Simple Hierarchical Unity Destruction Reactive System

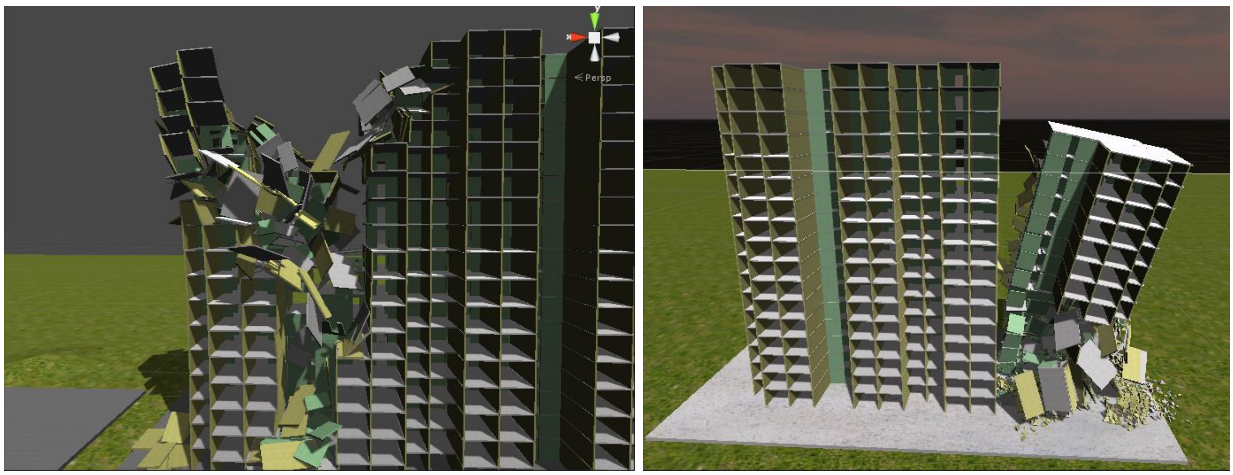
USER GUIDE

by NotSoOld Games, 2017 (c)

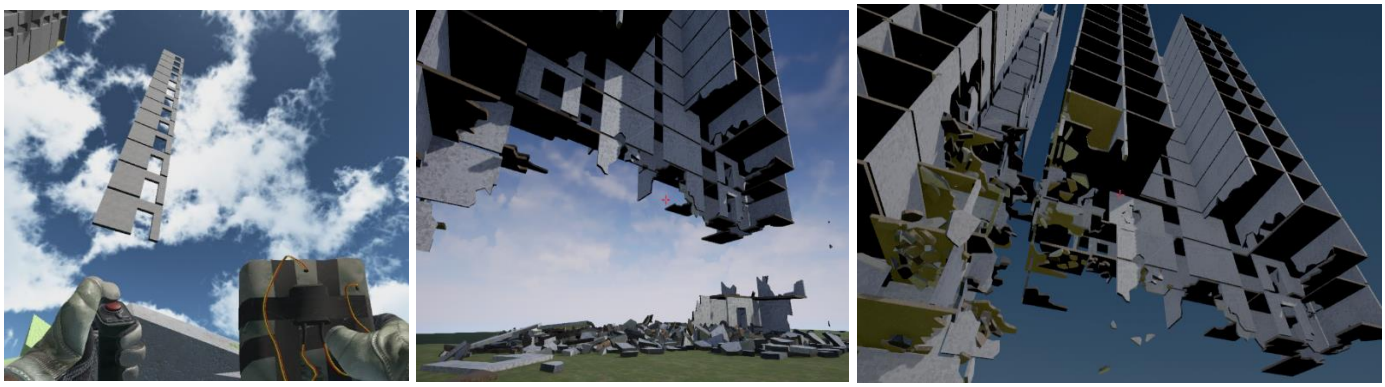
Table of Contents

Overview and History

Simple Hierarchical Unity Destruction Reactive System (or SHUDRS, like "shudder" of the destructible things, you see? :D Got dammit, you can simply call it SHDestruction, that already makes sense), or SHDestruction, is a scripting solution for the problem of destructing things in Unity at runtime. Though it isn't really a problem, because we can use instantiation to spawn fragmented object at a place of whole object when it's being destroyed and so on, but I'd experienced A LOT of problems dealing with huge objects like towers, houses, which need to take their stability into account. Main problem was, like always, performance. PhysX really doesn't like when you overload realtime scene with physical objects, so, my attempts to make all walls of the house rigid and/or connect them with joints cause lags…huh, they're even worse than lags, calculation time of ***one*** frame (deltaTime, for convenience) sometimes reached 1500-5000 ms. Plus, experiments in Unreal Engine cause PhysX (due to freezes) to just… delete problematic objects from the scene immediately. I needed to do something with all this crap.



Beautiful but laaaaaaaaaggy.



*That was neither falling nor working at all. Go f*ck yourself, PhysX…*

By the way, I'm not trying to say that NVidia brought us very bad an inefficient product, I'm too unexperienced to claim that. It was just not suitable for my mission.

So, after a while, I thought about a destructible like a structure that consists of smaller parts which are connected in a known way. And these structures may be multiple and also connected, and so on. Data about connections *inside* structure gives us a possibility to check *integrity*, i.e. keep together its parts if they appear to be connected, and divide structure into substructures if they don't. Data about connections *outside (between)* different structures gives us a possibility to manage whole destructible in a hierarchical way (that's why system is called so), piece by piece. It's about optimization and logic – you think about table made of wooden planks, house – of walls, and some walls have windows with frames, and so on. As turned out later, it also gives me a convenient possibility to check and manage *stability* in particular parts and structure itself.

Structure of a Destructible - Main Idea

For now, biggest possible Destructible in SHDestruction consists of Constructions. Constructions are made of smaller parts which are called Elements. A part of each Element – Fragment – is the smallest part. Important that you don't need to have your Destructible exactly in a form of a Structure, if it isn't too big: it can be just a Construction or even an Element.

So, how it works:

Fragment is the only Destructible's content on the scene that is presented physically, because it has a collider. In fact, it *must* have a collider to respond to collision events.

