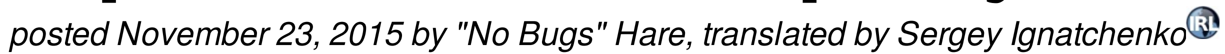
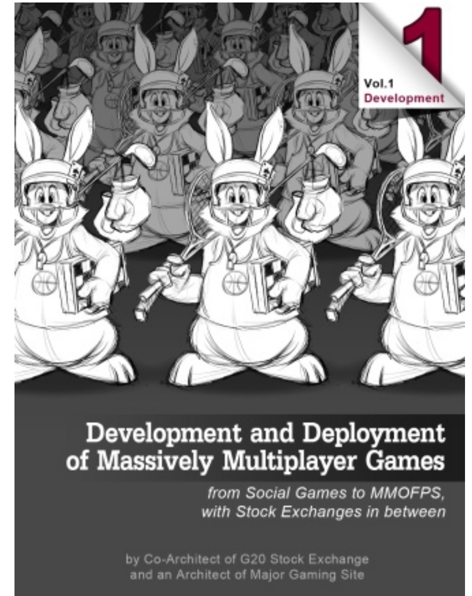
IT Hare on Soft.ware

# Chapter V(a). Modular Architecture: Client-Side. Graphics from "D&D of MMOG" upcoming book



*posted November 23, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko*




[[*This is Chapter V(a) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*

*To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.*]]

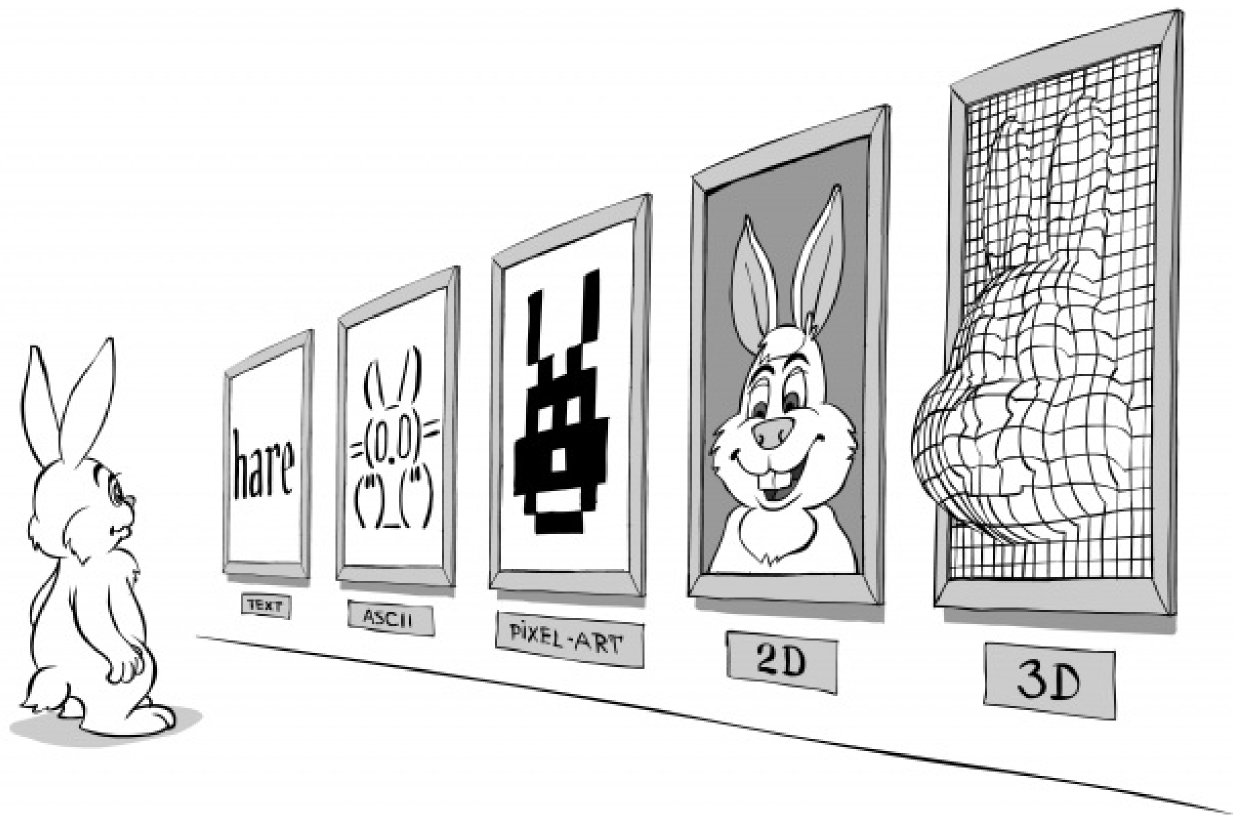
*– How do you program an elephant? – One byte at a time!*
*— (almost) proverb —*

