-support-and-a-built-in-ad-service/

[7] [StarCraft²'s] Actions per minute http://starcraft.wikia.com/wiki/Actions_per_minute

[8] A closer look at Titanfall's not-so-secret weapon: Microsoft's cloud http://www.engadget.com/2014/03/10/titanfall-cloud-explained/

[9] Orleans: a Platform for Cloud Computing - Microsoft Research http://research.microsoft.com/en-us/projects/orleans/

[10] [Wurtzite] Boron nitride http://en.wikipedia.org/wiki/Boron_nitride

[11] Vosdal Studios @ twitter http://twitter.com/vosdal

[12] Vosdal Studios @ facebook https://facebook.com/vosdal

[13] Vosdal Studios Company http://www.indiedb.com/company/vosdal-studios

[14] Rivers Under Roar @ twitter http://twitter.com/_r_u_r_

[15] Rivers Under Roar @ facebook https://www.facebook.com/RiversUnderRoar