Every time a destruction event happened, no matter what happened, Fragment sends a signal to its Element, which, after receiving signals from all child reporting Fragments in Update(), starts to execute its integrity check in LateUpdate(). (Order is critical, so I created a TimeManager to manage all integrity checks, by the way.) There may be two situations: when destruction event(-s) happened during current frame divide Element into parts that are not connected physically, and when they do not. In the first case, Element will be divided into separate Elements, following the logic that every separate Element in the hierarchy is a group of physically connected Fragments, and only of them. I think you call also call this one of the main ideas. In the second case, connection data is corrected, but nothing's separating from Element.

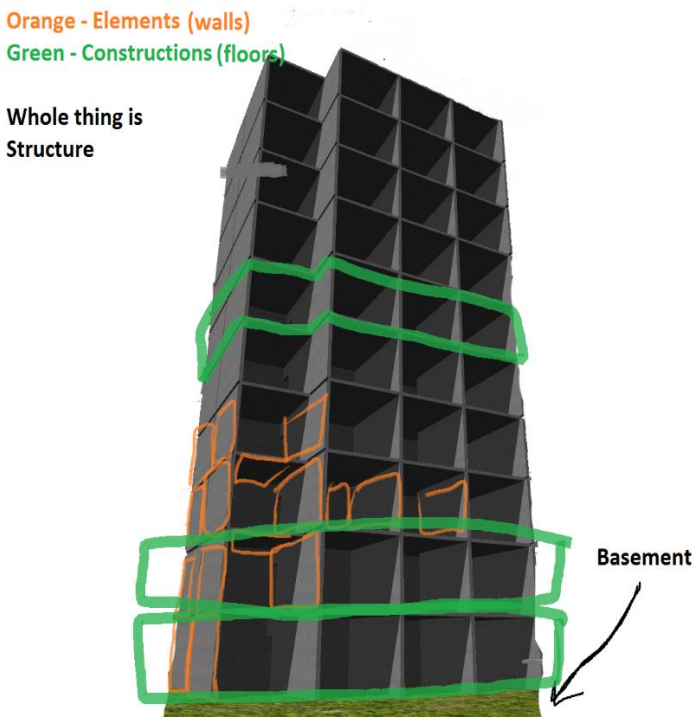
If this was a Root Element (which means it doesn't belong to someone bigger, it just lives his own life), this is where things stop. But if it isn't, the signal from Element comes to its parent Construction, which suspends its integrity check for one frame (to say exactly, execution starts right after Update() loop stops executing because of "yield return null" statement; see "Unity Event Execution Order" in the Unity Docs), and then integrity and division calculations repeat at Construction's level (code is different but meaning is the same).

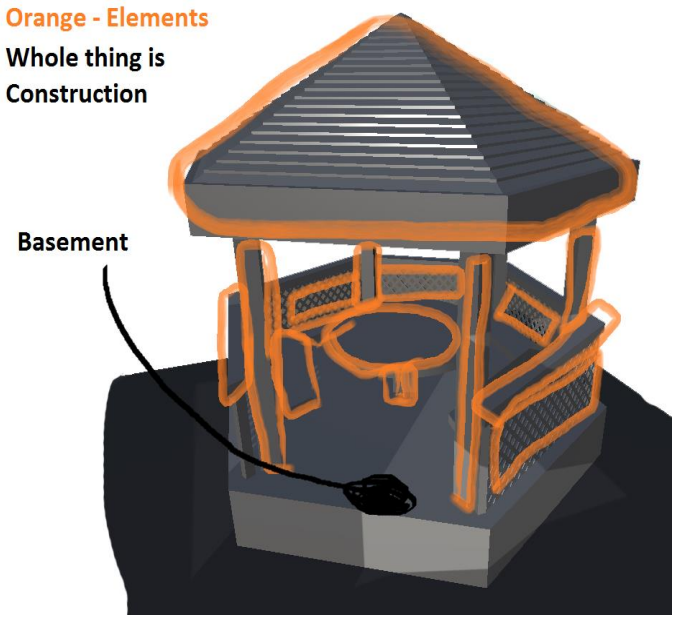
Then again – if it was Root Construction, that's all, otherwise, send signal to Structure, wait for next frame and calculate. After dividing (or not) parts in Structure, execution stops: destruction event has been successfully processed.

Why the hell FOUR levels of processing? :D I don't really know, but it happened to be more efficient that two and more than enough for nearly all situations when you need to crush something.

Why divide processing into three different frames? Well, proper destruction of big things is a cost operation, sure enough, so it was a deliberate solution. But it also turned out that it's far easier to manage processing order when you clock the work of your system (and I'm reminding that system needs at first manage all Elements, and only then - Constructions, and only then - Structure).

Simple examples how you can treat objects as hierarchical Destructibles:





These are specific examples, I suppose, but it's not hard to catch the point.

But wait, what's "basement"? And how Elements which are not connected to anything in the destructible, are behaving?

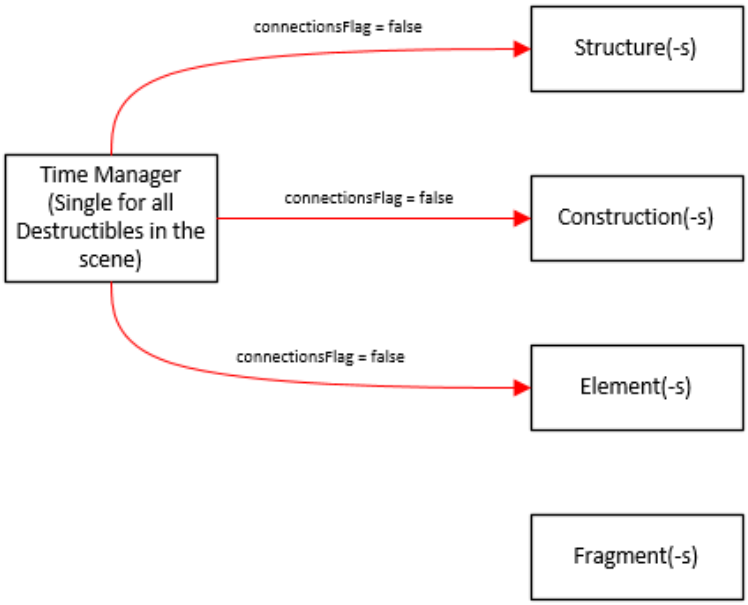
Good questions. There is where some PhysX features are used. Basement is a special object that supposed not to move (like Unity static) under (or above, maybe it's ceiling… just to mention it's possible) the Destructible. It uses basement object during initialization, generating data about connections to the so-called "ground". This data is essential, it is used to calculate whether we're stable or not, or maybe this part of Destructible after dividing is now floating in the air. So, if check with connection to the ground failed, Element gets a Rigidbody and behaves like a separate physical body. By the way, all Root Destructibles also have Rigidbody at the top of hierarchy (cinematic until they lost connection with basement). This is kinda… optimization, I think, because you don't have to have Rigidbody on every Element, and Unity Compound Collider System allows physics engine to treat the entire Destructible as *one* physical object.