As we've discussed in Chapter IV, there are basically only two viable approaches for building your game: we named one of them an "Engine-Centric Architecture", and another a "Modular Architecture". Which of these approaches is right for your game, depends a lot on the genre and other Business Requirements; the choice between the two was more or less explained in Chapter IV.

In this chapter, we'll discuss "Modular Architecture" in more detail. If you're going to implement your game as an "Engine-Centric" one, you still need to read this chapter; while most of these decisions we're about to discuss, are already made for you by your game engine, you still need to know what these decisions are (and whether you like what specific engine has chosen for you); and whatever-your-engine didn't decide for you, you need to make the right decisions yourself. Applicability of the findings from this Chapter to "Engine-Centric Architecture" and to specific popular game engines, will be discussed in the next Chapter ([[TODO]]).

## Graphics

One of the first things you need when dealing with client-side, is graphics engine. Here, depending on specifics of your game, there are significant differences, but there are still a few things which are (almost) universal. **Note that at this point we're not about to describe subtle implementation details of graphics engines (these will be discussed in Chapter [[TODO]]). For the time being, we only need to figure out a few very high-level things, which allow us to describe the engine(s) we need in very general terms (to filter out those which are obviously not a good fit for your game) and to know enough to be able to draw an overall client-side architecture.**




## On Developers, Game Designers, and Artists

For most of the games out there, there is a pretty obvious separation between developers and artists. There is usually a kind of mutual understanding, that developers do not interfere in drawing pictures (making 3D models, etc. etc.), and artists are not teaching developers how to program. This, however, raises a Big Fat Question about a toolchain which artists can use to do their job. These toolchains are heavily dependent on the graphics type, on the game you're using, etc. etc. When making decisions about your graphics, you absolutely need to realize which tools your artists will use (and which file formats they will produce so that you can use these formats within your game).

For some genres (notably FPS and RPG), there are usually also game designers. These folks are sitting in between developers and artists, and are responsible for creating levels, writing quests, etc. etc. And guess what – they need their own tools too ☹ .

Actually, these toolchains are so important, that I would say that at least half of the value that game engine provides to your project, comes from these toolchains. If you're going to write your own engine – you need to think about these toolchains, as they can easily make-or-break your game (and if you're using 3rd-party game engine – make sure that the toolchain they're providing, is understandable for both your artists and your developers – and for game designers too, if applicable).

**"Actually, these toolchains**

## On Using Game Engines as Pure Graphics Engines, and Vendor Lock-In

These days, if you want to use a 3rd-party graphics engine, most of the time you won't find "graphics engine", but will need to choose between "game engines". And "game engines" tend to provide much more functionality than "graphics engines", which has many positives, but there is also one negative too. Additional features provided by "game engines" in addition to pure graphic rendering capabilities, may include such things as processing user input, support for humanoid-like creatures (which may include, for example, inverse kinematics), asset management, scripting, network support, toolchains, etc. etc. etc. And most of these features even work.

**are so important, that I would say that at least half of the value that game engine provides to your project, comes from these toolchains.**

However, there is a dark spot in this overall bright picture. Exactly the same thing which tends to help a lot, backfires. The thing is that the more features the engine has – the more you will want to use ("hey, we can have this nice feature for free!"). And the more features you use – the more you're tied to a specific 3rd-party game engine, and this process will very soon make it your Absolute Dependency (as defined in Chapter IV), also known as a Vendor Lock-In.

