- uPhysics -

Content

Introduction

uPhysics is a highly scaleable, dynamic and performance friendly custom physics engine for Unity3D. It's intend is mostly to replace unity's internal trigger system, but is aswell fully capable of replacing collision detection and optimizing generic movement. It supports various collider shapes, including mesh colliders. Every shape can be set to be a trigger.
It is designed to simulate massive amounts of moving and interacting objects while still delivering high framerates for the main game logic. This is possible due to extensive multithreading, moving every time expensive work from unity's mainthread to other threads and therefore cores, setting all processors on fire. It is therefore aswell suited (and risen from the very same need) to run on Unity3D game servers, like for MMO's.
It uses modern techniques to squeeze the most speed out of it as possible and is fully implemented via C#, making end user API calls very cost friendly.

As of now, the system is not fully mature, but it is under constant development and updates/bugfixes are guaranteed to be implemented/fixed in short intervals. Content and "still-to-come-content" can be found in the respective section of this manual.

Components

uPSimulation

Technique

Game physics and physics engines made for such can be distingueshed into several different aspects which can take huge influence on the final performance. One such aspect is the so called “Broadphase”. Its intend is to detect and create pairs of objects which might collide during the current or in the timespan to the next physics frame. The performance and accuracy of this phase is commonly the most crucial one since checking huge amounts of objects can end up taking $O(n^2)$ time in the worst case scenario. Another factor is the accuracy, since lots of object pairs will create a possible bottleneck inside the so called “Narrow phase” which detects final collisions and respective collision meta data.

uPhysics uses a spartial space partioning method called the “Hierarchical Hash Grid”. The idea behind it is to create a grid out of uniform cells into which every object is finally inserted. However, one grid is not sufficient to deliver accurate results because bounding radii of objects might drastically vary. Therefore, a hierarchy of grids is dynamically created in order to fit every objects bounding radius. Cells of each grid will decrease/increase in size by a factor of two. Each object is assigned to a grid whichs cell size is greater than the objects radius but whichs next smaller grid cell size is smaller.

Objects are inserted into the cells of those grids by forming a hash out of the current world coordinates. Finally, when perfoming a broadphase, every occupied cell is checked against half of its neighbore cells and overlaying cells of greater grids. This will effectivly ensure that every object pair of neighbouring cells is only checked once. (if at all)

Furthermore, cells of a grid are not represent in an array form or anything similar since it would simply take way to much memory. Just consider an allocated grid of 2048*2048*2048 cells, each pointer taking 4-8 bytes… Not considering the acctual cell class yet… Therefore, only occupied cells of a grid are stored via a dictionary using the world position hash as key. This makes it aswell ultimatly possible to recycle cells and more note worthy, expand all grids on runtime, making it absolutly cost effective.

As a small explanation as of why I wrote this small size section, since it obviously doesn't have anything todo with acctual setup or anything similar. Its obviously always good to understand what technique is used in a certain system. But more crucial to me is to show the acctual progress which has been done.
In the previous version, one had to create multiple simulation areas for different spaces. Objects, were not allowed to walk out of the bound of a certain area. Objects, cluttering on the y axis would take the same cell, effectivly reducing performance since grids were only allocated in x-z directions. Aswell as max cell size was limited to the previously set size of the simulation, leaving huge objects without physics. Ultimatly, allocating huge areas would take up a good notch of memory.
However:

- Objects can be as small/great as you wish, the grids extend get rescaled if needed
- No bounds need to be set, the grids will grow as the game flows
- Objects cant move out of a simulation anymore, the grids will grow with objects traveling
- The simulation is now allocated in x-y-z directions (3 dimensional)
- The simulation is more memory effective since only occupied cells are stored and recycled
- The simulation is able to span unitys complete box, therefore only one simulation is needed

Physics Timestep

As in most common Physics engines, uPhysics aswell runs on a fixed timestep. This infact means that any change made to a colliders transform will take immediate effect (depending on load balancer) but the acctual collision detection and response will be delayed untill the next physics update takes place. This creates the need of several different collision detection methods as explained in the collider and rigidbody sections.
Setting a lower timestep will make the physics loop run more often per second, where as setting a higher timestep will make the physics loop run less often. The acctual physics loop doesn't take much performance from the mainthread,

however, eventually other cores beside the mainthread core might reach their performance limit so one has to consider what the best timestep would be. The more often the physics loop runs per second, the less interpenetration will happen and the more accurate the overall simulation will be. However, on the contrary more physics loops will obviously cause more pressure on the machine. Especially when dealing with huge object counts, one should set a higher timestep, letting the loop run less frequent. This is caused by the fact that not everything can be run in parallel. Certain parts simply need to run each after another to ensure a stable simulation. Furthermore, a physics engine can never run in O(1) time, instead the lowest one can archive is O(n). Meaning, the more objects need to be simulated, the longer a physics loop will take and the less loops will run per second.
(For the previously called “Movement Timestep”, see the uPLoadBalancer / uPTransform section)