It is also important to mention that, honestly speaking, integrity check is not the only check which happens after collision event. Each level of integrity check (Elemental, Constructional and Structural) triggers stability check on Destructibles' re-initialization (which happens after something had been separated from our Destructible). So, Destructible will wait for 30 frames and then re-calculate stability values, and, if stability check fails, system destroys some data in Destructible in a such way that Destructible is starting to fall (like it became unstable and can't stand still anymore). Because of 30 frames delay in *each* level, stability checks at different levels occur in different frames (because Elemental level will do it after 1+30 frames, Constructional – after 2+30 frames, and Structural – after 3+30 frames). SHUDRS allows you to tweak stability values (and disable stability check if needed) to achieve good-looking results.

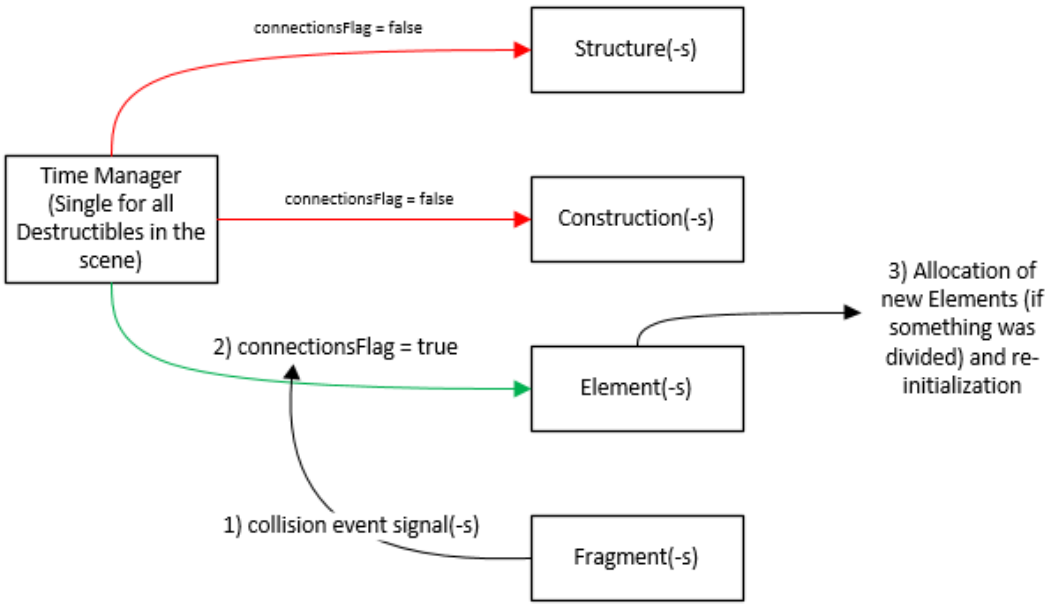
What is "stability check" in a nutshell? Well, system calculates two vector points in the world for each Element, Construction and Structure. They are "mass point" and "support point". Mass point as an average sum of all Fragments that are still presented in Destructible. It's like their mass distributed evenly. (So, it's not a good approach in the terms of physics, because fragments can have different volumes and densities, but it's still quite good.) Support point is generally (meaning switches from one type of Destructible to another, see code commentary for more detail) an average sum of all Fragments which participate in any type of external connections (like grounded ones or connections between Elements). In most cases instability means that mass and support points became too far from each other (like we hold a lot of mass by only one connection at the side, it must fall then). Again, this is not an ideal way to take down weird unstable constructions, but it is one of the best while I

still don’t want to work with PhysX and to learn discipline named “strength of materials”. :D

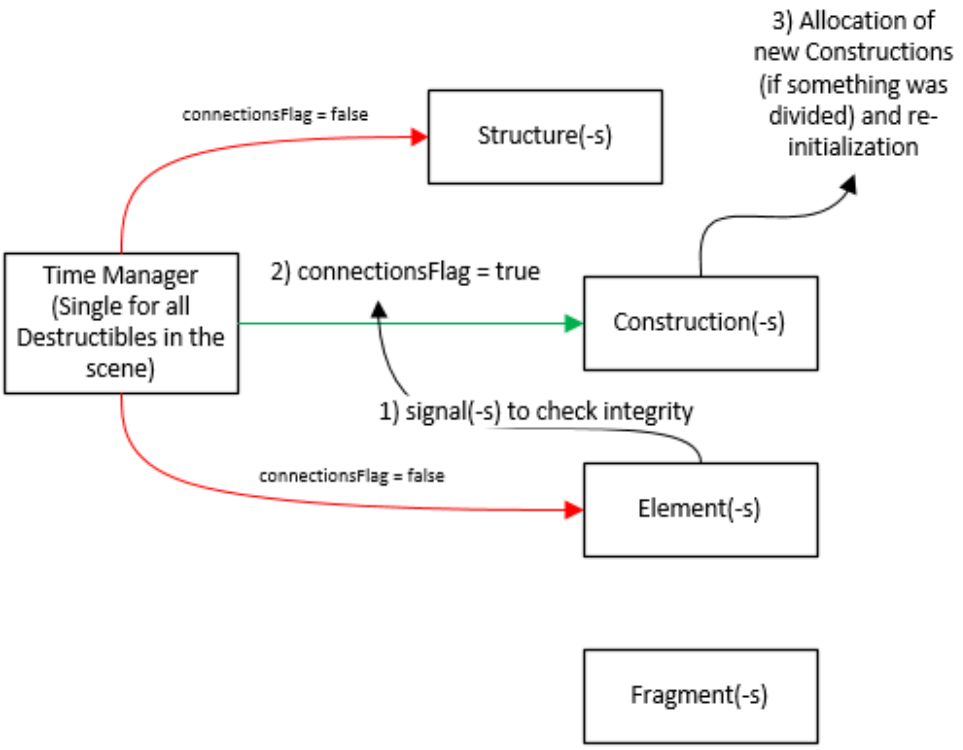
Full time scheme of working SHUDR system is presented below:



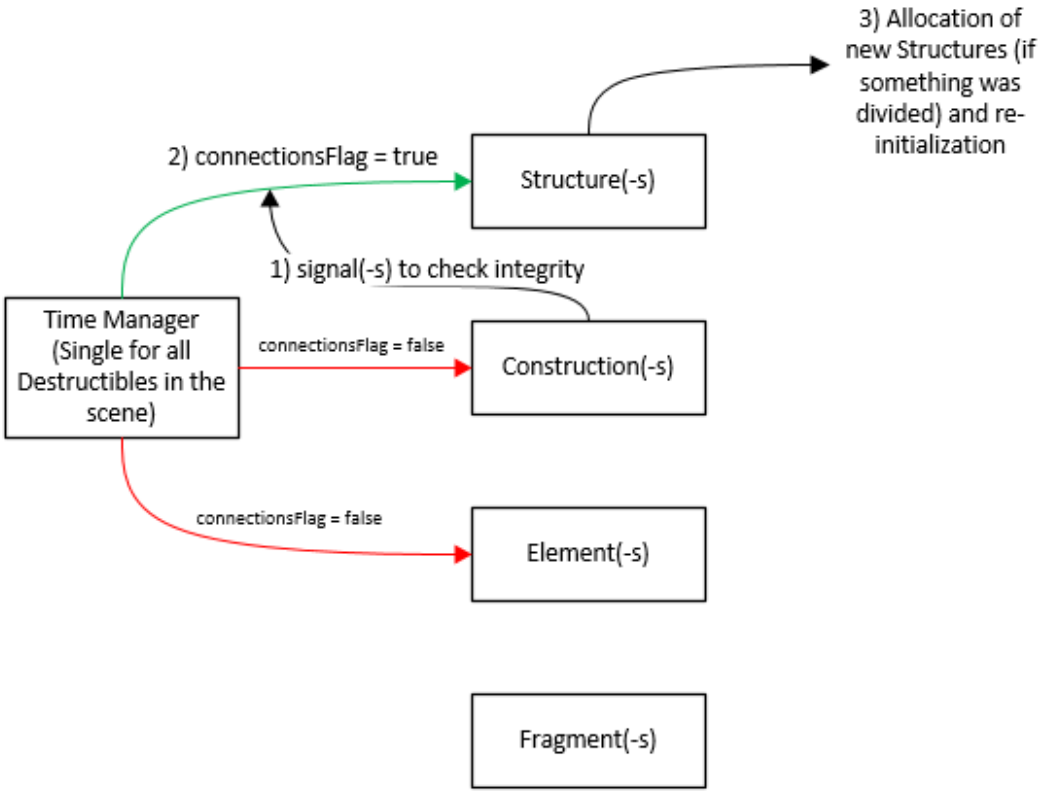
Idle



1st frame

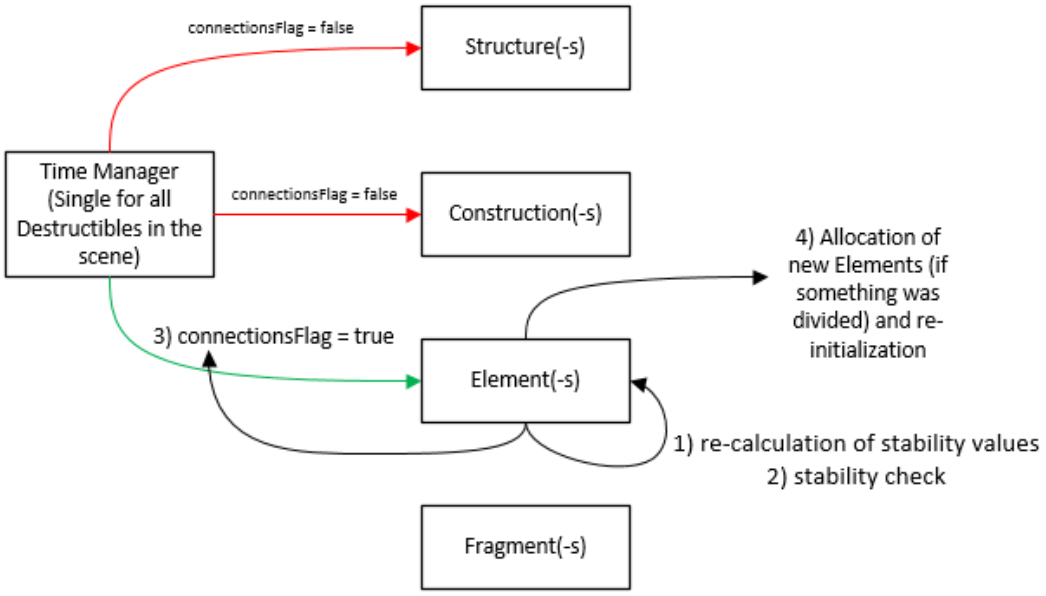


2nd frame

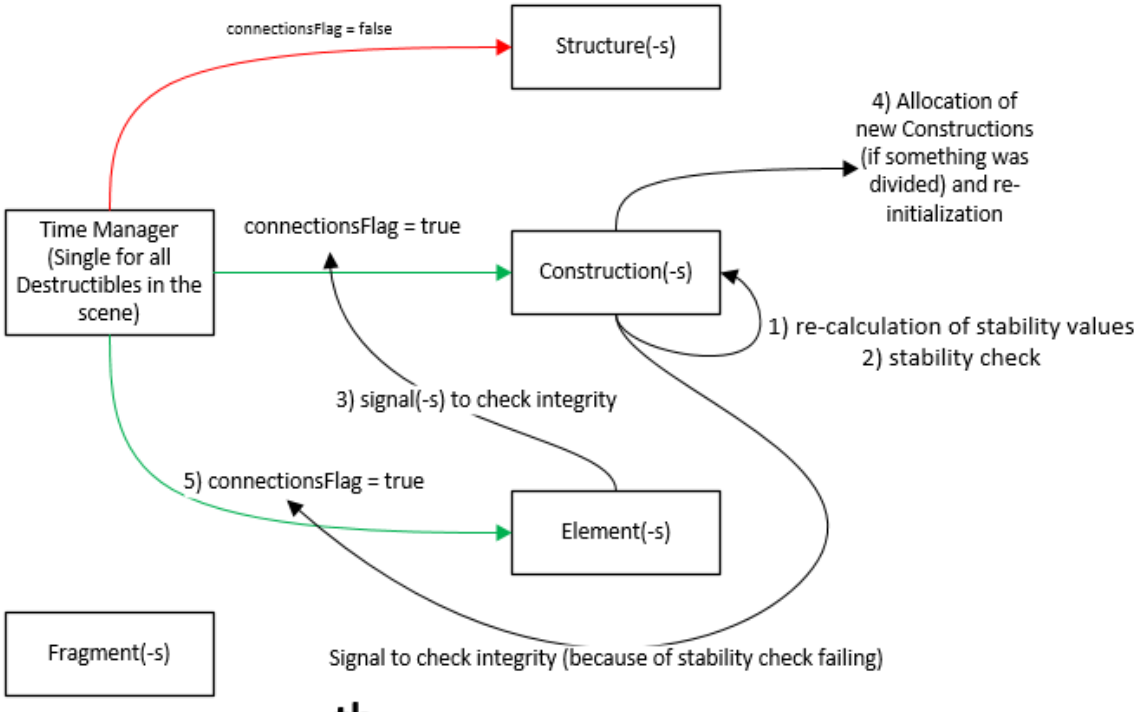


3rd frame

(There things become slightly complicated, but you don't need to worry and, maybe, even try to understand this, because stability checks nearly never fail and, then, divide Destructibles, so at frames 31-33 there will be only [successful] stability check; moreover, sometimes integrity checks do not send signals to upper levels because everything is fine and nothing needs to be checked. :))



31th frame



32th frame

The *SHUDRS* namespace

Fully relying on scripting, SHDestruction is a complex script system that has several scripts which are working together, passing information and control to each other. System namespace contains:

- *.Destructibles* namespace. Here are represented all scripts which work directly with Destructible objects in the scene.

    - *Fragment* is a class, objects of which must be attached to every smallest piece of a Destructible, like plank or wall fragment, etc... Requires a collider, and, because of that, whole bunch of Fragment objects represents Destructible as scene object that can be reached with Physics class methods. Has main entry point – DestroyFragment() method – which can be called from user's custom code to perform simple vanilla per-fragment destruction.

    - *Element* is an abstract base class for Root and Construction Elements. In fact, both are just containers for interconnected Fragments and can initialize and re-initialize Fragments (which means to produce/refresh useful data) and calculate integrity and stability of whole Element. Has an adjacency matrix of all its Fragments (which is used during integrity check) and a list of "Connection" structures which represent connections of this Element with other Elements. Due to wonderful C# use of object references nearly *everywhere*, simple Connection with references to two connected Elements also gives the system an information about which Constructions these Elements are connected to, and all respective data that can be reached. :D Implements *IDestructible* interface.

    - *RootElement* is just the same container as Element, but, since it doesn't belong to Construction, it has a Rigidbody, i.e. is an individual physical object. Methods differ respectively. Inherits *Element* class. Implements *IDestructible and IRootDestructible* interfaces.

    - *ConstrElement* is a container for Fragments in the case Element belongs to a Construction. Again, different calculation methods inside have differences comparing to RootElement. Inherits *Element* class. Implements *IDestructible* interface.

    - *Construction* is an abstract base class for both Root and Structural Constructions. Contains methods (again…) for initialization and re-

initialization and for computing some stability data. Has adjacency matrix of its Elements (if you can't understand how it works, replace Elements with Fragments and Construction with Element in this context, it's pretty the same), which is used during integrity check. Connection structures, if needed, are accessed through references to child Elements. Implements *IDestructible* interface.