It is not that Absolute Dependencies are bad per se (and, as mentioned in Chapter IV, for quite a few games the advantages of having it outweigh the negatives), but if you have an Absolute Dependency – it is Really Important to realize that you *are* Locked In, and that you SHOULD NOT rely on throwing it away in the future.

Just one example where this can be important. Let's consider you writing a game with an Undefined Life Span (i.e. you're planning your game to run for a really long while, see Chapter I for further details); then you've decided (to speed things up) to make a first release of your game using a 3rd-party game engine. Your game engine of choice is very good, but has one drawback – it doesn't support one of the platforms which you do want to support (for example, it doesn't support mobile, which you want to have ASAP after the very first release). So you're thinking that "hey, we'll release our game using this engine, and then we'll migrate our game from it (or will support another graphics engine for those platforms where it doesn't run, etc.)".

In theory, it all sounds very good. In practice, however, unless you're extremely vigilant (see on it a bit below), and not taking special measures to deal with dependencies, you'll find yourself in a hot water. By the time when you want to migrate away, your code and game in general will be that much intertwined and interlocked with the game engine, that separating them will amount to a full rewrite (which is rarely

possible within the same game without affecting too many subtle gameplay-affecting issues which make or break your game). It means that in our hypothetical example, you won't be able to support mobile devices, *ever* (well, unless you scrap the whole thing and rewrite it from scratch, which will almost inevitably require a re-release at least on a different set of servers, if not under a different title). This situation tends to be even worse for 3D game engines (to the point that I'm not sure that it is possible at all to avoid your 3D game engine Locking you In).

> **"Unless you're extremely vigilant, and not taking special measures to deal with dependencies, you'll find yourself in a hot water.**

The only way to avoid this kind of (very unpleasant) scenarios, is to be extremely vigilant and prohibit the use of all the game features, unless their use is explicitly allowed (and before allowing the use of a certain feature, you need to understand – and document! – how you're going to implement this feature when you are migrating away from the engine). For further details on the measures which you need to take to ensure that your component (such as graphics engine) doesn't become your Absolute Dependency – see Chapter IV.

Once again – having an Absolute Dependency is not necessarily evil, but if you have one – you'd better realize that you're pretty much at the mercy of the engine developer (the one who has successfully locked you in).

## Games with Rudimentary Graphics

Now, let's start considering different types of graphics which you may need for your game. First of all, let's see what happens if your game requires only a minimal graphics (or none at all).

Contrary to the popular belief, you *can* build a game without any graphics at all, or with a very rudimentary one. When speaking about rudimentary graphics, I mean static graphics, without animation, just pictures with defined areas to click. Such games-with-rudimentrary-graphics are not limited to obvious examples such as stock exchanges, but also include some of social games which are doing it with a great success (with one such example being quite popular Lords&Knights).

> **"Contrary to the popular belief, you *can* build a game without any graphics at all, or with a very rudimentary**

If your graphics is non-existent or rudimentary, you can (and probably should) write your graphics engine all by yourself. It won't take long, and having a dependency on a 3rd-party engine merely to render static images is usually not worth the trouble.

Artist's toolchain is almost non-existent too; all artists need to work with rudimentary graphics, is their favourite graphics editor to provide you with bitmaps of sizes-which-you-need.

one.

## Games with 2D Graphics

The next step on the ladder from non-existent graphics to the holy grail of realistic ray-traced 3D[1] is 2D graphics. 2D graphics is still very popular, especially for games oriented towards mobile phones, and for social games (which tend to have mobile phone version, so there is a strong correlation between the two). This section also covers 2D engines used by games with pre-rendered 3D graphics.

In general, if you're making a 2D game, your development, while more complicated than for games with rudimentary graphics, will be still much much simpler than that of 3D game [2]. First of all, 2D graphics (unlike 3D graphics) is rather simple, and you can easily write a simple 2D engine yourself (I've seen a 2D engine with double-buffering and virtually zero flickering written from scratch within about 8-10 man-weeks for a single target platform; not too much if you ask me).