Terrain/Heightmaps

Contrary to previous versions, Heightmaps of terrains no longer need to be baked. Instead they will be automatically created at runtime. It is aswell fully possible to update them incase they get changed procedurally.
As of now, real collisions with terrains are not possible since no mesh gets created. Infact, only the heights of the terrains heightmap are used to “clamp” a collider to the respective height when positioned below it. However, such an effect is only applied to rigidbodies.
Using the heights instead of a real mesh is definitly faster and delivers sufficient results in most cases. However, when using very long objects which have a great surface towards the heightmap, results might not be as accurate since it would than span a range over several different heights. This might end in objects interpenetrating with the terrain. The same effect takes place when an objects scale is rather big in relation to the terrains detail shape. (holes, hills, …)

FUTURE WORK: Heightmaps are definitly at a high priority and will be represented in a form of a collision mesh in the future. This will especially be needed when additionally work on the rigidbody dynamics is done.

uPTransform

In order for uPhysics to detect any changes made to an Gameobjects transform, means position, rotation and scale, the framework implements its own transform component. The uPTransform is just a simply copy of the built-in transform component which just keeps track of any changes made plus granting the possibility to load balance respective method calls.

Furthermore, uPTransforms are used to lock certain transforms in a hierarchy. When a collision appears, objects are either moved out of interpenetration or objects velocity is reduced to the time of first impact. However, when moving objects out of interpenetration it is not always entirly obvious which transform acctually moved the collider. Just assume a deep gameobject hierarchy with the last child containing a collider. When moving the root object, the child would get repositioned on collision. However, since this is most of the time not wanted, uPTransforms can be locked. On collision, a loop will move up the transform hierarchy starting from the collider untill it finds an unlocked transform or the root object and reposition that certain transform. This will grant fine grained controll over how your transform hierarchies behave. Commonly it is sufficient to just lock every single transform and submit movements to the root gameobject. This will cause repositioning and moves to be submitted on the same gameobject and might save some headach. Furthermore, when dealing with multiple colliders in a hierarchy, a so called compound collider, it is absolutly necessary to lock every transform untill the common transform is reached under which all other colliders sit in the hierarchy. Not doing so will eventually cause trouble since childs will move and reposition, not granting a static compound collider. The tip of locking all transforms and using the root aswell applies here.
Each uPTransform can have a so called load balancer. Each load balancer has a unique tag which an uPTransform needs to state. If an load balancer with such a tag does not exists, the tag will be reverted to the standart “Default” one. Additionally, it is possible to not use a load balancer at all. This will cause the properties of the transform to be set immediately.

FUTURE WORK: Locking of transforms might get obsolete in the future when the system is turned over into a pure dynamic rigidbody simulation.

uPLoadBalancer

The uPLoadBalancer is a component which will execute and dispatch calls made to any uPTransform (when setup to use a load balancer) to unity's transforms and therefore updates unity's world simulation. It operates on a fixed timestep and updates registered transforms on these discrete intervalls. However, transforms are not immediately updated when a timestep kicks in. Instead the update is split over several frames, the frames inbetween the current and the next timestep. (Therefore "load balancer") Objects which get updated don't follow a specific order so one cant say which object comes first and gets updated first. But it is guranteed that each and every object is updated when the next timestep takes effect.

As of why this is a worthy performance increasing tool is simple to explain. The bottleneck in uPhysics is NOT the physics itself or anything which comes with it. Infact, the bottleneck exists in unity itself and consists of updating the internal simulation. Any call to respective transform methods will be absolutly costy. Therefore it makes sense to split those calls over several frames and I highly recommend to use it.

The uPLoadBalancer aswell features a method to register custom methods to an event which is raised right before any update is done to the unity transforms of registered objects. This can be usefull to implement update like methods and general movement methods. The event can either be executed on the mainthread or on a seperated thread. Though, one needs to take care for not calling any mainthread protected methods from the unity API.

Colliders

Colliders in uPhysics completly replace unity's colliders and are responsible for collisions. Unlike in the previous version, there are now way more collider shapes available. Every collider can be rotated, scaled, positioned and centered as we are all used to from unity. However all this needs to be done via the uPTransform in order for uPhysics to know.

Each collider shape can aswell be set to be a trigger. Additionally, there are two different trigger modes. One mode consists of checking the bounds, means when both bounds overlap, the trigger will fire. The other mode will check if the centerpoint of an other object is within the trigger.

When dealing with fast moving objects, one should increase the "maximum movement per timestep" variable. This variable shows the bound radius of an object depending on which an object is inserted into a grid with a size directly greater than the radius. When having fast moving objects, it is likely the grid cell size is to small and objects jump several cells in an abitary direction. This might lead to collisions not being detected and triggers not being solved up correctly.

OnTrigger and OnCollision callbacks can be registered by fetching the collider on an gameobject and by subscribing to the respective event. By default, events will be executed on the current worker thread which is not the mainthread. This might be unwanted in cases where you need to accses the unity API in such an event callback. Therefore, colliders have the possibilty to execute their events aswell on the mainthread.

By default, collisions of normal colliders without rigidbody will use a discrete method to detect collisions. Means, objects are checked against each other at certain timesteps and moved out of interpenetration incase they overlap.

FUTURE WORK: There's a great chance that normal colliders might get static and unmoveable in the future. This is not bad at all since movement would be restricted to rigidbodies, granting a clean and stable simulation.