- *RootConstruction* is a class that represents container for ConstrElements. Can compute their integrity and divide new Constructions containing more than one Element (logically since they can be still physically connected). Has a Rigidbody, as it is root Destructible. Inherits *Construction* class. Implements *IDestructible and IRootDestructible* interfaces.

- *StructConstrucion* is a Construction-like container with pretty same methods but it's a part of Structure. Also, as usual, some methods vary, like integrity checking. Inherits *Construction* class. Implements *IDestructible* interface.

- *Structure* is the biggest possible Destructible. It can't belong to something. It has adjacency matrix of Constructions and can divide itself into separate Structures. Implements *IDestructible and IRootDestructible* interfaces.

- There also are two interfaces made mostly due to give a user an access to SHUDR System's components via scripting, for example, to give a possibility to perform destructions. So:

  - *IDestructible* is an interface containing two user entry points – one is for full-elemental/constructional/structural detachments (caved floor, fallen roof, all house is going nuts and lost its basement, etc. :D), other – for turning Destructible with connections into same Destructible without any connections, so, for example, house will behave, after calling this method, like house of cards; or entire Element will just blow into pieces. Obviously, both of these entry points can be reached in the class objects, not interfaces – and they really can, search for ones that are marked with a "*(can be accessed through interface)*" commentary. But I think there may be an advantage with the interface when you hit some Destructible but don't exactly know what type is it, and you just get IDestructible object and collapse things.

- *IRootDestructible* is also an interface that helps you to deal with any Root Destructible without knowing its exact type (I already used it in that way in OnCollisionEnter() method in Fragment, you can go and see it). And it can be expanded with parameters shared between Root Destructibles, it's up to you. :)

BY THE WAY, yes, I've almost forgot, I made the system flexible *on purpose*, especially classes from *.Weaponry* namespace, so you, my friends, can expand it for your needs. ;)

But we need to continue:

- *.Weaponry* namespace contains two base abstract classes which *must* have descendants declared by you. In general, these are two types of a weapon shells that have a default possibility to cause destruction events. These classes are:

  - *Projectile* is a weaponry type, which objects are flying in the air, affected by gravity and drag, and cause to do something when they hit something that they can hit; so, generally, its effect appears on collision. There can be lots of sub-types: bullets, tank shells, rockets, cannon ball, and much more. This base class contains some general methods, like raycast collision processing (note that we don't need Collider, only Rigidbody, to operate, and raycast's length is slightly more than the distance we will fly during fixed frame) parameters, like hit mask and impact type, and you can add some extra properties in your inherited classes. Bullet class is given like an example for Projectile class extension. Built-in PerformDestruction() method is given like an example how you can destroy things with Projectile-like weaponry.