Alternatively, you can use one of the many available "2D game engines"; however, you need to keep in mind the risk of becoming Locked-In (see section "On Using Game Engines as Pure Graphics Engines, and Vendor Lock-In" above). In particular, if you're planning to replace your 2D game engine in the future, you should stay away from using such things as "2D Physics" features provided by your game engine, and limit it to rendering only. In practice, it *is* possible to avoid Vendor Lock-In (and keep your options to migrate from this 2D engine, or to add another 2D or even 3D one alongside it, etc.); however, it still requires you to be extremely vigilant (see section "On Using Game Engines as Pure Graphics Engines, and Vendor Lock-In" above), but at least it has been done and is usually doable.

**MVC**
Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three

One good example of 2D game engine (which is mostly a 2D graphics engine), is [Cocos2D-X]. It is a popular enough cross-platform (including iOS, Android, and WinPhone, and going mobile is One Really Popular Reason for creating a 2D game these days), and has API which is good enough for practical use. If you're developing *only* for iOS, [SpriteKit] is a good choice too. BTW, if you're vigilant enough in avoiding dependencies, you can try making your game with Cocos2D-X, and then to support SpriteKit for iOS only (doing it the other way around is also possible, but usually more risky unless you're absolutely sure that most of your users are coming from iOS). NB: if you're serious about such development, make sure to make Logic-to-Graphics layer as described in "Logic-to-Graphics Layer" section below.

interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user
— *Wikipedia* —

About using 2D functionality of the primarily 3D engines such as Unity or Unreal Engine: personally, I would stay away from them when it comes to 2D development (for my taste, they are way too locking-in for such a relatively simple task as 2D). Such engines would have a Big Advantage for quite a few genres if they could support both 2D and 3D graphics for the same world (kind of MVC for games, also similar to Logic-to-Graphics layer as described below), but to the best of my knowledge, none of the major game engines provides such support. [[NOTE to BETA TESTERS: If you know about such capabilities in these or other engines, please let me know]].

About toolchains. For 2D, artist's toolchains are usually fairly simple, with artists using their favourite animation editor (for example, "Adobe After Effects", but there are other options out there; "Adobe Flash" has also been reported to support "sprite sheets" starting from CS6 version). As a result of their work, they will provide you with sprites (for example, in a form of series of .pngs-with-transparency, or "sprite sheets").

---

[1] Yes, I do know that nobody does raytracing for games (yet), but who said that we cannot daydream a bit?
[2] Hey, isn't it a good reason to scrap all 3D completely in the name of time to market? Well, probably not

## On pre-rendered 3D

Now, let's discuss see what happens if your game is supposed to be a 3D game. In this case, first of all, you need to think whether you really need 3D, or you will be fine with so-called pre-rendered 3D.

When speaking about pre-rendered 3D, the idea is to create your 3D models, and 3D animations, but then, instead of rendering them (such as "render using OpenGL") in real-time, to pre-render these 3D models and animations into 2D "sprites", to ship these 2D sprites with your game instead of shipping full 3D models, and then to render them as 2D sprites in your 2D graphics engine.

Fully 3D pre-rendered games[3] allow you to avoid having 3D engine on clients, replacing it with much simpler (and much more portable) 2D engine.

Usually, full 3D pre-rendering won't work good for first-person games (such as MMORPG/MMOFPS) but it may work

pretty good even for (some kinds of) MMORTS, and for many other kinds of popular MMO genres too. Full 3D pre-rendering is quite popular for platforms with limited resources, such as in-browser games, or games oriented towards mobile phones.