  - *Explosive,* on the contrary, produces its world impact when *activating*; it doesn't need to hit something. Like a grenade or, I don't know, breaching charge. Again, like in Projectile class, some basic properties are written, and you need to create and derive your own classes. Built-in PerformDestruction() method is given like an example how you can destroy things with Explosive-like weaponry.

Why did I make such "empty" classes and why did I make them at all? Well, I found it convenient to destroy things when you have such types of weapons, they are strongly defined and have a lot of flexibility at the same time. And, as turned out, it is more convenient to fire raycasts and overlaps from shells rather than try to solve all collisions in the Fragment OnCollisionEnter() method. (And fast projectiles tend to go through colliders sometimes or echo their collision position

wrongly because they collide between frames… so, there was a bunch of problems.) But if you like, you may re-write Weaponry system as you like. Since you don't touch any scripts from *.Destructibles* namespace, system will still work.

And there remain classes with *Editor_* prefix, which are all about Custom Unity Inspector scripting, not about SHUDR System (but if you need, I'll write a section how you can add your own properties and make them appear in Custom Inspector), and some utility classes. Here they are:

- *DestructibleBaseObject* is a class which is only needed to create objects and mark scene GameObjects with them as static for SHUDR System. If a GameObject has DestructibleBaseObject object attached, it supposed not to disappear, crush, move, etc., so Destructible can lean on it (some calculations also depend on that). Otherwise it will lead to strange Destructible behavior (generally, it will just fly in the air, because movement of DestructibleBaseObject is not tracked for performance reasons… I think :D).

- *DestructionUtility* is an utility file containing delegate for TimeManager, custom struct, custom class and a static class for SHURDS namespace (which is empty, and you can find it ironic if you read the commentary for this class :D). Struct is an already familiar Connection structure, which contains information about interconnected Fragments which belong to different Elements. Class is my own implementation of .NET BitArray (the standard one can't be serialized by Unity), specialized to store bit adjacency matrices.

- *FragmentationSettings* – one object of this class (at least one!) must be attached to every Destructible. May be more than one, but more "specific" settings will rewrite less "specific" (it means that special settings for particular Element or Construction in the hierarchy will replace common root settings for all Destructible). It represents some properties of a Destructible – its material tag, tags and name substrings of objects this Destructible (in fact, its Fragments) must react to, debris particle systems for type of Destructible, and so on.

- *TimeManager* is a class with functions of a chronometer. It clocks integrity checks of Elemental, Constructional and Structural (in that order) levels in a strict way, and all of these must be processed in LateUpdate(). When I used not to use TimeManager and just put integrity checks in LateUpdate() methods, I wasn't really sure when these checks were happening, so I decided to make it clear. So, now, you *must* have one (and only one!) TimeManager

object per scene to make your Destructibles work (if you don't, Root Destructibles will manage it by themselves).

All code of the system is well-commented (I worked REALLY hard on this), so you can easily understand every method and why it's being used. :)

Why no multithreading was used? Well, it's nearly impossible to implement in my case, and, in addition, is pretty useless. I mean, multithreading is sometimes useful, when you need to do some calculations in parallel, but SHUDRS really likes to do calculations successively, and almost all of them are tightly connected to MonoBehaviour namespace, so you can't just go and make multithreading; plus, every object will have its own thread, but I have hundreds of objects on the scene and only six processor cores… So, generally, multithreading is not a good idea here.

If you want to implement some – I won't stop you, be my guest and don't forget to share results, that will be great!

Why there is no proper code documentation (like Unity API Docs)? I think my system only needs a manual because its code is kinda self-documented, it had a lot of commentary how things work and are calculated, etc. In addition, system has only a couple of user entry points, i.e. methods which user is allowed to call (and properties to get/set) without causing system's malfunction, and these entry points are also well-commented and were described above.

How to Use?

- Setting up Your Destructible:

1. Arrange your future Destructible object's hierarchy in the following way:

Structure *(optional)*

Construction1 *(optional)*

Element1

Fragment1

Fragment2

…

Element2 *(optional)*

Fragment1

Fragment2

…

…

Construction2 *(optional)*

…

…

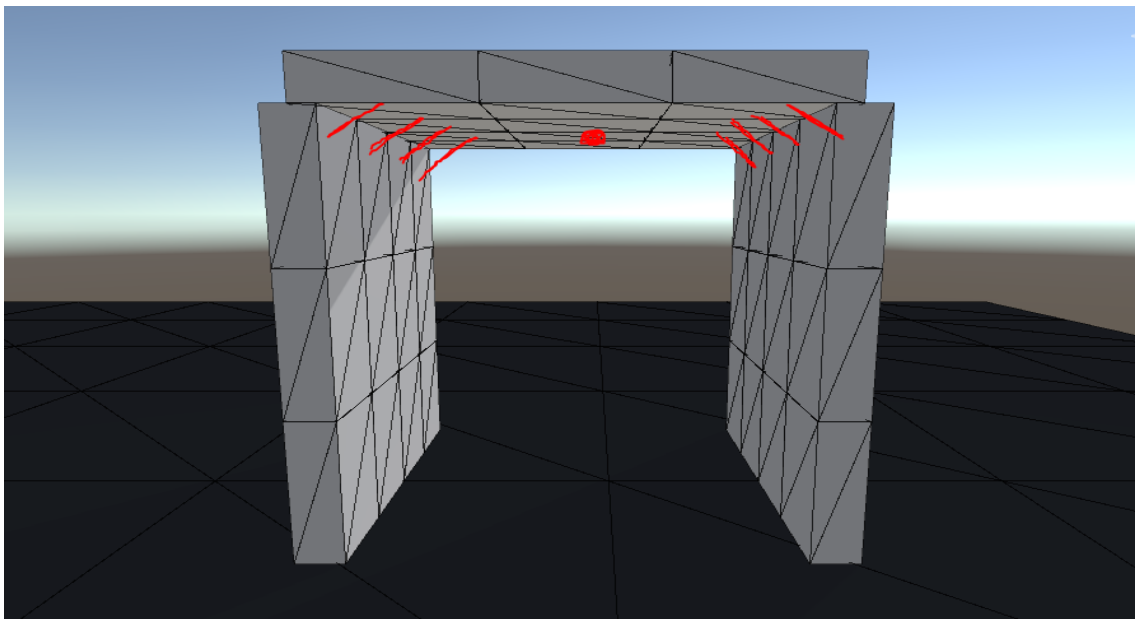
"(optional)" means that your Destructible may have not a Structure, but a Construction or Element as root type, and there may be as many Fragments, Elements and Constructions as you need. So, in simpler words, make a *hierarchy* of your object. MAKE SURE that all GameObjects in this hierarchy present Fragments or its containers – there ***must*** be no rubbish (excess) objects, because system sometimes uses number of children of the container and their child number; garbage will make this information incorrect.