> **"Fully 3D pre-rendered games allow you to avoid having 3D engine on clients, replacing it with much simpler (and much more portable) 2D engine.**

Technically, fully pre-rendered 3D development flow consists of:

- 3D design, usually made using readily available 3rd-party 3D toolchain. For this purpose, you can use such tools as Maya, 3D Max, Poser, or – for really adventurous ones – Blender. 3D design is not normally done by developers, but by designers. It includes both models (including textures etc.) and animations.
- pre-rendering of 3D design into 2D sprites. Usually implemented as a bunch of scripts which "compile" your 3D models and animations into 2D sprites, including animated sprite sequences; the same 3D tools are usually used for this 3D-to-2D rendering
- rendering of 2D sprites on the client, using 2D engine(s)

As an additional bonus, with 3D pre-rendering you don't need to bother with getting low-poly 3D models for your 3D toolchain, and can keep your 3D models in as high number of polygons as you wish. Granted, mostly these high-poly models won't usually make any visual difference (as each of 2D sprites is commonly too small to notice the difference, though YMMV), but at least you won't need to bother with polygon number reduction (and you can be sure that your artists will appreciate it, as low-poly-but-still-decent-looking 3D models are known to be a Big Headache).

3D pre-rendering is certainly not without disadvantages. Two biggest problems of 3D pre-rendering which come to mind, are the following. First of all, you can pre-render your models only at specific angles; it means that if you're showing a battlefield in isometric projection, pre-rendering can be fine, but doing it for a MMOFPS (or any other game with a first-person view) is usually not feasible. Second, if you're not careful enough, the size of your 2D sprites can easily become huge. There are other less prominent issues related to 3D pre-rendering, which we'll discuss in Chapter [[TODO]], but for our purposes now these two things should be enough (i.e. if you're fine with them – you can keep considering 3D pre-rendering).

> "If you can survive 3D pre-rendering without making your game unviewable (and without making it too huge in size), you can make your game run on the platforms which have no 3D at all (or their 3D is hopelessly slow)

On the positive side, if you can survive 3D pre-rendering without making your game unviewable (and without making it too huge in size), you can make your game run on the platforms which have no 3D at all (or their 3D is hopelessly slow); I'm mostly speaking about smartphones here (while smartphones have made huge improvements in 3D performance, they are still light years away from PCs – and it will probably stay this way for a long while).

Artist's toolchains are usually not a problem for pre-rendered 3D. In this case, artists are pretty much free what to use (though it is still advisable to use one tool across the whole project) ; it can be anything ranging from Maya to Poser, with 3D Max in between. They can keep all their work within this tool, and to provide you with ways to produce 2D sprites. In most cases, your job in this regard is about making artists backup their work on regular basis, and about writing the scripts for automated "build" of their source files (those in 3D Max or whatever-else-they're-using) into 2D.

**Bottom Line.** Whether you want/can switch your game to 3D pre-rendering – depends, but at least you should consider this option (that is, unless your game is an MMOFPS/MMORPG). While this technique is often frowned upon (usually, using non-arguments such as "it is not cool"), it might (or might not) work for you.

Just imagine – no need to make those low-poly models, no need to worry that your models become too "fat" for one of your resource-stricken target platforms as soon as you throw in 100 characters within one single area, no need to bother with texture sizes, and so on. It does sound "too good to be true" (and in most cases it will be), but if you're lucky enough to be able to exploit pre-rendering – you shouldn't miss the opportunity.

If you manage to get away with pre-rendered 3D, make sure to read section on 2D graphics above.

---

[3] in fact, partial 3D pre-rendering is also perfectly viable, and is used a lot in 3D games which do have 3D engine on the client-side, but this is beyond the scope of our discussion until Chapter [[TODO]]

## Games with 3D Graphics

If you have found that your 3D game is not a good match for pre-rendered 3D, you do need to have 3D engine on the client-side. This tends to unleash a whole lot of problems, from weird exchange formats between toolchain and your engine, to the inverse kinematics (if applicable); we'll discuss some of these problems in Chapter [[TODO]], for now let's just write down that non-pre-rendered 3D is a Big Pain in the Neck (compared to the other types of graphics). If you do need a 3D engine on client side, you basically have two distinct options.