2. Add a RootElement, RootConstruction or Structure script to root object (choose type of Destructible according to your hierarchy setup). After adding, click "Initialize" button.
3. If you change something in your hierarchy, to initialize Destructible again just click "Initialize" one more time.
4. If you want to clear your hierarchy from all scripts and components that were created during initialization, click "Cleanup …".

That's all for the most basic setup! :D But you also have a bunch of settings which are available in the SHUDRS.

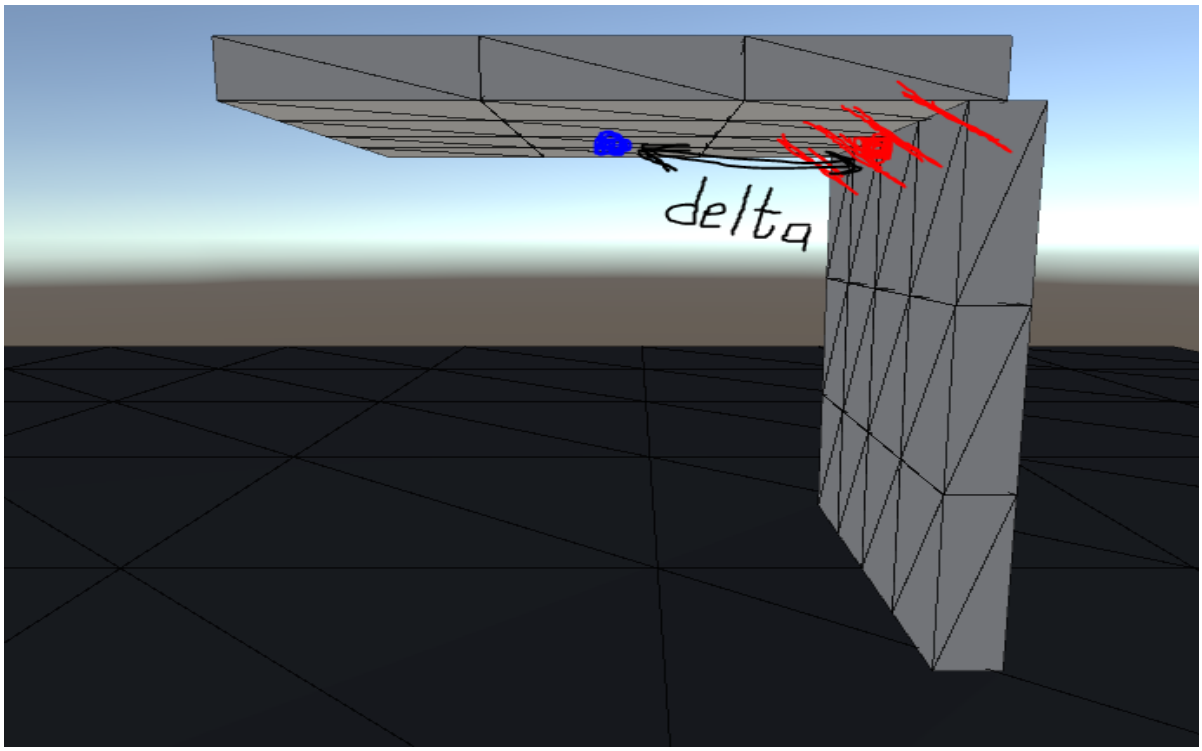
- Tweaking Stability Threshold:

Each container, like Element, Construction or Structure, has a property named "Stability Edge Value". Generally, it is a maximum magnitude (length) of a vector between mass and support points (you can see code to know how they are calculated for each type of container), or, simply, a maximum distance between these two points. Imagine a "table", consisting of three planks-Elements:



At the beginning the support point of upper plank will be in the center, because there are four connected Fragments at the left and four – at the right. By the way, mass point will be in the same place, because average sum of positions of all Fragments of upper plank is in plank's middle now.

But if we're going to destroy one of the support planks…



Mass point (blue) remains in the same place, but support point (red) now is in the place where remaining connections are. Length of delta vector between these two points is what we're trying to measure and compare to our threshold. If it is too big, construction becomes unstable and must fall.

So, what's the conclusion? Sometimes you should tweak stability threshold: if it is too small, objects will be going to seem very unstable, and if it is too big, they will always remain awkwardly stable.

Three additional moments:

1. Mass and support points are calculated for every type of a Destructible in a really different way. I think there can be no better explanation than the code with commentary itself.
2. Global default stability threshold value can be set in the scripts where the variable is declared (value after "=" will be default for every object of that type).
3. If you set threshold to a value, <= 0, you will disable stability checks.

- Rendering Optimizations:

For undamaged Elements and Constructions there is a possibility to display mesh of a whole Element/Construction instead of a group of Fragment meshes for lowering rendering load. If you have a mesh of a whole Element (Construction), add a Mesh Filter and Mesh Renderer components to the according Element (Construction) GameObject, fill theirs mesh and material references and tick "Use Renderers Switch" box of the Element (Construction). System will do all the rest for you. Note, that if you enable this option without adding components which are needed to render a mesh, system will disable renderers' switch back.