Option 1 is along "DIY" lines, with you writing your own rendering engine over either OpenGL, or over DirectX. In this case, be prepared to spend a lot of time on making your game look anywhere reasonable. Making 3D work is not easy to start with, but making it look good is a major challenge. In addition, you will need to remember about the artist's toolchain; at the very least you'll need to provide a way to import and use files generated by popular 3D design programs (hint: supporting wavefront .obj is not enough, you'll generally need to dig much deeper into specifics of 3D-program-you're-supporting and its formats).

"Making 3D work is not easy to start with, but making it look good is a major challenge.

On the plus side, if you manage to survive this ordeal and get a reasonably looking graphics with your own 3D engine, you'll get a solid baseline which will give you a lot of flexibility (and you may need this flexibility, especially if we're speaking about the games with Undefined Life Span).

Option 2 is to try using some "3D game engine" as your "3D engine". This way, unless you already decided that your game engine is your Absolute Dependency, is a risky one. 3D game engines tend to be so complicated, and have so many points of interaction with the game, that chances are that even if you're Extremely Vigilant when it comes to dependencies, you won't be able to replace the engine later. Once again – I am not saying that Vendor Lock-In is necessarily a bad thing, but you do need to realize that you're Locked In.

## Logic-to-Graphics Layer

Unless you've already decided that you want to be 100% Locked In, it is usually a good idea to have a separation layer between your logic and your graphics engine (whether it is 2D engine or 3D engine). Let's name this separation layer a Logic-to-Graphics Layer; this layer resides completely on the client side, and doesn't really affect your communication protocols or the server side. In a sense, it can be seen as a subset of a Model-View-Controller pattern (with game logic representing Model, and graphics engine representing View).

Let me explain the idea on one simple example. If your game is a blackjack, client-

side game logic needs to produce rendering instructions to your graphics engine. Usually, naive implementations will just have client-side game logic to issue instructions such as “draw such-and-such bitmap at such-and-such coordinates”. This approach works well, until you need to port your client to another device (in the extreme case – from PC to phone, with the latter having much less screen real estate).

With Logic-to-Graphics layer, your client-side blackjack game logic issues instructions in terms of “place 9S in front of player #3 at the table” (and *not* in terms of “draw 9S at the (234,567) point on screen”). Then, it becomes a job of Logic-to-Graphics Layer to translate this instruction into screen coordinates. And if your game is a strategy, client game logic should issue instructions in terms of “move unit A to position (X,Y)” (with the coordinates expressed in terms of simulated-world coordinates, not in terms of on-screen coordinates(!)), and again the translation between the two should be performed by our Logic-to-Graphics layer.

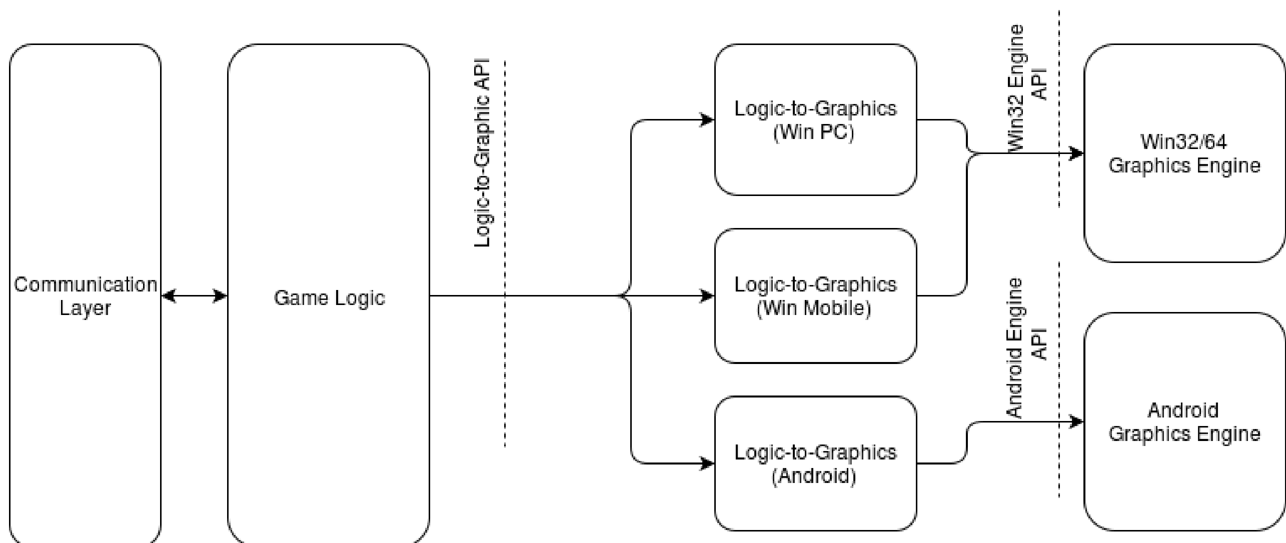
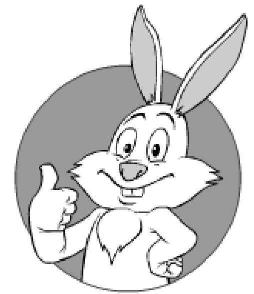


Fig V.1

One example incarnation of a system built using Logic-to-Graphics approach, is shown on Fig 1. Here, “Game Logic” doesn’t depend on a graphical engine (or a platform) and can be developed separately (which is very important because it will change very frequently). In contrast, two “Graphical Engines” are specific to the respective platforms, but they don’t know/depend on game logic at all, and are very-rarely changed. The “Logic-to-Graphics” layer is a “glue” layer which belongs in between “game logic” and “graphical engine”; by design, it depends both on game logic and graphical engine (ouch); however (provided that there is a reasonably clean separation achieved, see examples above) it doesn’t change nearly as often as “game logic” itself, so the whole thing becomes manageable. On Fig. 1, there are three implementations of “Logic-to-Graphics” layer: one is for Android and two for Windows; the reason for having two different implementations of Logic-to-Graphics layer for the same Win32 graphics engine, is that PC and mobile versions are usually quite different in terms of layout, and therefore it may be

simpler just to have two different implementations of Logic-to-Graphics layer (which is responsible, among other things, for translation of coordinates into screen coordinates).

If doing it this way, you'll get quite a few benefits.



> "First of all, you will have a very clear separation between the different layers of the program, which tends to help a whole lot in the long run.

- First of all, you will have a very clear separation between the different layers of the program, which tends to help a whole lot in the long run.
- Second, even if you're supporting only one platform now, you're leaving the door open to adding support for all the platforms you might want, in the future. This includes such things as adding an option to have a 3D version to your currently-2D-only game.
- Third, you don't have a strong dependency on any graphical engine, so if in 5 years from now a new, much-better engine will arise, you'll be able to migrate there without rewriting the whole thing.
- Fourth, such a clean separation facilitates using authoritative servers (which we'll discuss in Chapter [[TODO]], and which are extremely important for the reasons described there).
- Fifth, with Logic-to-Graphics layer, for quite a few genres you'll be able to produce a command-line client, which comes handy for testing (including automated testing, and testing of game logic without being affected by graphics), and also for development-of-the-new-features while the graphics is not ready yet.

We've discussed the benefits of this Logic-to-Graphics layer, but what about the costs? Is it all 100% positive, or there are some drawbacks? In fact, I can only think of two realistic negatives for having it:

- There is a certain development overhead which is necessary to achieve this clean separation. I'm not talking about performance overhead, but about development overhead. If the game logic developer needs to get something from the graphics engine, he cannot just go ahead and call the graphics-engine-function-which-he-wants. Instead, an interface to get whatever-he-needs should be created, has to be supported by all the engines, etc. etc. It's all easily doable, but it introduces quite a bit of mundane work. On the other hand, I contend that in the long run, such clean interfaces provide much more value than this development overhead takes away; in particular, clean interfaces have been observed as a strong obstacle to the code becoming "spaghetti code", which is already more-than-enough enough to justify them.