- Additional Fragment Properties:

- Damaged GO – if your Fragment isn't going to leave Destructible in its unbroken form, you can set this reference to replace undamaged model of object with damaged one. It's like when you deal with metal pipe and you want it to bend after being damaged. DamagedGO needs to have Mesh Renderer and Mesh Filter, other components will not be used anyway.

- Fragment Is Indestructible – some Destructibles have parts in their construction that you may not want to be damaged or annihilated, like armature in walls, house's metal skeleton, and so on. Since you are not allowed to have "rubbish" fragments (that are not related to a Destructible as a dynamic destructible

object) in the hierarchy, you can place them like functional and mark as indestructible after initialization. That will make them resistant to any collision event's impact, but they will still be considered as a functional part of Destructible.

- Start Health – just a float parameter from 90s, when every object has some health points and was destroyed when they reach zero :D You can use it for your purpose in the code as you want. This value in inspector is initial, and current health value is accessible through scripting.

- Using Fragmentation Settings:

Each Destructible must have at least one FragmentationSettings component at the root. But there can be additional components at different containers in the hierarchy; they will replace root settings with their own settings for Fragments they contain. You can always find which FragmentationSettings affect a Fragment because every Fragment displays name of the GameObject with settings component which it uses.

Fragmentation Settings by default contain:

- string arrays "tagsToMove" and "nameSubstringsToMove". Unlike projectiles and explosions, who command the Fragment to destroy itself in some way, some objects may have ability to move Fragments out of the Element because they're, for example, heavy and have much kinetic energy. Like tank colliding to a wall. Fragment will take tag and name of colliding object and compare them to strings in these two arrays. When tagging is not enough, you can just write a substring of a name of the object.
  Example: tagsToMove contains "rock", nameSubstringsToMove contains "tank" and "car". "tank_01_Heavy" with no tag will break the Fragment, "superccaar" with tag "bla"- won't, but all objects with "rock" tag – will. I think, you got the idea. :)
- materialTag and specialTags array is an additional way to get the information to the environment how different is this Destructible. While materialTag is pretty strict and can define material per-Construction or per-Element (glass,

concrete, wood, etc.), specialTags may be used as user wants, in any way it is possible and convenient.

- directionalDebris and goDownDebris are GameObjects-templates with particle systems (may also have child Particle Systems) that are, if references present, automatically "substituted" by system at the place of impact. Also, meshes for particles and type of one of two systems are chosen by system. You can look at example debris and make your own when you get the idea. Directional debris is a directional burst, used for replacing Fragments when they-re totally destroyed. "Go-down" debris is a bunch of particles that look like they simply go from the Fragment when it is dividing from Element.

- Debugging Features:

As SHUDRS is not a very simple thing to code and tweak, I added some debugging through process of scripting:

1. You can print adjacency matrices (that present interconnections between Fragments in Element, between Elements in Construction, and between Constructions in Structure) by clicking corresponding button in the Inspector. If you know theory of graphs and how adjacency matrices of graph work, they may help you in some issues.
2. Show Stability Gizmos – use this flag to hide/unhide two gizmos unique for each container in Destructible. They show both mass and support points and may be useful to tweak stability edge value.

- What About Destructible Inside or Connected to Another Destructible?

In the first case, just let it be inside (things I thought about were tables standing at destructible floor of a building) and turn "isKinematic" false at root's Rigidbody. In the second case, dealing with lamps and chandeliers attached to destructible walls, just make them a part of whole big Destructible. Remember, you can **not** use DestructibleBaseObject object on things which are going to be destroyed!

- Am I Allowed to Put Something in Hierarchy as Fragment's Child Objects?

Hmmm, let me see…

Tries to found documents how his system works

Makes a total mess on the desk

Remembers that he didn't write any documents about tests

Oh yeah, you are! :D Only some facts should be considered, like one that these objects will be destroyed along with the destroyed Fragment, and so forth. You are free to experiment.

Known Issues

By now, there are some bugs which I feel I must write down.

- Sometimes stability check still says that Destructible (or a part of it) is stable when it's not. It's not really a bug, but a shortcoming of simple systems with approximate approaches to calculations. I'm sorry for that :)
- In some very rare weird cases things may interrupt each other and cause a malfunction of a Destructible. This is a problem of complex scripting systems, it's not my fault :D

Conclusion

SHUDRS is a very simple, not heavy for physics engine and pretty fast for processing, reactive system for destroying big things which need to take their stability into account. You can use it to create large structures and process them very easily and using convenient instruments from your code – system has opportunities to be expanded and entry points to be used for controlling destruction processes.

Contact and Say "Thank You" :)

My name is Dmitriy, known as NotSoOld, (a.k.a. DimikYoo or TheDimaSomov), I'm a 20-old Russian programmer. If you have some questions or issues, you can contact me at 1000lop@gmail.com.

Say "Thank you":
PayPal: 1000lop@gmail.com
WebMoney: Z420064389985

BONUS: How It Is Made in Battlefield

When one talks about destruction, first game everybody might remember is Battlefield (3, 4 or 1; some old people may also mention Bad Company 2 :D). I played this title a lot of time, of course, and took some notes, watching environment of Caspian Border or Shanghai being destroyed.

So, the main idea here is "do not complicate things too much EVER". What do I mean? In Battlefield, you won't find any interactively destructible house (or something similar) which has more than 2-3 storey. They have some objects called "walls" which can be destroyed by something big and explosive, and when number of destroyed "walls" hits critical value, all construction collapses. Destruction and collapsing is pre-animated, so it doesn't load system at all and looks incredible for naked eye. 2-storey houses can have script with simple logic which tracks their stability.

Sometimes I saw indestructible houses, but they had sectional facades with had an ability to collapse, of course, sectionally.

Everything that is bigger is fully pre-animated, and in some cases, there is so little dust generated around destruction event that you can see whole building going down through ground and model of destroyed building going up. Lame. :D

Some destructible objects are made using principle similar to my interconnection matrix in the Element (like wooden fences) – while some planks still touch ground, they can support others.

So, what's the conclusion of all this information? DICE did a really good job, making stunning and interesting interactive things out of nearly… *nothing*. I'm very proud of them. Maybe, sometimes you need only not to think about over-complicated things, know when to stop, and you'll be successful. (And, unfortunately, it's not about me, that's for sure. :D)


E
42
[HEAT]Nightmareg


45
90
30MM CANNON READY
29 G
340.2 B
100
268.7