- A learning curve for those game developers coming from traditional limited-life-span (and/or not-massively-multiplayer) 3D games. In these classical games (I intentionally don't want to use the term "old-fashioned" to avoid being too blunt about it 😉 ) everything revolves around the 3D engine, so for these developers moving towards the model with clean separation between graphics and logic will be rather painful. However, unless you decided your game to be Engine-Centric, you need to move away from this approach anyway, and even for those guys-coming-from-classical-3D-games this clean separation model will be quite beneficial in the long run, so I wouldn't say that this drawback is that important.

Personally, for games with a potentially unlimited life span (and not having 3rd-party game engine as an Absolute Dependency a.k.a. Vendor Lock-In), I almost universally recommend to implement this Logic-to-Graphics Layer.

## Dual Graphics, including 2D+3D Graphics

In quite a few cases, you may need to support two substantially different types of graphics. One such example is when you need to support your game both for PC and phone; quite often the difference between available screen real estate is too large to keep your layout the same, so you usually need to redesign not only the graphics as such, but also redesign layout.

**"In such cases of dual graphics, it is paramount to have Logic-to-Graphics layer as described above.**

In such cases of dual graphics, it is paramount to have Logic-to-Graphics layer as described above. As soon as you have Logic-to-Graphics layer, adding new type of graphics is a breeze. You just need to add another implementation of Logic-to-Graphics layer (using either the same graphical engine, or different one, depending on your needs), and there is no need to change game logic (!). These two different implementations of Logic-to-Graphics layer may have different APIs on the boundary with graphics engines, but they always have the same API on the boundary with Game Logic. The latter fact will allow you to keep developing your game logic without caring about the specific engines you're using.

The reason why it is so important to have Logic-to-Graphics Layer is simple – for such a frequently changed piece of code as a client-side game logic, maintaining two separate code bases is usually not realistic. Pretty much any feature you're adding, will require some changes in game logic on the client side (hey, at least you need to receive that new server message you've just introduced and parse it!), and having two code bases for game logic will mean that you need to duplicate all such changes all the time. I've observed much more than one competitor going the route of multiple code bases, only to see that one of these code bases starts to lag behind the other,

and scrapping it 6 months down the road. It just illustrates the main point: you do need to keep your frequently-changed portions of the code as a single code base. And Logic-to-Graphics Layer allows to achieve it.

Of course, if you need to add a new instruction which comes from game logic to Logic-to-Graphics Layer (for example, if you're adding a new graphical primitive), you will still need to modify both your implementations of the Logic-to-Graphics Layer. However, if your separation API is clean enough, you will find that such changes, while still happening and causing their fair share of trouble, are the orders of magnitude more rare than the changes to game logic; this difference in change frequencies is the difference between workable and unworkable one.

An extreme case of dual graphics is dual 2D+3D graphics. Not all the game genres allow it (for example, first-person shooters usually won't work too good in 2D), but if your game genre is ok with it, *and* you have Logic-to-Graphics separation layer, this becomes perfectly feasible. You just need to have 2 different engines, a 3D one and a 2D one (they can be in separate clients, or even switchable in run-time), and an implementation of Logic-to-Graphisc layer for each of them. As soon as you have this, Bingo! – you've provided your players with a choice between 2D and 3D graphics (depending on their preference, or platform, or whatever else). Even better, when using a Logic-to-Graphics layer, you can start with the graphics which is simpler/more important/whatever, and to add another graphics (or even multiple ones) later.

## [[To Be Continued…

This concludes beta Chapter V(a) from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Stay tuned for beta Chapter V(b), "Modular Architecture: Client-Side. Programming Languages"

**EDIT: Chapter V(b). Modular Architecture: Client-Side. Programming Languages for Games, including Resilience to Reverse Engineering and Portability, has been published**

]]

## [–] References

[Cocos2D-X] http://www.cocos2d-x.org/
[SpriteKit] https://developer.apple.com/spritekit/

## Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Due to Popular Demand: PDFs of Beta Chapters from "Develop..*

*Chapter V(b). Modular Architecture: Client-Side. Programmin..* »

*Filed Under: Distributed Systems, Programming, System Architecture*
*Tagged With: 2D, 3D, client, game, graphics, multi-player*

*Copyright © 2014-2015 ITHare.com*